

MQL5

Stanislav Korotky

```

bool PlaceOrder(const ENUM_ORDER_TYPE type, const double range)
{
    bool success = false;
    MqlTradeRequestSync request;

    // origin is filled only for *STOP LIMIT orders
    if (type == ORDER_TYPE_BUY_STOP || type == ORDER_TYPE_SELL_STOP || type == ORDER_TYPE_BUY_LIMIT || type == ORDER_TYPE_SELL_LIMIT)
        request.origin = ORIGIN_FROM_PRICE;
    else
        request.origin = ORIGIN_DEFAULT;

    void OnTick()
    {
        //---placing a pending order
        if (!pending_done)
        {
            double ask=SymbolInfoDouble(_Symbol,SYMBOL_ASK);
            double buy_stop_price=NormalizeDouble(ask+100*_Point,(int)SymbolInfoInteger(_Symbol,SYMBOL_POINTS_PER_PIP));
            bool res=trade.BuyStop(buy_stop_price,_Symbol);
            //--- if BuyStop() function performs successfully
            if(res)
            {
                pending_done=true;
            }
        }
    }
}

```

目录

面向交易员的 MQL5 编程指南 13

第一章 MQL5 和开发环境简介	14
1.1 编辑、编译和运行程序	15
1.2 MQL 向导和程序草稿	18
1.3 语句、代码块和函数	21
1.4 第一个程序	23
1.5 数据类型和值	25
1.6 变量和标识符	26
1.7 赋值和初始化，表达式和数组	26
1.8 数据输入	28
1.9 错误修复与调试	30
1.10 数据输出	34
1.11 格式、缩进和空格	35
1.12 小结	37
第二章 编程基础知识	38
2.1 标识符	39
2.2 内置数据类型	40
2.2.1 整数	42
2.2.2 浮点数	44
2.2.3 字符类型	47
2.2.4 字符串类型	48
2.2.5 逻辑（布尔）类型	49
2.2.6 日期和时间	50
2.2.7 颜色	51
2.2.8 枚举	51
2.2.9 自定义枚举	54
2.2.10 Void 类型	56
2.3 变量	56
2.3.1 变量的声明和定义	57
2.3.2 变量的上下文、作用域和生存期	57
2.3.3 初始化	60
2.3.4 静态变量	65
2.3.5 常量变量	66
2.3.6 输入变量	67
2.3.7 外部变量	69
2.4 数组	72
2.4.1 数组特性	72
2.4.2 数组说明	73
2.4.3 使用数组	75
2.5 表达式	77
2.5.1 基本概念	78
2.5.2 赋值运算	79
2.5.3 算术运算	81
2.5.4 递增和递减	83
2.5.5 比较运算	84
2.5.6 逻辑运算	85
2.5.7 按位运算	87
2.5.8 修改运算	88
2.5.9 条件三元运算符	90
2.5.10 逗号	91
2.5.11 特殊运算符 sizeof 和 typename	92

2.5.12 用括号分组	93
2.5.13 运算优先顺序	94
2.6 类型转换	96
2.6.1 隐式类型转换	96
2.6.2 算术类型转换	97
2.6.3 显式类型转换	99
2.7 语句	100
2.7.1 复合语句（代码块）	100
2.7.2 语句/定义语句	101
2.7.3 简单语句（表达式）	104
2.7.4 控制语句概述	105
2.7.5 for 循环结构	106
2.7.6 while 循环结构	109
2.7.7 do 循环结构	110
2.7.8 if 选择结构	111
2.7.9 switch 选择结构	114
2.7.10 break 跳转结构	117
2.7.11 continue 跳转结构	119
2.7.12 return 跳转结构	120
2.7.13 空语句	121
2.8 函数	122
2.8.1 函数定义	122
2.8.2 函数调用	123
2.8.3 参数和自变量	124
2.8.4 值参数和引用参数	125
2.8.5 可选参数	127
2.8.6 返回值	128
2.8.7 函数声明	129
2.8.8 递归	130
2.8.9 函数重载	131
2.8.10 函数指针（typedef）	133
2.8.11 内联	135
2.9 预处理器	135
2.9.1 包括源文件（#include）	136
2.9.2 宏替换指令概述	137
2.9.3 简单形式的 #define	138
2.9.4 #define 作为伪函数的形式	140
2.9.5 #define 定义中的特殊运算符 # 和 ##	142
2.9.6 取消宏替换（#undef）	143
2.9.7 预定义的预处理器常量	143
2.9.8 条件编译（#ifdef/#ifndef/#else/#endif）	144
2.9.9 常规程序属性（#property）	145
第三章面向对象的编程	147
3.1 结构体和联合体	147
3.1.1 结构体定义	148
3.1.2 结构体中的函数（方法）	150
3.1.3 拷贝结构体	150
3.1.4 构造函数和析构函数	151
3.1.5 在内存中打包结构体以及与 DLL 交互	154
3.1.6 结构体布局与继承	154
3.1.7 访问权限	157
3.1.8 联合体	158
3.2 类和接口	159
3.2.1 OOP 基础知识：抽象	160
3.2.2 OOP 基础知识：封装	160
3.2.3 OOP 基础知识：继承	161

3.2.4 OOP 基础知识: 多态性	161
3.2.5 OOP 基础知识: 组合 (设计)	162
3.2.6 类定义	163
3.2.7 访问权限	165
3.2.8 构造函数: 默认、参数化和拷贝	167
3.2.9 析构函数	171
3.2.10 自引用: this	172
3.2.11 继承	175
3.2.12 动态创建对象: new 和 delete	180
3.2.13 指针	182
3.2.14 虚方法 (virtual 和 override)	187
3.2.15 静态成员	195
3.2.16 嵌套类型、命名空间和上下文运算符 ::	197
3.2.17 拆分类语句和定义	200
3.2.18 抽象类和接口	203
3.2.19 运算符重载	204
3.2.20 对象类型转换: dynamic_cast 和 void 指针 *	215
3.2.21 指针、引用和常量	219
3.2.22 继承管理: final 和 delete	222
3.3 模板	224
3.3.1 模板头文件	225
3.3.2 模板操作通用原则	225
3.3.3 模板与预处理器宏	227
3.3.4 模板中内置类型和对对象类型的特征	228
3.3.5 函数模板	232
3.3.6 对象类型模板	236
3.3.7 方法模板	240
3.3.8 嵌套模板	246
3.3.9 缺失模板特化	248
第四章常用 API	251
4.1 内置类型转换	251
4.1.1 数字与字符串的相互转换	252
4.1.2 双精度浮点数的规范化	255
4.1.3 日期和时间	256
4.1.4 颜色	265
4.1.5 结构体	267
4.1.6 枚举	269
4.1.7 复合型	271
4.2 使用字符串和符号	272
4.2.1 字符串的初始化和长度测量	273
4.2.2 字符串拼接	276
4.2.3 字符串比较	277
4.2.4 更改字符大小写和修剪空格	283
4.2.5 查找、替换和提取字符串片段	284
4.2.6 使用符号和代码页	288
4.2.7 数据到字符串的通用格式化输出	294
4.3 使用数组	300
4.3.1 数组的日志记录	300
4.3.2 动态数组	304
4.3.3 数组长度测量	311
4.3.4 初始化和填充数组	312
4.3.5 拷贝和编辑数组	314
4.3.6 移动 (交换) 数组	324
4.3.7 数组的比较、排序和搜索	326
4.3.8 数组中时间序列的索引方向	340
4.3.9 将对象和数组清零	344

4.4 数学函数	349
4.4.1 数字的绝对值	350
4.4.2 比较两个数字的大小	351
4.4.3 舍入函数	352
4.4.4 除法运算后的余数（取模运算）	353
4.4.5 幂运算和根运算	353
4.4.6 指数函数和对数函数	354
4.4.7 三角函数	355
4.4.8 双曲函数	358
4.4.9 实数的正态性检验	359
4.4.10 随机数生成	361
4.4.11 整数中的字节顺序控制	363
4.5 使用文件	364
4.5.1 信息存储方法：文本和二进制	366
4.5.2 简化模式下的文件读写	368
4.5.3 打开和关闭文件	372
4.5.4 管理文件描述符	377
4.5.5 选择文本模式的编码	384
4.5.6 读写数组	386
4.5.7 读写结构（二进制文件）	391
4.5.8 读写变量（二进制文件）	395
4.5.9 读写变量（文本文件）	403
4.5.10 文件中的位置管理	411
4.5.11 获取文件属性	417
4.5.12 强制将高速缓存写入磁盘	421
4.5.13 删除文件并检查文件是否存在	426
4.5.14 拷贝和移动文件	427
4.5.15 搜索文件和文件夹	429
4.5.16 使用文件夹	432
4.5.17 文件或文件夹选择对话框	433
4.6 客户端终端全局变量	436
4.6.1 读写全局变量	438
4.6.2 检查是否存在和上次活动时间	439
4.6.3 获取全局变量列表	440
4.6.4 删除全局变量	441
4.6.5 临时全局变量	442
4.6.6 使用全局变量同步程序	443
4.6.7 将全局变量刷新到磁盘	452
4.7 时间处理函数	452
4.7.1 本地时间和服务器时间	454
4.7.2 夏令时（本地）	457
4.7.3 世界时	462
4.7.4 暂停程序	463
4.7.5 时间间隔计数器	463
4.8 用户交互	465
4.8.1 记录消息	465
4.8.2 警报	468
4.8.3 在图表窗口中显示消息	469
4.8.4 消息对话框	473
4.8.5 声音警报	477
4.9 MQL 程序执行环境	478
4.9.1 获取终端和程序属性的总列表	479
4.9.2 终端内部版本号	483
4.9.3 程序类型和许可证	483
4.9.4 终端和程序操作模式	484
4.9.5 权限	487

4.9.6 检查网络连接	490
4.9.7 计算资源：内存、磁盘和 CPU	491
4.9.8 屏幕规格	493
4.9.9 终端和程序字符串属性	495
4.9.10 自定义属性：柱线数限制和界面语言	496
4.9.11 将程序绑定到运行时属性	496
4.9.12 检查键盘状态	505
4.9.13 检查 MQL 程序状态和终止原因	507
4.9.14 以编程方式关闭终端并设置返回码	509
4.9.15 处理运行时错误	511
4.9.16 用户定义的错误	512
4.9.17 调试管理	518
4.9.18 预定义变量	518
4.9.19 MQL5 语言的预定义常量	519
4.10 矩阵和向量	520
4.10.1 矩阵和向量的类型	520
4.10.2 创建和初始化矩阵和向量	522
4.10.3 拷贝矩阵、向量和数组	525
4.10.4 将时间系列拷贝到矩阵和向量	527
4.10.5 将分时报价历史拷贝到矩阵和向量	528
4.10.6 含矩阵和向量的表达式的求值	529
4.10.7 操作矩阵和向量	530
4.10.8 矩阵和向量的乘积	534
4.10.9 矩阵的变换（分解）	536
4.10.10 获取统计数据	538
4.10.11 矩阵和向量的特征	539
4.10.12 解方程	541
4.10.13 机器学习方法	547
第五章创建应用程序	557
5.1 执行 MQL 程序的一般原则	557
5.1.1 设计各种类型的 MQL 程序	559
5.1.2 线程	561
5.1.3 事件处理函数概述	562
5.1.4 各类启动和停止程序的特征	567
5.1.5 指标和 EA 交易的参考事件：OnInit 和 OnDeinit	570
5.1.6 脚本和服务的主要功能：OnStart	572
5.1.7 以编程方式移除 EA 交易和脚本：ExpertRemove	572
5.2 脚本和服务	574
5.2.1 脚本	574
5.2.2 服务	575
5.2.3 脚本和服务的局限性	579
5.3 时间序列	580
5.3.1 交易品种和时间范围	581
5.3.2 时间序列组织和存储的技术方面	584
5.3.3 获取价格数组的特征	585
5.3.4 可用柱线数量 (Bars/iBars)	586
5.3.5 按时间搜索柱线指数 (iBarShift)	587
5.3.6 用于获取报价数组的 copy 函数概述	589
5.3.7 作为 MqlRates 结构体的数组获取报价	593
5.3.8 单独请求价格、交易量、点差、时间的数组	595
5.3.9 通过柱线指数读取价格、成交量、点差和时间	597
5.3.10 查找时间序列中的最大值和最小值	600
5.3.11 在 MqlTick 结构体中使用实时分时报价数组	604
5.4 创建自定义指标	614
5.4.1 指标主要特征	614
5.4.2 主要指标事件：OnCalculate	615

5.4.3 两种类型的指标：主窗口和子窗口	618
5.4.4 设置缓冲区和绘图的数量	619
5.4.5 将数组分配为缓冲区：SetIndexBuffer	620
5.4.6 绘图设置：PlotIndexSetInteger	622
5.4.7 缓冲区和图表映射规则	628
5.4.8 对自定义绘图应用指令	631
5.4.9 设置绘图名称	633
5.4.10 可视化数据缺口（空元素）	634
5.4.11 独立子窗口中的指标：大小和级别	639
5.4.12 指标的一般属性：标题和值精度	646
5.4.13 图表逐项着色	646
5.4.14 跳过初始柱线的绘制	649
5.4.15 等待数据和管理可见性（DRAW_NONE）	655
5.4.16 多货币和多时段指标	665
5.4.17 跟踪柱线形成	686
5.4.18 检测指标	689
5.4.19 指标的优缺点	690
5.4.20 在 MQL 向导中创建指标草稿	692
5.5 使用 MQL 程序中的现成指标	694
5.5.1 指标所有者的句柄和计数器	695
5.5.2 创建指标实例的简单方法：iCustom	696
5.5.3 检查计算得出柱线的数量：BarsCalculated	699
5.5.4 从指标获取时间序列数据：CopyBuffer	701
5.5.5 支持多个交易品种和时间范围	710
5.5.6 内置指标概述	716
5.5.7 使用内置指标	721
5.5.8 指标创建的高级方式：IndicatorCreate	729
5.5.9 使用 IndicatorCreate 灵活创建指标	739
5.5.10 图表中指标管理功能概述	748
5.5.11 将输出组合到主窗口和辅助窗口	749
5.5.12 从存在偏移的图表中读取数据	752
5.5.13 删除指标实例：IndicatorRelease	755
5.5.14 通过句柄获取指标设置	760
5.5.15 定义指标的数据源	763
5.6 使用计时器	764
5.6.1 打开和关闭计时器	765
5.6.2 计时器事件：OnTimer	765
5.6.3 高精度计时器：EventSetMillisecondTimer	775
5.7 使用图表	777
5.7.1 用于获取当前图表基本属性的函数	777
5.7.2 图表标识	778
5.7.3 获取图表列表	779
5.7.4 获取任意图表的交易品种和时间范围	780
5.7.5 用于处理完整属性集的函数概述	781
5.7.6 图表的描述性属性	783
5.7.7 检查主窗口的状态	785
5.7.8 获取窗口/子窗口的数量和可见性	785
5.7.9 图表显示模式	787
5.7.10 管理图表元素的可见性	794
5.7.11 水平偏移	797
5.7.12 水平缩放（按时间）	799
5.7.13 垂直缩放（按价格和指标读数）	800
5.7.14 颜色	803
5.7.15 鼠标和键盘控制	806
5.7.16 分离图表窗口	809
5.7.17 获取图表上的 MQL 程序放置坐标	810

5.7.18	屏幕坐标与时间/价格的相互转换	812
5.7.19	沿时间轴滚动图表	814
5.7.20	图表重绘请求	817
5.7.21	切换交易品种和时间范围	817
5.7.22	管理图表上的指标	818
5.7.23	打开和关闭图表	823
5.7.24	使用 tp1 图表模板	826
5.7.25	保存图表图像	840
5.8	图形对象	842
5.8.1	对象类型以及用于指定对象坐标的功能	842
5.8.2	时间和价格约束对象	843
5.8.3	绑定到屏幕坐标的对象	845
5.8.4	创建对象	846
5.8.5	删除对象	848
5.8.6	查找对象	851
5.8.7	对象属性访问函数概述	853
5.8.8	主要对象属性	871
5.8.9	价格和时间坐标	872
5.8.10	锚定窗口角点和屏幕坐标	875
5.8.11	在对象上定义锚点	878
5.8.12	管理对象状态	881
5.8.13	对象的优先级 (Z 顺序)	881
5.8.14	对象显示设置: 颜色、样式、边框	884
5.8.15	字体设置	897
5.8.16	以任意角度旋转文本	900
5.8.17	确定对象宽度和高度	902
5.8.18	对象在时间范围内的可见性	909
5.8.19	为标签分配字符代码	912
5.8.20	含直线对象的射线特性	913
5.8.21	管理对象的按下状态	916
5.8.22	调整位图对象中的图像	918
5.8.23	图像的裁剪 (输出部分)	919
5.8.24	输入字段特性: 对齐和只读	922
5.8.25	标准偏差通道宽度	924
5.8.26	在级别对象中设置级别	924
5.8.27	Gann、Fibonacci 和 Elliot 对象的附加特性	928
5.8.28	图表对象	929
5.8.29	移动对象	933
5.8.30	在指定线条点获取时间或价格	934
5.9	图表上的交互事件	937
5.9.1	事件处理函数 OnChartEvent	938
5.9.2	事件相关图表特性	940
5.9.3	图表变化事件	942
5.9.4	键盘事件	943
5.9.5	鼠标事件	951
5.9.6	图形对象事件	954
5.9.7	生成自定义事件	958
第六章	交易自动化	963
6.1	金融工具和市场报价	963
6.1.1	获取可用的交易品种和市场报价列表	964
6.1.2	编辑市场报价列表	965
6.1.3	检查交易品种是否存在	967
6.1.4	检查交易品种数据相关性	968
6.1.5	获取交易品种的最新分时报价	969
6.1.6	交易和报价时段时间表	973
6.1.7	交易品种保证金率	978

6.1.8 用于获取交易品种特性的函数概述	979
6.1.9 检查交易品种状态	988
6.1.10 用于构建交易品种图表的价格类型	989
6.1.11 金融工具的基础货币、报价货币和保证金货币	995
6.1.12 价格表述准确性和变更步骤	1001
6.1.13 允许的交易操作量	1004
6.1.14 交易权限	1006
6.1.15 交易品种交易条件及订单执行模式	1010
6.1.16 保证金要求	1013
6.1.17 挂单到期规则	1021
6.1.18 点差以及订单偏离当前价格的幅度	1025
6.1.19 获取掉期规模	1029
6.1.20 当前市场信息（分时报价）	1034
6.1.21 交易品种的描述性特性	1036
6.1.22 市场深度	1039
6.1.23 交易品种的自定义特性	1041
6.1.24 特定特性（证券交易所、衍生工具、债券）	1042
6.2 市场深度	1043
6.2.1 管理对市场事件深度的订阅	1043
6.2.2 接收市场深度变化事件	1046
6.2.3 阅读最新的市场深度数据	1048
6.2.4 在应用算法中使用市场深度数据	1055
6.3 交易账户信息	1062
6.3.1 用于获取账户特性的函数概述	1063
6.3.2 识别账户、客户端、服务器和经纪人	1065
6.3.3 账户类型：真实、模拟或竞赛	1066
6.3.4 账户货币	1067
6.3.5 账户类型：净额结算或对冲	1067
6.3.6 账户操作的限制和权限	1068
6.3.7 账户保证金设置	1071
6.3.8 账户目前的财务业绩	1074
6.4 创建 EA 交易	1074
6.4.1 EA 交易的主要事件：OnTick	1075
6.4.2 基本原理和概念：订单、交易和仓位	1077
6.4.3 交易操作类型	1078
6.4.4 订单类型	1079
6.4.5 按价格和数量划分的订单执行模式	1081
6.4.6 挂单到期日期	1082
6.4.7 期货订单的保证金计算方法：OrderCalcMargin	1082
6.4.8 估算交易操作的利润：OrderCalcProfit	1094
6.4.9 MqlTradeRequest 结构体	1100
6.4.10 MqlTradeCheckResult 结构体	1102
6.4.11 请求验证：OrderCheck	1104
6.4.12 请求发送结果：MqlTradeResult 结构体	1108
6.4.13 发送交易请求：OrderSend 和 OrderSendAsync	1109
6.4.14 买入和卖出操作	1115
6.4.15 修改仓位的止损和/或止盈水平	1129
6.4.16 跟踪止损	1135
6.4.17 平仓：全部和部分	1145
6.4.18 反向平仓：全部和部分	1154
6.4.19 挂单	1163
6.4.20 修改挂单	1174
6.4.21 删除挂单	1184
6.4.22 获取活动订单列表	1187
6.4.23 订单特性（现行和历史）	1188
6.4.24 用于读取活动订单特性的函数	1192

6.4.25 按特性选择订单	1199
6.4.26 获取仓位列表	1215
6.4.27 仓位特性	1217
6.4.28 用于读取仓位特性的函数	1219
6.4.29 交易特性	1227
6.4.30 从历史中选择订单和交易	1231
6.4.31 用于从历史中读取订单特性的函数	1232
6.4.32 用于从历史中读取交易特性的函数	1235
6.4.33 交易类型	1247
6.4.34 OnTradeTransaction 事件	1250
6.4.35 同步和异步请求	1267
6.4.36 OnTrade 事件	1280
6.4.37 监测交易环境变化	1287
6.4.38 创建多交易品种 EA 交易	1316
6.4.39 EA 交易的优势和局限性	1330
6.4.40 在 MQL 向导中创建 EA 交易	1331
6.5 测试和优化 EA 交易	1334
6.5.1 在测试程序中生成分时报价	1335
6.5.2 测试程序中的时间管理：计时器、睡眠、GMT	1341
6.5.3 测试可视化对象：图表、对象、指标	1342
6.5.4 多币种测试	1343
6.5.5 优化测试条件	1348
6.5.6 获取测试财务统计数据：TesterStatistics	1349
6.5.7 OnTester 事件	1360
6.5.8 自动调优：ParameterGetRange 和 ParameterSetRange	1369
6.5.9 用于优化控制的 OnTester 事件组	1376
6.5.10 从代理向终端发送数据帧	1377
6.5.11 在终端获取数据帧	1378
6.5.12 测试程序的预处理器指令	1384
6.5.13 管理指标可见性：TesterHideIndicators	1389
6.5.14 模拟入金和出金	1390
6.5.15 强制测试停止：TesterStop	1393
6.5.16 EA 交易详细示例	1394
6.5.17 数学计算	1436
6.5.18 调试和概要分析	1437
6.5.19 测试程序的功能局限性	1439
第七章高级语言工具	1440
7.1 资源	1440
7.1.1 使用 #resource 指令描述资源	1440
7.1.2 共享使用不同 MQL 程序的资源	1442
7.1.3 资源变量	1443
7.1.4 作为资源连接自定义指标	1447
7.1.5 动态资源创建：ResourceCreate	1454
7.1.6 删除动态资源：ResourceFree	1458
7.1.7 读取和修改资源数据：ResourceReadImage	1458
7.1.8 将图像保存到文件：ResourceSave	1468
7.1.9 字体和图形资源的文本输出	1479
7.1.10 图形资源在交易中的应用	1492
7.2 自定义交易品种	1500
7.2.1 创建和删除自定义交易品种	1501
7.2.2 自定义交易品种特性	1503
7.2.3 设置保证金率	1505
7.2.4 配置报价和交易时段	1505
7.2.5 添加、替换和删除报价	1506
7.2.6 添加、替换和删除分时报价	1515
7.2.7 转换订单簿变更	1541

7.2.8 自定义交易品种交易细节	1547
7.3 经济日历	1563
7.3.1 日历的基本概念	1563
7.3.2 获取可用国家/地区的列表和说明	1570
7.3.3 按国家/地区和货币查询事件类型	1572
7.3.4 按 ID 获取事件说明	1576
7.3.5 按国家/地区或货币获取事件记录	1576
7.3.6 获取特定类型的事件记录	1580
7.3.7 按 ID 读取事件记录	1583
7.3.8 按国家/地区或货币跟踪事件变化	1587
7.3.9 按类型跟踪事件变化	1596
7.3.10 多条件筛选事件	1596
7.3.11 将日历数据库传输到测试程序	1615
7.3.12 日历交易	1637
7.4 加密	1646
7.4.1 可用信息变换方法概述	1647
7.4.2 加密、哈希和数据打包: CryptEncode	1649
7.4.3 数据解密与解压: CryptDecode	1660
7.5 网络功能	1665
7.5.1 发送推送通知	1666
7.5.2 发送电子邮件通知	1666
7.5.3 将文件发送到 FTP 服务器	1667
7.5.4 通过 HTTP/HTTPS 与网络服务器进行数据交换	1668
7.5.5 建立和断开网络套接字连接	1686
7.5.6 检查套接字状态	1687
7.5.7 设置套接字的数据发送和接收超时值	1689
7.5.8 通过不安全的套接字连接读取和写入数据	1690
7.5.9 准备安全套接字连接	1694
7.5.10 通过安全套接字连接读取和写入数据	1695
7.6 SQLite 数据库	1705
7.6.0 MQL5 中的数据库操作原理	1707
7.6.1 SQL 基础	1711
7.6.2 表结构: 数据类型和限制	1715
7.6.3 OOP (MQL5) 和 SQL 集成: ORM 概念	1718
7.6.4 创建、打开和关闭数据库	1719
7.6.5 执行没有 MQL5 数据绑定的查询	1721
7.6.6 检查表在数据库中是否存在	1730
7.6.7 准备绑定查询: DatabasePrepare	1731
7.6.8 删除和重置预置查询	1733
7.6.9 将数据绑定到查询参数: DatabaseBind/Array	1735
7.6.10 执行预置查询: DatabaseRead/Bind	1736
7.6.11 分别读取字段: DatabaseColumn 函数	1737
7.6.12 通过 ORM 对象在 SQLite 中执行 CRUD 操作的示例	1739
7.6.13 交易	1756
7.6.14 导入和导出数据库表	1760
7.6.15 将表和 SQL 查询打印到日志	1761
7.6.16 使用 SQLite 搜索交易策略的示例	1762
7.7 开发与连接二进制格式库	1773
7.7.1 创建 ex5 库; 导出函数	1774
7.7.2 包含库; 函数导入 (#import)	1777
7.7.3 库文件搜索顺序	1782
7.7.4 DLL 连接细节	1783
7.7.5 MQL5 库中的类和模板	1788
7.7.6 从 .NET 库导入函数	1802
7.8 项目	1802
7.8.1 使用本地项目的一般原则	1804

7.8.2 拷贝交易和信号的 Web 服务项目计划.....	1807
7.8.3 基于 Nodejs 的 web 服务器.....	1807
7.8.4 WebSockets 协议的理论基础.....	1809
7.8.5 基于 WebSocket 协议的 Web 服务的服务器组件.....	1811
7.8.6 MQL5 中的 WebSocket 协议.....	1819
7.8.7 MQL5 中回显和聊天服务的客户端程序.....	1828
7.8.8 交易信号服务和测试 Web 页.....	1837
7.8.9 MQL5 中的信号服务客户端程序.....	1842
7.9 原生 Python 支持.....	1857
7.9.1 安装 Python 和 MetaTrader5 软件包.....	1858
7.9.2 适用于 Python 的 MetaTrader5 包功能概述.....	1860
7.9.3 将 Python 脚本连接到终端和账户.....	1862
7.9.4 错误检查: last_error.....	1863
7.9.5 获取交易账户信息.....	1864
7.9.6 获取终端信息.....	1866
7.9.7 获取金融工具信息.....	1868
7.9.8 订阅订单簿变更.....	1871
7.9.9 读取报价.....	1874
7.9.10 读取分时报价历史.....	1879
7.9.11 计算保证金要求和评估利润.....	1882
7.9.12 检查和发送交易订单.....	1883
7.9.13 获取活动订单的数量和列表.....	1888
7.9.14 获取未平仓仓位数量和列表.....	1891
7.9.15 读取订单和交易的历史.....	1892
7.10 内置对并行计算的支持: OpenCL.....	1897
结语.....	1905

面向交易员的 MQL5 编程指南

现代交易深度依赖于计算机技术。自动交易早已不局限于交易所和经纪公司，通过专门的软件解决方案，日常用户也可以使用自动交易。21 世纪初出现的 MetaTrader 是该领域的先驱之一。这一平台的最新版本 **MetaTrader 5** 依然处于领先地位，不断发展创新，提供创新特性和功能。

MetaTrader 5 中持续优化的一个核心要素是其内置编程语言 MQL5。利用 MQL5，交易员可将交易自动化（通常称为“算法交易”）提升到一个新的层次。借助 MQL5，交易员可以自行编写分析指标、操作执行脚本或用于全自动交易的 EA 交易，将自己的策略转化为实际应用。作为一种自动化交易系统，EA 交易能够自主运行，追踪价格变化并通过电子邮件或短信即时通知交易员。

这种内置编程语言让交易员几乎可以实现任何交易理念，无论是简单策略还是基于神经网络的复杂算法。MQL5 无缝融合了特定领域和通用编程语言的特性。经过多年发展，这种编程语言取得了长足进步，例如支持 3D 图形、可通过 OpenCL 进行并行计算、集成 Python 集成以及支持 SQLite 数据库。

要发挥 MetaTrader 5 的全部潜能，你必须深入研究编程。在这本书的帮助下，你将掌握 MQL5，并学习如何创建自己的交易应用程序。

本书假设读者已经熟悉 MetaTrader 5。另一个前提条件是读者理解促进交易的分布式信息系统中的终端操作基本原理。终端的 **帮助** 菜单提供了所有可用功能的详细介绍。

此外，借助 MQL5 API，交易员能够使用的功能远胜于 MetaTrader 5 GUI 提供的功能。掌握实现复杂交易方案的编程语言，自动化终端操作的各种方面，提高交易策略的效率。

本书分为七章，每章侧重于 MQL5 编程的不同方面。

- **第一章** 介绍 MQL5 编程基本原理和 MQL5 标准框架 MetaEditor。如果用户有其他语言的编程经验，那么应注意该框架的独有特性。
- **第二章** 解释基本术语，如类型、指令、运算符、表达式、变量、代码块和程序结构体。介绍如何以 MQL5 过程式编程风格使用这些术语。熟悉 MQL4 的用户可以跳过本章，从第三章开始读起。
- **第三章** 介绍 MQL5 中的面向对象编程 (OOP)。尽管 MQL5 与其他支持 OOP 范例的语言（特别是 C++）有相似之处，但也具备独有的特性。从风格上看，MQL5 有点像 C++。
- **第四章** 介绍常用嵌入式函数，这些函数适用于任何程序。
- **第五章** 介绍 MQL 程序的架构特性，以及专门用于执行各类交易任务的类型，例如使用指标进行技术分析、图表管理及通过在图表上嵌入醒目图形对象来标记图表，还包括对涉及 MQL 程序的交互操作和事件的响应。
- **第六章** 解释如何使用机器人分析交易环境和自动化交易操作。本章还介绍了程序在各种模式下与测试程序的交互，包括策略优化。
- **第七章** 包含有关扩展专用 API 集的信息，这些 API 可促进 MQL5 与数据库、网络数据交换、OpenCL、Python 等相关技术的集成。

全书兼顾了通用方法、示例和技术细节，内容均衡。通过概念渐进式学习，对读者进行逐步引导，破解编程学习中难以绕过的“先有鸡还是先有蛋”难题。为强化理解，本书中讨论的大多数 MQL 程序均在 MetaEditor/MetaTrader 5 中以源代码格式提供，以便读者实际探索。

第一章 MQL5 和开发环境简介

在 MetaTrader 5 中, MQL5 升级后最重要的变化之一是它支持面向对象编程 (OOP) 的概念。在 MQL4 (MetaTrader 4 的语言) 刚出现时, 人们通常将其与 C 编程语言相提并论, 而现在将 MQL5 比作 C++ 则更合理。公平地说, 所有最初只在 MQL5 中可用的 OOP 工具如今都已被移植到了 MQL4 中, 这一点不容忽视。然而, 对编程几乎一无所知的用户仍然认为 OOP 太复杂了。

从某种意义上说, 本书的目标就是化繁为简。本书并非是要取代终端自带的《MQL5 语言参考》(也可以在 mq5.com 网站上获得), 而是作为补充。

在本书中, 我们将采取循序渐进的方式, 向您介绍 MQL5 中的所有组件和编程技术, 确保每个知识模块都清晰明了, 就像任何强大的工具一样, 只有正确合理地使用, OOP 技术会逐渐释放其特别显著的潜力, 因此, MQL 程序的开发者将能够选择适合于特定任务的优选编程风格, 而不仅是面向对象的, 也包括“旧式”程序风格, 以及将其组合使用。

交易终端的用户可以简单分为“程序员”(至少已经有一种语言编程经验)和“非程序员”(纯交易员, 只是对使用 MQL5 的终端的定制能力感兴趣)。前者可以选择跳过本书说明语言基本概念的第一章和第二章, 立即开始学习 MetaTrader 5 中嵌入的特定 API (应用程序编程接口)。对于后者, 建议采用渐进式的阅读方法。

在“程序员”这一类别中, 懂 C++ 的人最具优势, 因为 MQL5 和 C++ 是相似的。然而, 这一“优点”也有它不利的一面。问题是 MQL5 与 C++ 并不完全相同 (尤其是与最近的标准相比)。因此, 凭借“有利方面”尝试编写某种结构时常常因编译器的意外错误而中断。考虑到语言的特定元素, 我们将尽力指出这些差异。

技术分析、执行交易指令或与外部数据源集成, 终端用户可以通过用户界面和 MQL5 的嵌入式软件工具使用所有这些功能。

由于 MQL5 程序必须执行不同的功能, 因此 MetaTrader 5 中支持一些专门的程序类型。这是许多软件系统中的标准技术。例如, 在 Windows 中, 除了常见的窗口程序, 还有命令行驱动的程序和服务。

MQL5 中提供了以下程序类型:

- 指标 - 旨在以图形方式显示由给定公式计算的数据数组的程序, 通常基于一系列报价;
- EA 交易 - 用于完全或部分自动化执行交易的程序;
- 脚本 - 旨在一次执行一个操作的程序;
- 服务 - 用于执行永久后台操作的程序。

我们将在下文详细讨论每种类型的用途和特性。需要注意的是, 它们都是在 MQL5 中创建的, 并且有很多共性。所以我们将从共性特征开始学习, 逐渐了解每种类型的特异性。

MetaTrader 的基本技术特征在于在客户终端中施加整个控制, 而在客户终端中发起的命令被发送到服务器。换句话说, 基于 MQL 的应用程序只能在客户终端内工作, 它们中的大多数需要与服务器实时连接才能正常工作。服务器上并没有安装任何应用程序。服务器只处理从客户终端收到的订单, 并返回交易环境的变化。这些变化也适用于 MQL5 程序。

大多数类型的 MQL5 程序都是在图表环境中执行的, 也就是说, 要启动一个程序, 您应该将它拖放到所需的图表上。只有一种特殊类型例外, 即服务: 服务是在后台运行的, 不会附加到图表。

我们回想一下, 所有 MQL5 程序都在 MetaTrader 5 的工作文件夹中, 在名为 `/MQL5/<type>` 的嵌套文件夹下, 其中 `<type>` 分别是:

- 指标
- EA 交易
- 脚本
- 服务

基于 MetaTrader 5 安装方式，工作文件夹的路径可能不同（特别是在 Windows 中以用户权限受限方式安装、正常模式或便携模式下）。例如，它可以是：

```
C:/Program Files/MetaTrader 5/
```

或者

```
C:/Users/<username>/AppData/Roaming/MetaQuotes/Terminal/<instance_id>/
```

用户可以通过执行 **文件 -> 打开数据目录** 命令（终端和编辑器中都可以找到该命令）来获得该文件夹的确切位置。此外，创建一个新程序时，由于使用了编辑器中嵌入的 MQL 向导，您无需考虑查找正确的文件夹。通过 **文件 -> 新建** 命令即可调用向导，并允许选择所需的 MQL5 程序类型。完成母版后，包含源代码模板的相关文本文件将在必要时自动创建，然后打开进行编辑。

在 MQL5 文件夹中，除了上述文件夹之外，还有其他嵌套文件夹，它们也与 MQL5 编程直接相关，我们将在后面提到。

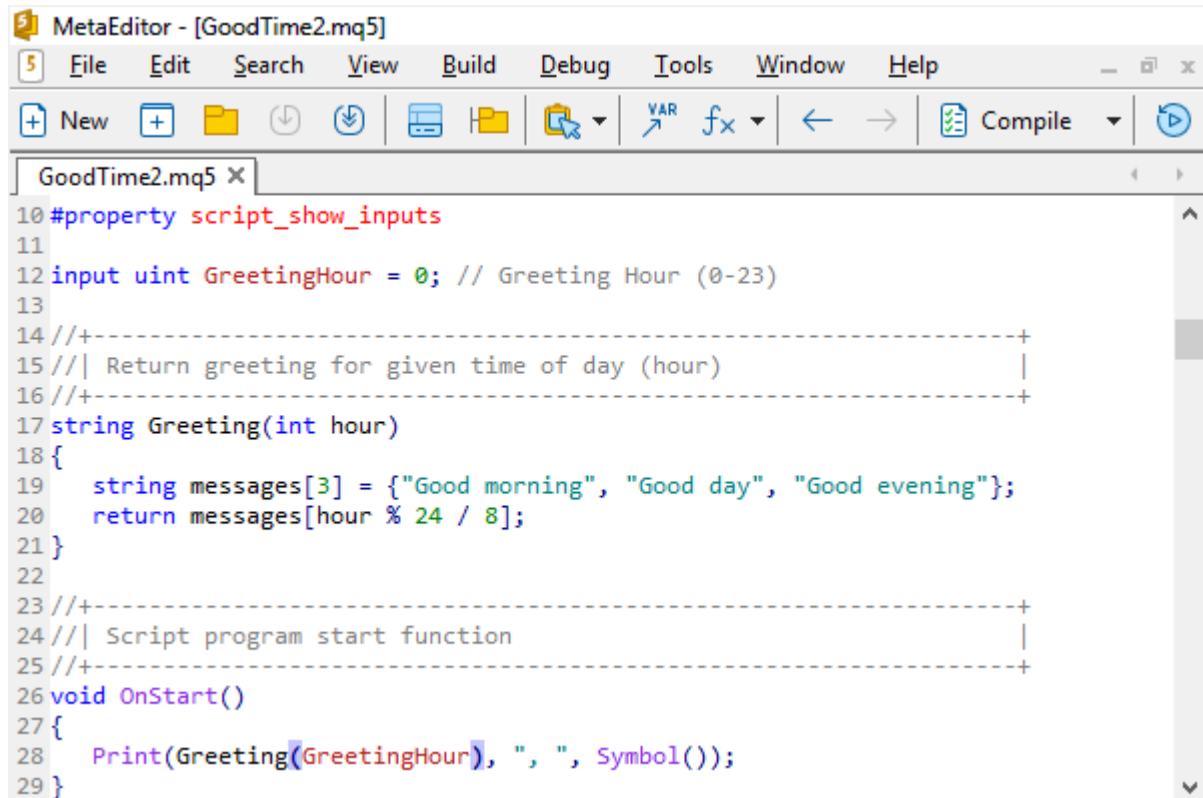
 面向交易员的 MQL5 编程指南 - 本书中的源代码第一章

 本书中的例子也可以在 **公共项目 \MQL5\Shared Projects\MQL5Book** 中找到。

1.1 编辑、编译和运行程序

所有 MetaTrader 5 程序都是可编译的。也就是说，用 MQL5 编写的源代码必须经过编译才能生成二进制文件，也就是终端中执行的文件。

程序通过 MetaEditor 进行编辑和编译。



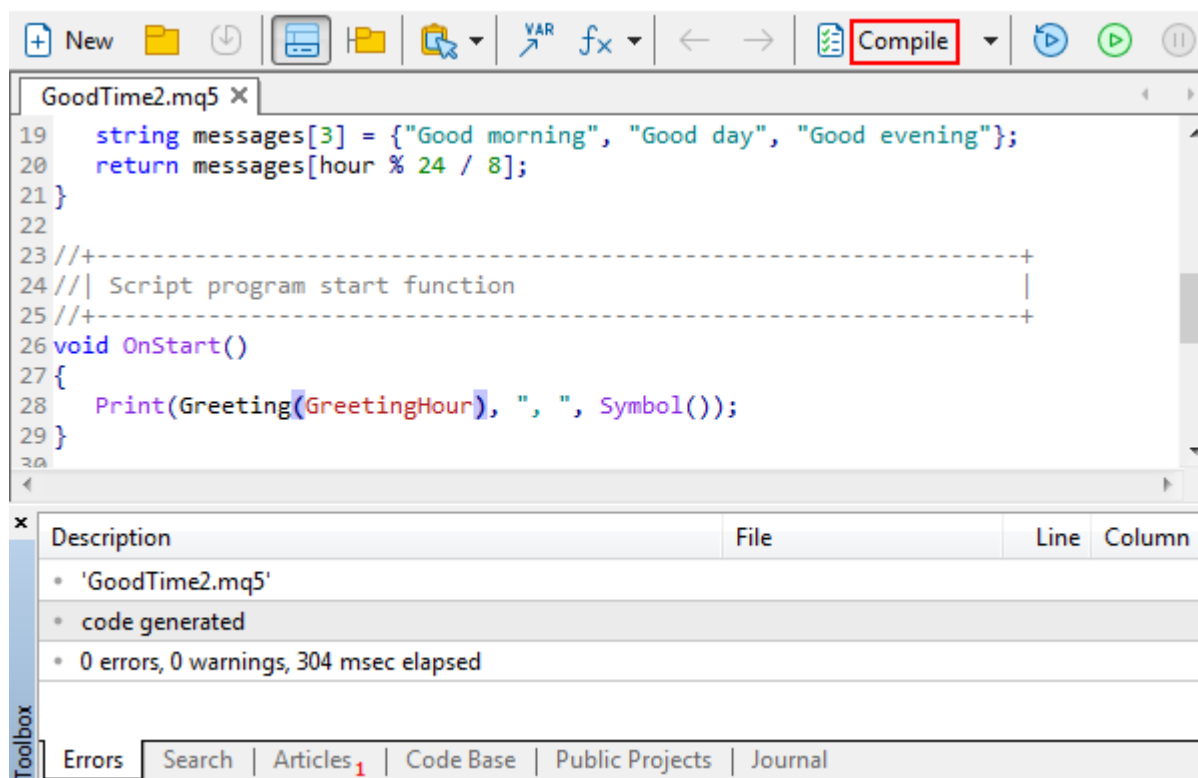
在 MetaEditor 中编辑 MQL 程序

源代码是根据 MQL5 规则编写的文本，保存为扩展名为 *mq5* 的文件。包含已编译程序的文件具有相同名称，但扩展名为 *ex5*。

在最简单的情况下，一个可执行文件对应一个包含源代码的文件；不过后面我们会看到，编写复杂程序时，经常需要将源代码拆分成多个文件：比如：一个主文件和若干个支持文件，支持文件以特殊方式从主文件中启用。在这种情况下，主文件的扩展名必须仍为 *mq5*，而通过这个主文件启用的文件的扩展名必须为 *mqh*。随后，所有源文件的语句将被传送到正在生成的可执行文件。因此，包含源代码的多个文件可以作为创建一个可执行文件/程序的起点。此处提到的所有这些内容将会在本书的第二章呈现，并进行全面论述。

我们将使用“MQL5 语法”一词表示 MQL5 中允许用于构造程序的所有规则的集合。只有严格遵守语法才能使编写的程序代码与编译器兼容。事实上，代码编写教学包括依次介绍一种特定语言的所有规则，在我们的例子中，这种语言就是 MQL5。这是本书的主要目的。

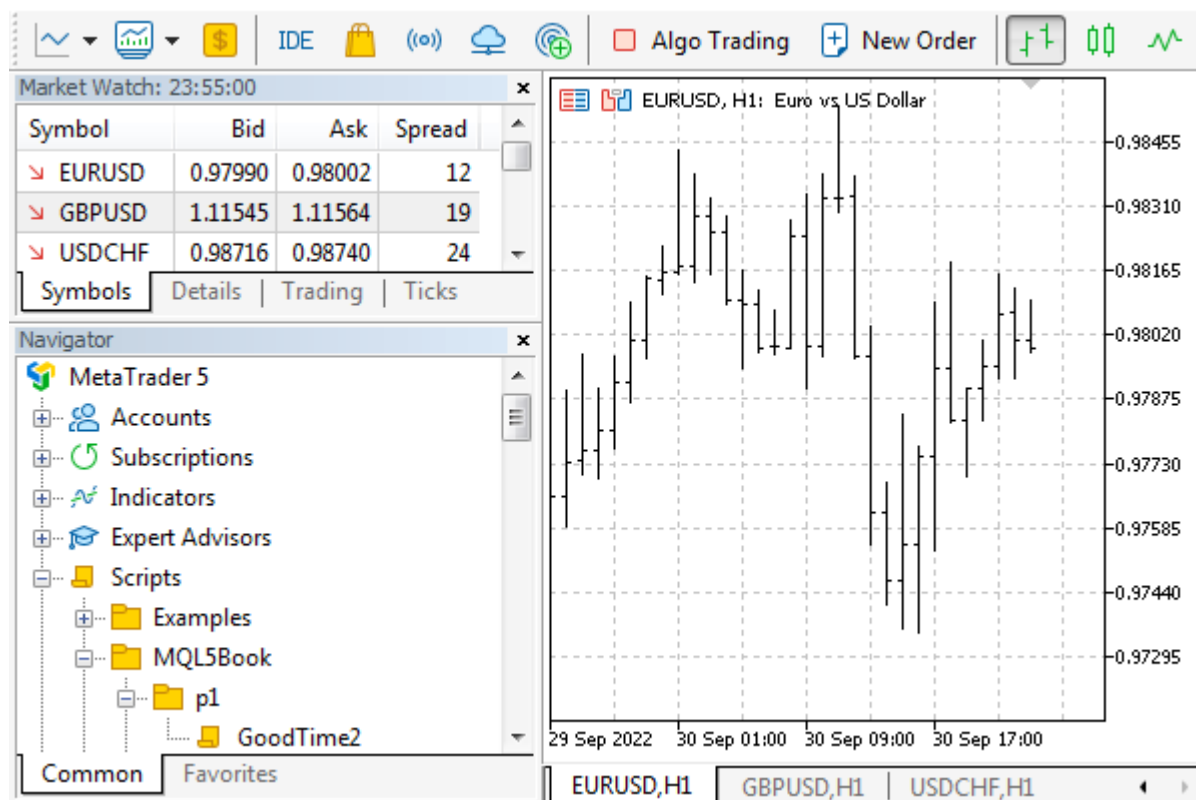
要编译源代码，我们可以使用 MetaEditor 命令 *文件* → *编译* 或直接按 **F7**。不过也有一些其他特殊的编译方法，我们将在后面进行讨论。编译的同时，编辑器日志中会显示状态变化（MQL5 程序由包含源代码的多个文件组成，并且会在单独一行日志中标记每个文件的启用）。



在 MetaEditor 中编译 MQL5 程序

编译成功的标志是零错误（0 个错误）。警告不会影响编译结果，只是告知潜在的问题。因此，建议按照解决错误的方式来解决这些警告（我们稍后将告诉您如何做）。理想情况下，不应有任何警告（0 个警告）。

成功编译 mql5 文件后，我们得到一个扩展名为 **ex5** 的同名文件。MetaTrader 5 导航器以树状结构显示文件夹 MQL5 及其子文件夹中的所有可执行 ex5 文件，包括刚编译的文件。



包含 MQL5 编译程序的 MetaTrader5 导航器

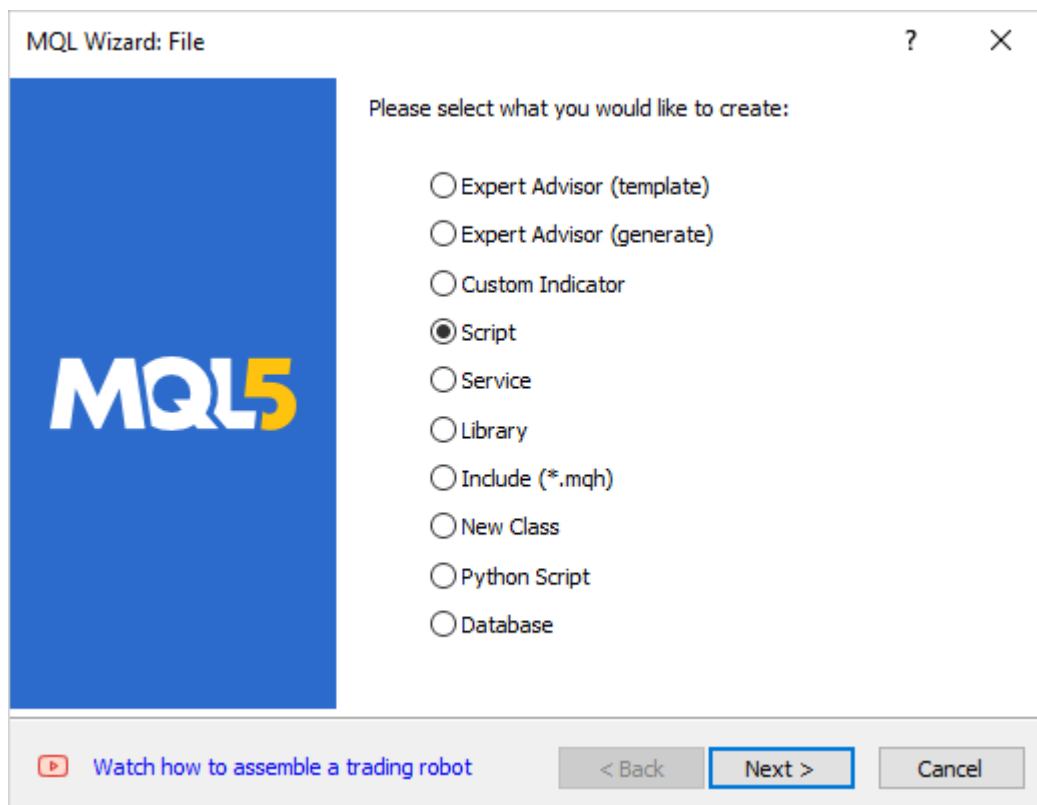
使用用户熟悉的任何方法在终端启动就绪的程序。例如，除了服务之外，可以用鼠标将任何程序从导航器拖动到图表中。对于服务的特性，我们将单独讨论。

此外，开发者通常需要在调试模式下执行程序，以找出导致错误的原因。针对这一需求，提供了多个特殊命令，我们将在 [Bug 修复和调试](#) 中介绍这些命令。

1.2 MQL 向导和程序草稿

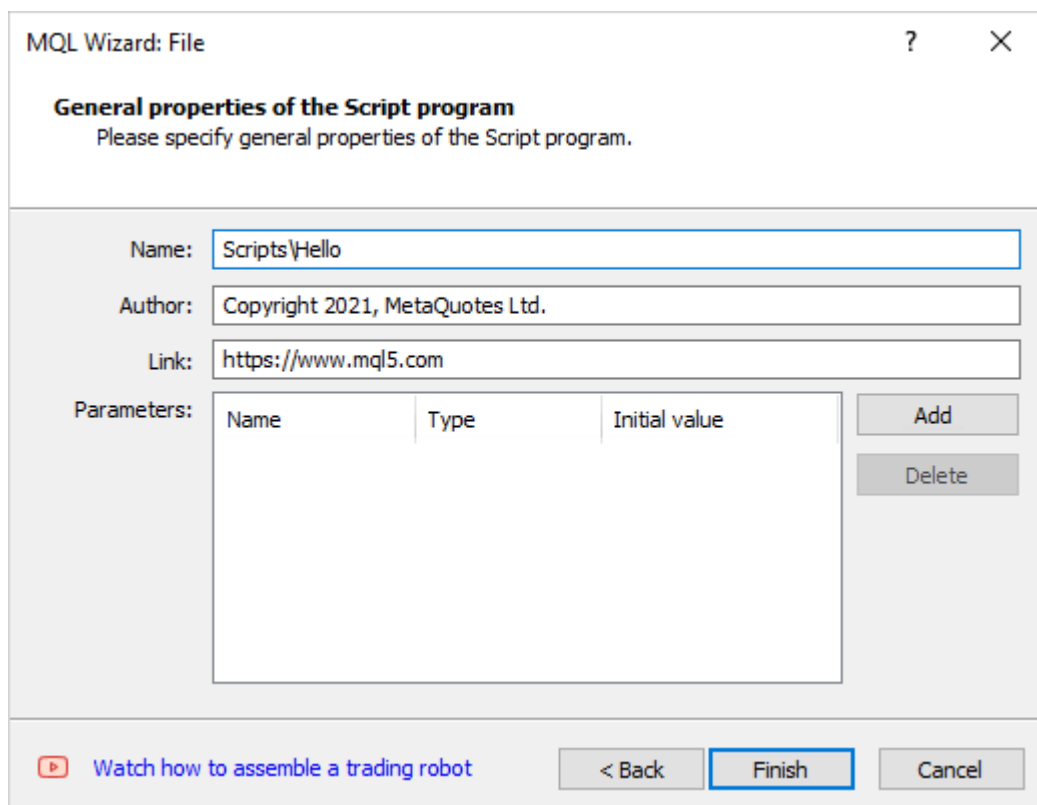
这里我们将分析最简单的 MQL 程序，没有任何实际功能。目的是介绍如何在编辑器中编写源代码、编译源代码以及在终端中启动编译后源代码。通过独立完成以下步骤，您可验证普通用户也能进行编程，并能开始逐步适应 MQL5 程序的集成开发环境。要巩固已学习的内容，这一步必不可少。

最简单的 MQL5 程序是脚本。因此，我们将尝试创建一个脚本。为此，我们启动 MQL5 向导（文件 → 新建）。第一步，我们在类型列表中选择脚本，然后按下下一步：



使用 MQL 向导创建脚本。第一步

第二步，我们在“名称”字段中填写脚本名称，需在上述默认文件夹和一个反斜杠的后面添加名称，即“Scripts\”的后面。例如，我们将脚本命名为“Hello”（即，“名称”字段将包含以下行：“Scripts\Hello”），不要更改任何其他选项，按**结束**。



使用 MQL 向导创建脚本。第二步

产生的结果是，向导将创建一个名为 **Hello.mq5** 的文件并打开以进行编辑。该文件位于 **MQL5/Scripts** 文件夹（脚本的标准位置），因为我们使用了默认文件夹；不过我们也可以添加任何子文件夹，甚至子文件夹层次结构。例如，如果我们在向导第一步的名称字段中填写 "Scripts\Exercise\Hello"，那么将在 **脚本** 文件夹中自动创建 **练习** 子文件夹，并且文件 **Hello.mq5** 将位于该子文件夹中。

本书中的所有示例都将位于为相关类型的 MQL 程序分配的目录下的 MQL5Book 文件夹中。这一点很有必要，既便于将示例安装到终端的工作副本中，也能避免与您已安装的其他 MQL 程序的产生名称冲突。

例如，本书随附的 **Hello.mq5** 文件位于 **MQL5/Scripts\MQL5Book\p1**，此处 **p1** 表示本示例与第一章有关。

Hello.mq5 脚本所生成的模板包含以下文本：

```
//+-----+
//|                                     Hello.mq5 |
//|                                     Copyright 2021, MetaQuotes Ltd. |
//|                                     https://www.mql5.com |
//+-----+

#property copyright "Copyright 2021, MetaQuotes Ltd."
#property link      "https://www.mql5.com"
#property version   "1.00"

//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
}

//+-----+
```

前面的 MetaEditor 和 MetaTrader 5 的截屏中显示的就是这个脚本。

所有以 `"/"` 开头的字符串都是注释，不会影响程序的功能。注释既不被编译器处理，也不被终端执行。只是用于在开发者之间交换解释性信息，或者在视觉上凸显代码部分以增强文本可读性。例如，在这个模板中，文件以一个包含注释的代码块开始，在这个代码块中应指定脚本名称和作者的版权。第二个注释代码块是脚本主函数的标题（将在下文详细介绍）。最后一个注释字符串从视觉上强调了文件结束。

以特殊指令 **#property** 开头的三个字符串，为编译器提供了一些特性，这些特性以特殊方式构建到程序中。在我们的例子中，到目前为止这些字符串并不重要，甚至可以删除。每个 MQL 程序类型都有具体的目录，在我们开始学习特定的程序类型后，就会知道这些目录。

脚本的主要部分由 **OnStart** 函数表示，我们将在此函数中描述程序操作的本质。这里我们必须学习“代码块”和“函数”的概念。

1.3 语句、代码块和函数

因此，在向导生成的脚本中，*OnStart* 函数如下所示：

```
void OnStart()
{
}
```

这正是我们在 MQL5 编程环境中的第一个主题。此时，我们再次遇到未知的概念和字符序列。为了进行解释，我们先要补充点其他知识。

程序在运行时通常必须实现以下几个典型阶段：

- 定义变量，即计算机存储器中的命名存储单元，用以存储数据；
- 组织源数据输入；
- 处理数据，应用的某种算法；以及
- 组织结果输出。

就维护程序的语法正确性而言，并非所有这些阶段都是必需的。例如，如果我们创建一个计算 2×2 乘积的程序，显然不需要任何输入数据，因为乘法所需的数字已整合在程序文本中。此外，由于 2 和 2 在该表达式中是常数值，所以在程序中不需要命名存储单元（变量）。由于我们已经知道了二乘二是多少，就不需要传达乘积数了。当然，这样的程序没有任何实际功能。但从技术角度来看，又是绝对正确的。

更有趣的是，程序可能不包含任何处理语句。我们的脚本模板专门提供一个示例程序 `'null'`。那么上面的文本片段是什么？

在 Niklaus Wirth（编程界的一位大腕）的时代，他给编程下了一个简单的概括性定义，即编程是算法和数据结构的共生关系。

“算法”是指特定编程语言的语句序列。语句也是一种句子，可以看做是根据其语法规则用编程语言表达的完整话语。“语句”这个名字本身就表明它被计算机视为操作指南。换句话说，语句描述了何时以及何种方式处理所需的应用数据结构。这恰恰解释了为何算法和数据结构的相互渗透能将作者的想法付诸实践。

遗憾的是，在大多数实际任务中，语句数量过于庞大，以至于必须在一定程度上系统化，以便人类识别和控制程序行为。

此时，分治算法也派上了用场，这种算法在编程中几乎无处不在，只是以不同形式出现。在本书的后续学习中，我们将继续学习这些概念，现在我们仅关注其本质。

我们知道，该算法简化为将一个庞大的复杂任务分解成多个规模缩小且难度降低的子任务。我们可以将这个过程的比作建造房子或组装航天器。这两种“产品”都是由多个不同的模块组成，这些模块又由组件组成，而组件又由更小的零件组成，以此类推。

将这种相似性延伸到算法领域，我们可以说语句是很小的零件，而整个程序就是一个房子/航天器。因此，我们需要大小适中的结构块。

这就是为什么在实现算法时，习惯于将逻辑上相关的语句组合成更大的已命名字段，即函数。在程序需要时，我们可以通过函数名来寻址函数（即调用函数），从而要求计算机执行该函数中包含的所有语句。事实上，整个程序是最大的外部块，因此，它也可以由函数来表示，从函数中调用更小的函数，或者如果语句不多，就立即执行语句。现在我们来分析 *OnStart* 函数。

在脚本中，**OnStart** 是保留名称，表示当用户使用上下文菜单命令或通过图表上拖动鼠标启动脚本时，终端本身调用的最终函数，以此作为对用户操作的响应。因此，代码前面的片段定义了 **OnStart** 函数，此函数预先决定了整个脚本的行为。

如果懂其他编程语言，比如 C、C++、Rust，或者 Kotlin，就会注意到该函数和 **main** 函数很相似，都是进入程序的核心入口。

任何脚本都必须包含 **OnStart** 函数。否则，编译结束时会出现报错。

脚本以任何方式启动后，终端首先会执行空函数 **OnStart**（如上述示例中所示），并立即完成其操作。严格来说，我们的脚本中还没有应用任何算法，但是已存在一个存根函数用于添加算法。

在其他类型的 MQL 程序中，还有一些特殊函数需要由程序员在其代码中定义。我们将在本书的相关章节讨论具体特性。

我们将在本书的第二章详细讨论函数定义语法。要通过动手实践来回顾，考虑以下基本要点就足以理解对 **OnStart** 的说明。

由于函数通常旨在获得相应的结果，因此期望值的特征在其定义中以特殊的方式描述：应获得哪些数据类型，以及这些数据是否必要。有些函数可以执行不需要返回值的操作。例如，某函数可用于更改当前图表的设置，或者在账户达到预定义的提款级别时发送推送通知。所有这些操作都可以通过函数中的语句进行编程，并且不会创建任何新数据（自然也无需返回给程序的任何其他部分）。

我们的情况与此类似：作为脚本的主函数，**OnStart** 可以在完成后仅将其结果返回给外部环境（直接返回给终端），但这不会脚本操作本身产生任何影响（因为脚本已经执行完毕）。

正因如此，**OnStart** 函数名前面有 **void** 一词，就是为了通知编译器该结果对我们不重要（**void** 意思是空）。**void** 是 MQL5 中的众多过程保留字之一。编译器知道所有保留字的含义，并且在审查源代码时以它们为指导。特别是，程序员可能使用保留字来为编译器定义新术语，例如 **OnStart** 函数本身。

名称后的圆括号对于任何函数说明都是必需的：圆括号中可以包含函数参数列表。例如，我们要编写一个函数，用于求一个数字的平方，则必须为该函数提供一个参数来表示这个数字。然后，我们可以从程序的任何部分调用该函数，并向其传送一个自变量，即该参数的具体值。后面我们会学习如何描述参数列表；本例中并未出现。此要求是针对 **OnStart** 函数提出的，因为它是由终端本身调用的，并且不会将任何内容作为参数发送至该函数。

最后，花括号用于标记包含语句的代码块的开始和结束。此类代码块紧跟在函数名字符串之后，包含由该函数执行的一组运算。也被称为函数体。在本例中，花括号内没有任何内容。因此，脚本模板不会执行任何操作。

上述由关键字 **void**、名称 **OnStart**、空参数列表和空代码块的序列，为编译器定义了 **OnStart** 函数的最少空实现。稍后我们会在函数体中添加语句，从而扩展 **OnStart** 函数的定义。

执行编译命令后，我们将确保脚本可以成功编译，并且就绪程序出现在文件夹 **Scripts/MQL5Book/p1** 终端的导航器中。这是因为在磁盘上的相关文件夹中，现在有了 **Hello.ex5** 文件。可以在任何文件管理器中轻松验证这一点。

我们可以对图表运行脚本，但要确认脚本是否已经执行，唯一证据是终端日志中的条目（工具窗口中的日志选项卡；请勿与工具栏混淆）：

```
Scripts      script Hello (EURUSD,H1) loaded successfully
Scripts      script Hello (EURUSD,H1) removed
```

即脚本被加载，控件被发送给 **OnStart** 函数，但因为函数没有处理而立即被返回至终端，之后终端从图表中卸载该脚本。

1.4 第一个程序

让我们试着在脚本中添加一些简单但具有说明性的内容，以演示脚本的运行。我们将修改后的脚本重命名为 **HelloChart.mq5**。

在许多编程教材中，第一个例子都是打印经典的 “Hello, world”。在 MQL5 中，类似的问候语如下所示：

```
void OnStart()
{
    Print("Hello, world");
}
```

但我们会让它的信息量更丰富：

```
void OnStart()
{
    Print("Hello, ", Symbol());
}
```

因此，我们只添加了一个字符串，但包含多个语言结构。

此处，**Print** 是嵌入在终端的函数的名称，用于在 *EA* 交易日志中显示消息（工具窗口中的选项卡 *EA* 交易；尽管此选项卡名为 *EA* 交易，但会收集所有 MQL 程序类型的消息）。与我们单独定义的 **OnStart** 函数不同，**Print** 函数是预先为我们定义的，并且是永久性的。**Print** 是构建 MQL5 API 应用程序编程接口)的多个嵌入式函数之一。

代码中新增的代码行表示调用 **Print** 函数的语句，用于向函数发送自变量列表（圆括号中），这些自变量将打印在日志中。列表中的自变量用逗号分隔。本例中有两个自变量，分别是：代码行 “Hello ” 以及对另一个嵌入式函数 **Symbol** 的调用，后者返回当前图表上活动金融工具的名称（从中获得的值将立即被传入 **Print** 函数自变量列表中，即调用 **Symbol** 函数的位置）。

Symbol 函数没有任何参数，因此不会向圆括号中传送任何内容。

例如，如果是对 “EURUSD” 图表运行脚本，那么调用 **Symbol()** 函数将返回 “EURUSD”，对于正在执行的程序，调用 **Print** 函数的语句将有所不同，如下所示：**Print("Hello, ", "EURUSD")**。当然，从用户的角度来看，所有这些对函数的调用和中间结果的动态替换都是无缝且即时完成。不过对于程序员来说，充分认识到程序的逐步执行过程至关重要，这样既能避免逻辑错误，又能确保代码严格符合预期设想。

双引号内的 “Hello ” 行被称为字面量，也就是被计算机按原样识别为文本的固定字符序列（与在程序源代码中的形式完全一致）。

因此，上述打印语句必须在日志中逐个打印这两个自变量，这应导致将两行合并，得到 “Hello, EURUSD”。

重要的是，引号内的逗号将作为该行的一部分在日志中打印，不会进行任何特殊处理。与此不同的是，右引号之后且 **Symbol()** 调用之前的逗号是自变量列表中的分隔字符，会影响程序行

为。如果省略第一个逗号，程序仍然是正确的，只是在打印的文字 “Hello” 后面没有逗号。但如果省略第二个逗号，程序将停止编译，因为函数自变量列表的语法将被破坏：其中的所有值（本例中有两行）必须用逗号分隔。

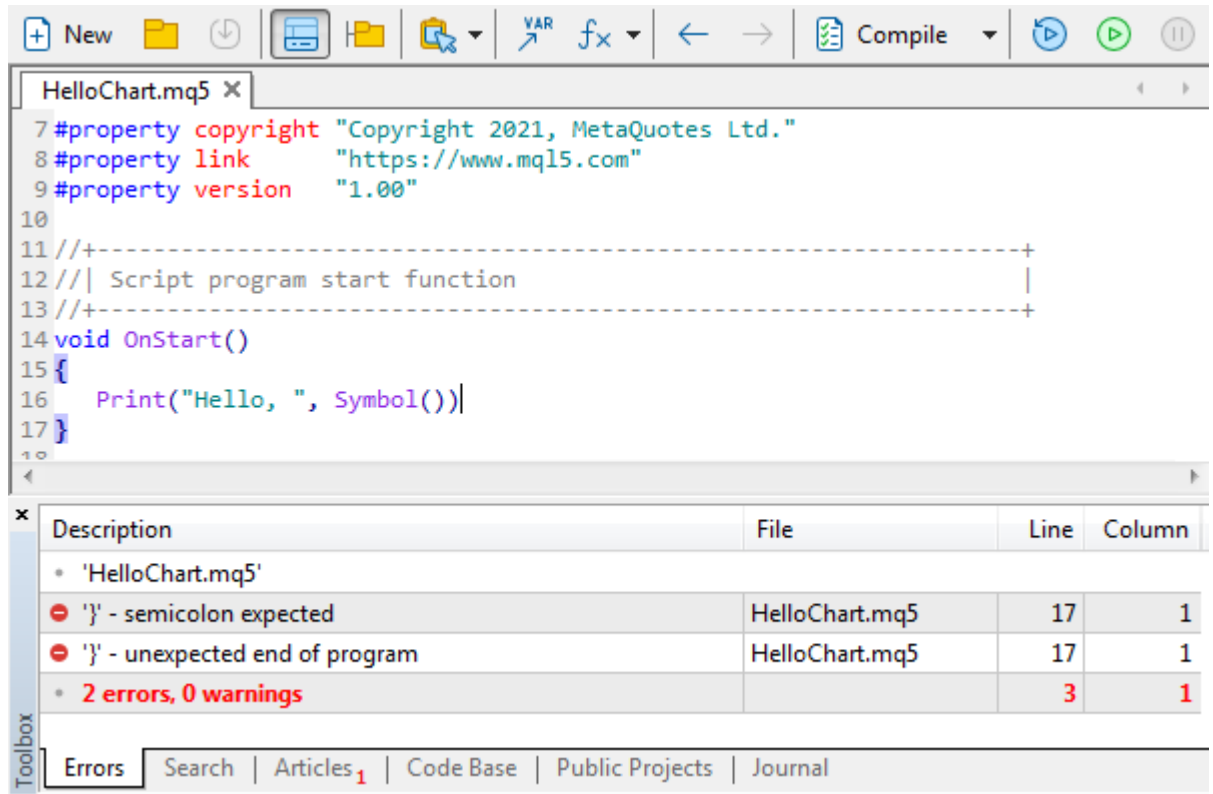
编译器错误如下所示：

```
'Symbol' - some operator expected HelloChart.mq5      16      19
```

编译器会报错，提示在引用 *Symbol* 之前缺少关键内容。这将中断编译，也不会创建程序的可执行文件。因此，我们要把逗号放回原处。

这个例子向我们展示了严格遵循语言语法的重要性。同样的字符，在程序中的位置不同，作用也会有所不同。可以说是失之毫厘，谬以千里。例如，注意调用 *Print* 的代码行结尾的分号。分号在这里表示语句结束。如果我们忘记输入这个分号，可能会出现奇怪的编译器错误。

要验证这一点，我们尝试移除这个分号并重新编译脚本。这将导致出现新错误，其中包含有关问题的说明以及在源代码中的位置。



MetaEditor 日志中的编译错误

```
'}' - semicolon expected      HelloChart.mq5      17      1
'}' - unexpected end of program HelloChart.mq5      17      1
```

第一个错误明确指出缺少编译器所期望的分号。第二个错误是传播性的：在当前语句结束之前，检测到了表示程序结束的右花括号。在编译器看来，它将继续编译，因为它还没有遇到分号。修复这类错误的方法很明显：必须将分号放回语句中的正确位置。

我们来进行编译并启动修复后的脚本。它执行速度非常快，几乎从图表中瞬间消失，同时在 *EA* 交易日志中显示一条记录，确认脚本运行。

```
HelloChart (EURUSD,H1)      Hello, EURUSD
```

1.5 数据类型和值

除了调用嵌入式函数 *Symbol*，我们还可以使用我们自己在源代码中定义的函数。假设我们希望在日志中不仅打印 "Hello"，还要根据一天中的不同时间打印不同的问候语。我们将一天中的时间精确到小时：0-8 是上午，8-16 是下午，16-24 是晚上。

合乎逻辑的建议是，新函数的定义结构必须与我们熟悉的 *OnStart* 函数的定义结构相似。但名称必须是唯一的，不能与其他函数或保留字的名称重复。我们将在本教材中进一步研究这些关键字的列表，幸运的是现在 *Greeting* 可以用作名称。

与 *Symbol* 函数一样，该函数必须返回一个字符串；不过该字符串必须是以下短语之一，具体取决于一天中的时间："Good morning"、"Good afternoon" 或 "Good evening"。

根据常识，我们在本文使用了字符串的常见概念。显然编译器很熟悉这一概念，因为我们看到了它生成打印预定义文本程序的方式。至此我们已顺利引入了编程语言中“类型”的概念，其中一种类型是字符串，即字符序列。

在 MQL5 中，这种类型由 *string* 关键字进行描述。这是我们知道的第二种类型，第一种类型是 *void*。我们此前已经看到了这种类型的值，只是当时不知道的是：它是字面量 "Hello, "。当我们只是在源代码中插入一个常量（特别是引用的文本）时，无需提供其类型说明，编译器会自动正确定义类型。

以 *OnStart* 函数说明为范例，我们可以大致推断 *Greeting* 函数的形式。

```
string Greeting()
{
}
```

该文本表明我们创建 *Greeting* 函数的意图，该函数可以返回任意 *string* 类型的值。不过要让函数真正返回一些内容，就必须使用带有 *return* 运算符的特殊语句。它是 MQL5 众多运算符中的一个：我们将在下文进行深入探讨。如果函数的返回值类型不是 *void*，则它必须包含运算符 *return*。

特别是，要从函数返回之前的问候语字符串 "Hello, "，我们应该这样编写：

```
string Greeting()
{
    return "Hello, ";
}
```

return 运算符停止函数执行，并将其右侧的内容作为结果返回。输出中会隐藏用于调用函数的源代码片段。

我们还没有探讨编写可形成任意字符串的表达式的所有方法。不过这里沿用了带有引用文本的最简单例子（未做任何改变）。重要的是，返回值类型必须与函数类型一致，如我们的例子中所示。在语句的结尾添加了一个分号。

但我们希望根据一天中的不同时间生成不同的问候语。因此，该函数必须有一个用于定义小时的参数，其取值范围为 0 到 23。显然，小时数是一个整数，即一个没有小数部分的数字。很明显，时间不会停留在整点，分钟是计算在小时内的，分钟数也是一个整数。同样可以看出，就本例而言，精确到分钟来判断时段没有实际意义。因此，我们将只限于按小时数选择问候语。

对于整数值，MQL5 中有一个特殊类型 *int*。需要从程序中调用 *Greeting* 函数的位置向该函数传递此值。这里，我们首先需要描述一个命名存储单元，即变量。

1.6 变量和标识符

变量是一个具有唯一名称的存储单元 (引用时不会出错), 可以存储某种类型的值。编译器以特殊的内部格式为变量分配足够的所需内存, 从而保证了这种能力: 每种类型都有大小和相应的内存存储格式。第二章中会详细介绍这一点。

基本上, 程序中有一个更严格的术语叫标识符, 这个术语适用于变量、函数和许多其他实体的名称, 我们后面会逐一学习。标识符遵循一些规则。特别是, 它只能包含拉丁字符、数字和下划线; 并且不得以数字开头。这就是为什么之前为函数选择的“问候语”符合这些要求。

一个变量可以有不同的值, 并且这些值可以在程序执行过程中使用特定语句更改。

除了变量的类型和名称之外, 变量还有一个特征是上下文, 即程序中的一个区域, 在此区域中, 可以定义并使用变量, 不会引发任何编译器错误。我们的例子可能有助于理解这个概念, 而不需要从一开始就进行详细的技术推理。

关键在于, 变量的特定实例是函数参数。该参数用于将某个值发送到函数中。很明显, 包含该变量的代码片段, 必须被限制在函数主体中。换句话说, 该参数可以在函数块内部的所有语句中使用, 但不能在函数块外部使用。如果编程语言允许参数在外部随意使用, 将成为许多错误的根源, 因为这有可能从与函数无关的随机程序片段中“破坏”函数内部。

在任何情况下, 上述对于变量的定义虽然略显简单, 但对于入门性章节来说已足够了。我们稍后将分析一些更细微的差别。

那么, 我们来概括一下变量和参数的相关知识: 它们必须具有类型、名称和上下文。我们将前两个特征明确地写在代码中, 而最后一个特征则取决于定义位置。

我们来看看如何在 **Greeting** 函数中定义小时数的参数。我们已经知道所需的类型为 **int**, 便能以合乎逻辑的方式选择名称: **hour**。

```
string Greeting(int hour)
{
    return "Hello, ";
}
```

无论小时数是多少, 该函数都将返回 “Hello,”。现在我们应添加一些语句, 根据 **hour** 参数的值选择返回不同的字符串。请记住, 有三种可能的函数响应选项: “Good morning”、“Good afternoon”和 “Good evening”。可以假设我们需要 3 个变量来描述这些字符串。然而, 在这种情况下使用数组要方便得多, 数组确保用统一的编码算法访问元素。

1.7 赋值和初始化、表达式和数组

数组是一组命名的同类型存储单元, 这些存储单元在内存中连续分布, 每个存储单元都可以通过其索引进行访问。在某种意义上, 数组是一个复合变量, 特征包括一个公共标识符、存储值类型和编号元素数量。

例如, 由 5 个整数组成的数组可以描述如下:

```
int array[5];
```

数组大小在名称后的方括号中指定。元素从 0 到 N-1 编号, 其中 N 是数组大小。访问元素 (即读取值) 时使用的语法类似。例如, 要将上述数组的第一个元素打印到日志中, 我们可以编写以下语句:


```
Print(array[0]);
```

请注意，索引 0 对应第一个元素。要打印最后一个元素，语句将替换为：

```
Print(array[4]);
```

当然，在打印数组元素之前，需先向数组中写入有效值。要完成这种记录操作，需要使用一种特殊语句，即赋值运算符。该运算符的一个特殊特性是使用了 `=` 符号，在符号左边指定数组元素（或变量），在其中进行了记录，符号右侧指定要记录的值或“等效表达式”。这里的“等效表达式”隐含了对计算算术、逻辑和其他类型表达式方面的语言能力要求（我们将在第二章学习这些内容）。表达式的语法与我们在学生时代学过的算术和代数中方程式的书写规则非常相似。例如，可以在表达式中使用加（+）、减（-）、乘（*）、除（/）运算。

以下是几个关于运算符的示例，用于填写上述数组中某些元素。

```
array[0] = 10;           // 10
array[1] = array[0] + 1; // 11
array[2] = array[0] * array[1] + 1; // 111
```

这些语句演示了以不同方法来进行赋值和构造表达式：在第一个字符串中，字面量 10 被写入 `array[0]` 元素中，而在第二行和第三行中则使用了表达式进行计算，从而获得结果（为便于查看，已在注释中注明）。

当表达式中包含数组元素（一般情况下是变量）时，计算机会在程序执行期间从内存中读取数组元素的值，并对这些值执行上述运算。

务必要区分赋值语句中 `=` 字符左边和右边的变量及数组元素的用法：左边是已处理数据的“接收者”（始终是单个），而右边是用于计算的初始数据的“源”（一个表达式中可以有多个“源”，就像本例的最后一个字符串，其中元素 `array[0]` 和 `array[1]` 的值相乘）。

在我们的示例中，`=` 字符用于将值赋给预定义数组的元素。但在有些情况下，在定义变量和数组时立即赋予初始值会很方便。这被称为初始化。`=` 字符也用于初始化。我们结合应用任务的背景来考虑这个语法。

我们用 **Greeting** 函数内部的问候语选项来描述字符串数组：

```
string Greeting(int hour)
{
    string messages[3] = {"Good morning", "Good afternoon", "Good evening"};
    return "Hello, ";
}
```

在添加的语句中，不仅定义了包含 3 个元素的 `messages` 数组，还定义了初始化方式，即使用所需的初始值进行填充。初始化操作会在变量/数组名称和类型说明上高亮显示 `=` 字符。对于变量，必须在 `=` 后面指定一个值，不需要花括号，而对于数组，可以看到，可以写入多个值，用逗号隔开，并用花括号括起来。

请勿将初始化和赋值相混淆。前者是在定义变量/数组时指定的（只进行一次），而后者出现在特定的语句中（同一个变量或数组元素可以反复被赋予不同的值）。数组元素只能单独赋值：MQL5 不支持一次为所有元素赋值（初始化时可以为所有元素赋值）。

在函数内部定义的 `messages` 数组只能在函数内部使用，比如参数 `hour`。然后我们将看到如何描述在整个程序代码中可用的变量。

如何将带有小时数的传入值 `hour` 转换为三个元素之一？

回想一下，根据我们的思路，*hour* 的值可以从 0 到 23。如果我们将这个值范围除以 8，将得到从 0 到 2 的值。例如，1 除以 8 将得到 0，7 除以 8 将得到 0（整除时，小数部分忽略不计）。然而，8 除以 8 得 1，那么所有从 8 到 15 的数字除以 8 都得 1。数字 16-23 对应除法结果 2。获得的整数 0、1 和 2 将用作索引以读取 *messages* 数组元素。

在 MQL5 中，/ 运算可用于整数的整除。

用于获得除法结果的表达式与我们刚刚处理 *array* 时使用的表达式类似，只是必须使用 *hour* 参数和 / 运算。我们将使用以下语句来演示将 *hour* 转换为元素索引的可能实现方式：

```
int index = hour / 8;
```

这里定义了一个新的整数变量 *index*，并由以上表达式的值进行初始化。

不过我们可以省略在 *index* 变量中保存中间值，并立即将这个表达式转移到方括号中（转移到 = 的右边），其中指定了数组元素编号。

在带有 *return* 运算符的语句中，我们可以提取相关的问候语如下：

```
string Greeting(int hour)
{
    string messages[3] = {"Good morning", "Good afternoon", "Good evening"};
    return messages[hour / 8];
}
```

这个函数已经基本完成。不过在学习几个章节后，我们会做一些修正。至此，我们将项目以另一个名称 *GoodTime0.mq5* 保存在文件中，并尝试调用我们的函数。因此在 *OnStart* 中，我们将在 *Print* 调用内部使用 *Greeting* 调用。

```
void OnStart()
{
    Print(Greeting(0), ", ", Symbol());
}
```

我们保留了问候语和金融工具名称之间的分隔逗号（在 *fileb*，”旁边）。现在 *Print* 函数调用中有三个自变量：第一个和最后一个自变量将分别调用函数 *Greeting* 和 *Symbol* 进行动态计算，而逗号将按原样发送打印。

至此，我们将向 *Greeting* 函数传送常量 0。它的值将进入 *hour* 参数。编译并启动程序后，我们可以确保它在日志中打印出所需文本。

```
GoodTime0 (EURUSD,H1)      Good morning, EURUSD
```

不过在实践中，问候语必须根据用户指定的时间动态地选择。

至此，我们已经进展到需要设计数据输入的阶段。

1.8 数据输入

将数据传输到 MQL 程序的基本方法是使用输入参数。从许多方面来看，它们类似于函数的参数，本质上就是变量，特别是在描述语法和在代码中后续使用的原则方面。

此外，输入参数说明有一些重要区别：

- 放在所有代码块之外的文本中（我们已经学习了组成函数体的代码块，我们将在后面学习其他代码块），或者换句话说，在任何一对花括号之外；

- 以 *input* 关键字开头；
- 用默认值初始化。

通常建议将输入参数放在源代码的开头。

例如，要在我们的脚本中定义一个输入参数以输入小时数，需在那三行指令 *#property* 后面直接添加以下字符串：

```
input int GreetingHour = 0;
```

这条记录有多重含义。

- 首先，脚本中现在有了 *GreetingHour* 变量，可以在源代码的任何位置使用，包括从任何函数内部使用。这个定义被称为全局级定义，这是由于执行了以上列表中的第 1 项。
- 其次，使用 *input* 关键字会使该变量在程序内部和 MQL5 程序属性对话框的用户界面中可见（该对话框在启动时打开）。因此，程序启动时，用户设置必要的参数值（本例中是一个参数 *GreetingHour*），这些值在程序执行过程中成为相应变量的值。

再次需要注意的是，我们在代码中指定的默认值将在对话框中显示给用户。不过用户能够更改这个值。在这种情况下，这个新的、手动输入的值（非初始值）将包含在程序中。

输入参数的初始值受多种因素的影响：代码中的初始化方式、用户在启动它们时选择的交互方式、MQL5 程序类型以及程序启动方式。问题是不同类型的 MQL5 程序在图表上启动后会有不同的生命周期。因此，指标和 EA 交易在图表中进行一次加载后，便会在其中永久等级，直到用户明确移除它们。因此，终端会记住最近选择的设置并自动使用，例如，在终端重启后。不过在不同终端会话之间，脚本不会保存在图表中。因此，当我们启动脚本时，可能只向我们显示默认值。

遗憾的是，由于某种原因，输入参数的说明无法保证在脚本开始时会调用设置对话框（仅针对作为独立 MQL5 程序类型的脚本）。要实现这一点，必须在代码中再添加一个特定于脚本的 *#property* 指令：

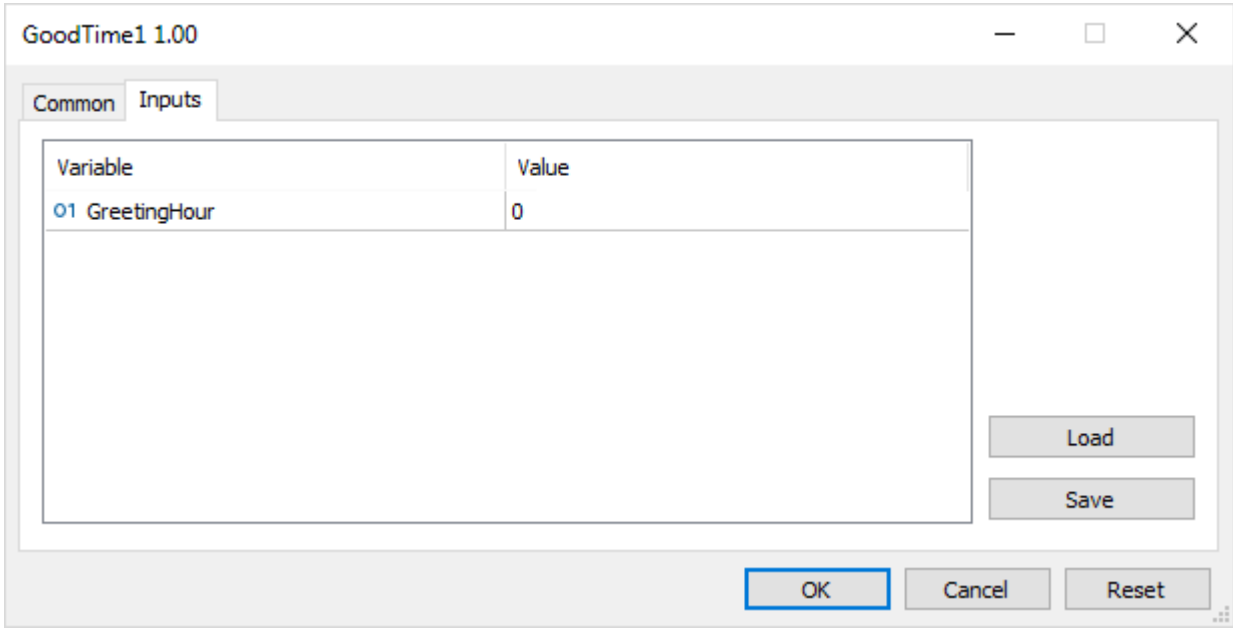
```
#property script_show_inputs
```

正如我们将看到的，其他类型的 MQL5 程序无需使用该指令。

我们需要 *GreetingHour* 以将它的值传入 *Greeting* 函数。为此，只需在 *Greeting* 函数调用中插入此变量，而不是 0：

```
void OnStart()
{
    Print(Greeting(GreetingHour), ", ", Symbol());
}
```

考虑到我们为描述输入参数所做的更改，我们将新的脚本版本保存在 *GoodTime1.mq5* 文件中。如果我们编译并启动这个文件，将看到数据输入对话框：



用于为 GoodTime1.mq5 脚本输入参数的对话框

例如，如果我们将值 *GreetingHour* 编辑为 10，那么脚本将显示以下问候语：

```
GoodTime1 (EURUSD,H1)      Good afternoon, EURUSD
```

这个结果是正确的，也是预期的。

为了好玩儿，我们再运行一次脚本并输入 100。我们得到的不是任何有意义的回应，而是：

```
GoodTime1 (EURUSD,H1)      array out of range in 'GoodTime1.mq5' (19,18)
```

我们遇到了一个新现象，即运行时错误。在这种情况下，终端提示，在第 19 行第 18 列的位置，我们的脚本尝试读取一个索引不存在的数组元素值（超出数组大小）。

由于程序员始终且必然会遇到各种错误，我们必须学习如何修复它们，所以让我们进行更加细致的探讨。

1.9 错误修复与调试

编程的艺术就在于对程序的操控，既要明确指示程序的执行内容和方式，也要加以防范，以免出现错误行为。遗憾的是，由于影响程序行为的因素众多且难以发现，导致后者的执行难度很大。数据错误、资源不足、他人和自己的编码错误只是问题的冰山一角。

没人能保证编写的程序不会出错。错误可能发生在不同的阶段，可以简单地分为以下几种：

- 编译错误：编译器在识别出不符合所需语法的源代码时返回（我们在上文已经了解了这种错误）；这种错误最容易修复，因为编译器会搜索这类错误；
- 程序运行时错误：终端在程序中出现不正确情况时返回（如被零除、计算负数的平方根或我们的例子中提到的试图引用数组中不存在的元素）；这类错误比较难以检测，因为这类错误通常不会在所有输入参数值中出现，而只在特定条件下出现；
- 程序设计错误：导致程序完全关闭，没有来自终端的任何提示，例如陷入无限循环；就定位和重现这些错误而言，这样的错误可能是最复杂的，而能否重现程序中某个问题是之后进行修复的必要条件；

- 隐藏错误：程序看似正常运行，但给出的结果不正确； $2*2$ 不等于 4 这样的错误很容易检测到，但一些细微偏差就很难发现了。

我们回到脚本的具体情况。根据 MQL 程序运行时环境提供给我们的错误消息，以下语句是错误的：

```
return messages[hour / 8]
```

在计算数组中元素的索引时，根据 *hour* 变量的值，可能会获得一个超过数组大小 3 的值。

嵌入在 MetaEditor 中的调试器确保这种情况确实会发生。调试器的所有命令都整合在“调试”菜单中。这些命令提供了许多有用的功能。在本文中，我们仅说明两个功能：*调试* → *从实际数据开始* (F5) 和 *调试* → *从历史数据开始* (Ctrl+F5)。您可以在 MetaEditor 帮助中查看其他功能的帮助信息。

这两个命令都以一种特殊的方式编译程序：使用调试信息。这种版本的程序还没有像在标准编译中那样进行优化（有关优化的更多详细信息，请参见文档），同时，它允许使用调试信息在执行期间透视程序内部状态：查看变量和函数调用堆栈的状态。

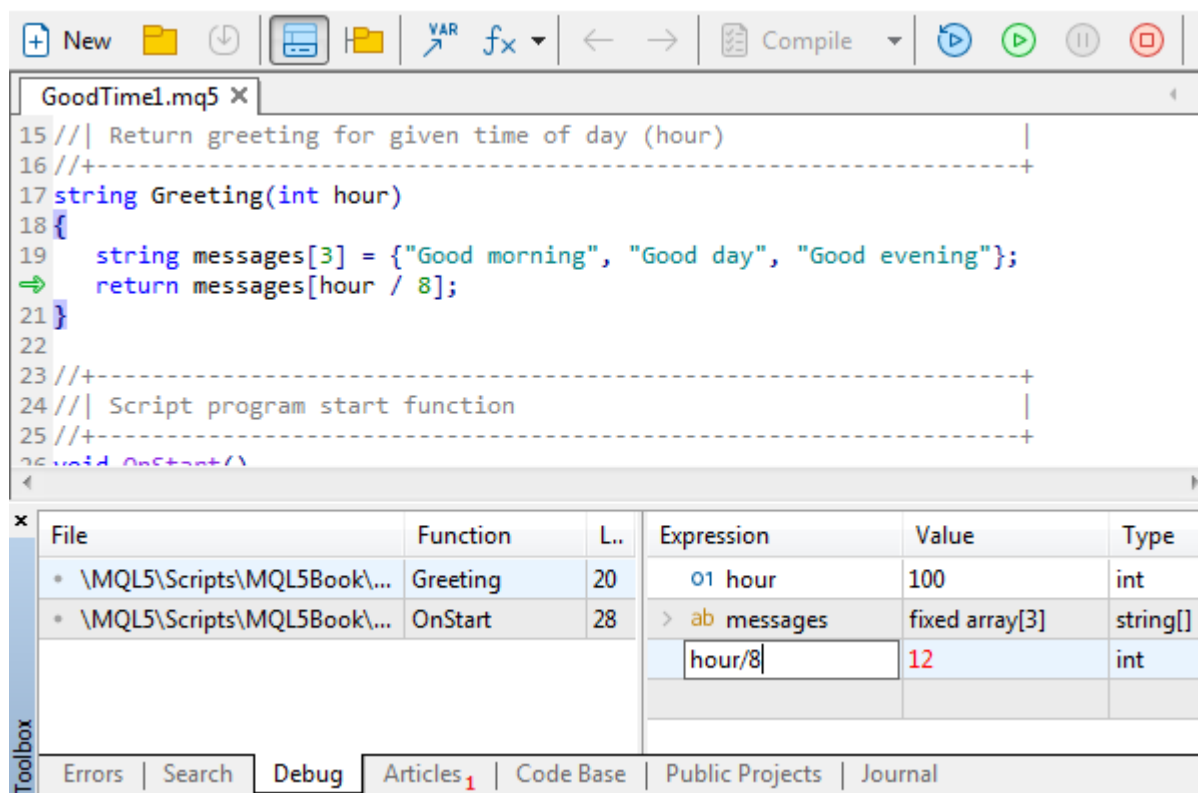
对真实数据和历史数据进行调试的区别在于，前者在在线图表上启动程序，后者在测试程序图表上以可视模式启动程序。要命令编辑器使用什么样的图表和设置，如交易品种、时间范围、日期范围等。您应该先打开 *设置* → *调试* 对话框并填写其中的必填字段。必须启用 *使用指定设置* 选项。如果禁用此选项，则来自 *市场报价* 的第一个交易品种和时间范围 H1 将用于在线调试，而在对历史数据调试时会使用测试程序设置。

请注意，在测试程序中只能调试指标和 EA 交易。脚本只能进行在线调试。

我们按 F5 来运行脚本，并在 *GreetingHour* 参数中输入 100 来重现上述情况。脚本开始执行，同时终端会立即显示一条错误消息并请求打开调试器。

```
Critical error while running script 'GoodTime1 (EURUSD,H1)'.  
Array out of range.  
Continue in debugger?
```

在用户确认后，我们将进入 MetaEditor，当前字符串在源代码中高亮显示，表明这里出现了错误（请注意左边字段中的绿色箭头）。



在发生错误的情况下处于调试模式的 MetaEditor

当前调用栈显示在窗口的左下方：其中列出了所有函数（按自下而上的顺序），这些函数是在代码在当前字符串处停止执行之前调用的。特别要注意，在我们的脚本中，调用了 **OnStart** 函数（由终端本身调用），然后又从中调用了 **Greeting** 函数（我们从代码调用这个函数）。窗口右下方有一个概览面板。可以在其中输入变量名，或者在表达式列中输入整个表达式，可在同一字符串的 *值* 列中查看变量的值。

例如，我们可以使用上下文菜单中的 *添加* 命令，或者用鼠标双击第一个自由字符串来输入表达式 "hour / 8"，并确保它等于 12。

由于调试因错误而停止，也就没有必要继续运行该程序了；因此我们可以执行 *调试* → *停止* 命令 (Shift+F5)。

在较复杂的情况下，问题的根源并不明显，调试器支持对语句执行顺序和变量内容进行逐字符串监控。

为了解决这个问题，必须确保代码中的元素索引始终落在 0 至 2 的范围内，即符合数组大小。严格地说，我们应该编写一些额外的语句来检查输入数据的正确性（在我们的例子中，**GreetingHour** 只能取 0 至 23 范围内的值），在违反条件的情况下，显示一个提示或自动修复。

在这个介绍性项目中，我们只做一些简单的修正：我们将改进计算元素索引的表达式，保证结果始终在要求的范围内。为此，让我们再学习一个运算符：模数运算符，这个运算符只适用于整数。我们使用 '%' 符号来表示模数运算。模数运算的结果是被除数除以除数的整除法的余数。例如：

$$11 \% 5 = 1$$

这里，用 11 除以 5 的得到整数为 2，对应 11 以内 5 的最大倍数 10。11 和 10 之间的余数正好是 1。

要修复 *Greeting* 函数中的错误，只需预先执行 *hour* 除以 24 的模数除法即可，这将确保小时数的范围在 0-23 之间。*Greeting* 函数如下所示：

```
string Greeting(int hour)
{
    string messages[3] = {"Good morning", "Good afternoon", "Good evening"};
    return messages[hour % 24 / 8];
}
```

尽管这种修正效果不错（我们会马上进行检查），但它并未涉及另一个我们暂未关注的问题。这个问题就是 *GreetingHour* 参数是 *int* 类型，也就是说，它可以是正值和负值。例如，如果我们尝试输入 -8 或者更小的负数，那么我们会得到相同的运行时错误，即超出数组范围；在这种情况下，索引没有超出最大值（数组大小），而是小于最小值（特别是，-8 导致引用第 -1 个元素，有趣的是，从 -7 到 -1 的值显示在第 0 个元素上，不会导致任何错误）。

为了修复这个问题，我们将用无符号整数替换 *GreetingHour* 参数的类型：我们将使用 *uint* 代替 *int*（我们将在第二章讲述所有可用的类型，这里我们需要 *uint*）。在 *uint* 编译器级别内置的非负值限制辅助下，MQL5 将自动确保用户（在属性对话框中）和程序（在其计算中）都不会输入负数。

让我们将新版本的脚本另存为 *GoodTime2*，编译并启动它。我们为 *GreetingHour* 参数输入值 100，并确保这次脚本执行时没有任何错误，同时在终端日志中打印问候语 “Good morning”。这种行为在预料之内（也是正确的），因为我们可以使用计算器并检查模除法 100 除以 24 得到的余数是 4，而 4 除以 8 得到的整数结果是 0，在我们的例子中，这意味着早晨。从用户的角度来看，当然可以认为这种行为是非预期的。但是，输入 100 作为小时数本身就是一个非预期的用户操作。用户可能以为我们的程序会失败。但实际并没有，这就很好。当然，对于真实的程序，输入的值必须经过验证，并且必须将错误告知用户。

作为防止输入错误数字的附加措施，我们还将使用一个特殊的 MQL5 特性来为输入参数提供一个更详细、更友好的名称。为此，我们将在同一个字符串中的输入参数说明之后使用注释。例如，像这样：

```
input uint GreetingHour = 0; // Greeting Hour (0-23)
```

请注意，我们已经将变量名中的文字单独写在了注释中（它不再是代码中的标识符，而是代码中给用户的提示）。此外，我们还在圆括号中添加了有效值的范围。启动脚本时，之前的 *GreetingHour* 将出现在对话框中用于输入参数，如下所示：

```
Greeting Hour (0-23)
```

现在我们可以肯定，如果用户输入 100 作为小时，这并不是我们的错误。

细心的读者可能会想，如果我们可以直接在函数中使用输入参数，那么为什么要用 *hour* 参数来定义 *Greeting* 函数并将 *GreetingHour* 发送到函数中。函数作为代码的一个离散逻辑片段，构建函数是为了将程序分成可见的、易于理解的部分，并在之后重用。函数通常从程序的几个部分调用，或者作为库的一部分调用（一个库连接到多个不同程序）。因此，正确编写的函数必须独立于外部上下文，并且可以在程序之间移动。

例如，如果我们需要将 *Greeting* 函数转移到另一个脚本，该函数将停止编译，因为函数中没有 *GreetingHour* 参数。要求添加此参数并不正确，因为其他脚本可以用另一种方式计算时间。换句话说，在编写函数时，我们应该尽量避免不必要的外部依赖项。相反，我们应该声明一些函数参数，这些参数可以用调用代码填充。

1.10 数据输出

在我们的脚本中，只需使用 **Print** 函数将问候语记录到日志中，就可以输出数据。必要时，MQL5 允许将结果保存在文件和数据库中，通过互联网发送，并在图表上显示为图形系列（以指标显示）或对象。

向用户传达一些简单的瞬时信息，而无需用户查看日志（这是一个用于监控程序运行的服务工具，可能在屏幕上隐藏），最简单的方法就是使用 MQL5 API 函数 **Comment**。它的用法几乎与 **Print** 一样。但在执行时，文本不会显示在日志中，而是显示在当前图表的左上角。

例如，将文本脚本中的 **Print** 替换为 **Comment**，我们将得到这样一个 **Greeting** 函数：

```
void OnStart()
{
    Comment(Greeting(GreetingHour), " ", Symbol());
}
```

在终端中启动更改后的脚本后，我们将看到以下结果：

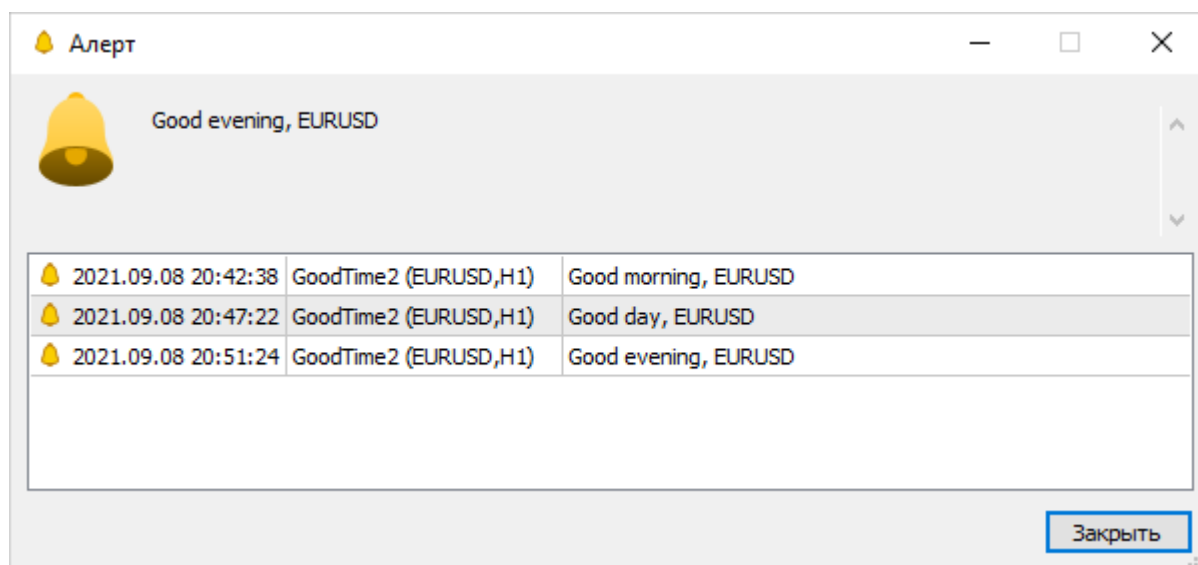


使用 **Comment** 函数在图表上显示文本信息

如果我们既需要为用户显示文本，又需要将他们的注意力转到与新信息相关的环境变化上，那么最好使用 **Alert** 函数。它将通知发送到一个独立的终端窗口，该窗口会在主窗口上弹出，并伴有语音提示。例如，在需要用户干预的交易信号或非常规事件的情况下，这是有用的。

Alert 的语法与 **Print** 和 **Comment** 的语法相同。

下图显示了 **Alert** 函数运行的结果。



使用 Alert 函数显示一条通知

本书不附带具有 **Comment** 和 **Alert** 函数的脚本版本，请读者独立尝试和编辑 *GoodTime2.mq5* 并复制此处提供的截屏。

1.11 格式、缩进和空格

MQL5 是一种所谓的自由格式语言，例如类 C 语言和多种其他语言。这意味着，只要遵循语法规则，则可以服务符号（如括号或运算符）和关键字可以随机放置。语法只限制这些符号和字的相互顺序，而每个字符串开头的缩进大小或语句元素之间的空格数对编译器没有任何意义。在文本中需插入空格来分隔语言元素（比如变量类型关键字和变量标识符）的位置，都可以使用大量的空格。此外，除了空格，还允许使用其他表示空白的符号，比如制表符和换行符。

如果在语句的一些元素之间有一个分隔符号（我们将在第二章学习），比如在函数参数之间有一个逗号 ‘,’，则无需使用任何空格。

对源代码格式的更改不会修改可执行代码。

总的来说，有许多非自由格式的语言。在某些非自由格式的语言中，代码块的形成（在 MQL5 中使用花括号匹配来实现）依赖于与左边缘的等距缩进。

由于格式自由，MQL5 允许程序员使用多种不同的技术来形成源代码，以提高其可读性、可视性和内部导航便利性。

让我们分析一些例子，看看如何从我们的脚本中记录 **Greeting** 函数的源文本，而不改变它的意图。

这是最“紧凑”的版本，没有任何多余的空格或换行符（此处添加的表示换行符的符号 ‘\’ 仅为遵守本书对发布源代码的限制）。

```
string Greeting(int hour){string messages[3]={"Good morning",\
"Good afternoon","Good evening"};return messages[hour%24/8];}
```

此处的版本中插入了过多的空格和换行符。

```

string
Greeting ( int hour )
{
    string messages [ 3 ]
        = {
            "Good morning" ,
            "Good afternoon" ,
            "Good evening"
        } ;

    return messages [ hour % 24 / 8 ] ;
}

```

MetaEditor 具有一个内置的代码样式器，可按照支持的一种样式，自动设置当前文件的源代码格式。可以在 **工具 → 设置 → 样式器** 对话框中选择特定的样式。使用 **工具 → 样式器** 命令应用样式。

应谨记，您的空间自由是有限的。特别是，您不能在标识符、关键字或数字中插入空格。否则，编译器将无法识别它们。例如，如果我们在数字 24 中的数字 2 和 4 之间插入一个空格，编译器尝试编译脚本时会发生大量错误。

以下是一个被故意错误修改的字符串：

```
return messages[hour % 2 4 / 8];
```

以下是错误日志：

```

'GoodTime2.mq5'      GoodTime2.mq5 1      1
'4' - some operator expected      GoodTime2.mq5 19      28
 '[' - unbalanced left parenthesis GoodTime2.mq5 19      18
'8' - some operator expected      GoodTime2.mq5 19      32
']' - semicolon expected      GoodTime2.mq5 19      33
']' - unexpected token      GoodTime2.mq5 19      33
5 errors, 0 warnings      6      1

```

编译器消息未必总是清晰易懂。需要注意的是，即使出现第一个（连续）错误时，程序的内部表示（如编译器中途对程序的理解）很可能与程序员的意图大相径庭。特别是，在这种情况下，只有前两条错误包含理解问题的关键，而所有其他错误都属于连带错误。

根据第一个错误，编译器期望找到 2 和 4 之间的运算符（因为它认为 2 和 4 是两个不同的数字，而不是由空格分隔的数字 24）。另一种逻辑是，这里省略了一个右方括号，因此编译器显示了第二个错误：“[- unbalanced left parenthesis.”运行后，整个表达式被完全打乱，因此编译器认为后面的数字 8 和右括号 ‘]’ 似乎不合适。但实际上，我们只用删除 2 和 4 之间多余的空格，情况就会变得正常。

当然，在我们刻意制造问题的情况下，进行这样的错误分析十分容易。而在实践中，我们不一定清楚如何修复各种错误。即使在上面的例子中，假设您从另一个程序员那里收到了这段不完整的代码，并且数组元素不包含此类琐碎信息，那么您会考虑其他修改方法：必须留下 2 或 4，因为作者可能希望用一个数字替换另一个数字，而没有清除“足迹”。

1.12 小结

在第一章中，我们熟悉了 MetaEditor 框架，使用 MQL 母版创建了一个脚本模板，并逐步用代码填充脚本来解决简单的问题。为此，我们运用了 MQL5 的一些基本原则和语法结构。随后我们在实践中尝试了调试器，修复了一些问题，让程序运行稳定。

我们的脚本样本的发展如下：



在本书的后续章节中，我们将详细探讨 MQL5 的这些特性以及多种其他特性，并探讨编程的技术方面及其在交易中的应用。

第二章 MQL5 编程基础

像任何其他编程语言一样，MQL5 也是基于一些基本概念，这些概念用于创建更复杂的结构，最终创建整个程序。在本章中，我们将学习大部分的概念，如数据类型、标识符、变量、表达式和运算符，以及在代码中组合各种语句以构建所需程序运行逻辑的技巧。

这些资料帮助读者们深入到过程化编程的独立实际应用中，而这是解决各类问题的最早期编程趋势之一。事实上，它是由一些小步骤（语句）构成的程序，这些小家伙（语句）按照所要求顺序执行，以进行数据处理。本书第一章展示的文本脚本就是这种风格的一个例子。

本节涵盖了广泛的基本概念和工具，它们对于熟练掌握 MQL5 编程至关重要，包括以下小节：

标识符：

①标识符构成了程序代码的基础。本小节讨论 MQL5 中标识符命名的目的和规则。

内置数据类型：

①MQL5 包括了各种内置数据类型，每种数据类型都被设计用来存储和处理特定类型的信息。本节对基本数据类型进行了综合介绍。

变量：

变量用于存储和管理程序中的数据。“变量”一节介绍了变量的应用基础知识，并讨论了如何对变量进行声明、初始化和赋值。

数组：

①数组提供了一种结构化的数据存储方式。本节涵盖了在 MQL5 中创建和使用数组的基础知识。

表达式：

①表达式构成了计算和程序逻辑的基础。在本小节中，您将学习如何在 MQL5 中构造和计算表达式。

类型转换：

①数据类型转换是编程不可或缺的一部分。“类型转换”一节介绍了 MQL5 中不同类型数据之间转换的相关过程。

语句：

①语句是控制程序执行的命令。在本节中，我们将研究各种类型的语句及其应用。

函数：

①函数可以实现对代码进行结构化和重用。本节深入探讨在 MQL5 中创建和调用函数的基础知识。

预处理器：

①MQL5 预处理器负责在编译前处理源代码。“预处理器”一节介绍了预处理器指令的应用原则及其对代码的影响。

我们后续学习大量范例时，将以过程化编程原则为基础，即面向对象的编程（OOP）。这将在第三章中提及。

 面向交易员的 MQL5 编程指南 - 本书中的源代码第二章

 本书中的例子也可以在 `公共项目 \MQL5\Shared Projects\MQL5Book` 中找到。

2.1 标识符

我们会发现，程序是由多个元素组成的，因此必须用唯一的名称来引用这些元素以避免混淆。这些名称就称为标识符。

标识符是依据一定规则而创建的词语：标识符只能包含拉丁字符、下划线字符（`_`）和数字，并且第一个字符不能是数字。字母可以是小写字母，也可以是大写字母。

标识符的最大长度是 63 个字符。标识符不能与 MQL5 的服务字（例如类型名）同名。可以在“帮助”中找到服务字的完整列表。违反任何一条标识符构成规则都会导致编译错误。

以下是一些正确的标识符：

```
i           // single character
abc         // lower-case letters
ABC         // upper-case letters
Abc         // mixed-case letters
_abc        // underscore at the beginning
_a_b_c_     // underscore anywhere
step1       // digit
_1step      // underscore and digit
```

我们已经在 `HelloChart` 脚本中看到标识符是如何被用作变量和函数名称的。

建议为标识符指定有意义的名称，直观反映相关元素的目的和内容。在某些情况下，会使用单字符标识符，我们将在 [循环结构](#) 的相关章节中讨论这类标识符。

编写标识符有以下几种常见方法。例如，如果我们为一个用于存储利润因子的变量选择一个名称，那么选择以下方法比较好：

```
ProfitFactor // "camel" style, all words start with a capital letter
profitFactor // "camel" style, all words but the first one start with a capital let
profit_factor // "snake" style, the underscore is put between all words
```

在许多编程语言中，命名不同实体时会采用不同样式。例如，可以遵循这样的做法：变量名只以小写字母开头，而类名（参见 [第三章](#)）以大写字母开头。遵循这些做法，无论是程序员在小组工作时分析源代码，还是在一大段间隔之后再次处理自己的代码片段，都会有所帮助。

除了以上这些，还有其他样式，其中一些样式用于特殊情况：

```
profitfactor // "smooth" style, all letters are lower-case
PROFITFACTOR // "smooth" style, all letters are upper-case
PROFIT_FACTOR // "macro" style, all letters are upper-case with underscores between
```

全大写格式有时用于 [常量名称](#)。

“宏”样式通常用于 [预处理器](#) 宏描述的名称。

2.2 内置数据类型

数据类型是我们在日常生活中经常使用的一个基本概念，我们甚至从未意识到它的存在。基于我们交换信息的含义以及能够对信息实行的处理程序，数据类型通常是隐藏的。例如，管理我们的家庭资产时，我们对数字进行加减，分别表示收入和支出。在这里，“数字”描述了一种类型，对于这种类型，我们充分认识到它的可能值以及这些值的算术运算。在交易背景中，MetaTrader 5 中有一个类似的值，即当前账户余额；因此，MQL5 提供了一种创建和操作数字的机制。

与数字不同，文本信息（如交易金融工具的名称）遵循其他规则。在这里，我们用字母组成单词或者用单词组成句子，但无法计算若干行的累计总和或算术平均值。因此，“行”（或称为“字符串”）是数值类型之外的另一种数据类型。

除了其用途和每种类型特有的一组典型操作之外，还有一个重要的属性来区分不同的类型。那就是它们的大小。例如，一年中的星期数不超过 52，而从年初开始经过的秒数却是一个天文数字。因此，为了在计算机内存中有效存储和处理这些不同的值，可以划分出不同大小的内存段。这样我们就会明白，“数字”的广义概念实际上隐喻不同的类型。

MQL5 允许使用一些数字类型，这些类型在内存单元分配大小以及一些附加特性方面各有不同。具体来说，一些数字可以取负值，如以点数为单位的浮动盈利，而另一些数字不能取负值，如账号。此外，一些值不能有小数部分，因此，相比用随机的“带小数点的数字”表示，用更严格的“整数”类型表示更具成本效益。例如，账户余额或交易金融工具的价格通常是带小数点的值。而历史订单数或账号始终是一个整数。

MQL5 支持一组通用类型，这些类型与大多数编程语言中可用的类型相似。这组类型包括整数类型（不同大小）、两种不同精度的实数（带小数点）、字符串和单个字符以及逻辑类型，逻辑类型只有两种可能值：**true** 和 **false**。此外，MQL5 还提供了自己的特定类型，用于处理时间和颜色。

出于内容完整性，我们需要说明一下，MQL5 允许通过在代码中声明应用类型（如结构、类和其他 OOP 的典型实体）来扩展类型集合；但这些我们将在后续章节中介绍。

由于存储值的内存单元大小是一个重要类型属性，我们来了解一下内存方法论。

计算机内存的最小单位是字节。也就是说，程序可以为单个值分配的最小内存单元大小是一个字节。一个字节由 8 个较小的颗粒（也就是“位”）组成，每个位可以有两种状态：启用（1）或禁用（0）。所有现代计算机在底层都使用这种位，因为用二进制表示信息非常便于在硬件中实现（比如在随机存取存储器中实现，在处理器中实现，或者通过网络电缆或 WiFi 传输数据时实现）。

在内存单元中对位状态的不同解释，确保了对不同类型值的处理。这通常由编译器进行处理。虽然程序员通常不需要深入到位级别的操作；但该语言提供了实现这种操作的工具（参见[按位运算](#)）。

MQL5 中有专门的保留字来描述数据类型。我们已经在第一章中了解了一些数据类型，比如 **void**、**int** 和 **string**。下文给出了一个完整的类型列表，每种类型都附带一个简要说明和以字节表示的大小。

根据用途，这些类型可以有条件地分为数字编码和字符编码数据（在相关列中标记），以及其他专用类型，如字符串、逻辑（或布尔）、日期/时间和颜色。其中 **void** 类型很特殊，表示没有任何值。除了标量类型，MQL5 为复数、矩阵和向量操作提供了对象类型：**complex**、**vector** 和 **matrix**。这些类型用于解决线性代数、数学建模、机器学习和其他领域的各种问题。我们将在本书的第四章深入学习。

类型	大小 (字节)	数字	字符	注释
char	1	+	+	单字节字符或有符号整数
uchar	1	+	+	单字节字符或无符号整数
short	2	+	+	双字节字符或有符号整数
ushort	2	+	+	双字节字符或无符号整数
int	4	+		有符号整数
uint	4	+		无符号整数
long	8	+		有符号整数
ulong	8	+		无符号整数
float	4	+		有符号浮点数
double	8	+		有符号浮点数
enum	4	(int)		枚举
datetime	8	(ulong)		日期和时间
color	4	(uint)		颜色
bool	1	(uchar)		逻辑
string	10 个字节 + 可变部分			字符串
void	0			Void
complex	16	+		具有两个双精度类型字段的结构
vector	向量长度 x 类型大小	+		实数或复数类型的一维数组
matrix	x 行 x 列类型大小	+		实数或复数类型的二维数组

根据大小，数值类型中可以存储不同的值范围。除此之外，对于相同大小的整数和浮点数，值范围可能会有很大的不同，因为它们使用了不同的内部表示。所有这些错综复杂的关系将在涉及特定类型的相关章节中介绍。

程序员可以根据预期值、效率考量或经济因素选择数值类型。特别是，类型越小，在内存中可存储该类型的值就越多，而整数类型的处理速度要快于浮点数类型。

请注意，数字编码和字符编码在类型上部分交叉。原因是字符在内存中以整数形式存储，即相关字符表中的代码：ANSI用于单字节字符编码，Unicode用于双字节字符编码。ANSI是一个以协会（美国国家标准协会）名称命名的编码标准，而Unicode，您猜对了，就是统一码（字符集）的意思。MQL5中使用Unicode字符来构成字符串（**string**类型）。在将程序与外部数据源（如来自互联网的数据源）集成时，通常需要单字节字符。

如上所述，数值类型可以分为整数类型和浮点数类型。让我们更详细地展开讨论。

2.2.1 整数

整数类型用于存储不带小数点的数字。如果值的应用意义与分数无关，就应选用整数类型。例如，图表上的条数或未平仓头寸数始终是整数。

在 MQL5 中，可以使用 *char*、*short*、*int*、*long* 等关键字，选择大小为 1 至 8 个字节的整数类型。整数均为有符号类型，即可以是正值和负值。如有必要，可以将相同大小的整数类型声明为无符号类型（名称都以 'u' 开头，表示“无符号”）：*uchar*、*ushort*、*uint*、*ulong*。

下表根据类型大小以及有符号还是无符号显示了可能值的范围。

类型	最小值	最大值
char	-128	127
uchar	0	255
short	-32768	32767
ushort	0	65535
int	-2147483648	2147483647
uint	0	4294967295
long	-9223372036854775808	9223372036854775807
ulong	0	18446744073709551615

我们没有必要记住上述每种整数类型的极限值。MQL5 中有许多预定义的命名常量，可在代码中用于替代魔术数字（包括最小/最大整数）。这种技术将在涉及 [预处理器](#) 的相关章节中讨论。这里，我们只列出相关的命名常量：CHAR_MIN、CHAR_MAX、UCHAR_MAX、SHORT_MIN、SHORT_MAX、USHORT_MAX、INT_MIN、INT_MAX、UINT_MAX、LONG_MIN、LONG_MAX 和 ULONG_MAX。

我们说明一下获得这些值的方法。为此我们需要回顾一下位和字节的概念。

一个字节包含 8 个位，每个位有“启用”和“禁用”这两种状态，所有可能组合数量为 256。因此，一个字节中可以存储的值的范围是 0 至 255。但是，对它们的编译取决于为此字节分配的类型。编译器会根据程序员的语句，确保不同编译方式都能实现。

一个字节中的低阶（最右边）位为 1，第二位为 2，第三位为 4，以此类推，直到高阶位 128。显而易见，这些数字等于二的位数次方的幂（从 0 开始计数）。以下是二进制的图解。

位	高阶				低阶			
数字	7	6	5	4	3	2	1	0
值	128	64	32	16	8	4	2	1

当所有位都被置位（设置为 1）时，得到二的所有幂次之和，即 255，这是一个字节的最大值。如果所有位都被重置（设置为 0），我们得到零。如果低阶位启用（设置为 1），这个数字是一个奇数。

在编码有符号数时，高阶位用于标记负值。因此，对于正数值范围内的单字节整数，最大值是 127。对于负数值，则有 128 种可能的组合，即最小值是 -128。当一个字节的所有位都被置位（设置为 1）时，编译器将其解释为 -1。如果在这样的数字中将低阶位重置（设置为 0），我们得到 -2，以此类推。如果只有高阶位（符号位）被置位（设置为 1），其他所有位为 0，我们得到 -128。

这种看似不合理的编码叫做“补码”。这样便可以在硬件层实现有符号和无符号数的统一计算。此外，补码还可以防止值丢失，在正负区域进行相同编码时，通常会发生值丢失的情况：此时我们将得到两个表示零的值，即正 0 和负 0。而这会带来歧义。

如果数字有多个字节（如 2、4 或 8 个），所有位连续进行编号，各自的值有规律变化。在所有情况下，判定数字为负数的条件是高阶字节的高阶位被置位（设置为 1）。

因此，我们可以使用一个字节来存储一个 0 至 255 范围内的无符号整数（`uchar`，即无符号字符的缩写）。我们也可以将一个有符号整数写入字节（为此，我们将把它的类型描述为 `char`）。在这种情况下，编译器会将 256 个组合的可用数在正负值之间平均分配，并将其显示在 -128 至 127 的区域中（第 256 个值为零）。显而易见，将在有符号和无符号字节的位级对值 0 至 127 进行相同编码。然而，大的绝对值（从 128 开始）将转换为负数（根据上文插入部分描述的方案）。这种“转换”仅在读取或执行与存储值相关的操作时才会发生，此时数据内部表示形式（位状态）不变。

我们将在涉及 [类型强制转换](#) 的相关章节中更深入讨论这个问题。

与单字节整数类似，很容易计算出 2 个字节的位组合数为 65536。因此，这类范围是针对有符号和无符号双字节整数 `short` 和 `ushort` 形成的。其他类型由于字节大小增加，因此可以存储更大的值。

请注意，使用相同大小的无符号类型可以将正值范围加倍。存储可能非常大的数量时（此时不会出现负值），则有必要进行这种操作。例如，MetaTrader 5 中的订单号是 `ulong` 类型的值。

我们已经在 [第一章](#) 中遇到过整数说明示例。特别是，`uint` 类型的输入参数 `GreetingHour` 也在此处定义：

```
input uint GreetingHour = 0;
```

除了附加的 `input` 关键字（使变量在 MQL 程序的参数列表中可见），其他组成部分（包括类型、名称和 '=' 符号后的可选初始化）是所有变量固有的属性。

我们将在 [变量](#) 一节中详细讨论变量说明语法。至此，请注意表示整型常量的方法。在描述变量时，常量可以被指定为默认值（在上面的例子中，是 0）。此外，常量可以用于 [表达式](#) 中，例如，用于公式事件中。

需注意，插入源代码中的任何类型的常量被称为字面量（字面意思：“逐字照搬”）。之所以称为字面量是因为它们“按原样”被引入程序中，并且在说明的位置立即使用。字面量与语言的其他元素（尤其是变量）有很大区别，字面量既没有名称，也不能从程序的其他位置引用。

对于负数，数字前必须有负号 '-'；而对于正数，则可以省略正号 '+'，即 +100 和 100 格式没有区别。

应该注意的是，数字值通常按我们惯用的十进制记数法记录在源代码中。但是，MQL5 允许使用另一种计数法，即十六进制计数法。这种方法对于处理位级信息（参见 [按位运算](#)）非常方便。

十进制常量的所有数位都允许使用从 0 到 9 的数字，而对于十六进制常量，除了数字之外，还会额外使用字符 **A** 到 **F** 或 **a** 到 **f** (即不区分大小写)。“十六进制数字”**A** 对应于十进制记数法的数字 10, **B** 对应 11, **C** 对应 12, 以此类推, 直至 **F** 对应 15。

十六进制常量的一个显著特征是它以前缀 0x 或 0X 开头, 后面跟数字的有效数位。例如, 在十六进制系统中, 数字 1 表示为 0x1, 而 16 表示为 0x10 (因为 16 大于 15, 即 0xF, 所以需要额外的高位)。十进制的 255 转为十六进制就是 0xFF。

让我们再举一些例子来说明使用整数类型描述变量的各种情况 (附在 `SQL5/Scripts/SQL5Book/p2/TypeInt.mq5` 脚本中):

```
void OnStart()
{
    int x = -10;           // ok, signed integer x = -10
    uint y = -1;          // ok, but unsigned integer y = 4294967295
    int z = 1.23;          // warning: truncation of constant value, z = 1
    short h = 0x1000;      // ok, h = 4096 in decimal
    long p = 10000000000;  // ok
    int w = 10000000000;   // warning, truncation..., w = 1410065408
}
```

x 变量由允许的负值 -10 正确初始化。

y 变量无符号。因此, 尝试表示一个负值, 我们会得到一个有趣的结果。数字 -1 采用位表示形式, 程序会根据无符号类型 **uint** 来解释。这样得到数字 4294967295 (它实际上等于 `UINT_MAX`)。

z 变量被赋予浮点数值 1.23 (将在下一节讨论相关内容), 这时编译器警告小数部分被截断。结果是, 整数 1 存入变量中。

h 变量由十六进制形式的常量 (0x1000 = 4096) 成功初始化。

大数值 10000000000 记录在 **p** 和 **w** 变量中, 前者是一个长整数类型 (**long**) 且成功处理, 而后者是一个普通类型 (**int**), 因此引发了编译器警告。由于该常量超过了 **int** 的最大值, 因此编译器会截断多余的高阶数字 (位), 这导致 1410065408 存入 **w**。

这一行为在类型转换开发过程中可能造成的潜在负面问题之一, 而程序员不一定会表明这一点。在后面的例子中, 就存在一个潜在错误。很明显, 在这个特殊的例子中, 故意选择了错误的值来演示警告。而在实际的程序中, 我们不一定一眼就能看出程序会将哪些值存入整数变量。因此, 您应该非常仔细地查看编译器警告, 并更改类型或明确指定所需的类型强制转换, 以尝试消除这些警告。这将在涉及 [类型强制转换](#) 的相关章节中讨论这一问题。

对于整数类型, 定义了算术运算、位运算和其他类型的运算 (参见 [表达式](#) 一章)。

2.2.2 浮点数

在日常生活中, 我们使用带小数点的数 (即实数) 的频率与使用整数的频率一样高。“实数”这个名称本身表明, 使用这些数, 您可以表达现实世界的一些有形的事物, 如重量、长度、体温, 也就是说, 这些事物可以用非整数单位数量来度量, 但要“带点零头”。

我们在交易中也经常用到实数。例如, 它们用于表示交易订单中的交易品种价格或交易量 (通常允许整手交易的小数部分)。

SQL5 中提供了 2 种实数类型: 表示单精度的 **float** 和表示双精度的 **double**。

在源代码中，*float* 和 *double* 类型的常量值通常记录为整数和小数部分（每个部分都是一个数字序列），由字符 `.` 分隔，如 `1.23` 或 `-789.01`。可以没有整数或小数（但不能都没有），但必须有点。例如，`.123` 表示 `0.123`，而 `123.` 表示 `123.0`。而 `123` 将创建一个整数类型常量。

然而，还有另一种记录实数常量的形式，即指数形式。其中，整数和小数部分后跟 `E` 或 `e`（不分大小写），整数表示以 `10` 为底的幂数，即附加因子。例如，以下几种表示法以指数形式表示同一个数 `0.57`：

```
.0057e2
0.057e1
.057e1
57e-2
```

记录实数常量时，后面的几个字节默认定义为 *double* 类型（占用 `8` 个字节）。要设置 *float* 类型，需要在常量右侧添加 `F` 或 `f` 后缀。

float 和 *double* 类型在大小、值范围和数字表示精度方面有区别。所有这些如下表所示。

类型	大小（字节）	最小值	最大值	精度（位数）
float	4	$\pm 1.18 \times 10^{-38}$	$\pm 3.4 \times 10^{38}$	6 至 9，通常是 7
double	8	$\pm 2.23 \times 10^{-308}$	$\pm 1.80 \times 10^{308}$	15 至 18，通常是 16

值范围以绝对值形式显示：最小值和最大值决定了正负区域中允许值的范围幅度。与整数类型类似，这些极限值也有内嵌的命名常量：`FLT_MIN`、`FLT_MAX`、`DBL_MIN`、`DBL_MAX`。

请注意，实数都是有符号的，也就是说，不存在无符号的实数。

精度是指相关类型的实数能够不失真地存储的有效位数（十进制位数）。

的确，实数类型数字的精度要低于整数类型。这是为了获得普遍性和更广泛的可能值而付出的代价。例如，如果无符号 4 字节整数 (*uint*) 的最高值为 `4294967295`，即大约 4 百万或 4.29×10^9 那么 4 字节实数 (*float*) 的最大值为 3.4×10^{38} ，后者比前者高了 29 个数量级。对于 8 字节类型，这种差异更加明显：*ulong* 可以容纳 `18446744073709551615` (18.44×10^{18} 或大约 18 个一百亿亿)，而 *double* 可以容纳 1.80×10^{308} ，后者比前者高 289 个数量级。插叙内容提供了关于精度的更多细节。

尾数和指数

实数在内存中的内部表示（以分配的字节为单位）十分复杂。高阶位被用作负号标志（我们在整数类型中也看到过）。所有其他位被分成两组。较大一组包含数字的尾数，即有效位（我们指的是二进制位）。较小一组存储以 `10` 为底的幂数（指数），通过将幂数与尾数相乘得到存储的数。特别是，*float* 类型尾数大小为 24 位 (`FLT_MANT_DIG`)，而 *double* 类型尾数大小为 53 位 (`DBL_MANT_DIG`)。就传统的十进制位而言，我们将得到与上表所示相同的精度：*float* 的最低有效位数为 6 (`FLT_DIG`)，而 *double* 的最低有效位数为 15 (`DBL_DIG`)。但某些特定数字会有“幸运”的位组合，对应着更大的十进制位数。*float* 和 *double* 的参数大小分别为 8 位和 11 位。

由于指数的原因，实数的值范围要大得多。同时，随着指数的增加，尾数低位的“比重”也在增加。这意味着，可在计算机内存中表示的两个相邻实数之间有很大区别。例如，对于数字 `1.0`，低位的“比重”在 *float* 情况下是 `1.192092896 e 07` (`FLT_EPSILON`)，而在 *double* 情况下是

2.2204460492503131e-016 (DBL_EPSILON)。也就是说,如果一个数小于 $1.192092896 \times 10^{-7}$, 那么编译器无法区分这个数字与 1.0 的区别。这好像不重要或者“没什么大不了的”,但是对于更大的数而言,这种不确定区域会变大。如果存储了一个 *float* 类型的数字,大小约为 10 亿 (1×10^9) 那么最后 2 位将无法安全存储,也无法从内存中恢复 (参见下文的代码范例)。但是,根本问题不在于数的绝对值,而在于其中位数的最大值,要无损失的召回。同样,我们可以尝试将一个数表示为 *float* 数 1234.56789 (在结构上很像金融工具的价格);由于其内部表示缺乏精度,它的最后两位会“浮动”。

如果表示为 *double*, 类似的情况会在处理更大的数字 (或更多有效数字) 时才会显现,但仍然可能发生,且并不罕见。当运算非常大或非常小的实数时,您应该考虑这一点,并在编写程序时额外检查一下,以防止潜在的精度损失。特别要注意的是,应该以一种特殊方式将实数与零进行比较。我们将在 [比较运算符](#) 一节中讨论。

细心的读者会发现,上面指定的尾数和指数的大小可能有错误。我们以 *float* 为例进行解释。它存储在大小为 4 字节的内存单元中,即占用 32 位。同时,尾数大小为 24,指数大小为 8,加起来已经是 32 位。那么符号位在哪里? 关键就在于 IT 专业人员以“规范化”的形式存储尾数。如果我们回顾一下记录普通十进制数的指数形式,就会容易理解这个问题。例如,数字 123.0 可以表示为 1.23E2、12.3E1 或 0.123E3。一种表示法被认为是规范化形式,其中小数点之前只有一个有效位 (即,非零)。对于 123.0 这个数字,可以表示为 1.23E2。根据定义,第 1 到 9 位被认为是十进制记数法中的有效位。顺理成章的,我们进入二进制记数法。其中只有一个有效位,1。似乎二进制记数法中的规范化形式总是从 1 开始,并且可以被省略 (无须占用内存)。以这种方式,尾数可以节省一位。事实上,它包含 23 位 (多出的一个高阶单位是隐式的,在重构数字并从内存中检索时会自动添加)。将尾数减少 1 位可以为有符号位腾出空间。

在应使用浮点类型时,我们主要选择 *double* 类型,这样会更精确。仅在需要节省内存时才会用 *float* 类型,例如处理非常大的数据数组。

MQL5/Scripts/MQL5Book/p2/TypeFloat.mq5 脚本中显示了一些实数类型常量的应用示例。

```
void OnStart()
{
    double a0 = 123;           // ok, a0 = 123.0
    double a1 = 123.0;         // ok, a1 = 123.0
    double a2 = 0.123E3;       // ok, a2 = 123.0
    double a3 = 12300E-2;      // ok, a3 = 123.0
    double b = -.75;           // ok, b = -0.75
    double q = LONG_MAX;       // warning: truncation, q = 9.223372036854776e+18
                                // LONG_MAX = 9223372036854775807
    double d = 9007199254740992; // ok, maximal stable long in double
    double z = 0.12345678901234567890123456789; // ok, but truncated
                                // to 16 digits: z = 0.1234567890123457
    double y1 = 1234.56789;     // ok, y1 = 1234.56789
    double y2 = 1234.56789f;    // accuracy loss, y2 = 1234.56787109375
    float m = 1000000000.0;     // ok, stored as is
    float n = 999999975.0;      // warning: truncation, n = 1000000000.0
}
```

a0、*a1*、*a2* 和 *a3* 变量包含用不同方法编写的同一个数 (123.0)。

在 *b* 变量的常量中,小数点之前省略了无意义的零。此外,这里演示了使用如何使用负号 ‘-’ 记录一个负数。

尝试在 `q` 变量中存储最大整数。此时，编译器给出一个警告，因为 `double` 无法精确地表示 `LONG_MAX`：实际存储的是 9223372036854776000 而不是 9223372036854775807。这明显表明，虽然 `double` 的值范围远远超过整数的值范围，但这是通过丢失低阶位而实现的。

作为对比，`double` 类型能够无失真存储的最大整数被赋予 `d` 变量的值。如果对整数序列使用 `double` 类型，序列中会间歇性跳过某些数字。

`z` 变量再次提醒我们关于最大有效位数 (16) 的局限性：超过该限制的常量会被截断。

在 `y1` 和 `y2` 变量中，当同一个数以不同格式 (`double` 和 `float`) 记录时，我们就会看到因转换为 `float` 而损失的精度。

实际上，`m` 和 `n` 变量将被编译器视为相等，因为 999999975.0 以内部表示存储时，就变成了 1000000000.0。

数字类型常用于公式计算；而且为数字类型定义了一组广泛的运算 (参见 [表达式](#))。

这类计算有时会产生不正确的结果，也就是说，它们无法表示为数字。例如，无法定义负数的平方根或零的对数。在这类情况下，实数类型可以存储一个名为 NaN (Not A Number, 非数字) 的特殊值。事实上，这类值有几种不同的类型，例如，区分正无穷大和负无穷大。MQL5 提供了一个特殊函数，`MathIsValidNumber`，该函数检查 `double` 值是数字还是 NaN 值。

2.2.3 字符类型

字符数据类型用于存储构成字符串的特定字符 (字母) (参见 [字符串](#))。MQL5 有 4 种字符类型：两个大小为 1 字节的类型 (`char`、`uchar`) 和两个大小为 2 字节的类型 (`short`、`ushort`)。以 `u` 为前缀的类型是无符号的。

事实上，字符类型是整数类型，因为字符类型实际存储的是相关表中字符的整数代码：`char` 类型对应 ASCII 字符表 (代码 0 至 127)；`uchar` 类型对应扩展 ASCII 表 (代码 0 至 255)；`short/ushort` 类型对应 Unicode 表 (无符号版本最多有 65535 个字符)。顺带说下，ASCII 是美国信息交换标准代码 (American Standard Code for Information Interchange) 的缩写。

对于 MQL5 字符串，使用 2 字节字符 `ushort`。传输随机数据数组时，通常会使用 1 字节 `uchar` 类型，这些 [数组](#) 会根据所应用的协议进行打包和解包，例如用于连接到加密平台。

字符常量被记录为用单引号括起来的字母。但是，您也可以使用上述整数记数法 (参见 [整数](#))。同时，整数必须在 1 字节或 2 字节格式的值范围内。

此外，我们可以使用转义序列表示法。转义序列使用反斜杠 (`\`) 作为第一个字符，后跟一个预定义的控制字符和/或数字代码。MQL5 支持以下转义序列：

- `\n` -换行
- `\r` -回车
- `\t` -制表
- `\\` -反斜杠
- `\"` -双引号
- `\'` -单引号
- `\X` 或 `\x` -前缀，用于随后指定十六进制格式的数字代码
- `\0` -前缀，用于随后指定八进制格式的数字代码

`MQL5/Scripts/MQL5Book/p2/TypeChar.mq5` 脚本中给出了使用字符类型常量的基本方法。

```

void OnStart()
{
    char a1 = 'a'; // ok, a1 = 97, English letter 'a' code
    char a2 = 97;  // ok, a2 = 'a' as well
    char b = '£';  // warning: truncation of constant value, b = -93
    uchar c = '£'; // ok, c = 163, pound symbol code
    short d = '£'; // ok
    short z = '\0'; // ok, 0
    short t = '\t'; // ok, 9
    short s1 = '\x5c'; // ok, backslash code 92
    short s2 = '\\';  // ok, backslash as is, code 92 as well
    short s3 = '\0134'; // ok, backslash code in octal form
}

```

a1 和 *a2* 变量以两种不同的方式获得字符 'a' (英文字母) 的值。

尝试将 '£' 记录在 *b* 变量中, 但是其代码 163 超出了 *char* 的范围 (127); 因此被转换为有符号数 -93 (编译器给出一个警告)。它后面的 *uchar*(c) 和 *short* (d) 类型的变量认为这个代码是正常的。

其他变量使用转义序列初始化。

可对字符采取的处理操作与整数相同 (参见 [表达式](#))。

2.2.4 字符串类型

字符串类型用于存储基于文本的信息, 由 *string* 关键字表示。字符串是 *ushort* 字符的序列, 支持完整的 Unicode 范围, 包括多国文字。例如, 金融工具的名称和交易订单中的注释都是字符串。

由于字符串的特殊性质, 它们的大小是一个变量值, 等于文本长度的两倍 (字符数量乘以字符的“宽度”, 即 2 个字节) 再加上一个字符。这个附加字符存储“终止零” (编码为 0 的字符), 表示行结尾。此外, MQL5 使用一些空间来存储服务信息, 即字符串在内存中开始位置的引用。

与 C++ 不同, 在 MQL5 中无法获得字符串或任何其他变量的地址。MQL5 中禁止直接内存访问。

字符串字面量在源代码中记录为以双引号括起的字符序列。例如: "EURUSD" 或 "\$"。我们应该区分由一个字符组成的字符串 (如 "\$") 和相同的单个字符 (如 '\$')。它们是不同的数据类型。

空字符串表示为 ""。至于隐式终止零, 除了服务信息之外, 还占用 2 个字节。

如果有必要在字符串中使用双引号字符, 则必须在它前面加上反斜杠字符, 转换成控制序列, 如 "Press \"OK\""

MQL5/Scripts/MQL5Book/p2/TypeString.mq5 脚本中给出了字符串初始化示例。


```

void OnStart()
{
    string h = "Hello";           // Hello
    string b = "Press \"OK\"";    // Press "OK"
    string z = "";                //
    string t = "New\nLine";       // New
                                   // Line
    string n = "123";             // 123, text (not an integer value)
    string m = "very long message "
               "can be presented "
               "by parts";
    // equivalent:
    // string m = "very long message can be presented by parts";
}

```

在 *h* 变量中放入字符串 "Hello"。

在 *b* 变量中写入包含双引号的文本。

z 变量由空字符串初始化。这基本相当于在没有初始化的情况下描述 *z*，但有一些细微差别。此外，如后面的 [变量的初始化](#) 章节中，我们将了解未初始化的字符串会得到一个特殊的值 NULL，与前面介绍过的 "" 不同，始终会为终止零分配内存。这种差异会影响字符串 [比较运算符](#) 以及其他某些运算符的执行。随着内容的深入，我们将涉及所有这些方面的内容。

t 变量将获得一个文本，当使用 *Print* 函数在日志中打印或通过其他方法显示时，该文本将被分成 2 个字符串。

n 变量中记录的字符串 "123" 并不是一个数字，虽然看起来很像。MQL5 中有一些函数可以将文本与数字相互转换（参见 [数据转换](#)）。此外，还有一组单独的函数用于 [处理字符串](#)。

为了方便起见，可以将长字面量写入多个字符串中，就像 *m* 变量。一般规则如下：编译器合并所有字面量，直至出现表示变量说明结束的分号。在这种格式中，切记必要时在字符串的每个片段中添加一个中间空格（例如，在上例中用空格分隔消息中的单词）。

对于字符串，定义了求和（拼接）运算，用 '+' 字符表示。我们将在涉及表达式的相关章节中讨论（参见 [算术运算](#)）。

可以单独读取字符串字符，将它们当做数组元素（参见 [数组的使用](#)）：如果 *s* 是一个字符串，那么 *s[i]* 是其中第 *i* 个字符的代码，类型为 *ushort*。

2.2.5 逻辑（布尔）类型

逻辑类型用于存储只有以下 2 种可能状态的特征：“启用”/“禁用”。其界面等效项是许多程序（也包括 MetaTrader 5）的设置对话框中的选项：每个标志都可以被启用或禁用。通过检查这些特征的状态，可以使程序执行逻辑产生分支，这就是该类型名称的由来。

在 MQL5 中，使用 *bool* 关键字来定义逻辑类型，占用 1 个字节的内存。这种类型保留了两个常量：*true* 和 *false*。此外，还有一些情况是允许的（而且程序员也经常这样使用），其中 *bool* 是整数和实数的计算结果，0 值为 *false*，其他值为 *true*。

还支持将 *bool* 类型值反向解释为一个数字，即：*true* 被视为 1，*false* 被视为 0。

MQL5/Scripts/MQL5Book/p2/TypeBool.mq5 文件中给出了逻辑类型变量的示例。

```

void OnStart()
{
    bool t = true;           // true
    bool f = false;          // false
    bool x = 100;             // x = true
    bool y = 0;               // y = false
    int i = true;             // i = 1
    int j = false;            // j = 0
}

```

对于逻辑类型，提供了一组特殊的逻辑运算（参见 [逻辑（布尔）运算](#) 和 [比较运算](#)）。

2.2.6 日期和时间

MQL5 提供了一种特殊类型 *datetime* 用以存储时间数据。正如其名称所示，*datetime* 的值包括日期和时间。但在必要时也可以只包含日期或时间。

这种类型的值可以在程序中用来监控事件，比如交易时间、新闻发布或在不良交易后临时禁用 EA 交易的超时时间。

datetime 在内存中的大小为 8 个字节。数据的内部表示与 *ulong* 类型完全相同，因为其内部存储的是自 1970 年 1 月 1 日以来经过的所有秒数。支持的最大日期是 3000 年 12 月 31 日。

datetime 常量被记录为用单引号括起来的字面量字符串，前面有字符 'D'。字符串中分配了 6 个字段，日期和时间的所有组成部分的数字采用以下格式：

```

D'YYYY.MM.DD HH:mm:ss'
D'DD.MM.YYYY HH:mm:ss'

```

此处，YYYY 表示年、MM 表示月、DD 表示日、HH 表示时、mm 表示分，ss 表示秒。你可以跳过日期或时间。也可以秒秒，或者同时省略分钟和秒。

关于日期的最大允许值，MQL5 中提供了一个特殊常量 DATETIME_MAX，其等于整数值 0x793406fff，对应 D"3000.12.31 23:59:59"。

MQL5/Scripts/MQL5Book/p2/TypeDateTime.mq5 文件给出了记录 *datetime* 类型值的示例。

```

void OnStart()
{
    // WARNINGS: invalid date
    datetime blank = D'';           // blank = day of compilation
    datetime noday = D'15:45:00';    // noday = day of compilation + 15:45
    datetime feb30 = D'2021.02.30'; // feb30 = 2021.03.02 00:00:00
    datetime mon22 = D'2021.22.01';  // mon22 = 2022.10.01 00:00:00
    // OK
    datetime dt0 = 0;                // 1970.01.01 00:00:00
    datetime all = D'2021.01.01 10:10:30'; // 2021.01.01 10:10:30
    datetime day = D'2025.12.12 12'; // 2025.12.12 12:00:00
}

```

前四个变量引发编译器警告，提示日期不正确。如果为 *blank*，则字面量完全为空。在 *noday* 变量中，没有日。在这两种情况下，编译器都会替换掉常量中的编译日期。*feb30* 和 *mon22* 变量包含的日编号和月编号都不正确。编译器会自动更正它们，将溢出转移到高阶字段中（将 2 月 30 日改为 3 月 2 日，将 22 月改为次年的 10 月）。但是，我们始终建议消除警告。

`dt0` 变量演示用一个整数对 `datetime` 值进行初始化。

`datetime` 类型支持整数内部的一组运算 (参见 [表达式](#))。例如, 允许将预定义的秒数与时间相加 (得出未来的某个时刻) 或计算日期之间的差值。

2.2.7 颜色

MQL5 有一种特殊类型, 用于处理颜色。这种类型可对图形对象进行着色。

可使用 `color` 关键字来表示这种类型。为 `color` 类型值分配了 4 字节的内存。它的内部表示是一个无符号整数, 包含 RGB (红、绿、蓝) 格式的颜色, 即红、绿、蓝分别具有单独的强度级别。调和这三个分量可以得到任何可见的色调。调和绿色和红色得到黄色, 调和红色和蓝色得到紫色, 等等。

为每种分量分配 1 个字节, 那么值范围就是 0 到 255。例如, 三个分量均为零会得到黑色, 而三个分量均为最大值 255 则得到白色。

如果我们将 `color` 表示为十六进制记数法中的 `uint`, 则颜色表示如下: `0x00BBGGRR`, 其中 `RR`、`GG` 和 `BB` 都是单字节无符号整数。

为了方便用户, MQL5 支持一种特殊形式的字面量, 用以记录颜色常量。字面量表示用逗号分隔并用单引号括起来的三个数字。字面量前面是字符 `'C'`。例如, `C'0,128,255'` 表示一种颜色, 其中 0 代表红色分量, 128 代表绿色分量, 255 代表蓝色分量。也可以使用数字的十六进制记数法: `C'0x00,0x80,0xFF'`。

此外, MQL5 中嵌入了一长串预定义的色调, 都以 `clr` 开头。例如, `clrMagenta`、`clrLightCyan` 和 `clrYellow`。当然, 它们也包括三原色: `clrRed`、`clrGreen` 和 `clrBlue`。完整列表可以在 MetaEditor 帮助中找到。

以下是一些设置颜色的示例 (也可以在 `MQL5/Scripts/MQL5Book/p2/TypeColor.mq5` 文件中找到):

```
void OnStart()
{
    color y = clrYellow;           // clrYellow
    color m = C'255,0,255';        // clrFuchsia
    color x = C'0x88,0x55,0x01';   // x = 136,85,1 (no such predefined color)
    color n = 0x808080;            // clrGray
}
```

2.2.8 枚举

枚举是 MQL5 中内置的一组类型, 每个枚举都包含一组命名常量来描述相关的概念或特性。这些常量也称为枚举元素。

例如, `ENUM_DAY_OF_WEEK` 枚举包含表示一周中所有天数的常量:

标识符 (ID)	说明	值
SUNDAY	星期日	0
MONDAY	星期一	1
TUESDAY	星期二	2
WEDNESDAY	星期三	3
THURSDAY	星期四	4
FRIDAY	星期五	5
SATURDAY	星期六	6

ENUM_ORDER_TYPE 枚举描述了 MetaTrader 5 中支持的所有订单类型：

标识符 (ID)	说明	值
ORDER_TYPE_BUY	市场买入订单	0
ORDER_TYPE_SELL	市场卖出订单	1
ORDER_TYPE_BUY_LIMIT	限价买入挂单	2
ORDER_TYPE_SELL_LIMIT	限价卖出挂单	3
ORDER_TYPE_BUY_STOP	买入止损挂单	4
ORDER_TYPE_SELL_STOP	卖出止损挂单	5
ORDER_TYPE_BUY_STOP_LIMIT	达到订单价格后，限价买入挂单会以限价止损价格成交	6
ORDER_TYPE_SELL_STOP_LIMIT	达到订单价格后，限价卖出挂单会以限价止损价格成交	7
ORDER_TYPE_CLOSE_BY	通过反向头寸平仓的指令	8

有几十种不同枚举。枚举的名称以 “ENUM_” 为前缀。我们将在接触到相关领域时学习这些枚举。

每个枚举都是独立的类型。但它们的内部表示是相同的，即四字节整数 (*int*)。每个枚举常量都被编码为一个特定数字，但在大多数情况下，程序员无需记住这些数字，因为使用枚举的主要目的就是用明显的标识符替换内部表示。

编译器确保枚举值始终是其中一个重新定义常量。否则，将出现警告或编译错误（具体请参见示例）。

这是 ENUM_DAY_OF_WEEK 枚举的底层表示形式 (*MQL5/Scripts/MQL5Book/p2/TypeEnum.mq5* 脚本)。

```

void OnStart()
{
    ENUM_DAY_OF_WEEK sun = SUNDAY;    // sun = 0
    ENUM_DAY_OF_WEEK mon = MONDAY;    // mon = 1
    ENUM_DAY_OF_WEEK tue = TUESDAY;   // tue = 2
    ENUM_DAY_OF_WEEK wed = WEDNESDAY; // wed = 3
    ENUM_DAY_OF_WEEK thu = THURSDAY;  // thu = 4
    ENUM_DAY_OF_WEEK fri = FRIDAY;    // fri = 5
    ENUM_DAY_OF_WEEK sat = SATURDAY;   // sat = 6

    int i = 0;
    ENUM_DAY_OF_WEEK x = i; // warning: implicit enum conversion
    ENUM_DAY_OF_WEEK y = 1; // ok, equals to MONDAY
    ENUM_ORDER_TYPE buy = ORDER_TYPE_BUY;    // buy = 0
    ENUM_ORDER_TYPE sell = ORDER_TYPE_SELL; // sell = 1
    // ...

    // warning: implicit conversion
    //      from 'enum ENUM_DAY_OF_WEEK' to 'enum ENUM_ORDER_TYPE'
    //      'ENUM_ORDER_TYPE::ORDER_TYPE_SELL' will be used
    //      instead of 'ENUM_DAY_OF_WEEK::MONDAY'
    ENUM_ORDER_TYPE type = MONDAY;
    // compilation error: uncomment to reproduce
    // ENUM_DAY_OF_WEEK day = ORDER_TYPE_CLOSE_BY; // cannot convert enum
    // ENUM_DAY_OF_WEEK z = 10; // '10' - cannot convert enum
}

```

表示一周内日期的所有常量被编码为数字 0 到 6，以星期日为起点。原则上说，常量不一定是连续数字，也不一定从 0 开始。某些枚举中就不是这样。

请注意，相同常量在不同枚举类型中可能代表不同含义。例如，对于 `ENUM_ORDER_TYPE` 枚举中的 `ORDER_TYPE_BUY` 和 `ORDER_TYPE_SELL` 订单，`ENUM_DAY_OF_WEEK` 中的 `SUNDAY` 和 `MONDAY` (表示一周内的日期) 具有相同的值 (0 和 1)。

把简单整数变量 `i` 的值拷贝到枚举变量 `x` 时，编译器会给出一个警告，因为在程序执行阶段，变量 `i` 中可能包含不允许的常量值。

在变量 `y` 中，我们获取数字 1 (表示 `MONDAY`)，编译器认为这是一个正确的操作。

如果尝试将一个枚举的常量写入另一个枚举的变量 (如上例中将 `MONDAY` 写入 `type` 变量) 可能会导致有关隐式类型转换的警告。如果正在写入的常量与某个目标枚举元素的值相同，就会发生这种情况。也就是说，这两个枚举各自有一个具有相关值的元素。然后，编译器代替程序员自动执行隐式转换，但会显示一个警告，提醒程序员检查是否一切正常：实际上，用 `ORDER_TYPE_SELL` 替代 `MONDAY` 确实很怪异；但为了进行演示，我们故意这么做了。

如果被拷贝的元素的值与另一个枚举的任何元素都不匹配 (例如将 `ORDER_TYPE_CLOSE_BY` 写入 `day` 变量中)，则会生成编译错误，因为隐式转换无法完成。

带有 `z` 变量的注释字符串也会导致编译错误，因为 `ENUM_DAY_OF_WEEK` 不包含值 10。如果程序员确信，在特殊情况下，仍然需要在枚举类型变量中获取随机值，则可以使用显式类型强制转换。

显式和隐式类型强制转换将在 [类型强制转换](#) 一节中讨论。

MQL5 允许程序员使用 `enum` 关键字来声明自己应用的枚举。这个特性将在下一节 [自定义枚举 \(enum\)](#) 中说明。

2.2.9 自定义枚举

自定义枚举在结构上是基于 `int` 类型的，它们的使用原则与上一节嵌入式枚举中介绍的原则完全一致。因此，本节我们介绍的是自定义枚举，严格来说，它们不属于嵌入式。

可使用 `enum` 关键字在 MQL5 代码中描述自定义枚举。最简单的说明形式如下：

```
enum name
{
    element1,
    element2,
    element3
};
```

该说明在程序中注册了一个命名为 `name` 的枚举类型，各元素用花括号括起来并用逗号分隔（元素数量只取决于 `int` 的最大值，在实际任务中可以认为没有上限）。随后，标识符 `element1`、`element2` 和 `element3` 便可以在程序中定义了这些标识符的上下文中使用：全局（即在所有函数外部）或函数内部（参见 [上下文、可见性和变量生命周期](#) 一节）。

请注意右花括号后面的分号。这个分号是必需的，因为枚举说明是一个独立语句，任何 MQL5 语句之后必须有分号。

默认情况下，标识符取常量值，从 0 开始，依次递增 1。如有必要，程序员可以在标识符右边使用 '=' 符号为每个元素定义特定值。例如，上文中的语句与以下语句等效：

```
enum name
{
    element1 = 0,
    element2 = 1,
    element3 = 2
};
```

指定的值必须是常量或者是编译器在编译阶段可以计算的表达式（更多详细信息，请参见以下示例）。

如果没有为所有元素定义值，则会自动计算跳过的值，计算方法是最近的（前面的）已知值自动加 1。例如，

```
enum name
{
    element1 = 1,
    element2,
    element3 = 10,
    element4,
    element5
};
```

这里，前两个元素取值 1 和 2（计算得出），从第三个元素开始取值 10（显式赋值）、11 和 12（后两个用 10 自动加 1 计算得出）。

`TypeUserEnum.mq5` 脚本中提供了一些描述自定义枚举的例子。

```

const int zero = 0; // runtime value is not known at compile time

enum
{
    MILLION = 1000000
};

enum RISK
{
    // OFF      = zero, // error: constant expression required
    LOW        = -1,
    MODERATE   = -2,
    HIGH       = -3,
};

enum INCOME
{
    LOW        = 1,
    MODERATE   = 2,
    HIGH       = 3,
    ENORMOUS   = MILLION,
};

void OnStart()
{
    enum INTERNAL
    {
        ON,
        OFF,
    };

    // int x = LOW; // ambiguous access, can be one of
    int x = RISK::LOW;
    int y = INCOME::LOW;
}

```

INTERNAL 枚举展示了在函数内部描述自定义枚举的可能性，通过这种描述，可以限定该类型的可见/可用域，这在处理名称冲突方面很有用。

RISK 枚举表明元素可被赋予负值。无法描述注释元素 OFF，原因是尝试用一个非常量表达式来初始化此元素：在这种情况下，指定了 **zero** 变量，但编译器无法计算出此变量的值。

在 INCOME 枚举中，用上文定义的其他枚举的 MILLION 元素的值成功初始化了 ENORMOUS 元素。枚举是在编译时创建的，因此它们在初始化表达式中是可用的。

包含 MILLION 的枚举没有名称，这样的枚举称为匿名枚举。它们的基本用途是声明常量。而命名枚举则更多地用于声明常量，因为它们允许根据元素的含义对元素进行分组。

示例中定义了 2 个枚举类型，它们的元素具有相同名称，因此在声明 **x** 变量时指定 LOW 标识符会导致编译错误“访问权限不明确”，因为未明确说明是哪个元素的枚举。请注意，标识符可能有不同的值（在本例中就是这样）。

为了解决这个问题，提供了一个特殊的上下文运算符：两个冒号 “::”。它们有助于形成语言元素（即枚举元素）的完整标识符，在我们的例子中：首先，指定枚举名称，然后是运算符 “::”，最后是元素标识符。示例：RISK::LOW 和 INCOME::LOW。我们将在相关章节中了解所有运算符。

2.2.10 Void 类型

void 类型是一种特殊类型。表示空（无类型），并且不占用内存。它仅用于描述不返回任何值或没有任何参数的函数。我们之前就学过了这种函数的一个例子：第一章中 *HelloChart* 脚本中的 *OnStart* 函数。将在 [函数](#) 一节详细讨论。

无法用 **void** 类型来描述变量；然而，在描述对类的随机对象的引用时，**void** 却是基本类型。这种可能性在 [第三章](#) 涉及面向对象的编程的内容中进行了介绍。

2.3 变量

在本章中，我们将学习在 MQL5 中使用变量的基本原则，即涉及嵌入式数据类型的原则。具体来说，我们将探讨变量的声明和定义、上下文要求的初始化特殊特性、生存期以及改变变量特性的基本修饰符。后面，我们将基于这些知识，用新的自定义类型（联合体、自定义枚举和别名）、类、指针以及引用来扩展变量的功能。

MQL5 中的变量提供了一种存储各种类型数据的机制，在组织程序逻辑和市场信息操作中起着重要作用。本节包括以下小节：

变量的声明和定义：

- ① 声明变量是在程序中创建变量的一个步骤。在本节中，我们将探讨如何声明和定义变量，以及如何指定变量的类型。

变量的上下文、作用域和生存期：

- ① 变量可以在不同的上下文和作用域中存在，这决定了它们的可用性和生存期。本小节涵盖的这些方面，能帮助您理解变量与代码的交互方式。

初始化：

- ① 变量的初始化包括给变量赋初始值。我们将学习初始化的方法，初始化有助于避免未定义的程序行为。

静态变量：

- ① 静态变量的值在函数调用之间保持不变。本节说明如何使用静态变量在不同的代码执行之间存储信息。

常量变量：

- ① 常量变量是指其值在程序执行过程中不会改变。本节介绍了常量变量的用法和特点。

输入变量：

- ① 在交易机器人中使用输入变量来配置策略参数。我们将看到如何使用输入变量来创建灵活、可定制的交易系统。

外部变量：

- ① 外部变量允许用户与程序交互，因为无需修改代码就可以更改外部变量的值。本节介绍外部变量的工作原理。

2.3.1 变量的声明和定义

变量是一个命名的内存单元，用于存储特定类型的数据。为了让程序能够操作一个变量，程序员必须在源代码中声明和/或定义该变量。在一般情况下，“声明”和“定义”这两个术语对于程序元素来说有着不同含义，而对于变量来说，这两个术语几乎没有区别。当我们开始了解函数、类和特殊（外部）变量时，就会涉及到这些错综复杂的关系。在本书中，我们将交替使用这两个术语，并以“说明”作为概括性词语。

可以肯定地认为，声明包含程序元素的说明以及在程序中使用时需要全部属性。而定义则包含此元素的具体实现，与声明对应。

声明允许编译器将程序的所有元素相互连接。编译器则基于定义生成一个可执行代码。

对于变量而言，它们的声明实际上总是充当定义，因为这可以确保根据变量的类型来分配内存和解释内容（这正是变量的实现）。唯一的例外是用‘extern’关键字声明变量（有关更多详细信息，参见[外部变量](#)一节）。

只有在对变量进行描述后，才能使用特殊语句向该变量输入值，读取这些值，并引用变量名以在程序的不同部分之间传递变量。

下面这个非常简单的例子显示了一个描述变量的语句：

```
type name;
```

此处，*name* 必须满足构造 [标识符](#) 的要求。对于 *type*，你可以指定我们在上一节中介绍过的任意 [嵌入式类型](#)，也可以指定其他自定义类型，稍后我们就会学习如何创建自定义类型。例如，整数变量 *i* 的声明方式如下：

```
int i;
```

如有必要，您可以同时描述多个相同类型的变量。在这种情况下，变量的名称在语句中指定，并用逗号分隔。

```
int i, j, k;
```

语句（包含变量说明）在程序中的位置至关重要。这会影响变量的生存期以及在程序不同部分中的可访问性。

2.3.2 变量的上下文、作用域和生存期

SQL5 是一种使用花括号将语句分组为代码块的编程语言。

回顾一下，一个程序由包含语句的代码块组成，其中必须明确存在一个代码块。在第一章的脚本示例中，我们看到了 **OnStart** 函数。这个函数的主体（函数名后面花括号里的文本）就是这样必要的代码块。

在每个代码块内部形成了局部上下文，也就是一个区域，用于限制其中描述的变量的可见性和生存期。到目前为止，我们只遇到了用花括号定义函数主体的例子。但是，花括号也可以用于构成 [复合运算符](#)，在用于 [描述类](#) 和 [命名空间](#) 的语法中使用。所有这些方法还定义了可见性区域，将在相关章节中讨论这些方法。在此阶段，我们只探讨一种类型的局部代码块，即函数内部的局部代码块。

除了局部区域之外，每个程序也有一个全局上下文，在这个区域中，变量、函数和其他实体的定义不受其他代码块的约束。

在简单的脚本端，MQL 向导创建了唯一的 `void` 函数 **OnStart**，那么该函数内部只有 2 个区域：一个全局区域和一个局部区域（局部区域在 **OnStart** 函数体的内部，尽管是空的）。以下脚本通过注释说明了这一点。

```
// GLOBAL SCOPE
void OnStart()
{
    // LOCAL SCOPE "OnStart"
}
// GLOBAL SCOPE
```

请注意，除了不涵盖 **OnStart** 函数，全局区域可以扩展到任何地方（包括 **OnStart** 函数前后）。全局区域基本涵盖了函数除外的所有内容（如果有多个函数的话），但是在这个脚本中，除了 **OnStart** 之外没有任何内容。

我们可以在文件的顶部描述变量，如 *i, j, k*，这些变量会成为全局变量。

```
// GLOBAL SCOPE
int i, j, k;
void OnStart()
{
    // LOCAL SCOPE "OnStart"
}
// GLOBAL SCOPE
```

全局变量在终端中启动 MQL 程序后立即创建，并且在整个程序执行期间都存在。

程序员可以在程序的任何地方记录和读取全局变量的内容。

原则上，建议只在文件顶部描述全局变量，但这并非强制要求。如果我们将声明移到 **OnStart** 函数整体的下方，本质上没有任何变化。只不过其他程序员很难立即就能弄清楚包含变量的代码的含义，因为他们仍然需要了解变量的定义。

有趣的是，**OnStart** 函数本身也是在全局上下文中声明的。如果我们添加另一个函数，也将在全局上下文中声明。回想一下我们在第一章中是如何创建 **Greeting** 函数，以及如何从 **OnStart** 函数调用 **Greeting** 函数。这就是因为函数名以及引用方法（即如何执行）在源代码中具有全局可见性。[命名空间](#) 给这一机制增加了一些细节；我们将在后面学习这些知识。

每个函数内部的局部区域只属于函数本身：一个局部区域在 **OnStart** 内部，另一个在 **Greeting** 内部，这是该函数本身的区域，既不同于 **OnStart** 的局部区域，也不同于全局区域。

函数主体中描述的变量称为局部变量。局部变量是根据程序执行期间调用相关函数时的说明而创建的。局部变量只能在包含它们的代码块的内部使用。在外部不可见，也无法从外部访问。退出函数时，局部变量被销毁。

描述 **OnStart** 函数中 *x, y, z* 局部变量的示例：


```
// GLOBAL SCOPE
int i, j, k;
void OnStart()
{
    // LOCAL SCOPE "OnStart"
    int x, y, z;
}
// GLOBAL SCOPE
```

应该注意，成对的花括号既可以用于说明函数和其他语句，它本身也可以用来构成内部代码块。单元嵌套没有限制。

通常添加嵌套代码块是为了最大程度上缩小逻辑隔离的小代码段中使用的变量的作用域（由于某些原因没有被函数设置为隔离）。这可以降低以下风险：在没有提供变量的情况下错误修改变量；或者由于试图将同一变量重用于各种需求（这种做法并不可取）而导致的意外副作用。

在下面这个示例函数中，单元嵌套级别为 2（如果我们将包含函数主体的代码块视为第一级），并且创建了 2 个这样的代码块并将连续执行。

```
void OnStart()
{
    // LOCAL SCOPE "OnStart"
    int x, y, z;

    {
        // LOCAL SUBSCOPE 1
        int p;
        // ... use p for task 1
    }

    {
        // LOCAL SUBSCOPE 2
        // y = p; // error: 'p' - undeclared identifier
        int p;    // from now 'p' is declared
        // ... use p for task 2
    }

    // p = x; // error: 'p' - undeclared identifier
}
```

在这两个代码块内部，都描述了变量 **p**，该变量在这两个代码块中用途不同。事实上这是两个不同的变量，尽管在每个代码块内部显示了相同的名称。

如果变量移动到函数的局部变量的初始列表中，那么在第一个代码块退出后，这个变量可能还包含一些残存值，从而中断第二个代码块的运行。此外，程序员偶尔会将 **p** 包含在函数一开头的某些其他元素中，那么副作用在第一个代码块就会出现。

在两个嵌套代码块之外，变量 **p** 是未知的，因此，试图从函数的公共代码块中引用该变量会导致编译错误（“未声明的标识符”）。

还应该注意的是，可以不在代码块最开始的位置描述变量，而是在代码块的中间或者靠近结尾处描述。这样，变量的作用域不是整个代码块，而是仅从定义的位置向下延伸。因此，在该变量的描述位置之前引用该变量也会出现同样的错误。

因此，变量作用域不同于上下文（整个代码块）。

下面的示例说明这个问题的两种情况：尝试使用语句 $p = x$ 和 $y = p$ 包含任何字符串，并编译源代码。

一旦在函数内部传递了控制权，就为函数的所有局部变量分配内存。然而，这并不是变量创建的终点。还要对变量进行初始化（设置初始值），初始化可由程序员显式定义，也可以由编译器的默认值隐式定义。同时，描述变量的上下文也很重要。

2.3.3 初始化

在描述变量时，有可能会设置初始值；初始值是在变量名和 '=' 符号之后指定的，并且必须与变量类型对应，否则就会被强制转换为该类型（类型强制转换请参见相关章节）。

```
int i = 3, j, k = 10;
```

此处 i 和 k 被显式初始化，而 j 没有。

常量（相关类型的字面量）和表达式（一种计算公式）都可以被指定为初始值。我们将单独讨论表达式。先看一个简单的例子：

```
int i = 3, j = i, k = i + j;
```

此处， j 变量取与 i 变量相同的值，而 k 变量取 i 与 j 之和。严格地说，这三种情况都涉及到表达式。不过常量（3）比较特殊，是一个退化表达式选项。在第二种情况下，唯一的变量名是一个表达式，即表达式结果将作为此变量的值（不进行任何转换）。在第三种情况下，表达式中访问 i 和 j 两个变量，用它们的值执行加法运算，之后运算结果传递给 k 变量。

由于包含多个变量说明的语句是从左到右处理，所以编译器在分析另一个说明时已经知道了前面变量的名称。

一个程序通常包含许多带有变量说明的语句。编译器以自上而下的自然方式读取语句。在后面的初始化中，可以使用取自前面说明中的名称。以下是由两个不同语句描述的不同变量。

```
int i = 3, j = i;
int k = i + j;
```

未进行显式初始化的变量也会获得一些初始值，但具体初始值取决于描述变量的位置，即变量的上下文。

在未进行初始化的情况下，局部变量在生成时取随机值：编译器根据类型大小为局部变量分配内存，而具体地址处的内容未知（计算机内存区域常因先前执行的程序不再需要而重新分配给其他程序使用）。

通常建议在算法代码中通过后续操作向未初始化的局部变量写入有效值，比如使用我们稍后要介绍的 **赋值运算**。从语法上来说，这种操作类似于初始化，因为也会使用等号 '=' 将值从它右边的结构体（可以是常量、变量、表达式或函数调用）传递给左边的变量。'=' 左边只能是变量。

程序员应该确保只有在为未初始化的变量赋予了有意义的值之后再读取其内容。否则编译器会显示一个警告（“可能使用了未初始化的变量”）。

而如果使用全局变量，情况就完全不同了。

全局变量的一个例子是第二章中 *GoodTime2* 脚本的 *GreetingHour* 输入参数。使用 *input* 关键字描述变量并不影响它作为变量而具备的其他特性。我们可以不对其进行初始化，以如下方式描述：

```
input uint GreetingHour;
```

这不会对程序产生任何影响，因为如果没有进行显式初始化，编译器会将全局变量隐式初始化为零（我们以前也将它显式初始化为零）。

无论是哪种变量类型，隐式初始化总是由一个等于零的值来执行。例如，对于 *bool* 变量，设置为 *false*，而对于 *datetime* 变量，设置为 *D'1970.01.01 00:00:00'*。对于字符串，则有一个特殊的值 *NULL*。可以说，这是一个比空引号 *""* 更“空”的字符串，仍然空引号仍会被分配一些内存并存储终止空字符。

除了局部变量和全局变量，还有另一种类型，即静态变量。如果程序员没有显式地写入初始值，编译器会将它们隐式初始化为零。将在下一节中介绍静态变量。

我们来创建一个新脚本，*VariableScopes.mq5*，用于演示局部变量和全局变量（*SQL5/Scripts/MQL5Book/VariableScopes.mq5*）。

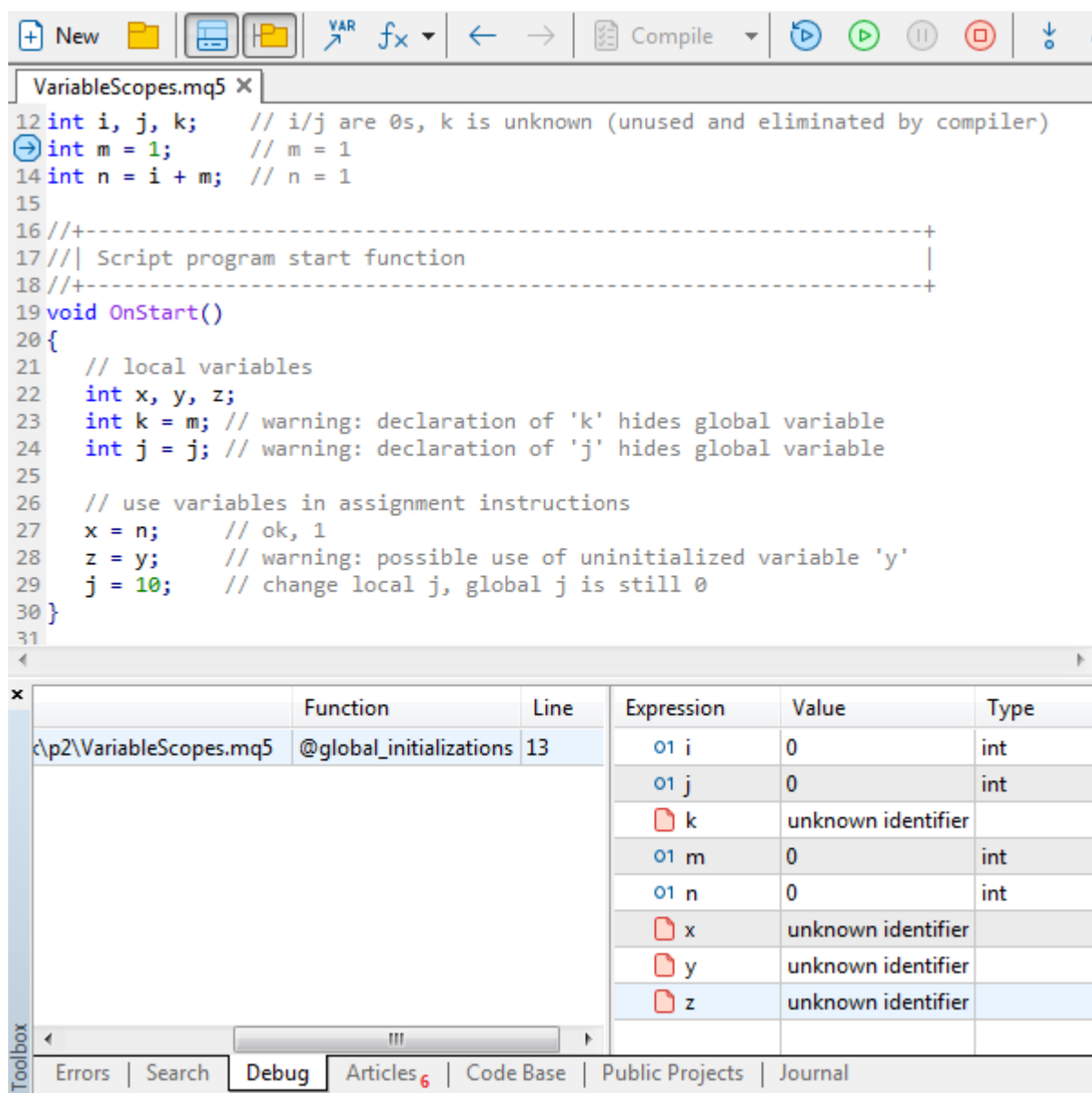
```
// global variables
int i, j, k;    // all are 0s
int m = 1;      // m = 1                (place breakpoint on this line)
int n = i + m;  // n = 1
void OnStart()
{
    // local variables
    int x, y, z;
    int k = m; // warning: declaration of 'k' hides global variable
    int j = j; // warning: declaration of 'j' hides global variable
    // use variables in assignment statements
    x = n;      // ok, 1
    z = y;      // warning: possible use of uninitialized variable 'y'
    j = 10;     // change local j, global j is still 0
}
// compilation error
// int bad = x; // 'x' - undeclared identifier
```

切记，在启动 MQL 程序时，终端首先初始化所有全局变量，然后调用一个作为相关类型程序的起始点的函数。在本例中，对于脚本来说，这个函数是 *OnStart*。

此处，仅变量 *i*, *j*, *k*, *m*, *n* 是全局的，因为它们在函数外部描述（本例中只存在脚本所必需的 *OnStart* 函数）。*i*, *j*, *k* 隐式取值 0。*m* 和 *n* 包含 1。

可以在调试模式下逐步运行脚本，并确认变量值完全是以这种方式变化。为此，您应该预先在其中一个全局变量的初始化代码行上设置断点，比如在 *m* 变量上。将文本光标放在这个字符串上，执行 *Debug -> Toggle Breakpoint* (F9)，该字符串将在左边字段中以蓝色标记高亮显示，这表示如果程序在调试器中运行，则执行将此处就会停止。

然后，您应该实际运行程序进行调试，为此请执行命令 *Debug -> Start on real data* (F5)。此时，将在终端打开新的图表，并开始在此图表中执行脚本（右上角出现标题“VariableScopes (Debugging)”），但是执行立即中止，系统回到 MetaEditor。我们应该看到一张图，如下所示。



在 MetaEditor 中进行逐步调试和查看变量

包含断点的字符串以箭头标记，这是程序准备执行但尚未执行的当前语句。程序的当前栈显示在左下方，目前只包含一个条目：`@global_initializations`。可在右下角输入表达式来监控它们的实时值。我们想查看变量值，因此需要连续输入 `i`, `j`, `k`, `m`, `n`, `x`, `y`, `z`（每行输入一个变量）。

你将看到，MetaEditor 会自动添加当前上下文的变量以便我们查看（例如，函数内部执行语句时的局部变量和函数输入）。但是现在，我们要提前手动添加 `x`, `y` 和 `z`，只是为了证明它们不是在函数外部定义的。

请注意，对于局部变量，会显示“未知标识符”而不是一个值，因为此时尚未出现这些局部变量所在的 `OnStart` 函数代码块。全局变量 `i` 和 `j` 将首先获得零值。全局变量 `k` 没有在任何地方使用，因此，编译器排除了这个变量。

如果我们使用 **Step Into** (F11) 或者 **Step Over** (F10) 命令来执行程序的一个步骤（在当前代码行执行语句），我们将看到变量 `m` 取值为 1。下一步是继续初始化变量 `n`，该变量也会变为 1。

全局变量的说明在此处结束，正如我们所知，终端在全局变量初始化完成时调用 `OnStart` 函数。在这种情况下，要以步进模式进入 `OnStart` 函数，请再次按 F11（或者也可以在 `OnStart` 函数开头再设置一个断点）。

当程序语句执行到定义了局部变量的代码块时，局部变量会被初始化。因此，`x`、`y`、`z` 变量只有单步进入 `OnStart` 函数时才会被创建。

当调试器进入 `OnStart` 函数时，如果运气好的话，你可以看到 `x`、`y` 和 `z` 中确实有初始随机值。这里的“运气”是指这类随机值很可能恰好是零。那么就无法将它们与编译器为全局变量隐式执行的隐式初始化（值为零）区分开来。如果脚本反复启动，局部变量中产生的“垃圾”可能会不同，这更能说明问题。局部变量没有被显式初始化，因此其内容可以是任何类型。

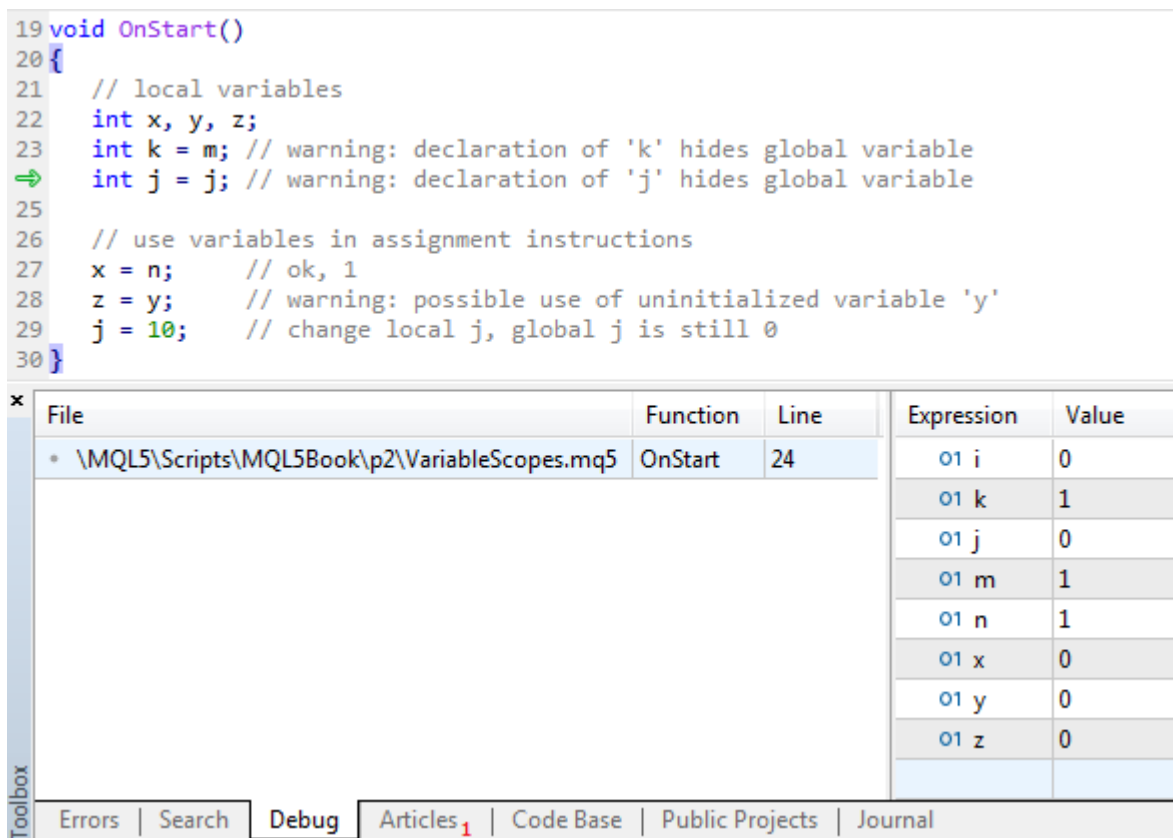
在以下的图像序列中，您可以使用调试器的逐步模式查看变量的变化过程。待执行（但尚未执行）的当前字符串在枚举的字段上用绿色箭头标记。

```
19 void OnStart()  
20 {  
21     // local variables  
22     int x, y, z;  
23     int k = m; // warning: declaration of 'k' hides global variable  
24     int j = j; // warning: declaration of 'j' hides global variable  
25  
26     // use variables in assignment instructions  
27     x = n;      // ok, 1  
28     z = y;      // warning: possible use of uninitialized variable 'y'  
29     j = 10;     // change local j, global j is still 0  
30 }
```

File	Function	Line	Expression	Value
• \MQL5\Scripts\MQL5Book\p2\VariableScopes.mq5	OnStart	23	01 i	0
			01 k	444065768
			01 j	0
			01 m	1
			01 n	1
			01 x	0
			01 y	0
			01 z	0

Errors | Search | **Debug** | Articles 1 | Code Base | Public Projects | Journal

在 MetaEditor 中进行逐步调试和查看变量（字符串 23）



在 MetaEditor 中进行逐步调试和查看变量 (字符串 24)

代码中进一步演示了如何以最简单的方式通过赋值运算符来使用这些变量。全局变量 *n* 的值被成功拷贝到局部变量 *x* 中, 因为 *n* 已经初始化。但是, 在将变量 *y* 的值复制到变量 *z* 的字符串中, 编译器发出一个警告, 因为 *y* 是局部变量, 而到目前为止, 还没有在其中写入任何值, 即它没有被显式初始化, 也没有其他运算符为它赋值。

在函数内部, 允许使用与全局变量相同的名称来描述变量。如果在内部代码块中创建的变量的名称存在于外部代码块中, 则在嵌套的局部代码块中也会出现类似的情况。但不建议采取这种做法, 因为可能会导致逻辑错误。在这种情况下, 编译器会给出警告 (“声明隐藏了全局/局部变量”)。

由于这种重新定义, 局部变量 (如上例中的 *k*) 会在函数内部覆盖同名的全局变量。虽然这两个变量名称相同, 但实际上是两个不同的变量。局部变量 *k* 仅在 *OnStart* 内部可见, 而全局变量 *k* 在除了 *OnStart* 以外的任何地方都是可见的。也就是说, 变量 *k* 的任何代码块内操作只会影响局部变量。因此, *OnStart* 函数退出时 (假设它不是脚本唯一的核心函数), 我们会发现全局变量 *k* 仍然为零。

局部变量 *j* 不仅会覆盖全局变量 *j*, 而且还会以后者的值完成初始化。在 *OnStart* 内包含 *j* 的说明的字符串中, 当从 *j* 的局部版本读取初始值时, *j* 的局部版本还在创建过程中 (创建还未完成)。在成功定义了局部变量 *j* 之后, 它与全局版本同名, 但它是局部版本, 后续对 *j* 的改变都属于局部版本。

在源代码的结尾, 我们注释了声明另一个全局变量 *bad* 的尝试, 其初始化中调用了变量 *x* 的值。该字符串会导致编译器错误, 因为变量 *x* 在 *OnStart* 函数内定义的, 在该函数外部则不可见。

2.3.4 静态变量

有时有必要在函数内部描述一个变量，确保它在整个程序执行期间中都存在。例如，我们要计算该函数被调用的次数。

那么这个变量不能是局部的，这样它将失去“长期记忆”，因为局部变量在每次调用函数时被创建，并且在函数退出时被移除。从技术角度，它可以被声明为全局变量；但是，如果该变量只在该函数内部使用，这种方法在程序设计方面是不合理的。

首先，全局变量可能会在程序的任何位置被意外更改。

其次，如果我们动不动就声明一个全局变量，那么可以想到，在程序的全局区域中会产生多么“杂乱无章”的变量。因此，我们建议在使用变量的最小代码块（如果有几个嵌套代码块）中声明这些变量。

因此，应该在函数内部描述函数执行的计数器。此时，变量的新属性“静态属性”就能派上用场了。

通过在变量类型声明的前面输入一个特殊关键字（修饰符）**static**，可以将变量的生存期延长至整个程序执行期间，即让它类似于全局变量。通常，静态变量只在一个函数内局部定义。因此，与普通局部变量一样，它的可见性受到相关代码块的限制。

静态变量也可以在全局级别描述，与普通全局变量没有任何区别（至少在撰写本书时是这样）。静态变量的这一行为与其在 C++ 中的行为有所不同：在 C++ 中，静态变量的可见性仅限于其所在的文件。在 `mq5` 中，程序包含一个主 `mq5` 文件，可能还包含一些头文件（参见 [指令 #include](#)）；因此，静态变量和普通全局变量都可以从程序的所有源文件中获得。

局部静态变量只创建一次：在程序首次进入描述该变量的函数时创建。这种变量只有在卸载程序时才会被移除。如果一个函数从未被调用过，那么在其中描述的局部静态变量（如有）将永远不会被创建。

例如，我们修改第一章中的 **Greeting** 函数，以便在每次调用该函数时显示不同的问候语。我们将新脚本命名为 **GoodTimes.mq5**。

我们将移除 **GreetingHour** 脚本的输入并移除 **Greeting** 函数的参数。在 **Greeting** 函数内部，我们将描述一个新的静态变量 **counter**（整数类型），初始值为 0。请注意，因为变量是静态的，所以初始化只会执行一次。

```
string Greeting()
{
    static int counter = 0;
    static string messages[3] =
    {
        "Good morning", "Good day", "Good evening"
    };
    return messages[counter++ % 3];
}
```

因为我们现在了解了 **static** 修饰符，所以可以合理地将其用于 **messages** 数组。问题是它之前被声明为局部变量，并且每次调用 **Greeting** 函数（被多次调用）时都会被重新创建（并在函数退出时被移除）。这样效率不高。

注意，数组是由多个相同类型的值组成的命名集，可以通过名称后方括号中指定的索引来访问。上面介绍的关于变量的大部分规则都直接适用于数组。关于使用数组的更多细微差别见[数组](#)一节。

让我们回到当前的问题。我们根据 `return` 语句中 `counter` 变量的值从数组中选取一个项，这一机制目前看起来有点晦涩难懂：

```
return messages[counter++ % 3];
```

我们在第一章中曾经简要提到了使用字符 `%` 执行取模运算。使用该字符，我们可以保证元素索引不会超过数组大小：不论计数器的值如何，除以 3 的模可以是 0、1 或 2。

这同样适用于 `counter++` 结构体，该结构体表示将变量值加 1（单次递增）。

需要注意的是，在这种表示法中，将在整个表达式计算完成之后执行递增操作，在本例中，是在除法 `counter % 3` 完成之后。这意味着计数将从初始值“零”开始。也有可能是在计算表达式之前执行递增操作，代码为：`++counter % 3`。那么计数将从 1 开始。我们将在[递增和递减](#)一节中讨论这种类型的运算。

我们从 `OnStart` 调用连续调用 `Greeting` 函数 3 次。

```
void OnStart()
{
    Print(Greeting(), " ", Symbol());
    Print(Greeting(), " ", Symbol());
    Print(Greeting(), " ", Symbol());
    // Print(counter); // error: 'counter' - undeclared identifier
}
```

然后我们将在日志中看到带所有问候语的三个字符串，与预期一致。

```
GoodTimes (EURUSD,H1)      Good morning, EURUSD
GoodTimes (EURUSD,H1)      Good afternoon, EURUSD
GoodTimes (EURUSD,H1)      Good evening, EURUSD
```

如果我们继续调用函数，计数器将递增，消息将不断循环。

尝试在 `OnStart`（已注释）末尾引用 `counter` 变量将不允许编译代码，因为静态变量虽然继续存在，但只在 `Greeting` 函数内部可用。

请注意，花括号用于构成代码块和初始化数组。请区分不同的用途。我们将在相关章节中详细介绍数组。然而，这些并不是花括号的全部用途：我们将在后面学习如何使用花括号定义自定义类型、结构体和类。也可以在结构体和类的内部定义静态变量。

2.3.5 常量变量

尽管这看起来有点矛盾，但大多数编程语言都支持“常量变量”这个概念。在 MQL5 中，通过添加 `const` 修饰符来说明常量变量。这个修饰符被放置在变量说明中的类型之前，意味着变量值在通过初始值进行初始化后便无法以任何方式更改。在变量的整个生存期中，将具有不变的值，即一个常量。

编译器会阻止对这个常量赋值的操作：相关字符串中将出现错误“无法修改常量”。

const 修饰符的目的是明确表明程序员不希望他人更改相关变量，特别是已知的固定值（如计算美元指数时使用的欧元指数，一年中的周数等）。如果不打算更改变量，建议使用 **const** 修饰符。这样做有助于避免因程序员或同事试图在常量中写入其他内容而引发可能的错误。

例如，我们可以为 **Greeting** 函数中的 **messages** 数组添加 **const** 修饰符。对于这样一个小程序来说，这似乎作用不大。但由于程序往往会不断扩大，任何字符串可能迟早都会出现在更复杂的软件环境中（如新增语句、操作模式等）。所以，有必要制定 B 计划；尤其是这种操作还非常简单。

```
string Greeting()
{
    static int counter = 0;
    static const string messages[3] =
    {
        "Good morning", "Good day", "Good evening"
    };
    // error demo: 'messages' - constant cannot be modified
    // messages[0] = "Good night";
    return messages[counter++ % 3];
}
```

在带注释的字符串中，我们测试了将“Good night”字符串记录到数组的第一个元素中（请谨记，编号从 0 开始）。在本例中，此操作的意义只是确保编译器会阻止这一行为。

显而易见，**static** 和 **const** 修饰符可以组合使用。它们的前后顺序并不重要。

顺便说一下，在 MQL5 中可以通过两种方式将变量变为常量：使用 **const** 修饰符；将变量声明为程序的输入变量。

2.3.6 输入变量

启动时，MQL5 中的所有程序都可以向用户查询参数。唯一例外是库，库无法独立执行，而是作为其他程序的一部分（参见相关章节以详细了解库）。

MQL 程序的输入参数是在代码中描述的全局变量，这些变量具有一个特殊修饰符 **input** 或 **sinput**。用户可以在程序特性对话框使用它们输入值。我们在第一章的脚本中看到了 **GreetingHour** 输入变量的说明。

输入变量的一个特殊特性是它们的值不能在程序代码中更改，即其表现与常量相似。

输入变量只能是简单的内置或枚举类型。对于枚举，通过下拉列表输入值；在所有其他情况下则使用输入字段输入值。不允许将以下类型描述为 **input**：数组、结构体或联合体以及类。

开发人员可以设置不同于变量标识符的输入参数名称。该名称将在程序特性对话框中显示给用户。定义输入参数时，应添加详细说明（单行字符串注释）。

```
input int HourStart = 0; // Start of trading (hour, including):
input int HourStop = 0; // End of trading (hour, excluding):
```

这会让界面对用户更友好、详细，并且不受 MQL5 对标识符的语法约束。此外，名称（以及注释）可以是您的母语。

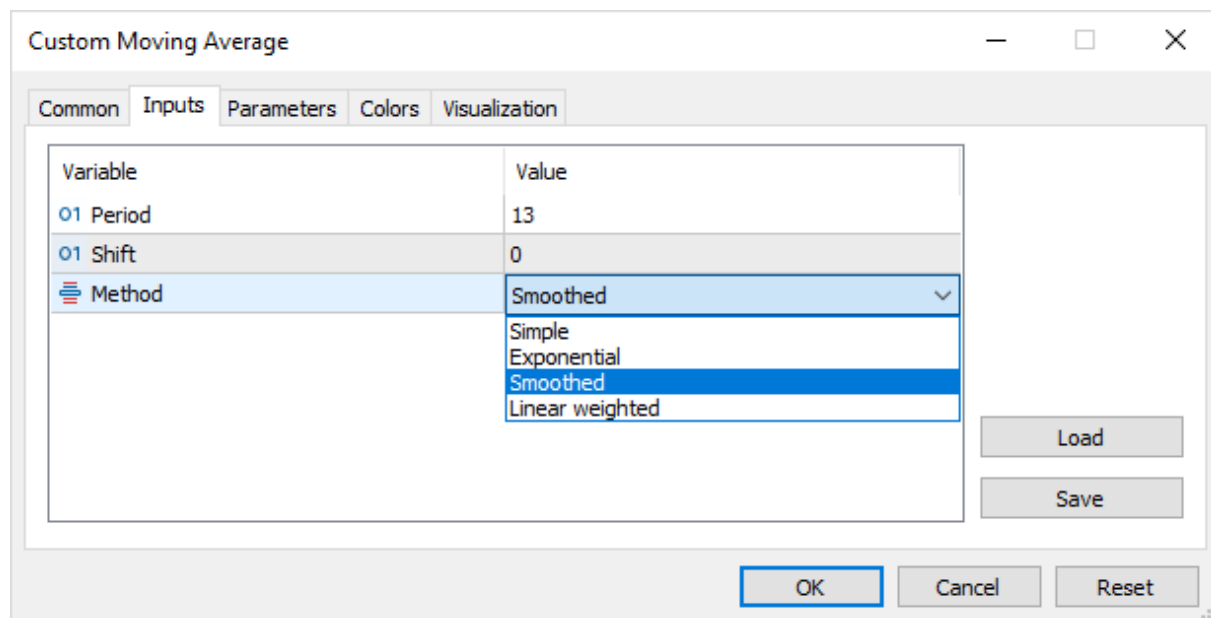
比如，MetaTrader 5 自带 **MQL5/Indicators/Examples/Custom Moving Average.mq5** 指标源代码，其中包含以下输入变量：

```

input int          InpMAPeriod = 13;          // Period
input int          InpMAShift  = 0;          // Shift
input ENUM_MA_METHOD InpMAMethod = MODE_SMMMA; // Method

```

该说明会生成以下特性对话框。



MQL 程序特性对话框示例

输入变量的文本表示 (格式为“标识符 = 值”对), 其最大长度 (包括字符 “=”) 不得超过 255 个字符 (该约束由终端和测试代理的内部数据交换协议施加)。这个限制对于字符串变量尤其重要, 因为其他类型的值长度永远不会超过字符串变量。我们知道, 标识符长度被限制为 63 个字符; 因此, 根据标识符长度, 输入字符串变量的值可以占用 191 至 253 个字符。超过组合阈值 (255 个字符) 的完整文本在传输到测试程序时可能会被截断。如果需要在 MQL 程序中输入一个更长的字符串, 请使用多个输入字段 (后面会介绍) 或允许用户指定文件的名称, 编译器从这个文件中读取文本。

为了方便操作 MQL 程序, 可以使用 **group** 关键字将输入与命名代码块组合 (组字符串结尾无需分号)。

```

input group "group_name"
input type identifier = value;
...

```

对于组说明后 (直到另一个组的说明或文件结尾) 带有 **input** 修饰符的所有变量, 在 MQL 程序的特性对话框中以嵌套列表形式显示在组标题下面。此外, 在适用于指标和 EA 的策略测试程序中, 通过鼠标点击可以展开或折叠参数组。

sinput 关键字是 **static input** 的缩写, 两种形式是等效的。

使用 **sinput** 和 **static input** 修饰符描述的变量不能进行优化。这两个修饰符只有在 EA 交易中使用时才具有意义, 因为 EA 交易是唯一支持优化的 MQL 程序类型。有关更多详细信息, 请参见涉及 [测试和优化 EA 交易](#) 的章节。

2.3.7 外部变量

本节的内容十分复杂，可以选读。本节要求读者了解基于 C++ 类别的概念，以及下文中讨论的内容。同时，所述语言结构的效果可以通过其他方式实现，而这种灵活性则成为了潜在错误来源。

SQL5 允许将变量描述为外部变量。使用 **extern** 关键字即可实现，并且仅允许在 [全局上下文中](#) 使用。

外部变量的语法基本与普通变量声明一致，但需额外添加 **extern** 关键字，且禁止初始化：

```
extern type identifier;
```

将变量描述为外部变量，意味着它的说明被延迟了，并且必须在源代码后面出现，通常是在另一个文件中（有关使用 [#include](#) 指令的[连接文件](#)将在[预处理器](#)相关章节中讨论）。多个不同的源文件可以具有同一个外部变量的说明，即具有相同类型和标识符的文件。所有这些说明都引用同一个变量。

假设该变量将在其中一个文件中完整描述。如果没有使用 **extern** 关键字在代码的任何位置定义变量，则返回编译错误“无法解析的外部变量”（这种情况下与 C++ 中的链接器错误类似）。

必须先描述外部变量，然后才能在特定文件的源代码中实际使用。也就是说，即使该变量并非在本模块中创建，也允许编译当前模块。

SQL5 中使用 **extern** 并不像在 C++ 中那样是强制要求，大多数情况下，可以通过以下方式来替代：启用一个头文件，其中包含要声明为 **extern** 的变量的一般说明。按照惯例进行这些定义就足够了。编译器确保每个附加文件仅被添加到源代码一次。鉴于在 SQL5 中，一个程序总始终包含一个可编译单元 `m q5`，因此不存在 C++ 中因在不同单元中启用了头文件导致变量定义重复的潜在错误。

即使在 [#include](#) 指令中附加了 `m q5`（并非 `m qh`）文件，也无法与启动编译的主单元公平竞争，而是被视为一个头文件。

与 C++ 不同，SQL5 不允许为外部变量指定初始值（C++ 中初始化会导致忽略 **extern**）。如果尝试设置一个初始值，会遇到编译错误“不允许外部变量初始化”。

总的来说，将变量描述为外部变量可以被认为是一种“软”说明：它确保了变量存在，并避免了在未使用 **extern** 修饰符的情况下，因在多个文件中描述变量而引发的覆盖错误。

然而，这也可能成为错误的来源。在不同的头文件中，如果出于不同目的碰巧描述了相同变量，那么在没有 **extern** 关键字的情况下，编译器可以识别名称冲突，而如果使用了 **extern**，则所有变量都成为了一个变量，程序运算逻辑很可能崩溃。

变量和函数都可以被描述为外部（这部分内容将在[下文](#)介绍）。对于函数来说，通过属性将它们描述为外部是一种遗留用法（即，虽然被编译了，但便会进行任何更改）。函数的以下两种声明方式是等效的：

```
extern return_type name([parameters]);  
return_type name([parameters]);
```

从这个意义上来说，是否使用 **extern** 仅用于风格上的区分：函数的前向说明是来自当前单元（不使用 **extern**）还是外部单元（使用 **extern**）。

可在待编译的 `m q5` 单元和待附加的头文件中使用 **extern**。

我们来分析 *extern* 的一些使用场景：它们写入在不同文件中，比如主脚本 *ExternMain.mq5* 和 3 个附加文件：*ExternHeader1.mqh*、*ExternHeader2.mqh* 和 *ExternCommon.mqh*。

在主文件中，只附加了 *ExternHeader1.mqh* 和 *ExternHeader2.mqh*，而稍后我们需要 *ExternCommon.mqh*。

```
// source code from mqh files will be substituted implicitly
// in the main mq5 file, instead of these directives
#include "ExternHeader1.mqh"
#include "ExternHeader2.mqh"
```

在头文件中，定义了两个条件性实用函数：在第一个头文件中，为 *x* 变量递增定义了 *inc* 函数，而在第二个头文件中，为 *x* 变量递减定义了 *dec* 函数。变量 *x* 在两个文件中都被描述为外部变量：

```
// ExternHeader1.mqh
extern int x;
void inc()
{
    x++;
}
// -----
// ExternHeader2.mqh
extern int x;
void dec()
{
    x--;
}
```

由于采用了这种说明，每个 *m qh* 文件都以常规方式编译。当它们一起包含在 *m q5* 文件中时，整个程序也能被编译。

如果在每个文件中定义变量时没有使用 *extern* 关键字，那么在编译整个程序时会发生“重新定义”错误。如果我们将 *x* 的定义从头文件转移到主单元，头文件就会停止编译（也许对某些人来说这不是问题；但在大型程序中，开发人员喜欢检查即时更正的编译能力，而不是编译整个项目）。

在主脚本中，我们定义了一个变量（在本例中，初始值为 2，如果我们没有指定值，将使用默认值 0），调用有条件的实用函数并打印 *x* 值。

```
int x = 2;

void OnStart()
{
    inc(); // uses x
    dec(); // uses x
    Print(x); // 2
    ...
}
```

short z 变量的说明（不使用 *extern*）包含在 *ExternHeader1.mqh* 文件中。主脚本中注释了一个类似的说明。如果我们激活这个字符串，将遇到之前提到的错误（“变量已定义”）。这样做是为了说明潜在的问题。

在 *ExternHeader1.mqh* 中, 还描述了 *extern long y*。与此同时, 在 *ExternHeader2.mqh* 文件中, 同名异义的外部变量具有另一种类型: *extern short y*。如果后面的说明没有被预先转移到注释中, 那么这里就会出现类型不兼容错误 (“变量 ‘y’ 已经用不同类型定义”)。总结: 类型必须一致或者变量不能是外部变量。如果这两个条件都无法满足, 这意味着其中一个变量的名称有拼写错误。

此外, 应该注意, 变量 *y* 没有显式初始化。但主脚本成功地调用了该变量, 并在日志中打印 0:

```
long y;

void OnStart()
{
    ...
    Print(y); // 0
}
```

最后, 脚本支持用户尝试替代外部双变量, 以已知变量 *x* 为例。不用描述 *extern int x*, *ExternHeader1.mqh* 和 *ExternHeader2.mqh* 文件都可以包含另一个公共头文件 *ExternCommon.mqh*, 其中包含 *int x* 的说明 (不使用 *extern*)。它作为项目中对 *x* 的唯一说明。

激活宏 `USE_INCLUDE_WORKAROUND` 时, 会启用这种组装程序的替代模式: 它位于脚本开头的注释中:

```
#define USE_INCLUDE_WORKAROUND // this string was in the comment
#include "ExternHeader1.mqh"
#include "ExternHeader2.mqh"
```

在这种配置中, 特定的 `include` 文件以及整个项目仍然是可编译的。在实际项目中, 如果不使用这种方法, 公共 `mqh` 文件将无条件地包含在 *ExternHeader1.mqh* 和 *ExternHeader2.mqh* 中 (无 `USE_INCLUDE_WORKAROUND` 条件)。在本例中, 指令的两个线程之间的切换是基于 `USE_INCLUDE_WORKAROUND` 宏, 该宏仅用于演示这两种模式。例如, *ExternHeader2.mqh* 的简化版本应如下所示:

```
// ExternHeader2.mqh
#include "ExternCommon.mqh" // int x; now here

void dec()
{
    x--;
}
```

在 `MetaEditor` 日志中, 我们可以发现 *ExternCommon.mqh* 文件仅被加载一次, 虽然它在 *ExternHeader1.mqh* 和 *ExternHeader2.mqh* 文件中都被引用。

```
'ExternMain.mq5'
'ExternHeader1.mqh'
'ExternCommon.mqh'
'ExternHeader2.mqh'
code generated
```

如果 *x* 变量在 *ExternCommon.mqh* 中 “注册”, 我们不得在主单元中重新定义它 (不使用 *extern*), 因为这会导致编译错误, 但我们可以在算法开头为该变量赋予所需的值。

2.4 数组

数组是用基于集群的方法存储和处理随机类型数据的工具。实际上，任何编程语言都支持数组。MQL5 中的数组特别重要，因为通过数组可以方便地整理与交易任务相关的串行数据。报价、指标读数、账户交易历史（包括订单和具体交易操作）以及资讯都是串行数据的例子，即随时间变化的值的序列。

数组可以被视为一个容器变量：它可以包含预定义数量的相同类型的值，这些值由它们的名称和索引（位置号）来标识。

在本节中，我们将以[嵌入式数据类型](#)为例，分析用于描述和调用数组的常用语法。在本书的后续章节中，随着对利用面向对象技术扩展类型系统的逐渐掌握，我们将这些与数组的使用结合，以获得新的方法。

2.4.1 数组特点

在介绍 MQL5 中声明数组的语法细节以及它们的应用实践之前，让我们来了解构造数组的一些基本概念。

数组的核心特点是维数。在一维数组中，元素是逐个放置的，就像一排士兵，只需一个数字（索引）就能引用元素。在给定历史深度开立的金融工具的逐根柱线价格便可以保存在这样的数组中。

在二维数组中，元素在两个逻辑上垂直的方向上发散，形成一个正方形（或者一般情况下为矩形），每个元素需要两个索引，即每个维度一个。这样的数组可以用来存储每个历史柱线的价格四方形（开盘价、最高价、最低价和收盘价）。柱线编号使用第一维度计数，而第二维度用于从 0 到 3 的数字，分别表示一种价格类型。

三维数组相当于一个有三个轴的立方体（或者更严格地说，是一个直角平行六面体）。继续逐根柱线价格数组的例子，我们可以给它添加第三个维度，该维度负责迭代来自市场报价的金融工具。

对于每个维度，数组都有固定的长度（大小），用于设定索引范围。如果计划载入 1,000 根柱线和 10 个金融工具的历史，我们将在数组的第一个维度中得到 1,000 个元素，在第二个维度（OHLC）中得到 4 个元素，在第三个维度中得到 10 个元素。

所有维度中大小的乘积即为数组元素总数；在我们的例子中，总数是 40,000。在 MQL5 中，总数不能超过 2147483647（int 的最大值）。

因为我们生活在一个三维世界中，所以很难想象一个 4 维数组的立体形状。但是 MQL5 允许创建最多包含四个维度的数组。

请注意，始终可以使用一维数组来代替具有随机维数（包括 4 维以上）的多维数组。只需要将多个索引重新计算，获得一个连续索引。例如，如果一个二维数组有 10 列（维度 1, X 轴）和 5 行（维度 2, Y 轴），则可以被转换为具有相同元素数量（即 50 个元素）的一维数组。在这种情况下，元素索引通过以下公式获得：

$$\text{index} = Y * N + X$$

此处，N 是第一个维度中元素的数量，在我们的例子中是 10；就是每行的大小；Y 是行号（0 到 4）；X 是列号（0 到 9）。

跨维度大小是区分数组和变量的另一个特征。因此，必须在说明中以某种方式指定维度的数量和每个维度的大小，以及数组名称和数据类型（参见[下一节](#)）。

请区分变量（数组元素）的大小（以字节为单位）和数组的大小（数组中元素的个数）。从理论上讲，就占用的内存而言，完整的数组大小必须是一个元素的大小（由数据类型决定）与元素个数的乘积。然而，这一公式在实践中不一定总是有效。特别是，由于字符串的长度各不相同，因此很难计算一个字符串数组占用的内存量。

根据内存分配方法，数组可以分为动态数组和定长数组。

代码中描述的定长数组在所有维度上都具有精确的大小。这些大小之后无法改变。然而，实际任务中经常会出现待处理的数据量不确定的情况，因此，需要在算法操作期间改变数组的大小。动态数组就是针对这个特定用途而出现的。接下来我们会看到，动态数组在声明时没有指定第一个维度的大小，然后使用特殊的 MQL5 API 函数进行“扩展”或“压缩”。

MQL5 文档中使用了一个有歧义的术语，将定长数组称为静态。这个概念也适用于 'static' 修饰符，该修饰符可应用于数组。这样的数组被声明为动态时，在内存分配方面不是静态的，而在 'static' 修饰符方面则是静态的。为了排除这种歧义，本书中的静态字符仅代表声明属性。

除了动态数组和定长数组，MQL5 中还有一些特殊数组，用于存储报价和技术指标缓冲区。这类数组称为时间序列数组，因为它们的索引与时间相对应。实际上，这些数组是动态的一维数组。然而，与其他动态数组不同，终端本身会为它们分配内存。我们将在涉及 [时间序列](#) 和 [指标](#) 的相关章节中介绍。

2.4.2 数组说明

数组说明继承了变量说明的一些特性。首先，我们应该注意，数组可以是全局的，也可以是局部的，取决于声明它们的位置。与变量类似，**const** 和 **static** 修饰符也可以用于描述数组。对于一维定长数组，声明语法如下所示：

```
type static1D[size];
```

此处，**type** 和 **static1D** 分别表示元素的类型名称和数组标识符，而方括号中的 **size** 是定义大小的整数常量。

对于多维数组，必须根据维度数量指定多个大小：

```
type static2D[size1][size2];
type static3D[size1][size2][size3];
type static4D[size1][size2][size3][size4];
```

动态数组以类似的方式描述，只是跳过了第一个方括号（在使用这种数组之前，必须使用 **ArrayResize** 函数为它分配所需的内存大小，参见涉及 [动态数组](#) 的章节）。

```
type dynamic1D[];
type dynamic2D[][size2];
type dynamic3D[][size2][size3];
type dynamic4D[][size2][size3][size4];
```

对于定长数组，允许初始化：在等号后面为各元素指定初始值，构成一个以逗号分隔的列表，整个列表用花括号括起来。例如：

```
int array1D[3] = {10, 20, 30};
```

此处，大小为 3 的整数数组取值 10、20 和 30。

使用初始化列表，就不需要在方括号中指定数组大小（仅限一维数组）。编译器将根据列表长度自动评估大小。例如：


```
int array1D[] = {10, 20, 30};
```

初始值可以是常量也可以是常量表达式，即编译器在编译时可以计算的公式。例如，下列数组由一分钟、一小时、一天、一周中的秒数构成（用公式表示比直接用 86400 或 604800 更直观）：

```
int seconds[] = {60, 60 * 60, 60 * 60 * 24, 60 * 60 * 24 * 7};
```

这些值通常被设计为代码开头的预处理器宏，随后可以将这个宏的名称插入文本中任何需要的位置。此方法在 [预处理器](#) 的相关章节中进行了说明。

初始化元素的数量不能超过数组大小。否则，编译器会显示错误消息“初始化器过多”。如果值数量小于数组大小，则剩余元素被初始化为零。因此，提供了一个将整个数组初始化为零的简单语句：

```
int array2D[2][3] = {0};
```

或者只使用一对花括号：

```
int array2D[2][3] = {};
```

不管是几维数组，这种方法都有效。

要初始化多维数组，必须嵌套列表。例如：

```
int array2D[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

此处，数组的第一个维度大小是 3；那么就用两个逗号分隔 3 个元素，并用花括号括起这三个元素。但是，由于数组是二维的，它的每个元素都是一个数组，大小都是 2。也就是每个元素用带花括号的列表表示，每个列表包含 2 个值。

假设，我们需要一个转置后的数组（第一个维度大小为 2，第二个维度大小为 3），那么数组的初始化会发生变化：

```
int array2D[2][3] = {{1, 3, 5}, {2, 4, 6}};
```

如果必要，我们可以跳过初始化列表中的一个或多个值，用逗号标记它们的位置。所有被跳过的元素也将被初始化为零。

```
int array1D[3] = {, , 30};
```

此处，第一个元素将等于 0。

语言的语法允许在最后一个元素后输入一个逗号：

```
string messages[] =
{
    "undefined",
    "success",
    "error",
};
```

这简化了添加新元素的步骤，尤其是对于多字符串条目。特别是，如果我们忘记在字符串数组中新增元素前面输入逗号，那么新旧字符串将会合并为一个元素（具有相同索引），而不会生成新元素。此外，一些数组可以自动生成（借助其他程序或宏）。因此，自然所有元素都具有统一外观。

“堆”和“栈”

对于可能很大的数组，区分内存中的全局位置和局部位置非常重要。

全局变量和数组占用的内存是在“堆”内部分配的，即程序可用的空闲内存。除了计算机物理特性和操作系统之外，这种内存实际上不受任何限制。“堆”的名称可以这样解释：程序不停在分配或释放不同大小的内存区域，这导致空闲区域随机分布在整个块中。

局部变量和数组位于栈中，即预先为程序分配的有限内存区域，尤其用于存储局部元素。栈的名称来源于这样一个事实：在算法执行期间发生函数的嵌套调用，这些调用根据“堆叠”原则累积其内部数据：例如，终端调用 *OnStart*，*OnStart* 调用应用代码中的另一个函数，这个函数又会调用另一个函数，以此类推。与此同时，每个函数时，都会创建其局部变量，在调用嵌套函数期间，这些变量一直存在。嵌套函数也会创建局部变量，这些局部变量在栈中的位置比此前创建的变量要高一些。结果是，栈通常包含多层局部数据，这些数据来自于到达当前代码字符串所经过的路径上激活的所有函数。直到位于栈顶部的函数执行完成，它的局部数据才会从此处移除。一般来说，栈是一种基于 *FIFO/LIFO*（先进后出，后进先出）原则的存储。

由于栈的大小有限，建议只在其中创建局部变量。但是，数组可能非常大，很快就会耗光栈空间。与此同时，程序执行完成时出现错误。因此，我们应该将全局级别的数组描述为静态的（*static*）或者动态地为这些数组分配内存（在堆上分配）。

2.4.3 使用数组

使用类似的语法在数组元素中读写值，用在方括号指定所需的索引。为了给元素赋值，我们要使用 **赋值运算** '='。例如，要替换一维数组的第 0 个元素的值：

```
array1D[0] = 11;
```

索引从 0 开始。最后一个元素的索引等于元素的个数减 1。当然，我们可以使用常量和任何其他可以简化为整数类型的表达式作为索引（有关表达式的更多详细信息，参见 [下一章](#)），如整数变量、函数调用或另一个整数数组的元素（这种方式称为间接寻址）。

```
int index;
// ...
// index = ... // assign an index somehow
// ...
array1D[index] = 11;
```

对于多维数组，必须为所有维度指定索引。

```
array2D[index1][index2] = 12;
```

用于索引时，允许的整数类型不包括 *long* 和 *ulong*。如果我们尝试使用“长整型”的值作为索引，它将被隐式转换为 *int*，因此编译器给出警告“类型转换可能会导致数据丢失。”

根据相同原则对数组元素进行读取访问。例如，以下是在日志中打印数组元素的命令：

```
Print(array2D[1][2]);
```

在 *GoodTimes* 脚本中，我们看到了带问候语字符串（包含在 *Greeting* 函数中）的本地静态数组 *messages* 的描述，以及该数组元素在 *return* 运算符中的使用。

```

string Greeting()
{
    static int counter = 0;
    static const string messages[3] = // description
    {
        "Good morning", "Good day", "Good evening" // initialization
    };
    return messages[counter++ % 3];    // using
}

```

执行 `return` 时，我们读取用表达式定义索引的元素：`counter++ % 3`。模 3 除法（用 `%` 表示）确保递增 1 的 `counter` 被限定在索引的正确值范围内：0、1 或 2。如果不用模除，则从第四次调用该函数开始，所请求元素的索引将超过数组大小。在这种情况下，会出现程序执行时错误（“数组超出范围”），并将其从图表中卸载。

MySQL API 包括用于许多数组操作的通用函数：分配内存（仅限动态数组）、填充、复制、排序和在数组中搜索都将在 [处理数组](#) 一节中介绍。不过现在我们就展示其中一个：`ArrayPrint` 允许以方便的格式打印日志中的数组元素（考虑维度）。

`Arrays.mq5` 脚本演示了一些描述数组的例子，其结果在日志中打印。在学习了循环和表达式之后，我们将讨论数组元素的操作。

```

void OnStart()
{
    char array[100];           // without initialization
    int array2D[3][2] =
    {
        {1, 2},               // illustrative formatting
        {3, 4},
        {5, 6}
    };
    int array2Dt[2][3] =
    {
        {1, 3, 5},
        {2, 4, 6}
    };
    ENUM_APPLIED_PRICE prices[] =
    {
        PRICE_OPEN, PRICE_HIGH, PRICE_LOW, PRICE_CLOSE
    };
    // double d[5] = {1, 2, 3, 4, 5, 6}; // error: too many initializers
    ArrayPrint(array);         // printing random "garbage" values
    ArrayPrint(array2D);       // showing the 2D array in the log
    ArrayPrint(array2Dt);      // a "transposed" appearance of the same data 2D
    ArrayPrint(prices);        // getting to know the values of the price enumeration element
}

```

其中一个日志条目选项如下所示。

```

[ 0]  0  0  0  0  0  0  0  0  0  0  0  0  0 -87 105 82 119  0
      0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
[34]  0  0  0 -32 -3 -1 -1  7  0  0  2  0  0  0  0  0  0
      0  2  0  0  0  0  0  0  0  0 -96 104 82 119  0  0  0  0
[68]  0  0  3  0  0  0  0  0  0 -1 -1 -1 -1  0  0  0  0 100
      48  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
      [,0][,1]
[0,]  1  2
[1,]  3  4
[2,]  5  6
      [,0][,1][,2]
[0,]  1  3  5
[1,]  2  4  6
2 3 4 1

```

名为 **array** 的数组未进行初始化，因此为其分配的内存可能包含随机值。每次脚本运行时，这些值都会发生变化。建议始终初始化局部数组，以避免出错。

array2D 和 **array2Dt** 数组以直观的矩阵形式在日志中打印。这与我们在源代码中以同样方式格式化初始化列表的做法完全无关。

prices 数组具有嵌入式枚举类型 `ENUM_APPLIED_PRICE`。基本上，数组可以是任何类型，包括结构、函数指针以及其他我们将要分析的类型。由于枚举类型是基于 **int** 类型，所以值是按数字显示，而不是按元素名称显示（要获得枚举的特定元素的名称，可以使用 [EnumToString](#) 函数，但 [ArrayPrint](#) 中不支持此模式）。

包含 **d** 数组说明的字符串中存在一处错误：初始值的实体超过了数组大小。

2.5 表达式

表达式是任何编程语言的基本元素。无论算法背后的应用思想是什么，最终都要归结为数据处理，也就是计算。表达式描述了一个或多个预定义的值计算结果的方式。这些值称为操作数，而对它们执行的操作由操作或运算符表示。

作为允许对操作数进行操作的运算符，在表达式中使用了独立字符或这些字符的序列，例如 '+' 表示加法，'*' 表示乘法。运算符分为几组，比如算术、按位、比较、逻辑，还有一些专门的运算符。

我们已经在本书前面的章节中使用过表达式，比如初始化变量。在最简单的情况下，表达式是一个常量（字面量），是唯一的操作数，而计算结果等于操作数的值。但操作数也可以是变量、数组元素、函数调用结果（直接从表达式中调用函数以获得结果）、嵌套表达式和其他实体。

所有运算符都将它们的结果返回到父表达式中，直接替换操作数，这使得运算符可以组合，形成非常复杂的层次结构。例如，在下面的表达式中，变量 **b** 乘以变量 **c** 的结果与变量 **a** 的值相加，然后得到的值将被存储在变量 **v** 中：

```
v = a + b * c;
```

在本节中，我们讨论构造和计算各种表达式的一般原则，以及 MQL5 中针对内置类型支持的标准运算符集合。稍后，在涉及 OOP 的章节中，我们将了解如何为自定义类型（即结构和类）重新加载（重新定义）运算符，这样我们便可以在表达式中使用对象，并对它们执行非标准操作。

2.5.1 基本概念

在讨论特定运算符组之前，我们应该介绍一些基本概念，这些是所有运算符的内在概念，并将影响它们在特定上下文中的适用性和行为。

首先，根据所需操作数的数量，运算符可以分为“一元”和“二元”。顾名思义，一元运算符处理一个操作数，而二元运算符处理两个操作数。在二元的情况下，运算符总是放在操作数之间。在一元运算符中，有些运算符必须放在操作数之前，有些则必须放在操作数之后。例如，一元减号 (-) 运算符允许对值的符号取反：

```
int x = 10;
int y = -x; // -10
```

同时，还有一个使用同一个字符 - 的二元运算符，用于减法运算。

```
int z = x - y; // 10 - -10 -> 20
```

编译器能否在特定上下文中选择正确运算符 (操作) 取决于在表达式中使用该运算符的上下文。

每个运算符都被分配了优先级。优先级决定了在有多个运算符的复杂表达式中计算运算符的顺序。最先计算优先级最高的运算符，最后计算优先级最低的运算符。例如，在表达式 $1 + 2 * 3$ 中，有两个运算 (加法和乘法) 和三个操作数。由于乘法的优先级高于加法，所以会先计算 $2 * 3$ 的乘积，然后再加 1。

稍后，我们将提供一份完整的运算表 (带优先级)。

此外，每个运算符都具有结合性特征。结合性可以是左或右，决定了具有相同优先级的连续运算符的执行顺序。例如，表达式 $10 - 7 - 1$ 理论上可以用两种方法计算：

- 从 10 中减去 7，然后从得到的 3 中减去 1，最后得到 2；或者
- 7 减 1 得 6，10 减 6 得 4。

在第一种情况下，计算是从左到右执行的，这对应于左结合性；由于减法运算具有左结合性，第一个答案是正确的。

第二种计算方式对应于右结合性，不会被使用。

我们来考虑另一个同时涉及优先级和结合性的例子： $11 + 5 * 4 / 2 + 3$ 。两种类型的运算 (即加法和乘法) 都是从左向右执行的。如果优先级没有区别，我们将得到 35，但正确答案是 24。改变右边的结合性将得到 14。

要显式重新定义表达式中的优先级，可以使用括号，例如： $(11 + 5) * 4 / (2 + 3)$ 。括号中的内容是之前计算的，中间结果被替换到表达式中用于其他运算。圆括号中的组可以嵌套。有关更多详细信息，请参见 [用圆括号分组](#) 一节。

右结合运算符可以用逻辑否定一元运算符 `!` 来举例说明。本质上，它的任务是将 `false` 变为 `true`，或者反向转换。类似其他一元运算符，结合性在此上下文中意味着操作数必须放置在运算符的哪一边。`!` 符号放在操作数之前，即操作数在右边。

```
int x = 10;
int on_off = !!x; // 1
```

在这种情况下，逻辑否定被执行两次：第一次对变量 `x` (右侧 `!`)，第二次针对前面的否定结果 (左侧 `!`)。这种双重否定通过先转换为 `bool` 类型再反向转换的方式，可将任何非零值转换为 1。

最终的运算表也将显示结合性。

最后需要注意的是，处理表达式时，计算操作数的顺序也很重要。请勿将其与属于运算符（而非操作数）的优先级混淆。二元运算的操作数的计算顺序未明确定义，这给了编译器优化代码和提高效率的空间。编译器只保证会在执行运算之前计算操作数。

仅有有限的运算符定义了操作数求值顺序。特别地，对于逻辑与（&&）和或（||），运算顺序是从左到右的，若左侧操作数的值已能确定最终结果，右侧操作数可以被省略。但对于三元条件运算符`?:`来说，运算顺序更加复杂：首先计算第一个条件，然后根据结果为真或假计算两个分支中的其中一个分支。更多详细信息，参见后续章节。

操作数的求值顺序取可以通过表达式中包含多个函数调用的情况来说明。例如，假设表达式中使用了 4 个函数：

```
a() + b() * c() - d()
```

优先级和结合性规则仅适用于调用这些函数的中间结果，而函数调用本身的顺序可由编译器根据源代码特性和编译器设置，以其“认为有必要”的任意顺序生成。例如，乘法中涉及到的函数 `b` 和 `c` 可能是按照 `[b(), c()]` 的顺序调用的，也可能是按相反顺序 `[c(), b()]`。如果这些函数在执行过程会影响相同数据，则表达式计算后的状态无法确定。

使用数组和递增运算符时，也会出现类似的问题（参见[递增和递减](#)）。

```
int i = 0;
int a[5] = {0, 1, 2, 3, 4};
int w = a[++i] - a[++i];
```

根据是左侧还是右侧求差操作数最先被计算，我们可以得到 `-1` (`a[1] - a[2]`) 或者 `+1` (`a[2] - a[1]`)。由于 MQL5 编译器一直在改进，所以不能保证当前的结果（-1）在未来版本中保持不变。

为了避免潜在的问题，如果某个操作数已经在同一个表达式中被修改，建议不要重复使用该操作数。

在任何表达式中，通常都会包含不同类型的操作数。这导致在对操作数执行任何操作之前，需要将其转换为某种常见类型。如果没有显式类型转换，必要时 MQL5 会执行隐式转换。此外，不同类型组合的转换规则也不相同。显式和隐式类型转换将在[相关章节](#)中介绍。

2.5.2 赋值运算

表达式计算结果通常必须存储在某个位置。在编程语言中，由 `=` 表示的赋值运算符就是为此目的而设计的。变量或数组元素的名称置于运算符左侧（用于存储运算结果），而表达式（实际上是计算公式）置于右侧。

我们已经使用这个运算符来初始化变量，仅在创建变量的过程中执行过一次初始化。但赋值允许在算法过程中任意次数地更改变量值。例如：

```
int z;
int x = 1, y = 2;
z = x;
x = y;
y = z;
```

变量 `x` 和 `y` 初始化为值 1 和 2，随后用辅助的第三个变量 `z` 和三次赋值来交换 `x` 和 `y` 值。

与所有运算符一样，赋值运算符也会将运算结果返回表达式。这样便能够串联书写赋值语句。

```
int x, y, z;
x = y = z = 1;
```

此处，1 首先被赋给变量 **z**，然后赋给变量 **y**，最后赋给变量 **x**。显然，这个运算符具有右结合性，因为被赋予的值在表达式中从右向左移动。

我们可以将赋值作为表达式的一部分。但是，由于其优先级低于所有其他运算符（除了“短号”以外，参见 [运算优先级](#)），必须用圆括号括起来（有关详细信息，请参见 [用圆括号分组](#) 一节）。这可能会导致以下情况：表达式中出现拼写错误（比如 `==` 误写为 `=`），导致无法按预期执行语句。此类行为的示例，请参见 [语句 if](#) 的相关章节。

赋值运算符对 `=` 左右两侧的实体施加了某些限制。在编程中，为了简化存储规则，这些实体的确切名称为：左值 (LValue) 和右值 (RValue)（左 (Left) 和右 (Right) 的英文单词首字母）。

左值和右值

左值代表一个实体，为其分配了内存，因此可以在其中写入值。左值的典型例子包括变量和数组元素。在学习了面向对象程序设计之后，我们将认识这一类别的另一个代表性实体：对象，对象中可以重新加载赋值运算符。左值的一个必要元素是标识符。

需要注意，变量和数组可以用 `const` 关键字描述，因此不能充当左值，因为常量禁止被修改。

右值是在表达式中使用的临时值，例如字面量、函数调的返回值或表达式片段的计算结果。

“左值”类别具有扩展性，即属于该类别的实体不仅可以放置在 `=` 左侧，而且不排斥将其与右值一样放置在 `=` 的右边。

“右值”类别则具有局限性，即任何右值都只能放置在 `=` 的右边。

当某个左值元素在 `=` 右侧使用时，其标识符实际上表示表达式公式中存储的当前内容。

但是，如果左值元素在 `=` 左侧使用，其标识符表示应写入新值（表达式计算结果）的内存地址（内存单元）。

不同运算符对于是否可用于左值或右值的操作数具有不同限制。例如，递增 `++` 和递减 `--` 运算符（参见 [递增和递减](#)）只能与左值一起使用。

以下是赋值运算符的允许和禁止操作的示例（[ExprAssign.mq5](#) 脚本）：


```
// description of variables
const double cx = 123.0;
int x, y, a[5] = {1};
string s;
// assignment
a[2] = 21;          // ok
x = a[0] + a[1] + a[2]; // ok
s = Symbol();      // ok
cx = 0;            // const variable may not be changed
                    // error: 'cx' - constant cannot be modified
5 = y;             // 5 - this number (literal)
                    // error: '5' - l-value required
x + y = 3;         // to the left of RValue (expression computation result)
                    // error: l-value required
Symbol() = "GBPUSD"; // to the left of RValue with the function call result
                    // error: l-value required
```

编译器返回一个错误，指出违反了运算符使用规则。

2.5.3 算术运算

算术运算包括 5 个二进制运算，即加、减、乘、除和模除，以及 2 个一元运算，即正号和负号。下表给出了每种运算使用的符号。

在包含示例的列中，**e1** 和 **e2** 是任意子表达式。用左（从左向右）和右（从右向左）标记结合性。第一列中的数字可以被视为执行运算的优先级。

优先级	符号	说明	示例	结合性
2	+	一元加法	+e1	右
2	-	一元减法	-e1	右
3	*	乘法	e1 * e2	左
3	/	除法	e1 / e2	左
3	%	模除法	e1 % e2	左
4	+	加法	e1 + e2	左
4	-	减法	e1 - e2	左

表中的顺序为降序优先级：一元加法和减法优先于乘法和除法，而乘法和除法优先于加法和减法。

```
double a = 3 + 4 * 5; // a = 23
```

事实上，一元加法在计算中没有任何作用，但可以用于更好地可视化表达式。一元减法可反转其操作数的符号。

算术运算用于数字类型或可以强制转换为数字类型的类型。计算结果是一个右值。在计算中，整数操作数的存储位置通常会扩展到所用整数中“最大值”或扩展到 `int`（如果所有整数类型都更小），并被强制转换为一种通用类型。有关更多详细信息，请参见 [类型强制转换](#)。

```
bool b1 = true;
bool b2 = -b1;
```

在本示例中，`b1` 变量“扩展”至值为 1 的 `int` 类型。反转符号得 -1，而类型强制转换为 `bool` 再反转得到 `true`（因为 -1 非零）。不建议在算术计算中使用逻辑类型。

整数相除结果为整数，即小数部分（如有）被忽略。可以使用 `ExprArithmetic.mq5` 脚本验证这一点。

```
int a = 24 / 7;      // ok: a = 3
int b = 24 / 8;      // ok: b = 3
double c = 24 / 7;   // ok: c = 3 (!)
```

虽然 `c` 变量被描述为 `double`，但是该变量的初始化表达式中包含整数；因此，执行的是整数除法。要执行带小数部分的除法，必须至少有一个操作数是实数类型（第二个操作数也将被强制转换为实数类型）。

```
double d = 24.0 / 7; // ok: d = 3.4285714285714284
```

% 运算符计算整数除法的余数（仅适用于两个整数类型的操作数）。

```
int x = 11 % 5;    // ok: x = 1
int y = 11 % 5.0;  // no real number can be used
                  // error: '%' - illegal operation use
```

操作数具有不同符号时，使用 `*` 和 `/` 运算符的结果为负数。% 运算符遵循以下规则：

- 如果 % 运算符的除数为负，则符号“转义”；
- 如果 % 运算符的被除数为负，则结果为负；

使用模除法的替代计算可以轻松验证这一点： $m \% n = m - m / n * n$ 。切记，整数除法 m / n 的结果将被舍入；所以一般情况下， $m / n * n$ 不等于 m 。

在 [数组特性](#) 一节中，我们深入研究了多维数组可以用一维数组来表示的思想（通过重新计算数组元素的索引）。我们还提供了通过二维数组的坐标（列号为 X ，行号为 Y ，字符串长度为 N ）获得一维数组的查询索引的公式。

```
index = Y * N + X
```

% 运算允许我们执行更便捷的反向计算，即通过索引穿透法求解 X 和 Y ：

```
Y = index / N
X = index % N
```

如果在计算表达式期间的某个阶段获得一个不可表示的结果 `NaN`（非数字，如无穷大、负数的平方根等），所有后续的运算也会产生一个 `NaN`。可以使用 `MathIsValidNumber` 函数判定结果是否为正常数值（参见 [数学函数](#)）。

```
double z = DBL_MAX / DBL_MIN - 1; // inf: Not A Number
```

此处是对 `NaN`（除法运算的结果）进行了减法运算，结果再次得到 `NaN`。

为字符串定义加法运算，并执行连接，即组合字符串。

```
string s = "Hello, " + "world!"; // "Hello, World!"
```

禁止对字符串进行其他运算。

2.5.4 递增和递减

递增和递减运算符允许以简化方式编写用于将操作数加/减 1 的表达式。这两个运算符大多在循环结构内部，用于在访问数组或其他支持枚举的对象时修改索引。

递增由两个连续的加号表示：'++'。递减由两个连续的减号表示：'--'。

这种运算符有两种类型：前缀和后缀。

前缀运算符，顾名思义，位于操作数之前（++x,--x）。前缀运算符更改操作数的值，而这个新值将参与表达式的后续计算。

后缀运算符位于操作数之后（x++,x--）。后缀运算符在表达式中替换当前操作数值的副本，然后更改其值（新值不会进入表达式）。ExprIncDec.mq5 脚本中给出了几个简单的例子。

```
int i = 0, j;
j = ++i;      // j = 1, i = 1
j = i++;      // j = 1, i = 2
```

对于需同时引用操作数原值并进行副作用修改的表达式，后缀形式有助于编写更紧凑的表达式（若采用其他书写方法则需要两条单独的语句）。在所有其他情况下，建议使用前缀形式（前缀形式不会创建“旧”值的临时副本）。

在下面的示例中，符号在数组元素中连续反转，直到找到第零个元素。通过后缀递增运算符 k++（包含在循环结构 while 中）可确保遍历数组索引。根据这个后缀，a[k++] = -a[k] 表达式首先更新第 k 个元素，然后将 k 加 1。然后检查赋值结果是否不等于零（!= 0，参见下一节）。

```
int k = 0;
int a[] = {1, 2, 3, 0, 5};
while((a[k++] = -a[k]) != 0){}
// a[] = {-1, -2, -3, 0, 5};
```

下表按优先级顺序显示了递增和递减运算符：

优先级	符号	说明	示例	结合性
1	++	后缀递增	e1++	左
1	--	后缀递减	e1--	左
2	++	前缀递增	++e1	右
2	--	前缀递减	--e1	右

所有递增和递减运算的优先级都高于算术运算。前缀的优先级低于后缀。在下面的例子中，x 的“旧”值与 y 值相加，随后对 x 进行递增。如果前缀优先级较高，将对 y 执行递增，此后将 x 与新值 6 相加，我们将得到 z = 6, x = 0（之前的值）。

```
int x = 0, y = 5;
int z = x+++y; // "x++ + y" : z = 5, x = 1
```

2.5.5 比较运算

顾名思义，这类运算旨在对两个操作数进行比较，并根据比较条件是否成立而返回逻辑特性值 *true* 或 *false*。

下表给出了所有比较运算及其特性，如使用的符号、优先级、示例和结合性。

优先级	符号	说明	示例	结合性
6	<	小于	<code>e1 < e2</code>	左
6	>	大于	<code>e1 > e2</code>	左
6	<=	小于等于	<code>e1 <= e2</code>	左
6	>=	大于等于	<code>e1 >= e2</code>	左
7	==	等于	<code>e1 == e2</code>	左
7	!=	不等于	<code>e1 != e2</code>	左

每种运算的原理是使用列（包含列的说明）中的标准对两个操作数进行比较。例如，`x < y` 条目表示检查 “*x* 小于 *y*” 是否成立。相应的，如果 *x* 确实小于 *y*，那么比较结果为 *true*，否则为 *false*。

可对任何类型的操作数进行比较（对于不同类型，会进行 [类型强制转换](#)）。

考虑到左结合性且返回结果是 *bool* 类型，构造一系列比较可能不会取得明显效果。例如，假设有一个表达式用于检查 *y* 值是否介于 *x* 和 *z* 的值之间，如下所示：

```
int x = 10, y = 5, z = 2;
bool range = x < y < z;    // true (!)
```

但是，这样的表达式会以另外一种方式处理。甚至编译器会对此进行标记并发出警告：“运算中未安全使用 `bool` 类型”。

根据左结合性，首先检查左侧条件 `x < y`，其结果会作为 *bool* 类型的临时值替换到后续表达式中，从而得到：`b < z`。然后将 *z* 的值与临时变量 *b* 中的 *true* 或 *false* 进行比较。要检查 *y* 的范围是否介于 *x* 和 *z* 之间，需使用两个比较运算并要结合逻辑运算“与”（将在 [下一节](#) 中介绍）。

```
int x = 10, y = 5, z = 2;
bool range = x < y && y < z;    // false
```

进行“等于/不等于”比较时，应考虑操作数类型的特征。例如，浮点数在计算后通常包含“近似”值（我们之前介绍过 *double* 和 *float* 的精度，详见 [实数](#) 一节）。例如，严格来说，0.6 与 0.3 之和不等于 0.9：

```
double p = 0.3, q = 0.6;
bool eq = p + q == 0.9;    // false
double diff = p + q - 0.9;    // -0.000000000000000111
```

差值是 1×10^{-16} ，但如此小的差值也足以让比较运算返回 *false*。

因此，比较实数的“等于/不等于”关系时，应基于其差值与可接受偏差，通过“大于”/“小于”运算符完成，这种偏差可根据计算特性手动确定，也可采用通用偏差。回顾一下，对于 *double* 和 *float* 类型，系统定义了嵌入式精度常量 `DBL_EPSILON` 和 `FLT_EPSILON`，这两个常量适用于值

1.0。要与其他值进行比较，必须对这两个常量进行缩放。在 *ExprRelational.mq5* 脚本中，给出了 *isEqual* 函数的一种可能实现来进行实数比较，其中就考虑了这一点。

```
bool isEqual(const double x, const double y)
{
    const double diff = MathAbs(x - y);
    const double eps = MathMax(MathAbs(x), MathAbs(y)) * DBL_EPSILON;
    return diff < eps;
}
```

这里我们使用获得无符号绝对值的函数 (*MathAbs*) 和取两个值中较大值的函数 (*MathMax*)。将在第四章的 [数学函数](#) 一节中介绍这些内容。使用 `<` 运算，将 *isEqual* 函数参数之间的绝对差值与 *eps* 变量中的校准公差进行比较。

切勿使用此函数与绝对零进行比较。要与绝对零进行比较，可以采用以下方法 (可能根据你的具体需求进行一定调整)：

```
bool isZero(const double x)
{
    return MathAbs(x) < DBL_EPSILON;
}
```

按字典顺序比较字符串，即逐个字母进行比较。将每个字符的代码与第二个字符串中相同位置字符的代码进行比较。一直进行比较，直到发现代码中的差异或其中一个字符串结束。字符串大小关系等于第一个出现的不同字符的大小关系，若出现的字符都相同，则认为较长的字符串大于较短的字符串。切记，同一个字母的大小格式对应的代码不同，表示大写字母的代码小于表示小写字母的代码 (很奇怪吧)。

空字符串 "" (实际上存储一个终止符 0) 不等于特殊值 NULL，后者表示没有字符串。

```
bool cmp1 = "abcdef" > "abs";      // false, [2]: 's' > 'c'
bool cmp2 = "abcdef" > "abc";      // true, by length
bool cmp3 = "ABCdef" > "abcdef";   // false, by case
bool cmp4 = "" == NULL;            // false
```

此外，MQL5 还提供了一些用于比较字符串的函数，将在 [处理字符串](#) 一节中介绍这些函数。

在进行“等于/不等于”比较时，不建议使用 *bool* 常量：*true* 或 *false*。这是因为，在类似 *v == true* 或 *v == false* 的表达式中，*v* 操作数可以被直观的解释为逻辑类型，而实际上它是一个数字。我们知道，数字中的零逻辑上被视为 *false*，而所有其他值都被视为 *true* (我们经常借此来表示某个结果存在或不存在)。但在这种情况下，类型强制转换会反向：*true* 或 *false* 被“扩展”为数字类型 *v*，实际上分别等于 1 和 0。这种比较的结果将不同于预期 (例如，*100 == true* 的比较结果为假)。

2.5.6 逻辑运算

逻辑运算对逻辑操作数执行计算，并返回相同类型的结果。

优先级	符号	说明	示例	结合性
2	!	逻辑“非”	!e1	右
11	&&	逻辑“与”	e1 && e2	左
12		逻辑“或”	e1 e2	左

逻辑“非”将 *true* 转换为 *false* 并将 *false* 转换为 *true*。

如果两个操作数都为 *true*，则逻辑“与”运算结果为 *true*。

如果至少有一个操作数为 *true*，则逻辑“或”运算结果为 *true*。

“与”和“或”运算符总是从左到右计算操作数，在可能的情况下，会采用短路求值机制。如果左操作数为 *false*，那么“与”运算符会跳过第二个操作数，因为结果已经为 *false*，此时“与”运算符对结果没有任何影响。如果左操作数为 *true*，那么出于同样的原因，“或”运算符跳过第二个操作数，因为在任何情况下，结果都为 *true*。

这通常在程序用于避免第二个（及后续）操作数引发的错误。例如，我们可以通过以下表达式来避免发生“访问不存在的数组元素”错误：

```
index < ArraySize(array) && array[index] != 0
```

这里我们使用 **ArraySize** 内置函数，该函数可返回数组长度。仅当 *index* 小于该长度时，才读取具有该索引的元素并与零进行比较。

对照检查时，也会使用 ||，例如：

```
ArraySize(array) == 0 || array[0] == 0
```

如果数组为空，则条件立即为真。并且只有在有元素的情况下，才会继续对内容进行其他检查。

如果表达式包含通过逻辑“或”组合而成的多个操作数，则当第一个操作数为 *true*（如有）时，将立即得到最终结果为 *true*。但是，如果操作数是通过逻辑“与”组合而成，则当第一个操作数为 *false* 时，将立即得到最终结果为 *false*。

当然您可以在一个表达式中组合不同的操作，但需要注意其优先级的不同：取反优先执行，然后是逻辑“与”相关的条件，最后是逻辑“或”相关的条件。如果需要另一个序列，必须使用圆括号明确指定。

例如，以下不带圆括号的表达式 `A && B || C && D` 实际上等同于：`(A && B) || (C && D)`。若要让逻辑“或”优先执行，需用圆括号括起来：`A && (B || C) && D`。有关使用圆括号的更多详细信息，参见 [用圆括号分组](#) 一节。

ExprLogical.mq5 脚本中给出了几个简单例子，让你通过实践来检查逻辑运算。

```
int x = 3, y = 4, z = 5;
bool expr1 = x == y && z > 0; // false, x != y, z does not matter
bool expr2 = x != y && z > 0; // true, both conditions are complied with
bool expr3 = x == y || z > 0; // true, it is sufficient that z > 0
bool expr4 = !x; // false, x must be 0 to get true
bool expr5 = x > 0 && y > 0 && z > 0; // true, all 3 are complied with
bool expr6 = x < 0 || y > 0 && z > 0; // true, y and z are sufficient
bool expr7 = x < 0 || y < 0 || z > 0; // true, z is sufficient
```

在计算 *expr6* 的字符串中，编译器发出警告：“检查运算符优先级可能存在的错误；请使用圆括号阐明优先级”。

请勿将逻辑运算 `&&` 和 `||` 与按位运算 `&` 和 `|` 混淆（后者在 [下一节](#) 中介绍）。

2.5.7 按位运算

有时您可能需要在位级别处理数字。为此，系统提供了一组适用于整数类型的按位运算。

下表提供了按位运算的所有符号和说明以及结合性，并按优先级排序。

优先级	符号	说明	示例	结合性
2	<code>~</code>	按位补码（反转）	<code>~e1</code>	右
5	<code><<</code>	左移	<code>e1 << e2</code>	左
5	<code>>></code>	右移	<code>e1 >> e2</code>	左
8	<code>&</code>	按位“与”	<code>e1 & e2</code>	左
9	<code>^</code>	按位“异或”	<code>e1 ^ e2</code>	左
10	<code> </code>	按位“或”	<code>e1 e2</code>	左

整个分组中，只有按位求补运算 `~` 是一元的，其他都是二元的。

在所有情况下，如果操作数大小小于 *int/uint*，则会先向高位添加 0 位，从而扩展到 *int/uint*。根据操作数类型是有符号还是无符号，高位可能会影响符号。

Windows 标准应用程序“计算器”可以帮助我们理解数字在位级的表示。如果在“查看”菜单中选择“程序员”操作模式，程序中将出现一组切换按钮，用于选择以十六进制 (Hex)、十进制 (Dec)、八进制 (Oct) 或二进制 (Bin) 格式表示数字。后面的显示了位数。此外，还可以选择数字位宽：1、2、4 和 8 个字节。按钮可以执行所有相关运算：“非” (`~`)、 “与” (`&`)、 “或” (`|`)、 “异或” (`^`)、 “左移” (`<<`) 和右移 (`>>`)。

由于计算器使用有符号数字，因此在切换到十进制模式时，可能会出现负值（切记，高位会被解释为符号）。为方便分析，合理做法是排除出现的减号，为此，有必要选择更高级的大小（以字节为单位）。例如，要检查 0 到 255 范围内的值 (`uchar`, 无符号单字节整数)，应选择 2 个字节（否则，只有 0 到 127 的十进制值为正，而其他值将显示为负数）。

按位补码会创建一个值，其中用 0 位代替所有 1 位，而用 1 位代替所有 0 位。例如，一个全是 0 位的字节的取反得到一个全是 1 位的字节。数字 50 以按位表示为 `'00110010'` (字节)。按位取反得到 `'11001101'`。

数值一的十六进制格式为 0x0001 (对应 *short*)。反转这些位得到 0xFFFE (参见 *ExprBitwise.mq5* 脚本)。

```
short v = ~1; // 0xfffe = -2
ushort w = ~1; // 0xfffe = 65534
```

按位“与”会检查两个操作数中的每个位，如果两个操作数中某一对位置均置位 (设为 1)，则在结果中存储 1 位。在所有其他情况下 (仅在一个操作数中设置了值为 1 的位，或者在两个对应位置均设置了值为 0 的位)，则在结果中写入 0 位。

若某一位在两个操作数中至少一个设置了值为 1 的位，则按位“或”运算会将 1 位写入结果中。

如果某一位在第一个或第二个操作数中设置为 1 (但没有同时设置为 1)，则按位“异或”在结果中的对应位置写入 1 位。以下显示了两个数字 X 和 Y 的二进制表示以及对它们进行按位运算的结果。

X	10011010	154
Y	00110111	55
X & Y	00010010	18
X Y	10111111	191
X ^ Y	10101101	173

编写包含多个不同运算符的复杂表达式时，请使用圆括号进行分组以避免优先级混淆。

移位运算可进行向左 (<<) 或向右 (>>) 移位的操作，移动的位数在第二个操作数 (必须为非负整数) 中定义。如果因移位运算而导致位超出了内存单元的边界，左侧 (对应 <<) 或右侧 (对应 >>) 位会被丢弃。左移后，会在右侧添加对应数量的 0 位。右移后，如果操作数无符号，则在左侧添加 0 位；如果操作数有符号，则复制符号位。在后一种情况下，正数左侧添加 0 位，负数左侧添加 1 位；即保留符号。

```
short q = v << 5; // 0xffc0 = -64
ushort p = w << 5; // 0xffc0 = 65472
short r = q >> 5; // 0xfffe = -2
ushort s = p >> 5; // 0x07fe = 2046
```

在上例中，最开始的左移“销毁”了 *p* 变量的高阶位，而随后右移相同位数并用零填充，导致值从 0xffc0 减小至 0x07fe。

移位大小 (位数) 必须小于操作数类型的大小 (需考虑其可能的扩展)。否则，所有初始位都将丢失。

移动 0 位则不会改变数字。

请勿将按位运算 & 和 | 与逻辑运算 && 和 || 混淆 (在 [上一节](#) 中介绍过)。

2.5.8 修改运算

“修改”也称为复合赋值，允许在单个运算符内实现 [算术](#) 或 [按位](#) 运算与常规 [赋值](#) 的结合。

优先级	符号	说明	示例	结合性
14	<code>+=</code>	赋值加法	<code>e1 += e2</code>	右
14	<code>-=</code>	赋值减法	<code>e1 -= e2</code>	右
14	<code>*=</code>	赋值乘法	<code>e1 *= e2</code>	右
14	<code>/=</code>	赋值除法	<code>e1 /= e2</code>	右
14	<code>%=</code>	赋值模除法	<code>e1 %= e2</code>	右
14	<code><<=</code>	赋值左移	<code>e1 <<= e2</code>	右
14	<code>>>=</code>	赋值右移	<code>e1 >>= e2</code>	右
14	<code>&=</code>	赋值按位“与”	<code>e1 &= e2</code>	右
14	<code> =</code>	赋值按位“或”	<code>e1 = e2</code>	右
14	<code>^=</code>	赋值按位“与/或”	<code>e1 ^= e2</code>	右

这些运算符对 ***e1*** 和 ***e2*** 操作数执行相关操作，运算结果存储在 ***e1*** 中。

类似 ***e1 @= e2*** 的表达式 (**@** 是表中的任意运算符) 大致相当于 ***e1 = e1 @ e2***。“大致”这个词强调了两者还是存在一些细微区别。

首先，如果 ***e2*** 的位置被一个运算符优先级低于 **@** 的表达式占用，则 ***e2*** 的计算顺序仍然优先。也就是说，如果用圆括号标示优先级，我们会得到 ***e1 = e1 @ (e2)***。

其次，如果 ***e1*** 表达式中存在对变量的副修改，此类修改只进行一次。下面的例子演示了这一点。

```
int a[] = {1, 2, 3, 4, 5};
int b[] = {1, 2, 3, 4, 5};
int i = 0, j = 0;
a[++i] *= i + 1;           // a = {1, 4, 3, 4, 5}, i = 1
                           // not equivalent!
b[++j] = b[++j] * (j + 1); // b = {1, 2, 4, 4, 5}, j = 2
```

此处，***a*** 和 ***b*** 数组包含相同的元素，并使用索引变量 ***i*** 和 ***j*** 进行处理。同时，***a*** 数组的表达式使用了 ***** 运算，而 ***b*** 数组的表达式使用了等效运算。结果并不相等：索引变量和数组都不同。

其他运算符在处理位级操作的问题时会很有用。因此，以下表达式可用于将特定位设置为 1：

```
ushort x = 0;
x |= 1 << 10;
```

此处，将 1 (0000 0000 0000 0001) 左移 10 位，获得一个第 10 位为 1 的数字 (0000 0100 0000 0000)。按位“或”运算将该位拷贝到变量 ***x***。

要重置相同的位，可以编写：

```
x &= ~(1 << 10);
```

这里，对左移 10 位的 1（我们在上述表达式中看到了结果）进行取反运算，这导致所有位的值都发生变化：'1111 1011 1111 1111'。按位“与”运算可重置 *x* 变量中的零位（在本例中为 1），而 *x* 中的所有其他位保持不变。

2.5.9 条件三元运算符

三元条件运算符允许在一个表达式中根据特定条件描述两个计算选项。该运算符的语法如下：

```
condition ? expression_true : expression_false
```

必须在第一个操作数 'condition' 中指定逻辑条件。可以是 [比较运算](#) 和 [逻辑运算](#) 的任意组合。两个分支都必须存在。

如果条件为真，将计算 *expression_true* 表达式，如果条件为假，将计算 *expression_false* 表达式。

该运算符保证 *expression_true* 和 *expression_false* 表达式中只有一个被执行。

两个表达式的类型必须相同，否则，将会尝试对其进行 [隐式类型强制转换](#)。

请注意，MQL5 中表达式的处理结果始终代表一个右值（在 C++ 中，如果表达式中只有左值，那么运算符的结果也将是左值）。因此，以下代码在 C++ 中正常编译，但在 MQL5 中则会出现错误：

```
int x1, y1; ++(x1 > y1 ? x1 : y1); // '++' - l-value required
```

条件运算符可以嵌套，也就是允许将另一个条件运算符用作条件或分支（*expression_true* 或 *expression_false*）。同时，如果不使用圆括号来明确表示分组，条件之间的关系难免出现混乱。我们来分析 *ExprConditional.mq5* 中的例子。

```
int x = 1, y = 2, z = 3, p = 4, q = 5, f = 6, h = 7;
int r0 = x > y ? z : p != 0 && q != 0 ? f / (p + q) : h; // 0 = f / (p + q)
```

在本例中，第一个逻辑条件表示 *x > y* 比较运算。如果为真，则执行带有 *z* 变量的分支。如果为假，则检查附加逻辑条件 *p != 0 && q != 0*（附带两个表达式选项）。

下面还有几个运算符，其中逻辑条件用大写字母表示，而计算选项用小写字母表示。为了简单起见，它们都被设为变量（来自上面的例子）。实际上，这三个组成部分中的任何一个都能成为更复杂的表达式。

对于每个字符串，你可以跟踪获得结果的方法（注释中显示了跟踪过程）。

```
bool A = false, B = false, C = true;
int r1 = A ? x : C ? p : q;           // 4
int r2 = A ? B ? x : y : z;           // 3
int r3 = A ? B ? C ? p : q : y : z;   // 3
int r4 = A ? B ? x : y : C ? p : q;    // 4
int r5 = A ? f : h ? B ? x : y : C ? p : q; // 2
```

由于该运算符具有右结合性，所以复合表达式是从右向左分析的，即最右侧的结构体包含三个操作数并用 '?' 和 ':' 进行组合，这个结构体成为外部条件的操作数（写在左侧）。然后基于这种替换，再次从右向左解析表达式，以此类推，直到获得最后完整的上层结构 '?:'。

因此，上述表达式的分组方式如下（圆括号代表编译器的隐式解释；但是可以在表达式中添加这类圆括号来提升源代码的易读性，实际上也建议使用这种方法）。

```
int r0 = x > y ? z : ((p != 0 && q != 0) ? f / (p + q) : h);
int r1 = A ? x : (C ? p : q);
int r2 = A ? (B ? x : y) : z;
int r3 = A ? (B ? (C ? p : q) : y) : z;
int r4 = A ? (B ? x : y) : (C ? p : q);
int r5 = (A ? f : h) ? (B ? x : y) : (C ? p : q);
```

对于 **r5** 变量，第一个条件 **A ? f : h** 计算后续表达式的逻辑条件，因此被转换为 **bool**。由于 **A** 等于 **false**，因此取 **h** 变量的值。**h** 不等于 0；因此，第一个条件被判定为真。这导致执行 **(B ? x : y)** 分支，由于 **B** 为 **false**，因此从该分支返回 **y** 变量的值。

运算符中必须包含所有 3 个组成部分（一个条件和 2 个备选）。否则，编译器将生成错误“意外标记”：

```
// ';' - unexpected token
// ';' - ':' colon sign expected
int r6 = A ? B ? x : y; // lack of alternative
```

从编译器的角度来看，“标记”是源代码不可分割的一部分，有其独立的含义或用途，如类型、标识符、标点符号等。编译器将整个源代码分成标记序列。之前介绍的运算符符号也是标记。在上述代码中，有两个 **?** 符号，那么必须有两个 **:** 符号与之匹配，然而实际上只有一个。因此，编译器会提示语句结束符号；提前出现，并“询问”具体缺失的内容：“需要冒号”。

由于条件运算符的优先级非常低（在整张表中为 13，参见 [运算优先级](#)），因此建议用将条件运算符用圆括号括起来。这样有助于避免条件运算符的操作数被优先级更高的相邻运算“捕获”的情况。例如，如果我们通过两个三元运算符之和来计算特定变量 **w** 的值，可以采用下面这种简单方法：

```
int w = A ? f : h + B ? x : y; // 1
```

计算方式会出乎我们的意料。由于求和表达式 **h + B** 的优先级更高，因此被视为单个表达式。由于从右向左解析，这个求和作为一个条件出现，并被转换为 **bool** 类型，编译器甚至对此发出警告“表达式不是布尔型”。使用圆括号可以直观展现编译器的解释逻辑：

```
int w = A ? f : ((h + B) ? x : y); // 1
```

为了解决这个问题，我们应该用自己的方式添加圆括号。

```
int v = (A ? f : h) + (B ? x : y); // 9
```

深度嵌套条件运算符会对代码的可理解性产生负面影响。建议最多只嵌套两层或三层。

2.5.10 逗号

逗号运算符（明确表示为 **,**）放在从左到右独立计算的两个表达式之间。也就是说，这个运算符本身不执行任何操作，仅指定在一个语句内部两个或多个表达式的顺序。

放在序列右侧的表达式可以使用左侧表达式的计算结果，因为左侧表达式已经处理完毕。

该运算符的结果是最右边表达式的结果。逗号运算符的优先级最低。

目前，在 MQL5 中使用该运算符受限于 [for 语句](#) 的头部。

示例：

```
for(i=0,j=99; i<100; i++,j--)
    Print(array[i][j]);
```

我们来重述一下 MQL5 中逗号运算符的几个主要方面：

求值顺序：

- ①表达式是从左向右计算的。因此，右侧表达式可以使用左侧表达式的结果，因为左侧表达式已经处理完毕。

结果和优先级：

- ①逗号运算符的结果是最右侧表达式的值。需要注意的是，逗号运算符的优先级最低，这意味着表达式中的其他运算符具有更高的优先级。

2.5.11 特殊运算符 sizeof 和 typename

sizeof

sizeof 运算符返回其操作数的大小（以字节为单位）。运算符语法：**sizeof(x)**，其中 **x** 可以是类型或表达式。本例中没有计算表达式，因为 **sizeof** 运算符是在编译阶段执行的，实际运算时，在表达式中用常量替代。

对于定长数组，运算符返回分配的内存总量，即所有维度中的元素数与类型大小（以字节为单位）的乘积。对于动态数组，该运算符返回存储数组特性的内部结构的大小。

下面举几个例子并加以说明 (*ExprSpecial.mq5*)。

```
double array[2][2];
double dynamic1[][1];
double dynamic2[][2];
Print(sizeof(double));           // 8
Print(sizeof(string));           // 12
Print(sizeof("This string is 29 bytes long!")); // 12
Print(sizeof(array));            // 32
Print(sizeof(array) / sizeof(double)); // 4 (quantity of elements)
Print(sizeof(dynamic1));         // 52
Print(sizeof(dynamic2));         // 52
```

注释中标记了即将在日志中打印的结果。

double 类型占用 8 个字节。**string** 类型的大小为 12 个字节。这 12 个字节存储我们在涉及 **字符串** 类型相关章节中提到的服务信息。任何字符串（即使未初始化）都会被分配这个 12 字节的内存。请注意，包含 29 个字符文本的字符串大小也是 12。这是因为空字符串和包含内容的字符串都有一个内部结构，用于存储对内存的引用。要获得文本长度，我们应该使用 **StringLen** 函数。

定长数组大小实际上是元素数量 ($2 \times 2 = 4$) 与 **double** 类型大小 (8) 的乘积，共计 32。因此，类似 **sizeof(array) / sizeof(double)** 的表达式可用于获取数组中元素的实体。

对于动态数组，内部结构的大小是 52 字节。**dynamic1** 和 **dynamic2** 数组说明中的差异不会影响该值。

`sizeof` 运算符对于获取 **类**和**结构体**的大小来说非常有效。

typename

`typename` 运算符返回一个字符串，内容是传递给它的参数名（参数可以是类型或表达式）。对于数组，除了数据类型关键字之外，还会输出用一对圆括号括起的标签（或者几对圆括号，取决于数组的维数）。

```
Print(typename(double));           // double
Print(typename(array));           // double [2][2]
Print(typename(dynamic1));        // double[][1]
Print(typename(1 + 2));           // int
```

对于自定义类型，比如类、结构等（我们将在第三章讨论），类型名称在实体类别之后，比如 `"class MyCustomType"`。此外，对于常量，字符串中说明中将添加 `"const"` 修饰符。

因此，要获得由一个单词组成的短类型名称，使用 `TypeName.mqh` 附加文件的宏 `TYPENAME`。

有必要学习所谓的 **模板** 中的类型名称，这些模板可以从源代码中为模板参数中定义的不同类型生成类似的实现。

2.5.12 用圆括号分组

在前面几节中，我们已经多次看到，由于运算优先级，一些表达式可能会产生意外结果。为了明确地改变计算顺序，我们应该使用圆括号。表达式中带圆括号的部分无论其默认优先级如何，都会比周围表达式具有更高优先级。成对的圆括号可以嵌套，但建议嵌套层数最多 3 到 4 层。最好把过于复杂的表达式分成几个简单表达式。

`ExprParentheses.mq5` 脚本展示了在表达式中放置圆括号的演算过程。其最初目的是使用左移运算 `<<` 在 `flags` 变量中设置位。如果位编号不为零，则从 `offset` 变量获取位编号，否则位编号取 1（切记，编号从零开始）。然后将获得的值乘以 `coefficient`。我们不用纠结这个例子有没有实际意义。但代码中的确可能出现更复杂的结构。

```
int offset = 8;
int coefficient = 10, flags = 0;
int result1 = coefficient * flags | 1 << offset > 0 ? offset : 1;    // 8
int result2 = coefficient * flags | 1 << (offset > 0 ? offset : 1);  // 256
int result3 = coefficient * (flags | 1 << (offset > 0 ? offset : 1)); // 2560
```

不带圆括号的第一个版本，甚至会让编译器产生自我怀疑。编译器给出一个我们熟悉的警告：“表达式不是布尔型”。问题是三元条件运算符是这里所有运算符中优先级最低的。基于此，`?` 之前的整个左边部分都被认为是它的条件。在条件中，按以下顺序进行计算：乘法、按位移位、“大于”比较和按位“或”运算，因此计算结果是一个整数。当然，可以将计算结果用作 `true` 或 `false`，但是最好使用 **显式类型强制转换** 将这种意图“传达”给编译器。如果不进行类型强制转换，编译器显然有理由认为该表达式有问题。第一次计算的结果是 8。这是不正确的。

让我们在三元运算符两边加上圆括号。编译器的警告就会消失。然而，表达式仍然计算错误。由于乘法的优先级高于按位“或”的优先级，因此 `coefficient` 和 `flags` 变量先进行相乘，然后再与左移获得的位掩码结合。结果是 256。

最后，添加另一对圆括号后我们得到了正确结果：2560。

2.5.13 运算优先级

以下是按优先级排序的所有运算的完整表格。

优先级	符号	说明	示例	结合性
0	::	作用域解析	n1 :: n2	左
1	()	分组	(e1)	左
1	[]	索引	[e1]	左
1	.	取消引用	n1.n2	左
1	++	后缀递增	e1++	左
1	--	后缀递减	e1--	左
2	!	逻辑“非”	!e1	右
2	~	按位补码(反转)	~e1	右
2	+	一元加法	+e1	右
2	-	一元减法	-e1	右
2	++	前缀递增	++e1	右
2	--	前缀递减	--e1	右
2	(type)	类型强制转换	(n1)	右
2	&	取地址	&n1	右
3	*	乘法	e1 * e2	左
3	/	除法	e1 / e2	左
3	%	模除法	e1 % e2	左
4	+	加法	e1 + e2	左
4	-	减法	e1 - e2	左
5	<<	左移	e1 << e2	左
5	>>	右移	e1 >> e2	左
6	<	小于	e1 < e2	左
6	>	大于	e1 > e2	左
6	<=	小于等于	e1 <= e2	左
6	>=	大于等于	e1 >= e2	左
7	==	等于	e1 == e2	左

优先级	符号	说明	示例	结合性
7	<code>!=</code>	不等于	<code>e1 != e2</code>	左
8	<code>&</code>	按位“与”	<code>e1 & e2</code>	左
9	<code>^</code>	按位“异或”	<code>e1 ^ e2</code>	左
10	<code> </code>	按位“或”	<code>e1 e2</code>	左
11	<code>&&</code>	逻辑“与”	<code>e1 && e2</code>	左
12	<code> </code>	逻辑“或”	<code>e1 e2</code>	左
13	<code>?:</code>	三元条件	<code>c1 ? e1 : e2</code>	右
14	<code>=</code>	赋值	<code>e1 = e2</code>	右
14	<code>+=</code>	赋值加法	<code>e1 += e2</code>	右
14	<code>-=</code>	赋值减法	<code>e1 -= e2</code>	右
14	<code>*=</code>	赋值乘法	<code>e1 *= e2</code>	右
14	<code>/=</code>	赋值除法	<code>e1 /= e2</code>	右
14	<code>%=</code>	赋值模除法	<code>e1 %= e2</code>	右
14	<code><<=</code>	赋值左移	<code>e1 <<= e2</code>	右
14	<code>>>=</code>	赋值右移	<code>e1 >>= e2</code>	右
14	<code>&=</code>	赋值按位“与”	<code>e1 &= e2</code>	右
14	<code> =</code>	赋值按位“或”	<code>e1 = e2</code>	右
14	<code>^=</code>	赋值按位“与/或”	<code>e1 ^= e2</code>	右
15	<code>,</code>	逗号	<code>e1 , e2</code>	左

如上所述，方括号用于指定数组元素的索引，因此具有最高的优先级。

除了之前讨论过的运算符，本节还包含一些我们未遇到过的运算符。

我们将学习面向对象的编程 (OOP) 中涉及的 [作用域解析](#) 运算符 `::`。同时我们还需学习取消引用操作符 `'.'`。类型 (类) 的标识符及其特性 (并非表达式) 充当它们的操作数。

在 OOP 中，取地址运算符 `&` 旨在 [通过引用进行函数参数传递](#) 并获取 [对象地址](#)。在这两种情况下，运算符都作用于变量 (左值)。

显式类型强制转换运算将在 [下一章](#) 中介绍 中介绍。

2.6 类型转换

在本节中，我们将讨论类型转换的概念，目前仅限于内置数据类型。后面在学习了 OOP 之后，我们将探讨对象类型内在的精妙作为补充。

MySQL 中的类型转换是指改变变量或表达式的数据类型的过程。MySQL 支持三种主要的类型转换：隐式、算术和显式。

隐式类型转换：

- ① 当上下文中实际使用的变量类型与预期类型不符时自动发生。例如，整数值可以隐式转换为实数值。

算术类型转换：

- ① 在对不同类型的操作数进行算术运算时出现。编译器试图保持最大的精度，但会发出数据可能丢失的警告。例如，在整数除法中，结果被转换为实数类型。

显式类型转换：

- ① 给予程序员对类型转换的控制权。可以用两种形式进行转换：C 语言风格 ((target)) 和 “函数式”风格 (target())。当您需要显式指示编译器执行类型间转换时，例如，舍入实数或需要连续的类型转换时便会用到。

务必要理解隐式、算术和显式类型转换之间的差异，这一点对于确保正确执行运算和避免数据丢失而言至关重要。这类知识有助于程序员在 MySQL 开发中有效地利用这种机制。

2.6.1. 隐式类型转换

如果在源代码中的某个位置使用了一种类型，但需要的是另一种类型，并且两者之间存在转换规则，那么会自动进行类型转换。这种转换称为隐式类型转换，但不一定符合程序员的意图。此外，一些转换操作有副作用，编译器不知道这些转换的使用是否是有意为之，因此会用警告突出显示相应的代码行。为了解决这些问题，提供了显式类型转换语法（参见 [显式类型强制转换](#)）。

在研究类型和变量时，我们已经发现了隐式类型转换的几个规则。

具体来说，如果一个非布尔类型的值被赋给一个 *bool* 变量，那么 0 值被认为 *false*，其余被认为 *true*。在更常见的情况下，所有假设存在逻辑条件的表达式都被转换为 *bool* 类型。例如，三元条件运算符的第一个操作数总是被转换为 *bool*。

但是，如果将一个 *bool* 类型的值赋给数字类型，那么 *true* 变为 1，*false* 变为 0。

将实数赋给整数类型变量时，小数部分被丢弃（编译器会发出警告）。另一方面，当整数被赋给实数类型变量时，精度可能会丢失（编译器发出警告）。我们已经在 [整数](#) 和 [实数](#) 相关章节中讨论了这些内容。

如果我们有整数和浮点数，那么都被转换为最大长度的浮点数（通常是 *double*，除非你显式指定 *float* 或者数字字面量有后缀 f，例如 1234.56789f）。

对于不同大小的整数，也有转换规则：如果需要，它们会扩展，这意味着它们会增大到表达式中使用的最大整数类型的大小（参见 [算术类型转换](#)）。

除了表达式，当 = 符号左右两边的类型不匹配时，我们经常需要在初始化和赋值期间隐式转换类型。通过函数参数传递值以及从函数返回结果时，相同的转换规则也适用（更多详细信息，请参见 [函数](#) 一节）。

综上所述，一行代码可以执行大量转换。如果这导致编译器警告，最好确保转换是有意为之，并通过插入显式类型转换来消除警告。

```
short s = 10;
long n = 10;
int p = s * n + 1.0;
```

在本例中，执行乘法时，*s* 变量的类型被扩展到第二操作数 *long* 的类型，并且获得 *long* 类型的中间结果。因为常数 1.0 是 *double* 型，所以乘积的结果在执行加法之前被转换为 *double*。总结果也是 *double* 型；而变量 *p* 是 *int* 型，因此执行从 *double* 到 *int* 的隐式转换。

特殊类型 *datetime* 和 *color* 根据长度分别为 8 字节和 4 字节的整数之规则进行转换。但对于日期和时间，最大值有更严格的限制 -32535244799，对应 D'3000.12.31 23:59:59'。

大多数类型都可以与字符串隐式相互转换，但结果并不总是令人满意，因此编译器发出警告“从‘数字’到‘字符串’的隐式转换”和“从‘字符串’到‘数字’的隐式转换”，以便程序员可以检查它们。例如，如果将字符串转换为整数，则字符串开头仅允许包含数字和 *+/-* 字符。从字符串转换为实数时，除了数字之外，还允许出现点号 *.* 以及表示“指数”的符号 *e* 或 *E*，例如 +1.2345e-1)。如果字符串中遇到不支持的字符（例如字母），则字符串的其余部分将被全部丢弃。

例如，字符串日期和时间（“2021.12.12 00:00”）不能无损地赋给 *datetime* 类型的变量，因为 *datetime* 是一个整数（秒数）。在这种情况下，从字符串中读取数字将在到达第一个点号时结束，即获得 2021 的数字值。该秒数对应于 1970 年的第 34 分钟。

还有一些特殊的函数用于这种转换（参见 [数据转换](#) 一节）。

唯一禁止的隐式和显式类型转换方向是从 *string* 转换为 *bool*。在这种情况下，编译器会显示错误消息“无法将‘string’类型隐式转换为‘bool’类型”。

本章的例子可在 *TypeConversion.mq5* 中找到。

2.6.2. 算术类型转换

在算术计算和比较表达式中，经常用不同类型的值作为操作数。为正确处理这些值，必须让这些类型达到某种“共同标准”。除非程序员指定了显式转换规则，否则编译器会尝试在没有程序员干预的情况下完成此操作（参见 [显式类型转换](#)）。在这种情况下，编译器会尽可能保持数字的最大精度。特别是，它产生了整数的表示范围扩大以及从整数到实数的转换（如涉及的话）。

整数扩展意味着从 *bool*、*char*、*unsigned char*、*short*、*unsigned short* 隐式转换为 *int*（或转换为 *unsigned int*，如果 *int* 不足以存储特定数字）。大值可以转换为 *long* 和 *unsigned long*。

如果变量的类型无法存储计算表达式所获得类型的结果，编译器将发出警告：

```
double d = 1.0;
int x = 1.0 / 10; // truncation of constant value
int y = d / 10;   // possible loss of data due to type conversion
```

初始化 *x* 和 *y* 变量的表达式包含实数 1.0，因此其他操作数（本例中为常数 10）被转换为 *double*，除法的结果也将是 *double* 类型。然而变量的类型是 *int*，因此它会被隐式转换。

计算 1.0/10 是由编译器在编译期间完成的，因此它得到一个 *double* 类型的常量 (0.1)。当然，在实践中，初始化常量不太可能超过接收变量的大小。因此，编译器警告“常量值截断”可以被认为是异常的。这只是复杂问题的简单化表示。

但是，基于变量的计算也会导致类似的数据丢失。我们在这里看到的第二个编译器警告（“类型转换可能会导致数据丢失”）出现得更为频繁。此外，不仅在从实数类型转换到整数类型时可能会丢失数据，反向转换时也可能丢失数据。

```
double f = LONG_MAX; // truncation of constant value
long m1 = 10000000000;
f = m1 * m1;          // possible loss of data due to type conversion
```

我们知道，*double* 类型不能准确表示大整数（虽然它的有效值范围比 *long* 大得多）。

由于类型不匹配，我们可能遇到另一个警告：“整数常量溢出”。

```
long m1 = 10000000000;
long m2 = m1 * m1;          // ok: m2 = 10000000000000000000
long m3 = 10000000000 * 10000000000; // integral constant overflow
// m3 = -1486618624
```

SQL5 中的整数常量为 *int* 型，因此在执行百万乘百万的运算时会考虑该类型的范围，这个范围是 `INT_MAX` (2147483647)。值 10000000000000000000 导致溢出，*m3* 为该值除以范围后的得出的余数（更多相关信息见下方侧栏）。

接收变量 *m3* 为 *long* 类型，但一事实并不意味着表达式中的值必须事先转换成该类型。转换只在赋值的时候进行。为了根据长整型的规则执行乘法，您需要以某种方式在表达式中直接指定 *long* 类型。这可以通过显式转换或使用变量来完成。特别是，使用 *long* 类型的变量 *m1* 获得相同的乘积（如 *m1 * m1*）会得到 *m2* 的正确结果。

有符号和无符号整数

程序编写并不总是完美无缺，也无法防止所有可能的失败情况。因此，有时计算过程中获得的整数不适用于所选整数类型的变量。那么取这个值除以可写入相应字节数（类型大小）的最大值（*M*）的余数，再加上 1。因此，对于大小为 1 到 4 字节的整数类型，*M + 1* 分别为 256、65536、4294967296 和 18446744073709551616。

但是对于有符号类型有一个细微的差别。我们知道，有符号数的值范围在正负区域之间大致均等地分布。因此，新的“余数”值可能在 50% 的情况下超过正数或负数限制。在这种情况下，数字变为了相反数：数字的符号改变，最终与原数相差 *M*。

关键要理解，这种转换本质上由于对内部表示中的位状态的不同解释而发生的，并且状态本身对于有符号和无符号数是相同的。

我们来用一个最小整数类型的例子来说明：*char* 和 *uchar*。

由于 *unsigned char* 可以存储从 0 到 255 的值，256 映射到 0，-1 映射到 255，300 映射到 44，以此类推。如果我们尝试将 300 写入一个常规的有符号 *char*，也会得到 44，因为 44 在 0 到 127 的范围内（*char* 的正值范围）。但是，如果将 *char* 和 *uchar* 变量设置为 3000，结果会有所不同。3000 除以 256 的余数是 184。这个余数在 *uchar* 的情况下保持不变。但如果是 *char*，相同的位组合会得到结果 -72。很容易发现 184 和 -72 相差 256。

在下面的例子中，多亏了编译器警告，我们轻松地发现了问题。

```
char c = 3000;      // truncation of constant value
Print(c);          // -72
uchar uc = 3000;   // truncation of constant value
Print(uc);         // 184
```

但如果在计算过程中得到一个特别大的数字，则不会发出警告。

```
char c55 = 55;
char sm = c55 * c55; // ok!
Print(sm);           // 3025 -> -47
uchar um = c55 * c55; // ok!
Print(um);           // 3025 -> 209
```

当在同一个表达式中使用相同大小的有符号和无符号整数时，也会产生类似的效果，因为有符号操作数被转换为无符号操作数。例如：

```
uint u = 11;
int i = -49;
Print(i + i); // -98
Print(u + i); // 4294967258 = 4294967296 - 38
```

两个负整数相加，我们得到预期的结果。第二个表达式将 `-38` 的和映射到“相反的”无符号数 `4294967258`。

由于这些潜在的问题，不建议在同一个表达式中混合使用有符号和无符号类型。

除此之外，如果我们从一个无符号整数中减去某个数，则需要确保结果不会是负数。否则，它将被转换为正数，并且会曲解算法的意思，特别是 **while 循环的意思**，该循环检查变量的“大于等于零”条件：由于无符号数始终是非负的，我们很容易得到一个无限循环，即程序挂起。

2.6.3.显式类型转换

对于显式类型转换，MQL5 支持两种形式的符号：C 语言风格和“函数式”。C 语言风格的语法如下：

```
target t = (target)s;
```

其中 **target** 是目标类型的名称。任何表达式都可以是数据源 **s**。如果其中执行了任何运算，则必须将表达式放入圆括号中，以便类型转换适用于整个表达式。

另一种“函数式”语法如下所示：

```
target t = target(s);
```

我们来看几个例子。

```
double w = 100.0, v = 7.0;
int p = (int)(w / v); // 14
```

此处，两个实数相除的结果被显式转换为 **int** 类型。这样程序员确认了丢弃小数部分的意图，编译器不会发出警告。应该注意的是，MQL5 有一组函数，用于以各种方式对实数进行舍入（参见数学函数）。

相反，如果希望对整数执行运算并得到实数结果，则需要对操作数应用类型转换（在表达式中）：

```
int x = 100, y = 7;
double d = (double)x / y; // 14.28571428571429
```

转换其中一个操作数就足以自动将其余的操作数转换为相同类型。

如有必要，您可以连续执行多个类型转换操作。因为转换操作具有右结合性，所以按从右到左的顺序应用目标类型。在下面的例子中，我们将商转换为 *float* 型（这种转换能够以更紧凑、更少字符表示值），然后转换为 *string*。如果没有显式转换为 *string*，我们将遇到编译器警告“数字隐式转换为字符串”。

```
Print("Result:" + (string)(float)(w / v)); // Result:14.28571
```

切勿只是为了避免编译器警告而使用显式类型转换。如果没有实践依据，那就是在掩盖程序中的潜在错误。

2.7 语句

到目前为止，我们已经学习了数据类型、变量声明以及它们在计算表达式中的使用。然而，对于程序而言，这些就好比是高楼大厦的一砖一瓦。即使是最简单的程序也包含大量的代码块，这些代码块可以对相关的数据处理操作进行分组，并控制它们的执行顺序。这些代码块被称为语句，实际上我们已经使用过一些。

具体来说，声明一个变量（或几个变量）就是一个语句。将表达式求值结果赋给变量也是一个语句。严格来说，赋值运算本身就是表达式的一部分，所以把这样的语句称为表达式语句更为正确。顺便说一下，表达式可能不包含赋值运算符（例如，只是简单地调用一些不返回值的函数，如 `Print("Hello");`）。

程序执行就是语句的逐步执行：从上到下，从左到右（如果一行包含多个语句）。在最简单的情况下，语句序列是线性执行的，一个接着一个。对于大多数程序来说，这样还不够，所以出现了各种控制语句。通过控制语句，您可以组织程序中的循环（重复计算）以及根据条件选择算法运算方法。

语句是特殊的句法结构，表示根据规则编写的源文本。每种特定类型的语句都各有自己的规则，但它们之间也存在一些共性。除了 [复合语句](#)，所有类型的语句都以 `;` 结尾。复合语句可以没有分号，因为它的开始和结束是通过一对花括号来设置的。值得注意的是，使用复合语句，我们可以将多组语句包含在其他语句中，从而构建任意层次的算法结构。

在本章中，我们将熟悉所有类型的 MQL5 控制语句，并深度介绍声明和表达式语句的特征。

2.7.1 复合语句（代码块）

复合语句是其他语句的通用容器，用花括号 `{` 和 `}` 括起。这种代码块可用于定义函数体，如果需要多个受控语句，则代码块位于其他控制语句的头文件之后，或者在函数体或其他语句内独立作为嵌套块。这样便可以为变量创建一个受到限制的局部作用域。我们已经在 [变量的上下文、作用域和生存期](#) 一节讨论过这个问题。

概括地说，对复合语句的描述如下所示：


```
{
  [statements]
}
```

在这个原理说明中，任何用半圆括号括起来并带有上标 ^{opt} 的片段说明它是可选的。在这种情况下，代码块中可能没有任何嵌套语句。

在下面几节中，我们将分析复合语句与其他类型的语句的组合使用，以及它们能够包含什么。

这里需要强调一个细微的差别：在复合语句的说明之后，不需要使用分号 `;`。这是复合语句与其他语句的区别。

2.7.2 声明/定义语句

声明变量、数组、函数或程序的任何其他命名元素（包括结构和类，这些将在第三章讨论）即构成一个语句。

声明必须包含元素的类型和标识符（参见 [声明和定义变量](#)），以及一个可选的初始值，用于完成 [初始化](#)。此外，在声明时，可以指定附加的修饰符来改变元素的某些特性。特别是，我们已经知道 `static` 和 `const` 修饰符，很快还会认识更多。数组需要对维度和元素数量进行附加说明（参见 [数组的说明](#)），而函数则需要说明一系列参数（更多详细信息，请参见 [函数](#)）。

变量声明语句可以总结如下：

```
[modifiers] identifier type
    [= initialization expressions] ;
```

对于数组，它是这个样子的：

```
[modifiers] identifier type [ [size_1]opt ] [ [size_N] ]opt(3)
    [= { initialization_list } ]opt ;
```

主要的区别是必须至少有一对方括号（可以在方括号指定数组大小，也可以不指定；基于此，我们得到一个固定或动态分布的数组）。最多允许 4 对方括号（支持的最大测量数是 4）。

在多数情况下，声明可以同时充当定义，也就是说，它为元素保留内存，决定它的行为，并使其能够在程序中使用。具体来说，变量或数组的声明也是一个定义。从这个角度来看，声明语句可以被称为定义语句，但这个做法并不十分常见。

我们掌握的函数的基本知识，足以让我们推断其定义的形式：

```
type identifier ( [list_of_arguments] )
{
    [statements]
}
```

类型、标识符和自变量列表构成了函数头。

请注意，这是一个定义，因为此说明包含了函数的外部属性（接口）和定义其内部本质（实现）的语句。后者是带有一对花括号的代码块，紧跟在函数头之后。你可能会认为，这是我们在 [上一节](#)中提到的复合语句的例子。在这种情况下，术语上的同义反复是必不可少的，因为它是完全合理的：复合语句是函数定义语句的一部分。

稍后，我们将学习为什么以及如何将接口说明与实现分离，从而能够直接实现 [函数声明](#)，而无需定义。我们还将演示 [声明和定义之间的区别](#)（以类为例）。

声明语句通过在语句所在代码块的上下文中声明新元素的名称，使之可用（参见 [变量的上下文、作用域和生存期](#)）。回顾一下，代码块构成了对象（变量、数组）的局部作用域。在本书的第一章，我们在介绍问候语函数时遇到了这种情况。

除了局部作用域，还始终存在一个全局作用域，在全局作用域中，您还可以使用声明语句来创建可以从程序中的任何位置访问的元素。

如果在声明语句中没有出现 **static** 修饰符，但在某个局部代码块中出现，那么相应的元素在语句执行的时候被创建和初始化（严格地说，为了提高效率，函数内部所有局部变量的内存是在进入函数时立即分配的，但那时它们还没有形成）。

例如，**OnStart** 函数开头的变量 **i** 的以下声明可确保一旦该函数获得控制权（即它作为脚本的主函数，被终端调用），即用指定初始值（0）创建这样一个变量。

```
void OnStart()
{
    int i = 0;
    Print(i);

    // error: 'j' - undeclared identifier
    // Print(j);
    int j = 1;
}
```

由于第一个语句中的声明，变量 **i** 在函数的后续行中是已知的且可用的，特别是在第二行中包含 **Print** 函数调用，此函数在日志中显示变量的内容。

在函数最后一行中描述的变量 **j** 将在函数结束之前创建（虽然这没有什么意义，但便于理解）。因此，该变量在该函数的所有早期字符串中都不可知。试图使用带注释的 **Print** 调用将 **j** 输出到日志将导致“未声明的标识符”编译错误。

以这种方式声明的元素（在代码块内部，并且没有使用 **static** 修饰符）被称为自动元素，因为程序本身在进入代码块时为自动元素分配内存，并在退出代码块时（在我们的例子中，是在退出函数之后）销毁自动元素。因此，发生这种情况的内存区域称为栈（“后进先出”）。

自动元素是按照声明语句的执行顺序创建的（先是 **i**，然后是 **j**）。销毁则以相反顺序执行（先是 **j**，然后是 **i**）。

如果一个变量在没有初始化的情况下被声明，并且在没有先向其中写入一个有意义的值的情况下就开始在后续语句中使用（例如，放置在 = 符号右侧），编译器会发出一个警告：“可能使用了未初始化的变量”。

```
void OnStart()
{
    int i, p;
    i = p; // warning: possible use of uninitialized variable 'p'
}
```

如果声明语句具有 **static** 修饰符，则对应元素只在语句第一次执行时被创建一次，并保留在内存中，无论是否退出或后续多次进出同一代码块。所有这些静态成员只有在程序卸载时才会被移除。

尽管生存期延长了，但这类变量的作用域仍然局限于定义它们的局部上下文，并且只能从后续语句（位于代码下方）访问。

相比之下，全局上下文中的声明语句会在程序加载之后（在调用任何标准的 `start` 函数之前，比如用于脚本的 `OnStart` 函数），按照它们在源代码中出现的顺序创建元素。当程序卸载时，全局对象以相反的顺序被移除。

为了演示上述内容，让我们创建一个更“巧妙”的例子（`StmtDeclaration.mq5`）。回顾一下在第一章中学习的技能，除了 `OnStart` 之外，我们将编写一个简单的 `Init` 函数，该函数将用于变量初始化表达式，并将记录一个调用序列。

```
int Init(const int v)
{
    Print("Init: ", v);
    return v;
}
```

`Init` 函数接受整数类型 `int` 的单个参数 `v`，其值返回给调用代码（`return` 语句）。

这样便可以将它作为包装器来设置变量的初始值，例如，为两个全局变量设置初始值：

```
int k = Init(-1);
int m = Init(-2);
```

通过调用函数并从中返回，传递的自变量的值将进入变量 `k` 和 `m`。但在 `Init` 内部，我们另外使用了 `Print` 输出这个值，因此我们可以跟踪变量的创建方式。

请注意，我们不能在全局变量的初始化中（在其定义的上方）使用 `Init` 函数。如果我们试图将 `k` 变量声明移到 `Init` 声明的上方，会遇到“`Init` 是一个未知标识符”错误。这种限制只适用于全局变量的初始化，因为函数也是全局定义的，编译器会一次性建立一个这类标识符的列表。在任何其他情况下，在代码中定义函数的顺序并不重要，因为编译器首先将它们全部注册在内部列表中，然后从代码块中相互关联它们的调用。特别是，您可以将整个 `Init` 函数和全局变量 `k` 和 `m` 的声明移到 `OnStart` 函数下方——这不会有任何问题。

在 `OnStart` 函数内部，我们将使用 `Init` 描述更多变量：局部变量 `i` 和 `j`，以及静态变量 `n`。为简单起见，所有变量都被赋予了唯一的值，以便区分它们。

```
void OnStart()
{
    Print(k);

    int i = Init(1);
    Print(i);
    // error: 'n' - undeclared identifier
    // Print(n);
    static int n = Init(0);
    // error: 'j' - undeclared identifier
    // Print(j);
    int j = Init(2);
    Print(j);
    Print(n);
}
```

这里的注释解释了“未定义相关变量就尝试调用”这种错误。

运行脚本并获取以下日志：

```

Init: -1
Init: -2
-1
Init: 1
1
Init: 0
Init: 2
2
0

```

正如我们所看到的，全局变量在 **OnStart** 函数被调用之前初始化，并且完全按照它们在代码中出现的顺序初始化。内部变量的创建顺序与其声明语句的编写顺序相同。

如果定义了一个变量，但没有在任何地方使用，编译器将发出“变量 名称’未使用”警告。这个迹象表明程序员可能犯错了。

从前瞻性的角度来说，通过声明/定义语句，不仅可以将数据元素（变量、数组）或函数引入程序中，还可以将不为我们所知的新的用户定义类型（结构、类、模板、名称空间）引入程序。这种语句只能在全局级别上声明，也就是说，不能在所有函数外部定义。

也不能在函数内部定义一个函数。以下代码将不会被编译：

```

void OnStart()
{
    int Init(const int v)
    {
        Print("Init: ", v);
        return v;
    }
    int i = 0;
}

```

编译器将生成一个错误：“仅允许对全局、命名空间或类作用域进行函数声明”。

2.7.3 简单语句（表达式）

简单语句包含 **表达式**，比如为变量赋予新值或计算结果，以及函数调用。

从形式上看，语法如下所示：

```
expression ;
```

结尾的分号在这里很重要。因为 MQL5 源代码支持自由格式，所以 **;** 是唯一的分隔符，它告诉编译器上一条语句的结束位置和下一条语句的开始位置。语句通常是按行编写，例如，像这样：

```

int i = 0, j = 1, k;    // declaration statement
++i;                   // simple statement
j += i;                 // simple statement
k = (i + 1) * (j + 1); // simple statement
Print(i, " ", j);      // simple statement

```

然而，规则上并不禁止用速记代码编写：

```
int i=0,j=1;++i;j+=i;k=(i+1)*(j+1);Print(i," ",j);
```

如果没有 `;`，相邻表达式会悄悄地“粘在一起”并导致意想不到的结果。例如，表达式 `x = y - 10 * z` 很可能成为两个表达式：`x = y;` 和 `-10 * z;` (`-10` 带有一元减号)。怎么会出现这种情况？

这是因为在语法上允许编写“无实际效果”的语句，即不保存结果。以下是另一个例子：

```
i + j; // warning: expression has no effect
```

编译器发出“表达式无效”警告。允许构建这种表达式是有必要的，因为某些对象类型（我们将在第三章中学习）允许运算符重载。即我们可以用一些特定的操作来代替运算符符号的常用含义。如果 `i` 和 `j` 的类型不是 `int`，而是重写了某个加法运算的类，这样的表示法会生效，编译器不会发出警告。

复合语句内部只支持编写简单语句。例如，在函数外部调用 `Print` 函数会无效：

```
Print("Hello ", Symbol());
void OnStart()
{
}
```

我们将得到一连串的错误：

```
'Print' - unexpected token, probably type is missing?
'Hello, ' - declaration without type
'Hello, ' - comma expected
'Symbol' - declaration without type
'(' - comma expected
')' - semicolon expected
')' - expressions are not allowed on a global scope
```

在本例中，最后一条错误最切题：“表达式不允许在全局上下文中使用。”

2.7.4 控制语句概述

控制语句设计用来组织其他语句的非线性执行，包括声明、表达式和嵌套的控制语句。它们可以分为三种类型：

- 重复语句，也称为循环结构
- 条件语句，用于从多个备选操作分支中选择一个
- 跳转语句，在必要情况下改变前两种语句的标准行为

重复语句和选择语句包含一个头（每个头有不同的语法），后接一个受控语句。如果受控部分需要指定多个语句，它会使用复合语句。此特性不适用于跳转语句。跳转语句只移动内部指针，程序会根据特定规则（这些我们将在后续章节中讨论）基于内部指针决定当前要执行哪条语句。

在最简单的情况下，如果没有控制语句，这些语句将按照代码块中的编写顺序依次执行（一个接一个）尤其是在脚本的 `OnStart` 主函数的函数体中）。如果在代码块中遇到调用另一个函数的表达式，程序根据相同的线性原理，开始执行被调用函数内部的语句，全部执行完毕后，程序将回到调用代码块，继续执行函数调用后的下一条语句。控制语句可以显著改变这种工作逻辑。

你可以在循环结构内部使用选择语句，也可以在选择语句内使用循环结构，并且嵌套层级不受限制。然而，嵌套过多会让程序员难以理解程序。因此，建议将代码块分配（转移）到（一个或多个）函数中：在每个函数内部，保持嵌套层级不超过 2 到 3 层比较合理。

MQL5 支持以下重复语句：

- **for** 循环结构
- **while** 循环结构
- **do** 循环结构

所有循环允许一个或多个语句被执行给定的次数，或者直到满足某个布尔条件。循环语句内容执行一次就称为一次迭代。数组通常在循环结构中处理或者执行周期性的重复操作（通常在脚本或服务中）。

条件语句包括：

- 选择时基于 **if**
- 选择时基于 **switch**

前者允许你指定一个或多个条件，根据这些条件的真假来执行分配给它们的选项（一个或多个语句）后者计算整数类型的表达式，并根据其值从中选择一个选项。

最后是跳转语句，如下所示：

- **break**
- **continue**
- **return**

稍后我们将逐个详细分析。

与 C++ 不同，MQL5 不使用 **go to** 语句。

2.7.5 for 循环结构

此循环结构由带有 **for** 关键字的语句实现，因此而得名。概括地说，它可以描述如下：

```
for ( [initialization] ; [condition] ; [expression] )
    loop body
```

在标题中，在单词 **for** 之后，以下内容用圆括号括起：

- 初始化：在循环结构开始之前进行一次性初始化的语句；
- 条件：在每次迭代开始时检查的布尔条件，只要为真，循环就运行；
- 表达式：当循环体中的所有语句都已通过时，在每次迭代结束时执行的计算公式。

循环体可以是简单语句，也可以是复合语句。

头部的三个组成部分都是可选的，可按任何组合省略，也可以全部省略。

初始化可能包括变量的声明（以及初始值的设置）或给已存在的变量赋值。这种变量称为循环变量。如果它们是在头部中声明的，那么它们的作用域和生存期受循环结构的限制。

如果初始化后条件为 **true**，则循环开始执行，并在每次后续迭代开始时只要条件为真就继续执行。如果在下一次检查期间发现违反了条件，则循环退出，即控制权被转移给在循环及循环体之后编写的语句。如果在循环开始前（初始化后）条件为 **false**，则循环永远不会被执行。

条件和表达式通常包括循环变量。

执行循环意味着执行循环体。

`for` 循环结构最常见的形式是，使用单个循环变量来控制迭代次数。在下面的例子中，我们计算 `a` 数组中各数的平方。

```
int a[] = {1, 2, 3, 4, 5, 6, 7};
const int n = ArraySize(a);
for(int i = 0; i < n; ++i)
    a[i] = a[i] * a[i];
ArrayPrint(a);    // 1  4  9 16 25 36 49
// Print(i);      // error: 'i' - undeclared identifier
```

该循环按以下步骤执行：

1. 创建初始值为 0 的变量 `i`。
2. 检查条件：变量 `i` 是否小于 `n` 循环的大小。只要为真，循环就会继续。如果为假，就会跳转到调用 `ArrayPrint` 函数的语句。
3. 如果条件为真，则执行循环体的语句。在本例中，数组的第 `i` 个元素获得该元素初始值与其本身的乘积，即每个元素的值被它的平方值代替。
4. 变量 `i` 递增 1。

然后从第 2 步开始重复所有步骤。退出循环后，它的变量 `i` 被销毁，试图访问该变量将导致错误。

第 4 步的表达式可以具有任意复杂度，而不仅仅是递增循环变量。例如，要迭代偶数或奇数元素，可以编写 `i += 2`。

不管循环体由多少条语句组成，都建议将它写到与头部隔开的独立代码行中。这有利于逐步调试过程。

初始化可以包括多个变量声明，但它们必须是同一类型，因为它们属于一条语句。例如，要以相反的顺序重新排列元素，可以编写这样一个循环（这只是为了演示循环，实际上有一个内置函数 `ArrayReverse` 可以反转数组中的顺序，参见 [拷贝和编辑数组](#)）：

```
for(int i = 0, j = n - 1; i < n / 2; ++i, --j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
ArrayPrint(a);    // 49 36 25 16  9  4  1
```

辅助变量 `temp` 在每条循环路径中被创建和删除，但是编译器只在进入函数时为这个变量分配一次内存，就像所有局部变量一样。此优化对于内置类型效果很好。然而，如果在循环中描述了 [自定义类对象](#)，那么每次迭代时都会调用其构造函数和析构函数。

在循环体中改变循环变量是可以接受的，但是这种技巧只在非常特殊的情况下使用。一般不建议这样做，因为这可能会导致错误（特别是，处理过的元素可能会被跳过，或者执行可能会进入无限循环）。

为了演示可以省略头部构成，我们来设想以下问题：我们需要找出同一个数组中元素的个数，个数总和小于 100。为此，我们需要在循环的前面定义一个计数器变量 `k`，因为这个变量在循环完成后必须继续存在。我们还将创建 `sum` 变量来累加计算总和。

```

int k = 0, sum = 0;
for( ; sum < 100; )
{
    sum += a[k++];
}

Print(k - 1, " ", sum - a[k - 1]); // 2 85

```

因此，不需要在头部进行初始化。此外，在计算总和的表达式中直接使用后缀递增，将 *k* 计数器递增（访问数组元素时）。因此，标题中不需要表达式。

在循环结束时，我们打印出 *k* 以及总和减去最后添加元素，因为正是这个元素导致超过了上限 100。

请注意，即使循环体中只有一条语句，我们也在使用复合代码块。这很有用，因为当程序增长时，在括号内添加附加语句的所有操作都已经完成。此外，这种方法保证了所有循环风格统一。但是最终的选择取决于程序员。

在明确、极简版本中，循环头部可能如下所示：

```

for( ; ; )
{
    // ...          // periodic actions
    Sleep(1000); // pause the program for 1 second
}

```

如果在这样的循环中没有因某些条件而中断循环的语句，它将被无限执行。我们将分别在 *Break* 跳转语句 和 *If* 选择语句 中学习如何中断循环和测试条件。

这种循环算法通常在服务中使用（它们被设计用于持续的后台工作），以监控终端或外部网络资源的状态。它们通常包含按指定间隔暂停程序的语句，例如，使用内置函数 *Sleep*。如果没有这种预防措施，无限循环将导致一个处理器内核的负载达到 100%。

StmtLoopsFor.mq5 脚本在结尾包含一个无限循环，但这只是出于演示目的。

```

for( ; ; )
{
    Comment(GetTickCount());
    Sleep(1000); // 1000 ms

    // the loop can be exited only by deleting the script at the user's command
    // after 3 seconds of waiting we will get the message 'Abnormal termination'
}
Comment(""); // this line will never be executed

```

在循环中，计算机的内部计时器 (*GetTickCount*) 使用 *Comment* 函数每秒显示一次：值显示在图表左上角。只有用户可以通过从图表中删除整个脚本来中断循环（“专家”对话框中的“删除”按钮）。这段代码不会检查循环内部是否有此类用户停止请求，但有一个内置函数 *IsStopped* 可以实现此目的。如果用户已发出了停止命令，它将返回 *true*。尤其是在程序中包含循环和长期计算的情况下，最好能检查该函数的值，并在收到 *true* 时主动终止循环和整个程序。否则，在本例中将发生这样的情况，终端将在等待 3 秒后强制终止脚本（向日志中输出“异常终止”）。

此循环结构的更优版本应该是：


```

for( ; !IsStopped(); ) // continue until user interrupt
{
    Comment(GetTickCount());
    Sleep(1000); // 1000 ms
}
Comment(""); // will clear the comment

```

但是，使用另一个重复语句 **while** 可以更好地实现这个循环。根据经验，只有当有明显的循环变量和/或预先确定的迭代次数时，才应该使用 **for** 循环结构。在这种情况下，不满足这些条件。

循环变量通常是整数 - 尽管也允许其他类型，如 **double**。这是因为循环操作的内在逻辑隐含了迭代的编号。此外，总是可以从整数索引中计算出所需的实数，而且精度更高。例如，以下循环以 0.01 为增量在 0.0 到 1.0 的值之间迭代：

```

for(double x = 0.0; x < 1.0; x += 0.01) { ... }

```

可以用一个具有整数变量的类似循环来代替：

```

for(int i = 0; i < 100; ++i) { double x = i * 0.01; ... }

```

在第一种情况下，当加上 $x += 0.01$ 时，浮点计算的误差逐渐累积。在第二种情况下，每个 x 值在一次 $i * 0.01$ 运算中获得，具有最大可用精度。

通常给循环变量指定以下单字母名称，例如 *i, j, k, m, p, q*。当循环彼此嵌套或者在同一个循环内部同时计算前向（递增）和后向（递减）索引时，需要多个名称。

顺便说一下，这是一个嵌套循环的例子。下面的代码计算乘法表并将其存储在二维数组中。

```

int table[10][10] = {0};
for(int i = 1; i <= 10; ++i)
{
    for(int j = 1; j <= 10; ++j)
    {
        table[i - 1][j - 1] = i * j;
    }
}
ArrayPrint(table);

```

2.7.6 while 循环结构

该循环结构用 **while** 关键字描述。只要循环结构头中的逻辑表达式为真，它就会重复执行受控语句。

```

while ( condition )
    loop body

```

条件是布尔类型的任意表达式。必须存在条件。如果在循环结构开始前条件为 **false**，循环将永远不会执行。

与 C++ 不同，MQL5 不支持在 **while** 循环头中定义变量。

条件中包含的变量必须在循环结构之前定义。

循环体可以是简单语句，也可以是复合语句。

未定义迭代次数时，通常使用 **while** 循环结构。例如，要编写一个每秒输出计算机计时器计数的循环结构，可以使用 **while** 循环结构并检查停止标志（通过调用 *IsStopped* 函数），如下所示（*StmtLoopsWhile.mq5*）：

```
while(!IsStopped())
{
    Comment(GetTickCount());
    Sleep(1000);
}
Comment("");
```

此外，当循环终止条件可以与表达式中的变量修改组合时，使用 **while** 循环结构会非常方便。依次执行下一个循环，直到变量 *i* 为零（0 被视为 *false*）。

```
int i = 5;
while(--i) // warning: expression not boolean
{
    Print(i);
}
```

但是，在这种情况下，头表达式不是布尔型的（并且被隐式转换为 *false* 或 *true*）。编译器生成相关警告。创建表达式时，最好始终考虑预期的（根据规则）特征。以下是正确的循环结构版本：

```
int i = 5;
while(--i > 0)
{
    Print(i);
}
```

循环结构也可以与简单语句（无代码块）一起使用：

```
while(i < 10)
    Print(++i);
```

注意，一条简单的语句以分号结束。它还表明，头中检查变量的更改是在循环结构内部进行的。

使用循环时，务必谨慎使用无符号整数。例如，下一个循环永远不会结束，因为它的条件始终为真（理论上，编译器可以在这些地方发出警告，但实际并没有）。在零之后，计数器会“变成”一个很大的正数（*UINT_MAX*），循环继续。

```
uint i = 5;
while(--i >= 0)
{
    Print(i);
}
```

从用户的角度来看，MQL 程序会冻结（停止响应命令），但仍然会消耗资源（处理器和内存）。

while 循环结构可以像其他类型的重复语句一样嵌套。

2.7.7 do 循环结构

此循环结构类似于 **while** 循环结构，但是其条件检查发生在循环体执行之后。因此，受控语句必然至少执行一次。

两个关键字，`do` 和 `while` 用于描述循环结构：

```
do
    loop body
while ( condition ) ;
```

因此，循环头是分开的，而且逻辑条件的右括号后面必须有一个分号。不能省略条件。条件变为假时，循环结构退出。

条件中包含的变量必须在循环结构之前定义。

循环体可以是简单语句，也可以是复合语句。

下面的例子，计算从 1 开始的数字序列，其中的后一项是将前一项乘以 2 的平方根（即预定义常量 `M_SQRT2` (*StmtLoopsDo.mq5*)) 而获得。

```
double d = 1.0;
do
{
    Print(d);
    d *= M_SQRT2;
}
while(d < 100.0);
```

当数字超过 100 时，该过程终止。

2.7.8 if 选择结构

`if` 语句有几种不同形式。在最简单的情况下，如果指定的条件为真，它将执行从属语句：

```
if ( condition )
    statement
```

如果条件为假，则跳过该语句，并立即跳转到执行算法的其余部分（后续语句，如有的话）。

该语句可以是简单语句，也可以是复合语句。条件是布尔或可转换类型的表达式。

第二种形式允许指定两个操作分支：不仅包括条件为真 (`statement_A`) 的操作分支，还包括条件为假 (`statement_B`) 的操作分支：

```
if ( condition )
    statement_A
else
    statement_B
```

无论执行哪个受控语句，算法都将继续执行 `if/else` 语句下面的语句。

例如，脚本可以遵循不同的策略，这取决于它所在图表的时间范围。为此，只需分析由 `Period` 内置函数返回的值。该值是 `ENUM_TIMEFRAMES` 枚举类型。如果小于 `PERIOD_D1`，则表示短线交易，否则表示长线交易 (*StmtSelectionf.mq5*)。

```

if(Period() < PERIOD_D1)
{
    Print("Intraday");
}
else
{
    Print("Interday");
}

```

else 分支中的语句允许指定 **if** 运算符，从而将它们组成一系列连续的检查。例如，下面的代码片段可计算字符串中大写字母和标点符号（更准确地说，非拉丁字母）的数量。

```

string s = "Hello, " + Symbol();
int capital = 0, punctuation = 0;
for(int i = 0; i < StringLen(s); ++i)
{
    if(s[i] >= 'A' && s[i] <= 'Z')
        ++capital;
    else if(!(s[i] >= 'a' && s[i] <= 'z'))
        ++punctuation;
}
Print(capital, " ", punctuation);

```

该循环会遍历字符串的所有字符（编号从 0 开始）并且 **StringLen** 函数返回字符串的长度。第一个 **if** 检查每个字符，判断是否属于范围 'A' 到 'Z'，如果是，将大写计数器加 1。如果字符不在此范围内，则运行第二个 **if**，其中用于判定是否属于小写字母范围的条件 **(s[i] >= 'a' && s[i] <= 'z')** 用 **!** 运算符进行取反。也就是说，该条件表示字符不在给定的范围内。经过两次连续检查后，如果字符既不是大写字母 (**else**) 也不是小写字母 (第二个 **if**)，则我们可以断定该字符不是拉丁字母表中的字母。在这种情况下，我们将 **punctuation** 计数器的值递增。

可以用更详细的形式编写同样的检查，即采用 {...} 的代码块以提高可读性。

```

int capital = 0, small = 0, punctuation = 0;
for(int i = 0; i < StringLen(s); ++i)
{
    if(s[i] >= 'A' && s[i] <= 'Z')
    {
        ++capital;
    }
    else
    {
        if(s[i] >= 'a' && s[i] <= 'z')
        {
            ++small;
        }
        else
        {
            ++punctuation;
        }
    }
}

```

使用花括号有助于避免相关的逻辑错误，程序员仅依赖代码缩进作为参考时可能会发生这种错误。特别是，最常见的问题被称为“挂起”*else*。

当 *if* 语句嵌套时，有时会出现 *else* 分支数量少于 *if* 分支数量的情况。下面是一个例子：

```

factor = 0.0;
if(mode > 10)
    if(mode > 20)
        factor = +1.0;
else
    factor = -1.0;

```

缩进表明程序员所指的逻辑类型：当 *mode* 大于 20 时，*factor* 应为 +1，当 *mode* 介于 10 和 20 之间时，应仍然为 0，否则，应变为 -1 (*mode* <= 10)。但是代码会那样运行吗？

在 MQL5 中，假设每个 *else* 引用最近的前一个 *if* (它没有 *else*)。其结果是，编译器将按如下方式处理语句：

```

factor = 0.0;
if(mode > 10)
    if(mode > 20)
        factor = +1.0;
else
    factor = -1.0;

```

因此，*mode* 在 10 到 20 范围内，*factor* 将为 -1，对于 *mode* <= 10，则为 0。最有趣的是，无论是在编译期间还是在执行期间，程序都没有产生任何形式错误。但就是结果不正确。

为了消除这种隐蔽的逻辑问题，允许添加花括号。

```

if(mode > 10)
{
    if(mode > 20)
        factor = +1.0;
}
else
    factor = -1.0;

```

为了保持设计的一致性，如果语句的任何一个分支中至少需要一个代码块，那么最好在所有分支中都使用代码块。

当使用循环检查相等性时，要考虑到可能的输入错误，比如误将两个等号 `==` 写成一个等号 `=`。这样就把比较变成了赋值，将被赋予的值作为逻辑条件进行分析。例如，

```

// should have been x == y + 1, which would give false and skip the if
if(x = y + 1) // warning: expression not boolean
{
    // assigned x = 5 and treated x as true, so if is executed
}

```

编译器将生成一个警告“表达式不是布尔型”。

2.7.9 switch 选择结构

switch 运算符提供了从多个算法选项中选择的一个的能力。通常，选项的数量明显多于两个，否则的话，使用 **if/else** 语句会更简单。理论上，**if/else** 语句链条在很多情况下（但不是所有情况下）实现与 **switch** 等效的功能。**switch** 的一个重要特征是基于整数表达式值（通常是一个变量）选择（识别）所有选项。

一般情况下，**switch** 语句如下所示：

```

switch ( expression )
{
    case constant-expression : statements [break; ]
    ...
    [ default : statements ]
}

```

语句头文件以 **switch** 关键字开头。后面必须跟带圆括号的表达式。还必须有带花括号的代码块。

应将通过计算表达式获得的整数值指定为常量，并在 **case** 关键字后面指定。常量是指任何 **整数类型** 的字面量，例如，**int(10, 123)**，**ushort**（字符 'A'、's'、'*' 等），或者 **enum** 元素。这里常量不允许是实数、变量或表达式。

可能有许多这种 **case** 选项，也可能根本没有，这用带索引 ^{opt(n)} 的半圆括号表示。所有变量必须有唯一常量（不重复）。

对于用 **case** 声明的每个备选项，在冒号后必须编写一个语句，如果表达式的值等于相应的常量，则执行该语句。同样，语句可以是简单语句，也可以是复合语句。此外，也可以编写几个简单的语句，不用花括号括起来：这些语句仍将作为一个组执行（复合语句）。

这些语句中的后面可以带有 **break** 跳转语句。

如果有一个 **break**，则在执行了 **case** 分支前面的语句之后，**switch** 语句退出，即控制权被转移到 **switch** 下面的语句。

如果没有 **break**，下一个分支或几个分支 **case** 的语句继续执行，即直到遇到 **break** 或直到 **switch** 块结尾。这被称为“贯穿执行”。

因此，**switch** 语句不仅允许将算法执行流分解成几个替代项，还允许将它们组合起来，而 **if** 运算符不支持这一特性。另一方面，与 **if** 不同，在 **switch** 语句中，您不能选择一个值范围作为激活替代项的条件。

default 关键字允许您设置默认的计算变量，即除了所有 **cases** 常量之外的任何其他表达式值。**default** 选项要么不存在，要么仅有一个。

可以按任意顺序列 **case** 常量和 **default**。

即使还没有 **default** 分支的算法，也建议将其显式设置为空，即包含 **break**。**default** 为空将提醒你和其他程序员存在其他选项只是不太重要，否则，**default** 分支必定会报错。

具有不同常量的多个 **case** 变量可以从上到下（或从左到右）逐个列出（不带语句），但最后一个必须包含语句。这种组合的 **cases** 在图中通过索引⁽¹⁾表示。

以下是最简单也是最没有实际用处的 **switch**：

```
switch(0)
{
}
```

我们考虑一个更复杂的不同模式的例子 (*StmtSelectionSwitch.mq5*)。其中，**switch** 运算符放置在循环结构内部，以显示其工作如何由控制变量 **i** 的值决定。


```

for(int i = 0; i < 7; i++)
{
    double factor = 1.0;

    switch(i)
    {
        case -1:
            Print("-1: Never hit");
            break;
        case 1:
            Print("Case 1");
            factor = 1.5;
            break;
        case 2: // fall-through, no break (!)
            Print("Case 2");
            factor *= 2;
        case 3: // same statements for 3 and 4
        case 4:
            Print("Case 3 & 4");
            {
                double local_var = i * i;
                factor *= local_var;
            }
            break;
        case 5:
            Print("Case 5");
            factor = 100;
            break;
        default:
            Print("Default: ", i);
    }

    Print(factor);
}

```

-1 选项将会失败，因为循环结构在从 0 到 6 的范围（包含 0 和 6）内更改变量 *i*。当 *i* 等于 0 时，**default** 分支被触发。当 *i* 等于 6 时，它也会取得控制权。所有其他可能的 *i* 值根据相应的 **case** 指令进行分配。与此同时，分支 2 之后没有 **break** 语句，因此除了选项 2 的代码，选项 3 和 4 的代码也将被执行（在这种情况下，始终建议保留一个注释，说明这是有意为之）。

分支 3 和分支 4 有一个共同的语句块。但是这里需要注意的是，如果您想在一个 **case** 选项中声明一个局部变量，需要将语句放在一个嵌套的复合块内（{...}）。这里的 *local_var* 变量就是以这种方式定义的。

需要说明的是，在 **default** 分支中没有 **break** 语句。这个语句是多余的，因为在这种情况下 **default** 都是最后写的。但是，大多数程序员建议在任何选项（即使是最后一个）的末尾插入 **break**，因为在后续修改代码的过程中，它可能不再是最后一个，这时候又很容易忘记添加 **break**，那么就很可能导致程序逻辑出现错误。

如果在 **switch** 中没有 **default**，并且头表达式与任何 **case** 常量都不匹配，则跳过整个 **switch**。

作为脚本执行的结果，我们将在日志中收到以下消息：

```

Default: 0
1.0
Case 1
1.5
Case 2
Case 3 & 4
8.0
Case 3 & 4
9.0
Case 3 & 4
16.0
Case 5
100.0
Default: 6
1.0

```

2.7.10 中断跳转

break 运算符用于提前终止 *for*、*while*、*do* 循环，以及从 *switch* 选择语句退出。该运算符只能用于指定的语句，如果有多个嵌套语句，则只作用于包含 **break** 的语句。执行完 **break** 语句后，程序继续执行被中断循环或 *switch* 后面的语句。

语法非常简单：**break** 关键字和一个分号：

```
break ;
```

当在循环内部使用时，**break** 通常在 *if/else* 条件运算符的一个分支内实现。

考虑一个每秒打印一次当前系统时间计数器，但不超过100次的脚本。它提供了用户对进程中断的处理：为此，*IsStopped* 函数在条件运算符 *if* 中被轮询，并且它的从属语句包含 **break** (*StmtJumpBreak.mq5*)。

```

int count = 0;
while(++count < 100)
{
    Comment(GetTickCount());
    Sleep(1000);
    if(IsStopped())
    {
        Print("Terminated by user");
        break;
    }
}

```

在下面的例子中，用乘法表填充对角矩阵（右上角始终用零填充）。

```

int a[10][10] = {0};
for(int i = 0; i < 10; ++i)
{
    for(int j = 0; j < 10; ++j)
    {
        if(j > i)
            break;
        a[i][j] = (i + 1) * (j + 1);
    }
}
ArrayPrint(a);

```

当内部循环变量 *j* 大于外部循环变量 *i* 时, *break* 语句中断内部循环。当然,这并不是对角填充矩阵的最佳方式:从 0 到 *i* 无任何 *break* 的循环 *j* 会更容易,但这里它验证了有 *break* 和无 *break* 的等效结构的存在。

虽然这种现象在生产项目中可能不那么明显,但是建议尽可能避免使用 *break* 运算符,并用附加变量(例如,一个名称“一目了然”的布尔变量 *needAbreak*)来代替它,这些变量应该用在循环文件头中的终端表达式中,以便按标准的方式中断它们。

假设使用两个嵌套循环来查找字符串中的重复字符。第一个循环按顺序使字符串中的每个字符成为当前字符,第二个循环(向右)遍历其余的字符。

```

string s = "Hello, " + Symbol();
ushort d = 0;
const int n = StringLen(s);
for(int i = 0; i < n; ++i)
{
    for(int j = i + 1; j < n; ++j)
    {
        if(s[i] == s[j])
        {
            d = s[i];
            break;
        }
    }
}

```

如果位置 *i* 和 *j* 处的字符匹配,记住这个匹配字符并通过 *break* 退出循环。

可以假设变量 *d* 在这个片段执行后包含字母 'l'。但是,如果把脚本套用在最流行的金融工具“EURUSD”上,答案会是 'U'。这是因为 *break* 只中断内部循环,在找到第一个匹配项(单词“Hello”中的 'l')后,从 *i* 处继续循环。因此,要同时从多个嵌套循环中退出,必须采取额外的措施。

最流行的方法是将一个(填充在内部循环中的)变量包含在外部循环(或所有外部循环)的条件中。在我们的例子中,已经有这样一个变量:*d*。

```

for(int i = 0; i < n && d == 0; ++i)
{
    for(int j = i + 1; j < n; ++j)
    {
        if(s[i] == s[j])
        {
            d = s[i];
            break;
        }
    }
}

```

检查 *d* 是否等于 0，在找到第一个 0 之后立即停止外部循环。但是可以将这个检查添加到无需使用 *break* 的内部循环中。

```

for(int i = 0; i < n && d == 0; ++i)
{
    for(int j = i + 1; j < n && d == 0; ++j)
    {
        if(s[i] == s[j])
        {
            d = s[i];
        }
    }
}

```

2.7.11 继续跳转

continue 语句中断包含 *continue* 的最内层循环的当前迭代，并启动下一次迭代。该语句只能在 *for*、*while* 和 *do* 循环内部使用。*for* 内部 *continue* 的执行导致循环头文件中表达式的下一次计算（循环变量的递增/递减），之后检查循环继续条件。在 *while* 或者 *do* 内部执行 *continue* 会立即检查循环文件头中的条件。

该语句由 *continue* 关键字和一个分号组成：

```
continue ;
```

它通常被放置在 *if/else* 或 *转换* 条件语句的一个分支中。

例如，我们可以生成一个有空格的乘法表：当两个索引的乘积为奇数时，对应的数组元素将保持为零 (*StmtJumpContinue.mq5*)。

```

int a[10][10] = {0};
for(int i = 0; i < 10; ++i)
{
    for(int j = 0; j < 10; ++j)
    {
        if((j * i) % 2 == 1)
            continue;
        a[i][j] = (i + 1) * (j + 1);
    }
}
ArrayPrint(a);

```

这是计算一个数组的正元素之和的方法。

```

int b[10] = {1, -2, 3, 4, -5, -6, 7, 8, -9, 10};
int sum = 0;
for(int i = 0; i < 10; ++i)
{
    if(b[i] < 0) continue;
    sum += b[i];
}
Print(sum); // 33

```

请注意，相同的循环可以不用 *continue* 进行重写，但会使用更多的代码块嵌套：

```

for(int i = 0; i < 10; ++i)
{
    if(b[i] >= 0)
    {
        sum += b[i];
    }
}

```

因此，*continue* 运算符通常用于简化代码格式（特别是需要通过几个条件时）。然而，选择哪种方法属于个人偏好问题。

2.7.12 返回跳转

return 运算符设计用于从 [函数](#) 返回控制。假设所有的可执行语句都在一个特定的函数内部，它可以被用来间接中断包含它的任何嵌套层次的 *for*、*while* 和 *do* 循环。应该考虑到，与 *continue*，尤其是 *break* 不同，函数内部中断的循环之后的所有语句都将被忽略。

return 运算符和语法：

```
return ([expression]) ;
```

指定表达式的需要是由函数签名决定的（关于这一点的更多内容将在 [相关章节中讨论](#)）。为了全面理解 *return* 在控制语句的上下文中的工作原理，让我们来看看主脚本函数 *OnStart* 的一个例子。由于它属于 *void* 类型，即不返回任何内容，因此运算符采用以下形式：

```
return ;
```

在 [break 相关章节中](#)，我们实施了一个算法来查找字符串中的重复字符。要中断两个嵌套循环，我们不仅要使用 *break*，还要修改外部循环的条件。

使用 `return` 操作符，这就可以用一种更简单的方式来完成 (*StmtJumpReturn.mq5*)。

```
void OnStart()
{
    string s = "Hello, " + Symbol();
    const int n = StringLen(s);
    for(int i = 0; i < n; ++i)
    {
        for(int j = i + 1; j < n; ++j)
        {
            if(s[i] == s[j])
            {
                PrintFormat("Duplicate: %c", s[i]);
                return;
            }
        }
    }

    Print("No duplicates");
}
```

如果在 `if` 运算符中发现等号，我们显示符号并退出函数。如果该算法在一个非 `OnStart` 函数的自定义函数内，我们可以为它定义一个返回类型（例如，`ushort` 而不是 `void`），并使用完整的 `return` 形式将找到的字符传递给调用代码。

由于已知双字母 'J' 存在于测试字符串中，所以循环 (`Print`) 之后的语句将不被执行。

2.7.13 空语句

空语句是语言中最简单的语句。它只包含一个字符，即分号；

在程序中，当语法上需要有语句但算法逻辑并不指示任何操作的时候，会使用空语句。

例如，以下 `while` 循环用于查找字符串中的空格。整个算法是直接循环文件头中执行的，所以它的主体必须是空的。我们可以写一个带花括号的空代码块，这里也可以是空语句。(*StmtNull.mq5*)。

```
int i = 0;
ushort c;
string s = "Hello, " + Symbol();
while((c = s[i++]) != ' ' && c != 0); // intentional ';' (!)
if(c == ' ')
{
    Print("Space found at: ", i);
}
```

请注意，如果省略了 `while` 头文件末尾的分号（可能是意外省略的），那么 `if` 语句将被视为循环体。其结果是，`Print` 函数不会输出到日志中。事实上，即使没有明显的错误，程序也不会正常运行。

反着来也行：在循环头文件后面多加一个分号（本不应该出现在这里）会把循环体从循环文件头中“分离”出来，也就是说，只有一个空语句会在循环中执行。

为此，应该检查代码中的分号，并且在有意放置它们的地方，留下解释性的注释。

根据正式的观点，空语句也用于 *for* 语句 - 当我们不初始化表达式时。实际上，初始化总是必须的：

```
for ( [initialization] ; [end loop condition]; [post-expression] )
    loop body
```

第一个字符 `;` 是初始化语句的一部分，初始化语句可以是表达式或空语句：两者都包含字符 `;`；而后者只包含 `;`。因此，实现了可选性（空性）。

2.8 函数

函数是带有语句的命名代码块。程序的几乎所有应用算法都包含在函数中。在函数外部，只是执行一些辅助操作，比如创建和删除全局变量。

当我们调用函数时，函数中的语句就会执行。当各种事件发生时，终端会自动调用一些主要的函数。也称为 MQL 程序入口点或事件处理程序。特别是我们已经知道，在图表上运行脚本时，终端会调用它的主函数 *OnStart*。在其他类型的程序中，终端还会调用其他函数，我们将在涉及 MQL5 API 交易架构的 [第五章](#) 和 [第六章](#) 详细讨论。

在本章中，我们将学习如何定义和声明一个函数，如何描述函数并为其传递参数，以及如何从函数返回它的运行结果。

我们还将讨论函数重载，即提供多个同名函数的能力，以及相关用途。

最后，我们将认识一种新的类型：指向函数的指针。

2.8.1 函数定义

函数定义包括返回的值类型、标识符、参数列表（用圆括号括起）和主体，主体是指包含语句的代码块。列表中的各参数用逗号分隔。每个参数都被赋予一个类型、一个名称和一个默认值（可选）。

```
result_type function_identifier ( [parameter_type parameter_identifier
                                = value_by_default] ,... )
{
    [statement]
    ...
}
```

允许创建不带参数的函数：那么就没有列表，并且函数名后面带空括号（不能省略括号）。或者也可以在括号中间写入 *void* 关键字，以强调没有参数。例如，像这样：

```
void OnStart(void)
{
}
```

列表中参数的返回类型、数量和类型的组合称为函数原型或签名。不同的函数可以有相同的原型。

在前面的章节中，我们已经看到了 *OnStart* 和 *Greeting* 之类的函数定义。现在让我们尝试将计算斐波那契数作为一个测试函数。斐波那契数的计算公式如下：


```
f[0] = 1
f[1] = 1
f[i] = f[i - 1] + f[i - 2], i > 1
```

第一和第二个数字是 1, 后面所有数字都是其前两个数字之和。我们给出数字串的开头: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

可以使用以下函数 (*FuncFibo.mq5*) 计算给定索引所指的字数。

```
int Fibo(const int n)
{
    int prev = 0;
    int result = 1;
    for(int i = 0; i < n; ++i)
    {
        int temp = result;
        result = result + prev;
        prev = temp;
    }
    return result;
}
```

取一个 *int* 类型的参数 *n* 并返回一个 *int* 类型的结果。*n* 参数有 *const* 修饰符, 因为我们不打算在函数内部改变 *n* (这种对变量“权限”约束的显式声明值得提倡, 因为有助于避免随机错误)。

局部变量 *prev* 和 *result* 将存储序列中最后两个数字的当前值。在对 *i* 的循环过程中, 我们计算它们的和, 获取序列的下一个数。此前, 旧值 *result* 被写入变量 *temp*, 因此在求和后, 它被传递到 *prev*。

执行给定次数的循环后, *result* 变量包含预期数字。我们使用 *result* 语句从函数中返回这个数。

函数的输入参数也是一个局部变量, 它将在函数调用期间被初始化为实际值。通过函数调用, 该值从语句“外部”传递。

参数名必须是唯一的, 并且不能与局部变量名相同。

函数体是一个代码块, 定义了局部变量的作用域和生存期。它们的定义和操作原理将在 [声明/定义语句](#) 和 [初始化](#) 章节中介绍。

2.8.2 函数调用

当一个函数的名称在表达式中被提到时, 这个函数就被调用了。在名称后面应该有一对括号, 在括号中标明了函数参数对应的自变量 (如果其定义中有一个参数列表的话), 用逗号隔开。

稍后, 我们将看到 [函数指针](#) 类型, 可用于创建指向具有特定特征的函数的变量, 然后调用时使用这个变量 (而不是函数名)。

继续 *Fibo* 函数的例子, 让我们从 *OnStart* 函数调用它。为此, 让我们创建一个 *f* 变量来存储计算结果数, 并在其初始化表达式中指定 *Fibo* 函数的名称和一个整数 (例如 10) 作为自变量 (放在圆括号中)。

```

void OnStart()
{
    int f = Fibo(10);
    Print(f); // 89
}

```

我们不需要创建一个变量来接收函数的值。相反，您可以从表达式中直接调用该函数，例如“2*Fibo(10)”或“Print(Fibo(10))”。那么它的值将在调用时被代入表达式。这里引入了辅助变量 *f*，用于在单独的语句中实现调用操作和返回值。

调用过程包括以下步骤：

- 暂停调用 (*OnStart*) 函数的语句序列的执行；
- 自变量的值进入被调用函数 (*Fibo*) 的输入参数 *n*；
- 开始执行其语句；
- 执行完成时，它将结果返回 (记得包含 *return* 语句)；
- 结果被写入变量 *f*；
- 之后，继续执行 *OnStart* 函数，也就是将数字打印到日志中 (*Print*)。

对于每一次函数调用，编译器都会生成辅助二进制代码 (程序员不需要为此操心)。此代码的思想是，在调用函数之前，它将程序中的当前位置推送到栈上，而在调用完成之后，检索这个位置并将它返回到函数调用之后的语句。如果一个函数调用另一个函数，而后者再调用另一个函数 (即第二个函数调用第三个函数)，以此类推，所有被调用函数层次结构中转换的返回地址都在栈上累积 (栈因此而得名)。随着嵌套函数调用的处理，栈将以相反的顺序被清除。注意，栈也为每个函数的局部变量分配内存。

2.8.3 参数和自变量

通过调用传递给函数的自变量，是相应函数参数的初始值。自变量的个数、次序和类型必须与函数原型相匹配。但是，自变量计算的顺序没有定义 (参见 [基本概念](#) 一节)。根据源代码的具体情况和优化考虑，编译器可能会选择一个对它来说较方便的方法。例如，对于一个包含两个自变量的列表，编译器可能会先计算第二个自变量，然后再计算第一个。只能保证在调用之前会完成对两个自变量的计算。

如有必要，每个自变量都以与 [变量初始化](#) 相同的方式，使用 [隐式强制转换](#)，映射到相应的参数。在函数启动之前，保证所有参数都有指定值。例如，根据传递的自变量，调用 *Fibo* 函数可能会导致以下结果 (在注释中说明)：

```

// warnings
double d = 5.5;
Fibo(d);           // possible loss of data due to type conversion
Fibo(5.5);         // truncation of constant value
Fibo("10");        // implicit conversion from 'string' to 'number'
// errors
Fibo();            // wrong parameters count
Fibo(0, 10);       // wrong parameters count

```

所有警告都与编译器执行的隐式转换有关，因为值类型与参数类型不匹配。应将其视为潜在的错误并消除。当自变量过少或过多时，会出现错误“参数数量不正确”。

理论上，函数参数不必有名称，即，类型本身就足以描述参数。这听起来很奇怪，因为如果函数内部没有名字，我们将无法访问参数。不过在创建基于一些标准接口的程序时，有时不得不编

写一些与给定原型相对应的函数。在这种情况下，函数内部的一些参数就没有必要了。为了明确指出这一事实，程序员可以省略这类参数的名字。例如，MQ5 API 需要使用以下原型实现 *OnDeinit* 事件处理程序函数：

```
void OnDeinit(const int reason);
```

如果我们不需要函数代码中的 *reason* 参数，则可以在说明中省略它：

```
void OnDeinit(const int);
```

终端事件处理函数通常由终端自己调用，但是如果我们需从代码中调用类似的函数（带有匿名参数），则需要传递所有自变量，无论参数是否被命名。

2.8.4 值参数和引用参数

自变量可以通过两种方式传递给函数：通过值传递和通过引用传递。

到目前为止，我们看到的所有情况都是通过值传递的。这种方法意味着，通过调用代码片段准备的自变量值被复制到一个新变量中，也就是与函数对应的输入变量。否则，自变量和输入变量无法关联起来。函数内部变量的所有后续操作都不会以任何方式影响自变量。

为说明引用参数，请在类型的右侧添加一个 `&` 符号。许多程序员喜欢在参数名后面附加一个 `&` 符号，从而强调参数是对给定类型的引用。例如，以下各项是等效的：

```
void func(int &parameter);
void func(int & parameter);
void func(int& parameter);
```

函数被调用时，不会为引用参数创建对应的局部变量。相反，为该参数指定的自变量，在函数内部以输入参数的名称（别名）出现。因此，该值不会被拷贝，而是在内存中的相同地址处使用。因此，对函数中参数的修改体现在其关联自变量的状态。由此出现一个重要的特征。

只能指定一个变量（左值，参见 [赋值运算符](#)）作为引用参数的自变量。否则，我们将遇到错误“参数作为引用传递，应为变量”。

按引用传递用于以下几种情况：

- 通过消除值拷贝来提高程序的效率；
- 在使用 `return` 返回单个值还不够的情况下，将修改后的数据从函数传递给发起调用的代码；

第一点对于潜在的大型变量（如字符串或数组）尤其重要。

为了区分引用参数的第一个和第二个用途，如果作者不希望改变函数内部的参数时，则建议添加 `const` 修饰符。这将提醒您并向其他开发人员表明，在函数内部传递变量不会带来副作用。

如果可以对引用参数应用 `const` 修饰符而未应用，则会导致整个函数调用层次结构出现问题。事实上，调用这样的函数需要非常量自变量。否则，将出现错误“常量变量不能作为引用传递”。结果可能会逐渐导致为了代码的可编译性，而不得不从所有函数中的所有参数中去掉 `const` 修饰符。这实际上扩大了潜在 bug 的范围，无意中破坏了变量。应该以相反的方式纠正这种情况：在不需要返回和修改值的地方使用 `const`。

为了比较在 *FuncDeclaration.mq5* 脚本中传递参数的几种方式，实现了几个函数：*FuncByValue* —按值传递，*FuncByReference*—按引用传递，*FuncByConstReference* —按常量引用传递。

```

void FuncByValue(int v)
{
    ++v;
    // we are doing something else with v
}

void FuncByReference(int &v)
{
    ++v;
}

void FuncByConstReference(const int &v)
{
    // error
    // ++v; // 'v' - constant cannot be modified
    Print(v);
}

```

在 **OnStart** 函数中，我们调用所有这些函数，并观察它们对用作自变量的 *i* 变量的影响。注意，通过引用传递参数不会改变函数调用语法。

```

void OnStart()
{
    int i = 0;
    FuncByValue(i);           // i cannot change
    Print(i);                 // 0
    FuncByReference(i);       // i is changing
    Print(i);                 // 1
    FuncByConstReference(i);  // i cannot change, 1
    const int j = 1;
    // error
    // 'j' - constant variable cannot be passed as a reference
    // FuncByReference(j);

    FuncByValue(10);          // ok
    // error: '10' - parameter passed as reference, variable expected
    // FuncByReference(10);
}

```

字面量只能传递给 **FuncByValue** 函数，因为其他函数需要引用（即变量）作为自变量。

FuncByReference 函数不能用变量 *j* 调用，因为后者被声明为常量，并且该函数声明了改变其参数的能力（或意图），因为它没有 **const** 修饰符。这将产生错误“常量变量不能作为引用传递”。

脚本还介绍了 **Transpose** 函数：该函数用于转置一个 2x2 矩阵，该矩阵通过引用作为二维数组传递。

```
void Transpose(double &m[][2])
{
    double temp = m[1][0];
    m[1][0] = m[0][1];
    m[0][1] = temp;
}
```

从 *OnStart* 函数调用演示了本地数组 **a** 内容的预期变化。

```
double a[2][2] = {{-1, 2}, {3, 0}};
Transpose(a);
ArrayPrint(a);
```

在 MQL5 中，数组参数总是作为动态数组的内部结构传递（参见 [数组特性](#) 一节）。因此，这种参数的说明必须在第一维中具有开放大小，即第一对方括号内为空。

这并不妨碍在必要时向函数传递实际自变量，实际自变量是一个定长数组（如我们的例子所示）。然而，*ArrayResize* 之类的函数无法调整大小或以其他方式重新组织这种掩码定长数组。

除了第一维之外，数组在所有维度上的大小都必须与参数和自变量相匹配。否则，我们将遇到错误“不允许参数转换”。具体来说，*TransposeVector* 函数在例子中定义如下：

```
void TransposeVector(double &v[])
{
}
```

尝试对二维数组 **a** 调用该函数在 *OnStart* 中被注释掉了，因为这生成了上面的错误：数组维数不匹配。

除了通过值或引用来传递参数，还有另一种方法：传递指针。与 C++ 不同，MQL5 只支持用于对象类型的 [指针用于（类）](#)。我们将在第三章研究这个特性。

2.8.5 可选参数

MQL5 允许在描述函数时为参数指定默认值。为此，可使用 [初始化](#) 语法，即在参数右边（= 符号之后）输入对应类型的字面量。例如：

```
void function(int value = 0);
```

调用函数时，可以省略这些参数的自变量。那么它们的值将被设置为默认值。这样的参数称为可选参数。

可选参数必须在参数列表的末尾。也就是说，如果第 *i* 个参数是用初始化声明的，那么所有后续参数也必须这样。否则，将显示编译错误“参数缺少默认值”。下面的函数说明就有这个问题。

```
double Largest(const double v1, const double v2 = -DBL_MAX,
               const double v3);
```

提供了两种解决方案：一个是 **v3** 参数也必须有默认值，一个是 **v2** 参数必须成为强制参数。

从右向左调用函数时，只能省略可选自变量。即如果函数有两个参数，并且都是可选的，那么在调用时，不能跳过第一个，而且要指定第二个。传递的单个值将与第一个参数匹配，第二个将被视为忽略。即使两个自变量都丢失，仍然需要空的圆括号。

分析寻找三个数中最大值的函数。第一个参数是必选的，后两个是可选的，默认等于 `double` 类型的最小可能数值。因此，在没有显式传递值的情况下，它们中的每个参数肯定会小于（或者极端情况下，等于）所有其他参数。

```
double Largest(const double v1, const double v2 = -DBL_MAX,
               const double v3 = -DBL_MAX)
{
    return v1 > v2 ? (v1 > v3 ? v1 : v3) : (v2 > v3 ? v2 : v3);
}
```

以下是调用方法：

```
Print(Largest(1));           // ok: 1
Print(Largest(0, -2));       // ok: 0
Print(Largest(1, 2, 3));     // ok: 3
```

利用可选参数，MQL5 在自定义函数中实现了具有可变数量参数的函数概念。

与 C++ 不同，MQL5 不支持使用省略语法来定义具有可变数量参数的函数。同时，MQL5 API 中的某些内置函数在描述时就使用了省略号，同时接受可变数量的任意参数。例如，`Print` 函数。它的原型是这样的：`void Print(argument, ...)`。因此，我们可以使用最多 64 个自变量（由逗号分隔，不包括数组）来调用这个函数，这些会在日志中显示。

2.8.6 返回值

函数可以返回内置类型的值，包含内置类型字段的 **结构体**，以及 **指向函数的指针** 和指向 **类对象** 的指针。类型名写在函数定义中的名称之前。如果函数不返回任何内容，则应被赋予 `void` 类型。

若要从数组函数返回，必须使用通过引用传递的参数（请参见 **值参数和引用参数**）。

使用 `return` 语句返回一个值，其中在 `return` 关键字后指定了一个表达式。可以使用两种形式中的任何一种：

```
return expression ;
```

或者：

```
return ( expression ) ;
```

如果函数的类型为 `void`，那么 `return` 语句将简化为：

```
return ;
```

`return` 语句不能在 `void` 函数内部包含任何表达式：编译器将生成一个错误“`return` - `void` 函数返回一个值”。

对于这样的函数，理论上不需要在函数体的代码块结尾使用 `return`。我们在 `OnStart` 函数的例子中看到了这一点。

如果函数的类型不是 `void`，那么必须用 `return` 语句。如果不存在，将出现编译错误“并非所有控制路径都返回值”。

```
int func(void)
{
    if(IsStopped()) return; // error: function must return a value
                           // error: not all control paths return a value
}
```

需要注意的是，一个函数体可以有多个 `return` 语句。特别是在根据条件而提前退出的情况下。任何 `return` 语句都会在其所在的位置中断函数的执行。

如果函数必须返回一个值（因为它不是 `void` 类型函数），并且没有在 `return` 运算符中指定，编译器将生成错误“函数必须返回一个值”。下面给出了 `func` 函数的编译器更正版本 (`FuncReturn.mq5`)。

```
int func(void)
{
    if(IsStopped()) return 0;
    return 1;
}
```

如果返回值与指定的函数类型不一致，编译器将尝试进行 **隐式转换**。如果类型需要显式转换，将会生成错误。

为了返回值，会隐式创建一个临时变量，并提供给调用代码使用。

在我们学习了对象类型（参见 **类一章**）和从函数返回对象指针的功能后，我们将回头分析如何安全地传递对象指针。与 C++ 不同，MQL5 中的函数不能返回引用。试图在结果类型中声明带有 `&` 符号的函数会导致错误“不能使用 `&` 引用”。

2.8.7 函数声明

函数声明说明了一个原型，但没有指定函数体。放置了一个分号，而不是一个有主体的代码块。

声明对于编译器来说是必要的，这样编译器才能在后续代码片段中检查函数调用是否正确（包括按名称调用、向它传递自变量和获得结果）。

整个函数定义（包括函数体）也是一个声明，所以除了定义之外不需要声明函数。

例如，上述 `Fibo` 函数的声明是这样的。

```
int Fibo(const int n);
```

从几个带有源文本的文件构建程序时，将使用单独的函数声明和定义：然后在扩展名为 `mqh` 的头文件中进行声明（参见 **#include 预处理程序指令一节**），这个头文件包含在使用该函数的文件中，函数定义仅在其中一个文件中实现。函数签名在声明和定义中的匹配提供了防错功能。也就是说，单个声明保证了对整个源代码所做更改的一致性。

如果我们声明一个函数并在代码的某处调用该函数，但没有为其提供完全合适的定义，编译器将抛出一个错误：“函数‘名称’必须有一个主体”。这种情况经常发生，比如因为声明或定义中有拼写错误或不准确，以及在更改源代码的过程中，一些更正已经完成，而另一些很可能被遗忘。

如果函数被声明但没有在任何地方使用，编译器也不需要这个函数的定义 - 这个元素就从二进制程序中“切掉”了。

在 [声明/定义语句](#) 一节中，我们分析了 `Init` 函数的一个例子 (`StmtDeclaration.mq5` 脚本)，该函数用于初始化变量。其中特别说明了一个问题，全局变量 `k` 不能在 `Init` 函数之前定义，因为初始值 `k` 是通过调用 `Init` 获得的。编译器会抛出错误“`Init` 是未知标识符”。

现在我们知道，这样的问题可以通过一个声明来解决。在 `FuncDeclaration.mq5` 脚本中，我们在 `k` 变量之前添加了以下 `Init` 函数的前向声明，并将 `Init` 定义留在 `k` 之后。

```
// preliminary declaration
int Init(const int v);
// before adding preliminary declaration above
// here was an error: 'Init' is an unknown identifier
int k = Init(-1);
int Init(const int v)
{
    Print("Init: ", v);
    return v;
}
```

现在脚本可以正常编译了。从技术上讲，在这种情况下，我们只需将函数移到变量上方，而不需要预先声明。这样做是为了解释这个概念。然而，在某些语言元素相互依赖的情况（如类）下，即使是在同一文件中也必须使用前置声明。

2.8.8 递归

允许从函数内部的语句调用同一个函数。这种调用称为递归调用。

我们回到计算斐波那契数的例子。根据将每个数计算为前两个数之和的公式（前两个数除外，它们都等于 1），就很容易写出计算斐波那契数的递归函数。

```
int Fibo(const int n)
{
    if(n <= 1) return 1;

    return Fibo(n - 1) + Fibo(n - 2);
}
```

递归函数必须能够在递归结束的情况下返回控制权，类似我们在条件语句 `if` 中索引 0 和 1 的情况。否则，函数调用序列可能会无限期地继续下去。实际上，因为未完成的函数调用会在称为栈的有限内存区域中累积（参见 [声明/定义语句](#) 一节，以及 [描述数组](#) 一节中的“堆”和“栈”侧栏），函数迟早会因“栈溢出”运行时错误而终止。这个问题在 `FiboEndless` 函数中表现出来。

```
int FiboEndless(const int n)
{
    return FiboEndless(n - 1) + FiboEndless(n - 2);
}
```

请注意，这不是一个编译错误。在这种情况下，编译器甚至不会生成警告（尽管从技术层面上来是可以的）。该错误发生在脚本执行期间。它将被打印终端中的 **Experts** 日志。

从函数本身调用一个函数时会发生递归。例如，如果 `F` 函数调用 `G` 函数，而 `G` 函数又调用 `F` 函数，这种情况就是间接递归。因此，递归可以是任何深度的循环调用的结果。

2.8.9 函数重载

MQ15 允许在同一源代码中定义同名但参数数量或类型不同的函数。这种方法被称为函数重载。如果不同输入可以触发相同操作，则通常应用这种方法。签名的差异允许编译器根据传递的自变量自动确定调用哪个函数。但有一些具体要注意的细节。

函数不能仅仅是返回类型不同。如果只是返回类型不同，并不会触发重载机制，并返回错误“函数已经定义并且具有不同的类型”。

如果同名函数有不同数量的参数，并且“额外的”参数被声明为可选，那么编译器无法确定调用哪个参数。这将产生错误“对具有相同参数的重载函数的调用不明确”。

当调用重载函数时，编译器匹配可用重载中的自变量和参数。如果没有找到完全匹配，编译器会尝试添加/移除 `const` 修饰符，并执行数字类型扩展和 [算术转换](#)。在 [对象指针](#) 的情况下，使用类继承规则。

使用相同位置处不同数量的参数或不相关的参数类型（例如一个数字和一个字符串），通常很容易选择。但是，如果要将参数类型隐式地从一种类型转换到另一种类型，可能会出现歧义。

例如，这里有两个求和函数：

```
double sum(double v1, double v2)
{
    return v1 + v2;
}

int sum(int v1, int v2)
{
    return v1 + v2;
}
```

那么下面的调用将导致一个错误：

```
sum(1, 3.14); // overloaded function call is ambiguous
```

对于每次重载，编译器都会认为有错误：对于 `double sum(double v1, double v2)` 函数，需要隐式地将第一个自变量转换为 `double`，而对于 `int sum(int v1, int v2)`，需要转换 `int` 中的第二个自变量。

可以这样理解术语“重载 (overload)”：一个重复使用的名称过度 (over) “承担” (load) 了比常规名称 (常规名称只用于一个函数) 重数倍的“职责”。

我们来试试重载矩阵转置的函数。我们已经有一个 `2x2` 数组的例子 (参见 [值参数和引用参数](#))。让我们对一个 `3x3` 数组实施同样的操作。多维数组的高维度 (非零) 参数的大小会改变类型，如 `double [][2]` 不同于 `double [][3]`。因此，我们将重载旧版本的函数：

```
void Transpose(double &m[][2]);
```

添加一个新函数 (`FuncOverload.mq5`):

```
void Transpose(double &m[][3]);
```

在新版本的实现中，使用 `Swap` 辅助函数来互换给定索引处的两个矩阵元素很方便。

```

void Transpose(double &m[][3])
{
    Swap(m, 0, 1);
    Swap(m, 0, 2);
    Swap(m, 1, 2);
}

void Swap(double &m[][3], const int i, const int j)
{
    static double temp;

    temp = m[i][j];
    m[i][j] = m[j][i];
    m[j][i] = temp;
}

```

对不同大小的数组使用相同表示法，现在我们可以从 *OnStart* 中调用这两个函数。编译器自己会生成对正确版本的调用。

```

double a[2][2] = {{1, 2}, {3, 4}};
Transpose(a);
...
double b[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
Transpose(b);

```

须注意，尽管参数上的 *const* 修饰符改变了函数的原型，但这个差异不足以构成重载。如果两个同名函数的区别仅在于某个参数有无 *const*，它们可以被认为是相同的。这将导致错误“函数已经定义并且有函数体”。发生这种错误是因为，对于值参数，在赋值自变量时 *const* 修饰符被丢弃（因为根据定义，值参数不能改变发起调用的代码中的自变量），这将不允许基于这种情况选择一个重合函数。

为了证明这一点，在脚本中添加一个函数就足够了：

```
void Swap(double &m[][3], int i, int j);
```

对于现有函数，这是一个不成功的重载：

```
void Swap(double &m[][3], const int i, const int j);
```

这两个函数之间的唯一区别是 *i* 和 *j* 参数的 *const* 修饰符。因此，它们都适合用 *int* 类型的自变量调用和按值传递。

按引用传递参数时，仅通过 *const*/非 *const* 属性差异实现的重载会成功，因为对于引用而言，*const* 修饰符至关重要（它会改变类型并消除隐式转换的可能性）。这一点在脚本中通过几个函数得到了证明：

```

void SwapByReference(double &m[][3], int &i, int &j)
{
    Print(__FUNCSIG__);
}

void SwapByReference(double &m[][3], const int &i, const int &j)
{
    Print(__FUNCSIG__);
}

void OnStart()
{
    // ...
    {
        int i = 0, j = 1;
        SwapByReference(b, i, j);
    }
    {
        const int i = 0, j = 1;
        SwapByReference(b, i, j);
    }
}

```

它们生成几乎为空的存根，其中每个函数的签名都是使用 `Print(__FUNCSIG__)` 调用打印的。这样可以确保根据自变量的 `const` 属性调用相应版本的函数。

2.8.10 函数指针 (typedef)

MQL5 具有 `typedef` 关键字，可用于描述一种特殊类型的函数指针。

`typedef` 在 C++ 中有更广泛的应用，而在 MQL5 中，`typedef` 只用于函数指针。

新类型声明的语法如下：

```
typedef function_result_type ( *function_type ) ( [list_of_input_parameters] ) ;
```

`function_type` 标识符定义了一个类型名称，该名称成为指向任意函数的指针的同义词（别名），这类函数返回给定类型 `function_result_type` 的值，并接受输入参数列表 (`list_of_input_parameters`)。

例如，我们可以有两个具有相同原型的函数（两个 `double` 类型的输入参数，结果类型也是 `double`），它们执行不同的算术运算：加法和减法 (`FuncTypedef.mq5`)。

```

double plus(double v1, double v2)
{
    return v1 + v2;
}

double minus(double v1, double v2)
{
    return v1 - v2;
}

```

它们的通用原型很容易描述为指针用途：

```
typedef double (*Calc)(double, double);
```

该条目将 *Calc* 类型引入到程序中，使用该条目，你可以定义一个变量/参数，用于存储/传递对具有这种原型的任何函数的引用，包括 *plus* 和 *minus* 函数。此为指针类型，因为说明中使用了字符 **(*Calc)*。在学习 OOP 的时候，我们会深入学习指针星号的特性。

使用这样的指针类来创建自定义算法是很方便的，这些算法可以根据输入数据动态调用与别名相对应的不同函数。

特别是，我们可以引入一个通用的计算器函数：

```
double calculator(Calc ptr, double v1, double v2)
{
    if(ptr == NULL) return 0;
    return ptr(v1, v2);
}
```

它的第一个参数是用 *Calc* 类型声明的。由于这一点，我们可以传递一个带有合适原型的任意函数给它，并因此执行一些操作，而 *calculator* 函数并不知道这些操作的本质。计算器函数通过调用委托给指针来实现这一点：*ptr(v1, v2)*。因为 *ptr* 是一个函数指针，所以这种语法不仅类似于函数调用，而且实际上调用指针所保存的函数。

注意，我们根据特殊值 *NULL* (*NULL* 相当于指针的零)预先检查 *ptr* 参数。事实是指针可能不指向任何位置，也就是说可能没有被初始化。因此，在脚本中，我们描述了一个全局变量：

```
Calc calc;
```

它没有指针。如果不是针对 *NULL* 的“保护”，用“空”指针 *calc* 调用 *calculator* 会导致运行时错误“函数指针调用无效”。

在第一个参数中使用不同的指针调用 *calculator* 函数会产生以下结果（显示在注释中）：

```
void OnStart()
{
    Print(calculator(plus, 1, 2)); // 3
    Print(calculator(minus, 1, 2)); // -1
    Print(calculator(calc, 1, 2)); // 0
}
```

注意，如果没有显式初始化，所有函数指针都用零值填充。这适用于给定类型的全局变量和局部变量。

用 *typedef* 定义的指针类型可以从函数中返回，例如：

```
Calc generator(ushort type)
{
    switch(type)
    {
        case '+': return plus;
        case '-': return minus;
    }
    return NULL;
}
```

此外，函数指针的类型经常用于回调函数（*callback*，参见 *FuncCallback.mq5*）。假设我们有一个执行冗长计算的 *DoMath* 函数（很可能是在一个单独的库中实现的）。就用户界面的便利性和

友好性而言，向用户显示一个进度指示是很棒的。为此，您可以定义一个特殊类型的函数指针，用于通知已完成工作的百分比 (*ProgressCallback*)，并将此类型的参数添加到 *DoMath* 函数中。在 *DoMath* 代码中，应该定期调用传递的函数：

```
typedef void (*ProgressCallback)(const float percent);

void DoMath(double &bigdata[], ProgressCallback callback)
{
    const int N = 1000000;
    for(int i = 0; i < N; ++i)
    {
        if(i % 10000 == 0 && callback != NULL)
        {
            callback(i * 100.0f / N);
        }

        // long calculations
    }
}
```

然后，发起调用的代码可以定义所需的回调函数，将指向它的指针传递给 *DoMath*，并随着计算的进行接收更新。

```
void MyCallback(const float percent)
{
    Print(percent);
}

void OnStart()
{
    double data[] = {0};
    DoMath(data, MyCallback);
}
```

函数指针仅适用于 MQL5 中定义的自定义函数。它们不能指向 MQL5 API 的 [内置函数](#)。

2.8.11 内联

为了提高代码效率，现代编译器经常使用以下技巧。在生成可执行代码时，有些函数调用直接被函数体（它的语句）代替。这种技巧称为内联。由于避免了与组织调用和从函数返回相关的开销，因此操作速度提高。从程序员的角度来看，内联并没有改变什么。

MQL5 默认支持内联。必要时可以禁用，但仅限于 [代码剖析](#) 模式。为了与 C++ 源代码兼容，MQL5 中保留了 *inline* 关键字。函数定义之前是否存在这个关键字并不影响生成的程序。

2.9 预处理器

到目前为止，我们一直在研究 MQL5 编程，假设源代码由编译器处理，编译器将其文本表示转换为二进制（可以由终端执行）。然而第一个工具是预处理器，它负责读取源代码并在必要时转换。MetaEditor 中内置的这个实用程序由直接插入源代码的特殊指令控制。它可以解决程序员在编写源代码时面临的许多问题。

与 C++ 预处理器类似, MQL5 支持宏替换的定义 (**#define**)、条件编译 (**#ifdef**) 和包含其他源文件 (**#include**)。在本章中, 我们将探索这些能力。与 C++ 相比, 其中一些预处理器存在一定的局限性。

除了标准指令之外, MQL5 预处理器还有自己特定的指令, 特别是一组 MQL 程序特性 (**#property**), 以及从独立 EX5 和 DLL 导入函数的指令 (**#import**)。我们将在第五、六和七章研究各种类型的 MQL 程序时讨论这些问题。

所有预处理器指令都以散列符号 **#** 开头, 后跟一个关键字和附加参数, 具体语法取决于指令的类型。

建议将预处理器指令作为一行的开头, 或者至少在空格缩进之后开始 (如果是嵌套指令)。在源代码语句中插入指令被认为是一种不良编程风格 (与 MQL5 不同, C++ 预处理器根本不允许这样做)。

预处理器指令不是语言语句, 不应以 **;** 结束。指令通常延续到当前代码行的结尾。在某些情况下, 它们可以以特殊的方式扩展到后面的代码行 (这方面的内容将单独讨论)。

指令按照它们在文本中出现的顺序执行, 并考虑前面指令的处理情况。例如, 如果使用 **#include** 指令将两个文件连接, 并且在被包含的文件中使用 **#define** 定义了一条替换规则, 那么该规则将对后续所有代码行生效, 包括后续包含的头文件中的代码。

预处理器不会处理注释。

2.9.1 包含源文件 (**#include**)

#include 指令用于将另一个文件的内容包含到源代码中。该指令执行的操作就相当于程序员将文本从包含文件复制到剪贴板并粘贴到当前文件中该指令位置。

编写复杂程序时, 将源代码分成多个文件是一种常见的做法。这类程序建立在模块化的基础上, 这样每个模块/文件包含解决一个或多个相关任务的逻辑相关代码。

包含文件也用于分发库 (现成的算法集)。相同的库可以包含在不同的程序中。在这种情况下, 库更新 (其头文件的更新) 将在下次编译时自动应用于所有程序。

如果 MQL 程序主文件必须具有 **m q5** 扩展名, 那么包含文件的扩展名一般是 **mqh** (单词末尾的 **h** 代表 header)。同时, 允许对其他类型的文本文件使用 **#include** 指令, 例如 ***.txt** (见下文)。在任何情况下, 当包含一个文件时, 由 **m q5** 主文件和所有头文件组合而成的最终程序仍然必须是语法正确的。例如, 将一个具有二进制信息的文件 (如 **png** 图像) 包含在内会破坏编译。

有两种类型的 **#include** 语句:

```
#include <file_name>
#include "file_name"
```

在第一类语句中, 文件名用尖括号括起来。编译器在终端数据目录的 **MQL5/Include/** 子文件夹中搜索此类文件。

在第二类语句中, 文件名用引号括起来, 则在包含使用 **#include** 语句的当前文件的同一目录中执行搜索。

无论是那种情况, 文件都可以位于搜索目录的子文件夹中。此时, 您应该在指令中的文件名之前指定文件夹的完整相对层次结构。例如, 除了 **MetaTrader 5** 之外, 还有许多常用的引导文件, 包括 **DateTime.mqh**, 它提供了一组处理日期和时间的方法 (它们被设计为结构体, 这是我们将

在第三章 OOP 专题中讨论的语言构造)。 *DateTime.mqh* 文件位于 *Tools* 文件夹中。要将它包含在源代码中，应该使用以下指令：

```
#include <Tools/DateTime.mqh>
```

为了演示如何使用指令包含来自与源文件相同文件夹的头文件，让我们来分析 *Preprocessor.mqh* 文件。它包含以下指令：

```
#include "Preprocessor.mqh"
```

它引用 *Preprocessor.mqh* 文件，该文件实际上位于 *Preprocessor.mqh* 的同级目录中。

反过来，包含文件也可以包含其他文件。具体来说，*Preprocessor.mqh* 内部有以下代码：

```
double array[] =
{
    #include "Preprocessor.txt"
};
```

这意味着数组的内容是从给定的文本文件初始化的。如果我们查看 *Preprocessor.txt*，就会看到符合数组初始化语法规则的文本：

```
1, 2, 3, 4, 5
```

因此，可以从定制组件中收集源代码，包括使用其他程序生成源代码。

请注意，如果未找到指令中指定的文件，编译将会失败。

如果有多个文件，则包含顺序决定了预处理器指令的处理顺序。

2.9.2 宏替换指令概述

宏替换指令包括两种形式的 *#define* 指令：

- 通常用于定义一个常量
- 通过参数将宏定义为伪函数

此外，还有一个 *#undef* 指令用于撤消任何之前的 *#define* 定义。如果未使用 *#undef*，每个定义的宏在源代码编译结束之前都是有效的。

宏注册后即可在代码中按名称使用，并遵循标识符的规则。按照惯例，宏名称用大写字母书写。宏名称可以与变量、函数和其他源代码元素的名称重名。有目的地利用这一点可以动态地更改和生成源代码。然而，宏名称与程序元素的无意重名将导致错误。

两种形式的宏替换的操作原理是相同的。使用 *#define* 指令时，会引入一个标识符，该标识符与某段文本（即定义）相关联。如果预处理器后来在源代码中找到了给定的标识符，就会将该标识符替换为相关联的文本。我们强调，宏名称只有在注册后才能在编译的代码中使用（这类似于变量声明原则，但只适用于编译阶段）。

用宏的定义替换宏名称就称为扩展。对源代码的分析是逐步进行的，每次分析一行，但是每一行的扩展可以执行任意次，就像在循环中一样，前提是在结果中找到宏名称。不能在宏定义中包含相同的名称：替换时，这样的宏将导致“未知标识符”错误。

在本书的第三章，我们将学习 **模板**，它允许您使用不同规则生成（事实上就是复制）源代码。如果源代码中既有宏替换指令又有模板，则首先扩展宏，然后从模板生成代码。

宏名称在 MetaEditor 中以红色高亮显示。

2.9.3 简单形式的 #define

简单形式的 #define 指令将注册一个标识符和一个字符序列 - 在源代码中该指令之后，直到程序的结尾，或者在具有相同标识符的 #undef 指令之前，都会用字符序列替换标识符。

语法如下：

```
#define macro_identifier [text]
```

文本从标识符之后开始，一直到当前行的结尾。标识符和文本必须由任意数量的空格或制表符来分隔。如果所需的字符序列太长，出于可读性考虑，可以在行结尾添加反斜杠字符 \ 将它分成几行。

```
#define macro_identifier text_beginning \
                                text_continued \
                                text_ending
```

文本可以由任何语言结构组成：常量、运算符、标识符和标点符号。如果用 *macro_identifier* 代替源代码中找到的结构，所有这些结构都将包含在编译中。

简单形式通常用于几个目的：

1. 标志声明，这随后又用于 条件编译 检查；
2. 命名常量声明；
3. 常用语句的简化表示。

第一个的特点是，在标识符之后不需要指定任何内容 - 只要存在带名称的指令就能注册的相应标识符，并且可以用于条件指令 *#ifdef/#ifndef*。对这些条件指令来说，最重要的是该标识符是否存在，也就是以标志模式运行：声明/未声明。例如，以下指令定义了 DEMO 标志：

```
#define DEMO
```

比如说，它可以用来构建一个排除了某些函数的程序的演示版本（参见条件编译一节中的例子）。

使用简单指令的第二种方法允许您用友好的名称替换源代码中的“幻数”。“幻数”是插入到源文本中的常量，其含义往往不明确（因为数字只是数字：最好至少在注释中解释）。此外，相同的值可能分散在代码的不同部分，如果程序员决定将它改为另一个值，就必须在所有地方都进行更改（希望他没有漏掉任何一处）。

有了命名宏，这两个问题就很容易解决了。例如，一个脚本可以生成一个包含斐波那契数的数组，直到达到某个最大深度。然后用预定义的数组大小定义一个宏并在数组本身的说明中使用这个宏（*Preprocessor.mq5*）。


```

#define MAX_FIBO 10

int fibo[MAX_FIBO]; // 10

void FillFibo()
{
    int prev = 0;
    int result = 1;

    for(int i = 0; i < MAX_FIBO; ++i) // i < 10
    {
        int temp = result;
        result = result + prev;
        fibo[i] = result;
        prev = temp;
    }
}

```

如果程序员随后决定需要将数组增大多少，只需在一个位置执行即可，也就是在 **#define** 指令中。因此，该指令实际上定义了算法的某个参数，该参数被“硬连线”到源代码中，并且不可用于用户配置。这种需求经常出现。

人们可能会疑惑，通过 **#define** 进行定义与全局上下文中的常量变量到底有何不同。事实上，我们可以声明一个具有相同名称和用途的变量，甚至保留大写字母：

```
const int MAX_FIBO = 10;
```

然而，在这种情况下，MQL5 不允许定义具有指定大小的数组，因为方括号中只允许放入常量，即字面量（尽管“常量变量”这个名字看起来像，但并不是常量）。为解决这个问题，我们可以将数组定义为动态的（不需要先指定大小），然后使用 **ArrayResize** 函数 - 将变量作为大小传递在这里并不困难。

定义命名常量的另一种方法是使用枚举，但仅限于整数值。例如：

```

enum
{
    MAX_FIBO = 10
};

```

但是宏可以包含任何类型的值。

```

#define TIME_LIMIT      D'2023.01.01'
#define MIN_GRID_STEP  0.005

```

在源文本中搜索宏名称进行替换时，会考虑语言的语法，也就是说，不可分元素（如变量标识符或字符串字面量）将保持不变，即使它们包含与某个宏匹配的子字符串。例如，给定以下宏 XYZ，XYZAXES 变量将保持不变，XYZ 名称（因为它与宏完全相同）将更改为 ABC。

```

#define XYZ ABC
int XYZAXES = 3; // int XYZAXES = 3
int XYZ = 0;     // int ABC = 0

```

宏替换可用于将代码嵌入到其他程序的源代码中。以 .mqh 头文件形式分发的库会采用这种技巧，这些库通过 **#include** 指令连接到程序。

特别是对于脚本，我们可以定义自己的 *OnStart* 函数库实现，它必须在不影响程序原有功能的情况下执行一些额外的操作。

```
void OnStart()
{
    Print("OnStart wrapper started");
    // ... additional actions
    _OnStart();
    // ... additional actions
    Print("OnStart wrapper stopped");
}

#define OnStart _OnStart
```

假设这部分在包含的头文件中 (*Preprocessor.mqh*)。

随后，预处理器会在源代码中将原始函数 *OnStart* (在 *Preprocessor.mqh5* 中) 重命名为 *_OnStart* (前提是该标识符不会在其他任何地方用于其他目的)。而头文件中新版本的 *OnStart* 调用 *_OnStart*，将其“包装”到附加语句中。

使用简单 *#define* 指令的第三种常见方法是缩短语言结构的表示法。例如，无限循环的标题可以用一个单词 *LOOP* 来表示：

```
#define LOOP for( ; !IsStopped() ; )
```

然后应用到代码中：

```
LOOP
{
    // ...
    Sleep(1000);
}
```

这个方法也是使用 *#define* 指令 (带参数) 的主要技巧 (参见下文)。

2.9.4 #define 作为伪函数的形式

参数形式 *#define* 的语法类似于函数语法。

```
#define macro_identifier(parameter,...) text_with_parameters
```

这样的宏具有一个或多个带圆括号的参数。各参数用逗号分隔。每个参数都是一个简单的标识符 (通常是一个字母)。此外，一个宏的所有参数必须有不同的标识符。

标识符和左圆括号之间不能有空格，这一点很重要，否则宏将被视为一个简单的形式，并且替换文本以左圆括号开始。

该指令注册后，预处理器将在源代码中搜索以下格式的行：

```
macro_identifier(expression,...)
```

可以指定任意表达式来代替参数。自变量的数量必须与宏参数的数量相匹配。所有找到的匹配项将被替换为 *text_with_parameters*，其中参数将被替换为传递的表达式。每个参数可以以任何顺序出现多次。

例如，以下宏查找两个值中的较大值：

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

如果代码包含语句：

```
int z = MAX(x, y);
```

它将被预处理器“扩展”成：

```
int z = ((x) > (y) ? (x) : (y));
```

宏替换适用于任何数据类型（在宏内部应用的操作对这些数据类型有效）。

不过这种替换也会有副作用。例如，如果实际参数是修改变量（比如++x）的函数调用或语句，那么相应的操作可以执行多次（而不是预期的一次）。如果为 MAX，这会发生两次，一次是在比较期间，另一次是在 ? : 运算符的一个分支内获取值时。这样看来，尽可能将这样的宏转换成函数是合理的（特别是考虑到 MQL5 中的函数会自动内联）。

各参数和整个宏定义都带有圆括号。使用圆括号可以确保，将表达式替换为参数或其他表达式内部的宏本身这一操作不会因不同优先级而改变计算顺序。假设宏定义了两个参数（未带圆括号）的乘积：

```
#define MUL(A,B) A * B
```

那么使用带有以下表达式的宏将会产生意外的结果：

```
int x = MUL(1 + 2, 3 + 4); // 1 + 2 * 3 + 4
```

不是进行乘法 $(1 + 2) * (3 + 4)$ 得到 21，而是计算 $1 + 2 * 3 + 4$ ，得到 11。正确的宏定义应该是这样的：

```
#define MUL(A,B) ((A) * (B))
```

可以指定另一个宏作为宏参数。此外，还可以在宏定义中插入其他宏。所有这样的宏都将被顺序替换。例如：

```
#define SQ3(X) (X * X * X)
#define ABS(X) MathAbs(SQ3(X))
#define INC(Y) (++(Y))
```

下面的代码将打印 504 *MathAbs* 是一个内置函数，返回一个数字的模数，即无符号数）：

```
int x = -10;
Print(ABS(INC(x)));
// -> ABS(++(Y))
// -> MathAbs(SQ3(++(Y)))
// -> MathAbs((++(Y))*++(Y))*++(Y)))
// -> MathAbs(-9*-8*-7)
// -> 504
```

在 x 变量中，值 -7 将保持不变（由于三次递增）。

宏定义可以包含不配对的圆括号。这种技巧正确巧通常用在一对宏中，其中一个宏打开某段代码，另一个宏关闭该段代码。在这种情况下，每个宏中的不配对圆括号将会变得匹配。特别是，在 MetaTrader 5 分发库的标准库文件中，*Controls/Defines.mqh* 中定义了 EVENT_MAP_BEGIN 和 EVENT_MAP_END。它们用于在图形对象中构建事件处理函数。

预处理器逐行读取程序的整个源文本，从 mq5 主文件开始，并在遇到头文件的位置插入文本。任何一行代码被读取时，一组已经定义好的宏就形成了。宏的定义顺序并不重要：很可能一个

宏在其定义中引用了另一个宏，而被引用的宏可能在该宏的上方或下方。最重要的是，在使用宏名称的源代码行中，所有引用宏的定义都是已知的。

我们来分析一个例子。

```
#define NEG(x) (-SQN(x))*TEN
#define SQN(x) ((x)*(x))
#define TEN 10
...
Print(NEG(2)); // -40
```

这里，NEG 宏使用了在其下方描述的 SQN 和 TEN 宏。这并不妨碍我们在所有三个 `#define` 之后成功地在代码中使用 NEG 宏。

但是，如果我们将所有列的相对位置更改为以下：

```
#define NEG(x) (-SQN(x))*TEN
#define SQN(x) ((x)*(x))
...
Print(NEG(2)); // error: 'TEN' - undeclared identifier
...
#define TEN 10
```

我们会遇到一个“未声明的标识符”编译错误。

2.9.5 #define 定义中的特殊运算符 # 和

在宏定义中，可以使用两个特殊的运算符：

- ① 宏参数名称前的单个哈希符号 # 会将该参数的内容转换为字符串；只允许在函数宏中使用该符号；
- ② 两个单词（标记）之间的双哈希符号 ## 会将这两个单词组合在一起，如果标记是宏参数，则替换其值，但如果标记是宏名称，则按原样替换，不展开宏；如果“粘合”后产生了另一个宏名称，则宏被扩展；

在本书的例子中，我们经常使用以下宏：

```
#define PRT(A) Print(#A, "=", (A))
```

它调用 `Print` 函数，在这个函数中，传递的表达式通过 #A 显示为一个字符串，在等号符号之后打印出 A 的实际值。

为了演示 ##，我们来分析另一个宏：

```
#define COMBINE(A,B,X) A##B(X)
```

使用这个宏，我们实际上可以生成对上文定义的 SQN 宏的调用：

```
Print(COMBINE(SQ,N,2)); // 4
```

SQ 和 N 字面量连接在一起，之后 SQN 宏扩展为 `((2)*(2))` 并生成结果 4。

以下宏允许您在代码中创建变量定义，方法是在给定宏参数的情况下生成变量名称：

```
#define VAR(TYPE,N) TYPE var##N = N
```

代码行：

```
VAR(int, 3);
```

相当于以下代码行：

```
int var3 = 3;
```

通过连接标记，可以使用宏在数组元素上实现循环速记。

```
#define for_each(I, A) for(int I = 0, max_##I = ArraySize(A); I < max_##I; ++I)

// describe and somehow fill in the array x
double x[];
// ...
// implement loop through the array
for_each(i, x)
{
    x[i] = i * i;
}
```

2.9.6 取消宏替换 (#undef)

使用 **#define** 注册的替换，如果在一段特定代码后不再需要，可以撤销。为此，将使用 **#undef** 指令。

```
#undef macro_identifier
```

如果您需要在代码的不同部分以不同的方式定义同一个宏，这是非常有用的。如果 **#define** 中指定的标识符已经在前面的代码行中注册过（由另一个 **#define** 指令注册），那么旧的定义将被新的定义替换，预处理器将生成“宏重定义”警告。使用 **#undef** 可以避免警告，同时明确表明程序员不打算在后面的代码中使用特定的宏。

#undef 无法取消定义 预定义宏。

2.9.7 预定义的预处理器常量

SQL5 有多个预定义的常量，相当于简单的宏，但它们是由编译器自己定义的。下表列出了其中一部分的名称和含义。

名称	说明
__COUNTER__	计数器（在宏展开过程中，文本中每引用一次都会将该宏的值加 1）
__DATE__	编译日期（天）
__DATETIME__	编译日期和时间
__FILE__	已编译文件的名称
__FUNCSIG__	当前函数签名
__FUNCTION__	当前函数名称

名称	说明
<code>__LINE__</code>	编译文件中的行号
<code>__MQLBUILD__</code> , <code>__MQL5BUILD__</code>	编译器版本
<code>__RANDOM__</code>	<code>ulong</code> 类型的随机数
<code>__PATH__</code>	编译文件的路径
<code>_DEBUG</code>	在调试模式下编译时定义
<code>_RELEASE</code>	在正常模式下编译时定义

2.9.8 条件编译 (#ifdef/#ifndef/#else/#endif)

条件编译指令允许你在编译过程中包含和排除代码片段。`#ifdef` 和 `#ifndef` 指令标记由它们控制的代码片段的开始。该代码片段以 `#endif` 指令结束。`#ifdef` 语法的最简单形式如下：

```
#ifdef macro_identifier
    statements
#endif
```

如果使用 `#define` 在代码上方定义了具有指定标识符的宏，那么这个代码片段将参与编译。否则会将其排除。除了应用程序代码中定义的宏之外，环境中还提供了一组预定义的常量，特别是 `_RELEASE` 和 `_DEBUG` 标志 (参见 [预定义常量](#) 一节)；也可以在条件编译指令中检查它们的名称。

扩展形式 `#ifdef` 允许指定两段代码：如果定义了宏标识符，将包含第一段代码，如果没有定义，将包含第二段代码。为此，在 `#ifdef` 和 `#endif` 之间插入一个片段分隔符 `#else`。

```
#ifdef macro_identifier
    statements_true
#else
    statements_false
#endif
```

`#ifndef` 指令的工作方式相似，但是片段的包含和排除是按照相反的逻辑进行的：如果头文件中指定的宏没有定义，则编译第一个片段，如果定义了，则编译第二个片段。

例如，根据是否存在 `DEMO` 宏替换，我们可能会也可能不会调用函数以计算斐波那契数。

```
#ifdef DEMO
    Print("Fibo is disabled in the demo");
#else
    FillFibo();
#endif
```

在这种情况下，如果启用了 `DEMO` 模式，将在日志中显示一条消息，而不是调用函数，但是由于在 `Preprocessor.mq5` 脚本和所有包含文件中都没有 `#define DEMO` 定义，所以编译将根据分支 `#else` 进行，即对 `FillFibo` 函数的调用将进入可执行文件 `ex5`。

指令可以嵌套。

```
#ifdef _DEBUG
    Print("Debugging");
#else
    #ifdef _RELEASE
        Print("Normal run");
    #else
        Print("Undefined mode!");
    #endif
#endif
#endif
```

2.9.9 程序的通用特性 (#property)

使用 *#property* 指令，程序员可以设置 MQL 程序的一些特性。其中一些特性是通用的，即适用于任何程序，我们将在这里分析这些特性。其余特性是 MQL5 程序特定类型的典型特性，将在第五章中介绍 MQL5 API 的相关节中讨论。

#property 指令的格式如下：

```
#property key value
```

最主要的是下表第一列中列出的一种特性。第二列说明如何解释该值。

特 性	值
copyright	包含版权所有者信息的字符串
link	带有开发人员网站链接的字符串
version	带有程序版本号的字符串 (对于 MQL5 市场，必须是 X.Y 格式，其中 X 和 Y 是对应于主要和次要内部版本号的整数)
description	包含程序说明的行 (允许多个 <i>#description</i> 指令，并且它们的内容是组合的)
icon	字符串，程序徽标 (ICO 格式) 文件的路径
stacksize	以字节为单位指定栈大小的整数 (默认从 4 MB 到 16 MB，取决于程序类型和环境，1 MB = 1024*1024 字节)；如有必要，大小可增至 64 MB (最大值)。

上述所有字符串特性都是程序特性对话框的信息来源，该对话框在程序启动时打开。但对于脚本，默认不显示此对话框。若要更改此行为，必须另外指定 *#property script_show_inputs* 指令。此外，当鼠标光标悬停在 MetaTrader 5 *Navigator* 中的程序上时，有关权限的信息会显示在工具提示中。

copyright、*link* 和 *version* 特性已经本书前面的例子中介绍过。

栈大小 *stacksize* 只是一个建议：如果编译器发现源代码中的局部变量 (通常是数组) 超过了指定的值，栈将在编译期间自动增大，但最多不超过 64 MB。如果超过此限制，程序甚至无法启动：在日志 (Log 标签，而不是 *Experts*) 中，出现错误“栈大小超过 64MB。请减少局部变量占用的内存”。

请注意，此类分析和阻止启动行为仅考虑启动时程序的固定快照。在递归函数调用的情况下，栈内存消耗会显著增加，并导致栈溢出错误，但这已经在程序执行阶段发生了。有关栈的更多信息，参见 [描述数组](#) 中的说明。

#property 指令只在编译后的 `m q5` 文件中有效，在所有用 ***#include*** 包含的文件中，该指令会被忽略。

第三章MQL5 中面向对象的编程

在软件开发过程中，当发现内置类型和函数集无法有效满足需求时，这个问题就会凸显出来。构成程序的众多微小实体的管理复杂度会像滚雪球般迅速膨胀，因此需要采用某种技术来提升程序员工作的便利性、生产力和质量。

其中一种技术称为面向对象，已在多种编程语言层面实现，而基于它的编程风格则相应地称为面向对象的编程（OOP）。MQL5 编程语言同样支持 OOP，因此与 C++ 一样，属于面向对象的语言。

顾名思义，面向对象的技术的核心在于“对象”。本质上，对象是用户定义类型的变量，即程序员使用 MQL5 工具定义的类型。通过创建能够模拟主题领域的类型，可以提高程序的易读性，并简化其编写和维护过程。

MQL5 提供了多种定义新类型的方式，每种方式都有其特定的功能和特点，我们将在后续相关章节中介绍。根据说明方式的不同，用户定义类型分为类、结构体和关联。其中每种类型都可以组合数据和算法，即描述所应用对象的状态和行为。

在本书的第一章，我们曾引用编程先驱 Nicklaus Wirth 的名言：程序是算法和数据结构的共生体。因此，对象本质上是微型程序，每个对象负责处理其自己的任务，这种任务虽然规模不大，但逻辑完整。通过将不同的对象组合成单个系统，您可以构建出复杂度各异的服务或产品。由此可见，OOP 为我们提供了“分而治之”策略的一种全新诠释。

应将 OOP 视为我们在第二章探讨的过程化编程风格的一种更强大且更灵活的替代方案。同时，这两种编程范式并非相互对立：在需要的情况下也可以结合使用。对于很简单的任务，可以不必采用 OOP。

因此，本书的第三部分将重点介绍 OOP 的基础知识以及在 MQL5 中实际应用的可能性。此外，我们还将讨论模板、接口和命名空间。

 面向交易员的 MQL5 编程指南 - 本书中的源代码第三章

 本书中的例子也可以在 [公共项目](#) 中的 `\MQL5\Shared Projects\MQL5Book` 中找到

3.1 结构体和联合体

结构体是最容易理解的对象类型，因此我们将从结构体开始介绍 OOP。结构体与类有很多共同之处，而类是 OOP 的主要构建块，因此理解结构体将为日后学习类打下基础。同时，结构体也有某些不同之处，其中一些可能是限制，而另一些则可能是优势。特别是，结构体不能有 [虚函数](#)，但可用于与第三方 DLL 集成。

在实现算法时，选择使用结构体还是类，通常取决于对对象元素的访问需求以及是否存在内部业务逻辑。如果只需要一个用来存放结构化数据的简单容器，并且不需要检查容器内部状态的正确性（在编程中称为“不变式”），那么使用结构体就完全可以满足需求。如果您想限制访问并支持根据某些规则进行写入和读取（这些规则以分配给对象的函数的形式来规定，我们将在后面讨论），那么最好使用类。

MQL5 内置了各种类型的结构体，用于描述交易中需要的实体，包括报价数据（`MqlRates`）、即时报价（`MqlTick`）、日期和时间（`MqlDateTime`）、交易请求（`MqlTradeRequest`）、请求的结果（`MqlTradeResult`）等。我们将在本书第六章中讨论它们。

3.1.1 结构体定义

结构体由变量组成，变量可以是内置类型，也可以是由用户定义的其他类型。结构体用于将逻辑上相关的数据组合在一个容器中。假设我们有一个执行特定计算的函数，并接受一组参数：显示报价历史以供分析的条数、分析开始日期、价格类型和分配的信号数（例如谐波）。

```
double calculate(datetime start, int barNumber,
                 ENUM_APPLIED_PRICE price, int components);
```

在现实情况中，可能会有更多参数，将它们以列表形式传递给函数并非易事。此外，根据多次计算的结果，将一些最佳设置保存在某种数组中也很有意义。因此，将一组参数表示为一个对象会很方便。

以下是对带有相同变量的结构体的说明：

```
struct Settings
{
    datetime start;
    int barNumber;
    ENUM_APPLIED_PRICE price;
    int components;
};
```

说明以 **struct** 关键字开头，后面是我们选择的标识符。随后是括在花括号中的代码块，代码块内是对结构体中所含变量的说明。此外，这些变量被称为字段或结构体的成员。花括号后有一个分号，因为整个符号是一个定义新类型的语句，而语句后必须有 ';'。

定义了类型后，我们就可以像应用内置类型一样应用自己定义的类型。具体而言，新类型可以让您以常规方式描述程序中的 **Settings** 变量。

```
Settings s;
```

值得注意的是，通过单个结构体描述，您可以创建任意数量的结构体变量，甚至是该类型的数组。每个结构体实例都有自己的一组元素，并且包含独立的值。

要访问结构体的成员，需要提供一个特殊的取消引用运算符，即点字符 '.'。它的左边应该是一个结构体类型的变量，右边则是其中一个可用字段的标识符。下面是为结构体元素赋值的方法：

```
void OnStart()
{
    Settings s;
    s.start = D'2021.01.01';
    s.barNumber = 1000;
    s.price = PRICE_CLOSE;
    s.components = 8;
}
```

还可以通过一种更便捷的方法来填充结构体，即聚合初始化。在本例中，'=' 符号写在结构体变量的右边，后面是所有字段的初始值列表，以英文逗号分隔并都括在花括号中。

```
Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
```

值的类型必须与相应的元素类型相匹配。允许指定的值数量少于字段数量，在这种情况下，剩余字段的值为零。

请注意，只有在变量被初始化时，即在定义变量时，这种方法才会起作用。无法用这种方法赋值一个已经存在的结构体的内容，否则会产生编译错误。

```
Settings s;
// error: '{' - parameter conversion not allowed
s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
```

使用取消引用运算符，还可以读取结构体元素的值。例如，我们使用条数来计算交易品种数量。

```
s.components = (int)(MathSqrt(s.barNumber) + 1);
```

这里的 *MathSqrt* 是内置的 [平方根](#) 函数。

我们引入了一种新类型 *Settings*，以便更轻松地将一组参数传递给函数。现在，它可以用作更新后的 *calculate* 函数的唯一参数：

```
double calculate(Settings &settings);
```

请注意参数名称前面的 `&`，这表示 [按引用传递](#)。结构体在作为参数传递时，只能按引用传递。

如果需要从函数返回一组值而不是单个值，也可以使用结构体。假设 *calculate* 函数不应返回 *double* 类型的值，而是几个系数和一些交易建议（交易方向和成功概率）。然后，我们就可以定义 *Result* 结构体的类型，并在函数原型 (*Structs.mq5*) 中使用。

```
struct Result
{
    double probability;
    double coef[3];
    int direction;
    string status;
};

Result calculate(Settings &settings)
{
    if(settings.barNumber > 1000) // edit fields
    {
        settings.components = (int)(MathSqrt(settings.barNumber) + 1);
    }
    // ...
    // emulate getting the result
    Result r = {};
    r.direction = +1;
    for(int i = 0; i < 3; i++) r.coef[i] = i + 1;
    return r;
}
```

Result r = {} 行中的空花括号代表最小聚合初始化器：它将结构体的所有字段都填为 0。

如有必要，可以分开进行结构体类型的定义和声明（通常，声明包含在头文件 `mqh` 中，而定义则包含在 `mq5` 文件中）。这种扩展语法将在 [关于“类”的章节](#)中介绍。

3.1.2 结构体中的函数（方法）

从 *calculate* 函数中得到结果后，最好能将其打印到日志中，但 *Print* 函数并不适用于用户定义的类型：用户定义的类型必须自己提供输出信息的方法。

```
void OnStart()
{
    Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
    Result r = calculate(s);
    // Print(r); // error: 'r' - objects are passed by reference only
    // Print(&r); // error: 'r' - class type expected
}
```

注释显示了调用结构体的 *Print* 函数的尝试，以及随后发生的情况。第一个错误是因为结构体实例是对象，而对象必须按引用传递给函数。同时，*Print* 期望得到值（一个或多个）。在第二个 *Print* 调用中，变量名前使用了 & 符号，这意味着在 MQL5 中接收到指针，而不是人们所认为的引用。MQL5 仅支持类对象（而不支持结构体）的指针，因此出现了第二个错误“应为类类型”。我们将在下一章深入学习有关指针的知识（参见 [类和接口](#)）。

我们可以在 *Print* 调用中分别指定结构体的所有成员（使用取消引用），但这相当麻烦。

如果需要以特殊方式处理结构体的内容，可以在结构体内部定义函数。定义的语法与我们熟悉的全局上下文函数并无不同，但定义本身位于结构块内部。

这些函数称为方法。由于这些函数位于相应代码块的上下文中，因此无需取消引用运算符即可从方法中访问结构体的字段。例如，我们在 *Result* 结构体中编写 *print* 函数的实现。

```
struct Result
{
    ...
    void print()
    {
        Print(probability, " ", direction, " ", status);
        ArrayPrint(coef);
    }
};
```

调用结构体实例的方法很简单，直接读取其字段即可：使用同样的 '.' 运算符。

```
void OnStart()
{
    Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
    Result r = calculate(s);
    r.print();
}
```

关于“类”的章节将对方法进行更详细的介绍。

3.1.3 复制结构体

可以使用 '=' 赋值运算符将同一类型的结构体完全拷贝到对方结构体中。我们以 *Result* 结构体为例来演示这一规则。我们从 *calculate* 函数得到 *r* 的第一个实例。

```

void OnStart()
{
    ...
    Result r = calculate(s);
    r.print();
    // will output to the log:
    // 0.5 1 ok
    // 1.00000 2.00000 3.00000
    ...
    Result r2;
    r2 = r;
    r2.print();
    // will output to the log the same values:
    // 0.5 1 ok
    // 1.00000 2.00000 3.00000
}

```

然后，我们额外创建了变量 **Result r2**，并将 **r** 变量的内容、所有字段同时复制到该变量中。可以使用 **print** 方法输出到日志来验证运算的准确性（注释中给出了行数）。

需要注意的是，用同一组字段定义两种类型的结构体并不能使这两种类型的结构体相同。无法将一个结构体完全赋值给另一个结构体，在这种情况下只允许逐成员赋值。

稍后我们将讨论结构体继承，这将提供更多拷贝选项。事实上，拷贝不仅适用于同一类型的结构体，也适用于相关类型的结构体。不过，这其中也有一些重要的细微差别，我们将在 [结构体的布局和继承](#) 一节中进行介绍。

3.1.4 构造函数和析构函数

在可以为结构体定义的方法中，有一些特殊的方法：构造函数和析构函数。

构造函数的名称与结构体名称相同，但不返回值（类型为 **void**）。如果定义了构造函数，将在初始化结构体的每个新实例时调用。因此，在构造函数中，可以通过特殊方式计算结构体的初始状态。

一个结构体可以有多个带有不同参数集的构造函数，编译器会在定义变量时根据自变量的数量和类型选择合适的构造函数。

例如，我们可以在 **Result** 结构体中描述一对构造函数：一个不带参数，另一个带一个字符串类型参数，用于设置状态。

```

struct Result
{
    ...
    void Result()
    {
        status = "ok";
    }
    void Result(string s)
    {
        status = s;
    }
};

```

顺便说一下，不带参数的构造函数称为默认构造函数。如果没有显式构造函数，编译器会为任何包含字符串和动态数组的结构体隐式创建一个默认构造函数，目的是在这些字段中填充 0。

重要的是，无论结构体是否有默认构造函数，其他类型的字段（例如所有数值字段）都不会被重置为零，因此内存分配后元素的初始值将是随机的。您应该创建构造函数，或者确保在创建对象后立即在代码中正确赋值。

如果存在显式构造函数，就无法使用聚合初始化语法。因此，*calculate* 方法中的 `Result r = {};` 行将不会进行编译。现在，我们只有权使用我们自己提供的其中一个构造函数。例如，以下语句调用了无参数构造函数：

```

Result r1;
Result r2();

```

可以按如下所示创建一个已填充状态的结构体：

```

Result r3("success");

```

创建结构体数组时，也会调用默认构造函数（显式或隐式）。例如，下面的语句为 10 个有结果的结构体分配内存，并使用默认构造函数对它们进行初始化：

```

Result array[10];

```

析构函数是在销毁结构体对象时调用的函数。析构函数的名称与结构体名称相同，但前缀是一个波浪号 '~'。与构造函数一样，析构函数不返回值，但也不接受参数。

析构函数只能有一个。

您不能显式地调用析构函数。程序本身会在退出定义了局部结构体变量的代码块或释放结构体数组时调用析构函数。

如果结构体在构造函数中分配了动态资源，那么析构函数的作用就是释放这些资源。例如，结构体可以具有 *persistence* 属性，即从内存中卸载结构体时将其状态保存到文件中，并在程序再次创建结构体时恢复状态。在这种情况下，需要打开和关闭的描述符会被用于内置 [文件函数](#) 中。

我们在 *Result* 结构体中定义一个析构函数，并逐步添加构造函数，以便所有这些方法都能跟踪对象实例的数量（随着它们的创建和销毁）。

```

struct Result
{
    ...
    void Result()
    {
        static int count = 0;
        Print(__FUNCSIG__, " ", ++count);
        status = "ok";
    }

    void Result(string s)
    {
        static int count = 0;
        Print(__FUNCSIG__, " ", ++count);
        status = s;
    }

    void ~Result()
    {
        static int count = 0;
        Print(__FUNCSIG__, " ", ++count);
    }
};

```

三个名为 *count* 的静态变量彼此独立存在：每个变量都在各自函数的上下文中计数。

运行脚本后，我们将收到以下日志：

```

Result::Result() 1
Result::Result() 2
Result::Result() 3
Result::~~Result() 1
Result::~~Result() 2
0.5 1 ok
1.00000 2.00000 3.00000
Result::Result(string) 1
0.5 1 ok
1.00000 2.00000 3.00000
Result::~~Result() 3
Result::~~Result() 4

```

我们来弄清楚它的含义。

结构体的第一个实例是在函数 *OnStart* 中创建的，也就是调用 *calculate* 的那一行。在进入构造函数时，计数器值 *count* 被初始化为 0 一次，然后每次执行构造函数时都会递增，因此第一次输出的值是 1。

在 *calculate* 函数内部，定义了一个 *Result* 类型的局部变量；该变量的寄存器编号为 2。

第三个结构体实例并不明显。关键在于，为了从函数中传递结果，编译器隐式创建了一个临时变量，并在其中拷贝了本地变量的数据。将来这种行为很可能会改变，届时本地实例将“移出”函数而不会重复。

最后一次构造函数调用是在一个带有字符串参数的方法中进行的，因此调用计数为 1。

重要的是，两个构造函数的总调用次数必须与析构函数的调用次数相同，也就是4次。

我们将在关于“类”的章节详细介绍 [构造函数](#) 和 [析构函数](#)。

3.1.5 在内存中打包结构体以及与 DLL 交互

要存储结构体的一个实例，需要在内存中分配一个足以容纳所有元素的连续区域。

与 C++ 不同的是，这里的结构体元素在内存中连续排列，不会根据元素本身的大小在 2、4、8 或 16 字节的边界上对齐（不同的编译器和运行模式有不同的对齐算法）。如果元素的大小小于指定的块，则通过在结构体组成中添加未使用的虚拟变量（程序无法直接访问这些变量）来对元素进行对齐。对齐用于优化内存性能。

SQL5 允许您在必要时更改对齐规则，主要是在将 SQL 程序与描述特定类型结构体的第三方 DLL 集成时。对于这些第三方 DLL，需要在 SQL5 中准备等效说明（参见有关 [导入库](#) 的章节）。需要注意的是，用于集成的结构体在其定义中只能包含有限类型的字段。因此，它们不能使用字符串、动态数组以及类对象和指向类对象的 [指针](#)。

对齐方式由添加到结构体头部的 **pack** 关键字控制。有两个选项：

```
struct pack(size) identifier
struct identifier pack(size)
```

在这两种情况下，大小都是整数 1、2、4、8、16。或者使用 **sizeof(built-in_type)** 运算符作为大小，例如 **sizeof(double)**。

选项 **pack(1)**，即字节对齐，与不带 **pack** 修饰符的默认行为相同。

通过特殊运算符 **offsetof()**，可以找出特定结构体元素从其起始位置开始的字节偏移量。它有两个参数：结构体对象和元素标识符。例如，

```
Print(offsetof(Result, status)); // 36
```

在 **Result** 结构体的 **status** 字段之前，有 4 个 **double** 值和 1 个 **int** 值：总共 36 个字节。

在设计自己的结构体时，建议先放置最大的元素，然后按大小递减的顺序放置其余元素。

3.1.6 结构体布局 and 继承

结构体可以将其他结构体作为其字段。例如，我们定义 **Inclosure** 结构体，并对 **Main** 结构体中的 **data** 字段使用这一类型 (**StructsComposition.mq5**):


```

struct Inclosure
{
    double X, Y;
};

struct Main
{
    Inclosure data;
    int code;
};

void OnStart()
{
    Main m = {{0.1, 0.2}, -1}; // aggregate initialization
    m.data.X = 1.0;           // assignment element by element
    m.data.Y = -1.0;
}

```

在初始化列表中，**data** 字段使用额外一层花括号表示，花括号内的字段值为 *Inclosure*。要访问这种结构体的字段，需要进行两次取消引用操作。

如果嵌套结构体未在其他地方使用，可以直接在外部结构体内声明它。

```

struct Main2
{
    struct Inclosure2
    {
        double X, Y;
    }
    data;
    int code;
};

```

另一种结构体布局方式是继承。这种机制通常用于构建类层次结构（将在相应 章节中详细讨论），但也可用于结构体。

在定义新的结构体类型时，程序员可以在结构体头部的冒号后注明父结构体的类型（必须在源代码的前面定义）。这样，父结构体的所有字段都将添加到子结构体中（在子结构体的开头），新结构体的字段将位于父结构体字段后面的内存中。

```

struct Main3 : Inclosure
{
    int code;
};

```

这里的父结构体不是嵌套结构，而是子结构体不可分割的一部分。因此，在初始化时填写字段不需要额外的花括号，也不需要一连串的多个取消引用操作符。

```

Main3 m3 = {0.1, 0.2, -1};
m3.X = 1.0;
m3.Y = -1.0;

```

所有三个结构体 *Main*、*Main2* 和 *Main3* 都具有相同的内存表示形式，大小为 20 字节。但它们的类型不同。

```
Print(sizeof(Main)); // 20
Print(sizeof(Main2)); // 20
Print(sizeof(Main3)); // 20
```

如前所述 (参见 [拷贝结构体](#))，赋值运算符 '=' 可用于复制相关类型的结构体，更确切地说，是拷贝由继承链链接的结构体。换言之，父类型的结构体可以写入子类型的结构体 (在这种情况下，派生结构中添加的字段将保持不变)，反之亦然，子类型的结构体可以写入父类型的结构体 (在这种情况下，“多余”的字段将被删除)。

例如：

```
Inclosure in = {10, 100};
m3 = in;
```

这里，*m3* 变量的类型为 *Main3*，继承自 *Inclosure*。执行赋值 *m3 = in* 操作后，字段 *X* 和 *Y* (两种类型共有的成员) 将从基类型变量 *in* 拷贝到派生类型变量 *m3* 的 *X* 和 *Y* 字段中。*m3* 变量的 *code* 字段将保持不变。

子结构体是祖代结构体的直接后代还是间接后代并不重要，继承链的深度不影响此行为。这种公共字段的拷贝机制适用于“子结构体”、“孙结构体”以及“结构体继承树”不同分支的各种类型组合。

如果父结构体只有带参数的构造函数，那么在继承派生结构体构造函数时，必须从初始化列表中调用父结构体。例如，

```
struct Base
{
    const int mode;
    string s;
    Base(const int m) : mode(m) { }
};

struct Derived : Base
{
    double data[10];
    // if we remove the constructor, we get an error:
    Derived() : Base(1) { } // 'Base' - wrong parameters count
};
```

我们在 *Base* 构造函数中填充了 *mode* 字段。由于此字段包含 *const* 修饰符，因此只能在构造函数中为它设置值，而且必须在冒号后使用特殊的初始化语法才能完成 (不能再在构造函数的正文中赋值常量)。有了显式构造函数，编译器就不会生成隐式 (无参数) 构造函数。但我们在 *Base* 结构体中没有显式无参数构造函数，如果没有它，任何派生类都不知道如何正确调用带参数的 *Base* 构造函数。因此，在 *Derived* 结构体中，必须显式地初始化基本构造函数：还需要在构造函数头文件中完成此操作，即在冒号 ':' 之后使用相应的初始化语法 (在本例中，我们调用 *Base(1)*)。

如果我们移除构造函数 *Derived*，基本构造函数中会出现“参数数量无效”错误，因为编译器会默认尝试调用 *Base* 的构造函数 (该构造函数应该只有 0 个参数)。

我们将在 [“类”一章](#) 中详细介绍语法和继承机制。

3.1.7 访问权限

如果有必要，可以在结构体说明中使用特殊关键字，这些关键字代表访问修饰符，可以限制字段在结构体外部的可见性。有三个修饰符：*public*、*protected* 和 *private*。默认情况下，所有结构体成员都是公共的，这相当于下面的条目（以 *Result* 结构体为例）：

```
struct Result
{
public:
    double probability;
    double coef[3];
    int direction;
    string status;
    ...
};
```

修饰符下面的所有成员都会获得相应的访问权限，除非遇到另一个修饰符或结构块结束。可以有许多具有不同访问权限的部分，但它们可以被任意修改。

标记为 *protected* 的成员只能从该结构体及其后代结构体的代码中访问，即假定这些成员必须有公共方法，否则任何对象都无法访问这些字段。

标记为 *private* 的成员只能在该结构体的代码中访问。例如，如果在 *status* 字段之前添加 *private*，则很可能需要一个方法来通过外部代码 (*getStatus*) 读取状态。

```
struct Result
{
public:
    double probability;
    double coef[3];
    int direction;

private:
    string status;

public:
    string getStatus()
    {
        return status;
    }
    ...
};
```

只能通过第二个构造函数的参数设置状态。直接访问该字段将导致错误“无法访问结构体‘Result’的私有成员‘status’”：

```
// error:
// cannot access to private member 'status' declared in structure 'Result'
r.status = "message";
```

在类中，默认访问权限为 *private*。这遵循封装原则，我们将在[关于“类”的章节](#)中进行介绍。

3.1.8 联合体

联合体是一种用户定义的类型，由位于同一内存区域的字段组成，原因是这些字段相互重叠。这样就可以将一种类型的值写入联合体，然后在另一种类型的解释中读取其内部表示（位级）。因此，我们可以提供从一种类型到另一种类型的非标准转换。

联合体字段可以是任何内置类型，但字符串、动态数组和指针除外。此外，在联合体中，可以使用具有相同简单字段类型且不带构造函数/析构函数的结构体。

编译器会为联合体分配一个内存单元，其大小等于所有元素类型的最大大小。因此，对于具有 *long* (8 字节) 和 *int* (4 字节) 等字段的联合体，将分配 8 字节。

联合体的所有字段都位于相同的内存地址，即在联合体的起始位置对齐（偏移量为 0，可以使用 `offsetof` 进行检查，参见 [打包结构体](#) 一节）。

描述联合体的语法与描述结构体类似，但使用的是 *union* 关键字。它的后面是一个标识符，然后是一个包含字段列表的代码块。

例如，某种算法可能会使用 *double* 类型的数组来存储各种设置，只因为 *double* 类型是最大字节数等于 8 的类型之一。假设设置中有诸如 *ulong* 的数字。由于 *double* 类型不能保证准确再现较大的 *ulong* 值，因此需要使用联合体将 *ulong* “打包”到 *double* 中，然后再“解包”回来。

```
#define MAX_LONG_IN_DOUBLE      9007199254740992
// FYI: ULONG_MAX              18446744073709551615

union ulong2double
{
    ulong U;    // 8 bytes
    double D;   // 8 bytes
};
ulong2double converter;

void OnStart()
{
    Print(sizeof(ulong2double)); // 8

    const ulong value = MAX_LONG_IN_DOUBLE + 1;

    double d = value; // possible loss of data due to type conversion
    ulong result = d; // possible loss of data due to type conversion

    Print(d, " / ", value, " -> ", result);
    // 9007199254740992.0 / 9007199254740993 -> 9007199254740992

    converter.U = value;
    double r = converter.D;
    Print(r); // 4.450147717014403e-308
    Print(offsetof(ulong2double, U), " ", offsetof(ulong2double, D)); // 0 0
}
```

ulong2double 结构体的大小等于 8，因为它的两个字段都具有此大小。因此，字段 U 和 D 完全重叠。

就整数而言，9007199254740992 是 `double` 类型能够可靠存储的最大精确整数值。在本例中，我们尝试用 `double` 来多存储一个数字。

从 `ulong` 到 `double` 的标准转换会造成精度损失：将 9007199254740993 写入 `double` 类型的变量 `d` 后，我们读取到的值是经过“舍入”的值 9007199254740992（有关以 `double` 类型存储数字的更多细节，请参见 [实数](#) 一节）。

使用转换器时，数字 9007199254740993 将“原样”地写入联合体，无需转换，因为我们将其赋值给 `ulong` 类型的 U 字段。该数字的 `double` 表示可从字段 D 中获得，同样没有经过转换。我们可以放心地将其复制到其他变量和数组（如 `double`）。

虽然生成的 `double` 值看起来很奇怪，但如果需要通过反向转换提取，它与原始整数完全匹配：写入 `double` 类型的 D 字段，然后从 `ulong` 类型的 U 字段读取。

联合体可以包含构造函数和析构函数，也可以包含方法。默认情况下，联合体成员具体公共访问权限，不过可以像在结构体中那样，使用访问修饰符进行调整。

3.2 类和接口

类是基于 OOP 方法进行程序开发的主要构建块。从广义上讲，“类”是指具有某些共同特征的东西（物、人、公式等）的集合。在 OOP 语境中，这一逻辑得以保留：一个类生成的对象具有一套相同的属性和行为。

在本书的前几章中，我们熟悉了 MQL5 的内置类型，如 `double`、`int` 或 `string`。编译器知道如何存储这些类型的值，以及可以对它们执行哪些操作。但是，这些类型可能不适用于描述任何应用领域。例如，交易员必须处理交易策略、信号过滤器、货币篮子和未平仓头寸组合等实体。每个实体都由一整套相关属性组成，并受特定的处理和一致性规则约束。

对这些对象进行自动操作的程序可能仅由内置类型和简单函数组成，但这样就必须想出一些棘手的方法来存储和链接属性。这时，OOP 技术就派上用场了，该技术为此提供了现成、统一且直观的机制。

OOP 提议将有关如何存储属性、正确填充属性以及允许对特定用户自定义类型的对象执行的所有指令，都编写到一个包含源代码的容器中。它以某种方式将变量和函数结合起来。如果按照能力和相关性从高到低的顺序排列，容器可分为类、结构体和关联。

在 [上一章](#) 中，我们已经接触过结构体和关联。我们学习的哪些知识也适用于类，但类在 OOP 工具箱中提供了更多工具。

类与结构体类似，是对用户定义类型的说明，具有任意的内部存储方法和使用规则。在此基础上，程序可以创建该类的实例，即应被视为复合变量的对象。

由用户定义的所有类型都共享一些基本概念，你可以将其称之为 OOP 理论，这些概念与类的关系最为紧密。包括：

- 抽象
- 封装
- 继承
- 多态性
- 组合（设计）

尽管这些概念看似很复杂，但都是现实世界中非常简单且常见的准则，只是被移植到了编程领域中。我们将从这些概念开始深入学习 OOP。至于用于描述类的语法以及如何创建对象，我们稍后再讨论。

3.2.1 OOP 基础知识：抽象

在日常生活中，我们经常使用概括性概念来表达信息的要点，而不需要纠缠细节。例如，在回答“你是怎么到这里来的”这个问题时，一个人可以直接说“开车来的”。这样一来，大家都会明白，我们所说的车有四个轮子、带发动机和供乘客乘坐的车身结构。汽车的具体品牌、颜色或生产年份对我们来说并不重要。

在使用程序时，只要程序能正确执行任务，用户也并不真正关心程序内部采用的是哪种算法。例如，可以使用十几种不同的方法对列表排序。

因此，抽象意味着提供一个简单的编程接口，隐藏所有复杂的实现细节。

编程接口是在类的上下文中定义的一组函数，根据对象的用途执行一系列操作。除了这些接口函数外，可能还有一些更小的辅助函数，但只能在类内部使用。与结构体类似，类的所有函数都有一个特殊的名称，称为方法。

实现通常使用属于对象的变量或数组（根据类描述）来存储信息。这些变量或数组称为“字段”这一术语来源于这样一个事实，即对象属性通常与用户界面中的输入字段或数据库中的字段以 1:1 的关系相关联，对象的当前状态会保存在这些字段中，以便下次启动程序时能够恢复。

字段和方法虽然是在类中描述的，但都与特定对象相关：每个实例都会分配一组自己的变量，它们的值与其他对象的状态无关，而方法则与其实例的字段搭配使用。

接口与实现必须相互独立。如果需要，一种实现方法应该很容易被另一种替换，而不会对编程接口产生任何影响。根据特定任务的需求来设计接口至关重要，而不是为了迎合特定的实现方式而定制接口。类的开发者需要能够从两种不同的视角审视自己创建的类：1) 作为内部算法和数据结构体的编写者；2) 作为潜在的挑剔客户，他们会将这个类看作一个“黑箱”，而接口就是这个“黑箱”的控制面板。推荐的做法是，在开始开发一个类时，首先考虑其编程接口的设计，然后再去寻找和选择具体的实现方法。

3.2.2 OOP 基础知识：封装

为了理解封装的概念，我们不妨回到现实生活。我们购买的家用电器通常是密封状态，并附带保修。我们可以在正常模式下使用，但制造商不建议我们拆开外壳“一探究竟”。例如，可以使用特殊工具将电脑处理器超频，但这也可能导致保修失效，因为这类操作可能导致设备故障。

类的开发也是如此。为了避免破坏类，任何人都不能直接访问其内部实现。这称为封装，即将所有重要的东西都包含在一个胶囊中。和 C++ 一样，MQL5 中也有 3 种级别的访问权限。默认情况下，类的组织方式为私有，即对所有用户隐藏。只有类自身的源代码才能访问其内容。

类的用户也包括程序员。即使是为自己编写类，也应该采取最严格的访问控制措施，以避免因疏忽而破坏类（毕竟人容易犯错，时间久了也可能忘记自己代码的细节，而且程序规模往往会越来越大）。

第二个访问级别允许“亲属”（更确切地说，是继承类；我们会在后面几个章节中再进行介绍）查看内部。

可以选择的第三个访问级别是公共访问级别，专门用于外部编程接口，这些接口允许从程序中的任何部分使用对象以实现对象的主要功能。

每个方法或字段都拥有这三种访问级别之一，具体由类的开发者决定。

3.2.3 OOP 基础知识：继承

在构建大型复杂项目时，人们总是寻求提高效率的方法。一种流行的方法是利用现有的开发成果。例如，开发一个建筑规划，基于之前的蓝图比从零开始要容易得多。

代码复用在编程中也非常流行。我们已经知道这样一种技术：将一段代码隔离成一个函数，然后在需要相应功能的不同位置调用函数。但 OOP 提供了一种更强大的机制：在开发一个新类时，它可以继承自另一个类，从而获得所有内部结构和外部接口，只需进行最小限度的调整即可满足需求。因此，从父类开始，您可以快速扩展一个具有附加或改进功能的派生类。此外，对父类的任何后续更改（例如增强功能或 bug 修复）都会自动影响所有子类。

如果一个类是另一个类的父类，则前者被称为基类。反过来，从基类继承的类被称为派生类。

当然，继承链（或者更形象地说，家谱）可以延续：每个类可以有多个继承人，这些继承人又有各自的继承人，依此类推。继承规则唯一不允许的是亲属关系的循环，例如，孙子不能成为祖父的父亲。

任何类与其任何一代的后代之间的关系都用“是 (is a)”这个词来描述，也就是说，后代可以充当祖先，但反之则不行。这是因为派生对象实际上包含祖先的数据模型，并为其添加了新的字段和行为。

通过相互继承类，我们有机会以统一的方式处理相关对象，因为它们的某些功能是通用的。

例如，假设一个绘图程序可以用来实现多种类型的形状，包括圆形、正方形、三角形等等。每个对象在屏幕上都有坐标（为简单起见，我们假设指定了形状中心的一对 X 和 Y 值）。此外，每个形状都使用自己的背景颜色、边框颜色和边框粗细进行渲染。

这意味着我们只需在描述抽象形状的父亲类中实现一次设置坐标和设置绘制样式的函数，这些函数就会自动被所有后代继承。

此外，为了简化源代码，不仅希望统一设置，也希望统一不同形状的绘制。这句话包含某种矛盾：既然形状各不相同，而且每种形状都必须以自己的方式显示，那么我们所说的统一究竟是什么？我们所说的是统一软件接口。确实，根据“抽象”的概念，有必要将外部接口与内部实现分离。而具体形状的显示本质上是一个实现细节。

统一的接口和针对不同形状类型的不同实现，顺理成章地引出了下一个概念：多态性。

3.2.4 OOP 基础知识：多态性

多态性是指可变性或多样性。多态性是抽象（结合了继承机制）的逆过程。当我们拥有一个通用的编程接口时，它可以由通过继承关系链接的不同类来实现。然后，调用接口方法将导致任务以不同的方式执行。

例如，想象一个抽象交通工具类族，其中包含两种特定类型：汽车和直升机。两者都能出色完成从点 A 移动到点 B 的命令，但汽车是在地面上行驶，而直升机则在空中飞行。

我们继续以绘图程序为例。我们可以说，其中的多样性体现在图形形状的层面上。用户可以自由绘制圆形、正方形和三角形的任意组合。这些对象中的每一个都必须能够使用其自身的坐标和样式在屏幕上显示，但最重要的是以一种能够产生合适形状的方式进行。

程序很可能会有一个数组（或其他容器）来存储用户创建的所有形状，而将整个绘图显示在屏幕上应该包含按顺序绘制每个形状的过程。如果我们将形状的绘制指令精简为一个单独的方法（我们称之为 *draw*），那么每个类都会有自己的实现。然而，这些函数的头文件将完全相同，因为它们执行相同的任务，并从对象中获取初始数据。

因此，我们有机会统一源代码，因为在形状循环中对 *draw* 方法的相同调用表现出多态性：所显示的形状将取决于对象的类型。

3.2.5 OOP 基础知识：组合（设计）

使用 OOP 进行程序设计时，存在一个问题，即根据某些给定的特征，找到最佳的类划分方法以及类之间的关系。“组合”一词可能存在歧义，并且经常用于表示不同的含义，包括对类“进行组合”的特殊情况。之所以要稍微离题一下，是因为在阅读其他计算机方面的文献时，您会发现对“组合”一词有不同的解释：既有广义的，也有狭义的。我们将尝试解释这个概念，并具体说明每个情况下这些术语的含义（当它表示软件界面的一般“设计/项目开发”时，以及当它表示“组合聚合”时）。

我们已经知道，类由字段（属性）和方法组成。属性反过来可以用自定义类型来描述，也就是说，它们可以是另一个类的对象。有几种方法可以逻辑方式连接这些对象：

- ① **将对象字段组合**（完全 纳入 或组合聚合）到所有者对象中。这些对象之间的关系可通过“整体-部分”关系描述，部分不能存在于整体之外。可以说所有者对象“拥有”属性对象，而属性对象是所有者对象的“一部分”。所有者创建并销毁其部分。删除所有者会移除其所有部分；所有者不能脱离部分而存在。
- ② **聚合**——所有者对象对对象字段的聚合是一种“更软”的包含关系。虽然这种关系也被描述为“整体-部分”，但所有者仅包含对可赋值、可更改且独立于整体存在的部分的引用。此外，一个部分可以被多个“所有者”使用。
- ③ **关联**，即独立对象之间的单向或双向连接，连接的含义可以任意规定；可以说一个对象“使用”另一个对象。

另一种需要记住的关系类型是“是（is a）”，我们在前面的 [继承](#) 一节讨论过。

完全包含的一个例子是汽车及其发动机。在这里，汽车被理解为一种功能齐全的交通工具。没有发动机的汽车显然不是。并且一个特定的发动机一次只能属于一辆汽车。当汽车中还没有发动机（在工厂）或发动机不再存在（在汽车修理厂）时，相当于我们破坏了程序的源代码。

聚合的一个例子是一起学习某些课程的学生小组：每门课程的小组包含多名学生，其中任何学生都可以属于其他小组（如果同时学习多个科目）。该小组“拥有”听众。一名学生退出小组不会影响小组的学习进程（其余学生将继续学习）。

最后，为了演示关联的概念，我们设想一台计算机和一台打印机。我们可以说计算机使用打印机进行打印。打印机可以根据需要打开或关闭，并且同一台打印机可以在不同的计算机上使用。所有计算机和打印机彼此独立存在，但可以共享。

至于通常用于指导类设计的特征，最著名的包括：

- ① **DRY**（不要重复自己）：将公共部分移到父类（可能是抽象类）中。
- ② **SRP**（单一职责原则）——一个类应该执行一项任务，如果不是这样，则需要将其拆分为更小的类。

①OCP (开放-封闭原则) —“编写的代码可以扩展但不能被修改”。如果 *X* 类中硬编码了多个计算选项，并且可能出现新的计算选项，则为单独的计算创建一个基类 (抽象类)，在其基础上创建特定选项 (功能的“扩展”)，并将其连接到类 *X* 而不进行修改。

这些只是类设计最佳实践中的一小部分。在掌握本书范围内的 OOP 基础知识之后，浏览其他关于该主题的专业信息来源可能也会对您有所帮助，因为它们为许多常见情况下的对象分解提供了现成的解决方案。

3.2.6 类定义

类定义语句包含许多可选组件，这些组件会影响其特性。其通用形式可以表示如下：

```
class class_name [: modifier_access name_parent_class ...]
{
    [ modifier_access:]
    [description_member...]
    ...
};
```

为了便于演示，我们将从最基本的可用语法开始，并随着学习内容的深入而不断延伸。

作为起点，我们使用一个包含条件绘图程序的任务，该程序支持多种形状。

要定义一个新类，请使用 `class` 关键字，后跟类标识符和用花括号括起来的代码块。与所有语句一样，这种定义必须以分号结尾。

代码块可以为空。例如，一个绘图程序的 *Shape* 类的可编译模板如下所示：

```
class Shape
{
};
```

从本书的前几章中，我们知道花括号表示变量的上下文或作用域。当这样的代码块出现在函数定义中时，它们定义了函数的局部上下文。除此之外，还有一个全局上下文，函数本身以及全局变量都在其中定义。

这次，类定义中的圆括号定义了一种新的上下文，即类上下文。它是类内部声明的变量和函数的容器。

用于存储类属性的变量的说明由块 (*Shapes1.mq5*) 内的常用语句完成。

```
class Shape
{
    int x, y;           // center coordinates
    color backgroundColor; // fill color
};
```

这里我们声明了理论小节讨论过的一些字段：形状中心的坐标和填充颜色。

完成这样的说明后，用户定义类型 *Shape* 便可以与内置类型一起用在程序中。具体来说，我们可以创建一个这种类型的变量并在其中包含指定的字段。但是，我们还不能对它们做任何操作，甚至不能确定它们是否存在。

```

void OnStart()
{
    Shape s;
    // errors: cannot access private member declared in class 'Shape'
    Print(s.x, " ", s.y);
}

```

默认情况下，类成员是私有的，因此不能从类外部代码的其他部分访问。这就是封装原则的作用。

如果我们尝试将形状输出到日志中，结果会让我们失望，原因有几个。

最直接的方法会导致错误“对象只能通过引用传递”（我们在结构体中也看到过这种情况）：

```
Print(s); // 's' - objects are passed by reference only
```

对象可能包含许多字段，由于其体积庞大，通过值传递的效率很低。因此，编译器要求对象类型参数通过引用传递，而 *Print* 则获取值。

在有关函数参数的章节（参见 [值参数和引用参数](#) 一节），我们知道符号 ‘&’ 用于描述引用。因此，我们可以合理地认为，要获取变量（本例中为 *Shape* 类型的 *s* 对象）的引用，必须在其名称前加上相同的符号。

```
Print(&s);
```

这条语句的编译和运行都没有问题，但并没有达到预期的效果。

程序在执行过程中会输出某个整数，例如 1 或 2097152（很可能不同）。变量名前的和号表示获取指向该变量的指针，而不是引用（与函数参数说明相反）。

指针 将在一个单独的小节中详细讨论。但请注意，MQL5 并不提供直接访问内存的功能，而指向对象的指针是一个描述符，或者简单地说，是一个唯一的对象编号（由终端自己分配）。但是，即使指针指向内存中的地址（就像 C++ 中那样），也无法提供读取对象内容的合法途径。

要将 *Shape* 对象的内容输出到日志或其他内容中，需要一个类成员函数。我们把它叫做 **toString**：它应该返回一个包含对象描述的字符串。我们可以稍后决定在字符串中显示什么内容。我们还要保留用于绘制形状的 *draw* 方法。现在，它将作为未来对象编程接口的声明。

```

class Shape
{
    int x, y;           // center coordinates
    color backgroundColor; // fill color

    string toString()
    {
        ...
    }

    void draw() { /* future drawing interface stub */ }
};

```

方法函数的定义方法与通常的方法相同，唯一不同的是它们位于构成类的代码块中。

今后，我们将学习如何将类代码块内的函数声明与 [类代码块外的定义](#) 分开。这种方法通常用于将声明放在头文件中，而将定义隐藏在 `mql5` 文件中。这使得代码更易于理解（因为编

程接口是以紧凑的形式单独呈现的，不含具体实现方式)。如果需要，还可以将 `软件库` 作为 `ex5` 文件分发 (不包含主源代码，只是提供一个足以调用外部接口方法的头文件)。

由于 `toString` 方法是类的一部分，因此它可以访问变量并将其转换为字符串。例如，

```
string toString()
{
    return (string)x + " " + (string)y;
}
```

但是，现在 `toString` 和 `draw` 是私有的，其余字段也是如此。我们需要将它们设为可以从类外部访问。

3.2.7 访问权限

为编辑类成员的访问权限，我们提供了一种特殊的语法 (在结构体一章中已经介绍过)。在块中类成员说明之前的任意位置，都可以插入一个修饰符：三个关键字 (`private`、`protected`、`public`) 之一和一个冒号。

修饰符之后的所有成员都将接受相应的可见性约束，除非遇到另一个修饰符或者类结束。

例如，下面的条目与前面对 `Shape` 类的说明相同，因为没有修饰符的类都假定为 `private` 模式：

```
class Shape
{
private:
    int x, y;           // center coordinates
    color backgroundColor; // fill color
    ...
};
```

如果我们想开放对所有字段的访问，可以将修饰符改为 `public`

```
class Shape
{
public:
    int x, y;           // center coordinates
    ...
};
```

但这违反了封装原则，我们不会这么做。但我们可以插入 `protected` 修饰符：它允许从派生类访问成员，同时又不对外界公开。我们计划将 `Shape` 类扩展到其他几个形状类，它们都需要访问父类的变量。

```

class Shape
{
protected:
    int x, y;           // center coordinates
    color backgroundColor; // fill color

public:
    string toString() const
    {
        return (string)x + " " + (string)y;
    }

    void draw() { /* shape drawing interface stub */ }
};

```

因此，我们将这两个函数都设为公共。

修饰符可以以任意方式交错出现在类说明中，并重复出现多次。不过，为了提高代码的可读性，建议将 *public*、*protected* 和 *private* 成员放在一个段中，并在项目的所有类中保持相同的顺序。

请注意，我们在 *toString* 函数头的末尾添加了 *const* 关键字。这意味着该函数不会改变对象字段的状态。尽管这不是必需的，但它有助于防止变量意外损坏，同时也让类的用户和编译器知道调用该函数不会产生任何副作用。

在 *toString* 函数中，就像在任何类方法中一样，字段可以通过名称访问。稍后，我们将看到如何将 *方法声明为静态*：它们完全与类相关，而与对象实例无关，因此字段不能被访问。

现在，我们可以从 *s* 对象变量中调用 *toString* 方法：

```

void OnStart()
{
    Shape s;
    Print(s.toString());
}

```

在这里，我们可以看到点字符 ‘.’ 用作特殊的取消引用操作符：它提供了对对象成员（字段和方法）的访问权限。它的左边应该是一个对象，右边是一个可用属性的标识符。

toString 方法是公共的，因此可以从 *OnStart* 类函数外部访问。如果我们试图在 *OnStart* 中通过取消引用来访问 *s.x* 或 *s.y* 字段，就会出现编译错误“无法访问 ‘Shape’ 类中声明的受保护成员”。

C++ 专业开发者需要注意一下，MQL5 不支持所谓的“发元”（对于其他语言的开发者，我们解释一下，在 C++ 中，如果有必要，可以将具有扩展权限的第三方类和方法列入“白名单”，尽管它们不是“亲属”）。

当我们运行程序时，会看到它输出了几个数字。不过，坐标值是随机的。即使偶尔看到空值，也不能保证下次运行脚本时会出现空值。通常，如果 MQL 程序执行列表在终端中没有变化，那么重复启动任何脚本都会为其分配相同的内存区域，这可能会给人一种“对象状态稳定”的假象。事实上，对象的字段和局部变量一样，默认情况下不会初始化（参见 *初始化* 一节）。

要对它们进行初始化，需要使用特殊的类函数—构造函数。

3.2.8 构造函数：默认、参数化和拷贝

我们已经在结构体一章中遇到过构造函数（参见 [构造函数和析构造函数](#) 一节）。对于类来说，它们的工作方式大致相同。我们回到重点，考虑更多特性。

构造函数是与类同名的方法，其类型为 `void`，即不返回值。通常，在构造函数名称前省略 `void` 关键字。一个类可以有多个构造函数：它们必须在参数数量或类型上有所不同。创建新对象时，程序会调用构造函数，以便为字段设置初始值。

要创建对象，我们采用的方法是在代码中描述相应类的变量。将对该字符串调用构造函数。这将自动进行。

根据参数是否存在以及类型，构造函数分为：

- 默认构造函数：无参数；
- 拷贝构造函数：只有一个参数，类型是对同一类中对象的引用；
- 参数化构造函数：具有任意参数集，除了上述用于拷贝的单一引用。

默认构造函数

最简单的构造函数，没有参数，称为默认构造函数。与 C++ 不同，MQL5 不会将“具有参数且所有参数均为默认值的构造函数”视为默认构造函数（也就是说，所有参数都是可选的，请参见 [可选参数](#) 一节）。

我们为 *Shape* 类定义一个默认构造函数。

```
class Shape
{
    ...
public:
    Shape()
    {
        ...
    }
};
```

当然，这应该在类的公共段内定义。

有时构造函数会被故意设置为受保护或私有，这是为了控制对象的创建方式（例如，通过工厂方法）。但在本例中，我们考虑的是类组合的标准版本。

要为对象变量设置初始值，我们可以使用通常的赋值语句：

```
public:
    Shape()
    {
        x = 0;
        y = 0;
        ...
    }
```

不过，构造函数语法提供了另一种选择。它被称为初始化列表，写在函数头之后，用冒号隔开。列表本身是以逗号分隔的字段名序列，每个字段名右侧是所需的初始值（用圆括号括起）。

例如，*Shape* 构造函数的代码如下：

```
public:
    Shape() :
        x(0), y(0),
        backgroundColor(cClrNONE)
    {
    }
```

这种语法比在构造函数体中赋值变量更受推崇，原因如下：

首先，函数体中的赋值是在创建相应变量的后进行的。根据变量的类型，这可能意味着首先调用默认构造函数，然后重写新值（这意味着有额外开销）。如果是初始化列表，则会立即用所需值创建变量。在没有初始化列表的情况下，编译器很可能会优化赋值，但在一般情况下，这并不能保证。

其次，某些类字段可以用 **const** 修饰符声明。然后，它们只能在初始化列表中设置。

第三，用户定义类型的字段变量可能没有默认构造函数（也就是说，其类中所有可用的构造函数都有参数）。这意味着在创建变量时，需要为其传递实际参数，而初始化列表允许您这样做：自变量值在圆括号内指定，就像在显式构造函数调用中一样。初始化列表可以在构造函数定义中使用，但不能在其他方法中使用。

参数化构造函数

根据定义，参数化构造函数有多个参数（至少一个）。

举例来说，假设为坐标 *x* 和 *y* 描述了一个具有参数化构造函数的特殊结构：

```
struct Pair
{
    int x, y;
    Pair(int a, int b): x(a), y(b) { }
};
```

这样，我们就可以使用新类型 *Pair* 的 *coordinates* 字段来代替 *Shape* 类中的两个整数字段 *x* 和 *y*。这种对象结构称为“包含”或“组合聚合”。*Pair* 对象是 *Shape* 对象的必要组成部分。坐标对与“宿主”对象一起自动创建和销毁。

由于 *Pair* 没有无参数构造函数，因此必须在 *Shape* 构造函数的初始化列表中指定 *coordinates* 字段，并指定两个参数 (*int, int*):

```

class Shape
{
protected:
    // int x, y;
    Pair coordinates; // center coordinates (object inclusion)
    ...
public:
    Shape() :
        // x(0), y(0),
        coordinates(0, 0), //object initialization
        backgroundColor(cclrNONE)
    {
    }
};

```

如果没有初始化列表，就无法创建此类自动对象。

鉴于对象中坐标存储方式的改变，我们需要更新 *toString* 方法：

```

string toString() const
{
    return (string)coordinates.x + " " + (string)coordinates.y;
}

```

但这还不是最终版本：我们很快还会做更多修改。

回顾一下，自动变量在 [声明/定义指令](#) 一节中介绍过。它们之所以被称为自动变量，是因为自动变量是由编译器自动创建的（分配内存），并在程序执行离开创建变量的上下文（代码块）时自动删除。

对于对象变量，自动创建不仅意味着内存分配，还意味着构造函数调用。自动删除对象时，也会调用其析构函数（参见下面的 [析构函数](#) 一节）。此外，如果对象是另一个对象的一部分，那么它的生命周期与其“所有者”的生命周期一致，例如，*coordinates* 字段就是 *Shape* 对象中 *Pair* 的实例。

静态（包括全局）对象也由编译器自动管理。

一种替代自动分配的方法是 [通过指针进行动态对象创建和操作](#)。

在 [继承](#) 一节中，我们将学习类的继承。这种情况下，只有通过初始化列表才能调用基类的参数化构造函数（不同于默认构造函数的情况，编译器不会隐式自动生成带参数的构造函数调用）。

我们为 *Shape* 类添加另一个构造函数，用于为变量设置特定值。它将只是一个参数化构造函数（您可以创建任意多个构造函数：用于不同的目的，使用不同的参数集）。

```

Shape(int px, int py, color back) :
    coordinates(px, py),
    backgroundColor(back)
{
}

```

初始化列表可确保在执行构造函数主体时，所有内部字段（包括嵌套对象，如有）都已创建并初始化。

类成员的初始化顺序与初始化列表无关，而是与它们在类中的声明顺序一致。

如果在类中声明了带参数的构造函数，并且需要允许创建不带自变量的对象，程序员必须明确实现默认构造函数

如果类中没有任何构造函数，编译器会以存根的形式隐式提供一个默认构造函数，负责初始化以下类型的字段：字符串、动态数组和具有默认构造函数的自动对象。如果没有此类字段，则隐式默认构造函数不会执行任何操作。其他类型的字段不受隐式构造函数的影响，因此它们将包含随机“垃圾”。为了避免这种情况，程序员必须明确声明构造函数并设置初始值。

拷贝构造函数

使用拷贝构造函数，您在创建对象时可以基于另一个对象，后者是通过引用而传递的，并作为唯一参数。

例如，对于 *Shape* 类，拷贝构造函数可能如下所示：

```
class Shape
{
    ...
    Shape(const Shape &source) :
        coordinates(source.coordinates.x, source.coordinates.y),
        backgroundColor(source.backgroundColor)
    {
    }
    ...
};
```

请注意，由于权限是在类层面上生效，所以当前对象可以访问同一类的另一个对象的受保护和私有成员。换句话说，当给定一个引用（或 指针）时，同一个类的两个对象可以互相访问彼此的数据。

如果有这样一个构造函数，就可以使用两种语法类型之一来创建对象：

```
void OnStart()
{
    Shape s;
    ...
    Shape s2(s);    // ok: syntax 1 - copying
    Shape s3 = s;   // ok: syntax 2 - copying via initialization
                    //                (if there is copy constructor)
                    //                - or assignment
                    //                (if there is no copy constructor,
                    //                but there is default constructor)

    Shape s4;       // definition
    s4 = s;         // assignment, not copy constructor!
}
```

有必要区分创建对象期间初始化与赋值之间的区别。

即使没有拷贝构造函数，但有默认构造函数，也可以使用第二种方法（标有“syntax 2”注释）。在这种情况下，编译器生成的代码效率较低：首先，编译器会使用默认构造函数创建一个接收变

量的空实例（本例中为 `s3`），然后逐元素复制样本的字段（本例中为 `s`）。事实上，这种情况与 `s4` 变量的情况相同，该变量的定义和赋值是通过单独的语句完成的。

如果没有拷贝构造函数，那么尝试使用第一种语法将出现错误“不允许参数转换”，因为编译器会尝试使用某个其他具有不同参数集的构造函数。

切记，如果类包含带 `const` 修饰符的字段，则禁止为此类对象赋值，原因显而易见：常量字段不能更改，只能在创建对象时设置一次。因此，拷贝构造函数是复制对象的唯一方式。

特别是在下面几个章节中，我们将完成 *Shape1.mq5* 示例，下面的字段将出现在 *Shape* 类中（具有说明字符串 `type`）。然后，赋值运算符就会产生错误（尤其是在包含 `s4` 变量的行中）：

```
attempting to reference deleted function
'void Shape::operator=(const Shape&)'
function 'void Shape::operator=(const Shape&)' was implicitly deleted
because member 'type' has 'const' modifier
```

得益于编译器详尽的提示信息，您可以清楚地理解问题的核心和缘由：首先，错误信息中指出的是赋值运算符（`=`），而非拷贝构造函数；其次，它说明了赋值运算符由于 `const` 修饰符的存在而被隐式地移除了。在这里，我们会遇到一些还没学过的概念，稍后我们将对其进行研究：[类中的运算符重载](#)、[对象类型转换](#)以及将方法标记为[已删除](#)的功能。

在[继承](#)一节中，学习了如何描述派生类之后，我们需要对类层次结构中的拷贝构造函数做一些说明。

3.2.9 析构函数

在结构体一章中，我们学习了析构函数（参见[构造函数和析构函数](#)章节）。我们简单回顾一下：析构函数是对象被销毁时调用的方法。析构函数与类的名称相同，但前缀是一个波浪号（`~`）。析构函数不返回值，也没有任何参数。一个类只能有一个析构函数。

即使类没有析构函数或析构函数为空，编译器也会隐式地对以下类型的字段进行“垃圾回收”：字符串、动态数组和自动对象。

通常，析构函数会放在类的公共部分，但在某些特殊情况下，开发者可以将其移到一组 `private` 或 `protected` 成员中。私有或受保护的析构函数不允许在代码中声明该类的自动变量。不过，我们稍后会了解[动态对象的创建](#)，对它们来说，这样的限制还是有意义的。

特别是，有些对象的实现方式是，当不再需要它们时，它们必须删除自己（用于确定需求的概念可能不同）。也就是说，只要程序的任何部分还在使用对象，这些对象就会一直存在；一旦任务完成，这些对象就会自动销毁（私有析构函数保留了从类方法中删除对象的可能性）。

对于有经验的 C++ 程序员来说，请务必注意，MQL5 中的析构函数总是虚拟的（更多关于虚方法的内容将在[虚方法（`virtual`和`override`）](#)一节中介绍）。这一因素不会影响说明的语法。

在绘图程序示例中，从技术上讲，析构函数对于形状来说可能不是必需的。不过，为了追踪构造函数和析构函数的调用顺序，我们还是要加入一个。我们从输出方法全名的简化大纲开始：

```

class Shape
{
    ...
    ~Shape()
    {
        Print(__FUNCSIG__);
    }
};

```

我们很快就会在这个方法和其他方法中添加新内容，以便区分对象的不同实例。

考虑以下示例。在两种不同的上下文中描述一对对象 *Shape*: 全局 (在函数外部) 和局部 (在 *OnStart* 内部)。全局对象构造函数将在加载脚本后、调用 *OnStart* 前调用，而析构函数将在卸载脚本前调用。局部对象的构造函数将在包含变量定义的行中调用，而析构函数将在包含变量定义的代码块退出时调用，在本例中就是 *OnStart* 函数。

```

// the global constructor and destructor are related to script loading and unloading
Shape global;

```

```

// object reference does not create a copy and does not affect lifetime
void ProcessShape(Shape &shape)
{
    // ...
}

```

```

void OnStart()
{
    // ...
    Shape local; // <- local constructor call
    // ...
    ProcessShape(local);
    // ...
} // <- local destructor call

```

按引用将对象传递给其他函数不会创建对象的副本，也不会调用构造函数和析构函数。

3.2.10 自引用: this

在每个类的上下文中，其方法代码中都存在一个特殊引用，指向当前对象本身: *this*。它本质上是一个隐式定义的变量。所有使用对象变量的方法也同样适用于 *this*。具体而言，您可以通过取消引用 *this* 来访问对象字段或调用某个方法。例如，在 *Shape* 类的一个方法中，以下两条语句是等效的 (为方便演示，我们仅以 *draw* 方法为例):

```

class Shape
{
    ...
    void draw()
    {
        backgroundColor = clrBlue;
        this.backgroundColor = clrBlue;
    }
};

```

如果同一上下文内存在其他同名的变量或参数，可能需要使用长形式。虽然通常不建议这样做，但在必要时，**this** 关键字可让您引用对象的重写成员。

如果任何局部变量或方法参数的名称与类成员变量的名称重叠，则编译器会发出警告。

在以下假设示例中，我们实现了 `draw` 方法，该方法接受一个可选的字符串参数 `backgroundColor`，表示颜色名称。由于参数名称与 `Shape` 类成员同名，编译器会发出第一个警告“`backgroundColor` 的定义隐藏了该字段”。

这种重叠的后果是，后面错误地赋值 `clrBlue` 时，实际操作的是参数而不是类成员。由于值和参数类型不匹配，编译器将发出第二个警告“数字到字符串隐式转换”（这里的数字是常量 `clrBlue`）。但是，`this.backgroundColor = clrBlue` 行将值写入对象的字段。

```

void draw(string backgroundColor = NULL) //warning 1:
    // declaration of 'backgroundColor' hides member
{
    ...
    backgroundColor = clrBlue; // warning 2:
    // implicit conversion from 'number' to 'string'
    this.backgroundColor = clrBlue; // ok

    {
        bool backgroundColor = false; // warning 3:
        // declaration of 'backgroundColor' hides local variable
        ...
        this.backgroundColor = clrRed; // ok
    }
    ...
}

```

随后（在嵌套的花括号块中）定义的局部布尔变量 `backgroundColor` 再次重写了之前对该名称的定义（这就是我们收到第三个警告的原因）。然而，通过取消引用 `this`，`this.backgroundColor = clrRed` 语句仍会引用对象字段。

如果没有指定 `this`，编译器会一律选择上下文中最接近的名称定义。

还有另一种需要使用 `this` 的情况：将当前对象作为参数传递给另一个函数。特别是，采用了一种方法，其中同一个类的对象负责创建/删除另一个类的对象，而从属对象必须知道其“上级”。然后，使用构造函数在“上级”类中创建从属对象，并将“上级”对象的 `this` 传递给它。这种技术通常使用动态对象分配和指针，因此 [指针](#) 一节中显示了相关示例。

`this` 的另一个常见用途是从成员函数返回指向当前对象的指针。这使得我们可以链式调用多个成员函数。由于我们尚未深入学习指针，目前只需要了解，指向某个类对象的指针是通过在该类名后添加星号 `*` 来声明的，并且可以通过指针像直接操作对象一样对其进行操作。

例如，我们可以为用户提供几种单独设置形状属性的方法：更改颜色、水平或垂直移动。其中每个方法都将返回指向当前对象的指针。

```
Shape *setColor(const color c)
{
    backgroundColor = c;
    return &this;
}

Shape *moveX(const int x)
{
    coordinates.x += x;
    return &this;
}

Shape *moveY(const int y)
{
    coordinates.y += y;
    return &this;
}
```

然后，可以方便地链式调用这些方法。

```
Shape s;
s.setColor(clrWhite).moveX(80).moveY(-50);
```

当一个类拥有大量属性时，这种方式能够让您以简洁且有选择性地配置对象。

在 [类定义](#) 一节中，我们曾尝试记录一个对象变量，但发现只能在 *Print* 调用中使用其名称并加上 `&` 符号来获取一个指针，或者更确切地说，是一个唯一的数字标识符（句柄）。在对象上下文中，同样的句柄也可以通过 *&this* 来获取。

为了方便调试，您可以通过对象的描述符来识别对象。接下来我们将探讨类继承，尤其是存在多个类继承时，这种识别机制将非常有用。因此，在所有的构造函数和析构函数中，我们都添加（并且日后在派生类中也会添加）以下 *Print* 调用：

```
~Shape()
{
    Print(__FUNCSIG__, " ", &this);
}
```

现在，所有创建和删除步骤都将在日志中标记上类名和对象编号。

我们在 *Pair* 结构体中实现了类似的构造函数和析构函数，但遗憾的是，结构体不支持指针，即无法写入 *&this*。因此，我们只能通过其内容（在本例中，是通过其坐标）来识别这些结构体：

```
struct Pair
{
    int x, y;
    Pair(int a, int b): x(a), y(b)
    {
        Print(__FUNCSIG__, " ", x, " ", y);
    }
    ...
};
```

3.2.11 继承

在定义类时，开发者可以让它继承自另一个类，从而体现 **继承的概念**。为此，需要在类名后面加上一个冒号，然后是一个可选的访问权限修饰符（可以是 **public**、**protected**、**private** 这几个关键字之一），最后是父类的名称。例如，下面展示了如何定义一个 **Rectangle** 类，使其派生自 **Shape** 类：

```
class Rectangle : public Shape
{
};
```

类的头文件中的访问修饰符控制包含在子类中的父类成员的“可见性”：

- 🕒 **public** -所有继承的成员保留其权限和限制
- 🕒 **protected** -将继承的 **public** 成员的权限更改为 **protected**
- 🕒 **private** -使所有继承的成员变为 **private** (**private**)

在绝大多数定义中使用修饰符 **public**。另外两个选项只有在极少数特殊情况下才有意义，因为它们违背了继承的基本原则：派生类的对象应该是父类家族的“是一个”关系：完全具备父类特征的代表。如果我们“截断”它们的权限，它们就会失去一部分应有的特性。结构体也可以用类似的方式相互继承。禁止类继承自结构体，也禁止结构体继承自类。

与 C++ 不同，MQL5 不支持多重继承。一个类最多只能有一个父类。

一个派生类内置一个基类对象。考虑到基类本身也可能继承自其他的父类，所创建的对象可以形象地比作俄罗斯套娃。

在新类中，我们需要一个构造函数，它可填充对象的字段，就像在基类中那样。

```
class Rectangle : public Shape
{
public:
    Rectangle(int px, int py, color back) :
        Shape(px, py, back)
    {
        Print(__FUNCSIG__, " ", &this);
    }
};
```

在这种情况下，初始化列表就简化为对 **Shape** 构造函数的一次调用。不能直接在初始化列表中设置基类的变量，因为基类的初始化工作是由基类构造函数负责的。不过，如有必要，我们可以在 **Rectangle** 构造函数的主体中更改基类的 **protected** 字段（函数主体中语句的执行是基本构造函数在初始化列表中完成了调用之后）。

矩形具有两个维度，因此我们添加两个受保护的字段 **dx** 和 **dy** 来表示它们。为了设置这两个字段的值，您需要补充构造函数参数列表。

```

class Rectangle : public Shape
{
protected:
    int dx, dy; // dimensions (width, height)

public:
    Rectangle(int px, int py, int sx, int sy, color back) :
        Shape(px, py, back), dx(sx), dy(sy)
    {
    }
};

```

需要注意的是, *Rectangle* 对象会隐式地包含继承自 *Shape* 的 *toString* 函数 (但是, *draw* 也在哪里, 但目前还是空的)。因此, 以下代码是正确的:

```

void OnStart()
{
    Rectangle r(100, 200, 50, 75, clrBlue);
    Print(r.toString());
};

```

这不仅演示了如何调用 *toString*, 还演示了如何使用我们的新构造函数创建矩形对象。

Rectangle 类中没有默认构造函数 (无任何参数)。这意味着该类的用户无法以简单的方式 (不带自变量) 创建矩形对象:

```
Rectangle r; // 'Rectangle' - wrong parameters count
```

编译器将显示错误 “自变量数量无效”。

我们来创建另一个子类 *Ellipse*。目前, 除了名称之外, 它与 *Rectangle* 没有任何不同。稍后, 我们将介绍它们之间的区别。

```

class Ellipse : public Shape
{
protected:
    int dx, dy; // dimensions (large and small radii)
public:
    Ellipse(int px, int py, int rx, int ry, color back) :
        Shape(px, py, back), dx(rx), dy(ry)
    {
        Print(__FUNCSIG__, " ", &this);
    }
};

```

随着类数量的增多, 在 *toString* 方法中显示类名会非常方便。在 特殊的 `sizeof` 和 `typename` 运算符 一节中, 我们介绍了 *typename* 运算符。现在我们来尝试使用。

回想一下, *typename* 需要一个参数, 它返回该参数的类型名称。例如, 如果我们分别创建了 *Shape* 和 *Rectangle* 类的对象 *s* 和 *r*, 我们可以通过以下方式获取它们的类型:

```

void OnStart()
{
    Shape s;
    Rectangle r(100, 200, 75, 50, clrRed);
    Print(typename(s), " ", typename(r));    // Shape Rectangle
}

```

但是，我们需要以某种方式在类内部获取这个名称。为此，我们向参数化构造函数 *Shape* 添加一个字符串参数，并将其存储在一个新字符串字段 *type* 中（注意 *protected* 部分和 *const* 修饰符：该字段对外部世界是隐藏的，一旦对象创建完毕，便无法修改）：

```

class Shape
{
protected:
    ...
    const string type;

public:
    Shape(int px, int py, color back, string t) :
        coordinates(px, py),
        backgroundColor(back),
        type(t)
    {
        Print(__FUNCSIG__, " ", &this);
    }
    ...
};

```

在派生类的构造函数中，我们使用 *typename(this)* 填充基本构造函数的这个参数：

```

class Rectangle : public Shape
{
    ...
public:
    Rectangle(int px, int py, int sx, int sy, color back) :
        Shape(px, py, back, typename(this)), dx(sx), dy(sy)
    {
        Print(__FUNCSIG__, " ", &this);
    }
};

```

现在，我们可以利用 *type* 字段来改进 *toString* 方法。

```

class Shape
{
    ...
public:
    string toString() const
    {
        return type + " " + (string)coordinates.x + " " + (string)coordinates.y;
    }
};

```

我们验证一下我们这个小小的类层次结构是否能按预期创建对象，以及在构造函数和析构函数被调用时能否打印测试日志条目。

```

void OnStart()
{
    Shape s;
    //setting up an object by chaining calls via 'this'
    s.setColor(clrWhite).moveX(80).moveY(-50);
    Rectangle r(100, 200, 75, 50, clrBlue);
    Ellipse e(200, 300, 100, 150, clrRed);
    Print(s.toString());
    Print(r.toString());
    Print(e.toString());
}

```

最终，我们会得到类似以下的日志输出（特意添加空行以分隔不同对象的输出）：


```

Pair::Pair(int,int) 0 0
Shape::Shape() 1048576

Pair::Pair(int,int) 100 200
Shape::Shape(int,int,color,string) 2097152
Rectangle::Rectangle(int,int,int,int,color) 2097152

Pair::Pair(int,int) 200 300
Shape::Shape(int,int,color,string) 3145728
Ellipse::Ellipse(int,int,int,int,color) 3145728

Shape 80 -50
Rectangle 100 200
Ellipse 200 300

Ellipse::~~Ellipse() 3145728
Shape::~~Shape() 3145728
Pair::~~Pair() 200 300

Rectangle::~~Rectangle() 2097152
Shape::~~Shape() 2097152
Pair::~~Pair() 100 200

Shape::~~Shape() 1048576
Pair::~~Pair() 80 -50

```

从日志中可以清晰地看到构造函数和析构函数的调用顺序。

对于每个对象而言，首先创建其中描述的对象字段（如果有），然后沿着继承链依次调用基类的构造函数以及所有派生类的构造函数。如果在派生类中存在某些对象类型的自有（已添加）字段，则这些字段的构造函数将在该派生类的构造函数之前被立即调用。有多个对象字段时，将按照在类中描述的顺序创建。

析构函数的调用顺序与构造函数刚好相反。

可以在派生类中定义拷贝构造函数，这部分内容已经在 [构造函数：默认构造函数、参数化构造函数、拷贝构造函数](#) 中介绍过。对于特定的形状类型，例如矩形，其拷贝构造函数的语法类似：

```

class Rectangle : public Shape
{
    ...
    Rectangle(const Rectangle &other) :
        Shape(other), dx(other.dx), dy(other.dy)
    {
    }
    ...
};

```

作用域会略微扩大。派生类对象可用于拷贝给基类（因为派生类包含了基类的所有数据）。不过在这种情况下，派生类中新增的字段自然会被忽略。

```

void OnStart()
{
    Rectangle r(100, 200, 75, 50, clrBlue);
    Shape s2(r);           // ok: copy derived to base

    Shape s;
    Rectangle r4(s);       // error: no one of the overloads can be applied
                           // requires explicit constructor overloading
}

```

要实现反向拷贝，您需要在基类中提供一个引用派生类的构造函数版本（这在理论上违反了 OOP 的原则），否则会出现编译错误：“没有重载可以应用于函数调用”。

现在，我们可以用脚本来编写几个甚至更多个形状变量，然后让它们使用 **draw** 方法来绘制自身。

```

void OnStart()
{
    Rectangle r(100, 200, 50, 75, clrBlue);
    Ellipse e(100, 200, 50, 75, clrGreen);
    r.draw();
    e.draw();
};

```

然而，这种方式会将形状数量、类型和参数的数量硬编码到程序中，而我们本应能够自己选择创建何种形状以及在何处绘制。因此，我们需要动态地创建形状。

3.2.12 动态创建对象：new 和 delete

到目前为止，我们只尝试创建了自动对象，也就是 **OnStart** 函数内部的局部变量。在全局上下文中声明的对象（在 **OnStart** 或其他函数外部）也会被自动创建（加载脚本时）和删除（卸载脚本时）。

除了这两种模式之外，我们还提到了描述对象类型的字段的能力（在我们的示例中，这是 **Shape** 对象内部用于 **coordinates** 字段的 **Pair** 结构体）。所有这些对象也都是自动的：由编译器在“宿主”对象的构造函数中创建，并在其析构函数中删除。

然而，在程序中仅依赖自动对象往往无法满足需求。在绘图程序中，我们需要根据用户的请求创建形状。此外，形状需要存储在数组中，而为了实现这一点，自动对象必须拥有默认构造函数（但在我们的例子中并非如此）。

对于这种情况，MQL5 提供了动态创建和删除对象的能力。创建通过 **new** 运算符实现，删除通过 **delete** 运算符实现。

运算符 new

new 关键字后面是所需类的名称，以及用于调用所有现有构造函数的自变量列表（用圆括号括起）。执行 **new** 运算符会导致创建类的实例。

new 运算符会返回特殊类型的值，即指向对象的指针。要描述这种类型的变量，需要在类名后添加星号 *****。例如：

```
Rectangle *pr = new Rectangle(100, 200, 50, 75, clrBlue);
```

在这里, *pr* 变量的类型是指向 *Rectangle* 类对象的指针。指针将在一个单独的 一节中详细讨论。

需要注意的是, 声明一个对象指针类型的变量本身并不会为对象分配内存, 也不会调用其构造函数。当然, 指针本身会占用一定空间 (8 个字节), 但实际上, 它是一个无符号长整型 *ulong*, 系统会以一种特殊的方式来解读它。

您可以像操作对象一样操作指针, 也就是说, 可以通过取消引用运算符来调用可用的方法, 并且可以访问字段。

```
Print(pr.toString());
```

一个尚未被赋予动态对象描述符的指针变量 (例如, *new* 运算符不是在初始化新变量时立即调用, 而是放在了源代码后面的某些行), 会包含一个特殊的空指针, 用 *NULL* 表示 (以区别于数字), 但实际上其值为 0。

运算符 *delete*

在算法结束时, 应该使用 *delete* 运算符释放通过 *new* 获得的指针。例如:

```
delete pr;
```

如果不这样做, *new* 运算符分配的实例将一直驻留在内存中。如果以这种方式创建越来越多的新对象, 并且不需要时没有将其删除, 这将导致不必要的内存消耗。程序终止时, 剩余的未释放动态对象会导致打印警告信息。例如, 如果您不删除指针 *pr*, 那么在脚本卸载后, 您将在日志中看到类似这样的信息:

```
1 undeleted object left
1 object of type Rectangle left
168 bytes of leaked memory
```

终端会报告程序员忘记释放了多少个对象以及它们的类名, 以及它们占用了多少内存。

一旦对某个指针调用了 *delete* 运算符, 该指针就会失效, 因为对象已不再存在。随后尝试访问其属性会导致运行时错误: “访问了无效指针”:

```
Critical error while running script 'shapes (EURUSD,H1)'.
Invalid pointer access.
```

MQL 程序随后会被中断。

然而, 这并不意味着同一个指针变量无法再使用。只需要将该指针分配给另一个新创建的对象实例即可。

MQL5 有一个内置函数 *CheckPointer*, 可用于检查变量中指针的有效性:

```
ENUM_POINTER_TYPE CheckPointer(object *pointer);
```

它接受一个指向类类型的指针作为参数, 并返回 *ENUM_POINTER_TYPE* 枚举类型的值:

- *POINTER_INVALID* - 指针不正确;
- *POINTER_DYNAMIC* - 指向动态对象的有效指针;
- *POINTER_AUTOMATIC* - 指向自动对象的有效指针。

只有当 `CheckPointer` 函数返回 `POINTER_DYNAMIC` 时, 执行 ***delete*** 语句才有意义。对于自动对象, `delete` 操作不会产生任何影响 (这类对象在控制流离开定义该变量的代码块时会自动删除)。

以下宏简化并确保指针的正确清理:

```
#define FREE(P) if(CheckPointer(P) == POINTER_DYNAMIC) delete (P)
```

为了获得动态对象和指针所提供的灵活性, 显式地进行“清理”是不可避免的代价。

3.2.13 指针

正如我们在 [类定义](#) 一节中所述, MQL5 中的指针是对象的一些描述符 (唯一编号), 而不是像 C++ 中那样的内存地址。对于自动对象, 我们通过在其名称前添加一个与号 (&) 来获取指针 (在这种上下文中, 与号是“取地址”运算符)。因此, 在以下示例中, 变量 *p* 指向自动对象 *s*。

```
Shape s;           // automatic object
Shape *p = &s;     // a pointer to the same object
s.draw();          // calling an object method
p.draw();          // doing the same
```

在前面的小节中, 我们学习了如何通过使用 ***new*** 动态创建对象来获取指向该对象的指针。此时, 不需要使用与号来获取描述符: 指针的值本身就是描述符。

MQL5 API 提供了函数 ***GetPointer***, 它的作用与与号运算符 ‘&’ 相同, 即返回对象的指针:

```
void *GetPointer(Class object);
```

使用这两种方式中的哪一种取决于个人偏好。

指针常用于将对象链接在一起。我们来演示一下创建从属对象的思路, 这些从属对象接收指向其对象创建程序 (*ThisCallback.mq5*) 的 *this* 的指针。我们在关于 [this](#) 关键字的章节中提到过这个技巧。

现在我们尝试利用这个技巧来实现一个方案, 以便不定期向“创建程序”通知从属对象中执行的计算完成百分比: 我们之前使用 [函数指针](#) 实现了类似的功能。***Manager*** 类负责控制计算, 实际计算 (很可能使用不同的公式) 则在单独的类中执行。本例中展示了其中一个类, 即 ***Element***。

```

class Manager; // preliminary announcement

class Element
{
    Manager *owner; // pointer

public:
    Element(Manager &t): owner(&t) { }

    void doMath()
    {
        const int N = 1000000;
        for(int i = 0; i < N; ++i)
        {
            if(i % (N / 20) == 0)
            {
                // we pass ourselves to the method of the control class
                owner.progressNotify(&this, i * 100.0f / N);
            }
            // ... massive calculations
        }
    }

    string getName() const
    {
        return typename(this);
    }
};

class Manager
{
    Element *elements[1]; // array of pointers (1 for demo)

public:
    Element *addElement()
    {
        // looking for an empty slot in the array
        // ...
        // passing to the constructor of the subclass
        elements[0] = new Element(this); // dynamic creation of an object
        return elements[0];
    }

    void progressNotify(Element *e, const float percent)
    {
        // Manager chooses how to notify the user:
        // display, print, send to the Internet
        Print(e.getName(), "=", percent);
    }
};

```

从属对象可以使用接收到的链接向“上级”通知工作进度。计算完成时，会向控制对象发送一个信号，表明可以删除计算器对象，或者让另一个计算器对象开始工作。当然，*Manager* 类中固定的单一元素数组可能不太起眼，但作为演示，足以说明问题。*Manager* 类不仅管理计算任务的分发，还提供了一个抽象层以实现用户通知功能：不仅可以将消息输出到日志，它可以写入单独文件、显示在屏幕上或发送到互联网。

顺便说一句，请注意在 *Element* 类定义之前对 *Manager* 类进行了前置声明。这样做是必要的，为的是在 *Element* 类中声明一个指向 *Manager* 类的指针，而 *Manager* 类会出现在后面的代码中进行定义。如果省略前向声明，我们会遇到错误：“*Manager*”-意外的标记，可能缺少类型？”。

当两个类通过其成员相互引用时，就需要使用前向声明：在这种情况下，无论我们以何种顺序排列这两个类的定义，都无法完整地定义其中任何一个。前向声明允许我们在没有完整定义的情况下保留一个类型名称。

指针的一个基本特性是，指向基类的指针可用来指向任何派生类的对象。这是 多态性 的一种体现。这种行为之所以可行，是因为派生类对象包含了父类的内置“子对象”，就像层层嵌套的俄罗斯套娃一样。

特别是对于我们的形状绘制任务，可以很容易地描述一个指向 *Shape* 指针的动态数组，并根据用户的请求向其中添加不同类型的对象。

类的数量将扩展到五个 (*Shapes2.mq5*)。除了 *Rectangle* 和 *Ellipse*，我们还将添加 *Triangle*，并创建一个派生自 *Rectangle* 且表示正方形的类 (*Square*)，以及一个派生自 *Ellipse* 且表示圆形的类 (*Circle*)。显然，正方形是边长相等的矩形，而圆形是长轴半径和短轴半径相等的椭圆。

为了沿着继承链传递字符串类名，我们在 *Rectangle* 和 *Ellipse* 类的 *protected* 部分添加带有额外字符串参数 *t* 的特殊构造函数：

```
class Rectangle : public Shape
{
protected:
    Rectangle(int px, int py, int sx, int sy, color back, string t) :
        Shape(px, py, back, t), dx(sx), dy(sy)
    {
    }
    ...
};
```

然后，在创建正方形时，我们不仅设置相等的边长，还从 *Square* 类传递 *typename(this)*：

```
class Square : public Rectangle
{
public:
    Square(int px, int py, int sx, color back) :
        Rectangle(px, py, sx, sx, back, typename(this))
    {
    }
};
```

此外，我们将 *Shape* 类中的构造函数移至 *protected* 部分：这一操作将禁止创建 *Shape* 对象自身，它只能作为其后代类的基类。

我们指定 *addRandomShape* 函数来生成形状，该函数返回指向新创建对象的指针。为了便于演示，此函数现在将实现随机生成形状：类型、位置、大小和颜色都是随机的。

支持的形状类型在 SHAPES 枚举中进行汇总：这些形状对应于五个已实现的类。

给定范围内的随机数由 *random* 函数返回（使用内置函数 *rand*，每次调用时返回一个 0 到 32767 范围内的随机整数）。形状的中心坐标在 0 到 500 像素的范围内生成，形状的大小最大为 200。颜色由三个 RGB 分量组成（参见 [颜色](#) 一节），每个分量的范围从 0 到 255。

```

int random(int range)
{
    return (int)(rand() / 32767.0 * range);
}

Shape *addRandomShape()
{
    enum SHAPES
    {
        RECTANGLE,
        ELLIPSE,
        TRIANGLE,
        SQUARE,
        CIRCLE,
        NUMBER_OF_SHAPES
    };

    SHAPES type = (SHAPES)random(NUMBER_OF_SHAPES);
    int cx = random(500), cy = random(500), dx = random(200), dy = random(200);
    color clr = (color)((random(256) << 16) | (random(256) << 8) | random(256));
    switch(type)
    {
        case RECTANGLE:
            return new Rectangle(cx, cy, dx, dy, clr);
        case ELLIPSE:
            return new Ellipse(cx, cy, dx, dy, clr);
        case TRIANGLE:
            return new Triangle(cx, cy, dx, clr);
        case SQUARE:
            return new Square(cx, cy, dx, clr);
        case CIRCLE:
            return new Circle(cx, cy, dx, clr);
    }
    return NULL;
}

void OnStart()
{
    Shape *shapes[];

    // simulate the creation of arbitrary shapes by the user
    ArrayResize(shapes, 10);
    for(int i = 0; i < 10; ++i)
    {
        shapes[i] = addRandomShape();
    }

    // processing shapes: for now, just output to the log
    for(int i = 0; i < 10; ++i)
    {
        Print(i, ": ", shapes[i].toString());
    }
}

```



```

        delete shapes[i];
    }
}

```

我们生成 10 个形状并将它们输出到日志（由于是随机选择类型和属性，结果可能会有所不同）。别忘了使用 *delete* 删除对象，因为这些对象是动态创建的（这里在同一个循环中完成删除是因为这些形状之后不再使用；在实际程序中，形状数组很可能会以某种方式存储到文件中，以便稍后加载并继续处理图像）。

```

0: Ellipse 241 38
1: Rectangle 10 420
2: Circle 186 38
3: Triangle 27 225
4: Circle 271 193
5: Circle 293 57
6: Rectangle 71 424
7: Square 477 46
8: Square 366 27
9: Ellipse 489 105

```

这些形状已成功创建，并输出了它们的属性信息。

现在我们已经准备好访问我们类的 API，也就是 *draw* 方法。

3.2.14 虚方法（virtual 和 override）

类旨在描述外部编程接口并提供其内部实现。由于我们的测试程序的功能是绘制各种形状，我们在 *Shape* 类及其后代中描述了若干变量以供未来实现，并且也为接口预留了 *draw* 方法。

在 *Shape* 基类中，它不应也无法进行任何操作，因为 *Shape* 不是一个具体的形状：我们稍后会将其 *Shape* 转换为抽象类（稍后我们会详细讨论 [抽象类和接口](#)）。

我们重定义 *Rectangle*、*Ellipse* 和其他派生类（*Shapes3.mq5*）中的 *draw* 方法。这涉及拷贝该方法并相应地修改其内容。尽管许多人将此过程称为“重写”，但我们将区分“重写”和“重定义”这两个术语，将“重写”专门用于虚方法（将在稍后讨论）。

严格来说，重定义方法只要求方法名称匹配。但为了确保在全部代码中用法一致，必须维护相同的参数列表和返回类型。

```

class Rectangle : public Shape
{
    ...
    void draw()
    {
        Print("Drawing rectangle");
    }
};

```

由于我们还不知道如何在屏幕上绘制，我们暂时只将消息输出到日志。

务必要注意，通过在派生类中提供该方法的新实现，我们因此得到了该方法的两个版本：第一个引用内置的基对象（内部的 *Shape*），第二个引用派生类对象（外部的 *Rectangle*）。

使用 *Shape* 类型的变量时，将调用第一个版本；使用 *Rectangle* 类型的变量时，将调用第二个版本。

在更长的继承链中，一个方法可以被多次重定义和多次传播。

可以更改新方法的访问类型，例如，将其从 `protected` (受保护) 改为 `public` (公开)，反之亦然。但在本例中，我们将 *draw* 方法保留在 `public` 部分中。

如果需要，程序员可以调用任何祖先类的方法实现：为此，需要使用一个特殊的 [上下文解析运算符](#)，即两个冒号 '::'。具体而言，我们可以从 *Square* 类的 *draw* 方法中调用 *Rectangle* 类的 *draw* 实现：为此，我们指定所需类的名称、 '::' 和方法名称，例如 *Rectangle::draw()*。调用 *draw* 时不指定上下文即表示调用当前类的方法。因此，如果直接在 *draw* 方法本身中调用，将会导致无限递归，最终导致堆栈溢出和程序崩溃。

```
class Square : public Rectangle
{
public:
    ...
    void draw()
    {
        Rectangle::draw();
        Print("Drawing square");
    }
};
```

然后，在 *Square* 对象上调用 *draw* 会在日志中记录两行：

```
Square s(100, 200, 50, clrGreen);
s.draw(); // Drawing rectangle
          // Drawing square
```

将方法绑定到声明该方法的类，可以实现静态调度 (或静态绑定)：编译器在编译阶段决定调用哪个方法，并将找到的匹配项“硬编码”到二进制代码中。

在决策过程中，编译器会在执行取消引用 (.) 的类的对象中查找要调用的方法。如果该方法存在，则调用该方法；如果不存在，则编译器检查父类中是否存在该方法，依此类推，沿着继承链查找，直到找到该方法。如果在继承链的任何类中都找不到该方法，则会出现编译错误“未声明的标识符”。

具体来说，以下代码在 *Rectangle* 对象上调用 *setColor* 方法：

```
Rectangle r(100, 200, 75, 50, clrBlue);
r.setColor(clrWhite);
```

但是，此方法仅在基类 *Shape* 中定义，并且在所有后代类中仅构建一次，因此它将在此处执行。

我们尝试在 *OnStart* 函数中从数组开始绘制任意形状 (回想一下，我们已经在所有后代类中复制并修改了 *draw* 方法)。

```
for(int i = 0; i < 10; ++i)
{
    shapes[i].draw();
}
```

奇怪的是，日志中没有任何输出。这是因为程序调用了 *Shape* 类的 *draw* 方法。

静态调度在这里有一个主要缺点：当我们使用指向基类的指针来存储派生类的对象时，编译器会根据指针的类型（而不是对象本身）来选择方法。事实上，在编译阶段，我们并不清楚指针在程序执行期间会指向哪个类对象。

因此我们需要一种更灵活的方法：动态调度（也称为绑定），可将方法的选择（包括后代链中所有被重写的方法版本）推迟到运行时。必须在分析指针所指向对象的实际类之后再进行选择。正是动态调度提供了 **多态性** 原则。

这种方法是在 MQL5 中使用虚方法实现。在描述这种方法时，必须在头文件开头添加 **virtual** 关键字。

我们将 *Shape* 类 (*Shapes4.mq5*) 中的 *draw* 方法声明为虚方法。这会自动将其在派生类中的所有版本都设为虚方法。

```
class Shape
{
    ...
    virtual void draw()
    {
    }
};
```

一旦方法被虚拟化，在派生类中对其进行修改便称为重写，而不是重定义。重写要求与方法的名称、参数类型和返回值匹配（考虑 **const** 修饰符存在与否）。

请注意，重写虚函数不同于 **函数重载**。重载使用相同的函数名，但参数不同（具体来说，我们在结构体的示例中看到了重载构造函数的可能性，请参见 **构造函数和析构函数** 一节），而重写则要求函数签名完全匹配。

被重写的函数必须定义在不同的类中，这些类之间存在继承关系。重载函数必须位于同一个类中，否则它不会被视作为重载，而很可能被视为重定义（并且工作方式会有所不同，参见对 *OverrideVsOverload.mq5* 示例的深入分析）。

如果您运行新脚本，日志中将出现预期的行，表明调用了每个类中 *draw* 方法的特定版本。

```
Drawing square
Drawing circle
Drawing triangle
Drawing ellipse
Drawing triangle
Drawing rectangle
Drawing square
Drawing triangle
Drawing square
Drawing triangle
```

在重写虚方法的派生类中，建议在其头文件中添加 **override** 关键字（虽然不是强制要求）。

```
class Rectangle : public Shape
{
    ...
    void draw() override
    {
        Print("Drawing rectangle");
    }
};
```

这可以让编译器知道我们是有意重写该方法。如果将来基类的 API 突然发生变化，导致被重写的方法不再是虚拟的（或直接被移除），编译器将生成一条错误消息：“声明方法时使用了‘override’说明符，但并未重写任何基类方法”。请注意，即使为方法添加或移除 **const** 修饰符，也会更改其签名，重写可能会因此而中断。

也可以在被重写的方法前使用 **virtual** 关键字，但并非强制要求。

为了使动态调度正常工作，编译器会为每个类生成一个虚函数表。每个对象都会添加一个隐式字段，其中包含指向其给定类表的链接。表由编译器进行填充，内容是关于特定类的继承链上所有虚方法及其重写版本的信息。

对虚方法的调用在二进制程序映像中以一种特殊的方式编码：首先，查找该表以查找特定对象（位于指针处）的类版本，然后转换到相应的函数。

因此，动态调度比静态调度慢。

在 MQL5 中，无论是否存在虚方法，类始终包含一个虚函数表。

如果虚方法返回指向类的指针，则在重写该类时，可以更改（使其更具体、更特化）返回值的对象类型。换句话说，指针的类型不仅可以与虚方法的初始声明相同，还可以与其任何后继类型相同。这种类型称为“协变”或可互换。

例如，如果我们在 *Shape* 类中将 *setColor* 方法设为虚方法：

```
class Shape
{
    ...
    virtual Shape *setColor(const color c)
    {
        backgroundColor = c;
        return &this;
    }
    ...
};
```

我们可以在 *Rectangle* 类中重写它，如下所示（仅作为该技术的演示）：

```

class Rectangle : public Shape
{
    ...
    virtual Rectangle *setColor(const color c) override
    {
        // call original method
        // (by pre-lightening the color,
        // no matter what for)
        Rectangle::setColor(c | 0x808080);
        return &this;
    }
};

```

请注意，返回类型是指向 *Rectangle* 的指针，而非指向 *Shape* 的指针。

如果方法的重写版本修改了对象中不属于基类的部分，导致对象实际上不再符合基类所允许的状态（不变式），那么使用类似技巧很有用。

我们绘制形状的示例即将完成。剩下的就是用实际内容填充虚方法 *draw*。我们将在 [图形](#) 一章中完成这项工作（参见示例 *ObjectShapesDraw.mq5*），但我们将在学习 [图形资源](#) 后对其进行改进。

考虑到继承的概念，编译器选择合适方法的过程看起来有点令人困惑。根据方法名称和调用指令中具体的自变量（其类型）列表，编译器会编译所有可用候选方法的列表。

如果不是虚方法，一开始只会分析当前类的方法。如果没有匹配的方法，编译器将继续搜索基类（然后再搜索更远的祖先类，直到找到匹配的方法）。如果在当前类的方法中找到一个合适的方法（即使需要隐式转换自变量类型），编译器会优先选取这个方法。即使基类中的方法具有更合适的自变量类型（无需转换或转换次数较少），编译器也不会选用。换句话说，非虚方法的分析将从当前对象的类开始，逐级向祖先类进行，直到找到第一个“有效”匹配。

对于虚方法，编译器首先在指针类中按名称查找所需方法，然后在虚函数表中，为指针类型和对象类型之间的继承链中实例化程度最高的类（最远的后代）选择重写了该方法的实现。在这种情况下，如果自变量类型没有完全匹配，也可以使用隐式自变量转换。

我们考虑以下示例（*OverrideVsOverload.mq5*）。链中有 4 个类：*Base*、*Derived*、*Concrete* 和 *Special*。它们都包含自变量类型为 *int* 和 *float* 的方法。在 *OnStart* 函数中，整数变量 *i* 和实数变量 *f* 用作所有方法调用的自变量。

```

class Base
{
public:
    void nonvirtual(float v)
    {
        Print(__FUNCSIG__, " ", v);
    }
    virtual void process(float v)
    {
        Print(__FUNCSIG__, " ", v);
    }
};

class Derived : public Base
{
public:
    void nonvirtual(int v)
    {
        Print(__FUNCSIG__, " ", v);
    }
    virtual void process(int v) // override
    // error: 'Derived::process' method is declared with 'override' specifier,
    // but does not override any base class method
    {
        Print(__FUNCSIG__, " ", v);
    }
};

class Concrete : public Derived
{
};

class Special : public Concrete
{
public:
    virtual void process(int v) override
    {
        Print(__FUNCSIG__, " ", v);
    }
    virtual void process(float v) override
    {
        Print(__FUNCSIG__, " ", v);
    }
};

```

首先, 我们创建 *Concrete* 类的对象和一个指向此对象的指针 *Base *ptr*。然后, 我们调用它们的非虚方法和虚方法。在第二部分中, 通过 *Base* 和 *Derived* 类指针来调用 *Special* 对象的方法。

```

void OnStart()
{
    float f = 2.0;
    int i = 1;

    Concrete c;
    Base *ptr = &c;

    // Static link tests

    ptr.nonvirtual(i); // Base::nonvirtual(float), conversion int -> float
    c.nonvirtual(i);   // Derived::nonvirtual(int)

    // warning: deprecated behavior, hidden method calling
    c.nonvirtual(f);   // Base::nonvirtual(float), because
                        // method selection ended in Base,
                        // Derived::nonvirtual(int) does not suit to f

    // Dynamic link tests

    // attention: there is no method Base::process(int), also
    // there are no process(float) overrides in classes up to and including Concrete
    ptr.process(i);    // Base::process(float), conversion int -> float
    c.process(i);      // Derived::process(int), because
                        // there is no override in Concrete,
                        // and the override in Special does not count

    Special s;
    ptr = &s;
    // attention: there is no method Base::process(int) in ptr
    ptr.process(i);    // Special::process(float), conversion int -> float
    ptr.process(f);    // Special::process(float)

    Derived *d = &s;
    d.process(i);      // Special::process(int)

    // warning: deprecated behavior, hidden method calling
    d.process(f);      // Special::process(float)
}

```

日志输出如下所示。

```

void Base::nonvirtual(float) 1.0
void Derived::nonvirtual(int) 1
void Base::nonvirtual(float) 2.0
void Base::process(float) 1.0
void Derived::process(int) 1
void Special::process(float) 1.0
void Special::process(float) 2.0
void Special::process(int) 1
void Special::process(float) 2.0

```

使用静态绑定调用 `ptr.nonvirtual(i)`，并将整数 `i` 预先转换为参数类型 `float`。

对 `c.nonvirtual(i)` 的调用也是静态的，由于 `Concrete` 类中没有 `void nonvirtual(int)` 方法，因此编译器会在 `Derived` 父类中查找这样的方法。

对同一对象使用 `float` 类型的值调用同名函数会将编译器转到 `Base::nonvirtual(float)` 方法，因为 `Derived::nonvirtual(int)` 不适用（转换会造成精度损失）。在此过程中，编译器会发出警告“行为已被弃用，调用了隐藏方法”。

重载的方法与重写的方法（名称相同，但参数不同）看似相同，但实际并不同，因为重写的方法位于不同的类中。当派生类中的方法重写父类中的方法时，它会替换父类方法的行为，这有时可能会导致意外的结果。程序员可能期望编译器选择其他合适的方法（就像重载一样），但实际上却调用了子类。

为了避免可能的警告，如果需要在派生类中将其编写为完全相同的函数，并从中调用基类。

```

class Derived : public Base
{
public:
    ...
    // this override will suppress the warning
    // "deprecated behavior, hidden method calling"
    void nonvirtual(float v)
    {
        Base::nonvirtual(v);
        Print(__FUNCSIG__, " ", v);
    }
    ...
}

```

我们回到 `OnStart` 中的测试。

调用 `ptr.process(i)` 演示了上述所示的重载和重写之间的区分难点。`Base` 类有一个 `process(float)` 虚方法，而 `Derived` 类添加了一个新的虚方法 `process(int)`，由于参数类型不同，该方法在本例中不属于重写。编译器根据基类中的名称选择一个方法，并在虚函数表中检查继承链中直至 `Concrete` 类（包含 `Concrete` 本身，这是指针指向的对象类）是否存在重写。由于未找到重写，编译器采用了 `Base::process(float)` 并将自变量的类型转换应用于参数（`int` 转换为 `float`）。

如果我们遵循“在表示重定义的情况下使用采用 `override`”这一规则，并将其添加到 `Derived` 中，则会遇到错误：


```

class Derived : public Base
{
    ...
    virtual void process(int v) override // error!
    {
        Print(__FUNCSIG__, " ", v);
    }
};

```

编译器会报告“声明 ‘Derived::process’ 方法时使用了 ‘override’ 说明符，但并未重写任何基类方法”。这条错误可以作为修复问题的提示。

在 *Concrete* 对象上调用 *process(i)* 是通过 *Derived::process(int)* 完成的。虽然我们在 *Special* 类中进行了进一步的重定义，但这无关紧要，因为它是在 *Concrete* 类之后的继承链中完成的。

当 *ptr* 指针稍后赋值给 *Special* 对象时，编译器会将 *process(i)* 和 *process(f)* 的调用解析为 *Special::process(float)*，因为 *Special* 重写了 *Base::process(float)*。选择 *float* 参数的原因与前面描述的相同：*Base::process(float)* 方法被 *Special* 重写了。

如果我们应用 *Derived* 类型的指针 *d*，那么我们最终会得到对字符串 *d.process(i)* 的预期调用 *Special::process(int)*。关键在于，*process(int)* 是在 *Derived* 中定义的，并且属于编译器的搜索范围。

请注意，*Special* 类不仅重写了继承的虚方法，还重载了类本身中的两个方法。

请勿在构造函数或析构函数中调用虚函数！虽然技术上可行，但构造函数和析构函数中的虚行为会完全失效，您可能会得到意想不到的结果。不仅应避免显式调用，还应避免间接调用（例如，从构造函数调用简单方法，而该方法又调用虚方法）。

我们以构造函数为例更详细地分析一下这种情况。事实上，在构造函数执行时，对象尚未完全沿着整个继承链组装完成，而仅组装到当前类。所有派生部分围绕现有核心部分的收尾工作尚未完成。因此，所有后续的虚方法重写（如果有）此时均不可用。因此，从构造函数中调用的是方法的当前版本。

3.2.15 静态成员

到目前为止，我们已经讨论了类的字段和方法，它们描述了给定类的对象的状态和行为。但在程序中，可能需要存储某些特性或对整个类（而不是其对象）执行操作。这种类属性称为静态，使用在类型前添加的 *static* 关键字进行描述。结构体和联合体也支持这样的类属性。

例如，我们可以在绘图程序中计算用户创建的形状数量。为此，我们将在 *Shape* 类中描述静态变量 *count(Shapes5.mq5)*。

```

class Shape
{
private:
    static int count;

protected:
    ...
    Shape(int px, int py, color back, string t) :
        coordinates(px, py),
        backgroundColor(back),
        type(t)
    {
        ++count;
    }

public:
    ...
    static int getCount()
    {
        return count;
    }
};

```

该变量在 *private* 部分中定义，因此无法从外部访问。

为读取当前计数器值，提供了一个公共静态方法 *getCount()*。理论上，由于静态成员是在类的上下文中定义的，因此它们的可见性会根据其所在部分的修饰符受到限制。

我们将在参数化构造函数 *Shape* 中将计数器加 1，并移除默认构造函数。因此，任何派生类型的形状的每个实例都将被考虑在内。

请注意，静态变量必须在类块之外显式定义（可以选择进行初始化）：

```
static int Shape::count = 0;
```

静态类变量类似于函数内的全局变量和静态变量（参见 [静态变量](#) 部分），因为它们在程序启动时创建，并在程序卸载前被删除。因此，与对象变量不同，它们必须从一开始就作为单个实例存在。

在这种情况下，可以省略零初始化，因为众所周知，全局变量和静态变量默认设为零。数组也可以是静态的。

在静态变量的定义中，我们看到使用了特殊的上下文选择运算符 `::`。使用这个运算符，可以形成一个完全限定的变量名。`::` 左侧是变量所属类的名称，右侧是其标识符。显然，完全限定名很有必要，因为在不同的类中，可以声明具有相同标识符的静态变量，并且需要一种方法能够唯一引用每个变量。

相同的 `::` 运算符不仅用于访问公共静态类变量，还用于访问方法。具体来说，为了在 *OnStart* 函数中调用 *getCount* 方法，我们使用语法 *Shape::getCount()*：

```

void OnStart()
{
    for(int i = 0; i < 10; ++i)
    {
        Shape *shape = addRandomShape();
        shape.draw();
        delete shape;
    }

    Print(Shape::getCount()); // 10
}

```

由于现在正在生成指定数量的形状 (10)，我们可以确认计数器正常运行。

如果您有一个类对象，您可以通过正常的取消引用来引用静态方法或属性（例如，`shape.getCount()`），但这样的符号可能会产生误导（因为它隐藏了该对象实际上未被访问这一事实）。

请注意，创建派生类不会以任何方式影响静态变量和方法：它们始终会被赋值给定义了它们的类。我们的计数器对于所有派生自 *Shape* 的形状的类都相同。

您不能在静态方法中使用 *this*，因为静态方法在执行时并不绑定到特定对象。此外，在静态方法中，如果不取消引用任何对象类型变量，则无法直接调用常规类方法或访问其字段。例如，如果您从 `getCount` 中调用 `draw`，则会遇到错误“访问非静态成员或函数”：

```

static int getCount()
{
    draw(); // error: 'draw' - access to non-static member or function
    return count;
}

```

出于同样的原因，静态方法不能为虚拟。

是否可以使用静态变量来计算形状的总数，而不是按类型计算其统计信息？是的，可以。此任务留待读者独立研究。感兴趣的读者可以在 *Shapes5stats.mq5* 脚本中找到一个实现示例。

3.2.16 嵌套类型、命名空间和上下文运算符 '::'

类、结构体和联合体不仅可以在全局上下文中描述，还可以在其他类或结构体中描述。甚至还可以在函数内部进行定义。这样，您可以在适当的上下文中描述任何类或结构体运行所需的所有实体，从而避免潜在的名称冲突。

具体来说，在绘图程序中，用于存储坐标的结构体 *Pair* 到目前为止是在全局定义的。随着程序规模增长，很可能需要另一个名为 *Pair* 的实体（尤其考虑到这个名称相当通用）。因此，最好将结构体的描述移到 *Shape* (*Shapes6.mq5*) 类内部。

```

class Shape
{
public:
    struct Pair
    {
        int x, y;
        Pair(int a, int b): x(a), y(b) { }
    };
    ...
};

```

嵌套说明的访问权限取决于指定的段修饰符。在本例中，我们将名称 *Pair* 设为公开可用。在 *Shape* 类内部，*Pair* 结构体类型的处理不会因转移而发生任何变化。但是，在外部代码中，您必须指定一个完全限定名称，其中包含外部类（上下文）的名称、上下文选择运算符 `::` 以及内部实体标识符本身。例如，要描述具有一对坐标的变量，可以这样写：

```
Shape::Pair coordinates(0, 0);
```

描述实体时的嵌套层级不受限制，因此完全限定名称可以包含由 `::` 分隔的多层级标识符（上下文）。例如，我们可以将所有绘图类包装在外部类 *Drawing* 的 *public* 部分中。

```

class Drawing
{
public:
    class Shape
    {
    public:
        struct Pair
        {
            ...
        };
    };
    class Rectangle : public Shape
    {
        ...
    };
    ...
};

```

然后，完全限定的类型名称（例如，用于 *OnStart* 或其他外部函数）将被加长：

```

Drawing::Shape::Rect coordinates(0, 0);
Drawing::Rectangle rect(200, 100, 70, 50, clrBlue);

```

虽然这很不方便，但在包含许多类的大型项目中，有时这很有必要。在我们的小型项目中，这种方法仅用于演示技术可行性。

为了将逻辑相关的类和结构体组合成命名组，MQL5 提供了一种比将它们包含在“空”包装器类中更简单的方法。

命名空间使用关键字 *namespace* 声明，后跟命名空间的名称以及包含所有必要定义的代码块（用花括号括起）。以下是使用 *namespace* 的相同绘图程序的示例：

```

namespace Drawing
{
    class Shape
    {
    public:
        struct Pair
        {
            ...
        };
    };
    class Rectangle : public Shape
    {
        ...
    };
    ...
}

```

主要区别在于：空间的内部内容始终公开可用（访问修饰符在其中不适用），并且右花括号后没有分号。

我们为 *Shape* 类添加 *move* 方法，该方法以 *Pair* 结构体作为参数：

```

class Shape
{
public:
    ...
    Shape *move(const Pair &pair)
    {
        coordinates.x += pair.x;
        coordinates.y += pair.y;
        return &this;
    }
};

```

然后，在 *OnStart* 函数中，您可以通过调用此函数来整理所有形状按给定值的偏移：

```

void OnStart()
{
    //draw a random set of shapes
    for(int i = 0; i < 10; ++i)
    {
        Drawing::Shape *shape = addRandomShape();
        // move all shapes
        shape.move(Drawing::Shape::Pair(100, 100));
        shape.draw();
        delete shape;
    }
}

```

请注意，*Shape* 和 *Pair* 类型必须分别使用全名 *Drawing::Shape* 和 *Drawing::Shape::Pair* 来描述。

可能有多个块的空间名称相同：它们的所有内容都将归入一个具有指定名称的逻辑统一上下文中。

全局上下文中定义的标识符，特别是 MQL5 API 的所有内置函数，也可以通过上下文选择运算符（前面不带任何符号）访问。例如，对 *Print* 函数的调用可能如下所示：

```
::Print("Done!");
```

从全局上下文中定义的任何函数进行调用时，不需要这样的条目。

如果类或结构体中定义了同名元素（函数、变量或常量），则其中需要这样的条目例如，我们将 *Print* 方法添加到 *Shape* 类中：

```
static void Print(string x)
{
    // empty
    // (likely will output it to a separate log file later)
}
```

由于派生类中 *draw* 方法的测试实现会调用 *Print*，因此它们现在被重定向到此 *Print* 方法：在多个相同标识符中，编译器会选择在更接近上下文中定义的标识符。在本例中，基类中的定义比全局上下文更接近 *Shape* 类。因此，*Shape* 类的日志输出将被抑制。

但是，从函数 *OnStart* 调用 *Print* 仍然有效（因为它位于 *Shape* 类的上下文之外）。

```
void OnStart()
{
    ...
    Print("Done!");
}
```

要修复类中的调试打印问题，您需要在所有 *Print* 调用之前添加一个全局上下文选择运算符：

```
class Rectangle : public Shape
{
    ...
    void draw() override
    {
        ::Print("Drawing rectangle"); // reprint via global Print(...)
    }
};
```

3.2.17 拆分类声明和定义

在大型软件项目中，为了方便起见，可以将类分成简短说明（声明）和定义（包含主要实现细节）。在某些情况下，如果类之间以某种方式相互引用，即没有事先声明就无法完整定义任何一个类，则这种隔离很有必要。

我们在 [指标](#) 一节中看到了一个前向声明示例（参见文件 *ThisCallback.mq5*），其中 *Manager* 和 *Element* 类包含相互指向的指针。其中的类以一种简短的形式预先声明：格式为包含关键字 *class* 和名称的头文件：

```
class Manager;
```

然而，这是尽可能简短的声明。它只注册名称，并且可以将编程接口的说明推迟到某个时间，但这种说明必须在代码的后面部分遇到。

更常见的是，声明包含接口的完整说明：它指定类的所有变量和方法头，但不包含它们的主体（代码块）。

方法定义是单独编写的：其方法头使用完全限定名称，名称中包含类名（如果方法上下文高度嵌套，则使用多个类和命名空间）。所有类名和方法名使用上下文选择运算符 '::' 连接。

```
type class_name [:: nested_class_name...] :: method_name([parameters...])
{
}
```

理论上，您可以直接在类说明块中定义方法的某一部分（通常小型函数会这样做），而某些部分则可以单独提取出来（通常是大型函数）。但是，每个方法只能有一个定义（也就是说，不允许先在类块中定义方法，然后再单独定义）和一个声明（类块中的定义也是声明）。

方法声明和定义中的参数列表、返回类型和 **const** 修饰符（如果有）必须完全匹配。

我们来看看如何将类的描述和定义与脚本 *ThisCallback.mq5*（[指针](#)一节中的示例）分开：我们创建一个名为 *ThisCallback2.mq5* 的类似脚本。

前置声明 *Manager* 仍然位于开头。此外，*Element* 和 *Manager* 这两个类仅仅声明，但没有实现：仅以分号结束，没有方法体代码块。

```
class Manager; // preliminary announcement

class Element
{
    Manager *owner; // pointer
public:
    Element(Manager &t);
    void doMath();
    string getName() const;
};

class Manager
{
    Element *elements[1]; // array of pointers (replace with dynamic)
public:
    ~Manager();
    Element *addElement();
    void progressNotify(Element *e, const float percent);
};
```

源代码的第二部分包含所有方法的实现（实现本身保持不变）。

```

Element::Element(Manager &t) : owner(&t)
{
}

void Element::doMath()
{
    ...
}

string Element::getMyName() const
{
    return typename(this);
}

Manager::~~Manager()
{
    ...
}

Element *Manager::addElement()
{
    ...
}

void Manager::progressNotify(Element *e, const float percent)
{
    ...
}

```

结构体也支持单独的方法声明和定义。

请注意，构造函数初始化列表（位于名称和 ‘:’ 之后）是定义的一部分，因此必须位于函数体之前（换句话说，在仅有头文件的构造函数声明中不允许使用初始化列表）。

通过单独编写声明和定义，可以开发源代码必须封闭的 **库**。在本例中，声明放在一个扩展名为 **mqh** 的单独头文件中，而定义放在一个扩展名为 **mq5** 的同名文件中。该程序被编译并以 **ex5** 文件（附带描述外部接口的头文件）的形式分发。

在这种情况下，可能会出现一个问题：为什么部分内部实现（尤其是数据（变量）的组织方式）在外部接口中可见？严格来说，这表明类层次结构的抽象程度不足。所有提供外部接口的类都不应公开任何实现细节。

换句话说，如果我们设定的目标是从某个库中导出上述类，那么我们需要将它们的方法拆分为若干基类（提供 API 说明，但不含数据字段），而 **Manager** 和 **Element** 则继承自这些基类。同时，在基类的方法中，我们不能使用任何来自派生类的数据，而且，总的来说，它们根本不包含任何实现。这怎么可能呢？

为此，需要使用抽象方法、抽象类和接口的技术。

3.2.18 抽象类和接口

为了探索抽象类和接口，我们回到端到端绘图程序示例。为了简单起见，它的 API 包含一个虚方法 `draw`。到目前为止，它一直是空的，但与此同时，即使是这样的空实现也是一个具体的实现。但是，`Shape` 类的对象无法绘制，它们的形状尚未定义。因此，将 `draw` 方法设为抽象方法（或者说纯虚方法）是合理的。

为此，应移除带有空实现的代码块，并在方法头中添加 `= 0`：

```
class Shape
{
public:
    virtual void draw() = 0;
    ...
}
```

至少有一个抽象方法的类也会变成抽象的，因为它的对象无法创建：没有实现。具体来说，我们的 `Shape` 构造函数可供派生类使用（这要归功于 `protected` 修饰符），并且它们的开发人员理论上可以创建一个 `Shape` 对象。但这是在声明抽象方法之前，声明抽象方法之后我们都停止了这种行为，因为我们（绘图接口的作者）禁止这样做。编译器会抛出错误：

```
'Shape' -cannot instantiate abstract class
'void Shape::draw()' is abstract
```

要描述接口，最佳方法是为接口创建一个抽象类，其中仅包含抽象方法。在我们的例子中，`draw` 方法应该移到新类 `Drawable` 中，并且 `Shape` 类应该从其继承（`Shapes.mq5`）。

```
class Drawable
{
public:
    virtual void draw() = 0;
};

class Shape : public Drawable
{
public:
    ...
    // virtual void draw() = 0; // moved to base class
    ...
};
```

当然，接口方法必须位于 `public` 段中。

MQ5 提供了另一种便捷的接口描述方式，即使用 `interface` 关键字。接口中的所有方法虽已声明但并未实现，被视为公共和虚拟。与上述类等效的 `Drawable` 接口的说明如下：

```
interface Drawable
{
    void draw();
};
```

在本例中，如果抽象类中没有任何字段（这将违反抽象原则），则其后代类无需进行任何更改。

现在需要扩展接口，并将 `setColor`、`moveX` 和 `moveY` 这三个方法也添加到接口中。

```
interface Drawable
{
    void draw();
    Drawable *setColor(const color c);
    Drawable *moveX(const int x);
    Drawable *moveY(const int y);
};
```

请注意，这些方法返回一个 *Drawable* 对象，因为我对 *Shape* 一无所知。在 *Shape* 类中，我们已经有了适合重写这些方法的实现，因为 *Shape* 继承自 *Drawable* (*Shape* 在某种程度上也算是一种 *Drawable* 对象)。

现在，第三方开发者可以将其他类型的 *Drawable* 类添加到绘图程序中，具体来说，不仅包括形状，还包括文本、位图，甚至还能包括其他 *Drawable* 的集合（惊喜吧！），这样您便可以将对象嵌套在一起，实现复杂的组合。只需从接口中集成并实现其方法即可。

```
class Text : public Drawable
{
public:
    Text(const string label)
    {
        ...
    }

    void draw()
    {
        ...
    }

    Text *setColor(const color c)
    {
        ...
        return &this;
    }
    ...
};
```

如果形状类以二进制 `ex5` 库（不包含源代码）的形式分发，我们将为其提供一个头文件，其中仅包含接口的说明，不包含任何内部数据结构的提示。

由于虚函数在程序执行期间会动态（稍后）绑定到对象，因此可能会出现“纯虚函数调用”致命错误：程序终止。如果程序员无意中“忘记”提供实现，就会发生这种情况。编译器不一定总能在编译时检测到此类遗漏。

3.2.19 运算符重载

在 [表达式](#) 一章中，我们学习了为内置类型定义的各种运算。例如，对于 *double* 类型的变量，我们可以计算以下表达式的值：

```
double a = 2.0, b = 3.0, c = 5.0;
double d = a * b + c;
```

使用类似的语法处理用户定义类型（例如矩阵）会很方便：

```
Matrix a(3, 3), b(3, 3), c(3, 3); // creating 3x3 matrices
// ... somehow fill in a, b, c
Matrix d = a * b + c;
```

MySQL 通过运算符重载提供了这样的功能。

这种技术的组织方式是:通过描述一些方法来实现,其名称以 **operator** 关键字开头,后跟一个受支持的运算符 (或符号序列)。一般来说,这可以表示为如下形式:

```
result_type operator@ ( [type parameter_name] );
```

此处 @ - 运算符。

完整的 MySQL 运算列表已在 [运算优先级](#) 一节中提供,但并非所有运算都允许重载。

以下运算禁止重载:

- 冒号 ':', 上下文权限;
- 括号 '()'、'函数调用'或 '分组';
- 点 '.'、'取消引用';
- 与号 '&','取地址'、一元运算符 (但与号可用作二元运算符 '按位与');
- 条件三元运算符 '?:';
- 逗号 ','。

所有其他运算符均可重载。重载运算符的优先级无法更改,它们保持与标准优先级相同,因此如有必要,应使用括号进行分组。

您不能为标准列表中未包含的某些新字符创建重载。

所有运算符的重载都会考虑其一元性和二元性,即保留所需操作数的数量。与任何类方法一样,运算符重载可以返回某种类型的值。在这种情况下,选择类型本身时应根据规划好的逻辑,也就是在表达式中使用函数所产生的结果 (请参见下文)。

运算符重载方法的格式如下 (用所需运算符的符号代替 '@' 符号):

名称	方法头	表达式中的用法	函数等同于
一元前缀	type operator@()	@object	object.operator@()
一元后缀	type operator@(int)	object@	object.operator@(0)
二元	type operator@(type parameter_name)	object@argument	object.operator@(argument)
索引	type operator[](type index_name)	object[argument]	object.operator[](argument)

一元运算符不接受参数。在一元运算符中,只有递增运算符 '++' 和递减运算符 '--' 除了支持前缀形式外,还支持后缀形式,所有其他一元运算符仅支持前缀形式。指定 **int** 类型的匿名参数即表示后缀形式 (以区别于前缀形式),但参数本身会被忽略。

二元运算符必须接受一个参数。对于同一个运算符,可以存在多个重载的变体,这些变体可带有不同类型的参数,包括与当前对象所属的类相同的类型。在这种情况下,对象作为参数只能按引用或按指针进行传递 ('按指针'仅适用于类对象,不适用于结构体)。

重载运算符既可以通过表达式中的运算语法使用 (这是重载的主要原因),也可以通过方法调用语法使用;这两种方式都显示在上表中。这种功能等效性清晰地表明,从技术角度来说,运

算符不过是对对象的方法调用，对于前缀运算符，对象在右侧，而对于所有其他运算符，对象在左侧。二元运算符方法将作为自变量传递给运算符右侧的值或表达式（特别要指出，这可以是另一个对象或内置类型的变量）。

由此可见，重载运算符不具有交换律属性： $a@b$ 通常不等于 $b@a$ ，因为对于 a ， $@$ 运算符可以重载，但 b 不能。此外，如果 b 是内置类型的变量或值，那么原则上不能重载它的标准行为。

举个例子，试想用于从斐波那契数列生成数字的 *Fibo* 类（我们已经使用函数完成了此任务的一个实现，请参见 [函数定义](#)）。在类中，我们将提供 2 个字段，分别用于存储行的当前编号和上一个编号：*current* 和 *previous*。默认构造函数会将它们的值初始化为 1 和 0。我们还将提供一个拷贝构造函数 (*FiboMonad.mq5*)。

```
class Fibo
{
    int previous;
    int current;
public:
    Fibo() : current(1), previous(0) { }
    Fibo(const Fibo &other) : current(other.current), previous(other.previous) { }
    ...
};
```

对象的初始状态：当前编号为 1，上一个编号为 0。为了查找序列中的下一个编号，我们重载了前缀和后缀递增运算符。

```
Fibo *operator++() // prefix
{
    int temp = current;
    current = current + previous;
    previous = temp;
    return &this;
}

Fibo operator++(int) // postfix
{
    Fibo temp = this;
    ++this;
    return temp;
}
```

请注意，前缀方法在编号修改后不会返回指向当前 *Fibo* 对象的指针，而后缀方法则会返回一个保存了上一个计数器的新对象，这符合后缀递增的原则。

当然，如果需要，程序员可以任意方式重载任何运算。例如，可以计算乘积、将数字输出到日志中，或者在递增的实现中执行其他操作。但还是建议遵循以下原则：使用运算符重载执行直观操作。

我们以类似的方式实现递减运算：递减运算将返回序列的上一个数字。

```

Fibo *operator--() // prefix
{
    int diff = current - previous;
    current = previous;
    previous = diff;
    return &this;
}

Fibo operator--(int) // postfix
{
    Fibo temp = this;
    --this;
    return temp;
}

```

要获取给定数字在序列中的编号，我们需要重载索引访问运算。

```

Fibo *operator[](int index)
{
    current = 1;
    previous = 0;
    for(int i = 0; i < index; ++i)
    {
        ++this;
    }
    return &this;
}

```

要获取 `current` 变量中包含的当前数字，我们需要重载 `~` 运算符（因为它很少使用）。

```

int operator~() const
{
    return current;
}

```

如果没有此重载，您仍然需要实现一些公共方法来读取私有字段 `current`。我们将使用此运算符通过 *Print* 输出数字。

为了方便起见，您也应该重载赋值运算。

```

Fibo *operator=(const Fibo &other)
{
    current = other.current;
    previous = other.previous;
    return &this;
}

Fibo *operator=(const Fibo *other)
{
    current = other.current;
    previous = other.previous;
    return &this;
}

```

我们来看看它是如何工作的。

```

void OnStart()
{
    Fibo f1, f2, f3, f4;
    for(int i = 0; i < 10; ++i, ++f1) // prefix increment
    {
        f4 = f3++; // postfix increment and assignment overloading
    }

    // compare all values obtained by increments and by index [10]
    Print(~f1, " ", ~f2[10], " ", ~f3, " ", ~f4); // 89 89 89 55

    // counting in opposite direction, down to 0
    Fibo f0;
    Fibo f = f0[10]; // copy constructor (due to initialization)
    for(int i = 0; i < 10; ++i)
    {
        // prefix decrement
        Print(~--f); // 55, 34, 21, 13, 8, 5, 3, 2, 1, 1
    }
}

```

结果符合预期。不过，我们需要考虑一个细节。

```

Fibo f5;
Fibo *pf5 = &f5;

f5 = f4; // call Fibo *operator=(const Fibo &other)
f5 = &f4; // call Fibo *operator=(const Fibo *other)
pf5 = &f4; // calls nothing, assigns &f4 to pf5!

```

指针赋值运算符的重载仅在通过对象访问时有效。如果通过指针访问，则采用指针间的标准赋值。

重载运算符的返回类型可以是内置类型、对象类型（类或结构体）或指针（仅限类对象）中的一种。

要返回对象（实例，而非引用），类必须实现拷贝构造函数。这种方式会导致实例重复，从而影响代码效率。如果可能，应该返回一个指针。

但在返回指针时，需要确保返回的不是局部自动对象（该对象在函数退出时会被删除，指针也将失效），而是某个已经存在的对象。通常会返回 `&this`。

通过返回对象或对象的指针，您可以将一个重载运算符的结果“发送”给另一个重载运算符，从而以我们处理内置类型惯用的方式构建复杂的表达式。返回 `void` 将导致无法在表达式中使用该运算符。例如，如果 `'='` 运算符定义为 `void` 类型，则多重赋值将失效：

```
Type x, y, z = 1; // constructors and initialization of variables of a certain class
x = y = z; // assignments, compilation error
```

赋值链从右到左运行，`y = z` 将返回空。

如果对象仅包含内置类型的字段（包括数组），则无需重新定义来自同一类中对象的赋值/拷贝运算符 `'='`。MQL5 默认提供所有字段的“一对一”拷贝。请勿将赋值/拷贝运算符与拷贝构造函数和初始化混淆。

现在，我们来看第二个示例：使用矩阵（`Matrix.mq5`）。

顺便提一下，内置对象类型 **矩阵和向量** 最近已出现在 MQL5 中。使用内置类型还是自定义类型（或者将它们组合使用）由每位开发者自行决定。内置类型中包含许多常用方法的现成快速实现，非常方便，避免了繁琐的编程工作。而自定义类允许您根据任务调整算法。我们在这里提供了 `Matrix` 类作为教程。

在矩阵类中，我们将其元素存储在一维动态数组 `m` 中。在“Sizes”下，选择变量 `rows` 和 `columns`。

```
class Matrix
{
protected:
    double m[];
    int rows;
    int columns;
    void assign(const int r, const int c, const double v)
    {
        m[r * columns + c] = v;
    }

public:
    Matrix(const Matrix &other) : rows(other.rows), columns(other.columns)
    {
        ArrayCopy(m, other.m);
    }

    Matrix(const int r, const int c) : rows(r), columns(c)
    {
        ArrayResize(m, rows * columns);
        ArrayInitialize(m, 0);
    }
}
```

主构造函数接受两个参数（矩阵维度）并为数组分配内存。还有一个拷贝构造函数来自另一个矩阵 `other`。此处和下文中大量使用了用于处理数组的内置函数（具体而言是 `ArrayCopy`、`ArrayResize`、`ArrayInitialize`），我们将在单独章节中讨论这些函数。

我们通过重载赋值运算符来安排从外部数组填充元素的操作：

```

Matrix *operator=(const double &a[])
{
    if(ArraySize(a) == ArraySize(m))
    {
        ArrayCopy(m, a);
    }
    return &this;
}

```

为了实现两个矩阵的加法，我们重载了 ‘+=’ 和 ‘+’ 运算符：

```

Matrix *operator+=(const Matrix &other)
{
    for(int i = 0; i < rows * columns; ++i)
    {
        m[i] += other.m[i];
    }
    return &this;
}

Matrix operator+(const Matrix &other) const
{
    Matrix temp(this);
    return temp += other;
}

```

请注意，‘+=’ 运算符在修改当前对象后返回该对象的指针，而 ‘+’ 运算符则按值返回一个新实例（将使用拷贝构造函数），并且该运算符本身带有 **const** 修饰符，因此不会更改当前对象。

‘+’ 运算符本质上是一个包装器，它先创建当前矩阵的临时副本以备调用（命名为 *temp*），然后将所有工作委派给 ‘+=’ 运算符。因此，通过内部调用 ‘+=’ 运算符（会修改 *temp*）将 *temp* 与 *other* 相加，然后返回 ‘+’ 运算的结果。

矩阵乘法的重载方式类似，使用两个运算符 ‘*=' 和 ‘*’。


```

Matrix *operator*=(const Matrix &other)
{
    // multiplication condition: this.columns == other.rows
    // the result will be a matrix of size this.rows by other.columns
    Matrix temp(rows, other.columns);

    for(int r = 0; r < temp.rows; ++r)
    {
        for(int c = 0; c < temp.columns; ++c)
        {
            double t = 0;
            //we add up the pairwise products of the i-th elements
            // row 'r' of the current matrix and column 'c' of the matrix other
            for(int i = 0; i < columns; ++i)
            {
                t += m[r * columns + i] * other.m[i * other.columns + c];
            }
            temp.assign(r, c, t);
        }
    }
    // copy the result to the current object of the matrix this
    this = temp; // calling an overloaded assignment operator
    return &this;
}

Matrix operator*(const Matrix &other) const
{
    Matrix temp(this);
    return temp *= other;
}

```

下面，我们将矩阵乘以一个数字：

```

Matrix *operator*=(const double v)
{
    for(int i = 0; i < ArraySize(m); ++i)
    {
        m[i] *= v;
    }
    return &this;
}

Matrix operator*(const double v) const
{
    Matrix temp(this);
    return temp *= v;
}

```

为了比较两个矩阵，我们使用运算符 '==' 和 '!='：

```

bool operator==(const Matrix &other) const
{
    return ArrayCompare(m, other.m) == 0;
}

bool operator!=(const Matrix &other) const
{
    return !(this == other);
}

```

为了方便调试，我们将矩阵数组的输出写入日志。

```

void print() const
{
    ArrayPrint(m);
}

```

除了上述重载之外，*Matrix* 类还提供了运算符 `[]` 的重载：此运算符可返回嵌套类 *MatrixRow* 的对象，即具有给定编号的行。

```

MatrixRow operator[](int r)
{
    return MatrixRow(this, r);
}

```

MatrixRow 类本身通过重载相同的运算符 `[]` 提供了对矩阵元素的更深层访问（也就是说，对于矩阵，可以自然地指定两个索引 `m[i][j]`）。

```

class MatrixRow
{
protected:
    const Matrix *owner;
    const int row;

public:
    class MatrixElement
    {
protected:
        const MatrixRow *row;
        const int column;

public:
        MatrixElement(const MatrixRow &mr, const int c) : row(&mr), column(c) { }
        MatrixElement(const MatrixElement &other) : row(other.row), column(other.col

double operator~() const
{
    return row.owner.m[row.row * row.owner.columns + column];
}

double operator=(const double v)
{
    row.owner.m[row.row * row.owner.columns + column] = v;
    return v;
}
};

MatrixRow(const Matrix &m, const int r) : owner(&m), row(r) { }
MatrixRow(const MatrixRow &other) : owner(other.owner), row(other.row) { }

MatrixElement operator[](int c)
{
    return MatrixElement(this, c);
}

double operator[](uint c)
{
    return owner.m[row * owner.columns + c];
}
};

```

对 `int` 类型的参数，`[]` 运算符会返回 `MatrixElement` 类的对象，您可以通过该对象在数组中写入特定元素。要读取元素，需要将 `[]` 运算符与 `uint` 类型参数一起使用。这看起来像是一个技巧，但实际上是语言方面的局限性：重载的参数类型必须不同。作为读取元素的替代方法，`MatrixElement` 类提供了运算符 `~` 的重载。

使用矩阵时，通常需要一个单位矩阵，因此我们来为其创建一个派生类：

```

class MatrixIdentity : public Matrix
{
public:
    MatrixIdentity(const int n) : Matrix(n, n)
    {
        for(int i = 0; i < n; ++i)
        {
            m[i * rows + i] = 1;
        }
    }
};

```

现在，我们开始实际使用矩阵表达式。

```

void OnStart()
{
    Matrix m(2, 3), n(3, 2); // description
    MatrixIdentity p(2);      // identity matrix

    double ma[] = {-1,  0, -3,
                   4, -5,  6};
    double na[] = {7,  8,
                   9,  1,
                   2,  3};
    m = ma; // filling in data
    n = na;

    //we can read and write elements separately
    m[0][0] = m[0][(uint)0] + 2; // variant 1
    m[0][1] = ~m[0][1] + 2;      // variant 2

    Matrix r = m * n + p;          // expression
    Matrix r2 = m.operator*(n).operator+(p); // equivalent
    Print(r == r2); // true

    m.print(); // 1.00000  2.00000 -3.00000  4.00000 -5.00000  6.00000
    n.print(); // 7.00000  8.00000  9.00000  1.00000  2.00000  3.00000
    r.print(); // 20.00000  1.00000 -5.00000  46.00000
}

```

这里，我们分别创建了两个矩阵：3*2 维和 2*3 维，然后用数组中的值填充这两个矩阵，并使用两个索引 `[][]` 的语法编辑选定元素。最后，我们计算了表达式 $m * n + p$ ，其中所有操作数都是矩阵。下面一行以方法调用的形式显示了相同的表达式。我们得到了相同的结果。

与 C++ 不同，MQL5 不支持全局级别的运算符重载。在 MQL5 中，运算符只能在类或结构体的上下文中重载，也就是说，通过类或结构体的方法进行重载。另外，MQL5 不支持重载类型转换运算符 *new* 和 *delete*。

3.2.20 对象类型转换: `dynamic_cast` 和 `pointer void *`

对象类型具有特定的转换规则, 这些规则在源和目标变量类型不匹配时适用。内置类型的规则已经在第 2.6 章 [类型转换](#) 中讨论过。复制时结构体类型转换的细节在 [结构体布局和继承](#) 一节中进行介绍。

对于结构体和类, 是否允许类型转换主要取决于它们是否应在继承链上存在关联。继承体系中属于不同分支的类型或根本不相关的类型不能相互转换。

对象 (值) 和指针的类型转换规则不同。

对象

如果类型 B 拥有一个接受类型 A 参数的构造函数 (有不同变体, 比如按值、按引用或按指针, 但通常形式为 `B(const A &a)`), 则类型 A 的对象可以赋值给类型 B 的对象。这样的构造函数也称为转换构造函数。

在没有显式转换构造函数的情况下, 编译器会尝试使用隐式拷贝赋值运算符, 即 `B::operator=(const B &b)`。要使这种隐式转换 (从 A 到 B) 有效, 类 A 和 B 必须在同一继承链中。如果 A 继承自 B (包括间接继承), 则在拷贝到 B 时, A 中新增的属性将会消失。如果 B 继承自 A, 则只会将 A 中存在的属性部分拷贝到 B。通常不推荐这种隐式转换。

此外, 编译器也不一定提供隐式拷贝运算符。特别是, 如果类具有带 `const` 修饰符的字段, 则认为禁止拷贝 (详见后文)。

在 `ShapesCasting.mq5` 脚本中, 我们使用形状类层次结构来演示对象类型转换。在 `Shape` 类中, `type` 字段被特意设为常量, 因此尝试将 `Square` 对象转换 (赋值) 为 `Rectangle` 对象会导致编译器错误, 并给出详细解释:

```
attempting to reference deleted function 'void Rectangle::operator=(const Rectangle&)'
function 'void Rectangle::operator=(const Rectangle&)' was implicitly deleted
because it invokes deleted function 'void Shape::operator=(const Shape&)'
function 'void Shape::operator=(const Shape&)' was implicitly deleted
because member 'type' has 'const' modifier
```

根据此消息可以看出, `Rectangle::operator=(const Rectangle&)` 拷贝方法已被编译器隐式移除 (编译器提供了默认实现), 因为它在 `Shape::operator=(const Shape&)` 基类中使用了类似的方法, 而后者又因存在带有 `const` 修饰符的 `type` 字段而被移除。此类字段只能在创建对象时设置, 而编译器不知道如何在这种限制下拷贝对象。

顺便说一句, “删除”方法的效果不仅对编译器有效, 对程序员也有效: 更多相关内容将在 [继承控制: final 和 delete](#) 一节中讨论。

可通过以下方法解决此问题: 移除 `const` 修饰符, 或自行提供各自的赋值运算符实现方式 (其中不涉及 `const` 字段, 并将内容与 “Rectangle” 类型的说明一起保存):

```

Rectangle *operator=(const Rectangle &r)
{
    coordinates.x = r.coordinates.x;
    coordinates.y = r.coordinates.y;
    backgroundColor = r.backgroundColor;
    dx = r.dx;
    dy = r.dy;
    return &this;
}

```

请注意，此定义返回指向当前对象的指针，而编译器生成的默认实现为 `void` 类型（如错误消息中所示）。这意味着，编译器提供的默认赋值运算符不能在 `x = y = z` 链中使用。如果需要此功能，请显式重写 `operator=` 并返回所需的非 `void` 类型。

指针

最大的作用是将指针转换为不同类型的对象。

理论上，用于转换对象类型指针的所有选项可以简化为三种：

- 从基类到派生类的转换，这属于向下类型转换（向下转换），因为通常用倒置树来绘制类层次结构；
- 从派生类到基类的转换，这属于向上类型转换（向上转换）；
- 在层次结构不同分支的类之间的转换，甚至不同家族的类之间的转换。

最后一种选项被禁止（我们会收到编译错误）。编译器允许前两种，但如果“向上转换”是自然且安全的，那么“向下转换”可能会导致运行时错误。

```

void OnStart()
{
    Rectangle *r = addRandomShape(Shape::SHAPES::RECTANGLE);
    Square *s = addRandomShape(Shape::SHAPES::SQUARE);
    Circle *c = NULL;
    Shape *p;
    Rectangle *r2;

    // OK
    p = c;    // Circle -> Shape
    p = s;    // Square -> Shape
    p = r;    // Rectangle -> Shape
    r2 = p;   // Shape -> Rectangle
    ...
};

```

当然，当使用指向基类对象的指针时，即使相应的对象位于指针处，也不能对其调用派生类的方法和属性。否则我们会遇到编译错误“未声明的标识符”。

但是，指针支持 [显式强制转换](#) 语法（参见 C 风格），该语法允许在表达式中将指针“动态”转换为所需类型，并对其进行取消引用，而无需创建中间变量。

```
Base *b;
Derived d;
b = &d;
((Derived *)b).derivedMethod();
```

这里，我们创建了一个派生类对象 (*Derived*) 和一个指向它的基类指针 (*Base **)。若要访问派生类的 *derivedMethod* 方法，需要将指针临时转换为 *Derived* 类型。

星号指针类型必须用圆括号括起来。此外，强制转换表达式本身 (包括变量名) 也必须用另一对圆括号括起来。

在我们的测试中，还有另一个编译错误，提示“类型不匹配”-“类型不匹配”。这个错误出现在我们试图将一个 *Rectangle* 类型的指针强制转换成 *Circle* 类型的指针的那行代码上，因为这两个类处于不同的继承分支中。

```
c = r; // error: type mismatch
```

当强制类型转换的指针与实际对象不匹配时，情况会更加糟糕 (尽管它们的类型兼容，因此程序可以正常编译)。这样的运算会在程序执行阶段就以错误结束 (也就是说，编译器无法捕获错误)。程序随后会被卸载。

例如，在 *ShapesCasting.mq5* 脚本中，我们描述了一个指向 *Square* 的指针，并为其赋予了一个指向 *Shape* 的指针，而 *Shape* 包含 *Rectangle* 对象。

```
Square *s2;
// RUNTIME ERROR
s2 = p; // error: Incorrect casting of pointers
```

终端返回错误“指针强制转换不正确”。更具体的 *Square* 类型的指针无法指向父对象 *Rectangle*。

为了避免运行时问题并防止程序崩溃，MQL5 提供了一个特殊的语言结构 *dynamic_cast*。使用此结构，您可以“仔细”检查是否可以将指针转换为所需的类型。如果可以转换，则编译器会进行转换。如果不可转换，我们将得到一个空指针 (NULL)，我们可以用特殊方式处理它 (例如，使用 *if* 以某种方式初始化或中断函数的执行，而不是整个程序)。

dynamic_cast 的语法如下：

```
dynamic_cast< Class * >( pointer )
```

在我们的例子中，只需编写：

```
s2 = dynamic_cast<Square *>(p); // trying to cast type, and will get NULL if unsuc
Print(s2); // 0
```

程序将按预期运行。

具体来说，我们可以再次尝试将矩形转换为圆形，并确保结果为 0：

```
c = dynamic_cast<Circle *>(r); // trying to cast type, and will get NULL if unsucc
Print(c); // 0
```

MQL5 中有一个特殊的指针类型，可以存储任何对象。此类型的表示法为：*void **。

我们来演示一下变量 *void ** 如何与 *dynamic_cast* 配合使用。

```

void *v;
v = s; // set to the instance Square
PRT(dynamic_cast<Shape *>(v));
PRT(dynamic_cast<Rectangle *>(v));
PRT(dynamic_cast<Square *>(v));
PRT(dynamic_cast<Circle *>(v));
PRT(dynamic_cast<Triangle *>(v));

```

前三行将记录指针（同一对象的描述符）的值，后两行将输出 0。

现在，回到 **指针** 一节中的前向声明示例（参见文件 *ThisCallback.mq5*），其中 *Manager* 和 *Element* 类包含相互的指针。

使用指针类型 **void ***，您可以删除初始声明（*ThisCallbackVoid.mq5*）。我们注释掉包含它的那一行，并将包含对象管理器指针的 *owner* 字段类型更改为 **void ***。在构造函数中，我们也会更改参数的类型。

```

// class Manager;
class Element
{
    void *owner; // looking forward to being compatible with the Manager type *
public:
    Element(void *t = NULL): owner(t) { } // was Element(Manager &t)
    void doMath()
    {
        const int N = 1000000;

        // get the desired type at runtime
        Manager *ptr = dynamic_cast<Manager *>(owner);
        // then everywhere you need to check ptr for NULL before using

        for(int i = 0; i < N; ++i)
        {
            if(i % (N / 20) == 0)
            {
                if(ptr != NULL) ptr.progressNotify(&this, i * 100.0f / N);
            }
            // ... lots of calculations
        }
        if(ptr != NULL) ptr.progressNotify(&this, 100.0f);
    }
    ...
};

```

这种方法可以提供更大的灵活性，但需要更加小心，因为 *dynamic_cast* 可能会返回 **NULL**。建议尽可能使用语言自身提供的类型控制的标准调度机制（静态和动态）。

通常，仅在特殊情况下才有必要使用 **void *** 指针。而带有初步说明的“额外”行不属于特殊情况。它在这里仅被用作 **void *** 指针通用性的最简单示例。

3.2.21 指针、引用和常量

在了解了内置类型和对象类型以及 [引用](#) 和 [指针](#) 的概念之后，比较一下所有可用的类型修改可能很有意义。

MQL5 中的引用仅用于描述函数和方法的参数。此外，对象类型参数必须通过引用传递。

```
void function(ClassOrStruct &object) { }           // OK
void function(ClassOrStruct object) { }           // wrong
void function(double &value) { }                 // OK
void function(double value) { }                  // OK
```

这里的 *ClassOrStruct* 是类或结构体的名称。

只允许作为引用类型参数的自变量来传递变量（左侧值），不允许传递常量或表达式计算产生的临时值。

您无法创建引用类型的变量，也无法从函数返回引用。

```
ClassOrStruct &function(void) { return Class(); } // wrong
ClassOrStruct &object;                          // wrong
double &value;                                    // wrong
```

MQL5 中的指针仅适用于类对象。不支持指向内置类型变量或结构体变量的指针。

您可以将变量或函数参数声明为指向对象的指针，也可以从函数返回指向对象的指针。

```
ClassOrStruct *pointer;                          // OK
void function(ClassOrStruct *object) { }         // OK
ClassOrStruct *function() { return new ClassOrStruct(); } // OK
```

但您不能返回指向局部自动对象的指针，因为后者在函数退出时会被释放，并且指针将失效。

如果函数返回一个指向在函数内部使用 `new` 动态分配的对象的指针，则调用代码必须记住使用 `delete` 释放指针。

指针可以为 `NULL`，这一点与引用不同。指针参数可以具有默认值，但引用不能（“无法初始化引用”错误）。

```
void function(ClassOrStruct *object = NULL) { } // OK
void function(ClassOrStruct &object = NULL) { } // wrong
```

链接和指针可以在参数说明中组合使用。因此，函数可以接受指向指针的引用：然后函数中对该指针的更改将在调用代码中生效。特别是，负责创建对象的工厂函数能以这种方式实现。

```
void createObject(className *&ref)
{
    ref = new className();
    // further customization of ref
    ...
}
```

确实，要从函数返回单个指针，通常习惯使用 `return` 语句，因此这个例子有些刻意。但是，在需要将指针数组传递到外部的情况下，在参数中引用数组是首选方案。例如，在一些用于处理带有 [键, 值] 对的映射类型容器类的标准库类 `MQL5/Include/Generic/SortedMap.mqh`、

`MQL5/Include/Generic/HashMap.mqh`)中,有一些 `CopyTo` 方法可用于获取具有 `CKeyValuePair` 元素的数组。

```
int CopyTo(CKeyValuePair<TKey,TValue> *&dst_array[], const int dst_start = 0);
```

`dst_array` 参数类型可能看起来不太熟悉:它是一个类模板。我们将在 下一章学习模板。目前,我们只关心这是一个对指针数组的引用。

`const` 修饰符对所有类型施加特殊行为。关于内置类型,我们已在 常量变量一节中进行了讨论。对象类型有其自身的特性。

如果变量或函数参数被声明为指针或对象的引用(引用仅在参数中有效),则其中的 `const` 修饰符会将可访问的方法和属性集限制为同样带有 `const` 修饰符的对象。换句话说,只有常量属性才能通过常量引用和指针访问。

当您尝试调用非常量方法或更改非常量字段时,编译器将生成错误:“对常量对象调用非常量方法”或“无法修改常量”。

非常量指针参数可以接受任何自变量(常量或非常量)。

需要注意的是,在指针说明中可以设置两个修饰符 `const`:一个引用对象,另一个引用指针:

- `Class *pointer` 指向对象的指针;对象和指针正常运行,不受任何限制;
- `const Class *pointer` 指向常量对象的指针;对于对象,只能使用常量方法和读取属性,但指针可以更改(为其赋予另一个对象的地址);
- `const Class * const pointer` 指向常量对象的常量指针;对于对象,只能使用常量方法和读取属性;指针不能更改;
- `Class * const pointer` 指向对象的常量指针;指针不能更改,但对象的属性可以更改。

以下面的 `Counter` 类(`CounterConstPtr.mq5`)为例。

```
class Counter
{
public:
    int counter;

    Counter(const int n = 0) : counter(n) { }

    void increment()
    {
        ++counter;
    }

    Counter *clone() const
    {
        return new Counter(counter);
    }
};
```

它人为地创建了公共变量 `counter`。该类还包含两个方法,其中一个是常量(`clone`),另一个不是(`increment`)。回想一下,常量方法无权更改对象的字段。

以下带有 `Counter *ptr` 类型参数的函数可以调用该类的所有方法并更改其字段。

```

void functionVolatile(Counter *ptr)
{
    // OK: everything is available
    ptr.increment();
    ptr.counter += 2;
    //remove the clone immediately so that there is no memory leak
    // the clone is only needed to demonstrate calling a constant method
    delete ptr.clone();
    ptr = NULL;
}

```

以下带有 `const Counter *ptr` 参数的函数将抛出几个错误。

```

void functionConst(const Counter *ptr)
{
    // ERRORS:
    ptr.increment(); // calling non-const method for constant object
    ptr.counter = 1; // constant cannot be modified

    // OK: only const methods are available, fields can be read
    Print(ptr.counter); // reading a const object
    Counter *clone = ptr.clone(); // calling a const method
    ptr = clone; // changing a non-const pointer ptr
    delete ptr; // cleaning memory
}

```

最后，以下带有 `const Counter * const ptr` 参数的函数执行的操作更少。

```

void functionConstConst(const Counter * const ptr)
{
    // OK: only const methods are available, the pointer ptr cannot be changed
    Print(ptr.counter); // reading a const object
    delete ptr.clone(); // calling a const method

    Counter local(0);
    // ERRORS:
    ptr.increment(); // calling non-const method for constant object
    ptr.counter = 1; // constant cannot be modified
    ptr = &local; // constant cannot be modified
}

```

在 `OnStart` 函数中，我们声明了两个 `Counter` 对象（一个是常量，另一个不是），您可以调用这些函数，但有一些例外：

```

void OnStart()
{
    Counter counter;
    const Counter constCounter;

    counter.increment();

    // ERROR:
    // constCounter.increment(); // call non-const method for constant object
    Counter *ptr = (Counter *)&constCounter; // trick: type casting without const
    ptr.increment();

    functionVolatile(&counter);

    // ERROR: cannot convert from a const pointer...
    // functionVolatile(&constCounter); // to a non-const pointer

    functionVolatile((Counter *)&constCounter); // type casting without const

    functionConst(&counter);
    functionConst(&constCounter);

    functionConstConst(&counter);
    functionConstConst(&constCounter);
}

```

首先，请注意，在非常量对象上尝试调用常量方法 *increment* 时，变量也会产生错误。

其次，*constCounter* 不能传递给 *functionVolatile* 函数，否则我们会收到错误“无法从常量指针转换为非常量指针”。

不过，这两个错误可以通过不带 *const* 修饰符的显式类型转换来避免。尽管我们不建议这样做。

3.2.22 继承管理：final 和 delete

MySQL 允许您对类和结构体的继承施加一些限制。

关键字 *final*

通过在类名后添加 *final* 关键字，开发者可以禁用该类的继承。例如 (*FinalDelete.mq5*):

```

class Base
{
};

class Derived final : public Base
{
};

class Concrete : public Derived // ERROR
{
};

```

编译器将抛出错误“无法从‘Derived’继承，因为它已被声明为‘final’”。

遗憾的是，对于使用此类限制的好处和场景，目前尚无共识。该关键字让类的使用者知道，由于某种原因，类的作者不建议将其作为基类（例如，其当前实现为草案，以后会发生很大变化，这可能会导致潜在的遗留项目停止编译）。

有些人试图鼓励以这种方式设计程序，即使用对象（组合）来代替继承。过度使用继承反而会增加类的耦合性（即相互影响），因为所有的子类都可以在某种程度上改变父类的数据或方法（特别是通过重定义虚函数）。因此，程序工作逻辑的复杂性和出现不可预见副作用的可能性都会增加。

使用 *final* 的另一个好处是编译器可以进行代码优化：对于“final”类型的指针，它可以用静态调度替换虚函数的动态调度。

关键字 *delete*

可以在方法头中指定 *delete* 关键字，使其在当前类及其后代类中不可访问。父类的虚方法不能以这种方式删除（这会违反类的“契约”，即继承者将不再“是（is a）”同一类型的代表）。

```

class Base
{
public:
    void method() { Print(__FUNCSIG__); }
};

class Derived : public Base
{
public:
    void method() = delete;
};

void OnStart()
{
    Base *b;
    Derived d;

    b = &d;
    b.method();

    // ERROR:
    // attempting to reference deleted function 'void Derived::method()'
    // function 'void Derived::method()' was explicitly deleted
    d.method();
}

```

尝试调用它将导致编译错误。

我们在 [对象类型转换](#) 章节中看到了类似的错误，因为编译器具有一定的智慧，可以在特定条件下“移除”方法。

建议将编译器提供隐式实现的以下方法标记为已删除：

- 默认构造函数：***Class(void) = delete;***
- 拷贝构造函数：***Class(const Class &object) = delete;***
- 拷贝/赋值运算符：***void operator=(const Class &object) = delete。***

如果您需要其中任何一个，则必须显式定义。否则，一般认为放弃隐式实现是一种良好习惯。问题是，隐式实现非常简单，并且可能会引发难以定位的问题，尤其是在强制转换对象类型时。

3.3 模板

现代编程语言中有许多内置功能，可以帮助您避免代码重复，从而最大限度地减少错误并提高程序员的工作效率。在 MQL5 中，这些工具包括已知的 [函数](#)、支持继承的对象类型 ([类](#) 和 [结构体](#))、[预处理器宏](#) 以及 [包含文件](#)。但如果没有模板，这份清单就不完整。

模板是对函数或对象类型的特制泛型定义，编译器可以根据该定义自动生成该函数或对象类型的工作实例。生成的实例包含相同的算法，但操作不同类型的变量，这与源代码中使用模板的特定条件相对应。

对于 C++ 爱好者，我们注意到 MQL5 模板不支持 C++ 模板的许多功能，特别是：

- ① 非类型参数；

- ④具有默认值的参数；
- ④可变数量的参数；
- ④类、结构体和关联的特化（完全特化和偏特化）；
- ④模板的模板。

一方面，这降低了 MQL5 中模板的潜力，但另一方面，它简化了不熟悉这些技术的人员的材料学习。

3.3.1 模板头部

在 MQL5 中，您可以将函数、对象类型（类、结构体、联合体）或其中的单独方法模板化。无论如何，模板说明都有一个标题：

```
template <typename T [, typename Ti ... ]>
```

头部以 **template** 关键字开头，后面是用逗号分隔的形式参数列表，这些参数都放在尖括号中：每个参数都用 **typename** 关键字和标识符表示。标识符在特定定义中必须是唯一的。

模板头部中的 **typename** 关键字会告知编译器，下面的标识符应被视为一种类型。将来，MQL5 编译器可能会像 C++ 编译器一样，支持其他类型的非类型参数。

请勿将 **typename** 的这种用法不应与内置的 **运算符typename** 混淆，后者返回的字符串包含所传递参数的类型名称。

模板头部之后是函数（方法）或类（结构体、联合体）的常规定义，在常规定义中，在语法要求类型名称的位置，模板的形式参数（标识符 T、Ti）被用于指令和表达式。例如，对于模板函数，模板参数描述函数参数或返回值的类型，而在模板类中，模板参数可以指定字段类型。

模板是一个完整的定义。模板的结尾是实体（函数、方法、类、结构体、联合体）的定义，前面加上 **template** 标题。

对于模板参数名，习惯上使用大写的单字符或双字符标识符。

参数的最小数量为 1，最大数量为 64。

参数的主要用例（以 T 参数为例）包括：

- 类型，在描述字段、函数/方法中的局部变量、其参数和返回值时使用 **T variable_name; T function(T parameter_name);**；
- 完全限定类型名称的组成部分之一，具体而言是：**T::SubType**、**T.StaticMember**；
- 使用修饰符构造新类型：**const T**、指针 **T***、引用 **T&**、数组 **T[]**、**typedef** 函数 **T(*func)(T)**；
- 构造新模板类型：**T<Type>**、**Type<T>**，包括从模板继承时（请参见 [不存在的模板特化](#) 一节）；
- 类型转换 (**T**)，能够添加修饰符，并通过 **new T()** 创建对象；
- **sizeof(T)** 作为 MQL 模板中不存在的值参数的原始替代（截止本书撰写时有效）。

3.3.2 模板操作通用原则

我们来回想 [函数重载](#)。它包括用不同的参数定义一个函数的多个版本，包括参数数量相同但类型不同的情况。通常，对于不同类型的参数，此类函数的算法是相同的。例如，MQL5 有一个内置函数 **MathMax**，可返回传递给它的两个值中较大的一个：

```
double MathMax(double value1, double value2);
```

虽然只为 `double` 类型提供了原型，但该函数实际上能够处理其他数字类型的参数对，如 `int` 或 `datetime`。换句话说，该函数是内置数值类型的重载内核。如果我们在源代码中实现同样的效果，就必须重载函数，方法是复制函数并搭配不同的参数，就像这样：

```
double Max(double value1, double value2)
{
    return value1 > value2 ? value1 : value2;
}

int Max(int value1, int value2)
{
    return value1 > value2 ? value1 : value2;
}

datetime Max(datetime value1, datetime value2)
{
    return value1 > value2 ? value1 : value2;
}
```

所有实现（函数体）都是一样的。改变的只是参数类型。

这时，模板就派上用场了。通过使用模板，我们可以描述一个具有所需实现的算法样本，而编译器本身会根据程序中涉及的特定类型生成多个实例。模板的生成是在编译过程中即时进行的，程序员通常不会察觉到这个过程（除非模板代码本身有错误）。自动生成的源代码不会插入程序文本，而是直接转换成二进制代码（`ex5` 文件）。

在模板中，一个或多个参数是类型的形式化名称，在编译阶段，将根据特殊的类型推断规则，从内置类型或用户定义类型中选择实际类型。例如，`Max` 函数可以使用以下带有 `T` 类型参数的模板来描述：

```
template<typename T>
T Max(T value1, T value2)
{
    return value1 > value2 ? value1 : value2;
}
```

然后将其应用于各种类型的变量（参见 `TemplatesMax.mq5`）：

```
void OnStart()
{
    double d1 = 0, d2 = 1;
    datetime t1 = D'2020.01.01', t2 = D'2021.10.10';
    Print(Max(d1, d2));
    Print(Max(t1, t2));
    ...
}
```

在这种情况下，编译器会自动为 `double` 和 `datetime` 类型生成 `Max` 函数的变体。

模板本身不会生成源代码。为此，您需要以某种方式创建模板的实例：调用模板函数或提及具有特定类型的模板类名称，以创建对象或派生类。

在此之前，编译器将忽略整个模式。例如，我们可以编写下面这个所谓的模板函数，它实际上包含了语法错误的代码。但只要不在任何地方调用该函数，编译包含该函数的模块就会成功。

```
template<typename T>
void function()
{
    it's not a comment, but it's not source code either
    !%^&*
}
```

每次使用模板时，编译器都会确定与模板形式参数相匹配的实际类型。根据这些信息，将为每个独特的参数组合自动生成模板源代码。这就是实例。

因此，在我们给出的 *Max* 函数示例中，我们调用了两次模板函数：一次是针对一对 *double* 类型的变量，另一次是针对一对 *datetime* 类型的变量。这就产生了两个 *Max* 函数实例，其源代码分别用于匹配 *T=double* 和 *T=datetime*。当然，如果在代码的其他部分调用了相同类型的模板，也不会生成新的实例。只有当模板应用于另一种类型（如果有不止 1 个参数，则为类型集）时，才需要一个新的模板实例。

请注意，*Max* 模板只有一个参数，它同时为函数的两个输入参数及其返回值设置了类型。换句话说，模板声明可以对有效自变量的类型施加某些限制。

如果我们对不同类型的变量调用 *Max*，编译器将无法确定实例化模板的类型，并会抛出错误“模棱两可的模板参数，必须是 'double' 或 'datetime'”：

```
Print(Max(d1, t1)); // template parameter ambiguous,
                    // could be 'double' or 'datetime'
```

这种根据模板使用上下文来发现模板参数实际类型的过程称为类型推断。在 MQL5 中，类型推断仅适用于函数和方法模板。

而对于类、结构体和联合体，则使用另一种方式将类型绑定到模板参数：在创建模板实例时，在尖括号中显式指定所需的类型（如果有多个参数，则相应的类型数量将以逗号分隔列表形式指示）。有关更多信息，请参见 [对象类型模板](#)。

同样的显式方法可以应用于函数，作为自动类型推断的替代方法。

例如，我们可以为 *ulong* 类型生成并调用 *Max* 实例：

```
Print(Max<ulong>(1000, 100000000));
```

在这种情况下，如果没有明确指示，模板函数将与 *int* 类型相关联（基于整数常量的值）。

3.3.3 模板与预处理器宏

有时可能会出现一个问题：是否可以使用宏替换来生成代码？这实际上是可以的。例如，*Max* 函数集可以很容易地表示为宏：

```
#define MAX(V1,V2) ((V1) > (V2) ? (V1) : (V2))
```

然而，宏的功能比较有限（仅限于文本替换），因此它们仅用于简单情况（例如上述情况）。

在比较宏和模板时，应注意以下区别。

宏在编译开始之前，会被预处理器在源文本中“扩展”并替换。同时，没有关于参数类型以及宏内容替换上下文的信息。特别是，宏 *MAX* 无法检查参数 *V1* 和 *V2* 的类型是否相同，以及是否已

为它们定义比较运算符 ‘>’。此外，如果在程序文本中遇到名为 MAX 的变量，预处理器会尝试用 MAX 宏的 “调用” 来替换它，但会因为缺少参数而报错。更糟糕的是，这些替换会忽略 MAX 标记所在的命名空间或类——基本上，任何命名空间或类都可以。

与宏不同，模板由编译器根据特定的参数类型及其使用位置进行处理，因此它们可以对模板中的所有表达式进行类型兼容性（和普遍适用性）检查以及上下文绑定。例如，我们可以在具体类中定义方法模板。

如果需要，可用不同方式定义同名模板，以用于不同类型，而给定名称的宏始终会被相同的 “实现” 替换。例如，对于像 MAX 这样的函数，我们可以为字符串定义不区分大小写的比较。

宏自身问题导致的编译错误很难诊断，特别是当宏包含多行代码时，因为高亮显示的错误行是宏的 “调用” 位置，而不是预处理器展开后的文本，后者才是传递给编译器的内容。

同时，模板是源代码中以现成形式存在的元素，因为它们会进入编译器，因此其中的任何错误都会在行中具有特定的行号和位置。

宏可能会产生副作用，这一点我们已在 [伪函数形式的 #define](#) 一节讨论过：如果 MAX 宏自变量是带有增量/减量的表达式，则它们将被执行两次。

然而，宏也有一些优势。宏能够生成任何文本，而不仅仅是正确的语言结构。例如，只需要几个宏，您便可以模拟用于字符串的 *switch* 指令（尽管不推荐这种方法）。

具体而言，在标准库中，宏用于组织图表上事件的处理（参见 *MQL5/Include/Controls/Defines.mqh*: `EVENT_MAP_BEGIN`、`EVENT_MAP_END`、`ON_EVENT` 等）。它不适用于模板，而且使用宏来构建事件映射显然不是唯一的选择，也不是最便捷的调试方案。在宏中，逐步（逐行）调试代码执行非常困难。相反，模板完全支持调试。

3.3.4 模板中的内置类型和对象类型的特性

切记，模板中类型的适用性受到三个重要方面的限制：

- 类型是内置类型还是用户定义类型（用户定义类型要求参数通过引用传递，而内置类型不允许字面量通过引用传递）；
- 对象类型是否为类（只有类支持指针）；
- 模板算法中对相应类型的数据执行的一组操作。

假设我们有一个 `Dummy` 结构（参见 *TemplatesMax.mq5* 脚本）：

```
struct Dummy
{
    int x;
};
```

如果我们尝试对该结构的两个实例调用 `Max` 函数，我们将收到一堆错误消息，主要内容如下：“对象只能通过引用传递”和“您无法应用模板”。

```
// ERRORS:
// 'object1' - objects are passed by reference only
// 'Max' - cannot apply template
Dummy object1, object2;
Max(object1, object2);
```

问题的核心在于按值传递模板函数参数，而这种方法与任何对象类型都不兼容。要解决这个问题，您可以将参数类型更改为链接：

```
template<typename T>
T Max(T &value1, T &value2)
{
    return value1 > value2 ? value1 : value2;
}
```

旧错误会消失，但随后会出现新错误：“>-使用了非法操作”（“>-使用了非法操作”）。关键在于，Max 模板中有一个带有 ‘>’ 比较运算符的表达式。因此，如果将自定义类型替换到模板中，则必须在模板中重载 ‘>’ 运算符（而结构体 *Dummy* 没有该运算符；我们稍后会讲到）。对于更复杂的函数，您可能需要重载大量的运算符。幸好，编译器会准确地告诉您具体缺少哪些。

然而，如果将函数传递参数的方式从引用改为其他方式，则会导致之前的调用无法正常工作：

```
Print(Max<ulong>(1000, 10000000));
```

现在会生成如下错误：“参数作为引用传递，应为变量”。因此，我们的函数模板无法处理字面量和其他临时值（特别是，无法直接向其传递表达式或另一个函数的调用结果）。

有人可能会认为，解决这个问题的通用方法是模板函数重载，即定义两个选项，它们之间的唯一区别在于参数是否带有和号：

```
template<typename T>
T Max(T &value1, T &value2)
{
    return value1 > value2 ? value1 : value2;
}

template<typename T>
T Max(T value1, T value2)
{
    return value1 > value2 ? value1 : value2;
}
```

但这行不通。现在编译器会抛出错误“具有相同参数的函数重载存在歧义”：

```
'Max' - ambiguous call to overloaded function with the same parameters
could be one of 2 function(s)
    T Max(T&,T&)
    T Max(T,T)
```

最终有效的重载要求在链接中添加修饰符 *const*。在此过程中，我们将 *Print* 运算符添加到 *Max* 模板，以便我们可以在日志中看到正在调用哪个重载以及 T 对应的参数类型。

```

template<typename T>
T Max(const T &value1, const T &value2)
{
    Print(__FUNCSIG__, " T=", typename(T));
    return value1 > value2 ? value1 : value2;
}

template<typename T>
T Max(T value1, T value2)
{
    Print(__FUNCSIG__, " T=", typename(T));
    return value1 > value2 ? value1 : value2;
}

struct Dummy
{
    int x;
    bool operator>(const Dummy &other) const
    {
        return x > other.x;
    }
};

```

我们还在 `Dummy` 结构体中实现了运算符 `>` 的重载。因此，测试脚本中所有 `Max` 函数调用均已成功完成：包括内置类型和用户定义类型，以及字面量和变量。日志中的输出如下：

```

double Max<double>(double,double) T=double
1.0
datetime Max<datetime>(datetime,datetime) T=datetime
2021.10.10 00:00:00
ulong OnStart::Max<ulong>(ulong,ulong) T=ulong
10000000
Dummy Max<Dummy>(const Dummy&,const Dummy&) T=Dummy

```

细心的读者会注意到，我们现在有两个完全相同的函数，它们的区别仅在于参数传递方式（按值和按引用），而这在这种情况下，恰好可以使用模板。如果函数体不像我们这样简单，这种重复调用可能会代价高昂。这可以通过常用方法解决：将实现分离成一个单独函数，并从两个“重载”中调用此函数，或者从一个“重载”中调用另一个（需要一个可选参数，以避免第一个版本的 `Max` 调用自身并导致堆栈溢出）：

```

template<typename T>
T Max(T value1, T value2)
{
    // calling a function with parameters by reference
    return Max(value1, value2, true);
}

template<typename T>
T Max(const T &value1, const T &value2, const bool ref = false)
{
    return (T)(value1 > value2 ? value1 : value2);
}

```

我们还需要考虑的另一个问题与用户定义的类型有关，即在模板中使用指针（回想一下，指针仅适用于类对象）。我们创建一个简单类 *Data*，并尝试调用模板函数 *Max* 来获取指向其对象的指针。

```

class Data
{
public:
    int x;
    bool operator>(const Data &other) const
    {
        return x > other.x;
    }
};

void OnStart()
{
    ...
    Data *pointer1 = new Data();
    Data *pointer2 = new Data();
    Max(pointer1, pointer2);
    delete pointer1;
    delete pointer2;
}

```

我们将在日志中看到 `T=Data*`（即指针属性）符合内联类型。这表明，如有必要，您可以编写模板函数的另一个重载，该函数将仅负责指针。

```

template<typename T>
T *Max(T *value1, T *value2)
{
    Print(__FUNCSIG__, " T=", typename(T));
    return value1 > value2 ? value1 : value2;
}

```

在这种情况下，指针属性 `*` 已存在于模板参数中，因此类型推断结果为 `T=Data*`。这种方法允许您为指针提供单独的模板实现。

如果有多个模板适用于生成具有特定类型的实例，则选择最特化的模板版本。具体来说，当使用指针自变量调用 *Max* 函数时，有两个模板分别带有参数 `T (T=Data*)` 和 `T* (T=Data)` 参数，但由于前者可以同时接受值和指针，因此比后者（只能使用指针）更通用。因此，对于指针，将选

择后者。换句话说，实际类型中用于替换 `T` 的修饰符越少，模板变体的优先级就越高。除了指针属性 `*` 之外，`const` 修饰符也同样适用。`const T*` 或 `const T` 参数分别比 `T*` 或 `T` 更特化。

3.3.5 函数模板

函数模板由一个包含模板参数的头文件（其语法已在 上文介绍过）和一个函数定义组成，在函数定义中，模板参数表示任意类型。

作为第一个例子，假设有一个 `Swap` 函数用于交换两个数组元素（*TemplatesSorting.mq5*）。模板参数 `T` 用作输入数组变量的类型以及局部变量 `temp` 的类型。

```
template<typename T>
void Swap(T &array[], const int i, const int j)
{
    const T temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

函数主体中的所有语句和表达式都必须适用于实数类型，模板随后将针对这些实数类型进行实例化。在本例中，使用了赋值运算符 `'='`。虽然它对于内置类型始终存在，但对于用户定义类型，可能需要显式重载。

编译器默认为类和结构体生成拷贝运算符的实现，但用户可以隐式或显式地删除（参见 *delete* 关键字）。具体来说，正如我们在 *对象类型转换* 一节中看到的，类中存在常量字段会导致编译器移除其隐式拷贝选项。因此，上述 `Swap` 模板函数不能用于此类的对象：编译器会报错。

对于 `Swap` 函数使用的类/结构体，最好不仅包含赋值运算符，还包含拷贝构造函数，因为 `temp` 变量的声明实际上是一个带有初始化的构造，而不是赋值。使用拷贝构造函数时，函数的第一行会一次性执行（`temp` 是基于 `array[i]` 创建的）；如果没有拷贝构造函数，则会先调用默认构造函数，然后对 `temp` 执行运算符 `'='`。

我们来看看如何在快速排序算法中使用 `Swap` 模板函数：另一个 `QuickSort` 模板函数实现了该算法。

```

template<typename T>
void QuickSort(T &array[], const int start = 0, int end = INT_MAX)
{
    if(end == INT_MAX)
    {
        end = start + ArraySize(array) - 1;
    }
    if(start < end)
    {
        int pivot = start;

        for(int i = start; i <= end; i++)
        {
            if(!(array[i] > array[end]))
            {
                Swap(array, i, pivot++);
            }
        }

        --pivot;

        QuickSort(array, start, pivot - 1);
        QuickSort(array, pivot + 1, end);
    }
}

```

请注意，*QuickSort* 模板的 *T* 参数指定了输入参数 *array* 的类型，然后该数组会被传递给 *Swap* 模板。因此，*QuickSort* 模板的类型推断 *T* 将自动确定 *Swap* 模板的类型 *T*。

内置函数 *ArraySize* (与许多其他函数一样)能够处理任意类型的数组：从某种意义上说，它也是一个模板，尽管它是直接在终端中实现的。

排序是通过 *if* 语句中的 *'>'* 比较运算符完成的。如前所述，必须为任何要排序的 *T* 类型定义此运算符，因为它适用于 *T* 类型数组的元素。

我们来了解一下内置类型数组的排序是如何进行的。

```

void OnStart()
{
    double numbers[] = {34, 11, -7, 49, 15, -100, 11};
    QuickSort(numbers);
    ArrayPrint(numbers);
    // -100.000000 -7.000000 11.000000 11.000000 15.000000 34.000000 49.000000

    string messages[] = {"usd", "eur", "jpy", "gbp", "chf", "cad", "aud", "nzd"};
    QuickSort(messages);
    ArrayPrint(messages);
    // "aud" "cad" "chf" "eur" "gbp" "jpy" "nzd" "usd"
}

```

两次调用 *QuickSort* 模板函数会根据传入数组的类型自动推断 *T* 的类型。因此，我们将获得两个分别对应 *double* 和 *string* 类型的 *QuickSort* 实例。

为了检查自定义类型的排序，我们创建一个包含 `x` 整数字段的 `ABC` 结构体，并在构造函数中用随机数填充此结构体。在结构体中重载 `>` 运算符也很重要。

```
struct ABC
{
    int x;
    ABC()
    {
        x = rand();
    }
    bool operator>(const ABC &other) const
    {
        return x > other.x;
    }
};

void OnStart()
{
    ...
    ABC abc[10];
    QuickSort(abc);
    ArrayPrint(abc);
    /* Sample output:
        [x]
        [0] 1210
        [1] 2458
        [2] 10816
        [3] 13148
        [4] 15393
        [5] 20788
        [6] 24225
        [7] 29919
        [8] 32309
        [9] 32589
    */
}
```

由于结构体值是随机生成的，我们会得到不同的结果，但它们始终按升序排列。

在这种情况下，也会自动推断出 `T` 类型。但在某些情况下，显式指定是将类型传递给函数模板的唯一方法。因此，如果模板函数必须返回唯一类型的值（与其参数的类型不同），如果没有参数，则只能显式指定。

例如，以下 `createInstance` 模板函数需要在调用指令中显式指定类型，因为无法根据返回值自动“计算”类型 `T`。如果不这样做，编译器会生成“模板不匹配”错误。


```

class Base
{
    ...
};

template<typename T>
T *createInstance()
{
    T *object = new T(); //calling the constructor
    ...                //object setting
    return object;
}

void OnStart()
{
    Base *p1 = createInstance();          // error: template mismatch
    Base *p2 = createInstance<Base>();    // ok, explicit directive
    ...
}

```

如果有多个模板参数，并且返回值的类型未绑定到函数的任何输入参数，则在调用时还需要指定特定类型：

```

template<typename T,typename U>
T MyCast(const U u)
{
    return (T)u;
}

void OnStart()
{
    double d = MyCast<double,string>("123.0");
    string f = MyCast<string,double>(123.0);
}

```

请注意，如果显式指定了模板的类型，则所有参数都需要这样做，即使第二个参数 `U` 可以从传递的自变量中推断出来。

编译器生成模板函数的所有实例后，将参与标准流程，从所有具有相同名称且参数数量合适的函数重载中选择最佳候选项。在所有重载选项（包括已创建的模板实例）中，选择类型最接近（转换次数最少）的选项。

如果模板函数包含某些特定类型的输入参数，则只有当这些类型与自变量完全匹配时，它才被视为候选：任何转换需求都会导致模板因不合适而被“丢弃”。

非模板重载优先于模板重载，越特化（“范围越狭窄”），就越容易在模板重载中胜出。

如果显式指定了模板自变量（类型），并且这些类型不同，则将对相应的函数自变量（传递的值）应用隐式类型转换规则（如有必要）。

如果一个函数的多个变体匹配度相同，我们将收到“对具有相同参数的重载函数的调用有歧义”错误。

例如，如果除了 `MyCast` 模板之外，还定义了一个将字符串转换为布尔类型的函数：

```
bool MyCast(const string u)
{
    return u == "true";
}
```

则调用 `MyCast<double,string>("123.0")` 将开始抛出指示的错误，因为这两个函数仅在返回值上有所不同：

```
'MyCast<double,string>' - ambiguous call to overloaded function with the same paramet
could be one of 2 function(s)
    double MyCast<double,string>(const string)
    bool MyCast(const string)
```

描述模板函数时，建议将所有模板参数包含在函数参数中。类型只能从自变量推断，而不能从返回值推断。

如果函数有一个带有默认值的模板化类型参数 `T`，并且在调用时省略了相应的参数，那么编译器也将无法推断 `T` 的类型，并抛出“无法应用模板”错误。

```
class Base
{
public:
    Base(const Base *source = NULL) { }
    static Base *type;
};

static Base* Base::type;

template<typename T>
T *createInstanceFrom(T *origin = NULL)
{
    T *object = new T(origin);
    return object;
}

void OnStart()
{
    Base *p1 = createInstanceFrom(); // error: cannot to apply template
    Base *p2 = createInstanceFrom(Base::type); // ok, auto-detect from argument
    Base *p3 = createInstanceFrom<Base>(); // ok, explicit directive, an argument
}
```

3.3.6 对象类型模板

对象类型模板定义以包含类型参数的头文件（参见[模板头文件](#)一节）以及类、结构体或联合体的常用定义开头。

```
template <typename T [, typename Ti ...] >
class class_name
{
    ...
};
```

与标准定义的唯一区别在于，模板参数可以出现在代码块中，出现在语言所有允许使用类型名称的语法结构中。

定义模板后，在代码中声明模板类型的变量时，会创建该模板的工作实例，并在尖括号中指定具体类型：

```
ClassName<Type1,Type2> object;
StructName<Type1,Type2,Type3> struct;
ClassName<Type1,Type2> *pointer = new ClassName<Type1,Type2>();
ClassName1<ClassName2<Type>> object;
```

与调用模板函数不同，编译器无法自行推断对象模板的实际类型。

声明模板类/结构体变量并非实例化模板的唯一方法。如果模板类型用作另一个特定（非模板）类或结构体的基类型，编译器也会生成实例。

例如，以下 *Worker* 类即使为空，也是 *Base* 针对 *double* 类型的实现：

```
class Worker : Base<double>
{
};
```

此最小定义（如果 *Base* 类需要，还可以添加构造函数）足以开始编译和验证模板代码。

在 [动态对象创建](#) 小节中，我们了解了使用 *new* 运算符获取的指向对象的动态指针的概念。这种灵活的机制有一个缺点：需要监视指针，并在不再需要时“手动”删除它们。具体来说，退出函数或代码块时，必须使用 *delete* 调用清除所有局部指针。

为了简化此问题的解决方案，我们创建一个模板类 *AutoPtr* (*TemplatesAutoPtr.mq5*、*AutoPtr.mqh*)。其参数 *T* 用于描述 *ptr* 字段，该字段存储指向任意类对象的指针。我们将通过构造函数参数 (*T *p*) 或重载运算符 '=' 获取指针值。我们将主要工作委托给析构函数：在析构函数中，指针将与 *AutoPtr* 对象一起被删除（为此分配了静态辅助方法 *free*）。

AutoPtr 的工作原理很简单：此类的局部对象在退出描述该对象的块时将自动销毁，如果之前指示该对象“跟随”某个指针，则 *AutoPtr* 也会释放该指针。

```

template<typename T>
class AutoPtr
{
private:
    T *ptr;

public:
    AutoPtr() : ptr(NULL) { }

    AutoPtr(T *p) : ptr(p)
    {
        Print(__FUNCSIG__, " ", &this, ": ", ptr);
    }

    AutoPtr(AutoPtr &p)
    {
        Print(__FUNCSIG__, " ", &this, ": ", ptr, " -> ", p.ptr);
        free(ptr);
        ptr = p.ptr;
        p.ptr = NULL;
    }

    ~AutoPtr()
    {
        Print(__FUNCSIG__, " ", &this, ": ", ptr);
        free(ptr);
    }

    T *operator=(T *n)
    {
        Print(__FUNCSIG__, " ", &this, ": ", ptr, " -> ", n);
        free(ptr);
        ptr = n;
        return ptr;
    }

    T* operator[](int x = 0) const
    {
        return ptr;
    }

    static void free(void *p)
    {
        if(CheckPointer(p) == POINTER_DYNAMIC) delete p;
    }
};

```

此外, *AutoPtr* 类实现了一个拷贝构造函数 (更准确地说, 是一个跳转构造函数, 因为当前对象将成为指针的所有者), 它允许您从函数返回一个 *AutoPtr* 实例以及一个受控指针。

为了测试 *AutoPtr* 的性能, 我们将描述一个虚构的 *Dummy* 类。

```

class Dummy
{
    int x;
public:
    Dummy(int i) : x(i)
    {
        Print(__FUNCSIG__, " ", &this);
    }
    ...
    int value() const
    {
        return x;
    }
};

```

在脚本的 *OnStart* 函数中, 输入 *AutoPtr<Dummy>* 变量并从 *generator* 函数获取其值。在 *generator* 函数本身中, 我们还将描述 *AutoPtr<Dummy>* 对象, 依次创建两个动态对象 *Dummy* 并将其附加到 *AutoPtr<Dummy>* 对象 (以检查是否正确释放了“旧”对象占用的内存)。

```

AutoPtr<Dummy> generator()
{
    AutoPtr<Dummy> ptr(new Dummy(1));
    // pointer to 1 will be freed after execution of '='
    ptr = new Dummy(2);
    return ptr;
}

void OnStart()
{
    AutoPtr<Dummy> ptr = generator();
    Print(ptr[].value());           // 2
}

```

由于所有主要方法都记录了对象描述符 (包括 *AutoPtr* 和受控指针 *ptr*), 因此我们可以跟踪所有指针“转换” (为方便起见, 所有行均已编号)。

```

01 Dummy::Dummy(int) 3145728
02 AutoPtr<Dummy>::AutoPtr<Dummy>(Dummy*) 2097152: 3145728
03 Dummy::Dummy(int) 4194304
04 Dummy*AutoPtr<Dummy>::operator=(Dummy*) 2097152: 3145728 -> 4194304
05 Dummy::~Dummy() 3145728
06 AutoPtr<Dummy>::AutoPtr<Dummy>(AutoPtr<Dummy>&) 5242880: 0 -> 4194304
07 AutoPtr<Dummy>::~~AutoPtr<Dummy>() 2097152: 0
08 AutoPtr<Dummy>::AutoPtr<Dummy>(AutoPtr<Dummy>&) 1048576: 0 -> 4194304
09 AutoPtr<Dummy>::~~AutoPtr<Dummy>() 5242880: 0
10 2
11 AutoPtr<Dummy>::~~AutoPtr<Dummy>() 1048576: 4194304
12 Dummy::~~Dummy() 4194304

```

我们暂时抛开模板, 详细描述一下该实用程序的工作原理, 因为这样的类对许多人来说都很有用。

在 *OnStart* 启动后, *generator* 函数立即被调用。它必须返回一个值, 以便在 *OnStart* 中初始

化 *AutoPtr* 对象；因此，*AutoPtr* 的构造函数此时尚未被调用。第 02 行在 *generator* 函数内部创建了一个对象 *AutoPtr*#2097152，并获取了一个指向第一个 *Dummy*#3145728 的指针。接下来，创建 *Dummy*#4194304 的第二个实例（第 03 行），它将用 *AutoPtr*#2097152 中的描述符 3145728（第 04 行）替换之前的副本，并删除旧副本（第 05 行）。第 06 行创建一个临时的 *AutoPtr*#5242880 来返回 *generator* 的值，并删除本地副本（07）。在第 08 行，*OnStart* 函数中 *AutoPtr*#1048576 对象的拷贝构造函数最终被使用，并将临时对象（在第 09 行立即被删除）的指针传递给该构造函数。接下来，我们使用指针的内容调用 *Print* 函数。当 *OnStart* 完成时，析构函数 *AutoPtr* (11) 将自动触发，从而使我们也删除工作对象 *Dummy* (12)。

模板技术使 *AutoPtr* 类成为一个动态分配对象的参数化管理器。但由于 *AutoPtr* 有一个 *T*ptr* 字段，它仅适用于类（更准确地说，是指向类对象的指针）。例如，尝试实例化字符串模板 (*AutoPtr<string> s*) 会导致模板文本中出现大量错误，这些错误表明该 *string* 类型不支持指针。

这并不算问题，因为此模板的用途仅限于类，但对于更通用的模板，应牢记这一细微差别（参见侧边栏）。

指针和引用

请注意，*T** 结构不能出现在您计划使用的模板中，包括内置类型或结构体。重点是，MQL5 中的指针仅允许用于类。这并不是说模板理论上不能同时适用于内置类型和用户定义类型，但可能需要进行一些调整。您可能需要放弃部分功能或牺牲模板的通用性（例如，创建多个模板而不是一个、重载函数等）。

将指针类型“注入”到模板的最直接方法是在实例化模板时，在实际类型中添加修饰符 *Ptr*（即必须与 *T=Type** 匹配）。但是，某些函数（例如 *CheckPointer*）、运算符（例如 *delete*）和语法结构（例如强制转换 *((T)variable)*）对其自变量/操作数是否为指针非常敏感。由于这个原因，同一个模板的文本对于指针类型和简单类型的值而言，在语法上并非总是适用。

另一个需要注意的重要类型差异是：对象只能通过引用传递给方法，而简单类型的字面量（常量）不能通过引用传递。因此，编译器可能会根据推断出的 *T* 类型，将和号的存在与否视为错误。作为一种“解决方法”，您可以选择将自变量常量“包装”到对象或变量中。

另一个技巧是使用模板方法。我们将在下一节中看到它。

需要注意的是，面向对象技术与模式非常契合。由于指向基类的指针可用于存储派生类的对象，因此 *AutoPtr* 适用于任何派生 *Dummy* 类的对象。

理论上，这种“混合”方法广泛应用于容器类（向量、队列、映射、列表等），这些类通常是模板。容器类可能根据实现方式对模板参数施加额外要求，特别是内联类型必须具有拷贝构造函数和赋值（copy）运算符。

MetaTrader 5 提供的 MQL5 标准库包含许多现成的模板：*Stack.mqh*、*Queue.mqh*、*HashMap.mqh*、*LinkedList.mqh*、*RedBlackTree.mqh*，等。它们都位于 MQL5/Include/Generic 目录中。没错，它们不提供对动态对象（指针）的控制。

我们将在 [方法模板](#) 中查看我们简单容器类的示例。

3.3.7 方法模板

不仅整个对象类型可以作为模板，其单独的方法（简单方法或静态方法）也可以作为模板。虚方法是个例外：它们不能被声明为模板。因此，模板方法不能在 [接口](#) 一节中显示了相关示例。但是，接口本身可以作为模板，而虚方法可以存在于类模板中。

当方法模板包含在类/结构体模板中时，两个模板的参数必须不同。如果有多个模板方法，它们的参数之间没有任何关联，并且可以具有相同的名称。

方法模板的声明方式类似于 **函数模板**，但仅限于类、结构体或联合体（它们可能是也可能不是模板）的上下文中。

```
[ template < typename T [, typename Ti ...] > ]
class class_name
{
    ...
    template < typename U [, typename Ui ...] >
    type method_name(parameters_with_types_T_and_U)
    {
    }
};
```

参数、返回值和方法主体可以使用类型 **T**（类的通用类型）和 **U**（方法的特定类型）。

只有在程序代码中调用特定参数组合的方法时，才会生成该方法的实例。

在上一节中，我们描述了用于存储和释放单个指针的模板类 **AutoPtr**。当存在多个相同类型的指针时，将它们放入容器对象中会很方便。我们创建一个具有类似功能的简单模板—**SimpleArray** 类（*SimpleArray.mqh*）。为了避免重复控制动态内存释放的功能，我们将在类契约中声明它用于存储值和对象，而不是指针。为了存储指针，我们将它们放入 **AutoPtr** 对象中，然后将这些对象放入容器中。

这还有另一个好处：由于 **AutoPtr** 对象很小，因此易于复制（不会耗费过多资源），这在函数之间交换数据时经常发生。**AutoPtr** 指向的应用程序类的对象可以很大，甚至不需要在其中实现自己的拷贝构造函数。

当然，从函数返回指针的开销更小，但您需要重新设计内存释放控制的方法。因此，使用 **AutoPtr** 形式的现成解决方案更为简单。

对于容器内的对象，我们将创建模板类型 **T** 的 **data** 数组。

```
template<typename T>
class SimpleArray
{
protected:
    T data[];
    ...
};
```

由于容器的主要操作之一是添加元素，因此我们将提供一个辅助函数来扩展数组。

```
int expand()
{
    const int n = ArraySize(data);
    ArrayResize(data, n + 1);
    return n;
}
```

我们将直接通过重载运算符 '<<' 添加元素。它使用通用模板参数 **T**。

```
public:
    SimpleArray *operator<<((const T &r)
    {
        data[expand()] = (T)r;
        return &this;
    }
```

此选项通过引用获取值，即变量或对象。您现在应该注意这一点，这一点的重要性稍后就会显现。

读取元素是通过重载运算符 ‘[]’ 来完成的（它具有最高优先级，因此表达式中不需要使用括号）。

```
T operator[](int i) const
{
    return data[i];
}
```

首先，我们确保该类能够处理结构体示例。

```
struct Properties
{
    int x;
    string s;
};
```

为此，我们将在 `OnStart` 函数中描述一个用于该结构体的容器，并将一个对象 (`TemplatesSimpleArray.mq5`) 放入其中。

```
void OnStart()
{
    SimpleArray<Properties> arrayStructs;
    Properties prop = {12345, "abc"};
    arrayStructs << prop;
    Print(arrayStructs[0].x, " ", arrayStructs[0].s);
    ...
}
```

通过调试日志记录，您可以验证该结构体是否位于容器中。

现在，让我们尝试在容器中存储一些数字。

```
SimpleArray<double> arrayNumbers;
arrayNumbers << 1.0 << 2.0 << 3.0;
```

遗憾的是，我们会收到“参数作为引用传递，应为变量”的错误，而这恰恰发生在重载运算符 ‘<<’ 中。

我们需要一个通过值传递参数的重载。但是，我们不能直接写一个不包含 `const` 和 ‘&’ 的类似方法：


```
SimpleArray *operator<<(T r)
{
    data[expand()] = (T)r;
    return &this;
}
```

如果这样做，新的变体将导致对象类型的模板无法编译：毕竟，对象只需要通过引用传递。即使该函数不用于对象，它仍然存在于类中。因此，我们将新方法定义为带有自身参数的模板。

```
template<typename T>
class SimpleArray
{
    ...
    template<typename U>
    SimpleArray *operator<<(U u)
    {
        data[expand()] = (T)u;
        return &this;
    }
}
```

只有当通过值传递数据给 ‘<<’ 运算符时，它才会出现在类中，这表明传递的绝对不是一个对象。诚然，我们无法保证 T 和 U 相同，因此会执行显式强制类型转换 $(T)u$ 。对于内置类型（如果两种类型不匹配），在某些情况下，转换可能会损失精度，但代码肯定能通过编译。唯一的例外是禁止将字符串转换为布尔类型，但该容器不太可能用于 *bool* 数组，因此这个限制并不重要。希望解决这个问题的人可以自行解决。

使用新的模板方法，容器 *SimpleArray<double>* 可以正常工作，并且不会与 *SimpleArray<Properties>* 冲突，因为这两个模板实例在生成的源代码中存在差异。

最后，我们检查一下包含 *AutoPtr* 对象的容器。为此，我们准备一个简单的 *Dummy* 类，它将为 *AutoPtr* 内部的指针“提供”对象。

```
class Dummy
{
    int x;
public:
    Dummy(int i) : x(i) { }
    int value() const
    {
        return x;
    }
};
```

在 *OnStart* 函数中，创建一个容器 *SimpleArray<AutoPtr<Dummy>>* 并填充它。

```

void OnStart()
{
    SimpleArray<AutoPtr<Dummy>> arrayObjects;
    AutoPtr<Dummy> ptr = new Dummy(20);
    arrayObjects << ptr;
    arrayObjects << AutoPtr<Dummy>(new Dummy(30));
    Print(arrayObjects[0][].value());
    Print(arrayObjects[1][].value());
}

```

回想一下，在 *AutoPtr* 中，运算符 '[]' 用于返回一个存储的指针，因此 *arrayObjects[0][]* 的含义是：将 *data* 数组的第 0 个元素返回到 *SimpleArray* 中，即 *AutoPtr* 对象，然后将第二对方括号应用于该卷，得到一个指针 *Dummy**。接下来，我们可以操作该对象的所有特性：在本例中，我们获取 *x* 字段的当前值。

由于 *Dummy* 没有拷贝构造函数，因此如果不使用 *AutoPtr* 便无法使用容器直接存储这些对象。

```

// ERROR:
// object of 'Dummy' cannot be returned,
// copy constructor 'Dummy::Dummy(const Dummy &)' not found
SimpleArray<Dummy> bad;

```

但机智的用户应该能猜到如何解决这个问题。

```

SimpleArray<Dummy*> bad;
bad << new Dummy(0);

```

这段代码可以编译并运行。然而，这个“解决方案”存在一个问题：*SimpleArray* 不知道如何控制指针，因此，当程序退出时，会检测到内存泄漏。

```

1 undeleted objects left
1 object of type Dummy left
24 bytes of leaked memory

```

作为 *SimpleArray* 的开发者，我们有责任弥补这个漏洞。为此，我们向类中添加另一个模板方法，并重载运算符 '<<'——这次是针对指针。由于它是一个模板，因此它只会在“按需”包含在生成的源代码中：当程序员尝试使用此重载时，即向容器写入一个指针时。否则，该方法将被忽略。

```

template<typename T>
class SimpleArray
{
    ...
    template<typename P>
    SimpleArray *operator<<(P *p)
    {
        data[expand()] = (T)*p;
        if(CheckPointer(p) == POINTER_DYNAMIC) delete p;
        return &this;
    }
}

```

当用指针类型实例化一个模板时，这种特化会抛出编译错误（“需要对象指针”）。因此，我们会告知用户不支持此模式。

```
SimpleArray<Dummy*> bad; // ERROR is generated in SimpleArray.mqh
```

此外，它还会执行另一项保护措施。如果客户端类仍然具有拷贝构造函数，那么将动态分配的对象保存在容器中将不再导致内存泄漏：所传递指针 *P *p* 处的对象副本将保留在容器中，而原始对象将被删除。当容器在 *OnStart* 函数末尾被销毁时，其内部 *data* 数组将自动调用其元素的析构函数。

```
void OnStart()
{
    ...
    SimpleArray<Dummy> good;
    good << new Dummy(0);
} // SimpleArray "cleans" its elements
// no forgotten objects in memory
```

方法模板和“简单”方法可以在主类块（或类模板）之外定义，类似于我们在 [拆分类的声明和定义](#) 一节中看到的内容。同时，它们前面都带有模板头文件 (*TemplatesExtended.mqh5*):

```
template<typename T>
class ClassType
{
    ClassType() // private constructor
    {
        s = &this;
    }
    static ClassType *s; // object pointer (if it was created)
public:
    static ClassType *create() // creation (on first call only)
    {
        static ClassType single; //single pattern for every T
        return single;
    }

    static ClassType *check() // checking pointer without creating
    {
        return s;
    }

    template<typename U>
    void method(const U &u);
};

template<typename T>
template<typename U>
void ClassType::method(const U &u)
{
    Print(__FUNCSIG__, " ", typename(T), " ", typename(U));
}

template<typename T>
static ClassType<T> *ClassType::s = NULL;
```

它还展示了模板化静态变量的初始化，表明采用了单例模式。

在 *OnStart* 函数中，创建模板实例并进行测试：

```
void OnStart()
{
    ClassType<string> *object = ClassType<string>::create();
    double d = 5.0;
    object.method(d);
    // OUTPUT:
    // void ClassType<string>::method<double>(const double&) string double

    Print(ClassType<string>::check()); // 1048576 (an example of an instance id)
    Print(ClassType<long>::check());   // 0 (there is no instance for T=long)
}
```

3.3.8 嵌套模板

模板可以嵌套在类/结构体或其他类/结构体模板中。联合体也是如此。

在 *联合体* 一节中，我们了解了将 *long* 值“转换”为 *double*，然后再转换回来，而不会损失精度。

现在，我们可以使用模板编写一个通用的“转换器”(TemplatesConverter.mq5)。模板类 *Converter* 有两个参数 T1 和 T2，用于指示将在哪些类型之间执行转换。为了根据一种类型的规则写入值，并根据另一种类型的规则读取值，我们同样需要一个联合体。我们还将其制作成一个带有参数 U1 和 U2 的模板 (*DataOverlay*)，并在类内部进行定义。

该类通过重载运算符 □ 提供了一种便捷的转换方式，在该运算符的实现中，可以对联合体字段进行写入和读取。

```

template<typename T1,typename T2>
class Converter
{
private:
    template<typename U1,typename U2>
    union DataOverlay
    {
        U1 L;
        U2 D;
    };

    DataOverlay<T1,T2> data;

public:
    T2 operator[](const T1 L)
    {
        data.L = L;
        return data.D;
    }

    T1 operator[](const T2 D)
    {
        data.D = D;
        return data.L;
    }
};

```

联合体用于描述类中的 *DataOverlay<T1,T2>data* 字段。我们可以在 *DataOverlay* 中直接使用 *T1* 和 *T2*，而无需将此联合体设为模板。但为了演示该技术本身，在生成 *data* 字段后，外部模板的参数会传递给内部模板。在 *DataOverlay* 内部，除了 *T1* 和 *T2* 之外，同一对类型将被称为 *U1* 和 *U2*。

我们来看看模板的实际应用。

```

#define MAX_LONG_IN_DOUBLE          9007199254740992

void OnStart()
{
    Converter<double,ulong> c;

    const ulong value = MAX_LONG_IN_DOUBLE + 1;

    double d = value; // possible loss of data due to type conversion
    ulong result = d; // possible loss of data due to type conversion

    Print(value == result); // false

    double z = c[value];
    ulong restored = c[z];

    Print(value == restored); // true
}

```

3.3.9 缺少模板特化

在某些情况下，可能需要为特定类型（或类型集）提供与泛型不同的模板实现。例如，通常需要为指针或数组准备一个特殊版本的 `swap` 函数。在这种情况下，C++ 允许您执行所谓的模板特化，即定义一个模板版本，其中泛型类型参数 `T` 被所需的具体类型替换。

在特化函数和方法模板时，必须为所有参数指定特定类型。这称为完全特化。

对于 C++ 对象类型模板，特化可以是完全特化，也可以是偏特化：它仅指定部分参数的类型（其余参数的类型将在模板实例化时推断或指定）。可以存在多个偏特化：唯一的条件是每个特化必须描述唯一的类型组合。

遗憾的是，MQL5 中没有完全意义上的特化。

模板函数特化与重载并无二致。例如，给定以下模板 *func*：

```
template<typename T>
void func(T t) { ... }
```

它允许以以下形式之一为给定类型（例如 *string*）提供其自定义实现：

```
// explicit specialization
template<>
void func(string t) { ... }
```

或：

```
// normal overload
void func(string t) { ... }
```

只能选择其中一种形式。否则，我们会收到编译错误“*func*’-函数已定义且具有主体”。

关于类的特化，从那些为部分模板参数指定了具体类型的模板进行继承，可以被看作是这些模板偏特化的一种等价实现。模板方法可以在派生类中被重写。

以下示例 (*TemplatesExtended.mq5*) 展示了将模板参数用作父类型的几种方案，包括将其中一个参数指定为特定类型的情况。

```

#define RTTI Print(typename(this))

class Base
{
public:
    Base() { RTTI; }
};

template<typename T>
class Derived : public T
{
public:
    Derived() { RTTI; }
};

template<typename T>
class Base1
{
    Derived<T> object;
public:
    Base1() { RTTI; }
};

template<typename T>                // complete "specialization"
class Derived1 : public Base1<Base> // 1 of 1 parameter is set
{
public:
    Derived1() { RTTI; }
};

template<typename T,typename E>
class Base2 : public T
{
public:
    Base2() { RTTI; }
};

template<typename T>                // partial "specialization"
class Derived2 : public Base2<T,string> // 1 of 2 parameters is set
{
public:
    Derived2() { RTTI; }
};

```

我们将使用变量根据模板实例化对象：

```
Derived2<Derived1<Base>> derived2;
```

使用 RTTI 宏进行调试类型日志记录会产生以下结果：

```
Base
Derived<Base>
Base1<Base>
Derived1<Base>
Base2<Derived1<Base>,string>
Derived2<Derived1<Base>>
```

在开发以封闭二进制形式提供的 [库](#) 时，必须确保针对未来的库用户预期使用的所有类型，显式实例化模板。您可以通过显式地调用函数模板，并在某些辅助函数中（例如，这些函数可以绑定到全局变量的初始化）使用具体的类型参数来创建对象，从而实现此目的。

第四章常用 MQL5 API

在本书前面章节中，我们探讨了使用 MQL5 语言结构的基本概念、句法和规则。然而，要想编写满足交易员要求的真实程序（如分析性数据处理和自动化交易等），这些还只是基础。没有大量的各种内置函数以及与 MetaTrader 5 终端交互的方式，不可能完成这些任务，而正是这些函数和交互手段构成了 MQL5 API。

在本章中，我们将开始掌握 MQL5 API，这一学习过程将直至本书结束，让你逐渐熟悉所有的专用子系统。

由内核（终端内 MQL 程序的运行时环境）提供给任何 MQL 程序的技术和功能非常庞杂。因此，最好首先学习最简单但对于大多数程序都有用的内容。尤其是我们将了解专用于处理数组、字符串、文件、数据转换、用户交互、数学函数以及环境控制的函数。

前面我们学习了在 MQL5 中描述自己的 [函数](#) 并调用它们。MQL5 API 的内置函数可从源代码获得，所谓的“开箱即用”，无需任何预先说明即可使用。

需要注意的是，不同于 C++，MQL5 无需额外的预处理程序指令即可将特定的内置函数集包括在程序内。所有 MQL5 API 函数的名称均始终无条件地出现在全局文本（命名空间）中。

一方面，这很方便，但另一方面，这需要你注意可能的名称冲突。如果你无意中试图使用内置函数的某个名称，其将覆盖标准实现，可能导致非预期后果：最好的情况是出现有关模糊重载的编译器错误，最糟糕的情况，则是所有常规调用将会在无任何警告的情况下被重定向至“新”实现。

理论上，相似名称可在其它上下文中使用，例如作为类方法名称，或者用于专用（用户）命名空间。在此类情况下，可使用上下文解算运算符调用全局函数：我们将在 [嵌套类型、命名空间以及 '::' 上下文运算符](#) 一节中探讨这种情况。

 [面向交易员的 MQL5 编程](#) 来自本书的源代码。第四章

 本书中的例子也可以在 [公共项目 \MQL5\Shared Projects\MQL5Book](#) 中找到

4.1 内置类型转换

程序通常操作不同的数据类型。我们已在 [类型强制转换](#) 一节中接触了内置类型的显式和隐式转换。它们提供了由于各种原因而非始终适合的通用转换方法。MQL5 API 提供了一系列函数，编程人员使用这些函数可管理从一种类型到另一种类型的数据转换，以及配置转换结果。

最常用的函数包括将各种类型数据转换为字符串或者进行相反转换的函数。具体而言，这包括转换数字、日期、时间、颜色、结构体和枚举。某些类型具有附加的特定运算。

本节考虑各种数据转换方法，为编程人员提供必要工具处理交易机器人使用的各种数据类型。本节包括以下分节：

[数字转换为字符串以及相反转换](#)：

- ① 本小节探讨将数值转换为字符串及相反转换的方法。它涵盖诸如数字格式设置以及处理各种数字系统等重要方面。

[双精度浮点数规范化](#)：

- ①在处理财务数据时，双精度浮点数规范化是一个重要的方面。本节探讨标准化方法、避免精度损失以及处理浮点值的方式。

日期和时间：

- 日期和时间转换在交易策略中扮演关键角色。本节探讨处理日期、时间间隔以及诸如 `datetime` 等特殊数据类型的方法。

颜色：

- ①在 MQL5 中，颜色以特殊数据类型表示。本分节探讨颜色值转换、表示及其在交易机器人的图形元素中的使用。

结构体：

- ①在处理复杂的结构体化数据时，结构体内的数据转换是一个重要主题。我们将了解与结构体及其元素交互的方法。

枚举：

- ①枚举可提供已命名常量，并增强代码可读性。本小节探讨如何转换枚举值以及在程序中有效使用它们。

复数类型：

- ①复数类型设计用于处理复数。本节讨论用于转换和使用复数的方法。

我们将在本章中研究所有此类函数。

4.1.1 数字与字符串的相互转换

数字与字符串的相互转换可以使用 **显式类型强制转换** 运算符完成。例如，对于 `double` 和 `string` 类型，可能如下所示：

```
double number = (double)text;
string text = (string)number;
```

字符串可转换为其它数值类型，诸如 `float`、`long`、`int` 等等。

请注意，强制转换为实数类型 (`float`) 可实现更少的有效数位，这在某些应用中可能有优势，因为实数类型更紧凑，表示形式更易读。

严格来讲，这种类型强制转换并非强制性的，因为即使没有显式强制转换运算符，编译器也会隐式生成类型强制转换。但是在这种情况下将会出现编译器警告，因此，建议始终进行显示类型强制转换。

MQL5 API 还提供了其他一些有用的函数，将在下文中介绍。说明后面附带一般示例。

double StringToDouble(string text)

`StringToDouble` 函数可将字符串转换为 `double` 数字。

它是对到 (`double`) 的类型强制转换的完整模拟。其实用目的实际上仅仅是保持与旧版源代码的向后兼容性。首选方法是类型强制转换，因为它更紧凑，并在语言句法内实现。

根据转换过程，字符串应包含满足数值类型 (包括 **浮点数** 和 **整数**) 字面量写入规则的一系列字符。尤其是字符串能以 '+' 号或 '-' 号开头，后面是数字，并且后面还可以一系列数字。

实数可包含单点字符 ‘.’，按以下格式分隔小数部分和可选的指数：字符 ‘e’ 或 ‘E’，后面是表示指数的一系列数字（其前面也可添加 ‘+’ 或 ‘-’）。

对于整数，支持十六进制计数法，即 “0x” 前缀后面不仅可以跟十进制数字，而且可以跟 ‘A’、‘B’、‘C’、‘D’、‘E’、‘F’（在任何位置）。

当字符串中碰到任何非预期字符（诸如字母、标点符号、第二句号或中间空格）时，转换终止。在此情况下，若在该位置前有允许的字符，它们被解释为数字，反之，结果将为 0。

初始空字符（空格、制表符、换行符）被跳过，不影响转换。如果后面是满足规则的数字及其它字符，则该数字将被正确接收。

下表提供了带有说明的一些有效转换示例。

字符串	双精度浮点数	结果
"123.45"	123.45	一个小数点
"\t 123"	123.0	开始部分的空白字符被忽略
"-12345"	-12345.0	有符号数
"123e-5"	0.00123	带指数的科学计数法
"0x12345"	74565.0	十六进制记数法

下表列出了不正确转换示例。

字符串	双精度浮点数	结果
"x12345"	0.0	以无法解析的字符（字母）开始
"123x45"	123.0	123 后面的字母中断了转换
" 12 3"	12.0	12 后面的空格中断了转换
"123.4.5"	123.4	123.4 后面的第二个小数点中断了转换
"1,234.50"	1.0	1 后面的逗号中断了转换
"-+12345"	0.0	太多符号（两个）

string DoubleToString(double number, int digits = 8)

DoubleToString 函数将数字转换为具有指定精度的字符串（位数从 -16 到 16）。

它的功能类似于将数字强制转换为 *(string)*，但允许你使用第二个参数选择生成的字符串的数字精度。

应用到 *double* 的运算符 *(string)* 显示 16 个有效数位（总计，包括尾数和小数部分在内）。函数无法实现与此完全等效的结果。

如果 *digits* 参数大于或等于 0，则其表示小数位数。在此情况下，小数点前面的字符数由数字本身（数字大小）确定，并且如果尾数中的总字符数以及 *digits* 中指示的总字符数大于 16，则最

不重要的数位将包含“垃圾”(由于实数存储的方式而导致)。16 个字符表示 *double* 类型的平均最大精度,也就是将 *digits* 设为 16 (最大)将仅提供低于 10 的准确值表示。

如果 *digits* 参数小于 0,则其指定有效数位数,该数字将以带指数的科学计数法输出。就精度(而不是记录格式)而言,在函数中设置 *digits=-16* 将生成接近强制转换 (*string*) 的结果。

该函数通常用于统一数据集的格式设置(包括某个表格的列右对齐),使得值具有相同的小数精度(例如,金融工具价格或手数的小数位数)。

请注意,数学计算过程中可能发生错误,导致结果不是一个有效数字,尽管该数字的类型为 *double* (或 *float*)。例如,一个变量可能包含计算负数的平方根的结果。

此类值称为“非数字”(NaN)值,并在强制转换为 (*string*) 时显示为简短错误类型提示,如 `-nan(ind)` (`ind` - 未定义)、`-nan(inf)` (`inf` - 无穷)。使用 *DoubleToString* 函数时,会得到一个不合理的大数字。

尤其重要的是,所有具有 NaN 的后续计算也将给出 NaN。要检查此类值,可使用 *MathIsValidNumber* 函数。

long StringToInteger(string text)

该函数将字符串转换为 *long* 类型的数字。请注意,结果类型肯定为 *long*,而不是 *int* (不论名称如何)或 *ulong*。

一种替代方法是使用运算符 (*long*) 进行类型强制转换。此外,你选择的任何其它整数类型均可用于强制转换: (*int*)、(*uint*)、(*ulong*) 等等。

转换规则类似于 *double* 类型,但不包括点字符以及来自允许字符的指数。

string IntegerToString(long number, int length = 0, ushort filling = ' ')

函数 *IntegerToString* 将 *long* 类型的整数转换为指定长度的字符串。如果数字表示少于一个字符,则其左填补一个字符 *filling* (默认填补一个空格)。否则,数字完整无限制显示。调用具有默认参数的函数等效于强制转换为 (*string*)。

当然,较小的整数类型(如 *int*、*short*)可以由函数成功处理。

使用所有上述函数的示例在脚本 *ConversionNumbers.mq5* 中提供。

```

void OnStart()
{
    const string text = "123.4567890123456789";
    const string message = "-123e-5 buckazoid";
    const double number = 123.4567890123456789;
    const double exponent = 1.234567890123456789e-5;

    // type casting
    Print((double)text);    // 123.4567890123457
    Print((double)message); // -0.00123
    Print((string)number);  // 123.4567890123457
    Print((string)exponent); // 1.234567890123457e-05
    Print((long)text);      // 123
    Print((long)message);   // -123

    // converting with functions
    Print(StringToDouble(text)); // 123.4567890123457
    Print(StringToDouble(message)); // -0.00123

    // by default, 8 decimal digits
    Print(DoubleToString(number)); // 123.45678901

    // custom precision
    Print(DoubleToString(number, 5)); // 123.45679
    Print(DoubleToString(number, -5)); // 1.23457e+02
    Print(DoubleToString(number, -16)); // 1.2345678901234568e+02
    Print(DoubleToString(number, 16)); // 123.4567890123456807
    // last 2 digits are not accurate!
    Print(MathSqrt(-1.0)); // -nan(ind)
    Print(DoubleToString(MathSqrt(-1.0))); // 9223372129088496176.54775808

    Print(StringToInteger(text)); // 123
    Print(StringToInteger(message)); // -123

    Print(IntegerToString(INT_MAX)); // '2147483647'
    Print(IntegerToString(INT_MAX, 5)); // '2147483647'
    Print(IntegerToString(INT_MAX, 16)); // '2147483647'
    Print(IntegerToString(INT_MAX, 16, '0')); // '0000002147483647'
}

```

4.1.2 双精度浮点数的规范化

MySQL API 提供了用于将浮点数字舍入为指定精度 (小数部分有效数位数) 的函数。

double NormalizeDouble(double number, int digits)

交易算法中需要舍入以设置 订单中的交易量和价格。根据标准规则执行舍入: 如果下一位 (舍弃的) 数字大于或等于 5, 则最后可见数字增加 1。

digits 参数的有效值: 0 至 8。

该函数的使用示例在 *ConversionNormal.mq5* 文件中提供。

```
void OnStart()
{
    Print(M_PI); // 3.141592653589793
    Print(NormalizeDouble(M_PI, 16)); // 3.14159265359
    Print(NormalizeDouble(M_PI, 8)); // 3.14159265
    Print(NormalizeDouble(M_PI, 5)); // 3.14159
    Print(NormalizeDouble(M_PI, 1)); // 3.1
    Print(NormalizeDouble(M_PI, -1)); // 3.14159265359
    ...
}
```

由于任何实数均具有有限的 内部表示 精度，因此即使规范化后，数字也可能是近似显示：

```
...
Print(512.06); // 512.05999999999999
Print(NormalizeDouble(512.06, 5)); // 512.05999999999999
Print(DoubleToString(512.06, 5)); // 512.060000000
Print((float)512.06); // 512.06
}
```

这是正常情况，无法避免。要获得更紧凑的格式，可使用 `DoubleToString`、`StringFormat` 函数或以中间方式强制转换为 `(float)`。

要将一个数字舍入为最近整数，使用 `MathRound`、`MathCeil`、`MathFloor` 函数（参见章节 [舍入函数](#)）。

4.1.3 日期和时间

`datetime` 类型的值（旨在用于存储 日期和/或时间）通常经过数种类型的转换：

- 与字符串相互转换，以向用户显示数据，以及从外部源读取数据
- 转换为特殊结构体 `MqlDateTime`（见下文），以处理单独日期和时间分量
- 转换为自 01/01/1970 以来的秒数，对应于 `datetime` 的内部表示，并等效于整数类型 `long`

对于最后一项，使用 `datetime` 到 `(long)` 强制转换或 `long` 到 `(datetime)` 反向强制转换，但请注意支持的日期范围从 1970 年 1 月 1 日（值 0）到 3000 年 12 月 31 日（32535215999 秒）。

对于前两个选项，MQL5 API 提供了以下函数。

`string TimeToString(datetime value, int mode = TIME_DATE | TIME_MINUTES)`

`TimeToString` 函数可根据 `mode` 参数（可在该参数中指定任意的标志组合），将 `datetime` 类型的值转换为包含日期和时间分量的字符串：

- `TIME_DATE` - 格式为 “YYYY.MM.DD” 的日期
- `TIME_MINUTES` - 格式为 “hh:mm” 的时间，即带小时和分钟
- `TIME_SECONDS` - 格式为 “hh:mm:ss” 的时间，即带小时、分钟和秒

要完整输出日期和时间数据，可将 `mode` 设置为等于 `TIME_DATE | TIME_SECONDS`（`TIME_DATE | TIME_MINUTES | TIME_SECONDS` 选项同样有效，但多余）。这等效于将类型 `datetime` 的值强制转换为 `(string)`。

使用示例在 `ConversionTime.mq5` 文件中提供。

```

#define PRT(A) Print(#A, "=", (A))

void OnStart()
{
    datetime time = D'2021.01.21 23:00:15';
    PRT((string)time);
    PRT(TimeToString(time));
    PRT(TimeToString(time, TIME_DATE | TIME_MINUTES | TIME_SECONDS));
    PRT(TimeToString(time, TIME_MINUTES | TIME_SECONDS));
    PRT(TimeToString(time, TIME_DATE | TIME_SECONDS));
    PRT(TimeToString(time, TIME_DATE));
    PRT(TimeToString(time, TIME_MINUTES));
    PRT(TimeToString(time, TIME_SECONDS));
}

```

脚本将打印以下日志：

```

(string)time=2021.01.21 23:00:15
TimeToString(time)=2021.01.21 23:00
TimeToString(time,TIME_DATE|TIME_MINUTES|TIME_SECONDS)=2021.01.21 23:00:15
TimeToString(time,TIME_MINUTES|TIME_SECONDS)=23:00:15
TimeToString(time,TIME_DATE|TIME_SECONDS)=2021.01.21 23:00:15
TimeToString(time,TIME_DATE)=2021.01.21
TimeToString(time,TIME_MINUTES)=23:00
TimeToString(time,TIME_SECONDS)=23:00:15

```

datetime StringToTime(string value)

StringToTime 函数将包含日期和/或时间的字符串转换为 *datetime* 类型的值。字符串可仅包含日期、仅包含时间，也可以同时包含日期和时间。

以下格式被视为日期：

- "YYYY.MM.DD"
- "YYYYMMDD"
- "YYYY/MM/DD"
- "YYYY-MM-DD"
- "DD.MM.YYYY"
- "DD/MM/YYYY"
- "DD-MM-YYYY"

以下格式支持时间：

- "hh:mm"
- "hh:mm:ss"
- "hhmmss"

日期和时间之间必须有至少一个空格。

如果字符串中仅有时间，则在结果中将替换为当前日期。如果字符串中仅有日期，则时间将设为 00:00:00。

如果字符串中支持的语法出现错误，则结果为当前日期。

函数使用示例在脚本 *ConversionTime.mq5* 中提供。

```
void OnStart()
{
    string timeonly = "21:01"; // time only
    PRT(timeonly);
    PRT((datetime)timeonly);
    PRT(StringToTime(timeonly));

    string date = "2000-10-10"; // date only
    PRT((datetime)date);
    PRT(StringToTime(date));
    PRT((long)(datetime)date);
    long seconds = 60;
    PRT((datetime)seconds); // 1 minute from the beginning of 1970

    string ddmmyy = "15/01/2012 01:02:03"; // date and time, and the date in
    PRT(StringToTime(ddmmyy)); // in "forward" order, still ok

    string wrong = "January 2-nd";
    PRT(StringToTime(wrong));
}
```

在日志，将看到类似如下所示的内容（####.##.## 是启动脚本的当前日期）：

```
timeonly=21:01
(datetime)timeonly=####.##.## 21:01:00
StringToTime(timeonly)=####.##.## 21:01:00
(datetime)date=2000.10.10 00:00:00
StringToTime(date)=2000.10.10 00:00:00
(long)(datetime)date=971136000
(datetime)seconds=1970.01.01 00:01:00
StringToTime(ddmmyy)=2012.01.15 01:02:03
(datetime)wrong=####.##.## 00:00:00
```

除了 *StringToTime*，还可以使用强制转换运算符 (*datetime*) 将字符串转换为日期和时间。然而，函数的优势在于，当检测到不正确的源字符串时，函数会设置错误代码为 *_LastError* 的内部变量（该错误代码也可通过函数 *GetLastError* 获取）。根据字符串的哪部分包含未解读数据，错误代码可能是 *ERR_WRONG_STRING_DATE* (5031)、*ERR_WRONG_STRING_TIME* (5032)，或者是列表中与从字符串获取日期和时间相关的其它选项。

bool TimeToStruct(datetime value, MqlDateTime &struct)

为单独解析日期和时间，MQL5 API 提供了 *TimeToStruct* 函数，该函数将 *datetime* 类型的值转换为 *MqlDateTime* 结构体：


```

struct MqlDateTime
{
    int year;           // year
    int mon;            // month
    int day;            // day
    int hour;           // hour
    int min;            // minutes
    int sec;            // seconds
    int day_of_week;    // day of the week
    int day_of_year;    // the number of the day in a year (January 1 has number 0)
};

```

周天数以美国格式编号:0 代表周日,1 代表周一,以此类推,直至 6 代表周六。可使用内置的 `ENUM_DAY_OF_WEEK` 枚举识别。

如果成功,则函数返回 *true*,如果出错,则返回 *false*,尤其是在传递了不正确日期的情况下。

让我们使用 `ConversionTimeStruct.mq5` 脚本检查函数的执行。为此,我们创建包含测试值的 *time* 数组 (类型为 *datetime*)。我们将在循环中为每个测试值调用 *TimeToStruct*。

结果将添加至结构体数组 *MqlDateTime mdt[]*。我们将首先将其初始化为零,但由于内置函数 *ArrayInitialize* 不知道如何处理这些结构体,我们必须为其编写一个重载 (以后我们将在以下章节学习以更方便的方式以零填充数组:将对象和数组清零 将介绍 *ZeroMemory* 函数)。

```

int ArrayInitialize(MqlDateTime &mdt[], MqlDateTime &init)
{
    const int n = ArraySize(mdt);
    for(int i = 0; i < n; ++i)
    {
        mdt[i] = init;
    }
    return n;
}

```

在该过程之后,我们将使用内置函数 *ArrayPrint*将结构体数组输出到日志。这是实现良好数据格式的最简便方法 (即使只有一个结构体也可以使用:只需将其放入一个大小为 1 的数组即可)。

```

void OnStart()
{
    // fill the array with tests
    datetime time[] =
    {
        D'2021.01.28 23:00:15', // valid datetime value
        D'3000.12.31 23:59:59', // the largest supported date and time
        LONG_MAX // invalid date: will cause an error ERR_INVALID_DATETIME (4010)
    };

    // calculate the size of the array at compile time
    const int n = sizeof(time) / sizeof(datetime);

    MqlDateTime null = {}; // example with zeros
    MqlDateTime mdt[];

    // allocating memory for an array of structures with results
    ArrayResize(mdt, n);

    // call our ArrayInitialize overload
    ArrayInitialize(mdt, null);

    // run tests
    for(int i = 0; i < n; ++i)
    {
        PRT(time[i]); // displaying initial data

        if(!TimeToStruct(time[i], mdt[i])) // if an error occurs, output its code
        {
            Print("error: ", _LastError);
            mdt[i].year = _LastError;
        }
    }

    // output the results to the log
    ArrayPrint(mdt);
    ...
}

```

结果，我们在日志中获得以下字符串：

```

time[i]=2021.01.28 23:00:15
time[i]=3000.12.31 23:59:59
time[i]=wrong datetime
wrong datetime -> 4010

```

	[year]	[mon]	[day]	[hour]	[min]	[sec]	[day_of_week]	[day_of_year]
[0]	2021	1	28	23	0	15	4	27
[1]	3000	12	31	23	59	59	3	364
[2]	4010	0	0	0	0	0	0	0

你可确保所有字段已接收到适当的值。对于不正确初始日期，我们将错误代码保存在 **year** 字段（在此情况下，仅有一个此类错误：4010, ERR_INVALID_DATETIME）。

别忘了，针对 MQL5 中最大日期值，引入了 `DATETIME_MAX` 常量，等于整数值 `0x793406fff`，其对应于 3000 年 12 月 31 日 23:59:59。

使用 ***TimeToStruct*** 函数解决的最常见问题，是获取特定日期/时间分量的值。因此，应准备一个具有就绪实施选项的辅助头文件 (***MQL5Book/DateTime.mqh***)。该文件具有 ***datetime*** 类。

```

class DateTime
{
private:
    MqlDateTime mdtstruct;
    datetime origin;

    DateTime() : origin(0)
    {
        TimeToStruct(0, mdtstruct);
    }

    void convert(const datetime &dt)
    {
        if(origin != dt)
        {
            origin = dt;
            TimeToStruct(dt, mdtstruct);
        }
    }

public:
    static DateTime *assign(const datetime dt)
    {
        _DateTime.convert(dt);
        return &_DateTime;
    }
    ENUM_DAY_OF_WEEK timeDayOfWeek() const
    {
        return (ENUM_DAY_OF_WEEK)mdtstruct.day_of_week;
    }
    int timeDayOfYear() const
    {
        return mdtstruct.day_of_year;
    }
    int timeYear() const
    {
        return mdtstruct.year;
    }
    int timeMonth() const
    {
        return mdtstruct.mon;
    }
    int timeDay() const
    {
        return mdtstruct.day;
    }
    int timeHour() const
    {
        return mdtstruct.hour;
    }
    int timeMinute() const

```

```

    {
        return mdtstruct.min;
    }
    int timeSeconds() const
    {
        return mdtstruct.sec;
    }

    static DateTime _DateTime;
};

static DateTime DateTime::_DateTime;

```

该类具有若干个宏，更方便调用其方法。

```

#define TimeDayOfWeek(T) DateTime::assign(T).timeDayOfWeek()
#define TimeDayOfYear(T) DateTime::assign(T).timeDayOfYear()
#define TimeYear(T) DateTime::assign(T).timeYear()
#define TimeMonth(T) DateTime::assign(T).timeMonth()
#define TimeDay(T) DateTime::assign(T).timeDay()
#define TimeHour(T) DateTime::assign(T).timeHour()
#define TimeMinute(T) DateTime::assign(T).timeMinute()
#define TimeSeconds(T) DateTime::assign(T).timeSeconds()

#define _TimeDayOfWeek DateTime::_DateTime.timeDayOfWeek
#define _TimeDayOfYear DateTime::_DateTime.timeDayOfYear
#define _TimeYear DateTime::_DateTime.timeYear
#define _TimeMonth DateTime::_DateTime.timeMonth
#define _TimeDay DateTime::_DateTime.timeDay
#define _TimeHour DateTime::_DateTime.timeHour
#define _TimeMinute DateTime::_DateTime.timeMinute
#define _TimeSeconds DateTime::_DateTime.timeSeconds

```

该类具有 *MqlDateTime* 结构体类型的 *mdtstruct* 字段。该字段用于所有内部转换中。结构体字段使用 *getter* 方法读取：为每个字段分配了对应方法。

类中定义了一个静态实例：*_DateTime*（一个对象已足够，因为所有 MQL 程序均为单线程）。构造函数为私有，因此试图创建其它 *datetime* 对象将会失败。

使用宏，我们可以从 *datetime* 方便地获取单独的组成部分，如年 (*TimeYear(T)*)、月 (*TimeMonth(T)*)、日 (*TimeDay(T)*)，或星期几 (*TimeDayOfWeek(T)*)。

如果从一个 *datetime* 值获取，则需要获取若干字段，最好在所有调用中使用相似的宏（第一个不带参数并以下划线符号开头的调用除外）：它们从结构体中读取所需的字段，无需重置日期/时间，也无需调用 *TimeToStruct* 函数。例如：

```
// use the DateTime class from MQL5Book/DateTime.mqh:
// first get the day of the week for the specified datetime value
PRT(EnumToString(TimeDayOfWeek(time[0])));
// then read year, month and day for the same value
PRT(_TimeYear());
PRT(_TimeMonth());
PRT(_TimeDay());
```

以下字符串应出现在日志中。

```
EnumToString(DateTime::_DateTime.assign(time[0]).__TimeDayOfWeek())=THURSDAY
DateTime::_DateTime.__TimeYear()=2021
DateTime::_DateTime.__TimeMonth()=1
DateTime::_DateTime.__TimeDay()=28
```

内置函数 *EnumToString* 可将任何一个枚举元素转换为一个字符串。这将在 [单独章节](#) 中介绍。

Datetime StructToTime(MqlDateTime &struct)

StructToTime 函数可将包含日期和时间组成部分的 *MqlDateTime* 结构体 (参见上述 *TimeToStruct* 函数) 转换为 *datetime* 类型的值。*day_of_week* 和 *day_of_year* 字段未使用。

如果其余字段的状态为无效 (对应于不存在或不支持的日期), 则函数可能返回不正确的值, 或者返回 *WRONG_VALUE* (*long* 类型表示的 -1), 视问题而定。因此, 应基于全局变量 *_LastError* 的状态检查是否存在错误。成功完成的转换显示代码 0。在转换之前, 应重置 *_LastError* 中的可能失败状态 (作为某些先前指令执行的遗留物), 可使用 *ResetLastError* 函数进行重置。

StructToTime 函数测试也在脚本 *ConversionTimeStruct.mq5* 中提供。结构体数组 *parts* 转换为循环中的 *datetime*。

```
MqlDateTime parts[] =
{
    {0, 0, 0, 0, 0, 0, 0, 0, 0},
    {100, 0, 0, 0, 0, 0, 0, 0, 0},
    {2021, 2, 30, 0, 0, 0, 0, 0, 0},
    {2021, 13, -5, 0, 0, 0, 0, 0, 0},
    {2021, 50, 100, 0, 0, 0, 0, 0, 0},
    {2021, 10, 20, 15, 30, 155, 0, 0, 0},
    {2021, 10, 20, 15, 30, 55, 0, 0, 0},
};
ArrayPrint(parts);
Print("");

// convert all elements in the loop
for(int i = 0; i < sizeof(parts) / sizeof(MqlDateTime); ++i)
{
    ResetLastError();
    datetime result = StructToTime(parts[i]);
    Print("[", i, "] ", (long)result, " ", result, " ", _LastError);
}
```

对于每个元素, 显示结果值和错误代码。

	[year]	[mon]	[day]	[hour]	[min]	[sec]	[day_of_week]	[day_of_year]
[0]	0	0	0	0	0	0	0	0
[1]	100	0	0	0	0	0	0	0
[2]	2021	2	30	0	0	0	0	0
[3]	2021	13	-5	0	0	0	0	0
[4]	2021	50	100	0	0	0	0	0
[5]	2021	10	20	15	30	155	0	0
[6]	2021	10	20	15	30	55	0	0

```
[0] -1 wrong datetime 4010
[1] 946684800 2000.01.01 00:00:00 4010
[2] 1614643200 2021.03.02 00:00:00 0
[3] 1638316800 2021.12.01 00:00:00 4010
[4] 1640908800 2021.12.31 00:00:00 4010
[5] 1634743859 2021.10.20 15:30:59 4010
[6] 1634743855 2021.10.20 15:30:55 0
```

请注意，该函数会在不生成错误标记的情况下更正某些值。在第 2 个元素中，我们将日期 2021 年 2 月 30 日传递进函数，函数会将其转换为 2021 年 3 月 2 日，并且 `_LastError = 0`。

4.1.4 颜色

SQL5 API 包含用于处理颜色的 3 个内置函数：其中两个用于 [颜色](#) 类型与字符串之间的相互转换，第三个用于提供带透明度的特殊颜色表示 (ARGB)。

string ColorToString(color value, bool showName = false)

`ColorToString` 函数将传递的颜色 `value` 转换为诸如 “R,G,B” 的字符串（其中 R、G、B 是 0 到 255 的数字，对应于颜色中红、绿、蓝分量的强度），或者对应于预定义 [web 颜色](#) 列表中的颜色名称（如果 `showName` 参数等于 `true`）。仅当颜色值完全匹配某个 web 设置颜色值时，才会返回颜色名称。

该函数的使用示例在 `ConversionColor.mq5` 脚本中提供。

```
void OnStart()
{
    Print(ColorToString clrBlue)); // 0,0,255
    Print(ColorToString(C'0, 0, 255', true)); // clrBlue
    Print(ColorToString(C'0, 0, 250')); // 0,0,250
    Print(ColorToString(C'0, 0, 250', true)); // 0,0,250 (no name for this color)
    Print(ColorToString(0x34AB6821, true)); // 33,104,171 (0x21,0x68,0xAB)
}
```

color StringToColor(string text)

`StringToColor` 函数将诸如 “R,G,B” 的字符串或者包含标准 [web 颜色](#) 的字符串转换为 [颜色](#) 类型值。如果字符串未包含格式正确的数字三元组或颜色名称，则函数将返回 0 (`clrBlack`)。

示例参见脚本 `ConversionColor.mq5`。

```

void OnStart()
{
    Print(StringToColor("0,0,255")); // clrBlue
    Print(StringToColor("clrBlue")); // clrBlue
    Print(StringToColor("Blue"));    // clrBlack (no color with that name)
    // extra text will be ignored
    Print(StringToColor("255,255,255 more text")); // clrWhite
    Print(StringToColor("This is color: 128,128,128")); // clrGray
}

```

uint ColorToARGB(color value, uchar alpha = 255)

ColorToARGB 函数将 *color* 类型值以及单字节值 *alpha* (指定透明度) 转换为 ARGB 颜色表示 (*uint* 类型值)。ARGB 颜色格式用于创建 [图形资源](#) 以及 [文本图](#) (在 [图表](#) 上)。

alpha 值介于 0 到 255 之间。“0”对应全颜色透明度 (当显示此颜色的像素时, 会让该点的现有图像保持不变), 255 表示应用全颜色浓度 (当显示该颜色的像素时, 会完全替换对应点的图像颜色)。值 128 (0x80) 为半透明。

如我们所知, *color* 类型使用三个颜色分量描述颜色: 红色 (Red)、绿色 (Green) 以及蓝色 (Blue), 分量以 0x00BBGGRR 的格式存储在 4 字节整数 (*uint*) 中。每个分量是一个指定对应颜色饱和度的字节, 范围 0 到 255 (十六进制则为 0x00 到 0xFF)。最高字节为空。例如, 白色包含所有颜色, 因此白色的 *color* 含义等于 0xFFFFFFFF。

但在某些任务中, 需要指定颜色透明度, 以描述当叠加在某个背景 (另一个已存在的图像) 上时的图像显示方式。为此, 引入了 *alpha* 通道的概念, 该通道由一个额外字节编码。

含 *alpha* 通道 (以 AA 表示) 的 ARGB 颜色表示为 0xAARRGGBB。例如, 值 0x80FFFF00 表示黄色 (红色和绿色分量混合) 半透明颜色。

当具有 *alpha* 通道的图像叠加到某个背景上时, 获得最终的颜色:

$$C_{result} = (C_{foreground} * \alpha + C_{background} * (255 - \alpha)) / 255$$

其中 C 分别取每个 R、G、B 分量的值。该公式供参考。使用具有 ARGB 颜色的内置函数时, 自动应用透明度。

ColorToARGB 应用示例在 *ConversionColor.mq5* 中提供。辅助结构体 *Argb* 和联合体 *ColorARGB* 已添加到脚本中, 以便于分析颜色分量。


```

struct Argb
{
    uchar BB;
    uchar GG;
    uchar RR;
    uchar AA;
};

union ColorARGB
{
    uint value;
    uchar channels[4]; // 0 - BB, 1 - GG, 2 - RR, 3 - AA
    Argb split[1];
    ColorARGB(uint u) : value(u) { }
};

```

该结构体用作联合体中的 *split* 类型字段，并提供按名称的 ARGB 分量访问。该联合体也具有字节数组 *channels*，允许你按索引访问分量。

```

void OnStart()
{
    uint u = ColorToARGB(cclrBlue);
    PrintFormat("ARGB1=%X", u); // ARGB1=FF0000FF
    ColorARGB clr1(u);
    ArrayPrint(clr1.split);
    /*
        [BB] [GG] [RR] [AA]
    [0]  255   0   0  255
    */

    u = ColorToARGB(cclrDeepSkyBlue, 0x40);
    PrintFormat("ARGB2=%X", u); // ARGB2=4000BFFF
    ColorARGB clr2(u);
    ArrayPrint(clr2.split);
    /*
        [BB] [GG] [RR] [AA]
    [0]  255  191   0   64
    */
}

```

稍后我们讨论 *print format* 函数（在相应 章节 讨论）。

没有内置函数可将 ARGB 转换回 *color*（因为通常不需要），但如果希望这样做，可使用以下宏：

```

#define ARGBToColor(U) (color) \
    (((U) & 0xFF) << 16) | ((U) & 0xFF00) | (((U) >> 16) & 0xFF)

```

4.1.5 结构体

当将 MQL 程序与外部系统集成时，尤其是当通过互联网发送或接收数据时，有必要将数据结构体转换为字节数组。为此，MQL5 API 提供了两个函数：*StructToCharArray* 和 *CharArrayToStruct*。

在两种情况下，假定结构体仅包含简单 [内置类型](#)，也就是除了 [行](#) 和 [动态数组](#) 外的所有内置类型。一个结构体也可能包含其它简单结构体。不允许包含类对象和指针。这种结构体也称为简单旧数据 (POD)。

bool StructToCharArray(const void &object, uchar &array[], uint pos = 0)

StructToCharArray 函数将 POD 结构体 *object* 复制到 *uchar* 类型的 *array* 数组。或者，使用参数 *pos* 可指定在数组中放置字节的开始位置。默认情况下，复制从数组开始位置进行，如果其当前大小不足以容纳整个结构体，则动态数组将自动增加大小。

函数返回成功指示器 (*true*) 或错误 (*false*)。

我们使用脚本 *ConversionStruct.mq5* 检查其执行。我们创建新的结构体类型 *DateTimeMsc*，其包括标准结构体 *MqlDateTime* (*mdt* 字段) 和 *int* 类型的附加字段 *msc* 以存储毫秒。

```
struct DateTimeMsc
{
    MqlDateTime mdt;
    int msc;
    DateTimeMsc(MqlDateTime &init, int m = 0) : msc(m)
    {
        mdt = init;
    }
};
```

在 *OnStart* 函数中，我们将测试值 *datetime* 转换为我们的结构体，然后转换为字节数组。

```
MqlDateTime TimeToStructInplace(datetime dt)
{
    static MqlDateTime m;
    if(!TimeToStruct(dt, m))
    {
        // the error code, _LastError, can be displayed
        // but here we just return zero time
        static MqlDateTime z = {};
        return z;
    }
    return m;
}

#define MDT(T) TimeToStructInplace(T)

void OnStart()
{
    DateTimeMsc test(MDT(D'2021.01.01 10:10:15'), 123);
    uchar a[];
    Print(StructToCharArray(test, a));
    Print(ArraySize(a));
    ArrayPrint(a);
}
```

我们将在日志中获得以下结果（数组被重新设置格式，带有附加换行符以强调字节与每个字段的对应性）：

```

true
36
229  7  0  0
  1  0  0  0
  1  0  0  0
 10  0  0  0
 10  0  0  0
 15  0  0  0
  5  0  0  0
  0  0  0  0
123  0  0  0

```

bool CharArrayToStruct(void &object, const uchar &array[], uint pos = 0)

CharArrayToStruct 函数将 *uchar* 类型的 *array* 数组复制到 POD 结构体 *object*。使用 *pos* 参数可指定在数组中开始读取字节的位置。

函数返回成功指示器 (*true*) 或错误 (*false*)。

继续相同示例 (*ConversionStruct.mq5*)，我们可从字节数组还原原始日期和时间。

```

void OnStart()
{
    ...
    DateTimeMsc receiver;
    Print(CharArrayToStruct(receiver, a));           // true
    Print(StructToTime(receiver.mdt), "'", receiver.msc); // 2021.01.01 10:10:15'123
}

```

4.1.6 枚举

在 MQL5 API 中，可使用 *EnumToString* 函数将枚举值转换为字符串。没有现成的逆转换。

string EnumToString(enum value)

该函数将任何类型的枚举的值（即传递元素的 ID）转换为字符串。

我们使用该函数来解决最常见的任务之一：找出枚举大小（其包含多少个元素）以及所有元素准确对应的值。为此，在头文件 *EnumToArray.mqh* 中，我们实现特殊 **模板函数**（由于模板类型为 E，因此适用于任何枚举）：

```

template<typename E>
int EnumToArray(E dummy, int &values[],
    const int start = INT_MIN,
    const int stop = INT_MAX)
{
    const static string t = "::";

    ArrayResize(values, 0);
    int count = 0;

    for(int i = start; i < stop && !IsStopped(); i++)
    {
        E e = (E)i;
        if(StringFind(EnumToString(e), t) == -1)
        {
            ArrayResize(values, count + 1);
            values[count++] = i;
        }
    }
    return count;
}

```

其运算概念基于以下原理。由于 MQL5 中的枚举存储为 `int` 类型的整数，支持将任何枚举隐式强制转换为 (`int`)，同时也允许将 `int` 隐式强制转换为任何枚举类型。在此情况下，如果值对应于枚举的某个元素，则 `EnumToString` 函数返回一个带有该元素 ID 的字符串。否则，该函数返回一个 `ENUM_TYPE::value` 格式的字符串。

因此，通过在可接受范围内的整数之间循环并将这些整数显式强制转换为一种枚举类型，即可分析输出字符串 `EnumToString` 中是否存在 `'::'`，以确定给定整数是否是枚举成员。

与其它字符串函数一样，此处使用的 `StringFind` 函数将在 [下一章](#)中讲解。

我们创建 `ConversionEnum.mq5` 脚本以测试概念。在脚本中，我们实现辅助函数 `process`，该函数将调用 `EnumToArray` 模板，报告枚举中元素的数量，并打印生成的数组（其中包含枚举元素的值匹配的项）。

```

template<typename E>
void process(E a)
{
    int result[];
    int n = EnumToArray(a, result, 0, USHORT_MAX);
    Print(typename(E), " Count=", n);
    for(int i = 0; i < n; i++)
    {
        Print(i, " ", EnumToString((E)result[i]), "=", result[i]);
    }
}

```

为了研究枚举，我们将使用具有 `ENUM_APPLIED_PRICE` 价格类型的内置枚举。在 `OnStart` 函数内，我们首先确保 `EnumToString` 生成如上所述的字符串。因此，对于元素 `PRICE_CLOSE`，函数将返回字符串 `"PRICE_CLOSE"`，而对于明显超出范围的值 `(ENUM_APPLIED_PRICE)10`，则将返回 `"ENUM_APPLIED_PRICE::10"`。

```

void OnStart()
{
    PRT(EnumToString(PRICE_CLOSE));           // PRICE_CLOSE
    PRT(EnumToString((ENUM_APPLIED_PRICE)10)); // ENUM_APPLIED_PRICE::10

    process((ENUM_APPLIED_PRICE)0);
}

```

接下来，我们调用函数 `process` 以将任何值强制转换为 `ENUM_APPLIED_PRICE` (或该类型的变量) 并且获得以下结果：

```

ENUM_APPLIED_PRICE Count=7
0 PRICE_CLOSE=1
1 PRICE_OPEN=2
2 PRICE_HIGH=3
3 PRICE_LOW=4
4 PRICE_MEDIAN=5
5 PRICE_TYPICAL=6
6 PRICE_WEIGHTED=7

```

我们看到枚举中定义了 7 个元素，并且编号不是从通常的 0 开始，而是从 1 (`PRICE_CLOSE`) 开始。了解与元素关联的值在某些情况下可优化算法编写。

4.1.7 复数类型

内置 `complex` 类型是一种具有两个 *双精度浮点数* 类型字段的结构体：

```

struct complex
{
    double    real;    // real part
    double    imag;    // imaginary part
};

```

该结构体在类型转换章节中介绍过，因为它将两个 `double` 数字“转换”为一个新实体，类似于 *结构体转换为字节数组及相反转换* 的方式。此外，如果不先描述结构体，读者可能很难理解这个类型。

`complex` 结构体没有构造函数，因此复数必须使用初始化列表创建。

```
complex c = {re, im};
```

对于复数，目前只能执行简单算术和比较运算：`=`、`+`、`-`、`*`、`/`、`+=`、`-=`、`*=`、`/=`、`==`、`!=`。对于 *数学函数* 的支持将在今后增加。

注意！不能将复数变量声明为 MQL 程序的输入 (使用 `input` 关键字)。

后缀 `i` 用于描述复数 (虚数部分) 常量，例如：

```

const complex x = 1 - 2i;
const complex y = 0.5i;

```

在下面的例子 (脚本 `Complex.mq5`) 中，创建了一个复数并计算了其平方值。

```

input double r = 1;
input double i = 2;

complex c = {r, i};

complex mirror(const complex z)
{
    complex result = {z.imag, z.real}; // swap real and imaginary parts
    return result;
}

complex square(const complex z)
{
    return (z * z);
}

void OnStart()
{
    Print(c);
    Print(square(c));
    Print(square(mirror(c)));
}

```

如果采用默认参数，该脚本将输出以下结果：

```

c=(1,2) / ok
square(c)=(-3,4) / ok
square(mirror(c))=(3,4) / ok

```

其中，圆括号中的数字对是该复数的字符串表示。

complex 类型可通过值传递为 MQL 函数的一个参数（不同于普通结构体，普通结构体仅可通过引用传递）。对于从 DLL 导入的函数，**complex** 类型应仅通过引用传递。

4.2 处理字符串和符号

虽然计算机的名称来源于动词“计算”，但它们不仅可以成功处理数字，同样可以成功处理任何非结构化信息，最典型的例子是文本。在 MQL 程序中，文本的应用也无处不在，从程序本身的名称，到交易订单的备注等。为了在 MQL5 中处理文本，提供了内置 **字符串类型**，其允许操作任意长度的字符系列。

为了执行典型字符串操作，MQL5 API 提供了可根据其用途而条件性划分为组的各种函数，诸如字符串初始化、字符串添加、字符串内片段搜索和替换、字符串转换为字符数组、访问单个字符以及设置格式等。

本章中的大部分函数返回执行状态指示：成功或错误。对于结果类型为 **bool** 的函数，**true** 通常表示成功，**false** 表示错误。对于结果类型为 **int** 的函数，值 0 或 -1 可视为错误：这在每个函数的说明中声明。在所有这些情况下，开发者可找出问题核心所在。为此，可调用 **GetLastError** 函数并获取具体错误代码：文档中提供了所有代码的列表，并附带解释。在接收了错误标记后须立即调用 **GetLastError**，这一点很重要，因为若继续调用算法中每个后续指令可能导致另一个错误。

4.2.1 字符串初始化和测量

我们从 [字符串类型](#) 章节已经知道，在代码中只需描述一个 **string** 类型的变量，该代码将可以执行。

对于 **string** 类型的任何变量，为服务结构体分配 12 个字节，该服务结构体是该字符串的内部表示。该结构体包含文本存储所在的内存地址（指针）信息以及一些其它元信息。文本本身也要求充足的内存，但该缓冲区的分配没有采用多少优化措施。

具体而言，我们可以描述字符串及其显式初始化，包括空字面量：

```
string s = ""; // pointer to the literal containing '\0'
```

在这种情况下，指针将直接设置到该字面量，没有为该缓冲区分配内存（即使该字面量较长）。很明显，已为该字面量分配了静态内存，可直接使用。除非程序中的任何指令更改了该行的内容，否则将为缓冲区分配内存。例如（注意字符串允许加法运算 '+'）：

```
int n = 1;
s += (string)n; // pointer to memory containing "1""\0"[plus reserve]
```

此后，字符串实际上包含文本 “1”，并且严格来说，需要占用两个字符的内存：数字 “1” 和隐式终结符零 ‘\0’（该字符串的终结符）。然而，系统将分配一个较大的缓冲区，其中预留了某些空间。

若声明一个没有初始值的变量，编译器仍然隐式初始化该变量，虽然在这种情况下是以特殊 NULL 值初始化：

```
string z; // memory for the pointer is not allocated, pointer = NULL
```

这种字符串对于每个结构体仅需要 12 个字节，指针不指向任何地方：这就是 NULL 代表的含义。

在将来版本的 MQL5 编译器中，此行为可能更改，最初始终为空字符串分配小块内存，提供某些预留空间。

除了这些内部特性外，**string** 类型的变量与其它类型的变量没有区别。然而，由于字符串的长度可能变化，更重要的是，在运算期间它们可能更改其长度，这会对内存分配效率和性能造成不利影响。

例如，如果在某个点，程序需要向字符串添加一个新字，但可能为该字符串分配的内存却不够。然后，在用户觉察不到的情况下，MQL 程序执行环境将找到新的增加了大小的空闲内存块，并将旧值随添加的新字一同拷贝到其中。之后，旧地址被该行的服务结构体中的一个新地址所替换。

如果这样的操作数量众多，由于拷贝导致的性能降低可能变得明显，此外，程序内存会碎片化：拷贝后释放的旧的小内存区域会形成一些空隙，这些空隙不适合大字符串使用，从而导致内存浪费。当然，终端能够控制此类情况并重新整理内存，但这也是有代价的。

解决这一问题最有效的方法，是事先明确指定字符串的缓冲区大小，并使用内置 MQL5 API 函数将其初始化，这些函数将在本节后文中探讨。

这一优化的依据是，分配的内存大小可超过字符串的当前长度以及可能的将来长度，该长度由文本中的第一个空字符确定。这样，我们可以分配 100 个字符的缓冲区，但从一开始将 ‘\0’ 放在最开头，这将提供一个零长度字符串 (“”)。

当然，此类情况下假定编程人员能够事先粗略计算字符串的预期长度及其增长速度。

由于 MQL5 中的字符串是基于双字节字符 (其确保 Unicode 支持), 以字符数表示的字符串和缓冲区大小应乘以 2, 以获得占用内存量和分配内存量 (以字节数表示)。

本节末尾会提供使用所有函数的一般示例 (*StringInit.mq5*)。

bool StringInit(string &variable, int capacity = 0, ushort character = 0)

StringInit 函数用于初始化 (分配和填充内存) 和反初始化 (释放内存) 字符串。要处理的变量在第一个参数中传递。

如果 **capacity** 参数大于 0, 则为字符串分配指定大小的缓冲区 (内存区域), 并以 **character** 符号填充。如果 **character** 为 0, 则字符串长度将为零, 因为第一个字符是终止符。

如果 **capacity** 参数为 0, 则之前分配的内存被释放。如果变量刚刚声明但还未初始化, 变量的状态变为与声明之前的状态相同 (缓冲区的指针为 NULL)。实现相同效果更简单的方法是将字符串变量设置为 NULL。

函数返回成功指示器 (**true**) 或错误 (**false**)。

bool StringReserve(string &variable, uint capacity)

StringReserve 函数增加或减少 **variable** 字符串的缓冲区大小, 至少为 **capacity** 参数中指定的字符数。如果 **capacity** 值小于当前字符串长度, 则函数不执行任何操作。实际上, 缓冲区大小可能大于请求的大小: 环境如此分配是出于该字符串将来操纵效率的考虑。因此, 如果以小于缓冲区的值调用该函数, 该函数可能忽略请求, 仍然返回 **true** (“无错误”)。

当前缓冲区大小可使用 **StringBufferLen** 函数 (见下文) 获取。

如果成功, 函数返回 **true**, 否则返回 **false**。

不同于 **StringInit**, **StringReserve** 函数不更改字符串的内容, 也不会使用字符填充。

bool StringFill(string &variable, ushort character)

StringFill 函数使用 **character** 字符填充指定 **variable** 的整个当前长度 (直至第一个零)。如果为字符串分配了缓冲区, 修改原位完成, 没有中间换行和拷贝操作。

函数返回成功指示器 (**true**) 或错误 (**false**)。

int StringBufferLen(const string &variable)

函数返回为 **variable** 字符串分配的缓冲区大小。

请注意, 对于字面量初始化的字符串, 最初不会分配缓冲区, 因为指针指向该字面量。因此, 即使 **StringLen** 字符串 (见下文) 的长度可能更大, 该函数也将返回 0。

值 -1 表示该行属于客户端终端, 不能更改。

bool StringSetLength(string &variable, uint length)

该函数为 **variable** 字符串设置指定的字符长度 **length**。**length** 的值不得大于字符串的当前长度。换言之, 该函数只允许你缩短该字符串, 而不能加长该字符串。调用 **StringAdd** 函数或者执行了加法运算 '+' 时, 会自动增加该字符串的长度。

函数 **StringSetLength** 的等效执行是 **StringSetCharacter(variable, length, 0)** 调用 (参见 [处理符号和代码页章节](#))。

如果在函数调用前已为字符串分配了缓冲区，则该函数不会更改缓冲区。如果字符串没有缓冲区（其指向字面量），则减少长度会分配新缓冲区并将缩短的字符串拷贝到该缓冲区。

如果成功或失败，函数分别返回 *true* 或 *false*。

int StringLen(const string text)

该函数返回字符串 *text* 中的字符数。终止零字符不计入。

请注意，该参数由值传递，因此，你不仅可以计算变量中的字符串长度，而且可以计算任何其它中间值的字符串长度：计算结果值或字面量。

创建了 *StringInit.mq5* 脚本以演示上述函数。它使用特殊版本的 PRT 宏 (PRTE)，其将表达式的结果解析为 *true* 或 *false*，并且对于后者，还输出错误代码：

```
#define PRTE(A) Print(#A, "=", (A) ? "true" : "false:" + (string)GetLastError())
```

为了将字符串及其当前指标（行长度和缓冲区大小）调试输出到日志，实现了 *StrOut* 函数：

```
void StrOut(const string &s)
{
    Print("", s, " [" , StringLen(s), " ] ", StringBufferLen(s));
}
```

该函数使用内置 *StringLen* 和 *StringBufferLen* 函数。

测试脚本对 *OnStart* 中的一个字符串执行一系列操作：

```
void OnStart()
{
    string s = "message";
    StrOut(s);
    PRTE(StringReserve(s, 100)); // ok, but we get a buffer larger than requested: 260
    StrOut(s);
    PRTE(StringReserve(s, 500)); // ok, buffer is increased to 500
    StrOut(s);
    PRTE(StringSetLength(s, 4)); // ok: string is shortened
    StrOut(s);
    s += "age";
    PRTE(StringReserve(s, 100)); // ok: buffer remains at 500
    StrOut(s);
    PRTE(StringSetLength(s, 8)); // no: string lengthening is not supported
    StrOut(s); // via StringSetLength
    PRTE(StringInit(s, 8, '$')); // ok: line increased by padding
    StrOut(s); // buffer remains the same
    PRTE(StringFill(s, 0)); // ok: string collapsed to empty because
    StrOut(s); // was filled with 0s, the buffer is the same
    PRTE(StringInit(s, 0)); // ok: line is zeroed, including buffer
    // we could just write s = NULL;

    StrOut(s);
}
```

该脚本将记录以下消息：

```

'message' [7] 0
StringReserve(s,100)=true
'message' [7] 260
StringReserve(s,500)=true
'message' [7] 500
StringSetLength(s,4)=true
'mess' [4] 500
StringReserve(s,10)=true
'message' [7] 500
StringSetLength(s,8)=false:5035
'message' [7] 500
StringInit(s,8,'$')=true
'$$$$$$$$' [8] 500
StringFill(s,0)=true
'' [0] 500
StringInit(s,0)=true
'' [0] 0

```

请注意，若 `StringSetLength` 调用时增加字符串长度，则结束时会出现错误 5035 (ERR_STRING_SMALL_LEN)。

4.2.2 字符串连接

字符串连接可能是最常见的字符串运算。在 MQL5，可以使用 '+' 或 '+=' 运算符完成。第一个运算符连接两个字符串（+ 号左边和右边的两个操作数），并创建一个临时连接字符串，可为其赋予一个目标变量或传递给表达式的另一部分（诸如函数调用）。第二个运算符将运算符 '+' 右边的字符串附加到本运算符左边的字符串（变量）。

此外，MQL5 API 提供了用于从其它字符串或其它类型的元素组成字符串的若干个函数。

这些函数的使用示例在 `StringAdd.mq5` 脚本中提供，该脚本在其说明之后进行讲解。

bool StringAdd(string &variable, const string addition)

该函数将指定 *addition* 字符串附加到 *variable* 字符串变量的末尾。在可能的情况下，系统使用 *variable* 字符串的可用缓冲区（如果其大小足以用于组合结果），而不会重新分配内存或拷贝字符串。

该函数等效于运算符 *variable += addition*。时间和内存消耗大致相同。

如果成功，函数返回 *true*，如果出错，返回 *false*。

int StringConcatenate(string &variable, void argument1, void argument2 [, void argumentl...])

该函数将两个或多个 **内置类型** 的自变量转换为字符串表示，并将它们在 *variable* 字符串中连接起来。这些自变量从该函数的第二个参数开始传递。不支持将数组、结构体、对象、指针作为自变量。

自变量的数量必须在 2 到 63 之间。

自变量字符串按原样添加至生成的变量。

double 类型的自变量以最大精度（最多 16 位有效数字）转换，如果可以达到更紧凑的结果，可以选择带指数的科学计数法。*float* 类型的自变量以 5 个字符显示。

datetime 类型的值转换为包含所有日期和时间字段 (“YYYY.MM.DD hh:mm:ss”) 的字符串。

枚举、单字节和双字节字符输出为整数。

color 类型的值显示为 “R,G,B” 分量三元组或者颜色名称 (如果在标准 web 颜色列表中可用)。

转换 *bool* 类型时, 使用字符串 “true” 或 “false”。

StringConcatenate 函数返回生成字符串的长度。

StringConcatenate 设计用于从接收变量以外的其它来源 (变量、表达式) 构建字符串。不建议使用 *StringConcatenate* 通过调用 *StringConcatenate(variable, variable, ...)* 来将新的数据块连接到相同的行。此函数调用未经优化, 相比运算符 ‘+’ 和 *StringAdd* 来说要慢得多。

StringAdd 和 *StringConcatenate* 函数在 *StringAdd.mq5* 脚本中测试, 该脚本使用 PRTE 宏和 *StrOut* 辅助函数 (参见 上一节)。

```
void OnStart()
{
    string s = "message";
    StrOut(s);
    PRTE(StringAdd(s, "r"));
    StrOut(s);
    PRTE(StringConcatenate(s, M_PI * 100, " ", clrBlue, PRICE_CLOSE));
    StrOut(s);
}
```

其执行之后, 日志中会显示以下行:

```
'message' [7] 0
StringAdd(s,r)=true
'messenger' [8] 260
StringConcatenate(s,M_PI*100, ,clrBlue,PRICE_CLOSE)=true
'314.1592653589793 clrBlue1' [26] 260
```

该脚本还包含头文件 *StringBenchmark.mqh*, 其中具有 *benchmark* 类。它为在脚本中实现的派生类提供了用于测量各种字符串添加方法的性能的框架。尤其是, 它们确保了使用运算符 ‘+’ 添加字符串与使用 *StringAdd* 函数添加字符串具有可比性。这方面在单独的研究中探讨。

此外, 本书提供了 *StringReserve.mq5* 脚本: 它对使用或未使用缓存器 (*StringReserve*) 添加字符串的速度进行了直观比较。

4.2.3 字符串比较

要在 MQL5 中比较字符串, 可使用标准 [比较运算符](#), 尤其是 ‘==’、‘!=’、‘>’、‘<’。所有这些运算符以逐字符、区分大小写的方式执行比较。

每个字符都有一个 *ushort* 类型整数的 Unicode 码。相应地, 首先, 比较两个字符串的第一个字符的代码, 然后比较第二个字符的代码, 以此类推, 直至出现第一个不匹配项或达到其中一个字符串的末尾。

例如, “ABC” 字符串小于 “abc”, 因为在字符表中大写字母的代码低于相应小写字母的代码 (对于第一个字符, 我们已经得到 “A” < “a”)。如果字符串在开头具有匹配字符, 但其中一个长于另一个, 则较长的字符串被视为更大 (“ABCD” > “ABC”)。

这些字符串关系形成词典序。当字符串“A”小于字符串“B”(“A”<“B”),“A”被视为在“B”之前。

要熟悉字符代码,可使用标准 Windows 应用程序“字符映射表”。在该字符映射表中,字符按增加代码的顺序排列。除了包括多种国家语言的常规 Unicode 表,还有代码页:含单字节字符代码的 ANSI 标准表 – 对于每种语言或语言组都各不相同。我们将在以下章节详细探讨该问题: [处理符号和代码页](#)。

字符映射表的初始部分 (代码 0 到 127)对于所有语言均相同。该部分显示在下表中。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	non-printable								BS	HT	LF	VT	FF	CR		
1	control codes								back space	horiz. tab	line feed	vertical tab	form feed	carriage return		
2	_	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

ASCII 字符代码表

要获取字符代码,取左边的十六进制数字 (该字符所在行的行号),然后与顶部的数字 (该字符所在列的列号)相加:结果是一个十二进制数字。例如,对于 ‘!’,左边为 2,顶上为 1,意味着字符代码为 0x21 或 33 (十进制)。

至 32 的代码为控制代码。其中,尤其你可以找到制表符 (代码 0x9)、换行 (代码 0xA)以及回车 (代码 0xD)。

Windows 文本文件中使用彼此跟随的一对字符 0xD 0xA,用于中断至新行。我们在 [字符类型](#) 章节熟悉了相应的 MQL5 字面量:0xA 可表示为 ‘\n’,而 0xD 表示为 ‘\r’。制表符 0x9 也有自己的表示方法:‘\t’。

MQL5 API 提供了 *StringCompare* 函数,可用于在比较字符串时禁用区分大小写。

int StringCompare(const string &string1, const string &string2, const bool case_sensitive = true)

该函数比较两个字符串并返回三个值之一:如果第一个字符串“大于”第二个,则返回 +1;如果字符串“相等”,则返回 0;如果第一个字符串“小于”第二个,则返回 -1。“大于”、“小于”和“等于”的概念取决于 *case_sensitive* 参数。

若 *case_sensitive* 参数等于 *true* (默认),比较区分大小写,大写字母视为大于对应小写字母。这与根据字符代码的标准词典序相反。

若区分大小写, *StringCompare* 函数使用不同于词典序的大小写字母序。例如,我们知道关系 “A”<“a”为 *true*,其中运算符 ‘<’ 以字符代码为依据。因此,在假设的字典 (数组)中,首字母大写的单词应出现在具有相同小写字母的单词之前。但是,当使用 *StringCompare*(“A”, “a”) 函数比较 “A”和 “a”时,我们得到 +1,表示 “A”大于 “a”。因此,在排序词典中,以小写字母开始的单词将在先,而以大写字母开始的单词在后。

换言之，该函数按字母顺序对字符串排序。此外，在区分大小写模式下，还有一条额外规则：如果字符串仅存在大小写的差别，包含大写字母的字符串会排在对应小写字母字符串的后面（在单词中相同位置的情况下）。

如果 `case_sensitive` 参数等于 `false`，则字母不区分大小写，因此字符串“A”和“a”视为相等，函数返回 0。

可以使用 `StringCompare.mq5` 脚本检查由 `StringCompare` 函数和运算符比较的不同比较结果。

```
void OnStart()
{
    PRT(StringCompare("A", "a"));           // 1, which means "A" > "a" (!)
    PRT(StringCompare("A", "a", false));    // 0, which means "A" == "a"
    PRT("A" > "a");                         // false, "A" < "a"

    PRT(StringCompare("x", "y"));           // -1, which means "x" < "y"
    PRT("x" > "y");                         // false, "x" < "y"
    ...
}
```

在 [函数模板](#) 章节中，我们创建了模板化快速排序算法。我们将其转换为模板类，并将其用于若干排序选择：在区分和不区分大小写的情况下使用比较运算符以及使用 `StringCompare` 函数。我们将新的 `QuickSortT` 类放入 `QuickSortT.mqh` 头文件，并将其连接到 `StringCompare.mq5` 测试脚本。

排序 API 几乎没有变化。

```

template<typename T>
class QuickSortT
{
public:
    void Swap(T &array[], const int i, const int j)
    {
        ...
    }

    virtual int Compare(T &a, T &b)
    {
        return a > b ? +1 : (a < b ? -1 : 0);
    }

    void QuickSort(T &array[], const int start = 0, int end = INT_MAX)
    {
        ...
        for(int i = start; i <= end; i++)
        {
            //if(!(array[i] > array[end]))
            if(Compare(array[i], array[end]) <= 0)
            {
                Swap(array, i, pivot++);
            }
        }
        ...
    }
};

```

主要的不同之处在于我们添加了 *Compare* 虚方法，其默认包含使用 ‘>’ 和 ‘<’ 运算符的比较，并以与 *StringCompare* 相同的方式返回 +1、-1 或 0。*Compare* 方法现在用于 *QuickSort* 方法中，代替简单比较，必须在子类中被重写，以使用 *StringCompare* 函数或任何其它比较方法。

尤其在 *StringCompare.mq5* 文件中，我们实现了派生自 *QuickSortT<string>* 的以下 “比较器”类：

```

class SortingStringCompare : public QuickSortT<string>
{
    const bool caseEnabled;
public:
    SortingStringCompare(const bool sensitivity = true) :
        caseEnabled(sensitivity) { }

    virtual int Compare(string &a, string &b) override
    {
        return StringCompare(a, b, caseEnabled);
    }
};

```

构造函数采用 1 个参数，用于指定字符串比较符号，同时考虑 (*true*) 或忽略 (*false*) 寄存器。字符串比较本身在重新定义的 *Compare* 虚方法中完成，该方法调用具有给定自变量和设置的 *StringCompare* 函数。

为测试排序，我们需要一系列组合了大小写字母的字符串。我们可以自行生成这些字符串：只需开发一个类，对预定义集合（字母表）中的字符按给定集合长度（字符串）执行排列组合（可重复）。例如，可以限定使用小写字母表“abcABC”，即大小写的前三个英语字母，然后据此生成所有可能的 2 个字符的字符串。

PermutationGenerator 类在 *PermutationGenerator.mqh* 文件中提供，稍后再单独探讨。在这里我们仅了解其公共接口。

```
class PermutationGenerator
{
public:
    struct Result
    {
        int indices[]; // indexes of elements in each position of the set, i.e.
    };
    // for example, the numbers of the letters of the "alphabet" in
    PermutationGenerator(const int length, const int elements);
    SimpleArray<Result> *run();
};
```

创建生成器对象时，必须指定生成的集合的长度 *length* 在本例中，这将是字符串的长度，比如 2) 以及将组成集合的不同元素的数量（在本例中，这是唯一字母的数量，也就是 6）。利用这些输入数据，应可获得 $6 * 6 = 36$ 的行变体。

该过程本身由 *run* 方法执行。模板类用于返回结果为 *SimpleArray* 的数组，这已在 [方法模板](#) 章节讨论过。在此情况下，其由 *result* 结构体类型进行参数化。

GenerateStringList 辅助函数负责调用生成器，并根据从生成器收到的排列数组（以所有可能字符串各位置的字母索引形式表示）实际创建字符串。

```
void GenerateStringList(const string symbols, const int len, string &result[])
{
    const int n = StringLen(symbols); // alphabet length, unique characters
    PermutationGenerator g(len, n);
    SimpleArray<PermutationGenerator::Result> *r = g.run();
    ArrayResize(result, r.size());
    // loop through all received character permutations
    for(int i = 0; i < r.size(); ++i)
    {
        string element;
        // loop through all characters in the string
        for(int j = 0; j < len; ++j)
        {
            // add a letter from the alphabet (by its index) to the string
            element += ShortToString(symbols[r[i].indices[j]]);
        }
        result[i] = element;
    }
}
```

我们使用几个还不熟悉的函数 (*ArrayResize*, *ShortToString*)，但我们将很快就会了解。目前，我们只需知道，*ShortToString* 函数返回由该单个字符基于 *ushort* 类型字符代码组成的字符串。通过 *+=* 运算符，我们将这些单字符字符串拼接成最终的字符串。别忘了，为字符串定义了运算符 *[]*，因此，*symbols[k]* 将返回 *symbols* 字符串的第 *k* 个字符。当然，*k* 也可反过来成为整数表达

式，并且这里的 `r[i].indices[j]` 指的是 `r` 数组的第 `i` 个元素，从该数组读取了字符串的第 `j` 位置对应的“字母”字符索引。

接收到的每个字符串存储在 `result` 数组参数中。

我们在 `OnStart` 函数中应用这一信息。

```
void OnStart()
{
    ...
    string messages[];
    GenerateStringList("abcABC", 2, messages);
    Print("Original data", ArraySize(messages), "]:");
    ArrayPrint(messages);

    Print("Default case-sensitive sorting:");
    QuickSortT<string> sorting;
    sorting.QuickSort(messages);
    ArrayPrint(messages);

    Print("StringCompare case-insensitive sorting:");
    SortingStringCompare caseOff(false);
    caseOff.QuickSort(messages);
    ArrayPrint(messages);

    Print("StringCompare case-sensitive sorting:");
    SortingStringCompare caseOn(true);
    caseOn.QuickSort(messages);
    ArrayPrint(messages);
}
```

脚本首先将所有字符串选择纳入消息数组，然后以 3 种模式排序：使用内置比较运算符、使用 `StringCompare` 函数且不区分大小写、使用该函数且区分大小写。

我们将得到以下日志输出：

```
Original data[36]:
[ 0] "aa" "ab" "ac" "aA" "aB" "aC" "ba" "bb" "bc" "bA" "bB" "bC" "ca" "cb" "cc" "cA"
[18] "Aa" "Ab" "Ac" "AA" "AB" "AC" "Ba" "Bb" "Bc" "BA" "BB" "BC" "Ca" "Cb" "Cc" "CA"
Default case-sensitive sorting:
[ 0] "AA" "AB" "AC" "Aa" "Ab" "Ac" "BA" "BB" "BC" "Ba" "Bb" "Bc" "CA" "CB" "CC" "Ca"
[18] "aA" "aB" "aC" "aa" "ab" "ac" "bA" "bB" "bC" "ba" "bb" "bc" "cA" "cB" "cC" "ca"
StringCompare case-insensitive sorting:
[ 0] "AA" "Aa" "aA" "aa" "AB" "aB" "Ab" "ab" "aC" "AC" "Ac" "ac" "BA" "Ba" "bA" "ba"
[18] "Bb" "bb" "bC" "BC" "Bc" "bc" "CA" "Ca" "cA" "ca" "CB" "cB" "Cb" "cb" "cC" "CC"
StringCompare case-sensitive sorting:
[ 0] "aa" "aA" "Aa" "AA" "ab" "aB" "Ab" "AB" "ac" "aC" "Ac" "AC" "ba" "bA" "Ba" "BA"
[18] "Bb" "BB" "bc" "bC" "Bc" "BC" "ca" "cA" "Ca" "CA" "cb" "cB" "Cb" "CB" "cc" "cC"
```

输出显示这三种模式的差异。

4.2.4 更改字符大小写和修剪空格

处理文本通常意味着使用某些标准运算，诸如将所有字符转换为大写或小写，以及删除字符串开头或末尾的多余空字符（如空格）等。为此，MQL5 API 提供了四个对应的函数。所有这些函数均原位修改字符串，即直接在可用缓冲区中（如果已分配）修改。

所有函数的输入参数均是对字符串的引用，也就是说，只能为其传递变量（而不是表达式），不能传递常量变量，因为这些函数涉及修改自变量。

所有这些函数的测试脚本遵循相关说明。

bool StringToLower(string &variable)

bool StringToUpper(string &variable)

这些函数将指定字符串的所有字符转换为适当大小写：**StringToLower** 用于转换为小写字母，**StringToUpper** 用于转换为大写字母。支持在 Windows 系统层面可用的国家语言。

如果成功，其返回 *true*。如果出错，其返回 *false*。

int StringTrimLeft(string &variable)

int StringTrimRight(string &variable)

该函数字符串开头（对于 **StringTrimLeft**）或末尾（对于 **StringTrimRight**）的回车（`\r`）、换行（`\n`）、空格（）、制表符（`\t`）以及一些其它不可显示字符。如果字符串内有空格（显示的字符之间），这些空格将被保留。

该函数返回移除的字符数。

StringModify.mq5 文件演示了上述函数的运算。

```

void OnStart()
{
    string text = "  \tAbCdE F1 ";
        // ↑      ↑ ↑
        // |      |  L2 spaces
        // |      Lspace
        // L2 spaces and tab
    PRT(StringToLower(text)); // 'true'
    PRT(text);                // ' \tabcde f1 '
    PRT(StringToUpper(text)); // 'true'
    PRT(text);                // ' \tABCDE F1 '
    PRT(StringTrimLeft(text)); // '3'
    PRT(text);                // 'ABCDE F1 '
    PRT(StringTrimRight(text)); // '2'
    PRT(text);                // 'ABCDE F1'
    PRT(StringTrimRight(text)); // '0' (there is nothing else to delete)
    PRT(text);                // 'ABCDE F1'
                                //      ↑
                                //      Lthe space inside remains

    string russian = "Russian text";
    PRT(StringToUpper(russian)); // 'true'
    PRT(russian);                // 'RUSSIAN TEXT'
    string german = "straßenführung";
    PRT(StringToUpper(german));  // 'true'
    PRT(german);                // 'STRÄßENFÜHRUNG'
}

```

4.2.5 查找、替换和提取字符串片断

处理字符串时，或许最常见的操作就是查找、替换以及提取片断。在本节中，我们将学习帮助解决这些问题的 MQL5 API 函数。这些函数的使用示例在 *StringFindReplace.mq5* 文件中进行了汇总。

int StringFind(string value, string wanted, int start = 0)

该函数在 **value** 字符串中搜索 **wanted** 子串，从 **start** 位置开始搜索。如果找到该子串，则该函数将返回该子串的开始位置，并将该字符串中的字符从 0 开始编号。如果找不到，则该函数将返回 -1。两个参数均通过值传递，不仅允许处理变量，而且允许处理计算的中间结果（表达式、函数调用）。

搜索基于严格字符匹配执行，即区分大小写。如果要以区分大小写的方式搜索，必须首先使用 *StringToLower* 或 *StringToUpper* 将源字符串转换为单大写或单小写。

我们尝试使用 *StringFind* 来统计文本中找到的目标子串的数量。为此，我们编写 *CountSubstring* 辅助函数，其将循环调用 *StringFind* 逐渐偏移最后参数 **start** 中的搜索起始位置。只要找到新的子串，循环就继续。

```

int CountSubstring(const string value, const string wanted)
{
    // indent back because of the increment at the beginning of the loop
    int cursor = -1;
    int count = -1;
    do
    {
        ++count;
        ++cursor; // search continues from the next position
        // get the position of the next substring, or -1 if there are no matches
        cursor = StringFind(value, wanted, cursor);
    }
    while(cursor > -1);
    return count;
}

```

务必要注意，所提供的实现是查找能够重叠的子串。这是因为在开始查找下一个匹配结果之前会将当前位置加 1 (`++cursor`)。因此，当在字符串“AAAAA”中查找子串“AAA”时，将找到 3 个匹配结果。但实践中的搜索技术要求可能与此行为存在差异。特别是有一种实践，是在之前找到的片断结束的位置之后继续搜索。这种情况下就需要修改算法，使得游标以等于 `StringLen(wanted)` 的步长移动。

我们为 `OnStart` 函数中的不同自变量调用 `CountSubstring`。

```

void OnStart()
{
    string abracadabra = "ABRACADABRA";
    PRT(CountSubstring(abracadabra, "A"));    // 5
    PRT(CountSubstring(abracadabra, "D"));    // 1
    PRT(CountSubstring(abracadabra, "E"));    // 0
    PRT(CountSubstring(abracadabra, "ABRA")); // 2
    ...
}

```

`int StringReplace(string &variable, const string wanted, const string replacement)`

该函数在 `variable` 字符串中，用 `replacement` 子串替换找到的所有 `wanted` 子串。

该函数返回完成的替换数，如果出错，则返回 -1。错误代码可通过调用函数 `GetLastError` 来获得。这些错误尤其可能是内存不足错误，或者是使用了未初始化的字符串 (NULL) 作为自变量。`variables` 和 `wanted` 参数必须是非零长度的字符串。

若为 `replacement` 自变量提供了空字符串“”，所有找到的 `wanted` 结果直接从原始字符串剪切。

如果没有进行替换，则函数结果为 0。

我们以 `StringFindReplace.mq5` 为例来观察 `StringReplace` 的实际效果。

```

string abracadabra = "ABRACADABRA";
...
PRT(StringReplace(abracadabra, "ABRA", "-ABRA-")); // 2
PRT(StringReplace(abracadabra, "CAD", "-")); // 1
PRT(StringReplace(abracadabra, "", "XYZ")); // -1, error
PRT(GetLastError()); // 5040, ERR_WRONG_STRING_PARAMETER
PRT(abracadabra); // '-ABRA---ABRA-'
...

```

接下来, 使用 *StringReplace* 函数, 我们尝试执行在处理任意文本的过程中遇到的其中一项任务。我们将确保某些分隔符始终用作单字符, 即若干个连续分隔符须被替换为一个字符。通常这是指单词之间的空格, 但在技术数据中可能有其它分隔符。我们在程序中测试分隔符 '。’。

我们用单独 *NormalizeSeparatorsByReplace* 函数实现该算法:

```

int NormalizeSeparatorsByReplace(string &value, const ushort separator = ' ')
{
    const string single = ShortToString(separator);
    const string twin = single + single;
    int count = 0;
    int replaced = 0;
    do
    {
        replaced = StringReplace(value, twin, single);
        if(replaced > 0) count += replaced;
    }
    while(replaced > 0);
    return count;
}

```

程序尝试在一个 *do-while* 循环中用单分隔符序列替换双分隔符序列, 并且只要 *StringReplace* 函数返回大于 0 的值 (即仍然有要替换的内容), 则循环继续。函数返回进行替换的总次数。

在 *OnStart* 函数中, 我们从文本中清除多个 '。’ 字符。

```

...
string copy1 = "--" + abracadabra + "--";
string copy2 = copy1;
PRT(copy1); // '--ABRA---ABRA--'
PRT(NormalizeSeparatorsByReplace(copy1, '-')); // 4
PRT(copy1); // '-ABRA-ABRA-'
PRT(StringReplace(copy1, "--", "")); // 1
PRT(copy1); // 'ABRAABRA'
...

```

int StringSplit(const string value, const ushort separator, string &result[])

该函数基于给定的分隔符将传递的 *value* 字符串分离为子串, 并将它们放入 *result* 数组。该函数返回接收的子串数, 如果出错, 则返回 -1。

如果字符串中没有分隔符, 则数组将具有一个等于整个字符串的元素。

如果源字符串为空或 NULL, 则函数将返回 0。

为演示该函数的运算，我们使用 `StringSplit` 以新的方式解决前面的问题。为此，我们编写 `NormalizeSeparatorsBySplit` 函数。

```
int NormalizeSeparatorsBySplit(string &value, const ushort separator = ' ')
{
    const string single = ShortToString(separator);

    string elements[];
    const int n = StringSplit(value, separator, elements);
    ArrayPrint(elements); // debug

    StringFill(value, 0); // result will replace original string

    for(int i = 0; i < n; ++i)
    {
        // empty strings mean delimiters, and we only need to add them
        // if the previous line is not empty (i.e. not a separator either)
        if(elements[i] == "" && (i == 0 || elements[i - 1] != ""))
        {
            value += single;
        }
        else // all other lines are joined together "as is"
        {
            value += elements[i];
        }
    }

    return n;
}
```

当源文本中先后出现分隔符，输出数组 `StringSplit` 中的对应元素最终为空字符串 `""`。此外，如果文本以分隔符开始，则空字符串将位于数组的开头，而如果文本以分隔符结束，则空字符串位于数组末尾。

为获得“清除过的”文本，需要从数组添加所有非空字符串，以单分隔符“粘合”它们。此外，只有当数组中的空元素前一个元素也是非空时，才应将该空元素转换为分隔符

当然，这只是实现此功能的可能选项之一。我们在 `OnStart` 函数中检验。

```
...
string copy2 = "--" + abracadabra + "--"; // '--ABRA---ABRA--'
PRT(NormalizeSeparatorsBySplit(copy2, '-')); // 8
// debug output of split array (inside function):
// "" "" "ABRA" "" "" "ABRA" "" ""
PRT(copy2); // '-ABRA-ABRA-'
```

`string StringSubstr(string value, int start, int length = -1)`

该函数从传递的文本 `value` 中提取一个从指定位置 `start` 开始且长度为 `length` 的子串。开始位置可以从 0 到字符串长度减 1。如果 `length` 长度为 -1 或者大于从 `start` 到字符串结束的字符数，则字符串剩余部分将被完整提取。

该函数返回一个子串，如果参数不正确，则返回空字符串。

我们看看其工作方式。

```
PRT(StringSubstr("ABRACADABRA", 4, 3));           // 'CAD'
PRT(StringSubstr("ABRACADABRA", 4, 100));          // 'CADABRA'
PRT(StringSubstr("ABRACADABRA", 4));               // 'CADABRA'
PRT(StringSubstr("ABRACADABRA", 100));             // ''
```

4.2.6 处理符号和代码页

由于字符串是由字符组成的，有时候需要在整数代码层级操纵字符串中的单个字符或字符组，或者说这样更方便。例如，需要一次一个地读取或替换一个字符，或者将它们转换为整数代码数组，以便通过通信协议进行传输，或者转换为 [动态库](#) DLL 的第三方编程接口。在所有这些情况下，将字符串作为文本传递可能面临各种困难：

- 确保正确编码（有很多种编码，具体的选择取决于操作系统区域设置、程序设置、与之进行通信的服务器的配置等因素）
- 本地文本编码与 Unicode 的国家/地区语言字符之间的相互转换
- 以统一方式进行内存分配和解除分配

使用包含整数代码的数组（此类使用实际上生成了字符串的二进制表示而不是文本表示）会让这些问题变得简单。

SQL API 提供了对单个字符或字符组进行运算的一系列函数，这些函数已考虑到了编码特性。

SQL 中的字符串包含两字节 Unicode 编码的字符。以单个（但非常大）字符表为所有各个国家的字母表提供了通用支持。两个字节即允许 65535 个元素的编码。

默认字符类型为 *ushort*。但在必要时，可将字符串以特定语言编码转换为单字节 *uchar* 字符系列。这种转换可能伴随某些信息的丢失（尤其是在本地化字符表中的字母可能“丢失”变音符或甚至“转变”为某种替代字符：取决于上下文，其可能以不同方式显示，但通常显示为 '?' 或方块字符）。

为避免可能包含任意字符文本的问题，建议始终使用 Unicode。如果应与 SQL 程序集成的某些外部服务或程序不支持 Unicode，或者如果该文本从一开始就打算用于存储有限的字符集（例如，仅数字和拉丁字母），则可以例外。

与单字节字符相互转换时，SQL API 默认使用 ANSI 编码，具体取决于当前 Windows 设置。然而，开发人员可指定不同的码表（参见函数 *CharArrayToString*、*StringToCharArray*）。

这些函数的使用示例在 *StringSymbols.mq5* 文件中提供。

bool StringSetCharacter(string &variable, int position, ushort character)

该函数将传递的 *variable* 字符串中 *position* 处的字符更改为 *character* 值。数字必须在 0 到字符串长度 (*StringLen*) 减 1 之间。

如果要写入的字符为 0，则其指定一个换行结束（作为终止符零），即行长度变为等于 *position*。为该行分配的缓冲区大小不会改变。

如果 *position* 参数等于字符串长度，并且要写入的字符不等于 0，则该字符被添加到字符串，其长度增加 1。这等效于表达式：*variable += ShortToString(character)*。

如果成功完成，该函数返回 *true*，如果出错，则返回 *false*。

```

void OnStart()
{
    string numbers = "0123456789";
    PRT(numbers);
    PRT(StringSetCharacter(numbers, 7, 0)); // cut off at the 7th character
    PRT(numbers);                          // 0123456
    PRT(StringSetCharacter(numbers, StringLen(numbers), '*')); // add '*'
    PRT(numbers);                          // 0123456*
    ...
}

```

ushort StringGetCharacter(string value, int position)

该函数返回位于字符串中指定位置的字符的代码。位置编号必须在 0 到字符串长度 (*StringLen*) 减 1 之间。如果出错，该函数将返回 0。

该函数等效于使用运算符 '[]' 写入：*value[position]*。

```

string numbers = "0123456789";
PRT(StringGetCharacter(numbers, 5)); // 53 = code '5'
PRT(numbers[5]);                    // 53 - is the same

```

string CharToString(uchar code)

该函数将字符的 ANSI 码转换为单字符字符串。根据设置的 Windows 代码页，代码的上半部分 (大于 127) 可能生成不同的字母 (字符样式不同，而代码保持相同)。例如，代码为 0xB8 (十进制数 184) 的符号在西欧语言中表示变音符，而在俄语中，此位置是字母 'ё'。再举一个例子：

```

PRT(CharToString(0xA9)); // "©"
PRT(CharToString(0xE6)); // "æ", "ж", or another character
                        // depending on your Windows locale

```

string ShortToString(ushort code)

该函数将字符的 Unicode 码转换为单字符字符串。对于 *code* 参数，可以使用字面量或整数。例如，希腊大写字母 "sigma" (数学公式中的求和符号) 可指定为 0x3A3 或 'Σ'。

```

PRT(ShortToString(0x3A3)); // "Σ"
PRT(ShortToString('Σ'));  // "Σ"

```

int StringToShortArray(const string text, ushort &array[], int start = 0, int count = -1)

该函数将字符串转换为 *ushort* 字符码系列并拷贝到数组中的指定位置：从编号为 *start* (默认为 0，即数组开头) 的元素开始，并且数量为 *count*。

请注意，*start* 参数指数组中的位置，而不是字符串中的位置。如果想转换字符串的某一部分，必须首先使用 *StringSubstr* 函数来提取这一部分。

如果 *count* 参数等于 -1 (或者 *WHOLE_ARRAY*)，则将拷贝截至字符串末尾的所有字符 (包括终止符 NULL) 或者根据数组大小确定的字符数量 (如果为固定大小)。

对于动态数组，如果必要，其大小将自动增加。如果动态数组的大小大于字符串的长度，则该数组的大小不减少。

要拷贝没有终止 `null` 的字符，必须将 `StringLen` 作为 `count` 自变量进行显式调用。否则，数组的长度将比字符串的长度大 1 (而在最后一个元素为 0)。

该函数返回拷贝的字符数。

```
...
ushort array1[], array2[]; // dynamic arrays
ushort text[5];           // fixed size array
string alphabet = "ABCDEABГД";
// copy with the terminal '0'
PRT(StringToShortArray(alphabet, array1)); // 11
ArrayPrint(array1); // 65 66 67 68 69 1040 1041 1042 1043 1044 0
// copy without the terminal '0'
PRT(StringToShortArray(alphabet, array2, 0, StringLen(alphabet))); // 10
ArrayPrint(array2); // 65 66 67 68 69 1040 1041 1042 1043 1044
// copy to a fixed array
PRT(StringToShortArray(alphabet, text)); // 5
ArrayPrint(text); // 65 66 67 68 69
// copy beyond the previous limits of the array
// (elements [11-19] will be random)
PRT(StringToShortArray(alphabet, array2, 20)); // 11
ArrayPrint(array2);
/*
[ 0] 65 66 67 68 69 1040 1041 1042
    1043 1044 0 0 0 0 0 14245
[16] 15102 37754 48617 54228 65 66 67 68
    69 1040 1041 1042 1043 1044 0
*/
```

请注意，如果拷贝位置大于数组大小，则中间元素将被分配，但不会初始化。因此，它们可能包含随机数据 (上面以黄色高亮显示的部分)。

string ShortArrayToString(const ushort &array[], int start = 0, int count = -1)

该函数将数组带字符码的部分转换为字符串。数组元素的范围分别由 `start` (表示起始位置) 和 `count` (表示数量) 参数设置。`start` 参数值必须在 0 到数组元素数量减 1 之间。如果 `count` 等于 -1 (或 `WHOLE_ARRAY`)，则将拷贝截至数组末尾或截至第一个 `null` 的所有元素。

使用来自 `StringSymbols.mq5` 的相同示例，我们尝试将一个数组转换为大小为 30 的 `array2` 字符串。

```
...
string s = ShortArrayToString(array2, 0, 30);
PRT(s); // "ABCDEABГД", additional random characters may appear here
```

由于在 `array2` 数组中，字符串 “ABCDEABCD” 被拷贝两次，具体为，第一次拷贝到开头位置，第二次拷贝到偏移 20 的位置，因此中间字符将为随机，能够形成一个比之前更长的字符串。

`int StringToCharArray(const string text, uchar &array[], int start = 0, int count = -1, uint codepage = CP_ACP)`

该函数将 *text* 字符串转换为一个单字节字符系列并拷贝到数组中的指定位置：从编号为 *start*（默认为 0，即数组开头）的元素开始，并且数量为 *count*。拷贝过程将字符从 Unicode 转换为选定代码页 *codepage* -默认为 CP_ACP，即 Windows 操作系统语言（下文会详细介绍）。

如果 *count* 参数等于 -1（或 WHOLE_ARRAY），则将拷贝截至字符串末尾的所有字符（包括终止符 NULL）或者根据数组大小确定的字符数量（如果为固定大小）。

对于动态数组，如果必要，其大小将自动增加。如果动态数组的大小大于字符串的长度，则该数组的大小不减少。

要复制没有终止 null 的字符，必须将 *StringLen* 作为 *count* 自变量进行显式调用。

该函数返回拷贝的字符数。

参见文档中 *codepage* 参数的有效代码页列表。下面是一些广泛使用的 ANSI 代码页：

语 言	代 码
中欧拉丁语	1250
西里尔语	1251
西欧拉丁语	1252
希腊语	1253
土耳其语	1254
希伯来语	1255
阿拉伯语	1256
波罗的海语	1257

因此，在采用西欧语言的计算机上，CP_ACP 为 1252，而在采用俄语的计算机上，则为 1251。

在转换过程中，某些字符在转换后可能损失信息，因为 Unicode 表比 ANSI 表大得多（每个 ANSI 码表有 256 个字符）。

在这方面，CP_UTF8 在所有 CP_*** 常量中特别重要。它可以通过可变长度编码确保国家字符正确保留：生成的数组仍然存储字节，但每个国家字符可以跨多个字节，以特殊格式写入。因此，数组长度可能比字符串长度大得多。UTF-8 编码广泛用于互联网及各种软件。顺便说一下，UTF 代表 Unicode Transformation Form at（统一码转换格式），还有其它修改格式，尤其是 UTF-16 和 UTF-32。

在我们熟悉“逆”函数 *CharArrayToString* 之后，我们将探讨 *StringToCharArray* 的示例：它们的工作原理必须共同演示。

`string CharArrayToString(const uchar &array[], int start = 0, int count = -1, uint codepage = CP_ACP)`

该函数将字节数组或其一部分转换为字符串。数组必须包含特定编码的字符。数组元素的范围分别由 *start*（表示起始位置）和 *count*（表示数量）参数设置。*start* 参数值必须在 0 和数组元素数

量之间。当 `count` 等于 `-1` (或 `WHOLE_ARRAY`)，则截至数组末尾或截至第一个 `null` 的所有元素被复制。

我们来了解函数 `StringToCharArray` 和 `CharArrayToString` 如何处理具有不同代码页面设置的不同国家字符。为此准备了测试脚本 `StringCodepages.mq5`。

将使用两行作为测试对象，分别是俄语和德语：

```
void OnStart()
{
    Print("Locales");
    uchar bytes1[], bytes2[];

    string german = "straßenführung";
    string russian = "Russian text";
    ...
}
```

我们将其复制到数组 `bytes1` 和 `bytes2`，然后将其还原为字符串。

首先，我们使用欧洲代码页 1252 转换德语文本。

```
...
StringToCharArray(german, bytes1, 0, WHOLE_ARRAY, 1252);
ArrayPrint(bytes1);
// 115 116 114 97 223 101 110 102 252 104 114 117 110 103 0
```

在欧洲版本的 Windows 上，这等效于使用默认参数的较简单函数调用，因为其中 `CP_ACP = 1252`：

```
StringToCharArray(german, bytes1);
```

然后我们使用以下调用将文本从数组还原，确保一切与原始内容匹配：

```
...
PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, 1252));
// CharArrayToString(bytes1,0,WHOLE_ARRAY,1252)='straßenführung'
```

现在我们尝试以相同的欧洲编码转换俄语文本 (或者可以在 `CP_ACP` 设置为 1252 作为默认代码页的 Windows 环境中调用 `StringToCharArray(english, bytes2)`)：

```
...
StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, 1252);
ArrayPrint(bytes2);
// 63 63 63 63 63 63 63 32 63 63 63 63 63 0
```

你会看到转换期间出现问题，因为 1252 没有西里尔语。从数组还原字符串可清晰看到本质：

```
...
PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, 1252));
// CharArrayToString(bytes2,0,WHOLE_ARRAY,1252)='??????? ?????'
```

我们在条件性俄语环境中重复试验，即我们使用西里尔语代码页 1251 双向转换字符串。

```

...
StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, 1251);
// on Russian Windows, this call is equivalent to a simpler one
// StringToCharArray(russian, bytes2);
// because CP_ACP = 1251
ArrayPrint(bytes2); // this time the character codes are meaningful
// 208 243 241 241 234 232 233 32 210 229 234 241 242 0

// restore the string and make sure it matches the original
PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, 1251));
// CharArrayToString(bytes2,0,WHOLE_ARRAY,1251)='Русский Текст'

// and for the German text...
StringToCharArray(german, bytes1, 0, WHOLE_ARRAY, 1251);
ArrayPrint(bytes1);
// 115 116 114 97 63 101 110 102 117 104 114 117 110 103 0
// if we compare this content of bytes1 with the previous version,
// it's easy to see that a couple of characters are affected; here's what happened
// 115 116 114 97 223 101 110 102 252 104 114 117 110 103 0

// restore the string to see the differences visually:
PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, 1251));
// CharArrayToString(bytes1,0,WHOLE_ARRAY,1251)='stra?enfuhrung'
// specific German characters were corrupted

```

这样，单字节编码的脆弱性就显而易见了。

最后，我们为两个测试字符串均启用 CP_UTF8 编码。不论 Windows 设置如何，示例的这一部分将能够稳定工作。

```

...
StringToCharArray(german, bytes1, 0, WHOLE_ARRAY, CP_UTF8);
ArrayPrint(bytes1);
// 115 116 114 97 195 159 101 110 102 195 188 104 114 117 110 103 0
PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, CP_UTF8));
// CharArrayToString(bytes1,0,WHOLE_ARRAY,CP_UTF8)='straßenführung'

StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, CP_UTF8);
ArrayPrint(bytes2);
// 208 160 209 131 209 129 209 129 208 186 208 184 208 185
// 32 208 162 208 181 208 186 209 129 209 130 0
PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, CP_UTF8));
// CharArrayToString(bytes2,0,WHOLE_ARRAY,CP_UTF8)='Русский Текст'

```

请注意，两个 UTF-8 编码的字符串需要的数组大小均超过 ANSI 编码字符串需要的数组。此外，俄语文本的数组实际上长度已变为 2 倍，因为所有字母现在占用 2 个字节。有兴趣者可在开源代码中找到有关 UTF-8 编码具体如何工作的详细信息。在本书中，重要的是 MQL5 API 提供了现成可用的函数。

4.2.7 数据到字符串的通用格式化输出

如果生成字符串以向用户显示，将字符串保存到文件或是通过互联网发送，可能需要将不同类型的若干个变量的值包括在其中。要解决这个问题，可以将所有变量显式强制转换为 (*string*) 类型并添加生成的字符串，但在这种情况下，MQL 代码指令将会很长而难以理解。如果使用 *StringConcatenate* 函数可能会更方便，但该方法不能完全解决该问题。

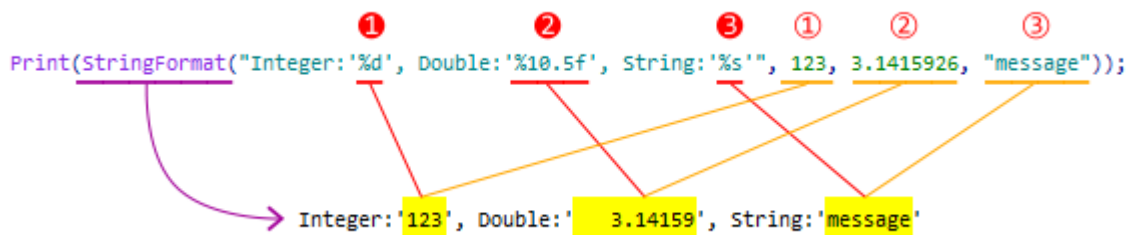
事实是，字符串通常不仅包含变量，而且还包含某些文本插入内容，这些文本内容充当链接纽带并提供整体消息的正确结构体。结果就造成一些设置格式的文本与变量混合在一起。这种代码难以维护，违背众所周知的编程原则：内容与呈现方式分离。

这个问题有一个特殊的解决方案：*StringFormat* 函数。

同样方案适用于另一个 MQL5 API 函数：*PrintFormat* 中深入了解有关格式化输出到字符串的内容。

string StringFormat(const string format, ...)

该函数根据指定格式将任意内置类型自变量转换为字符串。第一个参数是要准备的字符串模板，其中以特殊方式指示了要插入变量的位置，并且确定了其输出格式。这些控制命令可能与纯文本交错出现，后者被原样拷贝到输出字符串。由逗号分隔的以下函数参数以模板中为其保留的顺序和类型列出所有变量。



格式字符串和 StringFormat 自变量的交互

字符串中的每个变量插入点以格式说明符 (字符 `%`) 标记，在该说明符之后可指定若干设置。

格式字符串从左到右解析。碰到第一个说明符 (如有) 时，在格式字符串之后的第一个参数的值根据指定设置被转换并添加到生成的字符串。第二个说明符促成第二个参数被转换和打印，以此类推，直至格式字符串结束。模式中处于说明符之间的所有其它字符均按原样拷贝到生成的字符串。

模板可能不含任何说明符，即可能是简单字符串。在这种情况下，除了自变量外，还需要向函数传递一个虚拟自变量 (该自变量将不放置在字符串中)。

如果想在模板中显示百分号，则应在一行中写两次：`%%`。如果 `%` 号不写两次，则 `%` 之后的几个字符将始终解析为一个说明符。

说明符的其中一个强制属性是一个符号，指示下一个函数自变量的预期类型和解释方式。我们条件性地调用符号 `T`。然后在最简单的情况下，调用一个如 `%T` 的格式说明符。

该说明符为广义形态，可由若干字段组成 (可选字段以方括号指示)：

`%[Z][W][.P][M]T`

每个字段执行其函数并取允许的值之一。接下来，我们逐步了解所有字段。

类型 T:

对于整数，以下字符可用作 T，并解释了对应数字在字符串中显示方式。

- c -Unicode 字符
- C -ANSI 字符
- d, i -有符号十进制数
- o -无符号八进制数
- u -无符号十进制数
- x -无符号十六进制数 (小写)
- X -无符号十六进制数 (大写字母)

别忘了，根据内部数据存储方法，整数类型还包括内置 MySQL 类型 *datetime*、*color*、*bool* 以及枚举。

对于实数，以下符号适用于作为 T:

- e -带指数的科学格式 (小写 'e')
- E -带指数的科学格式 (大写 'E')
- f -规范格式
- g -类似于 f 或 e (选择最紧凑的形式)
- G -类似于 f 或 E (选择最紧凑的形式)
- a -带指数的科学格式，十六进制 (小写)
- A -带指数的科学格式，十六进制 (大写)

最后，只有一个版本的 T 字符可用于字符串，即 s。

整数大小 M

对于整数类型，还可显式指定变量的字节大小，方法是在 T 前添加以下字符 (或其组合，此处统一用字母 M 表示):

- h -2 字节 (short, ushort)
- l (小写 L) -4 字节 (int, uint)
- l32 (大写 i) -4 字节 (int, uint)
- ll (两个小写 L) -8 字节 (long)
- l64 (大写 i) -8 字节 (long, ulong)

宽度 W

W 字段是一个非负十进制数字，指定格式化值占用的最小字符空间数。如果变量值占用的字符数少于以上值，则在左侧或右侧添加对应数量的空格。选择在左侧还是右侧添加取决于对齐方式 (详见 Z 字段中的标志 '⁠)。如果有 '0' 标记，则对应的零数量被添加到输出值的前面。如果要输出的字符数大于指定宽度，则忽略宽度设置，并且不会截断输出值。

如果将星号 * 指定为宽度，则输出值的宽度应在传递参数列表中指定。其应为一个 *int* 类型的值，位于被格式化的变量前面。

精度 P

P 字段也包含一个非负十进制数，其前面始终有一个点 ‘.’’。对于整数 T，该字段指定最小有效位数。如果该值占用的数位较少，则在前面补零。

对于实数，P 指定小数位数（默认 6），但 g 和 G 说明符除外，对于这二者，P 表示总有效位数（尾数和小数）。

对于字符串，P 指定要显示的字符数。如果字符串长度超过精度值，则该字符串在显示时将被截断。

如果将星号 ‘*’ 指定为精度，则其处理方式与宽度相同，但控制精度。

固定宽度和/或精度结合右对齐便能以整齐列显示值。

标志 Z

最后，Z 字段描述标志：

- - (减号) -在指定宽度内左对齐（如果没有该标志，则完成右对齐）；
- + (加号) -在值前面无条件显示 ‘+’ 号或 ‘-’ 号（如果没有该标记，则仅对负值显示 ‘-’）；
- 0 -如果小于指定宽度，则在输出值前面添加零；
- (空格) -如果值有符号并为正值，则在显示值前面添加空格；
- # -控制 o、x 或 X 格式的八进制和十六进制数字前缀的显示（例如，对于 x 格式，会在显示的数字前面添加前缀 “0x”，对于 X 格式则添加前缀 “0X”）、带零小数部分的实数中的小数点（e、E、a 或 A 格式）以及一些其它细微差别。

可以在 [文档](#) 中深入了解有关格式化输出到字符串的内容。

函数参数总数不能超过 64。

如果传递到函数的自变量数量大于说明符数量，则多余自变量被省略。

如果格式字符串中的说明符数量大于自变量数，则系统将尝试显示零以代替缺失数据，但将为字符串说明符嵌入文本警告（“缺少字符串参数”）。

如果值的类型与对应说明符的类型不匹配，则系统将尝试根据格式从变量读取数据并显示结果值（由于错误解析了真实数据的内部表示方式，显示的值可能看起来奇怪）。对于字符串，结果中可能嵌入警告（“传递了非字符串”）。

我们使用 *StringFormat.mq5* 脚本测试函数。

首先，我们尝试 T 和数据类型说明符的不同选项。

```

PRT(StringFormat("[Infinity Sign] Unicode (ok): %c; ANSI (overflow): %C",
    '∞', '∞'));
PRT(StringFormat("short (ok): %hi, short (overflow): %hi",
    SHORT_MAX, INT_MAX));
PRT(StringFormat("int (ok): %i, int (overflow): %i",
    INT_MAX, LONG_MAX));
PRT(StringFormat("long (ok): %lli, long (overflow): %i",
    LONG_MAX, LONG_MAX));
PRT(StringFormat("ulong (ok): %llu, long signed (overflow): %lli",
    ULONG_MAX, ULONG_MAX));

```

此处同时展示了正确和不正确的说明符（不正确的说明符在每个指令中第二出现，并标记有“overflow”（溢出）字样，因为传递的值与格式类型不匹配）。

日志中显示以下内容（此处及下文的长行换行是为便于排版）：

```

StringFormat(Plain string,0)='Plain string'
StringFormat([Infinity Sign] Unicode: %c; ANSI: %C,'∞','∞')=
    '[Infinity Sign] Unicode (ok): ∞; ANSI (overflow): '
StringFormat(short (ok): %hi, short (overflow): %hi,SHORT_MAX,INT_MAX)=
    'short (ok): 32767, short (overflow): -1'
StringFormat(int (ok): %i, int (overflow): %i,INT_MAX,LONG_MAX)=
    'int (ok): 2147483647, int (overflow): -1'
StringFormat(long (ok): %lli, long (overflow): %i,LONG_MAX,LONG_MAX)=
    'long (ok): 9223372036854775807, long (overflow): -1'
StringFormat(ulong (ok): %llu, long signed (overflow): %lli,ULONG_MAX,ULONG_MAX)=
    'ulong (ok): 18446744073709551615, long signed (overflow): -1'

```

所有下列指令均正确：

```

PRT(StringFormat("ulong (ok): %I64u", ULONG_MAX));
PRT(StringFormat("ulong (HEX): %I64X, ulong (hex): %I64x",
    1234567890123456, 1234567890123456));
PRT(StringFormat("double PI: %f", M_PI));
PRT(StringFormat("double PI: %e", M_PI));
PRT(StringFormat("double PI: %g", M_PI));
PRT(StringFormat("double PI: %a", M_PI));
PRT(StringFormat("string: %s", "ABCDEFGHJIJ"));

```

其执行结果如下所示：

```

StringFormat(ulong (ok): %I64u,ULONG_MAX)=
    'ulong (ok): 18446744073709551615'
StringFormat(ulong (HEX): %I64X, ulong (hex): %I64x,1234567890123456,1234567890123456
    'ulong (HEX): 462D53C8ABAC0, ulong (hex): 462d53c8abac0'
StringFormat(double PI: %f,M_PI)='double PI: 3.141593'
StringFormat(double PI: %e,M_PI)='double PI: 3.141593e+00'
StringFormat(double PI: %g,M_PI)='double PI: 3.14159'
StringFormat(double PI: %a,M_PI)='double PI: 0x1.921fb54442d18p+1'
StringFormat(string: %s,ABCDEFGHJIJ)='string: ABCDEFGHJIJ'

```

现在我们来了解各种修饰符。

使用右对齐 (默认)和固定字段宽度 (字符数),我们可以使用不同选项在生成字符串的左侧进行填充:使用空格或零。此外,对于任何对齐方式,可以启用或禁用值符号的显式指示 (不仅能为负值显示负号,还可以为正值显示正号)。

```
PRT(StringFormat("space padding: %10i", SHORT_MAX));
PRT(StringFormat("0-padding: %010i", SHORT_MAX));
PRT(StringFormat("with sign: %+10i", SHORT_MAX));
PRT(StringFormat("precision: %.10i", SHORT_MAX));
```

我们在日志中获得以下内容:

```
StringFormat(space padding: %10i,SHORT_MAX)='space padding:      32767'
StringFormat(0-padding: %010i,SHORT_MAX)='0-padding: 0000032767'
StringFormat(with sign: %+10i,SHORT_MAX)='with sign:      +32767'
StringFormat(precision: %.10i,SHORT_MAX)='precision: 0000032767'
```

要左对齐,必须使用 ‘-’ (符号)标志,将在右侧添加指定宽度的字符串:

```
PRT(StringFormat("no sign (default): %-10i", SHORT_MAX));
PRT(StringFormat("with sign: %+ -10i", SHORT_MAX));
```

结果:

```
StringFormat(no sign (default): %-10i,SHORT_MAX)='no sign (default): 32767      '
StringFormat(with sign: %+ -10i,SHORT_MAX)='with sign: +32767      '
```

如果必要,我们可以显示或隐藏值的符号 (默认情况下,仅为负值显示负号),为正值添加空格,可在需要按列显示变量时确保格式设置相同:

```
PRT(StringFormat("default: %i", SHORT_MAX)); // standard
PRT(StringFormat("default: %i", SHORT_MIN));
PRT(StringFormat("space  : % i", SHORT_MAX)); // extra space for positive
PRT(StringFormat("space  : % i", SHORT_MIN));
PRT(StringFormat("sign   : %+i", SHORT_MAX)); // force sign output
PRT(StringFormat("sign   : %+i", SHORT_MIN));
```

日志中如下所示:

```
StringFormat(default: %i,SHORT_MAX)='default: 32767'
StringFormat(default: %i,SHORT_MIN)='default: -32768'
StringFormat(space  : % i,SHORT_MAX)='space   :  32767'
StringFormat(space  : % i,SHORT_MIN)='space   : -32768'
StringFormat(sign   : %+i,SHORT_MAX)='sign    : +32767'
StringFormat(sign   : %+i,SHORT_MIN)='sign    : -32768'
```

现在我们比较宽度和精度对实数的影响。


```

PRT(StringFormat("double PI: %15.10f", M_PI));
PRT(StringFormat("double PI: %15.10e", M_PI));
PRT(StringFormat("double PI: %15.10g", M_PI));
PRT(StringFormat("double PI: %15.10a", M_PI));

// default precision = 6
PRT(StringFormat("double PI: %15f", M_PI));
PRT(StringFormat("double PI: %15e", M_PI));
PRT(StringFormat("double PI: %15g", M_PI));
PRT(StringFormat("double PI: %15a", M_PI));

```

结果:

```

StringFormat(double PI: %15.10f,M_PI)='double PI:      3.1415926536'
StringFormat(double PI: %15.10e,M_PI)='double PI: 3.1415926536e+00'
StringFormat(double PI: %15.10g,M_PI)='double PI:      3.141592654'
StringFormat(double PI: %15.10a,M_PI)='double PI: 0x1.921fb54443p+1'
StringFormat(double PI: %15f,M_PI)='double PI:      3.141593'
StringFormat(double PI: %15e,M_PI)='double PI:      3.141593e+00'
StringFormat(double PI: %15g,M_PI)='double PI:      3.14159'
StringFormat(double PI: %15a,M_PI)='double PI: 0x1.921fb54442d18p+1'

```

如果未指定显式宽度, 则直接输出值, 不填充空格。

```

PRT(StringFormat("double PI: %.10f", M_PI));
PRT(StringFormat("double PI: %.10e", M_PI));
PRT(StringFormat("double PI: %.10g", M_PI));
PRT(StringFormat("double PI: %.10a", M_PI));

```

结果:

```

StringFormat(double PI: %.10f,M_PI)='double PI: 3.1415926536'
StringFormat(double PI: %.10e,M_PI)='double PI: 3.1415926536e+00'
StringFormat(double PI: %.10g,M_PI)='double PI: 3.141592654'
StringFormat(double PI: %.10a,M_PI)='double PI: 0x1.921fb54443p+1'

```

使用符号 `*` 并基于附加函数自变量来设置值的宽度和精度按如下执行:

```

PRT(StringFormat("double PI: %*.f", 12, 5, M_PI));
PRT(StringFormat("string: %*s", 15, "ABCDEFGHJIJ"));
PRT(StringFormat("string: %-*s", 15, "ABCDEFGHJIJ"));

```

请注意, 根据说明符中的星号 `*` 的数量, 在输出值之前传递 1 个或 2 个整数类型的值: 可以单独或同时控制精度和宽度。

```

StringFormat(double PI: %*.f,12,5,M_PI)='double PI:      3.14159'
StringFormat(string: %*s,15,ABCDEFGHJIJ)='string:      ABCDEFGHJIJ'
StringFormat(string: %-*s,15,ABCDEFGHJIJ)='string: ABCDEFGHJIJ      '

```

最后, 我们来了解几个常见格式设置错误。

```
PRT(StringFormat("string: %s %d %f %s", "ABCDEFGHJIJ"));
PRT(StringFormat("string vs int: %d", "ABCDEFGHJIJ"));
PRT(StringFormat("double vs int: %d", M_PI));
PRT(StringFormat("string vs double: %s", M_PI));
```

第一个指令的说明符数量超过自变量数量。在其它情况下,说明符的类型和传递值不匹配。结果,我们得到以下输出:

```
StringFormat(string: %s %d %f %s, ABCDEFGHJIJ)=
  'string: ABCDEFGHJIJ 0 0.000000 (missed string parameter)'
StringFormat(string vs int: %d, ABCDEFGHJIJ)= 'string vs int: 0'
StringFormat(double vs int: %d, M_PI)= 'double vs int: 1413754136'
StringFormat(string vs double: %s, M_PI)=
  'string vs double: (non-string passed)'
```

在每次 *StringFormat* 函数调用中使用单一格式字符串,尤其使其用于将程序的外部接口和消息转换为不同语言:只需下载并根据用户首选项或终端设置代入 *StringFormat* 各种格式字符串(事先准备)。

4.3 处理数组

任何没有数组的程序(尤其是与交易相关的程序)是不可想象的。在“数组”章节中,我们已经学习了描述和使用数组的一般原则。数组和处理数组的一系列内置函数有机地相辅相成。

一些函数提供了现成的最常用数组运算的实现,诸如查找最大值和最小值、排序、插入以及删除元素等。

然而,有很多函数是使用特定类型的数组时必不可少的。尤其是动态数组,必须先为其分配内存才能使用,若数组带有指标缓冲区数据(我们将在本书第五章学习这一 MQL 程序类型),则使用由特殊函数设置的特殊元素索引顺序。

我们将开始学习用于处理数组的函数,首先了解输出到日志的操作。我们在本书前面的章节已对其有所了解,这些函数在很多后续章节中也将很有用。

由于 MQL5 数组可能是多维的(1 到 4 维),我们在后续在文本中还会提及维度编号。我们将从第一维开始对其进行编号,这在几何上更易理解,也强调了数组必须至少有一个维度(即使为空)这一事实然而,每个维度的数组元素是有编号的,这是 MQL5 以及很多其它编程语言)的惯例,从零开始编号。因此,对于一个描述为 *array[5][10]* 的数组,第一维度是 5,第二维度是 10。

4.3.1 数组的日志记录

若要通知用户、进行调试以及诊断问题,最简单的方式就是将变量、数组和有关 MQL 程序状态的消息打印到日志。至于数组,我们可以使用 *Print* 函数实现逐元素打印,我们已从演示脚本了解了该函数。稍后我们将在关于 *与用户交互* 的章节正式描述该函数。

然而,将与元素迭代及其准确格式设置相关的整个例行程序委托给 MQL5 环境会更加方便。为此,API 提供了特殊的 *ArrayPrint* 函数。

我们已在 *使用数组* 章节了解了该函数的使用示例。现在我们深入探讨其功能。

```
void ArrayPrint(const void &array[], uint digits = _Digits, const string separator = NULL,
               ulong start = 0, ulong count = WHOLE_ARRAY,
               ulong flags = ARRAYPRINT_HEADER | ARRAYPRINT_INDEX | ARRAYPRINT_LIMIT | ARRAYPRINT_DATE |
               ARRAYPRINT_SECONDS)
```

该函数使用指定设置将一个数组记录到日志。该数组必须是内置类型之一或者是简单结构体类型。简单结构体是具有内置类型字段的结构体，字符串和动态数组除外。如果结构体组成中存在类对象和指针，则不属于简单类别。

数组必须具有维度 1 或 2。格式设置自动调整为数组配置，如果可能，以可视化形式显示（见下文）。尽管 MQL5 支持具有最多 4 个维度的数组，但该函数不显示具有 3 个或以上维度的数组，因为很难以平面形态表示这种数组。这种行为在程序编译或执行阶段不会引发错误。

除了第一个参数外，其他参数都可以省略，并为它们定义了默认值。

digits 参数用于实数数组以及结构体的数字字段。该参数设置数字小数部分显示的字符数。默认值是 [预定义图表变量](#) 之一，即 `_Digits`，这是当前图表的交易品种价格中的小数位数。

分隔字符 **separator** 用于指定在结构体数组中显示字段时的列。在默认值（NULL）情况下，该函数使用空格作为分隔符。

start 和 **count** 参数分别设置起始元素编号以及要打印的元素数量。默认情况下，该函数打印整个数组，但结果可能还会受到 `ARRAYPRINT_LIMIT` 标志的影响（见下文）。

flags 参数接受组合使用影响各种显示特性的标志。其中一些标志如下：

- `ARRAYPRINT_HEADER` 在结构体数组之前输出头部（其中包含结构体字段名称）；该标志不影响非结构体数组。
- `ARRAYPRINT_INDEX` 按维度输出元素索引（对于一维数组，索引显示在左边，对于二维数组，索引显示在左边和上方）。
- `ARRAYPRINT_LIMIT` 用于大数组，输出限于前一百条和后一百条记录（默认启用此限制）。
- `ARRAYPRINT_DATE` 用于 *datetime* 类型的值以显示日期。
- `ARRAYPRINT_MINUTES` 用于 *datetime* 类型的值以显示时间到最近分钟数。
- `ARRAYPRINT_SECONDS` 用于 *datetime* 类型的值以显示时间到最近秒数。

默认情况下，*datetime* 类型的值以 `ARRAYPRINT_DATE | ARRAYPRINT_SECONDS` 格式输出。

color 类型的值以十六进制格式输出。

枚举值显示为整数。

该函数不输出嵌套数组、结构体和对象指针。而是显示三个点代替。

ArrayPrint.mq5 脚本演示了该函数的工作方式。

OnStart 函数提供了若干数组（一维、二维、三维）的定义，这些数组使用 *ArrayPrint*（采用默认设置）输出。

```

void OnStart()
{
    int array1D[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    double array2D[][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}};
    double array3D[][3][5] =
    {
        {{ 1, 2, 3, 4, 5}, { 6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}},
        {{16, 17, 18, 19, 20}, {21, 22, 23, 24, 25}, {26, 27, 28, 29, 30}},
    };

    Print("array1D");
    ArrayPrint(array1D);
    Print("array2D");
    ArrayPrint(array2D);
    Print("array3D");
    ArrayPrint(array3D);
    ...
}

```

我们将在日志中获得以下行：

```

array1D
 1  2  3  4  5  6  7  8  9 10
array2D
      [,0]      [,1]      [,2]      [,3]      [,4]
[0,]  1.00000  2.00000  3.00000  4.00000  5.00000
[1,]  6.00000  7.00000  8.00000  9.00000 10.00000
array3D

```

array1D 数组不够大 (可以放入一行中)，因此未显示它的索引。

array2D 数组有多行 (索引)，因此显示它们的索引 (默认启用 `ARRAYPRINT_INDEX`)。

请注意，由于是对包含五位数价格 `_Digits=5` 的 EURUSD 图表运行脚本，其影响 *double* 类型值的格式设置。

array3D 数组被忽略：不输出它的行。

此外，*Pair* 和 *SimpleStruct* 结构体在脚本中定义：

```

struct Pair
{
    int x, y;
};

struct SimpleStruct
{
    double value;
    datetime time;
    int count;
    ENUM_APPLIED_PRICE price;
    color clr;
    string details;
    void *ptr;
    Pair pair;
};

```

SimpleStruct 包含内置类型字段、一个 *void* 指针，以及一个 *Pair* 类型的字段。

在 *OnStart* 函数中，使用 *ArrayPrint* 以两种模式创建并输出一个 *SimpleStruct* 类型的数组：以默认设置和自定义设置两种模式（“逗号”之后的数位数是 3，分隔符是 “;”，*datetime* 的格式是 “仅日期”）。

```

void OnStart()
{
    ...
    SimpleStruct simple[] =
    {
        { 12.57839, D'2021.07.23 11:15', 22345, PRICE_MEDIAN, clrBlue, "text message"},
        {135.82949, D'2021.06.20 23:45', 8569, PRICE_TYPICAL, clrAzure},
        { 1087.576, D'2021.05.15 10:01:30', -3298, PRICE_WEIGHTED, clrYellow, "note"},
    };
    Print("SimpleStruct (default)");
    ArrayPrint(simple);

    Print("SimpleStruct (custom)");
    ArrayPrint(simple, 3, ";", 0, WHOLE_ARRAY, ARRAYPRINT_DATE);
}

```

这将生成以下结果：

```

SimpleStruct (default)
      [value]           [time] [count] [type]   [clr]       [details] [ptr] [pair]
[0]   12.57839 2021.07.23 11:15:00   22345       5 00FF0000 "text message" ... ..
[1]   135.82949 2021.06.20 23:45:00    8569       6 00FFFFFF0 null          ... ..
[2] 1087.57600 2021.05.15 10:01:30   -3298       7 0000FFFF "note"          ... ..
SimpleStruct (custom)
    12.578;2021.07.23; 22345;      5;00FF0000;"text message"; ...; ...
    135.829;2021.06.20; 8569;      6;00FFFFFF0>null          ; ...; ...
1087.576;2021.05.15; -3298;      7;0000FFFF;"note"          ; ...; ...

```

请注意，我们在本例中以及在前面章节中使用的日志是在终端中生成，用户可在 *Toolbox* 窗口的 *Experts* 选项卡中查看。但是稍后我们将熟悉测试程序，其象终端本身一样为某些类型的 MQL 程序（指标和 EA 交易）提供相同的执行环境。如果它们在测试程序中启动，则

ArrayPrint 函数及其它相关函数 (在 [用户交互](#) 章节中介绍) 会将消息输出到 [测试代理](#) 的日志中。

但截至目前, 我们一直并且在一定时间内将继续仅使用脚本, 并且这些脚本只能在终端内执行。

4.3.2 动态数组

动态数组可在程序执行期间根据编程人员的要求更改其大小。别忘了, 要描述动态数组, 应将位于数组标识符后面的第一对括号留空。MQL5 要求所有后续维度 (如果多于一个维度) 必须具有以常量指定的固定大小。

无法为“旧于”第一个维度的任何维度动态增加元素数量。此外, 由于大小描述的严格性, 数组具有“正方形”形状, 例如, 无法构造列或行长度不同的二维数组。如果任何这些限制对于算法实现至为关键, 你不应使用标准 MQL5 数组, 而应使用你自己的以 MQL5 编写的结构体或类。

请注意, 如果一个数组在第一维度中没有大小, 但有一个可让你确定该大小的初始化列表, 则该数组是一个固定大小数组, 而不是动态数组。

例如, 在前一章节中, 我们使用了 **array1D** 数组:

```
int array1D[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

由于有初始化列表, 其大小对于编译器已知, 因此数组大小固定。

与这个简单示例不同的是, 判定真实程序中的一个特定数组是否是动态数组并不总是那么容易。尤其是, 数组可作为参数传递到函数。然而, 知道一个数组是否是动态数组可能很重要, 因为仅可为此类数组通过调用 **ArrayResize** 来手动分配内存。

在此等情况下, **ArrayIsDynamic** 函数可让你确定数组类型。

我们来了解一些处理动态数组的函数的一些技术内容, 然后使用 **ArrayDynamic.mq5** 脚本测试动态数组。

bool ArrayIsDynamic(const void &array[])

该函数检查传递的函数是否为动态。一个数组可以是任何允许的 1 维到 4 维的数组。数组元素可以是任何类型。

对于动态数组, 该函数返回 **true**, 对于其它数组, 返回 **false** (fixed 数组或具有 [时间序列](#) 的数组由终端本身或由指标控制的数组)。

int ArrayResize(void &array[], int size, int reserve = 0)

该函数设置动态 **array** 的第一维度中的新 **size**。一个数组可以是任何允许的 1 维到 4 维的数组。数组元素可以是任何类型。

如果 **reserve** 参数大于零, 则为数组分配的内存会预留出指定数量元素的存储空间。这可以提高具有多个连续函数调用的程序的运行速度。在数组的新请求大小超过当前大小 (含保留空间) 之前, 将不会进行物理内存重新分配, 新元素的内存将从预留部分中获取。

如果修改成功, 则函数返回数组的新大小, 如果出错, 则返回 -1。

如果函数被运用于 **fixed** 数组或时间序列, 其大小不会改变。在这些情况下, 如果请求的大小小于或等于数组的当前大小, 则函数将返回 **size** 参数的值, 如果小于, 则返回 -1。

增加现有数组的大小时，其元素的所有数据将保留。添加的元素不会以任何值初始化，并且可能包含任意不正确数据（“垃圾”）。

将数组大小设置为 0 (***ArrayResize(array, 0)***,) 并不会释放实际分配给该数组的内存，包括可能的预留空间。这种调用只会重置数组的元数据。这样做是为了优化数组的将来操作。要强制释放内存，请使用 ***ArrayFree*** (见下文)。

请务必理解，并非每次调用该函数时都会使用 ***reserve*** 参数，仅在实际执行了内存重新分配时才会使用，即当请求的大小超过数组的当前容量 (包括预留空间) 时才会使用。为直观说明这一工作原理，我们创建内部数组对象的不完整副本，并为其实现孪生函数 ***ArrayResize*** 以及类似函数 ***ArrayFree*** 和 ***ArraySize***，以获得完整工具包。

```

template<typename T>
struct DynArray
{
    int size;
    int capacity;
    T memory[];
};

template<typename T>
int DynArraySize(DynArray<T> &array)
{
    return array.size;
}

template<typename T>
void DynArrayFree(DynArray<T> &array)
{
    ArrayFree(array.memory);
    ZeroMemory(array);
}

template<typename T>
int DynArrayResize(DynArray<T> &array, int size, int reserve = 0)
{
    if(size > array.capacity)
    {
        static int temp;
        temp = array.capacity;
        long ul = (long)GetMicrosecondCount();
        array.capacity = ArrayResize(array.memory, size + reserve);
        array.size = MathMin(size, array.capacity);
        ul -= (long)GetMicrosecondCount();
        PrintFormat("Reallocation: [%d] -> [%d], done in %d μs",
            temp, array.capacity, -ul);
    }
    else
    {
        array.size = size;
    }
    return array.size;
}

```

DynArrayResize 函数相比内置 **ArrayResize** 函数的一个优势在于，我们针对数组的内部容量被重新分配的情况插入调试打印。

现在我们来看看来自 MQL5 文档的 **ArrayResize** 函数的标准示例，并以带有 “Dyn” 前缀的自制类似函数替换内置函数调用。修改后的结果在 **ArrayCapacity.mq5** 脚本中提供。


```

void OnStart()
{
    ulong start = GetTickCount();
    ulong now;
    int count = 0;

    DynArray<double> a;

    // fast option with memory reservation
    Print("--- Test Fast: ArrayResize(arr,100000,100000)");

    DynArrayResize(a, 100000, 100000);

    for(int i = 1; i <= 300000 && !IsStopped(); i++)
    {
        // set the new size and reserve to 100000 elements
        DynArrayResize(a, i, 100000);
        // on "round" iterations, show the size of the array and the elapsed time
        if(DynArraySize(a) % 100000 == 0)
        {
            now = GetTickCount();
            count++;
            PrintFormat("%d. ArraySize(arr)=%d Time=%d ms",
                count, DynArraySize(a), (now - start));
            start = now;
        }
    }
    DynArrayFree(a);

    // now this is a slow option without redundancy (with less redundancy)
    count = 0;
    start = GetTickCount();
    Print("---- Test Slow: ArrayResize(slow,100000)");

    DynArrayResize(a, 100000, 100000);

    for(int i = 1; i <= 300000 && !IsStopped(); i++)
    {
        // set new size but with 100 times smaller margin: 1000
        DynArrayResize(a, i, 1000);
        // on "round" iterations, show the size of the array and the elapsed time
        if(DynArraySize(a) % 100000 == 0)
        {
            now = GetTickCount();
            count++;
            PrintFormat("%d. ArraySize(arr)=%d Time=%d ms",
                count, DynArraySize(a), (now - start));
            start = now;
        }
    }
}

```

唯一主要差别是：在慢速版本中，`ArrayResize(a, i)` 调用替换为更适度的 `DynArrayResize(a, i, 1000)` 调用，即，并非每次迭代均请求重新分配，而是每 1000 次迭代请求一次（否则日志将发生消息溢出）。

运行该脚本后，我们将在日志中看到以下计时（绝对时间间隔取决于你的计算机，但我们感兴趣的是有预留和无预留情况下的执行差异）：

```
--- Test Fast: ArrayResize(arr,100000,100000)
Reallocation: [0] -> [200000], done in 17 µs
1. ArraySize(arr)=100000 Time=0 ms
2. ArraySize(arr)=200000 Time=0 ms
Reallocation: [200000] -> [300001], done in 2296 µs
3. ArraySize(arr)=300000 Time=0 ms
---- Test Slow: ArrayResize(slow,100000)
Reallocation: [0] -> [200000], done in 21 µs
1. ArraySize(arr)=100000 Time=0 ms
2. ArraySize(arr)=200000 Time=0 ms
Reallocation: [200000] -> [201001], done in 1838 µs
Reallocation: [201001] -> [202002], done in 1994 µs
Reallocation: [202002] -> [203003], done in 1677 µs
Reallocation: [203003] -> [204004], done in 1983 µs
Reallocation: [204004] -> [205005], done in 1637 µs
...
Reallocation: [295095] -> [296096], done in 2921 µs
Reallocation: [296096] -> [297097], done in 2189 µs
Reallocation: [297097] -> [298098], done in 2152 µs
Reallocation: [298098] -> [299099], done in 2767 µs
Reallocation: [299099] -> [300100], done in 2115 µs
3. ArraySize(arr)=300000 Time=219 ms
```

时间显著增加。此外，我们还会看到在哪些迭代时数组的真实容量（预留）发生更改以及如何更改。

`void ArrayFree(void &array[])`

该函数释放传递的动态数组的所有内存（包括使用 `ArrayResize` 函数的第三个参数设置的可能预留），并将其第一维度的大小设置为零。

理论上，当前块中的算法执行结束时，MQL5 中的数组会自动释放内存。无论数组是局部定义（在函数内）还是全局定义，无论其是 `fixed` 数组或动态数组，因为系统在任何情况下均将释放内存，无需编程人员的明确操作。

因此没有必要调用该函数。但有些情况下，算法中使用数组以某些内容从头重新填充，即需要在每次填充之前释放。在此情况下，此功能可能就比较方便。

请记住，如果数组元素包含动态分配对象的指针，则该函数不会删除这些对象：编程人员必须为这些对调用 `delete`（参见下文）。

我们测试上述函数：`ArrayIsDynamic`、`ArrayResize` 和 `ArrayFree`。

在 `ArrayDynamic.mq5` 脚本中，编写了 `ArrayExtend` 函数，该函数可将动态数组大小加 1，并将传递的值写入到新元素中。

```

template<typename T>
void ArrayExtend(T &array[], const T value)
{
    if(ArrayIsDynamic(array))
    {
        const int n = ArraySize(array);
        ArrayResize(array, n + 1);
        array[n] = (T)value;
    }
}

```

ArrayIsDynamic 函数用于确保仅当数组是动态数组时才更新数组。这一判断是通过条件语句完成的。**ArrayResize** 函数可让你更改数组的大小，而 **ArraySize** 函数用于找出当前大小（将在下一节讨论该函数）。

在脚本的主函数中，我们将为不同类别的数组（fixed 数组和动态数组）应用 **ArrayExtend**。

```

void OnStart()
{
    int dynamic[];
    int fixed[10] = {}; // padding with zeros

    PRT(ArrayResize(fixed, 0)); // warning: not applicable for fixed array

    for(int i = 0; i < 10; ++i)
    {
        ArrayExtend(dynamic, (i + 1) * (i + 1));
        ArrayExtend(fixed, (i + 1) * (i + 1));
    }

    Print("Filled");
    ArrayPrint(dynamic);
    ArrayPrint(fixed);

    ArrayFree(dynamic);
    ArrayFree(fixed); // warning: not applicable for fixed array

    Print("Free Up");
    ArrayPrint(dynamic); // outputs nothing
    ArrayPrint(fixed);
    ...
}

```

若代码行中调用了无法用于 **fixed** 数组的函数，计算机生成警告“无法用于静态分配数组”。务必注意的是，在 **ArrayExtend** 函数中没有此类警告，因为任何类别的数组均可被传递到该函数。这就是我们使用 **ArrayIsDynamic** 来检查的原因。

在 **OnStart** 中的一个循环后，**dynamic** 数组将扩展到 10，并且元素等于索引的平方。**fixed** 数组将保持填充零，大小不变。

使用 **ArrayFree** 释放 **fixed** 数组没有任何效果，动态数组实际上将被删除。在此情况下，最后一次打印尝试将不会在日志中生成任何行。

我们看看脚本执行结果。

```
ArrayResize(fixed,0)=0
Filled
  1   4   9  16  25  36  49  64  81 100
0 0 0 0 0 0 0 0 0 0
Free Up
0 0 0 0 0 0 0 0 0 0
```

特别需要注意的是带有对象指针的动态数组。我们定义一个简单虚拟类 *Dummy*，然后创建一个指针（指向这种对象）的数组。

```
class Dummy
{
};

void OnStart()
{
    ...
    Dummy *dummies[] = {};
    ArrayExtend(dummies, new Dummy());
    ArrayFree(dummies);
}
```

在使用一个新指针扩展了 *dummy* 数组之后，我们使用 *ArrayFree* 释放该数组，但在终端日志中有条目表明该对象还留在内存中。

```
1 undeleted objects left
1 object of type Dummy left
24 bytes of leaked memory
```

原因在于，该函数仅管理分配给该数组的内存。在此情况下，该内存保持一个指针，但该指针所指向的对象不属于该数组。换言之，如果该数组包含“外部”对象的指针，则你需要自己管理它们。例如：

```
for(int i = 0; i < ArraySize(dummies); ++i)
{
    delete dummies[i];
}
```

这种删除操作必须在调用 *ArrayFree* 之前开始。

为缩短条目，可使用以下宏（对元素循环，为每个元素调用 *delete*）：

```
#define FORALL(A) for(int _iterator_ = 0; _iterator_ < ArraySize(A); ++_iterator_)
#define FREE(P) { if(CheckPointer(P) == POINTER_DYNAMIC) delete (P); }
#define CALLALL(A, CALL) FORALL(A) { CALL(A[_iterator_]) }
```

此时指针删除操作可简化为以下表示：

```
...
CALLALL(dummies, FREE);
ArrayFree(dummies);
```

作为替代解决方案，可使用诸如 *AutoPtr* 的指针包装器类，我们已在 [对象类型模板](#) 章节讨论过。然后该数组应使用 *AutoPtr* 类型声明。由于该数组存储的是包装器对象，而不是指针，因此当该数组被清除时，每个“包装器”的析构函数将被自动调用，并释放它们的指针内存。

4.3.3 数组测量

数组的主要特性之一就是其大小，即其中的元素数量。务必要注意的是，对于多维数组，其大小是所有其维度的长度的乘积。

对于 fixed 数组，可以在编译阶段使用基于 *sizeof* 运算符的语言句法结构计算其大小：

```
sizeof(array) / sizeof(type)
```

其中，*array* 是标识符，而 *type* 是数组类型。

例如，如果在 *fixed* 代码中定义了一个数组：

```
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
```

则其大小为：

```
int n = sizeof(fixed) / sizeof(int); // 8
```

对于动态数组，此规则不适用，因为 *sizeof* 运算符为其生成的大小始终是内部动态数组对象的大小：52 字节。

请注意，在所有函数中，所有数组参数在内部均表示为动态数组包装器对象。这样做是为了让采用任何内存分配方法的数组（包括固定大小数组）都能被传递至函数。这就是 *sizeof(array)* 将为数组参数返回 52 的原因，即使通过该参数传递了一个固定大小数组。

“包装器”的存在仅影响 *sizeof*。*ArrayIsDynamic* 函数始终正确确定通过数组参数传递的实际自变量的类别。

要在程序执行阶段获取任何数组的大小，使用 *ArraySize* 函数。

```
int ArraySize(const void &array[])
```

该函数返回数组中的元素总数。数组可以是任意类型、任意维度。对于一维数组，该函数调用类似于 *ArrayRange(array, 0)*（参见下文）。

如果数组分配有保留部分（*ArrayResize* 函数的第三个参数），则其值不纳入考虑。

使用 *ArrayResize* 为动态数组分配了内存之前，*ArraySize* 函数将返回 0。同样，在为该数组调用了 *ArrayFree* 之后，大小也会变为零。

```
int ArrayRange(const void &array[], int dimension)
```

ArrayRange 函数返回指定数组维度中的元素数量。数组可以是任意类型、任意维度。*dimension* 参数必须在 0 和数组维度数量减 1 之间。索引 0 对应于第一维度，索引 1 对应于第二维度，以此类推。

ArrayRange(array, i) 所有值的乘积（其中 *i* 遍历所有维度）等于 *ArraySize(array)*。

我们看看上述函数的示例（参见 *ArraySize.mq5* 文件）。

```
void OnStart()
{
    int dynamic[];
    int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

    PRT(sizeof(fixed) / sizeof(int)); // 8
    PRT(ArraySize(fixed));           // 8

    ArrayResize(dynamic, 10);

    PRT(sizeof(dynamic) / sizeof(int)); // 13 (incorrect)
    PRT(ArraySize(dynamic));           // 10

    PRT(ArrayRange(fixed, 0));         // 2
    PRT(ArrayRange(fixed, 1));         // 4

    PRT(ArrayRange(dynamic, 0));       // 10
    PRT(ArrayRange(dynamic, 1));       // 0
    int size = 1;
    for(int i = 0; i < 2; ++i)
    {
        size *= ArrayRange(fixed, i);
    }
    PRT(size == ArraySize(fixed));     // true
}
```

4.3.4 初始化和填充数组

只有固定大小数组才能使用初始化列表进行描述。仅当由函数 *ArrayResize* 为数组分配了内存之后，才能填充动态数组。使用 *ArrayInitialize* 或 *ArrayFill* 函数填充数组。若要批量替换 *fixed* 数组或时间序列中的值，也可以在程序中使用这两个函数。

函数的使用示例在其说明之后提供。

int ArrayInitialize(type &array[], type value)

该函数将所有数组元素设置为指定值。仅支持内置数字类型的数组（*char*, *uchar*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *bool*, *color*, *datetime*, *float*, *double*）。字符串、结构和指针数组无法以这种方式填充：它们需要实现它们自己的初始化函数。一个数组可以有多个维度。

该函数返回元素数量。

如果动态数组分配有保留部分（*ArrayResize* 函数的第三个参数），则保留部分不进行初始化。

如果在数组初始化后，使用 *ArrayResize* 增加了其大小，则添加的元素将不会被自动设置为 *value*。可以使用 *ArrayFill* 函数填充这些元素。

void ArrayFill(type &array[], int start, int count, type value)

该函数以指定值填充整个或部分数字数组。具体的数组部分由参数 *start* 和 *count* 指定，分别表示元素的初始编号以及要填充的元素数量。

无论数组元素的编号顺序是否 [与时间序列中](#) 设置的一致，均不影响该函数：该属性被忽略。换言之，一个数组的元素始终从其开头到末尾计数。

对于多维数组，要获取 *start* 参数，可将所有维度的坐标转换为一个等效一维数组的穿透索引。因此，对于一个二维数组，在第一维度中具有 0 索引的元素首先被分配进内存，然后是在第一维度中具有索引 1 的元素，以此类推。*start* 的计算公式如下：

$$\text{start} = D1 * N2 + D2$$

其中 *D1* 和 *D2* 分别是第一和第二维度的索引，*N2* 是第二维度的元素数量。*D2* 变化范围是从 0 到 (*N2*-1)，*D1* 则是从 0 到 (*N1*-1)。例如，在数组 `array[3][4]` 中，索引为 `[1][3]` 的元素是一行中的第七个，因此，`ArrayFill(array, 7, 2, ...)` 调用将填充两个元素：`array[1][3]` 以及在其后的 `array[2][0]`。在图表上，这可描述如下（每个单元格包含该元素的穿透索引）：

	<code>[] [0]</code>	<code>[] [1]</code>	<code>[] [2]</code>	<code>[] [3]</code>
<code>[0] []</code>	0	1	2	3
<code>[1] []</code>	4	5	6	7
<code>[2] []</code>	8	9	10	11

`ArrayFill.mq5` 脚本提供了前述函数的用法示例。

```
void OnStart()
{
    int dynamic[];
    int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

    PRT(ArrayInitialize(fixed, -1));
    ArrayPrint(fixed);
    ArrayFill(fixed, 3, 4, +1);
    ArrayPrint(fixed);

    PRT(ArrayResize(dynamic, 10, 50));
    PRT(ArrayInitialize(dynamic, 0));
    ArrayPrint(dynamic);
    PRT(ArrayResize(dynamic, 50));
    ArrayPrint(dynamic);
    ArrayFill(dynamic, 10, 40, 0);
    ArrayPrint(dynamic);
}
```

可能结果如下所示（动态数组的未初始化元素中的随机数据将有所不同）：

```

ArrayInitialize(fixed,-1)=8
    [,0][,1][,2][,3]
[0,]  -1  -1  -1  -1
[1,]  -1  -1  -1  -1
    [,0][,1][,2][,3]
[0,]  -1  -1  -1   1
[1,]   1   1   1  -1
ArrayResize(dynamic,10,50)=10
ArrayInitialize(dynamic,0)=10
0 0 0 0 0 0 0 0 0 0
ArrayResize(dynamic,50)=50
[ 0]          0          0          0          0          0
          0          0          0          0          0
[10] -1402885947  -727144693   699739629   172950740  -1326090126
          47384          0          0          4194184          0
[20]          2          0          2          0          0
          0          0  1765933056  2084602885  -1956758056
[30]   73910037  -1937061701          56          0          56
          0   1048601  1979187200   10851          0
[40]          0          0          0  -685178880  -1720475236
          782716519  -1462194191  1434596297   415166825  -1944066819
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```

4.3.5 拷贝和编辑数组

在本节中，我们将学习如何使用内置函数插入和移除数组元素，更改其顺序以及拷贝整个数组。

bool ArrayInsert(void &target[], const void &source[], uint to, uint from = 0, uint count = WHOLE_ARRAY)

该函数将指定数量的元素从源数组（即源）拷贝到目标 *target* 数组。插入到 *target* 数组中的插入位置由 *to* 参数中的索引设置。元素的起始拷贝索引（从 *source* 数组开始进行拷贝的位置）由 *from* 索引给定。参数 *count* 中的 `WHOLE_ARRAY` 常量（(uint)-1）指定传输源数组的所有元素。

所有索引和计数均相对于数组的第一维度。换言之，对于多维数组，插入不是按单个元素执行，而是按照由“更高”维度描述的整个配置执行。例如，对于一个二维数组，*count* 参数中的值 1 表示插入一个长度等于第二维度的向量（参见示例）。

因此，目标数组和源数组必须具有相同配置。否则将会出错，拷贝也将失败。对于一维数组，这不会成为限制，但对于多维数组，务必注意第一维度以上各维度大小相等。尤其是，数组 `[[4]` 中的元素不能被插入到数组 `[[5]`，反之亦然。

该函数仅适用于固定大小或动态大小数组。不能使用此函数编辑时间序列（具有 [时间序列](#) 的数组）。禁止在 *target* 和 *source* 参数中指定相同数组。

当插入到 *fixed* 数组中时，新元素将现有元素向右偏移，并将 *count* 个最右边的元素移位到数组外面。*to* 参数必须具有一个 0 到数组大小减 1 之间的值。

插入到动态数组中时，旧元素也被向右偏移，但不会消失，因为数组本身会扩展 *count* 个元素。*to* 参数必须具有一个 0 到数组大小之间的值。如果等于数组大小，则新元素被添加到数组末尾。

指定的元素从一个数组拷贝到另一个数组，即原始数组中的元素保持不变，这些元素在新数组中的“副本”成为独立实例，与“原始”元素无任何关联。

如果成功，该函数返回 *true*，如果出错，则返回 *false*。

我们来看一些示例 (*ArrayInsert.mq5*)。 *OnStart* 函数提供了对不同配置的若干固定和动态数组的描述。

```
#define PRTS(A) Print(#A, "=", (string)(A) + " / status:" + (string)GetLastError())

void OnStart()
{
    int dynamic[];
    int dynamic2Dx5[][5];
    int dynamic2Dx4[][4];
    int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
    int insert[] = {10, 11, 12};
    int array[1] = {100};
    ...
}
```

为了方便起见，首先引入一个宏 *PRTS*，用于在调用测试指令之后立即显示错误代码（通过函数 *GetLastError* 获取）。该宏是对我们所熟悉的 *PRT* 宏的轻量修改版本。

试图在不同配置的数组之间拷贝元素会产生错误 4006 (*ERR_INVALID_ARRAY*)。

```
// you can't mix 1D and 2D arrays
PRTS(ArrayInsert(dynamic, fixed, 0)); // false:4006, ERR_INVALID_ARRAY
ArrayPrint(dynamic); // empty
// you can't mix 2D arrays of different configurations by the second dimension
PRTS(ArrayInsert(dynamic2Dx5, fixed, 0)); // false:4006, ERR_INVALID_ARRAY
ArrayPrint(dynamic2Dx5); // empty
// even if both arrays are fixed (or both are dynamic),
// size by "higher" dimensions must match
PRTS(ArrayInsert(fixed, insert, 0)); // false:4006, ERR_INVALID_ARRAY
ArrayPrint(fixed); // not changed
...
```

目标索引必须在数组内。

```
// target index 10 is out of the range of the array 'insert',
// could be 0, 1, 2, because its size = 3
PRTS(ArrayInsert(insert, array, 10)); // false:5052, ERR_SMALL_ARRAY
ArrayPrint(insert); // not changed
...
```

以下是成功的数组修改：

```

// copy second row from 'fixed', 'dynamic2Dx4' is allocated
PRTS(ArrayInsert(dynamic2Dx4, fixed, 0, 1, 1)); // true
ArrayPrint(dynamic2Dx4);
// both rows from 'fixed' are added to the end of 'dynamic2Dx4', it expands
PRTS(ArrayInsert(dynamic2Dx4, fixed, 1)); // true
ArrayPrint(dynamic2Dx4);
// memory is allocated for 'dynamic' for all elements 'insert'
PRTS(ArrayInsert(dynamic, insert, 0)); // true
ArrayPrint(dynamic);
// 'dynamic' expands by 1 element
PRTS(ArrayInsert(dynamic, array, 1)); // true
ArrayPrint(dynamic);
// new element will push the last one out of 'insert'
PRTS(ArrayInsert(insert, array, 1)); // true
ArrayPrint(insert);
}

```

日志中将会出现：

```

ArrayInsert(dynamic2Dx4, fixed, 0, 1, 1)=true
  [,0][,1][,2][,3]
[0,]  5   6   7   8
ArrayInsert(dynamic2Dx4, fixed, 1)=true
  [,0][,1][,2][,3]
[0,]  5   6   7   8
[1,]  1   2   3   4
[2,]  5   6   7   8
ArrayInsert(dynamic, insert, 0)=true
10 11 12
ArrayInsert(dynamic, array, 1)=true
10 100 11 12
ArrayInsert(insert, array, 1)=true
10 100 11

```

bool ArrayCopy(void &target[], const void &source[], int to = 0, int from = 0, int count = WHOLE_ARRAY)

该函数将 *source* 数组的部分或全部拷贝到 *target* 数组。元素在 *target* 数组中的写入位置由 *to* 参数中的索引指定。元素的起始拷贝索引 (从 *source* 数组开始进行拷贝的只) 由 *from* 索引给定。*count* 参数中的 WHOLE_ARRAY 常量 (-1) 指定传输源数组的所有元素。如果 *count* 小于零或者大于从 *from* 位置到 *source* 数组末尾的剩余元素的数量，则会拷贝整个数组剩余部分。

不同于 *ArrayInsert* 函数，*ArrayCopy* 函数不会偏移接收数组的现有元素，而是将新元素写入指定位置并覆盖旧元素。

所有索引以及元素的数量都是基于元素连续编号这一前提下设置，与数组中的维度数量及其配置无关。换言之，元素可从多维数组拷贝到一维数组，反之亦然；或者在具有不同大小的多维数组之间根据“更高”维度拷贝 (参见示例)。

该函数适用于固定和动态数组，以及被指定为 **指标缓冲区** 的时间序列数组。

允许从数组向其自身拷贝元素。但是如果 *target* 和 *source* 区域重叠，则你需要记住，迭代从左向右执行。

动态目标数组按需要自动扩展。固定数据保持其维度，被拷贝的内容必须适合数组大小，否则将会出错。

支持内置类型的数组以及具有简单类型字段的结构体数组。对于数字类型，如果源和目标类型不同，则函数将尝试转换数据。字符串数组只能被拷贝到字符串数组。类对象不允许拷贝，但对象指针可以被拷贝。

该函数返回拷贝的元素数量（出错则返回 0）。

在 *ArrayCopy.mq5* 脚本中，有该函数的若干用法示例。

```
class Dummy
{
    int x;
};

void OnStart()
{
    Dummy objects1[5], objects2[5];
    // error: structures or classes with objects are not allowed
    PRTS(ArrayCopy(objects1, objects2));
    ...
}
```

具有对象的数组会生成编译错误，指出“包含对象的结构体或类不允许拷贝”，但指针可以被拷贝。

```
Dummy *pointers1[5], *pointers2[5];
for(int i = 0; i < 5; ++i)
{
    pointers1[i] = &objects1[i];
}
PRTS(ArrayCopy(pointers2, pointers1)); // 5 / status:0
for(int i = 0; i < 5; ++i)
{
    Print(i, " ", pointers1[i], " ", pointers2[i]);
}
// it outputs some pairwise identical object descriptors
/*
0 1048576 1048576
1 2097152 2097152
2 3145728 3145728
3 4194304 4194304
4 5242880 5242880
*/
```

具有简单类型字段的结构体数组也可以正常拷贝。

```

struct Simple
{
    int x;
};

void OnStart()
{
    ...
    Simple s1[3] = {{123}, {456}, {789}}, s2[];
    PRTS(ArrayCopy(s2, s1)); // 3 / status:0
    ArrayPrint(s2);
    /*
        [x]
    [0] 123
    [1] 456
    [2] 789
    */
    ...

```

为进一步说明如何处理不同类型和配置的数组，定义了以下数组（包括固定和动态数组，以及具有不同维度的数组）：

```

int dynamic[];
int dynamic2Dx5[][5];
int dynamic2Dx4[][4];
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
int insert[] = {10, 11, 12};
double array[1] = {M_PI};
string texts[];
string message[1] = {"ok"};
...

```

从 *fixed* 数组中的位置 1（编号 2）拷贝一个元素时，整行 4 个元素被分配到接收动态数组 *dynamic2Dx4* 中，并且由于仅拷贝了 1 个元素，其余三个元素将包含随机“垃圾”（以黄色高亮显示）。

```

PRTS(ArrayCopy(dynamic2Dx4, fixed, 0, 1, 1)); // 1 / status:0
ArrayPrint(dynamic2Dx4);
/*
    [,0][,1][,2][,3]
[0,]  2  1  2  3
*/
...

```

接下来，我们从 *fixed* 数组拷贝所有元素，从第三个开始，拷贝到相同的 *dynamic2Dx4* 数组，但从位置 1 开始。由于拷贝了 5 个元素 *fixed* 数组中的总数是 8，减初始位置 3），并且它们于索引 1 处放置，将在接收数组中共占用 1 + 5 个位置，共 6 个元素。由于 *dynamic2Dx4* 数组每行有 4 个元素（在第二维度），可以仅为共是 4 的倍数个元素分配内存，即将分配 2 个额外元素，其中保留随机数据。

```

PRTS(ArrayCopy(dynamic2Dx4, fixed, 1, 3)); // 5 / status:0
ArrayPrint(dynamic2Dx4);
/*
    [,0][,1][,2][,3]
[0,]  2   4   5   6
[1,]  7   8   3   4
*/

```

当将多维数组拷贝到一维数组时，元素将以“平面”形式呈现。

```

PRTS(ArrayCopy(dynamic, fixed)); // 8 / status:0
ArrayPrint(dynamic);
/*
1 2 3 4 5 6 7 8
*/

```

将一维数组拷贝到多维数组时，元素根据接收数组的维数进行“扩展”。

```

PRTS(ArrayCopy(dynamic2Dx5, insert)); // 3 / status:0
ArrayPrint(dynamic2Dx5);
/*
    [,0][,1][,2][,3][,4]
[0,] 10  11  12   4   5
*/

```

在本例中，拷贝了 3 个元素，并且它们适配到长度为 5 个元素的一行（根据接收数组的配置）。为该系列的其余两个元素分配了内存，但未填充（包含“垃圾”）。

我们可以从另一个源覆写 *dynamic2Dx5* 数组，包括从一个不同配置的多维数组覆写。由于在接收数组中分配了两行，每行 5 个元素，而在源数组中分配了 2 行，每行 4 个元素，所以还有 2 个额外元素未填充。

```

PRTS(ArrayCopy(dynamic2Dx5, fixed)); // 8 / status:0
ArrayPrint(dynamic2Dx5);
/*
    [,0][,1][,2][,3][,4]
[0,]  1   2   3   4   5
[1,]  6   7   8   0   0
*/

```

使用 *ArrayCopy*，可以更改固定接收数组中的元素。

```

PRTS(ArrayCopy(fixed, insert)); // 3 / status:0
ArrayPrint(fixed);
/*
    [,0][,1][,2][,3]
[0,] 10  11  12   4
[1,]  5   6   7   8
*/

```

在这里，我们已经覆写了 *fixed* 数组中的前三个元素。然后我们覆写后 3 个。

```

PRTS(ArrayCopy(fixed, insert, 5)); // 3 / status:0
ArrayPrint(fixed);
/*
    [,0][,1][,2][,3]
[0,]  10  11  12   4
[1,]   5  10  11  12
*/

```

无法拷贝到一个等于 `fixed` 数组长度的位置 (在此情况下, 动态目标数组将扩展)。

```

PRTS(ArrayCopy(fixed, insert, 8)); // 4006, ERR_INVALID_ARRAY
ArrayPrint(fixed); // no changes

```

与其它类型的数组组合的字符串数组将会抛出错误:

```

PRTS(ArrayCopy(texts, insert)); // 5050, ERR_INCOMPATIBLE_ARRAYS
ArrayPrint(texts); // empty

```

但是可以在字符串数组之间进行拷贝:

```

PRTS(ArrayCopy(texts, message));
ArrayPrint(texts); // "ok"

```

不同数值类型的数组通过必要转换进行拷贝。

```

PRTS(ArrayCopy(insert, array, 1)); // 1 / status:0
ArrayPrint(insert); // 10  3 12

```

在这里, 我们已将数字 `Pi` 写入一个整数数组, 因此接收到了值 3 (其替换了 11)。

`bool ArrayRemove(void &array[], uint start, uint count = WHOLE_ARRAY)`

该函数从数组的 **start** 索引开始删除指定数量的元素。数组可以是多维的, 支持任何内置类型或字段为内置类型的结构体类型, 但字符串除外。

start 索引和 **count** 数量指数组的第一维度。换言之, 对于多维数组, 删除不是按单个元素执行, 而是按照由“更高”维度描述的整个配置执行。例如, 对于一个二维数组, **count** 参数中的值 1 表示删除一个长度等于第二维度的整个系列 (参见示例)。

值 **start** 必须在 0 和第一维度大小减 1 之间。

该函数不能应用于具有时间序列 (内置 [时间序列](#) 或 [指标缓冲区](#) 中的时间序列)。

为测试该函数, 我们准备了 `ArrayRemove.mq5` 脚本。特别要注意的是, 该脚本定义了 2 个结构体:

```

struct Simple
{
    int x;
};

struct NotSoSimple
{
    int x;
    string s; // a field of type string causes the compiler to make an implicit destru
};

```

具有简单结构体的数组可由 **ArrayRemove** 函数成功处理，而具有析构函数的对象数组（即使具有隐式解构函数，如在 **NotSoSimple** 中）会导致错误：

```

void OnStart()
{
    Simple structs1[10];
    PRTS(ArrayRemove(structs1, 0, 5)); // true / status:0

    NotSoSimple structs2[10];
    PRTS(ArrayRemove(structs2, 0, 5)); // false / status:4005,
                                     // ERR_STRUCT_WITHOBJECTS_ORCLASS
    ...
}

```

接下来，定义并初始化各种配置的数组。

```

int dynamic[];
int dynamic2Dx4[][4];
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

// make 2 copies
ArrayCopy(dynamic, fixed);
ArrayCopy(dynamic2Dx4, fixed);

// show initial data
ArrayPrint(dynamic);
/*
1 2 3 4 5 6 7 8
*/
ArrayPrint(dynamic2Dx4);
/*
      [,0][,1][,2][,3]
[0,]   1   2   3   4
[1,]   5   6   7   8
*/

```

从 **fixed** 数组中删除时，被移除片断之后的所有元素都将向左偏移。请务必要注意，数组的大小不会改变，因此，偏移的元素的副本会重复出现。

```

PRTS(ArrayRemove(fixed, 0, 1));
ArrayPrint(fixed);
/*
ArrayRemove(fixed,0,1)=true / status:0
    [,0][,1][,2][,3]
[0,]  5   6   7   8
[1,]  5   6   7   8
*/

```

我们将二维数组 *fixed* 的第一维的一个元素以偏移 0 (即初始行) 移除。下一行的元素上移, 保留在相同行中。

如果我们对动态数组 (内容与 *fixed* 数组相同) 执行相同操作, 其大小将自动减少移除的元素数量。

```

PRTS(ArrayRemove(dynamic2Dx4, 0, 1));
ArrayPrint(dynamic2Dx4);
/*
ArrayRemove(dynamic2Dx4,0,1)=true / status:0
    [,0][,1][,2][,3]
[0,]  5   6   7   8
*/

```

在一维数组中, 每个移除的元素对应一个值。例如, 在 *dynamic* 数组中, 当从索引 2 开始移除三个元素时, 我们获得以下结果:

```

PRTS(ArrayRemove(dynamic, 2, 3));
ArrayPrint(dynamic);
/*
ArrayRemove(dynamic,2,3)=true / status:0
1 2 6 7 8
*/

```

值 3、4、5 已被移除, 数组大小减少了 3。

bool ArrayReverse(void &array[], uint start = 0, uint count = WHOLE_ARRAY)

该函数可反转数组中指定元素的顺序。要反转的元素由 *start* 起始位置和 *count* 数量确定。如果 *start* = 0 且 *count* = *WHOLE_ARRAY*, 则访问整个数组。

支持任意维数和类型的数组, 无论是固定还是动态类型 (包括 [指标缓冲区](#) 中的时间序列)。数组可能包含对象、指针或结构体。对于多维数组, 仅第一维被反转。

count 值必须在 0 到第一维度中的元素数量之间。请注意, 小于 2 的 *count* 将不会有明显的影响, 但它可用于统一算法中的循环。

如果成功, 该函数返回 *true*, 如果出错, 则返回 *false*。

ArrayReverse.mq5 脚本可用于测试该函数。在其开头, 定义了一个类, 用于生成存储在数组中的对象。可以包含字符串和其它“复数”字段。


```

class Dummy
{
    static int counter;
    int x;
    string s; // a field of type string causes the compiler to create an implicit dest
public:
    Dummy() { x = counter++; }
};

static int Dummy::counter;

```

对象以序列号识别 (在创建时赋予)。

```

void OnStart()
{
    Dummy objects[5];
    Print("Objects before reverse");
    ArrayPrint(objects);
    /*
        [x]  [s]
    [0]    0 null
    [1]    1 null
    [2]    2 null
    [3]    3 null
    [4]    4 null
    */
}

```

在应用 *ArrayReverse* 之后, 我们获得预期的对象反转顺序。

```

PRTS(ArrayReverse(objects)); // true / status:0
Print("Objects after reverse");
ArrayPrint(objects);
/*
        [x]  [s]
    [0]    4 null
    [1]    3 null
    [2]    2 null
    [3]    1 null
    [4]    0 null
    */

```

接下来, 准备了不同配置的数值数组, 并以不同参数展开。

```

int dynamic[];
int dynamic2Dx4[][4];
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

ArrayCopy(dynamic, fixed);
ArrayCopy(dynamic2Dx4, fixed);

PRTS(ArrayReverse(fixed)); // true / status:0
ArrayPrint(fixed);
/*
    [,0][,1][,2][,3]
[0,]  5   6   7   8
[1,]  1   2   3   4
*/

PRTS(ArrayReverse(dynamic, 4, 3)); // true / status:0
ArrayPrint(dynamic);
/*
1 2 3 4 7 6 5 8
*/

PRTS(ArrayReverse(dynamic, 0, 1)); // does nothing (count = 1)
PRTS(ArrayReverse(dynamic2Dx4, 2, 1)); // false / status:5052, ERR_SMALL_ARRAY
}

```

在后者情况下，值 *start* (2) 超过了第一维度的大小，因此出错。

4.3.6 移动 (交换) 数组

MQL5 能够以一种资源高效的方式来调换两个数组的内容 (无需物理分配内存和复制数据)。在某些其它编程语言中，类似的操作不仅支持数组，而且还支持其它变量，这种操作称为移动。

bool ArraySwap(void &array1[], void &array2[])

该函数可调换相同类型的两个动态数组的内容。支持任何类型的数组。但是，该函数不适用于时间序列数组和指标缓冲区，以及具有 *const* 修饰符的任何数组。

对于多维数组，除第一个维度以外，其他所有维度中的元素数量必须匹配。

如果成功，该函数返回 *true*，如果出错，则返回 *false*。

该函数的主要用途是，在已确认不再需要源数组的情况下，避免在传递给函数或从函数返回时对数组进行物理拷贝，从而提高程序的运行速度。事实上，交换几乎是即时完成，因为应用程序数据不以任何方式移动。真正发生交换的是存储在服务结构中的关于数组的元数据 (该数据仅占 52 字节)，这些结构描述了动态数组。

假设有一个用于通过某种算法处理数组的类。同一数组可能需进行不同操作，因此最好将其保留为类成员。但是问题来了，如何将其传输到一个对象？在 MQL5 中，相关方法 (一般来说也就是函数) 允许仅通过引用来传递数组。不考虑包含数组并由指针传递的各种包装器类，唯一的简单解决方案似乎就是：例如，在类构造函数中描述数组参数，并使用 *ArrayCopy* 将其拷贝到内部数组。但使用 *ArraySwap* 更加高效。

```

template<typename T>
class Worker
{
    T array[];

public:
    Worker(T &source[])
    {
        // ArrayCopy(array, source); // memory and time consuming
        ArraySwap(source, array);
    }
    ...
};

```

由于在交换之前 *array* 数组为空，因此在操作之后，用作 *source* 自变量的数组将变为空，而 *array* 将被填充输入数据，几乎没有任何开销。

在类的对象成为数组的“所有者”之后，我们可以使用所需的算法修改它，例如，通过特殊方法 *process* 来修改，该方法将请求的算法的代码作为一个参数。它可能是排序、平滑、混合、添加噪点等等。但是，首先我们尝试使用 *ArrayReverse* 函数测试关于数组逆反的简单操作概念（参见 *ArraySwapSimple.mq5* 文件）。

```

bool process(const int mode)
{
    if(ArraySize(array) == 0) return false;
    switch(mode)
    {
        case -4:
            // example: shuffling
            break;
        case -3:
            // example: logarithm
            break;
        case -2:
            // example: adding noise
            break;
        case -1:
            ArrayReverse(array);
            break;
        ...
    }
    return true;
}

```

你可以使用两种方法提供对处理结果的访问：逐元素（通过重载 `[]` 运算符），或者通过整个数组（我们再次在对应方法 *get* 中使用 *ArraySwap*，但也可以通过 *ArrayCopy* 提供拷贝方法）。

```

T operator[](int i)
{
    return array[i];
}

void get(T &result[])
{
    ArraySwap(array, result);
}

```

为了通用性，该类被制作成模板。这将允许将来将其改用于任意结构的数组，但在目前，你可以检查 *double* 类型的简单数组的逆反：

```

void OnStart()
{
    double data[];
    ArrayResize(data, 3);
    data[0] = 1;
    data[1] = 2;
    data[2] = 3;

    PRT(ArraySize(data));          // 3
    Worker<double> simple(data);
    PRT(ArraySize(data));          // 0
    simple.process(-1); // reversing array

    double res[];
    simple.get(res);
    ArrayPrint(res); // 3.000000 2.000000 1.000000
}

```

排序任务更现实，对于结构体数组，可能需要按任何字段排序。在 [下一章节](#) 我们将详细了解 *ArraySort* 函数，该函数可让你升序排列任何内置类型的数组，但结构数组除外。届时我们将尝试消除这一“缺口”，让 *ArraySwap* 实际运行起来。

4.3.7 数组的比较、排序和搜索

MQL5 API 包含若干函数，可用于比较和排序数组，以及在数组中搜索最大值、最小值或任何特定值。

```
int ArrayCompare(const void &array1[], const void &array2[], int start1 = 0, int start2 = 0, int count = WHOLE_ARRAY)
```

该函数返回比较两个数组的比较结果，可以是内置类型数组或具有内置类型字段的结构体数组（字符串除外）。不支持类对象数组。此外，不能比较包含动态数组、类对象或指针的结构体数组。

默认情况下，为整个数组执行比较，但如果必要，可以指定数组的某一部分，为此，可以使用参数 *start1*（在第一个数组中的开始位置）、*start2*（在第二个数组中的开始位置）以及 *count*。

数组可以是固定或动态数组，也可以是多维数组。在比较期间，多维数组被表示为等效一维数组（例如，对于二维数组，第二行的元素跟随第一行的元素，第三行的元素跟随第二行，依此类

推)。因此，用于多维数组的 *start1*、*start2* 和 *count* 参数通过元素编号指定，而不是通过沿第一维度的索引指定。

使用各种 *start1* 和 *start2* 偏移，可以比较同一数组的不同部分。

逐元素比较数组，直至找到第一个不一致，或者直至到达某个数组的末尾。两个元素（在两个数组中的位置相同）之间的关系取决于类型：对于数字，使用运算符 '>'、'<'、'=='，而对于字符串，使用 *StringCompare* 函数。结构体通过逐字节比较，等效于为每对元素执行以下代码：

```
uchar bytes1[], bytes2[];
StructToCharArray(array1[i], bytes1);
StructToCharArray(array2[i], bytes2);
int cmp = ArrayCompare(bytes1, bytes2);
```

基于第一批差异元素的比率，获得 *array1* 和 *array2* 数组的批量比较结果。如果未找到差异，并且数组的长度相等，则数组被视为相同。如果长度不同，则较长的数组被视为更大。

如果 *array1* 小于 *array2*，函数返回 -1；如果 *array1* 大于 *array2*，返回 +1；如果相等，返回 0。

如果出错，结果为 -2。

我们看看 *ArrayCompare.mq5* 脚本中的一些示例。

我们创建一个简单结构，用引填充要比较的数组。

```
struct Dummy
{
    int x;
    int y;

    Dummy()
    {
        x = rand() / 10000;
        y = rand() / 5000;
    }
};
```

类字段填充了随机数字（每次运行该脚本，我们将接收到新值）。

在 *OnStart* 函数中，我们描述一个小结构体数组，并逐个比较连续元素（以 1 元素的长度移动数组的相邻片断）。

```

#define LIMIT 10

void OnStart()
{
    Dummy a1[LIMIT];
    ArrayPrint(a1);

    // pairwise comparison of neighboring elements
    // -1: [i] < [i + 1]
    // +1: [i] > [i + 1]
    for(int i = 0; i < LIMIT - 1; ++i)
    {
        PRT(ArrayCompare(a1, a1, i, i + 1, 1));
    }
    ...
}

```

下面是其中一个数组选项的结果 (为方便分析, 具有 “大于”(+1)/ “小于”(-1) 符号的列直接被添加到数组内容的右边):

	[x]	[y]	// result
[0]	0	3	// -1
[1]	2	4	// +1
[2]	2	3	// +1
[3]	1	6	// +1
[4]	0	6	// -1
[5]	2	0	// +1
[6]	0	4	// -1
[7]	2	5	// +1
[8]	0	5	// -1
[9]	3	6	

比较数组的两个对半部分, 给出的结果为 -1:

```

// compare first and second half
PRT(ArrayCompare(a1, a1, 0, LIMIT / 2, LIMIT / 2)); // -1

```

接下来, 我们将比较具有预定义数据的字符串数组。

```

string s[] = {"abc", "456", "$"};
string s0[][3] = {{{"abc", "456", "$"}}};
string s1[][3] = {{{"abc", "456", ""}}};
string s2[][3] = {{{"abc", "456"}}}; // last element omitted: it is null
string s3[][2] = {{{"abc", "456"}}};
string s4[][2] = {{{"aBc", "456"}}};

PRT(ArrayCompare(s0, s)); // s0 == s, 1D and 2D arrays contain the same data
PRT(ArrayCompare(s0, s1)); // s0 > s1 since "$" > ""
PRT(ArrayCompare(s1, s2)); // s1 > s2 since "" > null
PRT(ArrayCompare(s2, s3)); // s2 > s3 due to different lengths: [3] > [2]
PRT(ArrayCompare(s3, s4)); // s3 < s4 since "abc" < "aBc"

```

最后, 我们检查数组片段比率:

```
PRT(ArrayCompare(s0, s1, 1, 1, 1)); // second elements (with index 1) are equal
PRT(ArrayCompare(s1, s2, 0, 0, 2)); // the first two elements are equal
```

bool ArraySort(void &array[])

该函数按第一维度对数字数组（包括可能的多维数组）排序。排序顺序为升序。要将数组降序排序，将 [ArrayReverse](#) 函数应用到生成的数组，或者以逆序处理数组。

该函数不支持由字符串、结构或类构成的数组。

如果成功，该函数返回 *true*，如果出错，则返回 *false*。

如果为数组设置了“时间序列”属性，则其中的元素以逆序索引（详见章节 [数组中时间序列的索引方向](#)），这对排序顺序具有“外部”逆反效应：当直接处理此类数组时，将会获得降序值。在物理层面，数组始终以升序排序，这也是数组的存储方式。

在 *ArraySort.mq5* 脚本中，生成一个 10 × 3 的二维数组并使用 *ArraySort* 排序：

```
#define LIMIT 10
#define SUBLIMIT 3

void OnStart()
{
    // generating random data
    int array[][SUBLIMIT];
    ArrayResize(array, LIMIT);
    for(int i = 0; i < LIMIT; ++i)
    {
        for(int j = 0; j < SUBLIMIT; ++j)
        {
            array[i][j] = rand();
        }
    }

    Print("Before sort");
    ArrayPrint(array);    // source array

    PRTS(ArraySort(array));

    Print("After sort");
    ArrayPrint(array);    // ordered array
    ...
}
```

根据日志，第一列以升序排序（由于是随机生成，具体数字会变化）：

```

Before sort
      [,0] [,1] [,2]
[0,]  8955  2836 20011
[1,]  2860  6153 25032
[2,] 16314  4036 20406
[3,] 30366 10462 19364
[4,] 27506  5527 21671
[5,]  4207  7649 28701
[6,]  4838   638 32392
[7,] 29158 18824 13536
[8,] 17869 23835 12323
[9,] 18079  1310 29114
ArraySort(array)=true / status:0
After sort
      [,0] [,1] [,2]
[0,]  2860  6153 25032
[1,]  4207  7649 28701
[2,]  4838   638 32392
[3,]  8955  2836 20011
[4,] 16314  4036 20406
[5,] 17869 23835 12323
[6,] 18079  1310 29114
[7,] 27506  5527 21671
[8,] 29158 18824 13536
[9,] 30366 10462 19364

```

后续列中的值已与第一列中的“先导”值同步移动。换言之，全部行都进行置换，尽管仅第一列是排序标准。

但如果你想对一个二维数组排序，但又不想将第一个列作为排序标准，那该怎么办？可以为此编写一个专门算法。其中一个选项作为模板函数包括在 *ArraySort.mq5* 文件中：


```

template<typename T>
bool ArraySort(T &array[], const int column)
{
    if(!ArrayIsDynamic(array)) return false;

    if(column == 0)
    {
        return ArraySort(array); // standard function
    }

    const int n = ArrayRange(array, 0);
    const int m = ArrayRange(array, 1);

    T temp[][2];

    ArrayResize(temp, n);
    for(int i = 0; i < n; ++i)
    {
        temp[i][0] = array[i][column];
        temp[i][1] = i;
    }

    if(!ArraySort(temp)) return false;

    ArrayResize(array, n * 2);
    for(int i = n; i < n * 2; ++i)
    {
        ArrayCopy(array, array, i * m, (int)(temp[i - n][1] + 0.1) * m, m);
        /* equivalent
        for(int j = 0; j < m; ++j)
        {
            array[i][j] = array[(int)(temp[i - n][1] + 0.1)][j];
        }
        */
    }

    return ArrayRemove(array, 0, n);
}

```

给定函数仅对动态数组有效，因为 `array` 的大小翻倍以容纳数组后半部分中的中间结果，并且最后，使用 `ArrayRemove` 移除前半部分（原始部分）。这就是为什么 `OnStart` 函数中的原始测试数组通过 `ArrayResize` 分配空间。

我们鼓励你自己研究排序原理（或者查阅相关资料）。

对于具有大量维度的数组（如 `array[][][]`），应实施相似的操作。

回想在前一节中，我们提出了按任意字段对结构体数组排序的问题。如我们所知，标准 `ArraySort` 函数无法完成这一任务。我们尝试找到一种变通路线。我们以前一节中 `ArraySwapSimple.mq5` 文件中的类为基础。我们将其拷贝到 `ArrayWorker.mq5`，并添加需要的代码。

在 `Worker::process` 方法中，我们将提供对辅助排序方法 `arrayStructSort` 的调用，要排序的字段将以编号指定（具体操作将在下文介绍）：

```
...
bool process(const int mode)
{
    ...
    switch(mode)
    {
        ...
        case -1:
            ArrayReverse(array);
            break;
        default: // sorting by field number 'mode'
            arrayStructSort(mode);
            break;
    }
    return true;
}

private:
bool arrayStructSort(const int field)
{
    ...
}
```

现在应该明白为什么在 `process` 方法中所有之前的模式 (`mode` 参数的值)均为负数了：零和正数值保留用于排序，并对应于“列”号。

对结构体数组排序的概念来自于对二维数组排序。我们只需要以某种方式将单个结构体映射到一维数组（代表二维数组的一行）。为此，首先要决定数组应为哪种类型。

由于 `worker` 类已经是模板，我们将向该模板再添加一个参数，以便可以灵活设置数组类型。

```
template<typename T, typename R>
class Worker
{
    T array[];
    ...
}
```

现在，我们回到 **联合体**，联合体将不同类型的变量相互重叠。这样，我们得到以下巧妙的构造：

```
union Overlay
{
    T r;
    R d[sizeof(T) / sizeof(R)];
};
```

在该联合体中，结构体的类型与一个 `R` 类型的数组组合，其大小由编译器根据两种类型 `T` 和 `R` 的大小比率自动计算。

现在，在 `arrayStructSort` 方法中，我们能够部分复制二维数组排序的代码。

```

bool arrayStructSort(const int field)
{
    const int n = ArraySize(array);

    R temp[][2];
    Overlay overlay;

    ArrayResize(temp, n);
    for(int i = 0; i < n; ++i)
    {
        overlay.r = array[i];
        temp[i][0] = overlay.d[field];
        temp[i][1] = i;
    }
    ...
}

```

我们弃用了原始结构的数组，而是准备了 R 类型的 `temp[][2]` 数组，将其扩展至 `array` 中的记录数，并在循环中写入以下内容：每行第 0 个索引处“显示”必须的 `field` 字段，在第 1 个索引处“显示”该元素的原始索引。

该“显示”基于以下事实：结构体中的字段通常以某种方式对齐，因为它们使用标准型。因此，对于适当选择的 R 类型，能够在“重叠”的数组元素中提供全部或部分字段命中。

例如，在标准结构 `MqlRates` 中，前 6 个字段大小为 8 字节，因此正确映射到 `double` 或 `long` 数组（这些是 R 模板类型的候选）。

```

struct MqlRates
{
    datetime time;
    double open;
    double high;
    double low;
    double close;
    long tick_volume;
    int spread;
    long real_volume;
};

```

对于最后两个字段，情况更复杂。如果使用 `int` 类型作为 R 仍然能访问 `spread` 字段，则 `real_volume` 字段的偏移量就不是自身大小倍数（因为其前面的字段类型是 `int`，即 4 字节）。这些是特定方法的问题。可以改进这些方法，或者创造另一种方法。

我们继续讨论排序算法。填充了 `temp` 数组之后，可以使用常规函数 `ArraySort` 对其排序，然后原始索引可用于以正确结构顺序形成新的数组。

```

...
if(!ArraySort(temp)) return false;
T result[];

ArrayResize(result, n);
for(int i = 0; i < n; ++i)
{
    result[i] = array[(int)(temp[i][1] + 0.1)];
}

return ArraySwap(result, array);
}

```

在退出该函数之前，我们再次使用 *ArraySwap*，以高效利用资源的方式，用某些新的有序内容替换内部对象数组 *array* 的内容，这些内容在局部数组 *result* 中获得。

我们在操作中检查 *worker* 类：在函数 *OnStart* 中，我们定义结构体数组 *MqlRates*，并向终端请求数千条记录。

```

#define LIMIT 5000

void OnStart()
{
    MqlRates rates[];
    int n = CopyRates(_Symbol, _Period, 0, LIMIT, rates);
    ...
}

```

CopyRates 函数将在 [单独章节](#) 中介绍。目前而言，我们只需要知道，该函数用交易品种报价和对当前图表运行脚本的时间范围来填充传递的 *rates* 数组。LIMIT 宏指定请求的柱数量：你需要确保该值不大于你的终端对于每个窗口中的柱数量的设置。

为处理接收的数据，我们将创建一个类型为 *T=MqlRates* 和 *R=double* 的对象 *worker*：

```
Worker<MqlRates, double> worker(rates);
```

可以使用以下格式的指令开始排序：

```
worker.process(offsetof(MqlRates, open) / sizeof(double));
```

这里我们使用 *offsetof* 运算符获取结构体内 *open* 字段的字节偏移。它进一步除以 *double* 大小，并提供正确的“列”号用于按 *open* 价格排序。可以逐元素读取排序结果，也可以获取整个数组：

```

Print(worker[i].open);
...
worker.get(rates);
ArrayPrint(rates);

```

请注意，通过 *get* 方法获取数组会使用 *ArraySwap* 将其从内部数组 *array* 中移除到外部数组（作为自变量传递）。因此，在此之后的 *worker.process()* 调用没有意义：在对象 *worker* 中没有更多数据。

为简化按不同字段的排序启动过程，实现了辅助函数 *sort*：

```

void sort(Worker<MqlRates, double> &worker, const int offset, const string title)
{
    Print(title);
    worker.process(offset);
    Print("First struct");
    StructPrint(worker[0]);
    Print("Last struct");
    StructPrint(worker[worker.size() - 1]);
}

```

该函数向日志输出一个标头以及已排序数组的第一个和最后一个元素。在该函数的帮助下，在 *OnStart* 中对三个字段的测试如下：

```

void OnStart()
{
    ...
    Worker<MqlRates,double>worker(rates);
    sort(worker,offsetof(MqlRates,open) /sizeof(double),"Sorting by open price...");
    sort(worker,offsetof(MqlRates,tick_volume) /sizeof(double),"Sorting by tick volume...");
    sort(worker,offsetof(MqlRates,time) /sizeof(double),"Sorting by time...");
}

```

遗憾的是，标准函数 *print* 不支持打印单个结构体，并且在 MQL5 中没有内置函数 *StructPrint*。因此，我们必须基于 *ArrayPrint* 自己编写：事实上，将结构体放在一个大小为 1 的数组中即可。

```

template<typename S>
void StructPrint(const S &s)
{
    S temp[1];
    temp[0] = s;
    ArrayPrint(temp);
}

```

运行该脚本之后，我们可以获得类似如下的结果（取决于终端设置，即其执行所针对的交易品种/时间范围）：

Sorting by open price...

First struct

```
[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.07.21 10:30:00 1.17557 1.17584 1.17519 1.17561 1073 0 0
```

Last struct

```
[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.05.25 15:15:00 1.22641 1.22664 1.22592 1.22618 852 0 0
```

Sorting by tick volume...

First struct

```
[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.05.24 00:00:00 1.21776 1.21811 1.21764 1.21794 52 20 0
```

Last struct

```
[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.06.16 21:30:00 1.20436 1.20489 1.20149 1.20154 4817 0 0
```

Sorting by time...

First struct

```
[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.05.14 16:15:00 1.21305 1.21411 1.21289 1.21333 888 0 0
```

Last struct

```
[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.07.27 22:45:00 1.18197 1.18227 1.18191 1.18225 382 0 0
```

这里是 EURUSD,M15 的数据。

上述排序实现可能是最快的方式之一，因为它使用内置的 *ArraySort*。

然而，如果难以对齐结构体字段，或者对于将结构“映射”到数组的方法存疑，从而迫使我们放弃这一方法（继而放弃 *ArraySort* 函数），则经证实的 DIY 方法仍然可使用。

有大量的排序算法可轻松适配 MQL5。本书随附的 *QuickSortStructT.mqh* 文件中提供了一种快速排序选择。它就是改进版的 *QuickSortT.mqh*，我们在 [字符串比较](#) 的章节中介绍。它具有模板类 *QuickSortStructT* 的 *Compare* 方法，其完全是虚拟的，必须在子代类中重新定义，以便为要求的类型及其字段返回比较运算符 ‘>’ 的类似结果。为方便用户，在头文件中创建了一个宏：

```
#define SORT_STRUCT(T, A, F) \
{ \
    class InternalSort : public QuickSortStructT<T> \
    { \
        virtual bool Compare(const T &a, const T &b) override \
        { \
            return a.##F > b.##F; \
        } \
    } sort; \
    sort.QuickSort(A); \
}
```

使用该宏，要按给定字段对结构数组排序，编写一条指令就行了。例如：

```
MqlRates rates[];
CopyRates(_Symbol, _Period, 0, 10000, rates);
SORT_STRUCT(MqlRates, rates, high);
```

其中，*MqlRates* 类型的 *rates* 数组按 *high* 价格排序。

int ArrayBsearch(const type &array[], type value)

该函数在数值数组中搜索给定值。支持所有内置数值类型的数组。数组必须按第一维度以升序排序，否则结果将不正确。

该函数返回匹配元素的索引（如果有多个匹配元素，对返回其中第一个的索引）或者值最接近的元素的索引（如果没有完全匹配项），也就是说，可能是值大于或小于搜索值的元素。如果期望值小于第一个（最小值），则返回 0。如果搜索值大于最后一个（最大值），则返回其索引。

该索引取决于数组中元素的编号方向：正向（从开头到末尾）或反向（从末尾到开头）。可以使用数组中时间序列的索引方向一节中描述的函数确认和更改。

如果出错，则返回 -1。

对于多维数组，搜索限于第一维度。

在 *ArraySearch.mq5* 脚本中，可以找到 *ArrayBsearch* 函数的用法示例。

```
void OnStart()
{
    int array[] = {1, 5, 11, 17, 23, 23, 37};
    // indexes 0 1 2 3 4 5 6
    int data[][2] = {{1, 3}, {3, 2}, {5, 10}, {14, 10}, {21, 8}};
    // indexes 0 1 2 3 4
    int empty[];
    ...
}
```

对于三个预定义数组（其中一个为空），执行以下语句：

```
PRTS(ArrayBsearch(array, -1)); // 0
PRTS(ArrayBsearch(array, 11)); // 2
PRTS(ArrayBsearch(array, 12)); // 2
PRTS(ArrayBsearch(array, 15)); // 3
PRTS(ArrayBsearch(array, 23)); // 4
PRTS(ArrayBsearch(array, 50)); // 6

PRTS(ArrayBsearch(data, 7)); // 2
PRTS(ArrayBsearch(data, 9)); // 2
PRTS(ArrayBsearch(data, 10)); // 3
PRTS(ArrayBsearch(data, 11)); // 3
PRTS(ArrayBsearch(data, 14)); // 3

PRTS(ArrayBsearch(empty, 0)); // -1, 5053, ERR_ZEROSIZE_ARRAY
...
```

此外，在 *populateSortedArray* 辅助函数中，*numbers* 数组以随机值填充，并且使用 *ArrayBsearch* 将该数组持续保持为已排序状态。

```

void populateSortedArray(const int limit)
{
    double numbers[]; // array to fill
    double element[1]; // new value to insert

    ArrayResize(numbers, 0, limit); // allocate memory beforehand

    for(int i = 0; i < limit; ++i)
    {
        // generate a random number
        element[0] = NormalizeDouble(rand() * 1.0 / 32767, 3);
        // find where its place in the array
        int cursor = ArrayBsearch(numbers, element[0]);
        if(cursor == -1)
        {
            if(_LastError == 5053) // empty array
            {
                ArrayInsert(numbers, element, 0);
            }
            else break; // error
        }
        else
        if(numbers[cursor] > element[0]) // insert at 'cursor' position
        {
            ArrayInsert(numbers, element, cursor);
        }
        else // (numbers[cursor] <= value) // insert after 'cursor'
        {
            ArrayInsert(numbers, element, cursor + 1);
        }
    }
    ArrayPrint(numbers, 3);
}

```

每个新值首先进入一个元素的数组 *element*，因为这样更方便将其插入生成的 *numbers* 数组中（使用函数 *ArrayInsert*）。

ArrayBsearch 可让你确定应插入新值的位置。

该函数的结果显示在日志中：


```

void OnStart()
{
    ...
    populateSortedArray(80);
    /*
     example (will be different on each run due to randomization)
    [ 0] 0.050 0.065 0.071 0.106 0.119 0.131 0.145 0.148 0.154 0.159
         0.184 0.185 0.200 0.204 0.213 0.216 0.220 0.224 0.236 0.238
    [20] 0.244 0.259 0.267 0.274 0.282 0.293 0.313 0.334 0.346 0.366
         0.386 0.431 0.449 0.461 0.465 0.468 0.520 0.533 0.536 0.541
    [40] 0.597 0.600 0.607 0.612 0.613 0.617 0.621 0.623 0.631 0.634
         0.646 0.658 0.662 0.664 0.670 0.670 0.675 0.686 0.693 0.694
    [60] 0.725 0.739 0.759 0.762 0.768 0.783 0.791 0.791 0.791 0.799
         0.838 0.850 0.854 0.874 0.897 0.912 0.920 0.934 0.944 0.992
    */
}

```

```
int ArrayMaximum(const type &array[], int start = 0, int count = WHOLE_ARRAY)
```

```
int ArrayMinimum(const type &array[], int start = 0, int count = WHOLE_ARRAY)
```

ArrayMaximum 和 **ArrayMinimum** 函数分别在数值数组中搜索具有最大值和最小值的元素。搜索索引范围由 **start** 和 **count** 参数设置：如果是默认值，则搜索整个数组。

该函数返回找到的元素的位置。

如果为数组设置了“系列”属性（时间序列），则以反序对元素进行索引，这会影响该函数的结果（参见示例）。用于处理“系列”属性的内置函数在 [下一章节](#) 讨论。有关“系列”数组的更多细节将在关于 [时间序列](#) 和 [指标](#) 的章节中讨论。

在多维数组中，对第一维度执行搜索。

如果在具有最大值或最小值的数组中有若干个相同的元素，则函数将返回其中第一个的索引。

函数用法示例在 **ArrayMaxMin.mq5** 文件中提供。

```

#define LIMIT 10

void OnStart()
{
    // generating random data
    int array[];
    ArrayResize(array, LIMIT);
    for(int i = 0; i < LIMIT; ++i)
    {
        array[i] = rand();
    }

    ArrayPrint(array);
    // by default, the new array is not a timeseries
    PRTS(ArrayMaximum(array));
    PRTS(ArrayMinimum(array));
    // turn on the "serial" property
    PRTS(ArraySetAsSeries(array, true));
    PRTS(ArrayMaximum(array));
    PRTS(ArrayMinimum(array));
}

```

该脚本将记录类似下列字符串的内容（由于数据随机生成，每次运行都不同）：

```

22242 5909 21570 5850 18026 24740 10852 2631 24549 14635
ArrayMaximum(array)=5 / status:0
ArrayMinimum(array)=7 / status:0
ArraySetAsSeries(array,true)=true / status:0
ArrayMaximum(array)=4 / status:0
ArrayMinimum(array)=2 / status:0

```

4.3.8 数组中时间序列的索引方向

由于交易中存在的细节，MQL5 为处理数组引入了附加功能。其中一个功能是数组元素可包含对应于时间点的数据。例如，包括具有金融工具报价、价格最小变动单位以及技术指标读数的数组。数据的时间顺序意味着新元素会不断添加到数组末尾，其索引也随之递增。

然而，从交易角度而言，从现在到过去进行计数更为方便。这样一来，元素 0 始终包含最近、最新的值，元素 1 始终包含前一个值，以此类推。

MQL5 允许动态选择和切换数组索引方向。从现在到过去编号的数组称为时间序列。如果索引是从过去到现在递增，则属于常规数组。在时间序列中，时间随索引的增加而递减。在普通数组中，时间如现实生活中一样递增。

务必要注意，不要求数组必须包含时间相关值才能够切换寻址顺序。只不过该功能最常用于处理历史数据，事实上也正是为此类需求而设计。

此数组属性不影响数据在内存中的布局。只是编号顺序改变。特别是，我们可以在“从后向前”循环中遍历数组，从而能够在 MQL5 中实现其类似功能。但 MQL5 提供了现成函数，可以让应用程序编程人员无需关注所有这些例行工作。

时间序列可以是在 MQL 程序中描述的任何一维动态数组，以及从 MetaTrader 5 核心传递至 MQL 程序的外部数组，诸如实用函数的参数。例如，一种特殊类型的 MQL 程序“[指标](#)”在

OnCalculate 事件处理程序中接收具有当前图表价格数据的数组。我们将在后面本书第五章了解时间序列的所有实际应用特性。

在 MQL 程序中定义的数组默认不是时间序列。

我们来研究一组函数，用于确定和更改数组的“系列”属性，以及确定数组是否“归属于”终端。包括示例在内的常规 **ArrayAsSeries.mq5** 脚本将在说明后提供。

bool ArrayIsSeries(const void &array[])

该函数返回一个符号，表明指定数组是否是“真实”时间序列，即它是由终端本身控制和提供的。数组的这一特性无法更改。MQL 程序能够以“只读”模式使用这种数组。

在 MQL5 文档中，术语“时间序列”和“系列”用于描述数组的逆索引以及该数组可能“属于”终端这一事实（终端为其分配内存和填充数据）。在本书中，我们将尝试消除这种歧义，将具有逆索引的数组称为“时间序列”。而终端数组就是指终端的 **own** 数组。

你可以通过在时间序列模式与标准模式之间来回切换，可酌情更改终端的任何自定义数组的索引方向。可使用 **ArraySetAsSeries** 函数来进行切换，该函数不仅适用于自己的数组，而且也适用于自定义动态数组（参见下文）。

bool ArrayGetAsSeries(const void &array[])

该函数返回一个符号，指示是否为指定数组启用了时间序列索引模式（即索引在从现在到过去的方向上增加）。可以使用 **ArraySetAsSeries** 函数更改索引方向。

索引方向会影响 **ArrayBsearch**、**ArrayMaximum** 和 **ArrayMinimum** 函数返回的值（参见章节 [数组的比较、排序和搜索](#)）。

bool ArraySetAsSeries(const void &array[], bool as_series)

该函数根据 **as_series** 参数设置数组的索引方向：**true** 值表示逆序索引，而 **false** 值表示正常元素顺序。

如果属性设置成功，该函数返回 **true**，如果出错，则返回 **false**。

支持任何类型的数组，但对于多维数组和固定大小数组，禁止更改索引方向。

ArrayAsSeries.mq5 脚本描述了若干小型数组，用于进行涉及上述函数的试验。

```

#define LIMIT 10

template<typename T>
void indexArray(T &array[])
{
    for(int i = 0; i < ArraySize(array); ++i)
    {
        array[i] = (T)(i + 1);
    }
}

class Dummy
{
    int data[];
};

void OnStart()
{
    double array2D[][2];
    double fixed[LIMIT];
    double dynamic[];
    MqlRates rates[];
    Dummy dummies[];

    ArrayResize(dynamic, LIMIT); // allocating memory
    // fill in a couple of arrays with numbers: 1, 2, 3,...
    indexArray(fixed);
    indexArray(dynamic);
    ...
}

```

我们有二维数组 *array2D*、固定和动态数组 (类型均为 *double*) 以及结构体数组和类对象数组。出于演示目的, *fixed* 和 *dynamic* 数组以连续整数填充 (使用辅助函数 *indexArray*)。对于其它类型的数组, 我们将只检查 “系列” 模式的适用性, 因为逆索引效应的概念已经通过填充数组示例解释清楚。

首先, 确保没有数组是终端自身的数组:

```

PRTS(ArrayIsSeries(array2D)); // false
PRTS(ArrayIsSeries(fixed));   // false
PRTS(ArrayIsSeries(dynamic)); // false
PRTS(ArrayIsSeries(rates));   // false

```

所有 *ArrayIsSeries* 调用均返回 *false*, 因为我们在 MQL 程序中定义了所有数组。我们会看到指标中 *OnCalculate* 函数的参数数组的值为 *true* (在第五章)。

接下来, 我们检查数组索引的初始方向:

```

PRTS(ArrayGetAsSeries(array2D)); // false, cannot be true
PRTS(ArrayGetAsSeries(fixed));   // false
PRTS(ArrayGetAsSeries(dynamic)); // false
PRTS(ArrayGetAsSeries(rates));   // false
PRTS(ArrayGetAsSeries(dummies)); // false

```

我们会再次全部得到 *false*。

我们将 *fixed* 和 *dynamic* 数组输出到日志，以查看元素的原始顺序。

```
ArrayPrint(fixed, 1);
ArrayPrint(dynamic, 1);
/*
    1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
    1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
*/
```

现在我们尝试更改索引顺序：

```
// error: parameter conversion not allowed
// PRTS(ArraySetAsSeries(array2D, true));

// warning: cannot be used for static allocated array
PRTS(ArraySetAsSeries(fixed, true)); // false

// after this everything is standard
PRTS(ArraySetAsSeries(dynamic, true)); // true
PRTS(ArraySetAsSeries(rates, true)); // true
PRTS(ArraySetAsSeries(dummies, true)); // true
```

array2D 数组的一个语句导致编译错误，因此被注释掉。

fixed 数组的一个语句发出编译器警告，提示其不能被应用到恒定大小的数组。在运行时，所有 3 个最后语句均返回成功 (*true*)。我们看看数组的属性如何改变：

```
// attribute checks:
// first, whether they are native to the terminal
PRTS(ArrayIsSeries(fixed)); // false
PRTS(ArrayIsSeries(dynamic)); // false
PRTS(ArrayIsSeries(rates)); // false
PRTS(ArrayIsSeries(dummies)); // false

// second, indexing direction
PRTS(ArrayGetAsSeries(fixed)); // false
PRTS(ArrayGetAsSeries(dynamic)); // true
PRTS(ArrayGetAsSeries(rates)); // true
PRTS(ArrayGetAsSeries(dummies)); // true
```

如预期的那样，数组没有转变为终端自身的数组。然而，四个数组中有三个将它们的索引方式更改为时间序列模式，包括一个结构体数组和对象数组。为演示结果，*fixed* 和 *dynamic* 数组再次显示在日志中。

```
ArrayPrint(fixed, 1); // without changes
ArrayPrint(dynamic, 1); // reverse order
/*
    1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
    10.0 9.0  8.0  7.0  6.0  5.0  4.0  3.0  2.0  1.0
*/
```

由于该模式不适用于恒定大小的数组，因此其保持不变。*dynamic* 数组现在以逆序显示。

如果你将数组置于逆索引模式，调整其大小，然后恢复为之前的索引方式，则添加的元素将插入在数组的开头。

4.3.9 将对象和数组清零

通常，初始化或填充变量和数组并不会导致问题。因此，对于简单变量，我们可以配合 [初始化](#) 在定义语句中直接使用运算符 '='，或者在之后赋予所需的值。

聚合视图初始化适用于结构体（参见章节 [定义结构体](#)）：

```
Struct struct = {value1, value2, ...};
```

但是，仅当结构体中没有动态数组和字符串时才适用。此外，聚合初始化句法不能再用于清除结构体。相反，要么必须单独为每个字段赋值，要么在程序中保留空结构体的实例，并将其复制到可清除的实例。

如果我们所说的数组同时还是结构体数组，则源代码会因其辅助但却必要的指令而快速增加。

对于数组，有 [ArrayInitialize](#) 和 [ArrayFill](#) 函数，但它们只支持数值类型：字符串或结构体数组不能使用这两个函数填充。

在此类情况下，便可以使用 [ZeroMemory](#) 函数。虽然这个函数不是万能的（在范围上有很大限制），但了解它也有好处。

[void ZeroMemory\(void &entity\)](#)

该函数可应用于广泛的不同实体：简单类型或对象类型变量及其数组（固定、动态或多维数组）。

变量获取 0 值（对于数字）或其等效值（对于字符串和指针，为 NULL）。

对于数组，所有其元素均设置为零。别忘了，元素可以是对象，而对象又能包含对象。换言之，[ZeroMemory](#) 函数在单次调用中执行深度内存清理。

然而，对有效对象存在限制。只有满足以下条件的结构和类对象才能用零填充：

- 仅包含公共字段（即不包含访问类型为 *private* 或 *protected* 的数据）
- 不包含具有 *const* 修饰符的字段
- 不包含指针

前两条限制由编译器强制检查：若对象的字段不满足指定要求，则尝试对这些对象执行归零将会导致错误（见下文）。

第三个限制是一项建议：对指针进行外部清零将导致难以检查数据完整性，从而可能导致关联对象丢失和内存泄漏。

严格来说，对可空化对象中字段的公开性要求违反了 [封装](#) 原理，该原理为类对象中所固有，因此，[ZeroMemory](#) 主要用于简单结构体对象及其数组。

[ZeroMemory](#) 使用示例在 [ZeroMemory.mq5](#) 脚本中提供。

通过 [Simple](#) 结构体演示了聚合初始化列表的问题：

```

#define LIMIT 5

struct Simple
{
    MqlDateTime data[]; // dynamic array disables initialization list,
    // string s; // and a string field would also forbid,
    // ClassType *ptr; // and a pointer too
    Simple()
    {
        // allocating memory, it will contain arbitrary data
        ArrayResize(data, LIMIT);
    }
};

```

在 **OnStart** 函数中或在全局上下文中，我们不能定义并立即空化一个该结构体的对象：

```

void OnStart()
{
    Simple simple = {}; // error: cannot be initialized with initializer list
    ...
}

```

编译器会抛出错误“无法使用初始化列表”。它特定于诸如动态数组、字符串变量和指针等字段。特别是，如果 **data** 数组是固定大小，则不会发生错误。

因此，我们使用 **ZeroMemory** 代替初始化列表：

```

void OnStart()
{
    Simple simple;
    ZeroMemory(simple);
    ...
}

```

以零进行的初始填充也可在结构体构造函数中完成，但在外部执行后续清理会更方便（或者提供一个与 **ZeroMemory** 函数相同的方法）。

以下类在 **Base** 中定义。

```

class Base
{
public: // public is required for ZeroMemory
    // const for any field will cause a compilation error when calling ZeroMemory:
    // "not allowed for objects with protected members or inheritance"
    /* const */ int x;
    Simple t; // using a nested structure: it will also be nulled
    Base()
    {
        x = rand();
    }
    virtual void print() const
    {
        PrintFormat("%d %d", &this, x);
        ArrayPrint(t.data);
    }
};

```

由于该类后续会用于可通过 *ZeroMemory* 空化的对象数组中，因此我们不得不为其字段编写一个访问段 *public*（原则上对于类来说并不常见，这样做是为了说明 *ZeroMemory* 施加的要求）。此外须注意，这些字段不能有 *const* 修饰符。否则将会出现编译错误，以及与问题不太相称的文本提示：“对于具有受保护成员或继承的对象是禁止的”。

类构造函数以随机数字填充 *x* 字段，所以稍后就能明确看到由函数 *ZeroMemory* 执行的清理。*print* 方法显示要分析的所有字段的内容，包含唯一对象编号（描述符）*&this*。

SQL5 不会阻止将 *ZeroMemory* 应用于指针变量：

```

Base *base = new Base();
ZeroMemory(base); // will set the pointer to NULL but leave the object

```

然而不应该这样做，因为该函数仅重置 *base* 变量本身，如果其原本指的是一个对象，则该对象将在内存中保持“挂起”状态，由于指针丢失而无法从程序访问。

仅当使用 *delete* 运算符释放了指针实例之后，才能空化该指针。此外，对于上面示例中的单独指针，使用赋值运算符来重置会更简单，这一点与任何其它简单变量（非复合变量）没有区别。对于复合对象和数组，最好使用 *ZeroMemory*。

该函数允许你处理类层次结构的对象。例如，我们可以描述源自 *Base* 的 *Dummy* 类的派生类：


```

class Dummy : public Base
{
public:
    double data[]; // could also be multidimensional: ZeroMemory will work
    string s;
    Base *pointer; // public pointer (dangerous)

public:
    Dummy()
    {
        ArrayResize(data, LIMIT);

        // due to subsequent application of ZeroMemory to the object
        // we'll lose the 'pointer'
        // and get warnings when the script ends
        // about undeleted objects of type Base
        pointer = new Base();
    }

    ~Dummy()
    {
        // due to the use of ZeroMemory, this pointer will be lost
        // and will not be freed
        if(CheckPointer(pointer) != POINTER_INVALID) delete pointer;
    }

    virtual void print() const override
    {
        Base::print();
        ArrayPrint(data);
        Print(pointer);
        if(CheckPointer(pointer) != POINTER_INVALID) pointer.print();
    }
};

```

它包含带有 *double* 类型的动态数组、字符串以及 *Base* 类型指针的字段（这与该类所源自的类型相同，但在这里仅用于演示指针问题，以免描述另一个虚拟类）。当 *ZeroMemory* 函数空化 *Dummy* 对象时，*pointer* 处的对象丢失，无法在析构函数中释放。因此，在脚本终止后，这会导致其余对象中内存泄露的警告。

ZeroMemory 用于 *OnStart* 中以清除 *Dummy* 对象数组：

```

void OnStart()
{
    ...
    Print("Initial state");
    Dummy array[];
    ArrayResize(array, LIMIT);
    for(int i = 0; i < LIMIT; ++i)
    {
        array[i].print();
    }
    ZeroMemory(array);
    Print("ZeroMemory done");
    for(int i = 0; i < LIMIT; ++i)
    {
        array[i].print();
    }
}

```

日志将输出类似以下内容（初始状态将不同，因为它打印“dirty”的内容，新分配的内存；这里是代码的一小部分）：

```

Initial state
1048576 31539
    [year]    [mon]    [day] [hour] [min] [sec] [day_of_week] [day_of_year]
[0]         0      65665      32     0     0     0             0             0
[1]         0         0         0     0     0     0             65624            8
[2]         0         0         0     0     0     0             0             0
[3]         0         0         0     0     0     0             0             0
[4] 5242880 531430129 51557552     0     0 65665             32             0
0.0 0.0 0.0 0.0 0.0
...
ZeroMemory done
1048576 0
    [year] [mon] [day] [hour] [min] [sec] [day_of_week] [day_of_year]
[0]       0    0     0     0     0     0             0             0
[1]       0    0     0     0     0     0             0             0
[2]       0    0     0     0     0     0             0             0
[3]       0    0     0     0     0     0             0             0
[4]       0    0     0     0     0     0             0             0
0.0 0.0 0.0 0.0 0.0
...
5 undeleted objects left
5 objects of type Base left
3200 bytes of leaked memory

```

可使用描述符来比较清理之前和之后的对象状态。

因此，对 **ZeroMemory** 的单个调用能够重置任意分支数据结构（数组、结构体、带嵌套结构体字段和数组的结构体数组）的状态。

最后，我们看看 **ZeroMemory** 如何能够解决字符串数组初始化的问题。**ArrayInitialize** 和 **ArrayFill** 函数不能处理字符串。

```

string text[LIMIT] = {};
// an algorithm populates and uses 'text'
// ...
// then you need to re-use the array
// calling functions gives errors:
// ArrayInitialize(text, NULL);
//      -> no one of the overloads can be applied to the function call
// ArrayFill(text, 0, 10, NULL);
//      -> 'string' type cannot be used in ArrayFill function
ZeroMemory(text);           // ok

```

在注释的指令中，编译器会产生错误，声明这些函数中不支持 *string* 类型。

这个问题可以使用 *ZeroMemory* 函数解决。

4.4 数学函数

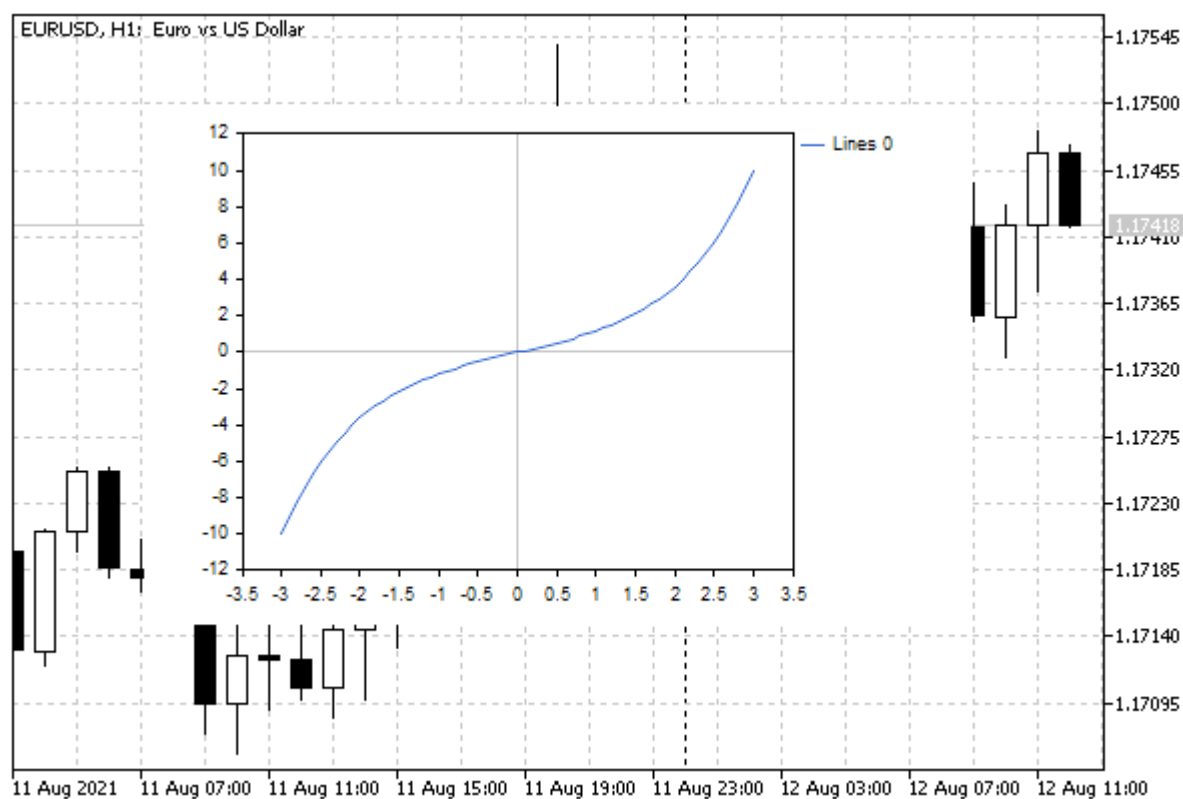
所有现代编程语言都支持常用的算术函数，MQL5 也不例外。在本章中，我们将了解若干组成可用的函数。这些函数包括舍入函数、三角函数、指数函数、对数函数、幂函数以及一些特殊函数，诸如生成随机数字以及检查实数正常性的函数。

大多数函数有两个名称：全名（带前缀“Math”并大写）和缩写（没有前缀，小写）。两种形式我们都会提供：它们的功能相同。可基于源代码的格式设置样式进行选择。

由于数学函数执行某些计算并返回实数结果，潜在错误可能导致结果未定义的情况。例如，不能取负数的平方根或取零的对数。在此类情况，函数会返回表示非数字特殊值（NaN，非数字）。我们已在以下章节接触过：[实数](#)、[算术运算](#)以及[数字与字符串的相互转换](#)。数字是否正确以及是否存在错误可使用 *MathIsValidNumber* 和 *MathClassify* 函数分析（参见[检查实数正常性](#)）。

只要有一个操作数的值为 NaN，就会导致隐含该操作数的任何后续计算（包括函数调用）都产生 NaN。

如需自学和获取直观学习材料，你可将 *MathPlot.mq5* 脚本用作附件，该脚本可根据上述函数的单一自变量来显示数学函数图形。该脚本使用 MetaTrader 5 中提供的标准图库 *Graphic.mqh*（不在本书范围）。下面是双曲正弦曲线在 MetaTrader 5 窗口中可能的显示样例。



MetaTrader5 窗口中的双曲正弦图

4.4.1 数字的绝对值

MQL5 API 提供了 **MathAbs** 函数，该函数能够移除数字的减号（如有）。因此，无需手动编写如下冗长的代码：

```
if(x < 0) x = -x;
```

numeric MathAbs(numeric value) ≡ numeric fabs(numeric value)

该函数返回传递给它的数字的绝对值，即其模数。自变量可以是任何类型的数字。换言之，该函数为 **char/uchar**、**short/ushort**、**int/uint**、**long/ulong**、**float** 和 **double** 重载，尽管对于无符号类型，值始终为非负。

在传递字符串时，字符串将被隐式转换为 **double** 数字，编译器将生成相关警告。

返回值的类型始终与自变量的类型相同，因此，如果类型不同，编译器可能需要将值强制转换为接收变量类型。

函数用法示例在 **MathAbs.mq5** 文件中提供。

```

void OnStart()
{
    double x = 123.45;
    double y = -123.45;
    int i = -1;

    PRT(MathAbs(x)); // 123.45, number left "as is"
    PRT(MathAbs(y)); // 123.45, minus sign gone
    PRT(MathAbs(i)); // 1, int is handled naturally

    int k = MathAbs(i); // no warning: type int for parameter and result

    // situations with warnings:
    // double to long conversion required
    long j = MathAbs(x); // possible loss of data due to type conversion

    // need to be converted from large type (4 bytes) to small type (2 bytes)
    short c = MathAbs(i); // possible loss of data due to type conversion
    ...
}

```

务必要注意，将有符号整数转换为无符号整数不等同于取数字的模数：

```

uint u_cast = i;
uint u_abs = MathAbs(i);
PRT(u_cast);           // 4294967295, 0xFFFFFFFF
PRT(u_abs);            // 1

```

同样需要注意的是，数字 0 可以有符号：

```

...
double n = 0;
double z = i * n;
PRT(z);           // -0.0
PRT(MathAbs(z));  // 0.0
PRT(z == MathAbs(z)); // true
}

```

MathAbs 的最佳用法示例之一是测试两个实数是否相等。众所周知，实数的值表示精度有限，在冗长计算过程中，精度可能进一步下降（例如，十个 0.1 的和并不准确得到 1.0）。在大多数情况下，严格条件 `value1 == value2` 可给出 *false*，而理论上它们应该相等。

因此，要比较实数值，通常使用以下记数法：

```
MathAbs(value1 - value2) < EPS
```

其中 EPS 是一个小的正数值，其指示精度（参见 [比较运算](#) 章节中的示例）。

4.4.2 比较两个数字的大小

为比较两个数字的大小，MQL5 提供了函数 **MathMax** 和 **MathMin**。它们的短别名分别是 *fmax* 和 *fmin*。

```
numeric MathMax(numeric value1, numeric value2) ≡ numeric fmax(numeric value1, numeric value2)
numeric MathMin(numeric value1, numeric value2) ≡ numeric fmin(numeric value1, numeric value2)
```

该等函数返回传递的两个值的大值和小值。该等函数为所有内置类型重载。

如果为函数传递了不同类型的参数，则“较低”类型的参数自动转换为“较高”类型，例如，在一对类型 *int* 和 *double* 中，*int* 将被转换为 *double*。有关显式类型强制转换的更多信息，参见 [算术类型转换](#) 章节。返回类型对应于“最高”类型。

有 *string* 类型的参数时，该参数将是“高级”类型，即全部缩减为字符串。字符串将从词典学角度比较，如同在 *StringCompare* 函数中一样。

MathMaxMin.mq5 脚本演示了这些函数的实际效果。

```
void OnStart()
{
    int i = 10, j = 11;
    double x = 5.5, y = -5.5;
    string s = "abc";

    // numbers
    PRT(MathMax(i, j)); // 11
    PRT(MathMax(i, x)); // 10
    PRT(MathMax(x, y)); // 5.5
    PRT(MathMax(i, s)); // abc

    // type conversions
    PRT(typeName(MathMax(i, j))); // int, as is
    PRT(typeName(MathMax(i, x))); // double
    PRT(typeName(MathMax(i, s))); // string
}
```

4.4.3 舍入函数

SQL5 API 包括若干函数可用于将数字（在某一个方向）舍入到最近整数。尽管有舍入运算，但所有函数都返回 *double* 类型的数字（小数部分为空）。

从技术角度而言，它们接受任何数值类型的自变量，但仅实数被舍入，整数只是被转换为 *double*。

如果想要舍入至特定符号，可使用 *NormalizeDouble*（参见 [双精度浮点数规范化](#) 章节）。

函数使用示例在 *MathRound.mq5* 文件中提供。

```
double MathRound(numeric value) ≡ double round(numeric value)
```

该函数将数字向上或向下舍入为最接近的整数。

```
PRT((MathRound(5.5))); // 6.0
PRT((MathRound(-5.5))); // -6.0
PRT((MathRound(11))); // 11.0
PRT((MathRound(-11))); // -11.0
```

如果小数部分的值大于或等于 0.5，则尾数加一（无论数字的符号为何）。

`double MathCeil(numeric value) ≡ double ceil(numeric value)`

`double MathFloor(numeric value) ≡ double floor(numeric value)`

该等函数返回传入 *value* 的最接近较大整数值 (对于 *ceil*) 或最接近较低整数值 (对于 *floor*)。如果 *value* 已经等于一个整数 (小数部分为零), 则返回该整数。

```
PRT((MathCeil(5.5))); // 6.0
PRT((MathCeil(-5.5))); // -5.0
PRT((MathFloor(5.5))); // 5.0
PRT((MathFloor(-5.5))); // -6.0
PRT((MathCeil(11))); // 11.0
PRT((MathCeil(-11))); // -11.0
PRT((MathFloor(11))); // 11.0
PRT((MathFloor(-11))); // -11.0
```

4.4.4 除法余数 (模运算)

为了进行有余数的整数除法, MQL5 提供了内置模数运算符 `%`, 该运算符在 [算术运算](#) 章节中介绍。然而, 该运算符不适用于实数。如果操作数除数和/或被除数均为实数, 则应使用函数 *MathMod* (或者简短形式为 *fmod*)。

`double MathMod(double dividend, double divider) ≡ double fmod(double dividend, double divider)`

该函数返回传递的第一个数字 (*dividend*) 除以传递的第二个数字 (*divider*) 的实数余数。

如果任何自变量为负, 则结果符号由上面 [章节](#) 中介绍的规则确定。

该函数的工作示例在 *MathMod.mq5* 脚本中提供。

```
PRT(MathMod(10.0, 3)); // 1.0
PRT(MathMod(10.0, 3.5)); // 3.0
PRT(MathMod(10.0, 3.49)); // 3.02
PRT(MathMod(10.0, M_PI)); // 0.5752220392306207
PRT(MathMod(10.0, -1.5)); // 1.0, the sign is gone
PRT(MathMod(-10.0, -1.5)); // -1.0
```

4.4.5 幂运算和根运算

MQL5 API 提供了用于将数字提升到任意次幂的通用函数 *MathPow*, 以及一个针对幂为 0.5 的特殊情况的函数, 也就是我们更为熟悉的求平方根运算 *MathSqrt*。

要测试这些函数, 可使用 *MathPowSqrt.mq5* 脚本。

`double MathPow(double base, double exponent) ≡ double pow(double base, double exponent)`

该函数将 *base* 提升到指定次幂 *exponent*。

```
PRT(MathPow(2.0, 1.5)); // 2.82842712474619
PRT(MathPow(2.0, -1.5)); // 0.3535533905932738
PRT(MathPow(2.0, 0.5)); // 1.414213562373095
```

`double MathSqrt(double value) ≡ double sqrt(double value)`

该函数返回一个数字的平方根。

```
PRT(MathSqrt(2.0));      // 1.414213562373095
PRT(MathSqrt(-2.0));     // -nan(ind)
```

SQL5 定义了若干个常量，包含涉及 *sqrt* 的现成计算值。

常量	说明	值
M_SQRT2	$\text{sqrt}(2.0)$	1.41421356237309504880
M_SQRT1_2	$1 / \text{sqrt}(2.0)$	0.707106781186547524401
M_2_SQRTPI	$2.0 / \text{sqrt}(M\_PI)$	1.12837916709551257390

其中 M_PI 是 Pi 数字 ($\pi=3.14159265358979323846$ ，更多详细信息参见 [三角函数](#) 章节)。

所有内置常量均在 [文档](#) 中介绍。

4.4.6 指数函数和对数函数

在 SQL5 中，可通过相应的 API 部分实现指数函数和对数函数的计算。

API 中缺少二进制对数，而计算机科学和组合数学中通常需要二进制对数，但这并不是问题，因为在需要时，可以通过可用的自然对数或十进制对数函数轻松计算。

```
log2(x) = log(x) / log(2) = log(x) / M_LN2
log2(x) = log10(x) / log10(2)
```

其中 *log* 和 *log10* 是可用的对数函数 (分别以 *e* 和 10 为底)，M_LN2 是一个等于 *log(2)* 的内置常量。

下表列出了可用于对数计算的所有常量。

常量	说明	值
M_E	<i>e</i>	2.71828182845904523536
M_LOG2E	$\log_2(e)$	1.44269504088896340736
M_LOG10E	$\log_{10}(e)$	0.434294481903251827651
M_LN2	$\ln(2)$	0.693147180559945309417
M_LN10	$\ln(10)$	2.30258509299404568402

下面描述的函数的示例收集在 *MathExp.mq5* 文件中。

`double MathExp(double value) ≡ double exp(double value)`

该函数返回提升到指定次幂 *value* 的指数，即数字 *e* (作为预定义常量 M_E 可用)。溢出时，函数返回 *inf* (一种表示无穷大的 NaN)。


```
PRT(MathExp(0.5));      // 1.648721270700128
PRT(MathPow(M_E, 0.5)); // 1.648721270700128
PRT(MathExp(10000.0));  // inf, NaN
```

double MathLog(double value) ≡ double log(double value)

该函数返回传递数字的自然对数。如果 *value* 为负，则函数返回 *-nan(ind)* (NaN “未定义值”)。如果 *value* 为 0，则函数返回 *inf* (NaN “无穷大”)。

```
PRT(MathLog(M_E));      // 1.0
PRT(MathLog(10000.0));  // 9.210340371976184
PRT(MathLog(0.5));      // -0.6931471805599453
PRT(MathLog(0.0));      // -inf, NaN
PRT(MathLog(-0.5));     // -nan(ind)
PRT(Log2(128));         // 7
```

最后一行通过 *MathLog* 使用二进制对数的实现：

```
double Log2(double value)
{
    return MathLog(value) / M_LN2;
}
```

double MathLog10(double value) ≡ double log10(double value)

该函数返回数字的十进制对数。

```
PRT(MathLog10(10.0));  // 1.0
PRT(MathLog10(10000.0)); // 4.0
```

double MathExp1(double value) ≡ double expm1(double value)

该函数返回表达式 $(\text{MathExp}(\text{value}) - 1)$ 的值。在经济计算中，该函数用于计算当期间数趋向于无穷时每单位时间的有效复利利息（收入或付款）。

```
PRT(MathExp1(0.1)); // 0.1051709180756476
```

double MathLog1p(double value) ≡ double log1p(double value)

该函数返回表达式 $\text{MathLog}(1 + \text{value})$ 的值，即，其执行与 *MathExp1* 函数相反的操作。

```
PRT(MathLog1p(0.1)); // 0.09531017980432487
```

4.4.7 三角函数

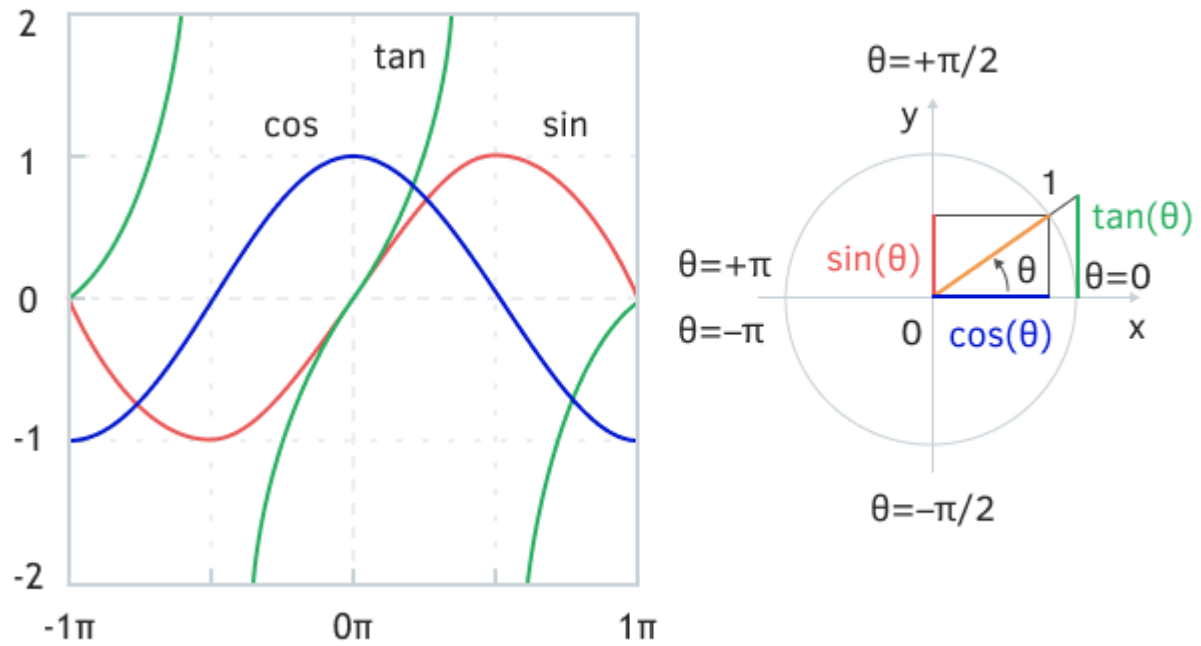
SQL5 提供了三个主要的三角函数 (*MathCos*, *MathSin*, *MathTan*) 以及它们的逆函数 (*MathArccos*, *MathArcsin*, *MathArctan*)。它们均处理以弧度为单位的角。对于以度数为单位的角，使用以下公式：

`radians = degrees * M_PI / 180`

其中 `M_PI` 是语言内置的若干常量之一，具有三角函数数量 `pi` 及其衍生值。

常量	说明	值
<code>M_PI</code>	π	3.14159265358979323846
<code>M_PI_2</code>	$\pi/2$	1.57079632679489661923
<code>M_PI_4</code>	$\pi/4$	0.785398163397448309616
<code>M_1_PI</code>	$1/\pi$	0.318309886183790671538
<code>M_2_PI</code>	$2/\pi$	0.636619772367581343076

也可以为由两个坐标 `y` 和 `x` 的比率表示的数量计算反正切值：该扩展版本称为 `MathArctan2`；它能够还原圆完整范围 `-M_PI` 到 `+M_PI` 内的角度，不同于 `MathArctan`，后者仅限于 `-M_PI_2` 到 `+M_PI_2`。



三角函数和单位圆的象限

计算示例在 `MathTrig.mq5` 脚本中提供（参见说明之后）。

`double MathCos(double value) ≡ double cos(double value)`

`double MathSin(double value) ≡ double sin(double value)`

这两个函数分别返回传递数字的余弦和正弦（角度以弧度为单位）。

`double MathTan(double value) ≡ double tan(double value)`

该函数返回传递数字的正切（角度以弧度为单位）。

`double MathArccos(double value) ≡ double acos(double value)`

`double MathArcsin(double value) ≡ double asin(double value)`

这两个函数分别返回传递数字的反余弦和反正弦，即弧度单位的角度。如果 $x = \text{MathCos}(t)$ ，则 $t = \text{MathArccos}(x)$ 。正弦和反正弦具有类似的方案。如果 $y = \text{MathSin}(t)$ ，则 $t = \text{MathArcsin}(y)$ 。

参数必须在 -1 到 +1 之间。否则，该函数将返回 NaN。

反余弦的结果在 0 到 M_PI 之间，而反正弦的结果在 $-M\_PI\_2$ 到 $+M\_PI\_2$ 之间。指示的范围称为主范围，因为函数是多值的，即它们的值定期重复。所选的半期间完全涵盖从 -1 到 +1 的定义区间。

得到的正弦的角度处于上半圆内，而下半圆中的对称解可通过添加一个符号获得，即 $t = -t$ 。对于正弦，获得的角度处于右半圆内，而左半圆内的第二解为 $M\_PI - t$ （若要求对负 t 也获取负的附加角，则解为 $-M\_PI - t$ ）。

`double MathArctan(double value) ≡ double atan(double value)`

该函数返回传递数字的反正切值，即以弧度为单位的角度，范围在 $-M\_PI\_2$ 到 $+M\_PI\_2$ 之间。

该函数是 *MathTan* 的反函数，但有一个注意事项。

请注意，由于正弦和余弦的比率因符号叠加而在相反象限（圆的四分之一）中重复，正切的期间是全期间（圆周）的二分之一。因此，仅依靠正切值无法唯一确定从 $-M\_PI$ 到 $+M\_PI$ 的全范围上的原始角度。可以使用 *MathArctan2* 函数实现，其中正切由两个单独的分量表示。

`double MathArctan2(double y, double x) ≡ double atan2(double y, double x)`

该函数返回弧度单位的角度值，其正切值等于两个指定数字的比率：沿 y 和沿 x 轴的坐标。

结果（将其表示为 r ）在 $-M\_PI$ 到 $+M\_PI$ 范围内，并且满足条件 $\text{MathTan}(r) = y / x$ 。

该函数考虑两个自变量的符号，以确定正确象限（遵循边界条件，当 x 或 y 等于 0，则它们处于象限的边界上）。

- 1 — $x \geq 0, y \geq 0, 0 \leq r \leq M\_PI\_2$
- 2 — $x < 0, y \geq 0, M\_PI\_2 < r \leq M\_PI$
- 3 — $x < 0, y < 0, -M\_PI < r < -M\_PI\_2$
- 4 — $x \geq 0, y < 0, -M\_PI\_2 \leq r < 0$

下面是在 *MathTrig.mq5* 脚本中调用三角函数的结果。

```

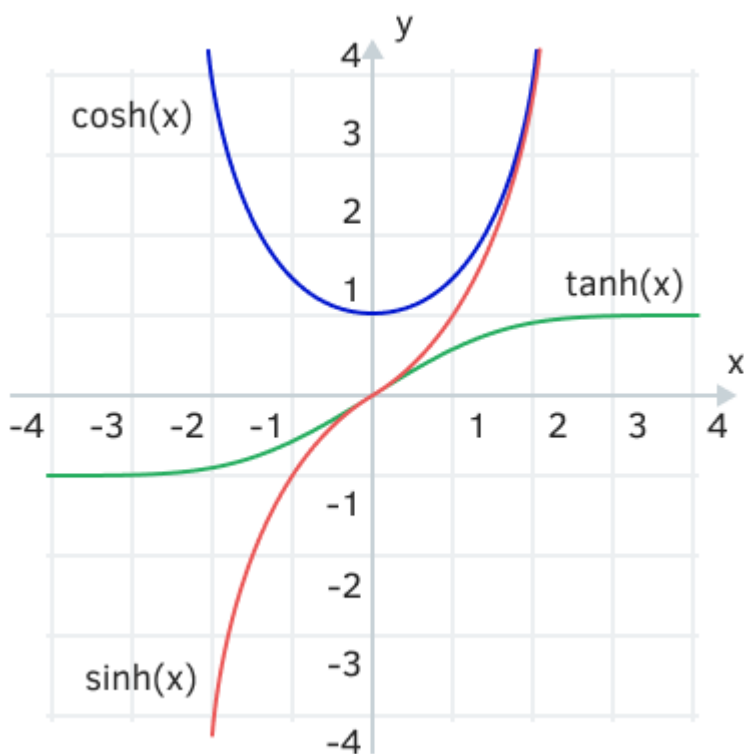
void OnStart()
{
    PRT(MathCos(1.0));      // 0.5403023058681397
    PRT(MathSin(1.0));      // 0.8414709848078965
    PRT(MathTan(1.0));      // 1.557407724654902
    PRT(MathTan(45 * M_PI / 180.0)); // 0.9999999999999999

    PRT(MathArccos(1.0));   // 0.0
    PRT(MathArcsin(1.0));   // 1.570796326794897 == M_PI_2
    PRT(MathArctan(0.5));   // 0.4636476090008061, Q1
    PRT(MathArctan2(1.0, 2.0)); // 0.4636476090008061, Q1
    PRT(MathArctan2(-1.0, -2.0)); // -2.677945044588987, Q3
}

```

4.4.8 双曲函数

SQL5 API 包括一系列双曲函数和反双曲函数。



双曲函数

`double MathCosh(double value) ≡ double cosh(double value)`

`double MathSinh(double value) ≡ double sinh(double value)`

`double MathTanh(double value) ≡ double tanh(double value)`

这三个基础函数计算双曲余弦、正弦和正切值。

```
double MathArccosh(double value) ≡ double acosh(double value)
double MathArcsinh(double value) ≡ double asinh(double value)
double MathArctanh(double value) ≡ double atanh(double value)
```

这三个反函数计算双曲反余弦、反正弦和反正切值。

对于反余弦值，自变量必须大于或等于 +1。否则，该函数将返回 NaN。

反正切的定义域是从 -1 到 +1。如果自变量超出这些限值，则函数将返回 NaN。

双曲函数示例显示在 *MathHyper.mq5* 脚本中。

```
void OnStart()
{
    PRT(MathCosh(1.0));    // 1.543080634815244
    PRT(MathSinh(1.0));    // 1.175201193643801
    PRT(MathTanh(1.0));    // 0.7615941559557649

    PRT(MathArccosh(0.5)); // nan
    PRT(MathArcsinh(0.5)); // 0.4812118250596035
    PRT(MathArctanh(0.5)); // 0.5493061443340549

    PRT(MathArccosh(1.5)); // 0.9624236501192069
    PRT(MathArcsinh(1.5)); // 1.194763217287109
    PRT(MathArctanh(1.5)); // nan
}
```

4.4.9 实数的正常性检验

由于实数计算可能出现异常情况，诸如超出函数范围、获得数字无穷大、丢失顺序等等，导致结果可能不包含数字。相反，结果可能包含特殊值，该特殊值实际上描述了问题的性质。所有这些特殊值具有一个通用名称：“非数字”（简称 NaN）。

我们在本书前面的章节中已接触了这种值。尤其是在输出到日志时（参见章节 [数字转换为字符串以及相反转换](#)），它们显示为文本标签（如 *nan(ind)*、*+inf* 等等）。另一个特征是，任何表达式的操作数中只要存在一个 NaN 值，整个表达式就会停止正确计算，开始给出 NaN 结果。唯一例外是表示正负无穷“非数字”：如果用其他数除以正负无穷，结果为零。然而有一个预期例外：如果我们将无穷除以无穷，我们又得到 NaN。

因此，重要的是让程序确定计算中出现 NaN 的时刻，并以特殊方式处理该情况：提示错误，替换某些可接受默认值，或者以其它参数重复计算（例如，减少迭代算法的精度/步骤）。

在 MQL5 中有 2 个函数可以分析实数的正常性：*MathIsValidNumber* 给出一个简单的回答：是 (*true*) 或否 (*false*)，而 *MathClassify* 生成更详细的类别划分。

在物理层面，所有特殊值都通过位的特殊组合编码在数字中，这些位组合不用于表示普通数字。对于 *double* 和 *float* 类型，这些编码当然不同。我们来深入了解一下 *double*（因为它比 *float* 更常用）。

在 [嵌套模板](#) 一章中，我们创建了 *Converter* 类，用于通过在一个联合体中组合两个不同的类型来切换视图。我们将该类用于研究 NaN 位设备。

出于方便，我们将把该类移动到一个单独的头文件 *ConverterT.mqh*。我们在测试脚本 *MathInvalid.mq5* 中连接该 *mqh* 文件，并为一系列类型 *double/ulong* 创建一个转换器实例（顺序不重要，因为该转换器能够双向工作）。

```
static Converter<ulong, double> NaNs;
```

NaN 中的位组合经过标准化，因此我们取几个由常量 *ulong* 表示的常用值，看看内置函数如何应对。

```
// basic NaNs
#define NAN_INF_PLUS 0x7FF0000000000000
#define NAN_INF_MINUS 0xFFF0000000000000
#define NAN_QUIET 0x7FF8000000000000
#define NAN_IND_MINUS 0xFFF8000000000000

// custom NaN examples
#define NAN_QUIET_1 0x7FF8000000000001
#define NAN_QUIET_2 0x7FF8000000000002

static double pinf = NaNs[NAN_INF_PLUS]; // +infinity
static double ninf = NaNs[NAN_INF_MINUS]; // -infinity
static double qnan = NaNs[NAN_QUIET]; // quiet NaN
static double nind = NaNs[NAN_IND_MINUS]; // -nan(ind)

void OnStart()
{
    PRT(MathIsValidNumber(pinf)); // false
    PRT(EnumToString(MathClassify(pinf))); // FP_INFINITE
    PRT(MathIsValidNumber(nind)); // false
    PRT(EnumToString(MathClassify(nind))); // FP_NAN
    ...
}
```

正如所预期的那样，结果是相同的。

我们看看 *MathIsValidNumber* 和 *MathClassify* 函数的正式说明，然后继续测试。

bool MathIsValidNumber(double value)

该函数检查实数的正确性。参数可以是 *double* 或 *float* 类型。结果 *true* 表示数字正确，*false* 表示“非数字”（某种 NaN）。

ENUM_FP_CLASS MathClassify(double value)

该函数返回实数的类别（*double* 或 *float* 类型），其是 *ENUM_FP_CLASS* 枚举值之一：

- *FP_NORMAL* 是一个正规数。
- *FP_SUBNORMAL* 是一个小于以正规化形式可表示的最小数字的数字（例如，对于 *double* 类型，这些是小于 *DBL_MIN*, 2.2250738585072014e-308 的值）；顺序（精度）丢失。
- *FP_ZERO* 为零（正或负）。
- *FP_INFINITE* 为无穷大（正或负）。
- *FP_NAN* 指所有其它类型的“非数字”细分为“静默型”和“信号型”NaN 系列）。

MQL5 不提供警报型 NaN，警报型 NaN 用于异常机制中，允许截获并回应程序内的关键错误。在 MQL5 中没有此类机制，因此，例如，如果出现除零情况，MQL 程序会直接终止其工作（从图表卸载）。

可能有很多“静默型”NaN，你可以使用转换器构造，以区分和处理计算算法中的非标准状态。

我们在 *MathInvalid.mq5* 中执行某些计算，以直观展示不同类别的数字的获得方式。

```
// calculations with double
PRT(MathIsValidNumber(0));           // true
PRT(EnumToString(MathClassify(0)));  // FP_ZERO
PRT(MathIsValidNumber(M_PI));        // true
PRT(EnumToString(MathClassify(M_PI))); // FP_NORMAL
PRT(DBL_MIN / 10);                  // 2.225073858507203e-309
PRT(MathIsValidNumber(DBL_MIN / 10)); // true
PRT(EnumToString(MathClassify(DBL_MIN / 10))); // FP_SUBNORMAL
PRT(MathSqrt(-1.0));                 // -nan(ind)
PRT(MathIsValidNumber(MathSqrt(-1.0))); // false
PRT(EnumToString(MathClassify(MathSqrt(-1.0)))); // FP_NAN
PRT(MathLog(0));                     // -inf
PRT(MathIsValidNumber(MathLog(0)));  // false
PRT(EnumToString(MathClassify(MathLog(0)))); // FP_INFINITE

// calculations with float
PRT(1.0f / FLT_MIN / FLT_MIN);      // inf
PRT(MathIsValidNumber(1.0f / FLT_MIN / FLT_MIN)); // false
PRT(EnumToString(MathClassify(1.0f / FLT_MIN / FLT_MIN))); // FP_INFINITE
```

我们可以在相反方向使用转换器：通过 *double* 值获取其位表示，进而探测“非数字”：

```
PrintFormat("%I64X", NaNs[MathSqrt(-1.0)]); // FFF8000000000000
PRT(NaN[MathSqrt(-1.0)] == NAN_IND_MINUS); // true, nind
```

PrintFormat 函数类似于 *StringFormat*；唯一的不同是结果立即打印到日志，而不是打印到字符串。

最后，我们确保“非数字”始终不相等：

```
// NaN != NaN always true
PRT(MathSqrt(-1.0) != MathSqrt(-1.0)); // true
PRT(MathSqrt(-1.0) == MathSqrt(-1.0)); // false
```

要在 MQL5 中获取 NaN 或无穷大，有一种基于将字符串“nan”和“inf”转化为 *double* 的方法。

```
double nan = (double)"nan";
double infinity = (double)"inf";
```

4.4.10 随机数生成

交易中的很多算法要求生成随机数。MQL5 提供了两个函数可用于初始化然后轮询伪随机整数生成器。

为获得更好的“随机性”，可以使用 `MetaTrader 5` 中提供的 `Alglib` 库（参见 *MQL5/Include/Math/Alglib/alglib.mqh*）。

void MathSrand(int seed) ≡ void srand(int seed)

该函数设置伪随机整数生成器的某个初始状态。在开始算法之前，应调用该函数一次。随机值本身应通过顺序调用 *MathRand* 函数来获取。

通过以相同的 *seed* 值来初始化生成器，可以得到可重现的数字序列。*seed* 值并非是从 *MathRand* 获得的第一个随机数。生成器保持某个内部状态，该状态在每个时刻（调用以生成新随机数的间隔期间）以一个整数值为特征，该整数值可从程序作为内置 *uint* 变量 *_RandomSeed* 获得。正是这一初始状态值建立了 *MathSrand* 调用。

每次调用 *MathRand* 时的生成器操作由以下两个公式描述：

$$X_n = Tf(X_p)$$

$$R = Gf(X_n)$$

Tf 函数称为转换。它基于之前的 *Xp* 状态计算 *Xn* 生成器的新内部状态。

Gf 函数使用新的内部状态生成 *MathRand* 函数将返回的另一个“随机”值。

在 MQL5 中，这些公式按以下方式实现（伪代码）：

```
Tf: _RandomSeed = _RandomSeed * 214013 + 2531011
Gf: MathRand = (_RandomSeed >> 16) & 0x7FFF
```

建议传递 *GetTickCount* 或 *TimeLocal* 函数作为 *seed* 值。

int MathRand() ≡ int rand()

该函数返回一个范围从 0 到 32767 的伪随机整数。生成的数字序列各不相同，具体取决于调用 *MathSrand* 完成的开端初始化。

该生成器的用法示例在 *MathRand.mq5* 文件中提供。它计算关于生成的数字在给定数量的子范围（组）上的分布统计。理想情况下，应获得均匀分布。

```
#define LIMIT 1000 // number of attempts (generated numbers)
#define STATS 10   // number of baskets

int stats[STATS] = {}; // calculation of statistics of hits in baskets

void OnStart()
{
    const int bucket = 32767 / STATS;
    // generator reset
    MathSrand((int)TimeLocal());
    // repeat the experiment in a loop
    for(int i = 0; i < LIMIT; ++i)
    {
        // getting a new random number and updating statistics
        stats[MathRand() / bucket]++;
    }
    ArrayPrint(stats);
}
```

三次运行的结果示例（每次将获得一个新序列）：


```

96  93 117  76  98  88 104 124 113  91
110 81 106  88 103  90 105 102 106 109
89  98  98 107 114  90 101 106  93 104

```

4.4.11 整数中的字节顺序控制

硬件层面的各种信息系统在内存中表示数字时，使用不同的字节顺序。因此，当将 MQL 程序与“外部世界”整合时，尤其是当实现网络通信协议或读写常见格式的文件时，可能需要更改字节顺序。

Windows 计算机应用小端模式（从最低有效字节开始），即在为变量分配的内存单元中，首先从最低字节开始，然后是具有更高位的字节，以此类推。互联网上则广泛使用另一种大端模式（从最高位，即最有效字节开始）。在这种情况下，内存单元中的第一个字节是高位字节，而最后一个字节是低位字节。这种顺序类似于我们在日常生活中“从左到右”写数字的方式。例如，值 1234 从 1 开始，代表千位，后面是 2 代表百位，3 代表十位，最后的 4 就是个位四（低序）。

我们看看 MQL5 中的默认字节序。为此，我们将使用 *MathSwap.mq5* 脚本。

该脚本描述了允许你将整数转换为字节数组的连接模式：

```

template<typename T>
union ByteOverlay
{
    T value;
    uchar bytes[sizeof(T)];
    ByteOverlay(const T v) : value(v) { }
    void operator=(const T v) { value = v; }
};

```

该代码允许你以可视化的方式将数字分为字节，并使用数组中的索引枚举这些字节。

在 *OnStart* 中，我们以值 0x12345678 描述 *uint* 变量（注意，数字为十六进制；在该记数法中，它们与字节边界完全对应：每 2 个数位是一个单独的字节）。我们将数字转换为数组，并将其输出到日志。

```

void OnStart()
{
    const uint ui = 0x12345678;
    ByteOverlay<uint> bo(ui);
    ArrayPrint(bo.bytes); // 120  86  52  18 <==> 0x78 0x56 0x34 0x12
    ...
}

```

ArrayPrint 函数不能打印十六进制的数字，因此我们看到的是它们的十进制表示，但可轻松将它们转换为十六进制，以确保它们匹配原始字节。直观来看，它们处于逆反顺序：即，在数组中的 0 号索引是 0x78，然后是 0x56、0x34 和 0x12。很明显，这一顺序是从最低有效字节开始（实际上，我们处于 Windows 环境中）。

现在让我们来熟悉函数 *MathSwap*，MQL5 提供该函数用于更改字节顺序。

integer MathSwap(integer value)

该函数返回一个整数，其中传入参数的字节顺序被反转。该函数使用 *ushort/uint/ulong* 类型的参数（即大小为 2、4、8 字节）。

我们以实操的方式来测试该函数：

```
const uint ui = 0x12345678;
PrintFormat("%I32X -> %I32X", ui, MathSwap(ui));
const ulong ul = 0x0123456789ABCDEF;
PrintFormat("%I64X -> %I64X", ul, MathSwap(ul));
```

结果如下：

```
12345678 -> 78563412
123456789ABCDEF -> EFCDAB8967452301
```

我们尝试在使用 *MathSwap* 转换值 0x12345678 之后，将字节数组记录到日志：

```
bo = MathSwap(ui); // put the result of MathSwap into ByteOverlay
ArrayPrint(bo.bytes); // 18 52 86 120 <==> 0x12 0x34 0x56 0x78
```

字节中索引为 0 的位置，之前是 0x78，现在是 0x12，在包含其它数字的元素中，值也已经交换。

4.5 处理文件

几乎所有程序都要使用数据输入输出。我们已经知道，MQL 程序可通过 [输入变量](#) 来接收设置，并将信息输出到日志，因为我们在几乎所有测试脚本中均使用日志。但在大多数情况下，这还不够。

例如，程序自定义的很大一部分涉及的数据量无法通过输入参数容纳。程序可能需要与某些外部分析工具集成，例如，以标准或特殊格式上传市场信息，处理这些信息并以新的格式加载到终端，尤其是作为交易信号加载，交易信号是一系列神经网络权重或决策树系数。此外，为 MQL 程序保持一份单独的日志可能会为我们提供方便。

文件子系统为此类任务提供了最普遍的方案。MQL5 API 提供了用于处理文件的各种函数，包括用于创建、删除、搜索、编写和读取文件的函数。我们将在本章中研究所有这些函数。

在 MQL5 中，所有文件操作均限制在磁盘上的一个特殊区域，称为沙盒。这样做是出于安全原因，避免 MQL 程序被用于不良目的，损害你的计算机或操作系统。

高级用户可使用特殊措施免除这一限制，对此我们将稍后再讨论。但这应仅在特殊情况下执行，且要遵守预防措施并自行承担全部责任。

对于计算机上安装的终端的每个实例，沙盒根目录位于 *<terminal_data_folder>/MQL5/Files/*。从 MetaEditor，可以使用 *File -> Open Data Folder* 命令打开数据文件夹。如果你对计算机有充分的访问权限，则该目录通常处于与终端安装所在位置相同的目录。如果你没有要求的权限，则路径可能象是这样：

```
X:/Users/<user_name>/AppData/Roaming/MetaQuotes/Terminal/<instance_id>/MQL5/Files/
```

其中 *X* 是安装系数的盘符，*<user_name>* 是 Windows 用户登录名，*<instance_id>* 是终端实例的唯一标识符。*Users* 文件夹也具有别名 *"Documents and Settings"*。

请注意，如果是通过 RDP (远程桌面协议) 远程连接到计算机，即使你有管理员权利，终端也将始终使用 *Roaming* 文件夹及其子文件夹。

别忘了，数据目录中的 MQL5 文件夹是存储所有 MQL 程序的位置：包括它们的源代码和编译的 ex5 文件。每种类型的 MQL 程序，包括指标、EA 交易、脚本及其它程序，都在 MQL5 文件夹中有专门的子文件夹。因此，工作文件的 *Files* 文件夹与它们并列存在。

除了在计算机上的每个终端副本的这个单独沙盒外，还有一个所有终端的共享沙盒：其路径穿过 Windows 用户的主文件夹，可能因操作系统版本而不同。例如，在标准安装的 Windows 7、8 和 10 中，该路径象是这样：

```
X:/Users/<user_name>/AppData/Roaming/MetaQuotes/Terminal/Common/Files/
```

同样，该文件夹可轻松通过 MetaTrader 访问：运行命令 **File -> Open Shared Data Folder**，即可进入 Common 文件夹。

某些类型的 MQL 程序 (EA 交易和指标)不仅可以在终端中执行，而且可以在测试程序中执行。运行 MQL 程序时，共享沙盒保持可访问，并且使用测试代理中的一个文件夹代替单个实例沙盒。一般如下所示：

```
X:/<terminal_path>/Tester/Agent-IP-port/MQL5/Files/
```

这在 MQL 程序本身内可能不可见，即所有文件函数均以完全相同的方式运行。然而，从用户角度，可能会觉得有点问题。例如，如果程将其工作结果保存到一个文件，在运行完成后，该文件将从测试程序的代理文件夹中删除 (好像该文件从未被创建过一样)。这一例行方法旨在防止一个程序的潜在有价值数据被泄露进入可能稍后某个时间在该相同代理上进行测试的另一个程序中 (尤其是由于代理是可以共享的)。提供了其它技术用于将文件传输给代理并从代理将结果返回至终端，对此我们将在本书第五章讨论。

要规避沙盒限制，可以使用 Windows 向系统对象赋予符号链接的功能。就我们的情况而言，连接 (接合)最适合于在本地计算机上重定向文件夹访问。它们使用以下命令 (即 Windows 命令行)创建：

```
mklink /J new_name existing_target
```

new_name 参数表示将指向真实文件夹 **existing_target** 的新的虚拟文件夹的名称。

要创建到沙盒外的外部文件夹的连接，建议在 MQL5/Files 内创建一个专用文件夹，例如 **Links**。然后，在进入该文件夹之后，可以通过选择 **new_name** 并将沙盒外面的真实路径替换为 **existing_target** 来执行上述命令。例如，以下命令将会在文件夹 **Links** 中创建一个名为 **Settings** 的新链接，该链接将提供对 MQL5/Presets 文件夹的访问：

```
mklink /J Settings "..\..\Presets\"
```

相对路径 **"..\..\Presets\"** 假定该命令在指定的 MQL5/Files/Links 文件夹中执行。两个点的组合 **".."** 表示从当前文件夹转到父文件夹。如果指定两次，则该组合指示在路径层次中上移两级。因此，目标文件夹 (**existing_target**) 将生成为 MQL5/Presets。但在 **existing_target** 参数中，你也可以指定一个绝对路径。

你可以象常规文件一样删除符号链接 (当然了，你应首先确保删除的是左下角带箭头图标的文件夹 (即链接)，而非原始文件夹)。建议不再需要访问沙盒外资源时立即删除链接。事实是，创建的虚拟文件夹对所有 MQL 程序可用，而不只是对你的程序可用，无法预测其他人的程序会如何使用这一附加的自由权限。

本章很多章节均在讨论文件名。它们作为文件系统元素标识符，具有类似规则，包括某些限制。

请注意，文件名不能包含在文件系统中扮演特殊角色的某些字符 (<、>、/、\、"、'、:、|、*、?)，以及代码为 0 至 31 (含)的任何字符。

以下文件名也被保留用于操作系统中的特殊用途，不能使用：CON、PRN、AUX、NUL、COM1、COM2、COM3、COM4、COM5、COM6、COM7、COM8、COM9、LPT1、LPT2、LPT3、LPT4、LPT5、LPT6、LPT7、LPT8、LPT9。

需要注意的是，Windows 文件系统并不区分不同大小写字母之间的基本差异，因此诸如“Name”、“NAME”以及“name”等名称均指同一元素。

Windows 允许将反斜杠 ‘\’ 和正斜杠 ‘/’ 用作路径要素（子文件夹和文件）之间的分隔符字符。然而，在 MQL5 字符串中，反斜杠需要进行转义（实际上写两次），因为 ‘\’ 字符本身特殊：它用于构造控制字符系列，诸如 ‘\r’、‘\n’、‘\t’ 等等（参见章节 [字符类型](#)）。例如，以下路径等效：“MQL5Book/file.txt”和“MQL5Book\\file.txt”。

点字符 ‘.’ 作为名称与扩展名之间的分隔符。如果一个文件系统元素在其标识符中有多个点，则最右侧点的右侧部分是扩展名，而其左侧的所有部分均为名称。标题（点之前）或扩展名（点之后）可以为空。例如，没有扩展名的文件名是“text”，而没有名称（仅有扩展名）的文件是“.txt”。

Windows 中路径和文件名的总长度有限制。同时，为管理 MQL5 中的文件，应考虑沙盒路径将被添加到其路径和名称，即将为 MQL 函数调用中文件对象的名称分配更少的空间。默认情况下，总体长度限值是系统常量 MAX_PATH，等于 260。从 Windows 10 (build 1607) 开始，可将该限值增加到 32767。为此，你需要将以下文本保存为一个 .reg 文件，并通过将其添加到 Windows 注册表来运行该文件。

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem]
"LongPathsEnabled"=dword:00000001
```

对于其它版本的 Windows，你可以使用命令行替代方法。尤其是你可以使用上面讨论过的连接（通过创建具有短路径的虚拟文件夹）来缩短路径。你还可以使用 shell 命令 **-subst**，如 **subst z: c:\very\long\path**（详见 Windows 帮助）。

4.5.1 信息存储方法：文本和二进制

在前面多个章节中，我们已经知道，相同的信息可以以文本和二进制形式表示。例如，*int*、*long* 数字和 *double* 格式、日期和时间（*datetime*）以及颜色（*color*）均以一定长度的字节序列存储在内存中。该方法比较紧凑，更适合计算机解读，但对于人类来说，以文本形式分析信息更为方便，虽然其长度更长。因此，我们更多地关注于 [数字与字符串的相互转换](#)以及 [用于处理字符串的函数](#)。

在文件层面，仍然保留了数据的二进制和文本表示的划分。二进制文件设计用于以内存中使用的相同内部表示来存储数据。文本文件包含字符串表示。

文本文件通常用于标准格式，诸如 CSV（逗号分隔值）、JSON（JavaScript 对象表示法）、XML（可扩展标记语言）、HTML（超文本标记语言）。

二进制文件当然也具有用于多种应用程序的标准格式，尤其是图像（PNG、GIF、JPG 和 BMP）、声音（WAV、MP3）或压缩档案（ZIP）。然而，二进制格式在一开始需要更大的数据保护和低层级的数据处理，因此更常用于解决内部问题——此时仅需关注存储效率和特定程序的数据可用性。换言之，任何应用结构体和类对象都可轻松在二进制文件中保存和恢复状态（实际生成内存快照），无需担心与任何标准的兼容性。

理论上，当写入二进制文件时，我们可以手动将数据转换为字符串，然后在读取文件时从字符串转换回数字（或结构体或数组）。这应会类似于文本文件模式自动提供的功能，但应还需要额外的人工。文本文件模式为我们省却了该项例行工作。此外，MQL5 文件子系统会隐式执行一些可选操作，但却是处理文本时所需的重要操作。

首先，文本概念是基于一些使用定界符字符的一般规则。尤其是假定所有文本均由字符串组成。这样在算法上更方便读取和分析。因此，有将一个特殊字符可用于将不同字符串分隔开。

这里我们面临第一个难点：操作系统不同，接受的分隔符组合也不同。在 Windows 中，行分隔符是两个字符 '\r\n' 构成的序列（十六进制代码为：0xD 0xA，或者作为名称 CRLF，代表回车和换行）。在 Unix 和 Linux 中，标准分隔符是单字符 '\n'，但在 MacOS 中的一些版本和程序可能使用单字符 '\r'。

虽然 MetaTrader5 是在 Windows 下运行，但是我们不能保证任何生成的文本文件保存后都不会带有异常分隔符。如果我们要以二进制模式读取，并自行检查定界符以形成字符串，则这些不一致将需要特定处理。在这里，MQL5 中的文件操作的文本模式可派上用场：该模式可在读写时自动标准化换行符。

MQL5 不一定在所有情况下均能够修复换行符。尤其是在读取文本文件时，单字符 '\r' 无法被解读为 '\r\n'，而单字符 '\n' 可以被正确解读为 '\r\n'。

其次，字符串可以以多种表示方式存储在内存中。默认情况下，MQL5 中的字符串（[字符串类型](#)）由两字节 [字符](#) 组成。这提供了对通用 Unicode 编码的支持，这很好，因为 Unicode 包括了各国文字。然而很多情况下并不需要这种通用性（例如以英文存储数字或消息时），若是不需要这种通用性，使用 ANSI 编码的单字节字符更高效。MQL5 API 函数允许你选择以文本模式将字符串写入文件的首先方式。但是如果我们将写入控制在我们的 MQL 程序中的写入，则我们可以保证从 Unicode 切换到单字节字符的有效性和可靠性。在此情况下，与某些外部软件或 Web 服务集成时，其文件中的 ANSI 代码页可能是任何语言的。由此引出了下一个要点。

第三点，由于存在多种不同的语言，你需要准备好处理各种 ANSI 编码的文本。如果无法正确解析编码，文本读写时可能会出现乱码，甚至无法读取。我们已在 [处理符号和代码页](#) 中了解过它。因此，文件函数包含了用于正确字符处理的方法：只需在参数中指定所需或预期的编码即可。编码选择在 [单独章节](#) 中详细介绍。

最后，文本模式内置支持流行的 CSV 格式。由于交易通常要求使用表数据，CSV 很适合于这类用途。在 CSV 模式的文本文件中，MQL5 API 函数不仅处理用于文本换行的定界符，而且处理用于列边框（表每行中的字段）的附加定界符。这通常是制表符 '\t'、逗号 ',' 或者分号 ';'。例如，一个包含外汇资讯的 CSV 文件大致如下所示（显示逗号分隔的部分）：

```
Title,Country,Date,Time,Impact,Forecast,Previous
Bank Holiday,JPY,08-09-2021,12:00am,Holiday,,
CPI y/y,CNY,08-09-2021,1:30am,Low,0.8%,1.1%
PPI y/y,CNY,08-09-2021,1:30am,Low,8.6%,8.8%
Unemployment Rate,CHF,08-09-2021,5:45am,Low,3.0%,3.1%
German Trade Balance,EUR,08-09-2021,6:00am,Low,13.9B,12.6B
Sentix Investor Confidence,EUR,08-09-2021,8:30am,Low,29.2,29.8
JOLTS Job Openings,USD,08-09-2021,2:00pm,Medium,9.27M,9.21M
FOMC Member Bostic Speaks,USD,08-09-2021,2:00pm,Medium,,
FOMC Member Barkin Speaks,USD,08-09-2021,4:00pm,Medium,,
BRC Retail Sales Monitor y/y,GBP,08-09-2021,11:01pm,Low,4.9%,6.7%
Current Account,JPY,08-09-2021,11:50pm,Low,1.71T,1.87T
```

为便于清晰易懂，表形式如下：

Title	Country	Date	Time	Impact	Forecast	Previous
Bank Holiday	JPY	08-09-2021	12:00am	Holiday		
CPI y/y	CNY	08-09-2021	1:30am	Low	0.8%	1.1%
PPI y/y	CNY	08-09-2021	1:30am	Low	8.6%	8.8%
Unemployment Rate	CHF	08-09-2021	5:45am	Low	3.0%	3.1%
German Trade Balance	EUR	08-09-2021	6:00am	Low	13.9B	12.6B
Sentix Investor Confidence	EUR	08-09-2021	8:30am	Low	29.2	29.8
JOLTS Job Openings	USD	08-09-2021	2:00pm	Medium	9.27M	9.21M
FOMC Member Bostic Speaks	USD	08-09-2021	2:00pm	Medium		
FOMC Member Barkin Speaks	USD	08-09-2021	4:00pm	Medium		
BRC Retail Sales Monitor y/y	GBP	08-09-2021	11:01pm	Low	4.9%	6.7%
Current Account	JPY	08-09-2021	11:50pm	Low	1.71T	1.87T

4.5.2 简化模式下的文件读写

在用于读写数据的 `SQL5` 文件函数中，分为两个数量不对等的组。第一组包括两个函数：***FileSave*** 和 ***FileLoad***，用于在单次函数调用中以二进制模式读写数据。一方面，该方法具有不可否认的优势：简单；但另一方面，也存在某些局限性（下文详述）。在第二个大组中，所有文件函数均以不同的方式使用：要执行一个逻辑完整的读或写操作，必须按顺序调用若干函数。这看起来比较复杂，但却提供了灵活性以及对过程的控制。第二组函数使用特殊整数（文件描述符）操作，这些整数应使用 ***FileOpen*** 函数获取（参见 下一章节）。

我们看看这两个函数的正式描述，然后研究它们的示例（***FileSaveLoad.mq5***）。

`bool FileSave(const string filename, const void &data[], const int flag = 0)`

该函数将传递的 ***data*** 数组的所有元素写到一个名为 ***filename*** 的二进制文件。***filename*** 参数可不仅包含文件名，而且还包含若干嵌套层级的文件夹的名称：如果指定文件夹尚不存在，该函数将创建这些文件夹。如果文件存在，将被覆写（除非被另一个程序占用）。

如同 ***data*** 参数一样，可传递任何内置类型的数组，字符串除外。它还可以是包含内置类型字段的简单结构体数组，但字符串、动态数组和指针除外。也不支持类。

如果必要，***flag*** 参数可包含预定义常量 `FILE_COMMON`，该常量表示创建一个文件并将其写入到所有终端的共用数据目录（***Common/Files/***）。如果未指定 ***flag*** 参数（对应于默认值 0），则文件被写入到常规数据目录（如果 `SQL` 程序在终端中运行）或者写入到测试代理目录（如果是在测试程序中）。在后两种情况下，在该目录中使用 ***SQL5/Files/*** 沙盒，如本章开头部分所描述。

该函数返回操作成功（***true***）或出错（***false***）标志。

```
long FileLoad(const string filename, void &data[], const int flag = 0)
```

该函数将二进制文件 *filename* 的全部内容读取到指定的 *data* 数组。文件名可包括 *MQL5/Files* 或 *Common/Files* 沙盒中的文件夹层次。

data 数组必须是除 *string* 以外的任何内置类型，或者是简单结构体类型（见上文）。

flag 参数控制要搜索和打开文件的目录的选择：默认情况下（默认值 0），该目录是标准沙盒，但是如果设置了 `FILE_COMMON` 值，则是由所有终端共享的沙盒。

该函数返回读取的项数，如果出错，则返回 -1。

请注意，请注意，文件中的数据按一个数组元素大小的块读取。如果文件大小不是元素大小的倍数，则跳过剩余数据（不读取）。例如，如果文件大小是 10 字节，将其读取到一个 *double* 类型（`sizeof(double)=8`）的数组中，则实际仅载入 8 个字节，即 1 个元素（该函数将返回 1）。文件末尾的剩余 2 字节将被忽略。

在 *FileSaveLoad.mq5* 脚本中，我们定义两个结构体用于测试。

```
struct Pair
{
    short x, y;
};

struct Simple
{
    double d;
    int i;
    datetime t;
    color c;
    uchar a[10]; // fixed size array allowed
    bool b;
    Pair p;      // compound fields (nested simple structures) are also allowed

    // strings and dynamic arrays will cause a compilation error when used
    // FileSave/FileLoad: structures or classes containing objects are not allowed
    // string s;
    // uchar a[];

    // pointers are also not supported
    // void *ptr;
};
```

Simple 结构体包含大多数允许类型的字段，以及一个包含 *Pair* 结构体类型的复合字段。在 *OnStart* 函数中，我们填充一个 *Simple* 类型的小型数组。

```

void OnStart()
{
    Simple write[] =
    {
        {+1.0, -1, D'2021.01.01', clrBlue, {'a'}, true, {1000, 16000}},
        {-1.0, -2, D'2021.01.01', clrRed, {'b'}, true, {1000, 16000}},
    };
    ...
}

```

我们将选择与 MQL5Book 子文件夹一起写入数据的文件，以便我们的试验不会与你的工作文件混合：

```
const string filename = "MQL5Book/rawdata";
```

我们将一个数组写到一个文件，将其读入另一个数组，然后比较它们。

```

PRT(FileSave(filename, write/*, FILE_COMMON*/)); // true

Simple read[];
PRT(FileLoad(filename, read/*, FILE_COMMON*/)); // 2

PRT(ArrayCompare(write, read)); // 0

```

FileLoad 返回 2，即读取了 2 个元素 2 个结构体。如果比较结果为 0，则表示数据匹配。你可以在自己喜欢的文件管理器中打开 **MQL5/Files/MQL5Book** 文件夹，然后确认存在 'rawdata' 文件（不建议使用文本编辑器查看其内容，建议使用支持二进制模式的查看器）。

在脚本中，我们进一步将读取的结构体数组转换为字节，并将它们以十六进制代码的形式输出到日志。这是一种内存转储，可让你了解什么是二进制文件。

```

uchar bytes[];
for(int i = 0; i < ArraySize(read); ++i)
{
    uchar temp[];
    PRT(StructToCharArray(read[i], temp));
    ArrayCopy(bytes, temp, ArraySize(bytes));
}
ByteArrayPrint(bytes);

```

结果：

```

[00] 00 | 00 | 00 | 00 | 00 | 00 | F0 | 3F | FF | FF | FF | FF | 00 | 66 | EE | 5F |
[16] 00 | 00 | 00 | 00 | 00 | 00 | FF | 00 | 61 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
[32] 00 | 00 | 01 | E8 | 03 | 80 | 3E | 00 | 00 | 00 | 00 | 00 | 00 | F0 | BF | FE |
[48] FF | FF | FF | 00 | 66 | EE | 5F | 00 | 00 | 00 | 00 | FF | 00 | 00 | 00 | 62 |
[64] 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 01 | E8 | 03 | 80 | 3E |

```

由于内置 **ArrayPrint** 函数不能打印十六进制格式，我们必须开发自己的函数 **ByteArrayPrint**（此处未提供其源代码，参见随附文件）。

接下来，我们别忘了，**FileLoad** 能够将数据载入任何类型的数组，因此我们将使用该函数直接把同一个文件读取到字节数组中。


```
uchar bytes2[];
PRT(FileLoad(filename, bytes2/*, FILE_COMMON*/)); // 78, 39 * 2
PRT(ArrayCompare(bytes, bytes2)); // 0, equality
```

成功比较两个字节数组可表明, **FileLoad** 可以按指定方式任意操作文件中的原始数据 (文件中并未存储其包含 **Simple** 结构体数组的信息)。

这里务必要注意的是, 由于字节类型的最小大小为 1, 因此它是任何文件大小的倍数。因此, 任何文件始终能被完整读取到字节数组中, 无任何剩余。这里 **FileLoad** 函数返回了数字 78 (元素数量等于字节数)。这是该文件的大小 (两个结构体, 每个 39 字节)。

基本上来说, **FileLoad** 解读任何类型数据的能力需要编程人员的介入和校验。尤其是在脚本中, 我们将相同文件读入结构数组 **MqlDateTime**。这当然是错误的, 但却没有报错。

```
MqlDateTime mdt[];
PRT(sizeof(MqlDateTime)); // 32
PRT(FileLoad(filename, mdt)); // 2
// attention: 14 bytes left unread
ArrayPrint(mdt);
```

结果包含一系列无意义的数字:

	[year]	[mon]	[day]	[hour]	[min]	[sec]	[day_of_week]	[day_of_year]
[0]	0	1072693248	-1	1609459200	0	16711680	97	
[1]	-402587648	4096003	0	-20975616	16777215	6286950	-16777216	1644167

因为 **MqlDateTime** 的大小是 32, 所以一个 78 字节的文件只能包含两个这样的结构体, 还剩余 14 个字节。存在剩余表示有问题。但即使没有剩余, 也不保证执行的操作有意义, 因为两个不同的大小可能纯属巧合地与文件长度成倍数关系, 但含义并不相同。此外, 不同含义的两个结构体可能具有相同大小, 但这并不意味着它们彼此之间可以相互读写。

毫不意外, 结构体数组 **MqlDateTime** 的日志显示奇怪的值, 因为它实际上是完全不同的数据类型。

为了使读取操作更谨慎, 脚本实现了 **FileLoad** 函数的类似函数 **-MyFileLoad**。我们将在后续章节中详细分析该函数及其配套函数 **MyFileSave**, 届时将学习新的文件函数, 并将它们用于对内部结构体 **FileSave/FileLoad** 进行建模。同时, 请注意, 在我们的版本中, 我们能够检查文件中是否存在未读取剩余部分并显示警告。

最后, 我们看看脚本中演示的更多的几个潜在错误。

```
/*
// compilation error, string type not supported here
string texts[];
FileSave("any", texts); // parameter conversion not allowed
*/

double data[];
PRT(FileLoad("any", data)); // -1
PRT(_LastError); // 5004, ERR_CANNOT_OPEN_FILE
```

第一个错误发生在编译时 (这是代码块被注释掉的原因), 因为不允许字符串数组。

第二个错误是读取不存在的文件, 这是 **FileLoad** 返回 -1 的原因。解释性错误代码可使用 **GetLastError** (或 **_LastError**) 轻松获得。

4.5.3 打开和关闭文件

要从文件读写数据，大多数 MQL5 函数要求首先打开该文件。为此，提供了 **FileOpen** 函数。执行要求的操作后，应使用 **FileClose** 函数关闭打开的文件。事实是，根据应用的选项，打开的文件可能被阻止从其它程序访问。此外，出于性能考虑，文件操作会在内存（缓存）中进行缓冲，如果不关闭文件，新数据在一定时间内可能不会实际上传到该文件。如果要写入的数据正在等待外部程序（例如，将 MQL 程序与其它系统集成时），这一点尤其关键。根据 **FileFlush** 函数的描述，我们可以了解一种将缓冲区刷写到磁盘的替代方法。

在 MQL 程序中，一种称为描述符的特殊整数与打开的文件关联。这个整数由 **FileOpen** 函数返回。与访问或修改文件的内部内容相关的所有操作均需要在对应的 API 函数中指定该标识符。对整个文件进行操作（复制、删除、移动、检查存在性）的函数不需要描述符。你不需要打开文件就能执行这些步骤。

```
int FileOpen(const string filename, int flags, const short delimiter = '\t', uint codepage = CP_ACP)
int FileOpen(const string filename, int flags, const string delimiter, uint codepage = CP_ACP)
```

该函数以 **flags** 参数指定的模式打开具有指定名称的文件。**filename** 参数可能在实际文件名之前包含子文件夹。在此情况下，如果文件被打开以进行写入并且要求的文件夹层次不存在，则将创建该文件夹层次。

flags 参数必须包含描述处理该文件所需模式的常量组合。组合使用 按位“或”的运算来执行。下面是可用常量表。

标识符	值	说明
FILE_READ	1	打开文件进行读取
FILE_WRITE	2	打开文件进行写入
FILE_BIN	4	二进制读写模式，无需字符串到字符串的数据转换
FILE_CSV	8	CSV 类型文件；要写入的数据被转换为适当类型（Unicode 或 ANSI，见下文）的文本，读取时，执行从文本到所要求类型（在读取函数中指定）的逆转换；一条 CSV 记录对应一行文本，以换行字符（通常是 CRLF）分隔；在 CSV 记录内，元素以定界符字符（参数 <i>delimiter</i> ）分隔；
FILE_TXT	16	纯文本文件，类似于 CSV 模式，但未使用定界符字符（ <i>delimiter</i> 参数的值被忽略）
FILE_ANSI	32	ANSI 类型字符串（单字节字符）
FILE_UNICODE	64	Unicode 类型字符串（双字节字符）
FILE_SHARE_READ	128	若干个程序共享的读取访问
FILE_SHARE_WRITE	256	多个程序共享的写入访问
FILE_REWRITE	512	允许覆写函数中的文件（如已存在） FileCopy 和 FileMove
FILE_COMMON	4096	所有客户端终端的共享文件夹中的文件位置 /Terminal/Common/Files（当打开文件（ FileOpen ）、复制文件（ FileCopy 和 FileMove ）以及检查文件存在性（ FilesExist ）时，使用标志）

当打开一个文件时，必须指定 FILE_WRITE、FILE_READ 标志其组合之一。

即使有了 FILE_SHARE_READ 和 FILE_SHARE_WRITE 标志，也不意味着不需要指定 FILE_READ 和 FILE_WRITE 标志。

SQL 程序执行环境始终会对要读取的文件进行缓冲处理，这等同于隐式添加 FILE_READ 标志。因此，FILE_SHARE_READ 应始终用于正确处理共享文件（即使已知另一个进程打开了一个只写文件）。

如果没有指定 FILE_CSV、FILE_BIN 和 FILE_TXT 标志，则以 FILE_CSV 作为最高优先级。如果至少指定了这三个标志中的两个，则会应用传递的最高优先级（它们在上面以降序优先级列出）。

对于文本文件，默认模式为 FILE_UNICODE。

仅影响 CSV 的 *delimiter* 参数可以是 *ushort* 或 *string* 类型。在第二种情况下，如果字符串长度大于 1，则将仅使用其第一个字符。

codepage 参数仅影响以文本模式（FILE_TXT 或 FILE_CSV）打开的文件，且仅当为字符串选择了 FILE_ANSI 模式时才有效。如果字符串以 Unicode（FILE_UNICODE）存储，则代码页无关紧要。

如果成功，函数返回文件描述符（一个正整数）。它仅在特定 MQL 程序内唯一；没有必要与其它程序共享。为进一步处理文件，描述符被传递到对其它函数的调用。

如果出错，结果为 `INVALID_HANDLE (-1)`。错误本质应通过 [GetLastError](#) 函数返回的代码来阐明。

文件打开时进行的所有操作模式设置在文件保持打开期间保持不变。如果需要更改模式，应关闭文件再以新的参数重新打开。

对于每个打开的文件，MQL 程序执行环境保持一个内部指针，也就是在文件中的当前位置。文件打开之后，指针立即设置到开头（位置 0）。在读写过程中，该位置根据从各种文件函数传递或接收到的数据量相应进行偏移。也可以直接影响位置（前移或后移）。所有这些可能情况将在后续章节讨论。

`FILE_READ` 和 `FILE_WRITE` 的不同组合可让你实现若干场景：

- `FILE_READ` - 仅当文件存在时才打开文件；否则，该函数返回错误，不会创建新文件。
- `FILE_WRITE` - 如果文件不存在，则创建新文件；或者打开现有文件，清除其内容，大小被重置为零。
- `FILE_READ|FILE_WRITE` - 打开现有文件并保留全部内容，或者如果文件不存在，则创建新文件。

可以看到，某些场景无法实现完全是因为标志的原因。尤其是如果文件尚不存在，则不能打开文件进行写入操作。这可以使用其它函数实现，例如，[FileExists](#)。此外，无法“自动”重置一个打开用于读写组合操作的文件：在此情况下，MQL5 始终保留内容。

要将数据附加到文件，必须不仅以 `FILE_READ|FILE_WRITE` 模式打开该文件，而且还要通过调用 [FileSeek](#) 将文件中的当前位置移动到尾。

正确描述文件的共享访问是成功执行 **File Open** 的前提条件。这方面的管理规则如下：

- 若未指定 `FILE_SHARE_READ` 和 `FILE_SHARE_WRITE` 标志，如果当前程序首先打开该文件，则获得对该文件的独占访问。如果该文件之前已被打开（被其它程序或同一程序），则函数调用将会失败。
- 若设置了 `FILE_SHARE_READ` 标志，程序允许后续请求打开该相同文件进行读取。如果在函数调用时该文件已被其它程序或同一程序打开进行读取，并且未设置该标志，则该函数调用将会失败。
- 若设置了 `FILE_SHARE_WRITE` 标志，程序允许后续请求打开该相同文件进行写入。如果在函数调用时该文件已被其它程序或同一程序打开进行写入，并且未设置该标志，则该函数调用将会失败。

访问共享检查不仅相对于其它 MQL 程序或 MetaTrader 5 以外的外部进程，而且相对于同一 MQL 程序（如果它重新打开该文件）。

因此，冲突最少的模式意味着同时指定两个标志，但这仍然不能保证在某一方获得对该文件的描述符且未共享的情况下，可以打开该文件。然而，根据计划的读写，应遵循更严格的规则。

例如，当打开文件进行读取时，允许其他程序读取该文件是合理的。此外，如果某个文件是追加补充型（例如日志），你可能允许其他程序对该文件写入。但是，当打开文件进行写入时，便没有必要允许其他程序进行写入了，否则会导致不可预测的数据重叠。

void FileClose(int handle)

该函数通过句柄关闭先前打开的文件。

该文件关闭后，其在程序中的句柄变为无效：试图对该文件调用任何文件函数将导致错误。但是，如果你重新打开同一文件或不同文件，你可以使用同一变量存储不同句柄。

当程序终止，打开的文件强制关闭，如果写缓冲区不为空，则被写入磁盘。但是建议显式关闭文件。

务必遵循以下规则：完成文件的处理后关闭文件。这不仅因为未关闭的文件可能导致写入的信息因被缓存而在 RAM 中保留一段时间且不会保存到磁盘（如前所述）。此外，打开的文件会消耗操作系统的内部资源，不是消耗磁盘空间。同时打开的文件的数量受到限制（可能是数百个或数千个，取决于 Windows 设置）。如果很多程序让大量文件处于打开状态，可能达到该限值，导致尝试打开新文件失败。

对此，最好使用一个具有以下功能的包装器类来防止可能的描述符丢失：该包装器类会打开文件并在创建对象时接收到一个描述符，在析构函数中释放描述符，文件自动关闭。

我们将在测试纯 *FileOpen* 和 *FileClose* 函数后创建一个包装器类。

但在深入了解文件细节之前，我们准备一个新版本的宏，用以说明我们的函数在调用日志中的输出。之所以需要使用新版本的宏，是因为截至目前，诸如 PRT 和 PRTS 的宏（在前面章节中使用）在打印期间会“吸收”函数返回值。例如，我们编写：

```
PRT(FileLoad(filename, read));
```

其中 *FileLoad* 调用的结果被发送到日志，但无法在调用代码字符串中获取它。坦白讲，我们之前并不需要它。但是现在 *FileOpen* 函数将返回一个文件描述符，并且应被存储在一个变量中，用于该文件的进一步操作。

之前的宏有两个问题。首先，它们基于函数 *Print*，该函数消耗传递的数据（将其发送到日志），但不返回任何内容。其次，有结果的变量的任何值只能从表达式获得，*Print* 调用不能成为表达式的一部分，因为它的类型为 *void*。

为解决这些问题，我们需要打印能够返回可打印值的辅助函数。我们将其调用包装进一个新的 PRTF 宏：

```
#include <MQL5Book/MqlError.mqh>

#define PRTF(A) ResultPrint(#A, (A))

template<typename T>
T ResultPrint(const string s, const T retval = 0)
{
    const string err = E2S(_LastError) + "(" + (string)_LastError + ")";
    Print(s, "=", retval, " / ", (_LastError == 0 ? "ok" : err));
    ResetLastError(); // clear the error flag for the next call
    return retval;
}
```

使用 '#' 魔法字符串转换运算符，我们获得被作为第一自变量传递到 *ResultPrint* 的代码段（表达式 A）的详细描述符。该表达式本身（宏自变量）会被求值（如果有函数，该函数被调用），其结果被作为第二自变量传递给 *ResultPrint*。接下来，常规 *Print* 函数就派上用场了，最后，将相同结果返回至调用代码。

为避免查阅帮助文档解析错误代码，准备了 *E2S* 宏，该宏使用包含所有 MQL5 错误的 MQL_ERROR 枚举。可以在头文件 *MQL5/Include/MQL5Book/MqlError.mqh* 中找到这个宏。新宏和 *ResultPrint* 函数在 PRTF.mqh 文件中定义，与测试脚本同属一个级别。

在 *FileOpenClose.mq5* 脚本中，我们尝试打开不同的文件，尤其是同一文件将并行打开若干次。这在真实程序中通常可以避免。对于程序实例中的特定文件，单个句柄即可满足大多数任务的需求。

其中一个文件 *MQL5Book/rawdata* 必须已经存在，因为它由章节 [简化模式下的文件读写](#) 中的脚本创建。另一个文件将在测试期间创建。

我们将选择文件类型 `FILE_BIN`。在这一阶段，处理 `FILE_TXT` 或 `FILE_CSV` 相似。

我们为文件描述符预留一个数组，以便在脚本结束时一次性关闭所有文件。

首先，我们以读模式打开 *MQL5Book/rawdata*，禁用访问共享。假设该文件未被任何第三方应用程序使用，则我们预期可成功接收到句柄。

```
void OnStart()
{
    int ha[4] = {}; // array for test file handles

    // this file must exist after running FileSaveLoad.mq5
    const string rawdata = "MQL5Book/rawdata";
    ha[0] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ)); // 1 / ok
```

如果我们尝试再次打开同一文件，将会碰到错误，因为第一或第二个调用均不允许共享。

```
    ha[1] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ)); // -1 / CANNOT_OPEN_FILE(5004)
```

我们关闭第一个句柄，再次打开该文件，但是以共享读权限打开，确保现在可以再次打开（尽管仍然需要允许共享读取）：

```
    FileClose(ha[0]);
    ha[0] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ | FILE_SHARE_READ)); // 1 / ok
    ha[1] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ | FILE_SHARE_READ)); // 2 / ok
```

打开文件进行写入（`FILE_WRITE`）则不行，因为对 *FileOpen* 的先前两次调用仅允许 `FILE_SHARE_READ`。

```
    ha[2] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ | FILE_WRITE | FILE_SHARE_READ)); // -1 / CANNOT_OPEN_FILE(5004)
```

现在我们尝试创建一个新文件 *MQL5Book/newdata*。如果将其以只读模式打开，则不会创建文件。

```
    const string newdata = "MQL5Book/newdata";
    ha[3] = PRTF(FileOpen(newdata, FILE_BIN | FILE_READ));
    // -1 / CANNOT_OPEN_FILE(5004)
```

要创建文件，必须指定 `FILE_WRITE` 模式（在这里，是否包含 `FILE_READ` 并不重要，但包含的话可以提高调用的通用性：我们应该还记得，在该组合中，指令保证旧文件如果存在则将被打开，或者将创建新文件）。

```
    ha[3] = PRTF(FileOpen(newdata, FILE_BIN | FILE_READ | FILE_WRITE)); // 3 / ok
```

我们尝试使用我们已经知道的函数 *FileSave* 将某些内容写入到新文件。它扮演“外部玩家”的角色，因为它绕过我们的描述符来处理文件，方式大致与其它 *MQL* 程序或第三方应用程序的处理方式相同。

```
long x[1] = {0x123456789ABCDEF0};
PRTF(FileSave(newdata, x)); // false
```

该调用失败，因为在没有共享权限的情况下打开了句柄。关闭该文件并以最大“权限”重新打开。

```
FileClose(ha[3]);
ha[3] = PRTF(FileOpen(newdata,
    FILE_BIN | FILE_READ | FILE_WRITE | FILE_SHARE_READ | FILE_SHARE_WRITE)); // 3
```

这次 *FileSave* 如预期的那样工作。

```
PRTF(FileSave(newdata, x)); // true
```

你可以在文件夹 *SQL5/Files/SQL5Book/* 中找到 *newdata* 文件，长度为 8 个字节。

请注意，我们关闭了该文件之后，其描述符返回到空闲描述符池中，下次打开一个文件（或是另一个文件）时，相同编号再次出现。

为了让文件正常关闭，我们将显式关闭所有打开的文件。

```
for(int i = 0; i < ArraySize(ha); ++i)
{
    if(ha[i] != INVALID_HANDLE)
    {
        FileClose(ha[i]);
    }
}
```

4.5.4 管理文件描述符

由于我们需要不断记住打开的文件，还要记得在退出函数时释放本地描述符，要是能把这一整个例行工作交给特殊对象处理，将会很高效。

这种做法在编程领域很普遍，称为资源获取即初始化（RAII）。使用 RAII 可更轻松控制资源，确保它们处于正确状态。特别是，若打开文件（并为其创建一个所有者对象）的函数从若干个不同位置退出，这种做法尤其有效。

RAII 的范围不限于文件。在 [对象类型模板](#) 章节中，我们创建了 *AutoPtr* 类，该类管理对象指针。它是此概念的另一个例子，因为指针同时也是资源（内存），不仅很容易导致内存泄露，而且在算法的多个不同分支中释放指针也会消耗资源。

文件包装器类还能以另一种方式发挥作用。文件 API 未提供通过描述符来获取文件名的函数（尽管内部肯定存在这样一种关系）。同时，在对象内部，我们可以存储该名称并实现我们自己的描述符绑定方式。

在最简单的情况下，我们需要某个类能存储文件描述符并且能在析构函数中自动关闭该描述符。*FileHandle.mqh* 文件中提供了一个实现示例。

```

class FileHandle
{
    int handle;
public:
    FileHandle(const int h = INVALID_HANDLE) : handle(h)
    {
    }

    FileHandle(int &holder, const int h) : handle(h)
    {
        holder = h;
    }

    int operator=(const int h)
    {
        handle = h;
        return h;
    }
    ...
}

```

两个构造函数以及一个重载赋值运算符确保一个对象绑定到一个文件（描述符）。第二个构造函数允许你将一个引用（从调用代码）传递给一个局部变量，局部变量将额外获取一个新的描述符。这将是相同描述符的一种外部别名，其可以常规方式用于其它函数调用中。

但你也可以不使用别名。在这些情况下，类定义运算符 `~`，其返回内部 *handle* 变量的值。

```

int operator~() const
{
    return handle;
}

```

最后，实现该类的核心目的是智能析构函数：


```

~FileHandle()
{
    if(handle != INVALID_HANDLE)
    {
        ResetLastError();
        // will set internal error code if handle is invalid
        FileGetInteger(handle, FILE_SIZE);
        if(_LastError == 0)
        {
            #ifdef FILE_DEBUG_PRINT
                Print(__FUNCTION__, ": Automatic close for handle: ", handle);
            #endif
            FileClose(handle);
        }
        else
        {
            PrintFormat("%s: handle %d is incorrect, %s(%d)",
                __FUNCTION__, handle, E2S(_LastError), _LastError);
        }
    }
}

```

在智能析构函数中，经过多次检查后，为受控的 *handle* 变量调用 *FileClose*。关键在于，该文件可以在程序中其它地方显式关闭，虽然有了该类之后不再需要这一操作。因此，当算法执行离开了定义 *FileHandle* 对象的块时调用析构函数，此时描述符可能已失效。为证实这点，使用对 *FileGetInteger* 函数的虚拟调用。这是虚拟的，因为它不会执行任何有用的操作。如果在调用后内部错误代码仍为 0，则描述符有效。

我们可以略过所有这些检查，直接编写以下内容：

```

~FileHandle()
{
    if(handle != INVALID_HANDLE)
    {
        FileClose(handle);
    }
}

```

如果描述符损坏，*FileClose* 将不会返回任何警告。但我们已添加了能够输出诊断信息的检查。

我们来以实操的方式测试 *FileHandle* 类：其测试脚本称为 *FileHandle.mq5*。

```

const string dummy = "MQL5Book/dummy";

void OnStart()
{
    // creating a new file or open an existing one and reset it
    FileHandle fh1(PRTF(FileOpen(dummy,
        FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ))); // 1
    // another way to connect the descriptor via '='
    int h = PRTF(FileOpen(dummy,
        FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ)); // 2
    FileHandle fh2 = h;
    // and another supported syntax:
    // int f;
    // FileHandle ff(f, FileOpen(dummy,
    //     FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));

    // data is supposed to be written here
    // ...

    // close the file manually (this is not necessary; only done to demonstrate
    // that the FileHandle will detect this and won't try to close it again)
    FileClose(~fh1); // operator '~' applied to an object returns a handle

    // descriptor handle in variable 'h' bound to object 'fh2' is not manually closed
    // and will be automatically closed in the destructor
}

```

根据日志中的输出，一切如计划正常运行：

```

FileHandle::~FileHandle: Automatic close for handle: 2
FileHandle::~FileHandle: handle 1 is incorrect, INVALID_FILEHANDLE(5007)

```

然而，如果有很多文件，为每个文件创建一个跟踪对象副本就没那么方便了。在这种情况下，最好是设计单个对象，收集给定环境（例如，在一个函数内）中的所有描述符。

这样一个类在 *FileHolder.mqh* 文件中实现，并显示在 *FileHolder.mq5* 脚本中。*FileHolder* 本身的一个副本会根据请求创建 *FileOpener* 类的辅助观察对象，该类与 *FileHandle*（尤其是析构函数）以及 *handle* 字段共享通用特性。

要通过 *FileHolder* 打开一个文件，应使用其 *FileOpen* 方法（其签名与标准 *FileOpen* 函数的签名一致）。

```

class FileHolder
{
    static FileOpener *files[];
    int expand()
    {
        return ArrayResize(files, ArraySize(files) + 1) - 1;
    }
public:
    int FileOpen(const string filename, const int flags,
                 const ushort delimiter = '\\t', const uint codepage = CP_ACP)
    {
        const int n = expand();
        if(n > -1)
        {
            files[n] = new FileOpener(filename, flags, delimiter, codepage);
            return files[n].handle;
        }
        return INVALID_HANDLE;
    }
}

```

所有 *FileOpener* 对象都会添加到 *files* 数组中以跟踪它们的生命周期。在同一位置，零值元素标记创建 *FileHolder* 对象的本地环境（代码块）的注册时刻。*FileHolder* 构造函数负责这一操作。

```

FileHolder()
{
    const int n = expand();
    if(n > -1)
    {
        files[n] = NULL;
    }
}

```

我们已经知道，在程序执行期间会进入嵌套代码块（其调用函数）。如果它们要求管理本地文件描述符，则应在其中描述 *FileHolder* 对象（为每个代码块声明一个，或者不声明）。根据堆栈规则（先入后出），所有这些描述在 *files* 加总，然后当程序退出环境时以逆序释放。每次退出时都会调用析构函数。

```

~FileHolder()
{
    for(int i = ArraySize(files) - 1; i >= 0; --i)
    {
        if(files[i] == NULL)
        {
            // decrement array and exit
            ArrayResize(files, i);
            return;
        }

        delete files[i];
    }
}

```

它的任务是移除数组中的所有 *FileOpener* 对象，直至遇到第一个零值元素，该零值元素指示上下文的边界（数组中的后续元素是来自另一个外部上下文的描述符）。

你可以自己研究整个类。

我们看看其在 *FileHolder.mq5* 测试脚本中用法。除了 *OnStart* 函数，还有 *SubFunc*。文件操作在两个上下文中均执行。

```

const string dummy = "MQL5Book/dummy";

void SubFunc()
{
    Print(__FUNCTION__, " enter");
    FileHolder holder;
    int h = PRTF(holder.FileOpen(dummy,
        FILE_BIN | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));
    int f = PRTF(holder.FileOpen(dummy,
        FILE_BIN | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));
    // use h and f
    // ...
    // no need to manually close files and track early function exits
    Print(__FUNCTION__, " exit");
}

void OnStart()
{
    Print(__FUNCTION__, " enter");

    FileHolder holder;
    int h = PRTF(holder.FileOpen(dummy,
        FILE_BIN | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));
    // writing data and other actions on the file by descriptor
    // ...
    /*
    int a[] = {1, 2, 3};
    FileWriteArray(h, a);
    */

    SubFunc();
    SubFunc();

    if(rand() > 32000) // simulate branching by conditions
    {
        // thanks to the holder we don't need an explicit call
        // FileClose(h);
        Print(__FUNCTION__, " return");
        return; // there can be many exits from the function
    }

    /*
    ... more code
    */

    // thanks to the holder we don't need an explicit call
    // FileClose(h);
    Print(__FUNCTION__, " exit");
}

```

我们尚未手动关闭任何句柄，*FileHolder* 的实例会在析构函数中自动关闭句柄。

下面是一个日志输出示例：

```
OnStart enter
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=1 / ok
SubFunc enter
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=2 / ok
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=3 / ok
SubFunc exit
FileOpener::~FileOpener: Automatic close for handle: 3
FileOpener::~FileOpener: Automatic close for handle: 2
SubFunc enter
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=2 / ok
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=3 / ok
SubFunc exit
FileOpener::~FileOpener: Automatic close for handle: 3
FileOpener::~FileOpener: Automatic close for handle: 2
OnStart exit
FileOpener::~FileOpener: Automatic close for handle: 1
```

4.5.5 选择文本模式的编码

对于写入的文本文件，编码应基于文本特性选择，或适配外部程序对于生成文件的特定要求。如果没有外部要求，你可以遵循这一规则：对于具有数字、英文字母和符号的纯文本，使用 ANSI 编码（包含 128 个国际字符表格详见 [字符串比较](#) 章节中讨论过）。处理各种语言或特殊字符时，使用 UTF-8 或 Unicode，即分别为：

```
int u8 = FileOpen("utf8.txt", FILE_WRITE | FILE_TXT | FILE_ANSI, 0, CP_UTF8);
int u0 = FileOpen("unicode.txt", FILE_WRITE | FILE_TXT | FILE_UNICODE);
```

例如，这些设置很适合用于将金融工具的名称保存到文件，因为它们有时候使用表示货币或交易模式的特殊字符。

读取你自己的文件应没有问题，因为在读取时只需指定与写入时的相同编码设置即可。然而，文本文件的来源可能各不相同。它们的编码可能未知，或者会未经通知而更改。因此，问题来了，如果一些文件以单字节字符串（ANSI）提供，一些是双字节字符串（Unicode），而一些是 UTF-8 编码，该怎么办。

可以通过程序的 [输入参数](#) 来选择编码。然而，这仅对一个文件有效，如果你必须打开很多不同文件，它们的编码可能不匹配。因此，最好指示系统即时（根据文件）选择正确的模型。

SQL5 不允许 100% 自动检查和应用正确编码，然而，有一种用于读取各种文本文件的最通用的模式。为此，你需要设置 *FileOpen* 函数的以下输入参数：

```
int h = FileOpen(filename, FILE_READ | FILE_TXT | FILE_ANSI, 0, CP_UTF8);
```

这幕后有几个因素。

首先，UTF-8 编码明确跳过上述任何 ANSI 编码中的 128 个字符（即，这些字符是“一对一”传输）。

其次，它对互联网协议来说是使用最广泛的。

再次，SQL5 具有针对两字节 Unicode 的附加内置分析，可让你在必要时将文件操作模式自动切换为 FILE_UNICODE，不论指定的参数如何。事实是，Unicode 格式的文件通常在开头有一对特殊

标识符: 0xFFFE 或者 0xFEFF。该序列称为字节顺序标记 (BOM)。这是必需的, 因为我们已经知道, 不同平台上数字的字节存储顺序可能不同 (这在 [整数中的字节顺序控制](#) 章节中讨论过)。

FILE_UNICODE 格式中, 每字符使用 2 个字节整数 (代码), 因此字节顺序变得重要, 不同于其它编码。Windows 字节顺序 BOM 是 0xFFFE。如果 MQL5 核心在一个文本文件的开头即找到该标签, 则其读取模式将自动切换为 Unicode 模式。

我们看看不同模式设置如何处理不同编码的文本文件。为此, 我们将使用 *FileText.mq5* 脚本以及具有相同内容但不同编码的若干文本文件 (括号中是字节数大小):

- ① ansil252.txt (50): European 编码 1252 (在欧洲语言的 Windows 中, 其将完整无损显示)
- ① unicode1.txt (102): 两字节 Unicode, 开头是固有的 Windows BOM 0xFFFE
- ① unicode2.txt (100): 两字节 Unicode, 没有 BOM (一般来说, BOM 可选)
- ① unicode3.txt (102): 两字节 Unicode, 开头是 Unix 固有的 BOM, 0xFEFF
- ① utf8.txt (54): UTF-8 编码

在 *OnStart* 函数中, 我们将以 *FileOpen* 的不同设置循环读取这些文件。请注意, 使用 *FileHandle* (在 [上一节](#) 中回顾过), 我们不必担心关闭文件的问题: 所有操作会在每次迭代中自动完成。

```
void OnStart()
{
    Print("====> UTF-8");
    for(int i = 0; i < ArraySize(texts); ++i)
    {
        FileHandle fh(FileOpen(texts[i], FILE_READ | FILE_TXT | FILE_ANSI, 0, CP_UTF8));
        Print(texts[i], " -> ", FileReadString(~fh));
    }

    Print("====> Unicode");
    for(int i = 0; i < ArraySize(texts); ++i)
    {
        FileHandle fh(FileOpen(texts[i], FILE_READ | FILE_TXT | FILE_UNICODE));
        Print(texts[i], " -> ", FileReadString(~fh));
    }

    Print("====> ANSI/1252");
    for(int i = 0; i < ArraySize(texts); ++i)
    {
        FileHandle fh(FileOpen(texts[i], FILE_READ | FILE_TXT | FILE_ANSI, 0, 1252));
        Print(texts[i], " -> ", FileReadString(~fh));
    }
}
```

FileReadString 函数从文件读取字符串。我们将在关于 [读写变量](#) 的章节中探讨。

这里是个带有脚本执行结果的示例日志:

```

=====> UTF-8
MQL5Book/ansi1252.txt -> This is a text with special characters: ?? / ? / ?
MQL5Book/unicode1.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode2.txt -> T
MQL5Book/unicode3.txt -> ??
MQL5Book/utf8.txt -> This is a text with special characters: ±Σ / £ / ¥
=====> Unicode
MQL5Book/ansi1252.txt -> 桔獫桀 整瑛眠瑩 𐀀 𐀁 𐀂 𐀃 𐀄 𐀅 𐀆 𐀇 𐀈 𐀉 𐀊 𐀋 𐀌 𐀍 𐀎 𐀏 𐀐 𐀑 𐀒 𐀓 𐀔 𐀕 𐀖 𐀗 𐀘 𐀙 𐀚 𐀛 𐀜 𐀝 𐀞 𐀟 𐀠 𐀡 𐀢 𐀣 𐀤 𐀥 𐀦 𐀧 𐀨 𐀩 𐀪 𐀫 𐀬 𐀭 𐀮 𐀯 𐀰 𐀱 𐀲 𐀳 𐀴 𐀵 𐀶 𐀷 𐀸 𐀹 𐀺 𐀻 𐀼 𐀽 𐀾 𐀿
MQL5Book/unicode1.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode2.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode3.txt -> 𐀀 𐀁 𐀂 𐀃 𐀄 𐀅 𐀆 𐀇 𐀈 𐀉 𐀊 𐀋 𐀌 𐀍 𐀎 𐀏 𐀐 𐀑 𐀒 𐀓 𐀔 𐀕 𐀖 𐀗 𐀘 𐀙 𐀚 𐀛 𐀜 𐀝 𐀞 𐀟 𐀠 𐀡 𐀢 𐀣 𐀤 𐀥 𐀦 𐀧 𐀨 𐀩 𐀪 𐀫 𐀬 𐀭 𐀮 𐀯 𐀰 𐀱 𐀲 𐀳 𐀴 𐀵 𐀶 𐀷 𐀸 𐀹 𐀺 𐀻 𐀼 𐀽 𐀾 𐀿
MQL5Book/utf8.txt -> 桔獫桀 整瑛眠瑩 𐀀 𐀁 𐀂 𐀃 𐀄 𐀅 𐀆 𐀇 𐀈 𐀉 𐀊 𐀋 𐀌 𐀍 𐀎 𐀏 𐀐 𐀑 𐀒 𐀓 𐀔 𐀕 𐀖 𐀗 𐀘 𐀙 𐀚 𐀛 𐀜 𐀝 𐀞 𐀟 𐀠 𐀡 𐀢 𐀣 𐀤 𐀥 𐀦 𐀧 𐀨 𐀩 𐀪 𐀫 𐀬 𐀭 𐀮 𐀯 𐀰 𐀱 𐀲 𐀳 𐀴 𐀵 𐀶 𐀷 𐀸 𐀹 𐀺 𐀻 𐀼 𐀽 𐀾 𐀿
=====> ANSI/1252
MQL5Book/ansi1252.txt -> This is a text with special characters: ±? / £ / ¥
MQL5Book/unicode1.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode2.txt -> T
MQL5Book/unicode3.txt -> pÿ
MQL5Book/utf8.txt -> This is a text with special characters: Â±Î£ / Â£ / Â¥

```

unicode1.txt 文件始终被正确读取，因为它有 BOM 0xFFFE，系统忽略源代码中的设置。然而，如果标签缺失或为大端模式，则这种自动检测不起作用。同时，若设置了 `FILE_UNICODE`，我们便无法读取单字节文本和 UTF-8。

因此，前述 `FILE_ANSI` 和 `CP_UTF8` 的组合应被视为更能适应格式设置中的变化。仅当明确要求时，才建议选择特定国家代码页。

尽管在文本模式下处理文件时，API 为编程人员提供了极大帮助，但如果必要，我们可以避免 `FILE_TXT` 或 `FILE_CSV` 模式，而是以二进制模式 `FILE_BINARY` 打开一个文本文件。这将把解析文本和确定编码的所有复杂任务将由编程人员负责完成，但能够支持其它非标准格式。但要点在于，可以从以二进制模式打开的文件读取或写入文本。但是，反过来一般不行。以文本模式打开的带有任意数据的二进制文件（即，其不是仅包含字符串）将很大可能被解读为乱码文本。如果想要将二进制数据写入到一个文本文件，则首先使用 [CryptEncode](#) 函数和 `CRYPT_BASE64` 编码。

4.5.6 读写数组

两个 MQL5 函数设计用于读写数组：`FileWriteArray` 和 `FileReadArray`。利用二进制文件，可让你处理除字符串以外任何内置类型的数组，以及不包含字符串字段、对象、指针和动态数组的简单结构体数组。这些限制与读写过程的优化相关，由于排除了可变长度类型，这种优化才得以实现。字符串、对象和动态数组就是可变长度类型的数组。

同时，在处理文本文件时，这些函数能够操作 `string` 类型的数组（这些函数不允许具有 `FILE_TXT/FILE_CSV` 模式的文件中其它类型的数组）。这些数组使用以下格式存储在文件中：每行一个元素。

如果你需要没有类型限制地在文件中存储结构体或类，请使用类型特定函数（每次调用处理一个值）。这些函数在有关读写内置类型变量的两个章节中描述：分别针对 [二进制](#) 和 [文本](#) 文件。

此外，对于带字符串结构体的支持可通过信息存储的内部优化来实现。例如，相比使用字符串字段，你可以改为使用整数字段，此类字段将在包含带字符串的单独数组中存储对应字符串的索引。鉴于使用 OOP 工具重新定义多种运算（尤其是赋值运算）以及按编号获取数组的结构元素的可能性，算法的外观形式将基本不变。但在编写时，你可以首先以二进制模式打开一个文

件，然后为具有简化结构类型体的数组调用 `FileWriteArray`，然后以文本模式重新打开该文件，并再次 `FileWriteArray` 将一个所有字符串的数组添加到该文件。要读取这样一个文件，你应在其开头提供一个包含数组中元素数量的标头，以便将其作为 `count` 参数传递到 `FileReadArray` 中（参见下文）。

如果你需要保存或读取的不是一个结构体数组，而是单个结构，则使用 `FileWriteStruct` 和 `FileReadStruct` 函数，这两个函数在下一章节中描述。

我们来了解函数签名，然后看一个普通示例 (`FileArray.mq5`)。

```
uint FileWriteArray(int handle, const void &array[], int start = 0, int count = WHOLE_ARRAY)
```

该函数将 `array array` 写入到一个具有 `handle` 描述符的文件。该数组可以是多维数组。`start` 和 `count` 参数允许设置元素范围；默认等于整个数组。对于多维数组，`start` 索引和元素数量 `count` 指跨所有维度（而不只是数组的第一维度）的连续编号。例如，如果数组具有配置 `[] [5]`，则等于 7 的 `start` 值将指向索引为 `[1][2]` 的元素，并且 `count = 2` 将把元素 `[1][3]` 添加到数组。

该函数返回写入元素的数量。如果出错，则将返回 0。

如果以二进制模式接收到 `handle`，则数组可以是除字符串以外的任何内置类型，或者简单结构体类型。如果 `handle` 以任何文本模式打开，则数组必须是 `string` 类型。

```
uint FileReadArray(int handle, const void &array[], int start = 0, int count = WHOLE_ARRAY)
```

该函数将数据从一个具有 `handle` 描述符的文件读入到一个数组。该数组可以是多维数组和动态数组。对于多维数组，`start` 和 `count` 参数基于如上所述的所有维度中元素的连续编号生效。动态数组在必要时自动增加大小以适配要读取的数据。如果 `start` 大于数组的原始长度，则这些中间元素将在内存分配后包含随机数据（参见示例）。

请注意，该函数不能控制写文件时使用的数组配置与读取时接收数组配置的一致性。一般来说，无法保证被读取的文件是以 `FileWriteArray` 写入。

为检查数据结构的有效性，通常使用初始标头的预定义格式或文件中的其它描述符。这些函数将读取在文件大小内的任何文件内容，并将其放入指定数组中。

如果以二进制模式接收到 `handle`，则数组可以是任何内置非字符串类型或简单结构体类型。如果 `handle` 以文本模式打开，则数组必须是 `string` 类型。

我们使用 `FileArray.mq5` 脚本检查二进制模式和文本模式工作情况。为此，我们将保留两个文件名。

```
const string raw = "MQL5Book/array.raw";
const string txt = "MQL5Book/array.txt";
```

`OnStart` 函数中描述了三个 `long` 类型的数组和两个 `string` 类型的数组。仅每种类型的第一个数组被填充数据，所有其它数组将在文件被写入后检查读取情况。

```

void OnStart()
{
    long numbers1[][2] = {{1, 4}, {2, 5}, {3, 6}};
    long numbers2[][2];
    long numbers3[][2];

    string text1[][2] = {{"1.0", "abc"}, {"2.0", "def"}, {"3.0", "ghi"}};
    string text2[][2];
    ...
}

```

此外，为测试结构体操作，定义了以下 3 个类型：

```

struct TT
{
    string s1;
    string s2;
};

struct B
{
private:
    int b;
public:
    void setB(const int v) { b = v; }
};

struct XYZ : public B
{
    color x, y, z;
};

```

我们将不能够在描述的函数中使用 *TT* 类型的结构体，因为它包含字符串字段。需要展示注释语句中潜在的编译错误（参见下文）。*B* 和 *XYZ* 结构体之间的继承关系以及是否存在封闭字段并不影响 *FileWriteArray* 和 *FileReadArray* 函数。

结构体用于声明一对数组：

```

TTtt[]; // empty, because data is not important
XYZ xyz[1];
xyz[0].setB(-1);
xyz[0].x = xyz[0].y = xyz[0].z = clrRed;

```

我们从二进制模式开始。我们创建一个新文件或者打开一个现有文件，转储其内容。然后，在三次 *FileWriteArray* 调用中，我们将尝试写入三个数组：*numbers1*、*text1* 以及 *xyz*。

```

int writer = PRTF(FileOpen(raw, FILE_BIN | FILE_WRITE)); // 1 / ok
PRTF(FileWriteArray(writer, numbers1)); // 6 / ok
PRTF(FileWriteArray(writer, text1)); // 0 / FILE_NOTTEXT(5012)
PRTF(FileWriteArray(writer, xyz)); // 1 / ok
FileClose(writer);
ArrayPrint(numbers1);

```

numbers1 和 *xyz* 数组成功写入，写入的项目数表明了这一点。*text1* 数组失败，错误代码 FILE_NOTTEXT(5012)，因为字符串数组要求文件以文本模式打开。因此，内容 *xyz* 将位于文件中紧跟在 *numbers1* 的所有元素之后。

注意每个读/写函数开始在文件中的当前位置开始读/写数据，并以读/写数据的大小偏移位置。如果在写操作之前指针位于文件末尾，则文件大小增加。如果在读取过程中到达文件末尾，则指针不再移动，系统发出特殊内部错误代码 5027 (FILE_ENDOFFILE)。在零大小的新文件中，开头和末尾相同。

从数组 *text1* 写入了 0 项，因此文件中无任何内容提示你在两次成功的 *FileWriteArray* 调用之间有一次失败。

在测试脚本中，我们只是将函数结果和状态（错误代码）输出到日志，但在真实程序中，我们应即时分析问题并采取某些措施：修正参数中的错误以及文件设置中的问题，或者中断进程并向用户发送消息。

我们将文件读入 *numbers2* 数组。

```

int reader = PRTF(FileOpen(raw, FILE_BIN | FILE_READ)); // 1 / ok
PRTF(FileReadArray(reader, numbers2)); // 8 / ok
ArrayPrint(numbers2);

```

由于不同的数组被写入文件（不仅是 *numbers1*，还有 *xyz*），8 个元素被读入接收数组中（即从头至尾整个文件，因为未使用参数指定范围）。

实际上，结构体 *XYZ* 的大小是 16 个字节（4 个 4 字节的字段）：一个 *int* 和三个 *color*，其对应于 *numbers2* 数组中的一行（2 个 *long* 类型的元素）。在这里只是巧合。因为如上面所提到的，函数并不清楚原始数据的配置和大小，可能将任何内容读入任何数组：编程人员必须监控操作的有效性。

我们比较初始状态和接收到的状态。源数组 *numbers1*：

	[,0][,1]
[0,]	1 4
[1,]	2 5
[2,]	3 6

生成数组 *numbers2*：

	[,0]	[,1]
[0,]	1	4
[1,]	2	5
[2,]	3	6
[3,]	1099511627775	1095216660735

numbers2 数组的开头与原始 *numbers1* 数组完全匹配，即通过文件进行的读写操作正常。

最后一行完全由单个结构体 *XYZ* 占用（值正确，但表示方式不正确，表示为两个 *long* 类型的数字）。

现在我们转到文件开头（使用 *FileSeek* 函数，我们将稍后在 [文件内的位置控制](#) 章节中讨论该函数），并调用 *FileReadArray* 指示元素的编号和数量，即我们执行部分读取。

```
PRTF(FileSeek(reader, 0, SEEK_SET)); // true
PRTF(FileReadArray(reader, numbers3, 10, 3));
FileClose(reader);
ArrayPrint(numbers3);
```

从该文件读取了三个元素，并从索引 10 开始读入接收数组 *numbers3*。由于该文件是从开头读取，因此这些元素是值 1、4、2。由于二维数组具有配置 `[] [2]`，因此穿透索引 10 指向元素 `[5,0]`。内存中的情况是这样的：

	[,0]	[,1]
[0,]	1	4
[1,]	1	4
[2,]	2	6
[3,]	0	0
[4,]	0	0
[5,]	1	4
[6,]	2	0

标为黄色的项是随机产生的（对于不同的脚本运行可能有所变化）。有可能它们全部为零，但不能保证。*numbers3* 数组最初为空，*FileReadArray* 调用发起内存分配，且分配以偏移 10 接收 3 个元素（共 13）所需的内存量。所选块未填充任何内容，仅从该文件读取 3 个数字。因此，具有穿透索引 0 至 9 的元素（即前 5 行）以及最后一个元素（索引 13）包含无用信息。

多维数组沿第一维缩放，因此，增加 1 个数字表示沿更高维度添加整个配置。在此情况下，分布涉及一个由两个数字组成的系列（`[] [2]`）。换言之，要求的大小 13 被舍入为二的倍数，亦即 14。

最后，我们测试函数处理字符串数组的方式。我们创建一个新文件或者打开一个现有文件，转储其内容。然后，在两次 *FileWriteArray* 调用中，我们将写入 *text1* 和 *numbers1* 数组。

```
writer = PRTF(FileOpen(txt, FILE_TXT | FILE_ANSI | FILE_WRITE)); // 1 / ok
PRTF(FileWriteArray(writer, text1)); // 6 / ok
PRTF(FileWriteArray(writer, numbers1)); // 0 / FILE_NOTBIN(5011)
FileClose(writer);
```

字符串数组成功保存。数值数组被忽略并提示 `FILE_NOTBIN(5011)` 错误，因为必须以二进制模式打开文件。

当尝试写入一个结构体数组 *tt* 时，会出现编译错误，并显示长消息“不允许使用带对象的结构体或类”。编译器的实际意思是，它不接受像 *string* 这样的字段（假定字符串和动态数组具有某些服务对象的内部表示）。因此，尽管文件是以文本模式打开并且在结构体中仅有文本字段，这一组合在 MQL5 中也不被支持。

```
// COMPILATION ERROR: structures or classes containing objects are not allowed
FileWriteArray(writer, tt);
```

字符串字段的存在使得结构体变得复杂，不适合以任何模式处理 *FileWriteArray/FileReadArray* 函数。

运行该脚本之后，你可以切换到 *MQL5/Files/MQL5Book* 目录，检查生成文件的内容。

早前在 [简化模式下的文件读写](#) 章节中，我们讨论了 *FileSave* 和 *FileLoad* 函数。在测试脚本 (*FileSaveLoad.mq5*) 中，我们使用 *FileWriteArray* 和 *FileReadArray* 实现了这些函数的等效版本。但我们尚未详细了解它们。现在我们熟悉了这些新函数，我们可以检查源代码：

```
template<typename T>
bool MyFileSave(const string name, const T &array[], const int flags = 0)
{
    const int h = FileOpen(name, FILE_BIN | FILE_WRITE | flags);
    if(h == INVALID_HANDLE) return false;
    FileWriteArray(h, array);
    FileClose(h);
    return true;
}

template<typename T>
long MyFileLoad(const string name, T &array[], const int flags = 0)
{
    const int h = FileOpen(name, FILE_BIN | FILE_READ | flags);
    if(h == INVALID_HANDLE) return -1;
    const uint n = FileReadArray(h, array, 0, (int)(FileSize(h) / sizeof(T)));
    // this version has the following check added compared to the standard FileLoad:
    // if the file size is not a multiple of the structure size, print a warning
    const ulong leftover = FileSize(h) - FileTell(h);
    if(leftover != 0)
    {
        PrintFormat("Warning from %s: Some data left unread: %d bytes",
            __FUNCTION__, leftover);
        SetUserError((ushort)leftover);
    }
    FileClose(h);
    return n;
}
```

MyFileSave 基于对 *FileWriteArray* 的单次调用实现，而 *MyFileLoad* 则基于对 *FileReadArray* 调用而实现，两者均在一对 *FileOpen/FileClose* 调用之间。在两种情况下，所有可用数据都会被读写。凭借模板，我们的函数能够接受任意类型的数组。但是如果任何不支持的类型（例如类）被推导为元参数 T，则将会发生编译错误，与对内置函数的不正确访问时的情况一样。

4.5.7 读写结构体（二进制文件）

在前一节中，我们学习了如何对结构体数组执行输入/输出操作。当读/写与单独结构体相关时，使用 *FileWriteStruct* 和 *FileReadStruct* 函数对更方便。

uint FileWriteStruct(int handle, const void &data, int size = -1)

该函数将一个简单 *data* 结构体的内容写入到一个具有 *handle* 描述符的二进制文件。如我们所知，此类结构体只能包含内置字符串类型和嵌套简单结构体的字段。

该函数的主要特征是 *size* 参数。该参数可帮助设置要写入的字节数，允许我们丢弃结构体的某些部分（其末尾）。默认情况下，该参数为 -1，表示整个结构体被保存。如果 *size* 大于结构体的大小，则超过部分被忽略，即仅写入结构体的 *sizeof(data)* 个字节。

如果成功，则该函数返回写入的字节数，如果出错，则返回 0。

```
uint FileReadStruct(int handle, void &data, int size = -1)
```

该函数从一个具有 *handle* 描述符的二进制文件将内容读取到 *data* 结构体。*size* 参数指定要读取的字节数。如果未指定或超过结构体的大小，则使用指定结构体的准确大小。

如果成功，则该函数返回读取的字节数，如果出错，则返回 0。

截断结构体末尾的选项仅出现在 *FileWriteStruct* 和 *FileReadStruct* 函数中。因此，在循环中使用这两个函数便成为保存和读取经裁剪结构体数组的最适合替代方法：*FileWriteArray* 和 *FileReadArray* 函数不具备此功能，而按单个字段读写会更耗费资源（我们将在后续章节中了解对应函数）。

应注意，要使用此功能，在设计结构体时，应将哪些不需要保存的所有临时和中间计算字段放在结构体末尾。

我们看看在 *FileStruct.mq5* 脚本中使用这两个函数的示例。

假设我们要不定期存档最新报价，以便能够将来检查它们的恒定性，或者与其它提供者的相似期间进行比较。一般来说，这可以通过 MetaTrader 5 中的“交易品种”对话框（在“柱”选项卡中）手动完成。但这会需要额外的工作以及遵循计划表才能完成。而通过程序则可以更轻松地自动。此外，可以用 CSV 格式手动导出报价，我们可能需要将文件发送到外部服务器。因此，最好是将它们以紧凑二进制形式保存。除此之外，我们假设我们对有关分时报价、点差和真实交易量的信息不感兴趣（这些信息对于外汇交易品种始终为空白）。

在 [数组的比较、排序和搜索](#) 章节中，我们探讨了 *MqlRates* 结构体和 *CopyRates* 函数。将在 [稍后](#) 详细描述这两者，现在我们先再次使用它们测试文件操作。

使用 *FileWriteStruct* 中的 *size* 参数，我们仅可保存 *MqlRates* 结构体的一部分，不含最后几个字段。

在脚本开头，我们定义宏以及测试文件的名称。

```
#define BARLIMIT 10 // number of bars to write
#define HEADSIZE 10 // size of the header of our format
const string filename = "MQL5Book/struct.raw";
```

尤其关注的是 HEADSIZE 常量。如前面提到的，这样的文件函数不负责文件中数据的一致性，以及这些数据被读入到的结构体类型。编程人员必须在他们的代码中提供此类控制。因此，通常在文件的开头写入某个标头，在该标头的帮助下，首先，可以确保这是一个所要求格式的文件，其次，可以在其中保存正确读取数据所需的元信息。

尤其是标题可指示条目数。严格来讲，后者并非始终必要，因为我们可以逐渐读取文件直至文件结束。然而，可以基于标头中的计数器一次性为所有预期记录分配内存，这样更高效。

针对我们的需求，开发了一个简单结构体 *FileHeader*。

```
struct FileHeader
{
    uchar signature[HEADSIZE];
    int n;
    FileHeader(const int size = 0) : n(size)
    {
        static uchar s[HEADSIZE] = {'C','A','N','D','L','E','S','1','.','0'};
        ArrayCopy(signature, s);
    }
}
```

```
};
```

它以文本签名“CANDLES”（在 *signature* 字段中）、版本号“1.0”（相同位置）以及条目数（*n* 字段）开始。由于我们不能将字符串字段用于签名（否则结构体将不再是简单结构体，不能满足文件函数的要求），因此文本实际上被包装在固定大小 HEADSIZE 的 *uchar* 数组中。其在实例中的初始化由构造函数基于本地静态副本完成。

在 *OnStart* 函数中，我们请求最后几个柱的 *BARLIMIT*，以 *FILE_WRITE* 模式打开文件，并以截断形式将其后跟随生成报价的标头写入到文件。

```
void OnStart()
{
    MqlRates rates[], candles[];
    int n = PRTF(CopyRates(_Symbol, _Period, 0, BARLIMIT, rates)); // 10 / ok
    if(n < 1) return;

    // create a new file or overwrite the old one from scratch
    int handle = PRTF(FileOpen(filename, FILE_BIN | FILE_WRITE)); // 1 / ok

    FileHeaderfh(n); // header with the actual number of entries

    // first write the header
    PRTF(FileWriteStruct(handle, fh)); // 14 / ok

    // then write the data
    for(int i = 0; i < n; ++i)
    {
        FileWriteStruct(handle, rates[i], offsetof(MqlRates, tick_volume));
    }
    FileClose(handle);
    ArrayPrint(rates);
    ...
}
```

对于 *FileWriteStruct* 函数中的 *size* 参数值，我们使用一个包含熟悉的运算符 *offsetof* 的表达式：*offsetof(MqlRates, tick_volume)*，即在写入到文件时，以 *tick_volume* 开头的所有字段均被抛弃。

为测试数据读取，我们以 *FILE_READ* 模式打开同一文件，并读取 *FileHeader* 结构体。

```
handle = PRTF(FileOpen(filename, FILE_BIN | FILE_READ)); // 1 / ok
FileHeader reference, reader;
PRTF(FileReadStruct(handle, reader)); // 14 / ok
// if the headers don't match, it's not our data
if(ArrayCompare(reader.signature, reference.signature))
{
    Print("Wrong file format; 'CANDLES' header is missing");
    return;
}
```

reference 结构体包含未更改的默认标头（签名）。*reader* 结构体从该文件得到 14 个字节。如果两个签名匹配，则我们可以继续，因为文件格式证明是正确的，且 *reader.n* 字段包含从该文件读取的条目数。我们为接收数组 *candles* 分配所需的内存大小并清零，然后将所有条目读入其中。


```

    PrintFormat("Reading %d candles...", reader.n);
    ArrayResize(candles, reader.n); // allocate memory for the expected data in advance
    ZeroMemory(candles);

    for(int i = 0; i < reader.n; ++i)
    {
        FileReadStruct(handle, candles[i], offsetof(MqlRates, tick_volume));
    }
    FileClose(handle);
    ArrayPrint(candles);
}

```

清零是必要的, 因为 *MqlRates* 结构体被部分读取, 如果没有清零, 剩余字段将会包含无效数据。

以下是显示 XAUUSD,H1 的初始数据 (整体) 的日志。

```

[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.08.16 03:00:00 1778.86 1780.58 1778.12 1780.56 3049 5 0
[1] 2021.08.16 04:00:00 1780.61 1782.58 1777.10 1777.13 4633 5 0
[2] 2021.08.16 05:00:00 1777.13 1780.25 1776.99 1779.21 3592 5 0
[3] 2021.08.16 06:00:00 1779.26 1779.26 1776.67 1776.79 2535 5 0
[4] 2021.08.16 07:00:00 1776.79 1777.59 1775.50 1777.05 2052 6 0
[5] 2021.08.16 08:00:00 1777.03 1777.19 1772.93 1774.35 3213 5 0
[6] 2021.08.16 09:00:00 1774.38 1775.41 1771.84 1773.33 4527 5 0
[7] 2021.08.16 10:00:00 1773.26 1777.42 1772.84 1774.57 4514 5 0
[8] 2021.08.16 11:00:00 1774.61 1776.67 1773.69 1775.95 3500 5 0
[9] 2021.08.16 12:00:00 1775.96 1776.12 1773.68 1774.44 2425 5 0

```

现在我们看看读取的内容。

```

[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.08.16 03:00:00 1778.86 1780.58 1778.12 1780.56 0 0 0
[1] 2021.08.16 04:00:00 1780.61 1782.58 1777.10 1777.13 0 0 0
[2] 2021.08.16 05:00:00 1777.13 1780.25 1776.99 1779.21 0 0 0
[3] 2021.08.16 06:00:00 1779.26 1779.26 1776.67 1776.79 0 0 0
[4] 2021.08.16 07:00:00 1776.79 1777.59 1775.50 1777.05 0 0 0
[5] 2021.08.16 08:00:00 1777.03 1777.19 1772.93 1774.35 0 0 0
[6] 2021.08.16 09:00:00 1774.38 1775.41 1771.84 1773.33 0 0 0
[7] 2021.08.16 10:00:00 1773.26 1777.42 1772.84 1774.57 0 0 0
[8] 2021.08.16 11:00:00 1774.61 1776.67 1773.69 1775.95 0 0 0
[9] 2021.08.16 12:00:00 1775.96 1776.12 1773.68 1774.44 0 0 0

```

报价匹配, 但每个结构体中的最后三个字段为空。

你可以打开 *MQL5/Files/MQL5Book* 文件夹检查 *struct.raw* 文件的内部表示 (使用支持二进制模式的查看器; 参见下面示例)。

addr	hex	byte	codes	hex	byte	codes	symbols	
0000:	43	41	4E 44	4C	45	53 31	2E 30 0A 00 00 00 B0 D4	CANDLES1.0. 00
0010:	19	61	00 00	00 00	3D 0A	D7 A3 70 CB	98 40 B8 1E	.а =.ЧЗрл>@ē.
0020:	85	EB	51 D2	9B 40	14 AE	47 E1 7A C8	9B 40 0A D7	лQT>@.°GбzИ>@.ч
0030:	A3	70	3D D2	9B 40	C0 E2	19 61 00 00	00 00 3D 0A	Зр=Т>@АВ.а =.
0040:	D7	A3	70 D2	9B 40	B8 1E	85 EB 51 DA	98 40 66 66	ЧЗрТ>@ē.лQЪ>@ff
0050:	66	66	66 C4	9B 40	EC 51	B8 1E 85 C4	9B 40 D0 F0	fffд>@MQē.д>@Pp
0060:	19	61	00 00	00 00	EC 51	B8 1E 85 C4	98 40 00 00	.а MQē.д>@
0070:	00	00	00 D1	9B 40	29 5C	8F C2 F5 C3	98 40 A4 70	С>@)\ЦВХГ>@Kp
0080:	3D	0A	D7 CC	9B 40	E0 FE	19 61 00 00	00 00 D7 A3	=.ЧМ>@аю.а ЧЗ
0090:	70	3D	0A CD	9B 40	D7 A3	70 3D 0A CD	98 40 48 E1	p=.Н>@ЧЗр=.Н>@Нб
00A0:	7A	14	AE C2	9B 40	5C 8F	C2 F5 28 C3	98 40 F0 0C	z.°B>@ЦВХ(Г>@p.
00B0:	1A	61	00 00	00 00	5C 8F	C2 F5 28 C3	98 40 8F C2	.а ЦВХ(Г>@ЦВ
00C0:	F5	28	5C C6	9B 40	00 00	00 00 00 BE	98 40 33 33	x(\Ж>@ s>@33
00D0:	33	33	33 C4	9B 40	00 1B	1A 61 00 00	00 00 85 EB	333д>@ .а л
00E0:	51	B8	1E C4	9B 40	F6 28	5C 8F C2 C4	9B 40 1F 85	Qē.д>@ц(\ЦВд>@.л
00F0:	EB	51	B8 B3	9B 40	66 66	66 66 66 B9	98 40 10 29	лQē1>@ffffffN>@.)
0100:	1A	61	00 00	00 00	EC 51	B8 1E 85 B9	98 40 71 3D	.а MQē.л>@q=
0110:	0A	D7	A3 BD	9B 40	8F C2	F5 28 5C AF	98 40 B8 1E	.чЗS>@ЦВХ(\ī>@ē.
0120:	85	EB	51 B5	9B 40	20 37	1A 61 00 00	00 00 D7 A3	лQμ>@ 7.a ЧЗ
0130:	70	3D	0A B5	9B 40	48 E1	7A 14 AE C5	98 40 8F C2	p=.μ>@Нбz.°E>@ЦВ
0140:	F5	28	5C B3	9B 40	E1 7A	14 AE 47 BA	98 40 30 45	x(\ī>@бz.°Gε>@0E
0150:	1A	61	00 00	00 00	3D 0A	D7 A3 70 BA	98 40 48 E1	.а =.ЧЗре>@Нб
0160:	7A	14	AE C2	9B 40	F6 28	5C 8F C2 B6	98 40 CD CC	z.°B>@ц(\ЦВЅ>@НМ
0170:	CC	CC	CC BF	9B 40	40 53	1A 61 00 00	00 00 A4 70	МММī>@S.a Kp
0180:	3D	0A	D7 BF	9B 40	14 AE	47 E1 7A C0	98 40 1F 85	=.чī>@.°GбzА>@.л
0190:	EB	51	B8 B6	9B 40	F6 28	5C 8F C2 B9	98 40	лQēЅ>@ц(\ЦВN>@

用于在外部查看器中呈现带报价档案的二进制文件的选项

二进制文件的典型显示方式是：左列显示地址（相对于文件开头的偏移），中间列显示字节代码，而右列显示对应字节的符号表示。第一和第二列使用十六进制记数法。取决于选择的 ANSI 代码页，右列中的字符可能不同。只有已确定存在文本的部分才值得关注。对我们来说，签名“CANDLES1.0”在一开头就明确显现。数字应按中间列分析。在该列中，例如，在签名之后，可以看到 4 字节值 0x0A000000，即逆格式式的 0x0000000A（回顾章节 [整数中的字节顺序控制](#)）：这是 10，即写入的结构体数。

4.5.8 读写变量（二进制文件）

如果一个结构体包含简单结构体所禁止的字段类型（字符串、动态数组、指针），使用前面介绍的函数将无法将其写入到文件或从文件读取。类对象也是如此。然而，此类实体通常包含程序中的大部分数据，并且也要求保存和还原它们的状态。

使用前一节中的标头结构体示例，很明显，可以避免字符串（以及其它可变长度类型），但在此情况下，就不得不创造替代性的更繁琐的算法实施（例如，以字符数组替换字符串）。

为读写任意复杂度的数据，MQL5 提供了一套低层级函数，可对特定类型的单个值进行操作：*double*、*float*、*int/uint*、*long/ulong* 或者 *string*。所有其它内置 MQL5 类型均等效于不同大小的整数：*char/uchar* 是 1 字节，*short/ushort* 是 2 字节，*color* 是 4 字节，枚举是 4 字节，*datetime* 是 8 字节。这些函数可以称为原子函数（即不可见），因为用于在位层级读写文件的函数已不再存在。

当然，逐元素读写也可消除动态数组文件操作限制。

至于对象指针，根据 OOP 范式精神，我们可以允许它们保存和还原对象：只需在每个类中实现一个接口（一系列方法），该接口负责使用低层级函数负责向文件传输和回传重要内容。然后，如果对象中包含指向其他对象的指针字段，我们直接将保存或读取任务委托给该对象，而该对象将处理自己的字段（其中可能有其它指针），该委派将逐层递进，直至涵盖所有元素。

请注意，在本节中，我们将探讨二进制文件的原子函数。文本文件的原子函数将在 下一章节中介绍。本节中所有的函数均返回写入的字节数，如果出错，则返回 0。

uint FileWriteDouble(int handle, double value)

uint FileWriteFloat(int handle, float value)

uint FileWriteLong(int handle, long value)

这些函数将 *value* (*double*, *float*, *long*) 参数中传递的对应类型的值写入到一个具有 *handle* 描述符的二进制文件。

uint FileWriteInteger(int handle, int value, int size = INT_VALUE)

该函数将 *value* 整数写入到一个具有 *handle* 描述符的二进制文件。值的大小（以字节为单位）由 *size* 参数设置，并且可以是预定义的常量之一：CHAR_VALUE (1)、SHORT_VALUE (2)、INT_VALUE (默认 4)，其对应于 *char*、*short* 以及 *int* 类型（有符号和无符号）。

该函数支持 3 字节整数的未正式公开的写入模式。不建议使用。

文件指针以写入的字节数移动（而不是以 *int* 大小）。

uint FileWriteString(int handle, const string value, int length = -1)

该函数将来自 *value* 参数的字符写入到一个具有 *handle* 描述符的二进制文件。你可以指定用于写入 *length* 参数的字符数。如果它小于字符串长度，则仅指定的字符串部分将被包括在文件中。如果 *length* 为 -1 或未指定，则整个字符串被传输到没有终止符 `null` 的文件。如果 *length* 大于字符串的长度，则额外字符以零填充。

请注意，当写入到一个以 FILE_UNICODE 标志（或者没有使用 FILE_ANSI 标志）打开的文件时，字符串以 Unicode 格式保存（每个字符占 2 个字节）。当写入到一个以 FILE_ANSI 标志打开的文件时，每个字符占用 1 个字节（外语字符可能失真）。

FileWriteString 函数也能够处理文本文件。这方面的应用在下一节中描述。

double FileReadDouble(int handle)

float FileReadFloat(int handle)

long FileReadLong(int handle)

这些函数可从具有指定描述符的二进制文件读取适当类型的数字 (*double*、*float* 或 *long*)。如果必要，将结果转换为 *ulong*（如果在文件中该位置需要无符号长整型数字）。

int FileReadInteger(int handle, int size = INT_VALUE)

该函数从一个具有 *handle* 描述符的二进制文件读取整数值。值大小（以字节数表示）在 *size* 参数中指定。

由于该函数的结果是 *int* 类型，因此，如果要求的目标类型不是 *int*，则必须显式转换为要求的目标类型（即转换为 *uint* 或 *short/ushort* 或 *char/uchar*）。否则，将至少会出现编译器警告，甚至丢失符号。

事实是，在读取 CHAR_VALUE 或 SHORT_VALUE 时，默认结果始终为正（即对应于 *uchar* 和 *ushort*，其完全适配 *int*）。在这些情况下，如果数字实际上是 *uchar* 和 *ushort* 类型，则编译器警告完全是名义上的，因为我们已经确定，在 *int* 类型的值中，仅填充 1 或 2 个低字节，并且它们是无符号的。这并不会产生失真。

然而，在将无符号值 (*char* 和 *short* 类型) 存储到文件中时，有必要进行转换，因为如果不转换，则负值将变为具有相同位表示的逆序正值 (参见 [算术类型转换](#) 一节中的“无符号和有符号整数”部分)。

任何情况下，最好进行显式类型转换以避免出现警告。

该函数支持 3 字节整数读取模式。不建议使用。

文件指针以写入的字节数移动 (而不是以 *int* 大小)。

`string FileReadString(int handle, int size = -1)`

该函数从一个具有 *handle* 描述符的文件读取一个具有以字符数表示的指定大小的字符串。处理二进制文件时，必须设置 *size* 参数 (默认值仅适合于使用分隔符字符的文本文件)。否则，不能读取字符串 (函数返回空字符串)，并且内部错误代码 `_LastError` 为 5016 (`FILE_BNSTRINGSIZE`)。

因此，即使在将字符串写入二进制文件的阶段，也需要考虑字符串的读取方式。有三个主要选项：

- ① 写入末尾带空终止符的字符串。在此情况下，必须在循环中逐字符分析它们，并将字符组合为一个字符串，直至遇到 0 为止。
- ② 始终写入固定 (预定义) 长度的字符串。对于大多数场景，长度应带余量选择，或者根据规范 (参考范围、协议等) 选择，但这并不经济，并且不能百分百保证在写入到文件时，某些罕见字符串不会被缩短。
- ③ 在字符串前面写入长度值 (整数)。

`FileReadString` 函数也能够处理文本文件。这方面的应用在下一节中描述。

另外，请注意，如果 *size* 参数为 0 (在某些计算中可能发生)，则该函数不执行读取操作：文件指针保持在相同位置，函数返回一个空字符串。

作为本节的一个示例，我们将改进上一节中的 `FileStruct.mq5` 脚本。新的程序名称为 `FileAtomic.mq5`。

任务不变：将一个给定的带有报价的截断 `MqlRates` 结构体的数字保存到一个二进制文件。但现在 `FileHeader` 结构体将变为一个类 (并且格式签名将存储在一个字符串中，而不是存储在字符数组中)。一个该类型的标头和一个报价数组将成为另一个控制类 `Candles` 的一部分，并且两个类都将继承自 `Persistent` 接口，用于将任意对象写入到一个文件以及从一个文件读取任意对象。

接口如下：

```
interface Persistent
{
    bool write(int handle);
    bool read(int handle);
};
```

在 `FileHeader` 类中，我们将实现对格式签名 (我们将其更改为 "CANDLES/1.1") 以及当前交易品种名称和图表时间范围的保存和检查 (有关 `_Symbol` 和 `_Period` 的更多信息)。

写入是在对继承自接口的 `write` 方法的实现中完成。

```

class FileHeader : public Persistent
{
    const string signature;
public:
    FileHeader() : signature("CANDLES/1.1") { }
    bool write(int handle) override
    {
        PRTF(FileWriteString(handle, signature, StringLen(signature)));
        PRTF(FileWriteInteger(handle, StringLen(_Symbol), CHAR_VALUE));
        PRTF(FileWriteString(handle, _Symbol));
        PRTF(FileWriteString(handle, PeriodToString(), 3));
        return true;
    }
}

```

签名准确根据其长度写入，因为样本存储在对象中，在读取时将设置相同的长度。

对于当前图表中的金融工具，我们首先将其名称长度保存在文件中（对于最大 255 的长度，1 个字节即足够），然后我们才保存字符串本身。

如果排除常量前缀“PERIOD_”，时间范围的名称绝不超过 3 个符号，因此，为该字符串选择一个固定长度。不带前缀的时间范围名称在辅助函数 *PeriodToString* 中获得：该函数在一个单独的头文件 *Periods.mqh* 中（将在 *交易品种和时间范围* 章节中详述讨论）。

以相反顺序在 *read* 方法中执行读取（当然，假定读取将在一个不同的新对象中执行）。

```

bool read(int handle) override
{
    const string sig = PRTF(FileReadString(handle, StringLen(signature)));
    if(sig != signature)
    {
        PrintFormat("Wrong file format, header is missing: want=%s vs got %s",
            signature, sig);
        return false;
    }
    const int len = PRTF(FileReadInteger(handle, CHAR_VALUE));
    const string sym = PRTF(FileReadString(handle, len));
    if(_Symbol != sym)
    {
        PrintFormat("Wrong symbol: file=%s vs chart=%s", sym, _Symbol);
        return false;
    }
    const string stf = PRTF(FileReadString(handle, 3));
    if(_Period != StringToPeriod(stf))
    {
        PrintFormat("Wrong timeframe: file=%s(%s) vs chart=%s",
            stf, EnumToString(StringToPeriod(stf)), EnumToString(_Period));
        return false;
    }
    return true;
}

```

如果文件中的任何特性（签名、交易品种、时间范围）与当前图表不匹配，则函数返回 *false* 以表明错误。

时间范围名称逆变换为 `ENUM_TIMEFRAMES` 枚举是由 *StringToPeriod* 函数完成，该函数同时是在 *Periods.mqh* 文件中。

以下是用于请求、保存和读取报价档案的主 *Candles* 类。

```
class Candles : public Persistent
{
    FileHeader header;
    int limit;
    MqlRates rates[];
public:
    Candles(const int size = 0) : limit(size)
    {
        if(size == 0) return;
        int n = PRTF(CopyRates(_Symbol, _Period, 0, limit, rates));
        if(n < 1)
        {
            limit = 0; // initialization failed
        }
        limit = n; // may be less than requested
    }
}
```

字段是 *FileHeader* 类型的标头、请求的柱 *limit* 数量，以及从 MetaTrader 5 接收 *MqlRates* 结构体的数组。该数组在构造函数中填充。如果出错，*limit* 字段重置为零。

衍生自 *Persistent* 接口的 *Candles* 类需要实现方法 *write* 和 *read*。在 *write* 方法中，我们首先指示头文件对象保存其自身，然后将报价数、日期范围（供参考）以及数组本身附加到文件。

```
bool write(int handle) override
{
    if(!limit) return false; // no data
    if(!header.write(handle)) return false;
    PRTF(FileWriteInteger(handle, limit));
    PRTF(FileWriteLong(handle, rates[0].time));
    PRTF(FileWriteLong(handle, rates[limit - 1].time));
    for(int i = 0; i < limit; ++i)
    {
        FileWriteStruct(handle, rates[i], sizeof(MqlRates, tick_volume));
    }
    return true;
}
```

读取则以逆顺序执行：

```

bool read(int handle) override
{
    if(!header.read(handle))
    {
        return false;
    }
    limit = PRTF(FileReadInteger(handle));
    ArrayResize(rates, limit);
    ZeroMemory(rates);
    // dates need to be read: they are not used, but this shifts the position in th
    // it was possible to explicitly change the position, but this function has not
    datetime dt0 = (datetime)PRTF(FileReadLong(handle));
    datetime dt1 = (datetime)PRTF(FileReadLong(handle));
    for(int i = 0; i < limit; ++i)
    {
        FileReadStruct(handle, rates[i], offsetof(MqlRates, tick_volume));
    }
    return true;
}

```

在用于存档报价的真实程序中，若存在日期范围，便能够按文件标头构建长期历史时段的正确系列，并且在一定程度上可防止对文件随意重命名。

有一个简单的 *print* 方法控制该过程：

```

void print() const
{
    ArrayPrint(rates);
}

```

在脚本的主函数中，我们创建两个 *Candles* 对象，先用其中一个保存报价档案，然后用一个还原报价档案。这由我们已经了解过的包装器 *FileHandle* 管理（参见 [文件描述符管理](#) 章节）。

```

const string filename = "MQL5Book/atomic.raw";

void OnStart()
{
    // create a new file and reset the old one
    FileHandle handle(PRTF(FileOpen(filename,
        FILE_BIN | FILE_WRITE | FILE_ANSI | FILE_SHARE_READ)));
    // form data
    Candles output(BARLIMIT);
    // write them to a file
    if(!output.write(~handle))
    {
        Print("Can't write file");
        return;
    }
    output.print();

    // open the newly created file for checking
    handle = PRTF(FileOpen(filename,
        FILE_BIN | FILE_READ | FILE_ANSI | FILE_SHARE_READ | FILE_SHARE_WRITE));
    // create an empty object to receive quotes
    Candles inputs;
    // read data from the file into it
    if(!inputs.read(~handle))
    {
        Print("Can't read file");
    }
    else
    {
        inputs.print();
    }
}

```

下面是一个 XAUUSD,H1 的初始数据日志示例：

```

FileOpen(filename,FILE_BIN|FILE_WRITE|FILE_ANSI|FILE_SHARE_READ)=1 / ok
CopyRates(_Symbol,_Period,0,limit,rates)=10 / ok
FileWriteString(handle,signature,StringLen(signature))=11 / ok
FileWriteInteger(handle,StringLen(_Symbol),CHAR_VALUE)=1 / ok
FileWriteString(handle,_Symbol)=6 / ok
FileWriteString(handle,PeriodToString(),3)=3 / ok
FileWriteInteger(handle,limit)=4 / ok
FileWriteLong(handle,rates[0].time)=8 / ok
FileWriteLong(handle,rates[limit-1].time)=8 / ok
[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.08.17 15:00:00 1791.40 1794.57 1788.04 1789.46 8157 5 0
[1] 2021.08.17 16:00:00 1789.46 1792.99 1786.69 1789.69 9285 5 0
[2] 2021.08.17 17:00:00 1789.76 1790.45 1780.95 1783.30 8165 5 0
[3] 2021.08.17 18:00:00 1783.30 1783.98 1780.53 1782.73 5114 5 0
[4] 2021.08.17 19:00:00 1782.69 1784.16 1782.09 1782.49 3586 6 0
[5] 2021.08.17 20:00:00 1782.49 1786.23 1782.17 1784.23 3515 5 0
[6] 2021.08.17 21:00:00 1784.20 1784.85 1782.73 1783.12 2627 6 0
[7] 2021.08.17 22:00:00 1783.10 1785.52 1782.37 1785.16 2114 5 0
[8] 2021.08.17 23:00:00 1785.11 1785.84 1784.71 1785.80 922 5 0
[9] 2021.08.18 01:00:00 1786.30 1786.34 1786.18 1786.20 13 5 0

```

下面是一个恢复的数据的示例 别忘了，结构体是根据我们的假设性技术任务以截断形式保存的）：

```

FileOpen(filename,FILE_BIN|FILE_READ|FILE_ANSI|FILE_SHARE_READ|FILE_SHARE_WRITE)=2 /
FileReadString(handle,StringLen(signature))=CANDLES/1.1 / ok
FileReadInteger(handle,CHAR_VALUE)=6 / ok
FileReadString(handle,len)=XAUUSD / ok
FileReadString(handle,3)=H1 / ok
FileReadInteger(handle)=10 / ok
FileReadLong(handle)=1629212400 / ok
FileReadLong(handle)=1629248400 / ok
[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.08.17 15:00:00 1791.40 1794.57 1788.04 1789.46 0 0 0
[1] 2021.08.17 16:00:00 1789.46 1792.99 1786.69 1789.69 0 0 0
[2] 2021.08.17 17:00:00 1789.76 1790.45 1780.95 1783.30 0 0 0
[3] 2021.08.17 18:00:00 1783.30 1783.98 1780.53 1782.73 0 0 0
[4] 2021.08.17 19:00:00 1782.69 1784.16 1782.09 1782.49 0 0 0
[5] 2021.08.17 20:00:00 1782.49 1786.23 1782.17 1784.23 0 0 0
[6] 2021.08.17 21:00:00 1784.20 1784.85 1782.73 1783.12 0 0 0
[7] 2021.08.17 22:00:00 1783.10 1785.52 1782.37 1785.16 0 0 0
[8] 2021.08.17 23:00:00 1785.11 1785.84 1784.71 1785.80 0 0 0
[9] 2021.08.18 01:00:00 1786.30 1786.34 1786.18 1786.20 0 0 0

```

很容易确保数据得以正确存储和读取。现在我们看看它们在文件中的情况：

[illegible]

在外部程序中查看具有报价档案的二进制文件的内部结构体

其中，标头的各种字段以彩色高亮显示：签名、交易品种名称长度、交易品种名称、时间范围名称等等。

4.5.9 读写变量 (文本文件)

文本文件具有它们自己的一组函数用于原子（逐元素）保存和读取数据。与前一节中的二进制文件集略有不同。还需要注意的是，不存在类似的用于将结构体或结构体数组写入/读入一个文本文件的函数。如果试图将任何这些函数用于文本文件，这些函数将无效，且引发内部错误代码 5011 (FILE NOTBN)。

我们已经知道，MQL5 中的文本文件有两种形式：纯文本和 CSV 格式的文本。对应的模式 FILE_TXT 或 FILE_CSV 是在打开文件时设置，在未关闭并重新获取句柄的情况下无法更改。它们之间的差异只有在读取文件时才表现出来。两种模式均以相同方式记录。

在 `TEXT` 模式下,每次调用读函数(我们将在本节学习的任何函数之一)都会查找文件中的下一个换行符(一个 `\n` 字符或一对 `\r\n`),并处理截至该换行符的所有内容。处理的意义在于将来自文件的文本转换为对应于调用函数的特定类型值。在最简单的情况下,如果调用了 `FileReadString` 函数,则不执行任何处理(字符串“按原样”返回)。

在 CSV 模式下,每次调用读函数,文件中的文本不仅按换行符,而且还按打开文件时指定的附加定界符进行逻辑分割。从文件当前位置到最近定界符的片段处理方式类似。

换言之，在文件内读取文本和转移内部位置是按片段从定界符到定界符完成，其中定界符不仅指 *FileOpen* 参数列表中的 *delimiter* 字符，而且指换行符（'\n'，'\r\n'）以及文件的开头和末尾。

附加定界符对于将文本写入到 FILE_TXT 和 FILE_CSV 文件具有相同效果, 但仅当使用 **FileWrite** 函数时; 其自动将该字符插入在记录的元素之间。**FileWriteString** 函数分隔符被忽略。

我们来看看对这些在函数的正式描述，然后探讨 *FileTxtCsv.mq5* 中的一个示例。

uint FileWrite(int handle, ...)

该函数属于可变参数函数类别。这些参数在函数原型中用省略号指示。仅支持内置数据类型。要写入结构体或类对象，必须解除它们的元素引用，然后单个传递元素。

该函数将在第一个之后传递的所有自变量写入到一个具有 *handle* 描述符的文本文件。与常规自变量列表一样，自变量以逗号分隔。输出到文件的自变量数量不能超过 63。

输出时，数值数据根据 (*string*) 的标准转换规则转换为文本格式。*double* 类型的值以传统格式或科学指数格式 (选择更紧凑的选项) 输出为 16 位有效数位。*float* 类型的数据以 7 个有效数位的精度显示。要以不同精度或明确指定的格式显示实数，使用 *DoubleToString* 函数 (参见 [数字转换为字符串以及相反转换章节](#))。

datetime 类型的值以 "YYYY.MM.DD hh:mm:ss" 格式输出 (参见 [日期和时间章节](#))。

标准颜色 (来自 web 颜色列表) 显示为名称，非标准颜色显示为 RGB 分量值三元组 (参见 [颜色](#))，以逗号分隔 (注意: 逗号是 CSV 中最常用的分隔符字符)。

对于枚举，显示表示元素的整数，而不是显示其标识符 (名称)。例如，当写入 FRIDAY (来自 ENUM_DAY_OF_WEEK, 参见 [枚举](#)) 时，我们在文件中得到数字 5。

bool 类型的值输出为字符串 "true" 或 "false"。

如果在打开文件时指定了一个非 0 定界符，将通过相应自变量的转换而被插入在两个相邻行之间该定界符。

所有自变量均被写入文件之后，会添加一个行终止符 '\r\n'。

该函数返回写入的字节数，如果出错，则返回 0。

uint FileWriteString(int handle, const string text, int length = -1)

该函数将 *text* 字符串参数写入到一个具有 *handle* 描述符的文本文件。*length* 参数仅适用于二进制文件，在此环境中忽略 (行完全写入)。

FileWriteString 函数也能够处理二进制文件。该函数的这一应用在前一节中介绍过。

任何分隔符 (在一行中的元素之间) 和换行符必须由编程人员插入/添加。

该函数返回写入字节数 (在 FILE_UNICODE 模式下，是以字符数表示的字符串长度的 2 倍)，如果出错，则返回 0。

string FileReadString(int handle, int length = -1)

该函数从一个具有 *handle* 描述符的文件读取一个截至下一个定界符的字符串 (CSV 文件中的定界符字符、任何文件中的换行字符，或者直至文件末尾)。*length* 参数仅适用于二进制文件，在此环境中忽略。

生成的字符串可以使用标准 [缩减规则](#) 或使用 [转换函数](#) 转换为所要求类型的值。或者，可以使用专用读取函数: *FileReadBool*、*FileReadDatetime* 和 *FileReadNumber* 在下文中介绍。

如果出错，将返回空字符串。错误代码可通过变量 *_LastError* 或函数 *GetLastError* 获取。特别要注意的是，到达文件末尾时错误代码为 5027 (FILE_ENDOFFILE)。

bool FileReadBool(int handle)

该函数读取一个 CSV 文件中直至下一个定界符的片段，或截至行末的片段，并将其转换为一个 **bool** 类型的值。如果该片段包含文本 “true” (任何大小写形式，包括混合大小写例如 “True”) 或包含一个非零数字，则我们得到 **true**。否则得到 **false**。

单词 “true” 必须占用整个读取的元素。即使字符串以 “true” 开头，但是后面还有连续部分 (例如 “True Volume”), 我们也会得到 **false**。

datetime FileReadDatetime(int handle)

该函数从 CSV 文件读取以下任一格式的字符串: “YYYY.MM.DD hh:mm:ss”、“YYYY.MM.DD” 或 “hh:mm:ss”, 并将其转换为 **datetime** 类型的值。如果该片段不包含日期和/或日期的有效文本表示, 则函数将返回零或 “奇怪” 时间, 取决于其可以解读为日期和时间片段的字符。对于空字符串或非数值字符串, 我们获得当前日期, 但时间为零。

更灵活的日期和时间读取 (支持更多格式) 可以通过组合使用两个函数实现:

StringToTime(FileReadString(handle))。有关 **StringToTime** 的更多详细信息, 参见 [日期和时间](#)。

double FileReadNumber(int handle)

该函数从读取一个 CSV 文件直至下一个定界符的片段, 或截至行末的片段, 并将其转换为一个 **double** 类型的值 (根据标准 [类型转化](#) 规则进行转换)。

请注意, **double** 可能丢失极大值的精度, 从而可能影响对于 **long/ulong** 类型大数字的读取 (**double** 中的整数的失真临界值是 9007199254740992; 此类现象的一个示例见 [联合体](#) 章节)。

在前一节中讨论过的函数, 包括 **FileReadDouble**、**FileReadFloat**、**FileReadInteger**、**FileReadLong** 和 **FileReadStruct**, 不能被应用于文本文件。

FileTxtCsv.mq5 脚本演示了如何处理文本文件。上次我们将报价上传到了一个二进制文件。现在我们将以 TXT 和 CSV 格式进行。

一般来说, MetaTrader 5 允许你通过 “交易品种” 对话框以 CSV 格式导出和导入报价。但出于学习目的, 我们将复现这一过程。此外, 软件实施允许偏离默认生成的确切格式。下面显示了以标准方式导出的 XAUUSD H1 历史记录片段。

```
<DATE> » <TIME> » <OPEN> » <HIGH> » <LOW> » <CLOSE> » <TICKVOL> » <VOL> » <SPREAD>
2021.01.04 » 01:00:00 » 1909.07 » 1914.93 » 1907.72 » 1913.10 » 4230 » 0 » 5
2021.01.04 » 02:00:00 » 1913.04 » 1913.64 » 1909.90 » 1913.41 » 2694 » 0 » 5
2021.01.04 » 03:00:00 » 1913.41 » 1918.71 » 1912.16 » 1916.61 » 6520 » 0 » 5
2021.01.04 » 04:00:00 » 1916.60 » 1921.89 » 1915.49 » 1921.79 » 3944 » 0 » 5
2021.01.04 » 05:00:00 » 1921.79 » 1925.26 » 1920.82 » 1923.19 » 3293 » 0 » 5
2021.01.04 » 06:00:00 » 1923.20 » 1923.71 » 1920.24 » 1922.67 » 2146 » 0 » 5
2021.01.04 » 07:00:00 » 1922.66 » 1922.99 » 1918.93 » 1921.66 » 3141 » 0 » 5
2021.01.04 » 08:00:00 » 1921.66 » 1925.60 » 1921.47 » 1922.99 » 3752 » 0 » 5
2021.01.04 » 09:00:00 » 1922.99 » 1925.54 » 1922.47 » 1924.80 » 2895 » 0 » 5
2021.01.04 » 10:00:00 » 1924.85 » 1935.16 » 1924.59 » 1932.07 » 6132 » 0 » 5
```

其中, 我们尤其可能不满意默认分隔符字符 (制表符, 表示为 “\t”)、列顺序, 或者日期和时间被分为两个字段。

在我们的脚本中, 我们将选择逗号作为分隔符, 并且我们将以 **MqlRates** 结构体字段的顺序生成列。上传以及随后的测试读取将在 FILE_TXT 和 FILE_CSV 模式下执行。

```
const string txtfile = "MQL5Book/atomic.txt";
const string csvfile = "MQL5Book/atomic.csv";
const short delimiter = ',';
```

报价将在 *OnStart* 函数开头以标准方式请求：

```
void OnStart()
{
    MqlRates rates[];
    int n = PRTF(CopyRates(_Symbol, _Period, 0, 10, rates)); // 10
```

我们将单独指定数组中各列的名称，并使用辅助函数 *StringCombine* 将它们组合起来。需要单独的标题，因为我们使用可选的定界符字符将它们组合为一个共用标题（一种替代解决方案可基于 *StringReplace*）。我们鼓励你独立学习源代码 *StringCombine*：它相对于内置 *StringSplit* 执行相反操作。

```
const string columns[] = {"DateTime", "Open", "High", "Low", "Close",
                          "Ticks", "Spread", "True"};
const string caption = StringCombine(columns, delimiter) + "\r\n";
```

最后一列本应名为 “Volume”，但我们使用其示例检查函数 *FileReadBool* 的执行。你可以假定当前名称的含义是 “True Volume”（但这样一个字符串不会被解释为 *true*）。

接下来，我们以 *FILE_TXT* 和 *FILE_CSV* 模式打开两个文件，并将准备好的标头写入这两个文件中。

```
int fh1 = PRTF(FileOpen(txtfile, FILE_TXT | FILE_ANSI | FILE_WRITE, delimiter)); //
int fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_WRITE, delimiter)); //

PRTF(FileWriteString(fh1, caption)); // 48
PRTF(FileWriteString(fh2, caption)); // 48
```

由于 *FileWriteString* 函数不会自动添加换行符，我们已将 “\r\n” 添加到了 *caption* 变量。

```
for(int i = 0; i < n; ++i)
{
    FileWrite(fh1, rates[i].time,
              rates[i].open, rates[i].high, rates[i].low, rates[i].close,
              rates[i].tick_volume, rates[i].spread, rates[i].real_volume);
    FileWrite(fh2, rates[i].time,
              rates[i].open, rates[i].high, rates[i].low, rates[i].close,
              rates[i].tick_volume, rates[i].spread, rates[i].real_volume);
}

FileClose(fh1);
FileClose(fh2);
```

从 *rates* 数组写入结构体字段以相同方式完成，通过在一个循环中为两个文件中分别调用 *FileWrite*。别忘了，*FileWrite* 函数会在自变量之间自动插入一个定界符字符，并在字符串末尾添加 “\r\n”。当然，可以独立将所有输出值转换为字符串并使用 *FileWriteString* 将它们发送到文件，但那样的话我们必须自行处理分隔符和换行符。某些情况下不需要分隔符和换行符，例如，如果你以 JSON 格式以紧凑形式写入时（基本上在一个大行内）。

因此，在记录阶段，两个文件是相同的方式管理，结果是相同的。这里是它们的针对 XAUUSD,H1 的内容示例（你的结果可能不同）：

```

DateTime,Open,High,Low,Close,Ticks,Spread,True
2021.08.19 12:00:00,1785.3,1789.76,1784.75,1789.06,4831,5,0
2021.08.19 13:00:00,1789.06,1790.02,1787.61,1789.06,3393,5,0
2021.08.19 14:00:00,1789.08,1789.95,1786.78,1786.89,3536,5,0
2021.08.19 15:00:00,1786.78,1789.86,1783.73,1788.82,6840,5,0
2021.08.19 16:00:00,1788.82,1792.44,1782.04,1784.02,9514,5,0
2021.08.19 17:00:00,1784.04,1784.27,1777.14,1780.57,8526,5,0
2021.08.19 18:00:00,1780.55,1784.02,1780.05,1783.07,5271,6,0
2021.08.19 19:00:00,1783.06,1783.15,1780.73,1782.59,3571,7,0
2021.08.19 20:00:00,1782.61,1782.96,1780.16,1780.78,3236,10,0
2021.08.19 21:00:00,1780.79,1780.9,1778.54,1778.65,1017,13,0

```

处理这些文件的差异将在读取阶段开始显现。

我们打开一个文本文件进行读取，并使用 `FileReadString` 函数在一个循环中“扫描”它，直至它返回一个空字符串（即直至文件末尾）。

```

string read;
fh1 = PRTF(FileOpen(txtfile, FILE_TXT | FILE_ANSI | FILE_READ, delimiter)); // 1
Print("==== Reading TXT");
do
{
    read = PRTF(FileReadString(fh1));
}
while(StringLen(read) > 0);

```

日志将显示类似这样的内容：

```

==== Reading TXT
FileReadString(fh1)=DateTime,Open,High,Low,Close,Ticks,Spread,True / ok
FileReadString(fh1)=2021.08.19 12:00:00,1785.3,1789.76,1784.75,1789.06,4831,5,0 / ok
FileReadString(fh1)=2021.08.19 13:00:00,1789.06,1790.02,1787.61,1789.06,3393,5,0 / ok
FileReadString(fh1)=2021.08.19 14:00:00,1789.08,1789.95,1786.78,1786.89,3536,5,0 / ok
FileReadString(fh1)=2021.08.19 15:00:00,1786.78,1789.86,1783.73,1788.82,6840,5,0 / ok
FileReadString(fh1)=2021.08.19 16:00:00,1788.82,1792.44,1782.04,1784.02,9514,5,0 / ok
FileReadString(fh1)=2021.08.19 17:00:00,1784.04,1784.27,1777.14,1780.57,8526,5,0 / ok
FileReadString(fh1)=2021.08.19 18:00:00,1780.55,1784.02,1780.05,1783.07,5271,6,0 / ok
FileReadString(fh1)=2021.08.19 19:00:00,1783.06,1783.15,1780.73,1782.59,3571,7,0 / ok
FileReadString(fh1)=2021.08.19 20:00:00,1782.61,1782.96,1780.16,1780.78,3236,10,0 / c
FileReadString(fh1)=2021.08.19 21:00:00,1780.79,1780.9,1778.54,1778.65,1017,13,0 / ok
FileReadString(fh1)= / FILE_ENDOFFILE(5027)

```

每次调用 `FileReadString` 均以 `FILE_TXT` 模式读取整行（截至 `\r\n`）。要将其分为两个元素，我们应实现额外处理。或者，我们可以使用 `FILE_CSV` 模式。

我们来对 CSV 文件执行相同操作。

```

fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_READ, delimiter)); // 2
Print("==== Reading CSV");
do
{
    read = PRTF(FileReadString(fh2));
}
while(StringLen(read) > 0);

```

这次日志中将有更多条目：

```

==== Reading CSV
FileReadString(fh2)=DateTime / ok
FileReadString(fh2)=Open / ok
FileReadString(fh2)=High / ok
FileReadString(fh2)=Low / ok
FileReadString(fh2)=Close / ok
FileReadString(fh2)=Ticks / ok
FileReadString(fh2)=Spread / ok
FileReadString(fh2)=True / ok
FileReadString(fh2)=2021.08.19 12:00:00 / ok
FileReadString(fh2)=1785.3 / ok
FileReadString(fh2)=1789.76 / ok
FileReadString(fh2)=1784.75 / ok
FileReadString(fh2)=1789.06 / ok
FileReadString(fh2)=4831 / ok
FileReadString(fh2)=5 / ok
FileReadString(fh2)=0 / ok
...
FileReadString(fh2)=2021.08.19 21:00:00 / ok
FileReadString(fh2)=1780.79 / ok
FileReadString(fh2)=1780.9 / ok
FileReadString(fh2)=1778.54 / ok
FileReadString(fh2)=1778.65 / ok
FileReadString(fh2)=1017 / ok
FileReadString(fh2)=13 / ok
FileReadString(fh2)=0 / ok
FileReadString(fh2)= / FILE_ENDOFFILE(5027)

```

重点在于，**FileReadString** 函数在 `FILE_CSV` 模式下将定界符字符纳入考虑，并将字符串拆分为元素。每次调用 **FileReadString** 都会从 CSV 表格返回一个值（单元格）。很明显，生成的字符串需要随后转换为适当类型。

这个问题可以使用专用函数 **FileReadDatetime**、**FileReadNumber** 和 **FileReadBool** 以通用方式解决。然而，任何情况下，开发者必须跟踪当前可读列的编号，并确定其实际含义。该算法的一个示例见测试的第三步。该算法使用同一个 CSV 文件（为简单起见，我们在每步结束时关闭该文件，并在下一步开始时打开它）。

为简化按列号向 **MqlRates** 结构体中下一字段进行赋值的操作，我们创建了包含模板方法 **set** 的子结构体 **MqlRates**：


```

struct MqlRatesM : public MqlRates
{
    template<typename T>
    void set(int field, T v)
    {
        switch(field)
        {
            case 0: this.time = (datetime)v; break;
            case 1: this.open = (double)v; break;
            case 2: this.high = (double)v; break;
            case 3: this.low = (double)v; break;
            case 4: this.close = (double)v; break;
            case 5: this.tick_volume = (long)v; break;
            case 6: this.spread = (int)v; break;
            case 7: this.real_volume = (long)v; break;
        }
    }
};

```

在 *OnStart* 函数中，我们已经描述了一个该结构体的数组，我们将在这个数组中添加传入值。需要该数组以简化使用 *ArrayPrint* 进行的日志记录（在 MQL5 中没有用于自行打印结构体的现成函数）。

```

Print("==== Reading CSV (alternative)");
MqlRatesM r[1];
int count = 0;
int column = 0;
const int maxColumn = ArraySize(columns);

```

之所以需要用于统计记录数量的 *count* 变量，不仅是出于统计目的，也是一种用作跳过第一行（包含标头，不含数据）的手段。当前列号在 *column* 变量中跟踪。其最大值不应超过列数 *maxColumn*。

现在我们只需打开该文件并使用各种函数从文件中循环读取元素，直至发生错误，尤其诸如 5027 (FILE_ENDOFFILE) 的预期错误，该错误表示已到达文件末尾。

当列号为 0 时，我们应用 *FileReadDatetime* 函数。对于其它列，则使用 *FileReadNumber*。第一行带标头的情况例外：对于这种情况，我们调用 *FileReadBool* 函数以演示它将如何处理被故意添加到最后一列的“True”标头。

```

fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_READ, delimiter)); // 1
do
{
    if(column)
    {
        if(count == 1) // demo for FileReadBool on the 1st record with headers
        {
            r[0].set(column, PRTF(FileReadBool(fh2)));
        }
        else
        {
            r[0].set(column, FileReadNumber(fh2));
        }
    }
    else // 0th column is the date and time
    {
        ++count;
        if(count > 1) // the structure from the previous line is ready
        {
            ArrayPrint(r, _Digits, NULL, 0, 1, 0);
        }
        r[0].time = FileReadDatetime(fh2);
    }
    column = (column + 1) % maxColumn;
}
while(_LastError == 0); // exit when end of file 5027 is reached (FILE_ENDOFFILE)

// printing the last structure
if(column == maxColumn - 1)
{
    ArrayPrint(r, _Digits, NULL, 0, 1, 0);
}

```

这是日志中记录的情况：


```

===== Reading CSV (alternative)
FileOpen(csvfile,FILE_CSV|FILE_ANSI|FILE_READ,delimiter)=1 / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=true / ok
2021.08.19 00:00:00 0.00 0.00 0.00 0.00 0 0 1
2021.08.19 12:00:00 1785.30 1789.76 1784.75 1789.06 4831 5 0
2021.08.19 13:00:00 1789.06 1790.02 1787.61 1789.06 3393 5 0
2021.08.19 14:00:00 1789.08 1789.95 1786.78 1786.89 3536 5 0
2021.08.19 15:00:00 1786.78 1789.86 1783.73 1788.82 6840 5 0
2021.08.19 16:00:00 1788.82 1792.44 1782.04 1784.02 9514 5 0
2021.08.19 17:00:00 1784.04 1784.27 1777.14 1780.57 8526 5 0
2021.08.19 18:00:00 1780.55 1784.02 1780.05 1783.07 5271 6 0
2021.08.19 19:00:00 1783.06 1783.15 1780.73 1782.59 3571 7 0
2021.08.19 20:00:00 1782.61 1782.96 1780.16 1780.78 3236 10 0
2021.08.19 21:00:00 1780.79 1780.90 1778.54 1778.65 1017 13 0

```

可以看到，在所有标头中，仅最后一个被转换为 *true* 值，而所有前面的标头均为 *false*。

读取结构体的内容与原始数据相同。

4.5.10 管理在文件中的位置

我们已知知道，系统将某个指针与每个打开的文件关联：该指针确定下次任何 I/O 函数被调用时在文件中开始写入或读取数据的位置（相对于开头位置的偏移）。函数执行后，指针将偏移写入或读取数据的大小。

在某些情况下，你需要在没有 I/O 操作的情况下更改指针的位置。尤其是我们需要将数据附加到文件末尾时，我们以“混合”模式 `FILE_READ | FILE_WRITE` 打开文件，此时必须想办法将指针定位在文件末尾（否则我们将从开头开始覆写数据）。虽然我们可以在有内容需要读取的情况下调用读函数（从而偏移指针），但这种方式的效率不高。最好使用专用函数 `FileSeek`。***FileTell*** 函数允许获取指针的实际值（在文件中的位置）。

在本节中，我们将探讨这两个函数以及与文件中当前位置相关的一些其它函数。有些函数对于文本和二进制模式的文件工作方式相同，而其它一些函数则不同。

`bool FileSeek(int handle, long offset, ENUM_FILE_POSITION origin)`

该函数将文件指针移动 *offset* 字节数（使用 *origin* 为参考点，该参考点是 `ENUM_FILE_POSITION` 枚举中描述的预定义位置之一）。*offset* 可以是正值（移动到文件末尾及以外）或负值（移动到开头）。`ENUM_FILE_POSITION` 具有以下成员：

- `SEEK_SET` 表示文件开头
- `SEEK_CUR` 表示当前位置
- `SEEK_END` 表示文件末尾

如果相对于锚点的新位置计算结果为负值（即请求了向文件开头左边的偏移），则文件指针将被设置为文件开头。

如果你将位置设为超出文件末尾（值大于文件大小），则随后对文件的写入将不是从文件末尾进行，而是从设置的位置进行。在此情况下，未定义的值将被写入到之前的文件末尾与给定位置之间（见下文）。

如果成功，函数返回 *true*，如果出错，返回 *false*。

ulong FileTell(int handle)

对于使用 *handle* 描述符打开的文件，该函数返回内部指针的当前位置（相对于文件开头的偏移）。如果出错，将返回 `ULONG_MAX ((ulong)-1)`。该错误代码在 *_LastError* 变量中提供，也可以通过 *GetLastError* 函数获取。

bool FileIsEnding(int handle)

该函数返回关于指针是否在 *handle* 文件末尾的指示。如果是，则结果为 *true*。

bool FileIsLineEnding(int handle)

对于具有 *handle* 描述符的文件，该函数返回一个指示文件指针是否在文件末尾的符号（紧接在换行符字符 '\n' 或 '\r\n' 之后）。换言之，返回值 *true* 表示当前位置在下一行的开头（或者在文件末尾）。对于二进制文件，结果始终为 *false*。

前述函数的测试脚本名为 *FileCursor.mq5*。它处理三个文件：两个二进制文件和一个文本文件。

```
const string fileraw = "MQL5Book/cursor.raw";
const string filetxt = "MQL5Book/cursor.csv";
const string file100 = "MQL5Book/k100.raw";
```

为简化对当前位置的记录，除了文件末尾（End-Of-File, EOF）以及行末（End-Of-Line, EOL）符号，我们还创建了辅助函数 *FileState*。

```
string FileState(int handle)
{
    return StringFormat("P:%I64d, F:%s, L:%s",
        FileTell(handle),
        (string)FileIsEnding(handle),
        (string)FileIsLineEnding(handle));
}
```

在二进制文件上测试这种函数的场景包括以下步骤。

以读/写模式新建或打开一个现有 *fileraw* 文件（"MQL5Book/cursor.raw"）。在打开之后以及在每个操作之后，立即通过调用 *FileState* 输出文件的当前状态。

```
void OnStart()
{
    int handle;
    Print("\n * Phase I. Binary file");
    handle = PRTF(FileOpen(fileraw, FILE_BIN | FILE_WRITE | FILE_READ));
    Print(FileState(handle));
    ...
}
```

将指针移动到文件末尾，这将允许我们在每次执行脚本时将数据附加到该文件（而不是从开头覆盖文件）。最明显的表示文件末尾的方式：相对于 *origin=SEEK_END* 的零 *offset*。

```
PRTF(FileSeek(handle, 0, SEEK_END));
Print(FileState(handle));
```

如果文件不再为空（非新），则我们可以在其任意位置（相对或绝对位置）读取现有数据。尤其是，若 *FileSeek* 函数的 *origin* 参数等于 *SEEK_CUR*，这表明如果 *offset* 为负，则当前位置将往后（向左）移动对应的字节数，如果为正，则往前移动（向右）。

在本示例中，我们尝试以 *int* 类型值的大小回退。稍后我们将会看到，在该位置应该有一个以下结构体的 *day_of_year* 字段（最后字段）：*MqlDateTime*，因为我们在后续指令中将其写入到一个文件，在下次运行时可从文件中读取该数据。读取的值被记录，与之前保存的值进行比较。

```
if(PRTF(FileSeek(handle, -1 * sizeof(int), SEEK_CUR)))
{
    Print(FileState(handle));
    PRTF(FileReadInteger(handle));
}
```

在一个新的空文件中，*FileSeek* 调用将以错误 4003 (*INVALID_PARAMETER*) 结束，而 *if* 语句块将不被执行。

接下来，以数据填充该文件。首先，使用 *FileWriteLong* 写入计算机的当前本地时间（8 字节的 *datetime*）。

```
datetime now = TimeLocal();
PRTF(FileWriteLong(handle, now));
Print(FileState(handle));
```

然后我们尝试从当前位置回退 4 个字节（-4）并读取 *long*。

```
PRTF(FileSeek(handle, -4, SEEK_CUR));
long x = PRTF(FileReadLong(handle));
Print(FileState(handle));
```

这一尝试将出现错误 5015 (*FILE_READERROR*)，因为此时处于文件末尾，向左偏移 4 个字节后无法从右侧读取 8 个字节（*long* 大小）。但是，如同我们将会从日志所看到的那样，虽然这一尝试不成功，但指针仍将回移到文件末尾。

如果回退 8 个字节（-8），则后续对 *long* 值的读取将会成功，并且两个时间值（包括原始值和从文件接收到的值）必须匹配。

```
PRTF(FileSeek(handle, -8, SEEK_CUR));
Print(FileState(handle));
x = PRTF(FileReadLong(handle));
PRTF((now == x));
```

最后，将填充了相同时间的 *MqlDateTime* 结构体写入到文件。文件中的位置将增加 32（以字节表示的结构体大小）。

```

MqlDateTime mdt;
TimeToStruct(now, mdt);
StructPrint(mdt); // display the date/time in the log visually
PRTF(FileWriteStruct(handle, mdt)); // 32 = sizeof(MqlDateTime)
Print(FileState(handle));
FileClose(handle);

```

在针对使用 *fileraw* 文件 (MQL5Book/cursoraw) 的场景第一次脚本运行之后, 我们得到如下内容 (时间将不相同):

```

first run
* Phase I. Binary file
FileOpen(fileraw,FILE_BIN|FILE_WRITE|FILE_READ)=1 / ok
P:0, F:true, L:false
FileSeek(handle,0,SEEK_END)=true / ok
P:0, F:true, L:false
FileSeek(handle,-1*sizeof(int),SEEK_CUR)=false / INVALID_PARAMETER(4003)
FileWriteLong(handle,now)=8 / ok
P:8, F:true, L:false
FileSeek(handle,-4,SEEK_CUR)=true / ok
FileReadLong(handle)=0 / FILE_READERROR(5015)
P:8, F:true, L:false
FileSeek(handle,-8,SEEK_CUR)=true / ok
P:0, F:false, L:false
FileReadLong(handle)=1629683392 / ok
(now==x)=true / ok
    2021      8    23      1    49    52              1          234
FileWriteStruct(handle,mdt)=32 / ok
P:40, F:true, L:false

```

根据状态, 文件大小初始为零, 因为偏移到文件末尾 (“F:true”) 之后位置为 “P:0”。每次记录之后 (使用 *FileWriteLong* 和 *FileWriteStruct*), 位置 P 将增大写入数据的大小。

在第二次脚本运行之后, 你会注意到日志中的某些变化:

```

second run
* Phase I. Binary file
FileOpen(fileraw,FILE_BIN|FILE_WRITE|FILE_READ)=1 / ok
P:0, F:false, L:false
FileSeek(handle,0,SEEK_END)=true / ok
P:40, F:true, L:false
FileSeek(handle,-1*sizeof(int),SEEK_CUR)=true / ok
P:36, F:false, L:false
FileReadInteger(handle)=234 / ok
FileWriteLong(handle,now)=8 / ok
P:48, F:true, L:false
FileSeek(handle,-4,SEEK_CUR)=true / ok
FileReadLong(handle)=0 / FILE_READERROR(5015)
P:48, F:true, L:false
FileSeek(handle,-8,SEEK_CUR)=true / ok
P:40, F:false, L:false
FileReadLong(handle)=1629683397 / ok
(now==x)=true / ok
    2021      8    23      1    49    57          1          234
FileWriteStruct(handle,mdt)=32 / ok
P:80, F:true, L:false

```

首先，打开后的文件大小为 40（依据是偏移到文件末尾后的位置“P:40”）。每次脚本运行后，文件大小将增加 40 字节。

其次，由于文件不为空，无法在其中导航并读取“旧”数据。尤其是从当前位置（同时也是文件末尾）回退到 `-1*sizeof(int)` 之后，我们成功读取了值 234，这是结构体 `MqlDateTime` 的最后字段（它是一年中“日”，对于你来说，很可能不同）。

第二个测试场景是处理文本 CSV 文件 `filetxt`（`ML5Book/cursor.csv`）。我们同样将以读写组合模式打开它，但不将指针移动到文件末尾。因此，每次脚本运行将从文件开头开始覆写数据。为方便看到差异，CSV 文件第一列中的数字是随机生成的。在第二列中，相同字符串始终替换为 `StringFormat` 函数中的模板。

```

Print(" * Phase II. Text file");
srand(GetTickCount());
// create a new file or open an existing file for writing/overwriting
// from the very beginning and subsequent reading; inside CSV data (Unicode)
handle = PRTF(FileOpen(filetxt, FILE_CSV | FILE_WRITE | FILE_READ, ','));
// three rows of data (number,string pair in each), separated by '\n'
// note that the last element does not end with a newline '\n'
// this is optional, but allowed
string content = StringFormat(
    "%02d,abc\n%02d,def\n%02d,ghi",
    rand() % 100, rand() % 100, rand() % 100);
// '\n' will be replaced with '\r\n' automatically, thanks to FileWriteString
PRTF(FileWriteString(handle, content));

```

下面是生成数据的示例：

```

34,abc
20,def
02,ghi

```

然后我们将回到文件开头，并使用 *FileReadString* 在循环中读取数据，持续记录状态。

```

PRTF(FileSeek(handle, 0, SEEK_SET));
Print(FileState(handle));
// count the lines in the file using the FileIsLineEnding feature
int lineCount = 0;
while(!FileIsEnding(handle))
{
    PRTF(FileReadString(handle));
    Print(FileState(handle));
    // FileIsLineEnding also equals true when FileIsEnding equals true,
    // even if there is no trailing '\n' character
    if(FileIsLineEnding(handle)) lineCount++;
}
FileClose(handle);
PRTF(lineCount);

```

下面是第一和第二次脚本运行后 *filetxt* 文件的日志。首先是第一个：

```

first run
* Phase II. Text file
FileOpen(filetxt,FILE_CSV|FILE_WRITE|FILE_READ,',')=1 / ok
FileWriteString(handle,content)=44 / ok
FileSeek(handle,0,SEEK_SET)=true / ok
P:0, F:false, L:false
FileReadString(handle)=08 / ok
P:8, F:false, L:false
FileReadString(handle)=abc / ok
P:18, F:false, L:true
FileReadString(handle)=37 / ok
P:24, F:false, L:false
FileReadString(handle)=def / ok
P:34, F:false, L:true
FileReadString(handle)=96 / ok
P:40, F:false, L:false
FileReadString(handle)=ghi / ok
P:46, F:true, L:true
lineCount=3 / ok

```

下面是第二个：

```

second run
* Phase II. Text file
FileOpen(filetxt,FILE_CSV|FILE_WRITE|FILE_READ,',')=1 / ok
FileWriteString(handle,content)=44 / ok
FileSeek(handle,0,SEEK_SET)=true / ok
P:0, F:false, L:false
FileReadString(handle)=34 / ok
P:8, F:false, L:false
FileReadString(handle)=abc / ok
P:18, F:false, L:true
FileReadString(handle)=20 / ok
P:24, F:false, L:false
FileReadString(handle)=def / ok
P:34, F:false, L:true
FileReadString(handle)=02 / ok
P:40, F:false, L:false
FileReadString(handle)=ghi / ok
P:46, F:true, L:true
lineCount=3 / ok

```

如你所见，文件大小不变，但以相同偏移写入了不同的数字。由于该 CSV 文件有两列，在每个我们读取的第二个值之后，我们都会看到有一个 EOL 标记（“L:true”）。

检测到的行数为 3，尽管文件中仅有 2 个换行符：最后一行（第三行）结束文件。

最后一个测试场景使用文件 *file100* (MQL5Book/k100.raw) 将指针移动超过文件末尾（至 1000000 字节标记），从而增加的文件大小（为可能的将来写操作预留磁盘空间）。

```

Print(" * Phase III. Allocate large file");
handle = PRTF(FileOpen(file100, FILE_BIN | FILE_WRITE));
PRTF(FileSeek(handle, 1000000, SEEK_END));
// to change the size, you need to write at least something
PRTF(FileWriteInteger(handle, 0xFF, 1));
PRTF(FileTell(handle));
FileClose(handle);

```

该脚本的日志输出对于每次运行都相同，只是最终位于为文件分配的空间中的随机数据可能不同（其内容在这里未显示：请使用外部二进制文件查看器）。

```

* Phase III. Allocate large file
FileOpen(file100,FILE_BIN|FILE_WRITE)=1 / ok
FileSeek(handle,1000000,SEEK_END)=true / ok
FileWriteInteger(handle,0xFF,1)=1 / ok
FileTell(handle)=1000001 / ok

```

4.5.11 获取文件特性

在处理文件的过程中，除了直接读写数据，还经常需要分析它们的特性。主要属性之一是文件大小，可以使用 *FileSize* 函数获得。还有一些特征可以使用 *FileGetInteger* 获取。

请注意，*FileSize* 函数需要打开的文件句柄。*FileGetInteger* 具有一些可以通过文件名识别的特性，包括大小，无需首先打开文件。

ulong FileSize(int handle)

该函数根据文件的描述符返回打开文件的大小。如果出错，则结果等于 0，该结果是该函数正常执行的有效大小，因此你应始终使用 `_LastError` 或 `GetLastError` 分析潜在错误。

要获取文件大小，也可以将指针移动到文件末尾 `FileSeek(handle, 0, SEEK_END)` 然后调用 `FileTell(handle)`。这两个函数已在前一节中介绍过。

long FileGetInteger(int handle, ENUM_FILE_PROPERTY_INTEGER property)

long FileGetInteger(const string filename, ENUM_FILE_PROPERTY_INTEGER property, bool common = false)

该函数有两种工作方式：通过打开的文件描述符，以及通过文件名称（包括已关闭文件的名称）。

该函数返回在 *property* 参数中指定的文件属性之一。每种方式的有效属性列表不同（见下文）。虽然值类型是 *long*，但是取决于请求的属性，它不仅可能包含整数数字，而且可能包含 *datetime* 或 *bool*：显式执行要求的类型强制转换。

通过文件名请求属性时，还可以使用 *common* 参数指定要搜索的文件所在的文件夹：当前终端文件夹 `SQL5/Files(false, 默认)` 或者公用文件夹 `Users/<user_name>...MetaQuotes/Terminal/Common/Files(true)`。如果 MQL 程序正在测试程序中运行，则工作目录位于测试代理文件夹中（`Tester/<agent>/SQL5/Files`），参见 [处理文件](#) 章节的介绍。

下表列出了 `ENUM_FILE_PROPERTY_INTEGER` 的所有成员。

特 性	说 明
FILE_EXISTS *	检查存在性 (类似于 File IsExist)
FILE_CREATE_DATE *	创建日期
FILE_MODIFY_DATE *	上次修改日期
FILE_ACCESS_DATE *	上次访问日期
FILE_SIZE *	文件大小, 以字节为单位 (类似于 FileSize)
FILE_POSITION	文件中的指针位置 (类似于 FileTell)
FILE_END	文件末尾位置 (类似于 FileIsEnding)
FILE_LINE_END	字符串末尾位置 (类似于 FileIsLineEnding)
FILE_IS_COMMON	在终端共享文件夹中打开的文件 (FILE_COMMON)
FILE_IS_TEXT	以文本方式打开的文件 (FILE_TXT)
FILE_IS_BINARY	以二进制方式打开的文件 (FILE_BIN)
FILE_IS_CSV	以 CSV 方式打开的文件 (FILE_CSV)
FILE_IS_ANSI	以 ANSI 方式打开的文件 (FILE_ANSI)
FILE_IS_READABLE	打开进行读取的文件 (FILE_READ)
FILE_IS_WRITABLE	打开进行写入的文件 (FILE_WRITE)

允许由文件名使用的特性标有星号如果你试图获取其它特性, 则该函数的第二个版本将返回错误 4003 (INVALID_PARAMETER)。

在处理打开的文件时, 某些特性可能改变: FILE_MODIFY_DATE、FILE_ACCESS_DATE、FILE_SIZE、FILE_POSITION、FILE_END、FILE_LINE_END (仅限文本文件)。

如果出错, 调用结果为 -1。

该函数的第二个版本始终允许你检查指定的名称是文件名称还是目录名称。如果在通过名称获取特性时指定了目录, 则该函数将设置一个特别内部错误代码 5018 (ERR_MQL_FILE_IS_DIRECTORY), 而返回的值将是正确的。

我们将使用 *FileProperties.mq5* 脚本来测试本节的函数。它将处理具有预定义名称的文件。

```
const string fileprop = "MQL5Book/fileprop";
```

在 *OnStart* 的开头, 我们尝试通过错误描述符请求大小 (不是通过 *File Open* 调用获得)。在 *FileSize* 之后, 要求执行 *LastError* 变量检查, *FileGetInteger* 立即返回一个特殊值, 一个错误指示器 (-1)。

```

void OnStart()
{
    int handle = 0;
    ulong size = FileSize(handle);
    if(_LastError)
    {
        Print("FileSize error=", E2S(_LastError) + "(" + (string)_LastError + ")");
        // We will get: FileSize 0, error=WRONG_FILEHANDLE(5008)
    }

    PRTF(FileGetInteger(handle, FILE_SIZE)); // -1 / WRONG_FILEHANDLE(5008)

```

接下来，我们创建一个新文件或者打开一个现有文件并重置，然后写入测试文本。

```

handle = PRTF(FileOpen(fileprop, FILE_TXT | FILE_WRITE | FILE_ANSI)); // 1
PRTF(FileWriteString(handle, "Test Text\n")); // 11

```

我们选择性地请求某些特性。

```

PRTF(FileGetInteger(fileprop, FILE_SIZE)); // 0, not written to the disk yet
PRTF(FileGetInteger(handle, FILE_SIZE)); // 11
PRTF(FileSize(handle)); // 11
PRTF(FileGetInteger(handle, FILE_MODIFY_DATE)); // 1629730884, number of seconds si
PRTF(FileGetInteger(handle, FILE_IS_TEXT)); // 1, bool true
PRTF(FileGetInteger(handle, FILE_IS_BINARY)); // 0, bool false

```

通过描述符获取的有关文件长度的信息会考虑到当前缓存缓冲区，如果信息是通过文件名称获取，则实际长度只有在文件关闭后或者调用 *FileFlush* 函数（参见 *将缓存强制写入磁盘*）。

该函数返回自 1970 年 1 月 1 日以来的标准历元的日期和时间（以秒来表示），其对应于 *datetime* 类型并且可以转换为后者。

对于具有描述符的函数版本，文件打开标志（其模式）请求会成功，特别是，我们会收到响应，告知该文件是文本而不是二进制文件。然而，下一个类似的文件名请求将会失败，因为仅当传递了有效句柄时，才支持该特性。即使名称指向我们已经打开的同一文件也会失败。

```

PRTF(FileGetInteger(fileprop, FILE_IS_TEXT)); // -1 / INVALID_PARAMETER(4003)

```

我们稍作等待，关闭文件，再次检查修改日期（这次通过名称来检查，因为描述符不再有效）。

```

Sleep(1000);
FileClose(handle);
PRTF(FileGetInteger(fileprop, FILE_MODIFY_DATE)); // 1629730885 / ok

```

这里你可以清楚看到时间已经增加 1。

最后，确保特性对目录（文件夹）可用。

```

PRTF((datetime)FileGetInteger("MQL5Book", FILE_CREATE_DATE));
// We will get: 2021.08.09 22:38:00 / FILE_IS_DIRECTORY(5018)

```

由于本书的所有示例均位于“MQL5Book”文件夹，其必须已经存在。不过你的实际创建时间将不同。在此情况下，FILE_IS_DIRECTORY 错误代码由 PRTF 宏显示。在工作程序中，应直接调用函数而不使用宏，然后在 *_LastError* 中读取代码。

4.5.12 强制将高速缓存写入磁盘

MQL5 中的文件读写采用缓存机制。这意味着系统会为数据维护内存缓冲区，从而提升工作效率。因此，在写入期间通过函数调用传输的数据进入输出缓冲区，仅当缓冲区满了之后，才进行对磁盘的物理写入。相反，在读取时，从磁盘读入缓冲区的数据量要超过程序使用函数请求的数据量（如果不是文件末尾），后续读操作（很可能发生）速度更快。

缓存是在大多数应用程序中以及在操作系统本身层面使用的一种标准技术。然而缓存技术并非完美，也有一些缺点。

尤其是在文件被用作程序之间数据交换的手段的情况下，延迟写入可能显著降低通信速度，并使得通信难以预测，由于缓冲区大小可能很大，因此可能会根据某些算法调整转储到磁盘的频率。

例如，在 MetaTrader 5 中，有一整类的 MQL 程序用于将交易信号从终端的一个实例拷贝到另一个实例。它们趋向于使用文件传输信息，对于其信息传输来说，不能让缓存降低速度，这很重要。为此，MQL5 提供了 **FileFlush** 函数。

void FileFlush(int handle)

该函数将具有 *handle* 描述符的文件的 I/O 文件缓冲区中剩余的所有数据强制刷写到磁盘。

如果不使用此函数，在最糟糕的情况下，从程序“发送”的数据部分可能只有在文件关闭后才写入磁盘。

这一特性可进一步保障发生非预见性事件（如操作系统或程序死机）时重要数据的安全。然而，另一方面，也不建议在批量记录期间频繁调用 **FileFlush**，因为可能会对性能造成不利影响。

如果文件以混合模式（读写同时进行）打开，则在向文件进行读和写之间必须调用 **FileFlush** 函数。

举个例子，考虑 **FileFlush.mq5** 脚本，其中我们实现模拟交易拷贝器（deal copier）的操作的两种模式。我们将需要在不同图表上运行该脚本的两个实例，其中一个成为数据发送者，另一个成为接收者。

该脚本有两个输入参数：**EnableFlashing** 允许你在使用和不使用 **FileFlush** 函数的情况下比较程序的操作；**UseCommonFolder** 表明需要创建一个可供选择的文件，以作为数据传输手段：在终端的当前实例的文件夹中，或者在共享文件夹中（在后者情况下，你可以在不同终端之间测试数据传输）。

```
#property script_show_inputs
input bool EnableFlashing = false;
input bool UseCommonFolder = false;
```

别忘了，要在启动脚本时出现一个带有输入变量的对话框，必须还要设置 **script_show_inputs** 特性。

中转文件（transit file）的名称在 **dataport** 变量中指定。**UseCommonFolder** 选项控制 **FILE_COMMON** 标志，该标志已被添加到 **File Open** 函数中打开文件的模式开关集合中。

```
const string dataport = "MQL5Book/dataport";
const int flag = UseCommonFolder ? FILE_COMMON : 0;
```

主函数 **OnStart** 实际上由两个部分组成：打开文件的设置，定期发送或接收数据的循环。

我们将需要运行该脚本的两个实例，每个实例都具有自己的文件描述符，虽然文件描述符指向的是磁盘上的同一个文件，但打开文件的模式不同。

```
void OnStart()
{
    bool modeWriter = true; // by default the script should write data
    int count = 0;          // number of writes/reads made
    // create a new or reset the old file in read mode, as a "sender"
    int handle = PRTF(FileOpen(dataport,
        FILE_BIN | FILE_WRITE | FILE_SHARE_READ | flag));
    // if writing is not possible, most likely another instance of the script is already
    // so we try to open it for reading
    if(handle == INVALID_HANDLE)
    {
        // if it is possible to open the file for reading, we will continue to work as
        handle = PRTF(FileOpen(dataport,
            FILE_BIN | FILE_READ | FILE_SHARE_WRITE | FILE_SHARE_READ | flag));
        if(handle == INVALID_HANDLE)
        {
            Print("Can't open file"); // something is wrong
            return;
        }
        modeWriter = false; // switch model/role
    }
}
```

一开始，我们尝试以 `FILE_WRITE` 模式在不共享写权限 (`FILE_SHARE_WRITE`) 的情况下打开该文件，以便运行脚本的第一个实例将捕获该文件并防止第二个实例以写模式处理该文件。第二个实例将在第一次调用 `FileOpen` 后出现错误和 `INVALID_HANDLE`，并将使用 `FILE_SHARE_WRITE` 并行写入标志第二次调用 `FileOpen` 尝试以读模式 (`FILE_READ`) 打开该文件。理想情况下，这应该没有问题。然后，`modeWriter` 变量将被设置为 `false` 以指示脚本的实际角色。

主要操作循环具有以下结构体：

```
while(!IsStopped())
{
    if(modeWriter)
    {
        // ...write test data
    }
    else
    {
        // ...read test data
    }
    Sleep(5000);
}
```

该循环会一直执行，直至用户将脚本从图表手动删除：这种状态由 `IsStopped` 函数发出信号。在循环内，该操作通过调用 `Sleep` 函数每 5 秒触发一次，该函数以指定的毫秒数（在本例中为 5000）“冻结”程序。这样做是为了更易于分析持续变化，避免过于频繁的状态日志。在没有详细日志的真实程序中，你可以按照 100 毫秒的频率或更高频率发送数据。

发送的数据将包括当前时间（一个 *datetime* 值，8 个字节）。在写入文件的 *if(modeWriter)* 指令的第一个分支中，我们以最后计数调用 *FileWriteLong*（获取自函数 *TimeLocal*），将操作计数器加 1 (*count++*)，并将当前状态输出到日志。

```
long temp = TimeLocal(); // get the current local time datetime
FileWriteLong(handle, temp); // append it to the file (every 5 seconds)
count++;
if(EnableFlashing)
{
    FileFlush(handle);
}
Print(StringFormat("Written[%d]: %I64d", count, temp));
```

务必要注意，仅当输入参数 *EnableFlashing* 设置为 *true* 时，才会在每次输入之后调用 *FileFlush* 函数。

在进行数据读取的 *if* 运算符的第二个分支中，我们首先通过调用 *ResetLastError* 来重置内部错误标志。这是必要的，因为只要文件中有任何数据，我们就会一直读取。在没有更多数据需要读取时，程序将出现特定错误代码 5015 (ERR_FILE_READERROR)。

由于内置 MQL5 计时器（包括 *Sleep* 函数）的精度有限（约 10 ms），我们不能排除在两次连续读取文件尝试期间发生了两次连续写入的情况。例如，一次读取发生在 10:00:00'200，第二次读取发生在 10:00:05'210（以记数法 "*hours:minutes:seconds'milliseconds*" 表示）。在此情况下，两个记录并行发生：一个是在 10:00:00'205，第二个是在 10:00:05'205，两个均处于上述期间内。此类情况虽然概率极小，但也有可能发生。即使时间间隔绝对精确，但如果程序总数过多并且没有足够的处理器核心处理它们，则 MQL5 运行时系统也可能被迫在两个运行脚本之间作出选择（哪个优先调用）。

MQL5 提供 **高精度计时器**（精确到微秒），但这对于当前任务来说并不是最重要的。

之所以需要嵌套循环，还有一个原因。一旦脚本作为数据“接收者”启动，其必须立即处理自从“发送者”启动以来已经累积的来自文件的所有记录（不太可能两个脚本同时启动）。或许有人会倾向采用不同的算法：跳过所有“旧”记录而仅跟踪新记录。这可以做到，但在这里实施了“无损”选项。

```

ResetLastError();
while(true)// loop as long as there is data and no problems
{
    bool reportedEndBeforeRead = FileIsEnding(handle);
    ulong reportedTellBeforeRead = FileTell(handle);

    temp = FileReadLong(handle);
    // if there is no more data, we will get an error 5015 (ERR_FILE_READERRC
    if(_LastError)break; // exit the loop on any error

    // here the data is received without errors
    count++;
    Print(StringFormat("Read[%d]: %I64d\t"
        "(size=%I64d, before=%I64d(%s), after=%I64d)",
        count, temp,
        FileSize(handle), reportedTellBeforeRead,
        (string)reportedEndBeforeRead, FileTell(handle)));
}

```

请注意以下这点。有关打开进行读取的文件的元数据（比如 **FileSize** 函数返回的文件大小，参见[获取文件属性](#)）在文件被打开后不会更改。如果另一个程序之后向我们打开进行读取的文件中添加了内容，即使我们为读描述符调用 **FileFlash**，该文件的“可检测”长度也将不会更新。可以关闭并重新打开文件（在每次读取之前执行，但效率不高）来显示新描述符对应的文件新长度。但我们将借助另一种技巧避免这个操作：

这个技巧就是持续使用读取函数（即 **FileReadLong**）读取数据，直至其返回错误。重要的是，不使用操作元数据的其它函数。尤其是，由于只读性的文件尾保持不变，因此使用 **FileIsEnding** 函数进行检查（参见[文件内的位置控制](#)）将会对旧位置给出 **true** 结果，尽管另一个进程可能在文件中追加了内容。此外，试图将内部文件指针移动到末尾 **FileSeek(handle, 0, SEEK_END)**，关于 **FileSeek** 函数，参见同一[章节](#)）将不会跳转到数据的实际末尾，而是跳转到打开文件时末尾位置（实际上已过期）。

该函数告诉我们在文件 **FileTell** 内的真实位置（参见同一[章节](#)）。随着信息从脚本的另一个实例添加到文件并在该循环中读取，指针将不断向右移动，最终超过 **FileSize**（尽管这很奇怪）。为了直观说明指针如何移动超出文件大小，我们保存在调用 **FileReadLong** 之前和之后的值，然后将这些值与大小一起输出到日志。

一旦使用 **FileReadLong** 读取产生任何错误，则内部循环将中断。正常循环退出意味着错误 5015 (ERR_FILE_READERROR)。尤其是当在文件中当前位置没有数据可供读取时就会发生这一错误。

最后成功读取的数据被输出到日志，可以轻松将其与发送者脚本输出到日志的内容进行比较。

我们运行新脚本两次。为区分其副本，我们将在不同金融工具的图表上运行。

运行两个脚本时，重要的是注意 **UseCommonFolder** 参数的值相同。在我们的测试中，将其设置为等于 **false**，因为我们将在一个终端上进行全部操作。对于独立测试，建议为不同终端之间的数据传输将 **UseCommonFolder** 设置为 **true**。

首先，我们在 EURUSD,H1 图表上运行第一个实例，保留所有默认设置，包括 **EnableFlashing=false**。然后，我们将在 XAUUSD,H1 图表上运行第二个实例（同样使用默认设置）。日志内容如下（你看到的时间将会不同）：

```

(EURUSD,H1) *
(EURUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=1 / ok
(EURUSD,H1) Written[1]: 1629652995
(XAUUSD,H1) *
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(EURUSD,H1) Written[2]: 1629653000
(EURUSD,H1) Written[3]: 1629653005
(EURUSD,H1) Written[4]: 1629653010
(EURUSD,H1) Written[5]: 1629653015

```

根据包含 “Written” 单词的行以及递增的值，发送者成功打开文件进行写入，并开始每 5 秒发送数据。在发送者启动后不到 5 秒内，接收者也启动。接收者给出错误消息，因为它不能打开文件进行写入。但随后接收者成功打开文件进行写入。然而，没有记录表明接收者能够在文件找到传输的数据。数据仍然“挂”在发送者的缓存中。

我们停止两个脚本，然后以相同顺序再次运行它们：我们首先在 EURUSD 上运行发送者，然后在 XAUUSD 上运行接收者。但这次我们将为发送者指定 *EnableFlashing=true*。

日志中的内容如下：

```

(EURUSD,H1) *
(EURUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=1 / ok
(EURUSD,H1) Written[1]: 1629653638
(XAUUSD,H1) *
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(XAUUSD,H1) Read[1]: 1629653638 (size=8, before=0(false), after=8)
(EURUSD,H1) Written[2]: 1629653643
(XAUUSD,H1) Read[2]: 1629653643 (size=8, before=8(true), after=16)
(EURUSD,H1) Written[3]: 1629653648
(XAUUSD,H1) Read[3]: 1629653648 (size=8, before=16(true), after=24)
(EURUSD,H1) Written[4]: 1629653653
(XAUUSD,H1) Read[4]: 1629653653 (size=8, before=24(true), after=32)
(EURUSD,H1) Written[5]: 1629653658

```

同一文件再次以不同模式在两个脚本中成功打开，但这次写入的值可被接收者正常读取。

需要注意的是，除了首次读取外，每次后续读取数据前，*FileIsEnding* 函数返回 *true*（显示在与接收的数据相同的字符串中，在 “before” 字符串之后的圆括号中）。因此，这表明我们已经处于文本末尾，但随后 *FileReadLong* 成功读取一个应在文件限值范围以外的值，并将位置向右偏移。例如，条目 “size=8, before=8(true), after=16” 表示报告给 MQL 程序的文件大小是 8，而在调用 *FileReadLong* 之前的当前指针也等于 8，并且启用了文件尾符号。在成功调用 *FileReadLong* 之后，指针移动到 16。然而，在下一次以及所有其它迭代时，我们再次看到 “size=8”，并且指针逐渐移动越来越远，超出文件范围。

由于在发送者的写入以及在接收者的读取都是每 5 秒发生一次，根据它们的循环偏移阶段，我们可以观察到这两个操作之间的不同延迟（最糟糕的情况延迟达将近 5 秒）的影响。然而，这并不意味着缓存刷写也是如此之慢。事实上，这一操作几乎是瞬间完成。为确保更快速的变化检测，可以缩短循环中的睡眠期（请注意，如果延迟过短，本测试将快速填满日志 - 不同于真实程序，在这里新数据的生成始终是基于发送者精确至秒的当前时间）。

顺便说一句，你可以运行多个接收者，而相比之下，发送者必须只有一个。下面的日志显示了一个发送者在 EURUSD 图表上的操作，以及两个接收者在 XAUUSD 和 USDRUB 图表上的操作。


```

(EURUSD,H1) *
(EURUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=1 / ok
(EURUSD,H1) Written[1]: 1629671658
(XAUUSD,H1) *
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(XAUUSD,H1) Read[1]: 1629671658 (size=8, before=0(false), after=8)
(EURUSD,H1) Written[2]: 1629671663
(USDRUB,H1) *
(USDRUB,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(USDRUB,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(USDRUB,H1) Read[1]: 1629671658 (size=16, before=0(false), after=8)
(USDRUB,H1) Read[2]: 1629671663 (size=16, before=8(false), after=16)
(XAUUSD,H1) Read[2]: 1629671663 (size=8, before=8(true), after=16)
(EURUSD,H1) Written[3]: 1629671668
(USDRUB,H1) Read[3]: 1629671668 (size=16, before=16(true), after=24)
(XAUUSD,H1) Read[3]: 1629671668 (size=8, before=16(true), after=24)
(EURUSD,H1) Written[4]: 1629671673
(USDRUB,H1) Read[4]: 1629671673 (size=16, before=24(true), after=32)
(XAUUSD,H1) Read[4]: 1629671673 (size=8, before=24(true), after=32)
(EURUSD,H1) Written[5]: 1629671678

```

当 USDRUB 上的第三个脚本启动时，在文件中已经有 2 条长度 8 字节的记录，因此内部循环立即从 *FileReadLong* 执行 2 次迭代，文件大小“似乎”等于 16。

4.5.13 删除文件并检查文件是否存在

检查文件是否存在并删除它，这是与文件系统（即文件所依赖的外部环境）相关的关键操作。目前为止，我们已经了解了操作文件内部内容的函数。从本节开始，重点将转向将文件作为不可分割的单元来管理的函数。

bool FileExists(const string filename, int flag = 0)

该函数检查名称为 *filename* 的文件是否存在，如果存在，则返回 *true*。使用 *flag* 参数选择搜索目录：如果该参数为 0（默认值），则在当前终端实例的目录（*MT5/Files*）中搜索文件；如果 *flag* 等于 `FILE_COMMON`，则检查所有终端的公用目录 *Users/<user>...MetaQuotes/Terminal/Common/Files*。如果 MQL 程序在测试程序中运行，则工作目录位于测试程序代理文件夹内（*Tester/<agent>/MT5/Files*），参见 [处理文件](#) 一章的引言部分。

指定的名称可能不属于文件，而是属于目录。在此情况下，*FileExists* 函数将返回 *false*，并且错误代码 5018 (FILE_IS_DIRECTORY) 将被记录到 *_LastError* 变量中。

bool FileDelete(const string filename, int flag = 0)

该函数删除具有指定名称 *filename* 的文件。*flag* 参数指定该文件的位置。若使用默认值，删除操作是在当前终端实例的工作目录（*MT5/Files*）中执行，如果程序在测试程序中运行，则在测试程序代理的目录（*Tester/<agent>/MT5/Files*）中执行。如果 *flag* 等于 `FILE_COMMON`，则文件必须位于所有终端的公用文件夹内（*/Terminal/Common/Files*）。

该函数返回成功 (*true*) 或出错 (*false*) 标志。

该函数不允许删除目录。要删除目录，请使用 *FolderDelete* 函数（参见 [处理文件夹](#)）。

为了解描述的函数如何工作，我们将使用 *FileExist.mq5* 脚本。我们将对一个临时文件进行若干操作。

```
const string filetemp = "MQL5Book/temp";
void OnStart()
{
    PRTF(FileIsExist(filetemp)); // false / FILE_NOT_EXIST(5019)
    PRTF(FileDelete(filetemp)); // false / FILE_NOT_EXIST(5019)

    int handle = PRTF(FileOpen(filetemp, FILE_TXT | FILE_WRITE | FILE_ANSI)); // 1

    PRTF(FileIsExist(filetemp)); // true
    PRTF(FileDelete(filetemp)); // false / CANNOT_DELETE_FILE(5006)

    FileClose(handle);

    PRTF(FileIsExist(filetemp)); // true
    PRTF(FileDelete(filetemp)); // true
    PRTF(FileIsExist(filetemp)); // false / FILE_NOT_EXIST(5019)

    PRTF(FileIsExist("MQL5Book")); // false / FILE_IS_DIRECTORY(5018)
    PRTF(FileDelete("MQL5Book")); // false / FILE_IS_DIRECTORY(5018)
}
```

该文件最初不存在，因此 *FileIsExist* 和 *FileDelete* 函数均返回 *false*，错误代码为 5019 (FILE_NOT_EXIST)。

然后我们创建一个文件，则 *FileIsExist* 函数报告该文件存在。然而，不能将其删除，因为它被我们的进程打开且占用 (错误代码 5006, CANNOT_DELETE_FILE)。

在文件关闭后才能被删除。

在脚本末尾，检查了“MQL5Book”目录并尝试删除它。*FileIsExist* 返回 *false*，因为它不是一个文件，但错误代码 5018 (FILE_IS_DIRECTORY) 表明它是一个目录。

4.5.14 复制和移动文件

系统层级的主要文件操作是复制和移动。为此，MQL5 实现了具有相同原型的两个函数。

bool FileCopy(const string source, int flag, const string destination, int mode)

该函数将 *source* 文件复制到 *destination* 文件。上述参数既可以只包含文件名，也可以同时包含文件名和 MQL5 沙箱中的路径前缀 (文件夹层级结构)。*flag* 和 *mode* 参数确定要搜索源文件的工作文件夹以及目标工作文件夹: 0 表示当前终端的本地当前实例的文件夹 (如果程序在测试程序中运行，则为测试程序代理)，而值 FILE_COMMON 表示所有终端的公用文件夹。

此外，在 *mode* 参数中，可以选择性指定 FILE_REWRITE 常量 (如果需要将 FILE_REWRITE 和 FILE_COMMON 组合，则使用逐位运算符 OR (|) 完成)。如果没有 FILE_REWRITE，则禁止对现有文件进行复制。换言之，如果路径和名称与 *destination* 参数中指定值均相符的文件已经存在，则必须通过设置 FILE_REWRITE 来确认你要覆写文件。否则函数调用将会失败。

如果成功完成则该函数返回 *true*，如果出错，则返回 *false*。

如果源文件或目标文件被另一个进程占用 (打开)，则复制可能失败。

复制文件时，通常会保存它们的元数据（创建时间、访问权限、替代数据流）。如果需要仅对文件本身的数据执行“纯”复制，可以使用连续调用 *FileLoad* 和 *FileSave*，参见 [简化模式下的文件读写](#)。

bool FileMove(const string source, int flag, const string destination, int mode)

该函数可移动或重命名一个文件。源路径和名称在 *source* 参数中指定，而目标路径和名称在 *destination* 中指定。

参数列表及其操作原理与 *FileCopy* 函数相同。大体来说，*FileMove* 与 *FileCopy* 的功能类似，但前者在成功复制后还会删除了原始文件。

我们使用 *FileCopy.mq5* 脚本来学习如何实际使用这些函数。该脚本有两个文件名变量。当脚本运行时，两个文件均不存在。

```
const string source = "MQL5Book/source";
const string destination = "MQL5Book/destination";
```

在 *OnStart* 中，我们根据一个简单场景执行一系列操作。首先，我们尝试将 *source* 文件从本地工作目录复制到总目录中的 *destination* 文件。如预期的那样，我们得到 *false*，并且 *_LastError* 中的错误代码将为 5019 (FILE_NOT_EXIST)。

```
void OnStart()
{
    PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // false / FILE_NOT_EXIST(501
    ...
}
```

因此，我们将以常规方式创建一个源文件，写入一些数据，并将其刷写到磁盘。

```
int handle = PRTF(FileOpen(source, FILE_TXT | FILE_WRITE)); // 1
PRTF(FileWriteString(handle, "Test Text\n")); // 22
FileFlush(handle);
```

由于该文件仍然处于打开中，并且在打开时未指定 FILE_SHARE_READ 权限，以其它方式（绕过句柄）访问该文件仍然被阻止。因此，下一次复制尝试将再次失败。

```
PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // false / CANNOT_OPEN_FILE(5
```

我们来关闭文件并重试。但首先我们将生成文件的特性输出到日志：其创建时间和修改时间。两个特性均将包含你的计算机的当前时间戳。

```
FileClose(handle);
PRTF(FileGetInteger(source, FILE_CREATE_DATE)); // 1629757115, example
PRTF(FileGetInteger(source, FILE_MODIFY_DATE)); // 1629757115, example
```

我们等待 3 秒，然后调用 *FileCopy*。这样你可以看到原始文件与其副本在特性方面的差异。这一暂停与之前对该文件的锁定无关：我们可以在关闭文件后立即复制，或者如果启用了 FILE_SHARE_READ 选项，甚至在写入它的同时进行复制。

```
Sleep(3000);
```

我们复制文件。这次操作成功了。我们看看副本的特性。

```
PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // true
PRTF(FileGetInteger(destination, FILE_CREATE_DATE, true)); // 1629757118, +3 second
PRTF(FileGetInteger(destination, FILE_MODIFY_DATE, true)); // 1629757115, example
```

每个文件有自己的创建时间 (对于副本, 创建时间比原文件迟 3 秒钟), 但修改时间相同 (副本继承了原始文件的特性)。

现在我们尝试将副本移动回本地文件夹。没有 FILE_REWRITE 选项便无法完成移动, 因为没有覆写原始文件的权限。

```
PRTF(FileMove(destination, FILE_COMMON, source, 0)); // false / FILE_CANNOT_REWRITE
```

通过更改该参数的值, 我们将成功实现文件传输。

```
PRTF(FileMove(destination, FILE_COMMON, source, FILE_REWRITE)); // true
```

最后, 原始文件被删除, 为此脚本的新测试留下一个干净的环境。

```
...
FileDelete(source);
}
```

4.5.15 搜索文件和文件夹

MQ15 允许你在终端沙盒、测试程序代理以及所有终端的公用沙盒中搜索文件和文件夹 (有关沙盒的详细信息, 参见章节介绍 [处理文件](#))。如果你准确知道所需文件/目录的名称和位置, 则使用 *FilesExist* 函数。

long FileFindFirst(const string filter, string &found, int flag = 0)

该函数根据传递的筛选器开始搜索文件和文件夹。筛选器可包含在沙盒中的子文件夹组成的路径, 且必须包含要搜索的文件系统元素的确切名称或名称模式。*filter* 参数不能为空。

模板是包含一个或多个通配符字符的字符串。有两种这样的字符: 星号 (*) 可替换任何数量的任何字符 (包括零), 而问号 (?) 可替换一个任意字符。例如, 筛选器 "*" 将查找所有文件和文件夹, 而 "???.*" 将仅查找名称长度不超过 3 个字符的文件和文件夹, 可以选择包含扩展名。扩展名为 "csv" 的文件可通过 "*.csv" 筛选器查找 (但需注意, 文件夹也可能有扩展名)。筛选器 "*" 查找没有扩展名的元素, 而 ".*" 查找没有名称的元素。然而, 对此应记住以下两点。

在很多版本的 Windows 中, 为文件系统元素生成两种名称: 长名称 (默认名称, 最多 260 个字符) 和短名称 (继承自 MS-DOS 的 8.3 格式)。如果长名称超过 8 个字符或扩展名长于 3, 则自动根据长名称生成第二种短名称。如果没有软件使用短名称, 则可以在系统上禁用短名称生成, 但通常情况下是启用的。

搜索文件时会同时搜索这两种类型的名称, 这就是为什么返回的列表乍一看可能包含非预期元素的原因。尤其是由系统根据长名称生成的短名称始终在点号前面包含一个初始部分, 最多长 8 个字符。可能偶尔会找到具有期望模式的匹配项。

如果你需要查找具有若干个扩展名的文件, 或名称中包含无法用单一模式概括的不同片段的文件, 则必须以不同的设置重复若干次搜索过程。

搜索仅在特定文件夹中执行 (如果筛选器中没有路径, 则在根文件夹或在沙盒中, 如果筛选器包含路径, 则在指定文件夹中), 不包括子目录。

搜索不区分大小写。例如, 请求搜索 "*.txt" 文件也将返回扩展名 "TXT"、"Txt" 之类的文件。

如果找到具有匹配名称的文件或文件夹, 则该名称被放在输出参数 *found* 中 (需要一个变量, 因为结果通过引用传递), 并且函数返回一个搜索句柄: 这将需要被传递给 *FileFindNext* 函数以继续对匹配项迭代 (如果有很多匹配项)。

在 *found* 参数中, 仅返回名称和扩展名, 不会返回筛选器中可能已经指定的路径 (文件夹层次结构)。

如果找到的项目是一个文件夹, 会在其名称右侧追加 `\` (反斜杠) 字符。

flag 参数允许在当前终端副本的本地工作文件夹 (值 0) 或所有终端的公用文件夹 (值 `FILE_COMMON`) 之间选择搜索区域。在测试程序中执行 MQL 程序时, 其本地沙盒 (0) 位于测试程序代理目录中。

搜索过程完成后, 应通过调用 *FileFindClose* (参见下文) 来释放收到的句柄。

`bool FileFindNext(long handle, string &found)`

该函数继续以 *FileFindFirst* 函数开始搜索文件系统的适当元素。第一个参数是从 *FileFindFirst* 接收到的描述符, 据此应用所有先前搜索条件。

如果找到下一个元素, 则其名称通过自变量 *found* 传递给调用代码, 然后函数返回 *true*。

如果没有更多元素, 则函数返回 *false*。

`void FileFindClose(long handle)`

该函数可关闭因调用 *FileFindFirst* 而接收到的搜索描述符。

搜索过程完成后, 必须调用该函数以释放系统资源。

举个例子, 我们看看脚本 *FileFind.mq5*。在前面的章节中, 我们测试了用于在 *MQL5/Files/MQL5Book* 目录中创建文件的很多其它脚本。请求所有这种文件的列表。

```
void OnStart()
{
    string found; // receiving variable
    // start searching and get descriptor
    long handle = PRTF(FileFindFirst("MQL5Book/*", found));
    if(handle != INVALID_HANDLE)
    {
        do
        {
            Print(found);
        }
        while(FileFindNext(handle, found));
        FileFindClose(handle);
    }
}
```

即使你已清除该目录, 你也仍然可以将随本书提供的各种编码的样本文件复制到其中。因此脚本 *FileFind.mq5* 应至少输出以下列表 (枚举顺序可能有变化):

```

ansi1252.txt
unicode1.txt
unicode2.txt
unicode3.txt
utf8.txt

```

为简化搜索过程，该脚本包含了辅助函数 *DirList*。这个辅助函数包含对内置函数的所有必要调用，还有一个用于构建具有匹配筛选器的元素列表的字符串数组的循环。

```

bool DirList(const string filter, string &result[], bool common = false)
{
    string found[1];
    long handle = FileFindFirst(filter, found[0]);
    if(handle == INVALID_HANDLE) return false;
    do
    {
        if(ArrayCopy(result, found, ArraySize(result)) != 1) break;
    }
    while(FileFindNext(handle, found[0]));
    FileFindClose(handle);

    return true;
}

```

我们利用该辅助函数来请求本地沙盒中目录的列表。为此，我们假定目录通常没有扩展名（理论上，并非总是如此，因此，如果程序员希望，应实施更严格的子文件夹列表请求）。无扩展名元素筛选器是 “*”（你可以使用 Windows shell 命令 *dir* “dir *” 进行检查）。但是，该模板在 MQL5 函数中会导致错误 5002 (WRONG_FILENAME)。因此，我们将指定一个更“模糊”的模板 “*.*”：它表示没有扩展名或扩展名为 1 个字符的元素。

```

void OnStart()
{
    ...
    string list[];
    // try to request elements without extension
    // (works on the Windows command line)
    PRTF(DirList("*. ", list)); // false / WRONG_FILENAME(5002)

    // expand the condition: the extension must be no more than 1 character
    if(DirList("*.?", list))
    {
        ArrayPrint(list);
        // example: "MQL5Book\" "Tester\"
    }
}

```

在我的 MetaTrader 5 实例中，脚本找到两个文件夹 “MQL5Book\” 和 “Tester\”。如果你运行过先前的测试脚本，你应该也会有第一个文件夹。

4.5.16 处理文件夹

要是有一个文件系统不能够通过任意式目录层次结构（逻辑相关文件集容器）来结构化存储的信息，那简直是不可思议。在 MQL5 层面，也支持该功能。如果必要，我们可以使用内置函数 **FolderCreate**、**FolderClean** 以及 **FolderDelete** 来创建、清理和删除文件夹。

早前我们已经了解一种创建文件夹的方法，并且或许不只是创建一个文件夹，而是一次性创建要求的子文件夹的整个层次结构。为此，当使用 **FileOpen** 创建（打开）文件，或者复制文件（**FileCopy**、**FileMove**）时，不仅要指定名称，还要在名称前加上要求的路径。例如，

```
FileCopy("MQL5Book/unicode1.txt", 0, "ABC/DEF/code.txt", 0);
```

该语句将在沙盒中创建“ABC”文件夹，在该文件夹中创建“DEF”文件夹，并以新名称复制其中的文件（源文件必须存在）。

如果不想事先创建一个源文件，可以临时创建一个虚拟文件：

```
uchar dummy[];
FileSave("ABC/DEF/empty", dummy);
```

我们将获得与前面示例相同的文件夹结构，但获得一个零大小的“空”文件。

利用此类方法，文件夹的创建一定程度上成为处理文件的副产品。然而，有时候需要将文件夹作为独立实体进行操作且不产生附带影响，尤其是仅创建一个空文件夹。**FolderCreate** 函数可实现这一功能。

bool FolderCreate(const string folder, int flag = 0)

该函数创建一个名为 *folder* 的文件夹，其中可包括一个路径（若干顶层文件夹名称）。在任一情况下，单个文件夹或文件夹结构在由 *flag* 参数定义的沙盒中创建。默认情况下，当 *flag* 为 0 时，使用终端或测试程序代理（如果程序在测试程序中运行）的本地工作文件夹 **MQL5/Files**。如果 *flag* 等于 **FILE_COMMON**，则使用所有终端的共享文件夹。

如果成功或者如果文件夹已经存在，则函数返回 **true**。如果出错，则结果为 **false**。

bool FolderClean(const string folder, int flag = 0)

该函数删除在指定 *folder* 目录中的所有文件以及任何嵌套层级的文件夹（以及所有内容）。*flag* 参数指定进行操作的沙盒（本地或全局）。

使用该功能要小心，因为所有文件和子文件夹（及其文件）将被永久删除。

bool FolderDelete(const string folder, int flag = 0)

该函数删除指定文件夹（*folder*）。在调用该函数之前，文件夹必须为空，否则无法将其删除。

FileFolder.mq5 脚本中演示了这三个函数的使用技巧。你可以使用调试模式分步（逐语句）执行该脚本，并在文件管理器中观察文件夹和文件如何出现和消失。然而，请注意，在执行下一指令之前，你应使用文件管理器将创建的文件夹退出到“MQL5Book”层级，否则文件夹可能被文件管理器占用，这将会中断脚本运行。

我们首先创建若干子文件夹，作为向其中写入空虚拟文件时生成的附带内容。

```
void OnStart()
{
    const string filename = "MQL5Book/ABC/DEF/dummy";
    uchar dummy[];
```

```
PRTF(FileSave(filename, dummy)); // true
```

接下来，我们使用 **FolderCreate** 在底层嵌套层级创建另一个文件夹：这次该文件夹仅显示其本身，没有辅助文件。

```
PRTF(FolderCreate("MQL5Book/ABC/GHI")); // true
```

如果你试图删除“DEF”文件夹，将会失败，因为它不为空（其中有一个文件）。

```
PRTF(FolderDelete("MQL5Book/ABC/DEF")); // false / CANNOT_DELETE_DIRECTORY(5024)
```

要想将其移除，必须首先清理，最简便的方法是使用 **FolderClean**。但我们将尝试模拟这样一个常见情况：要清理的文件夹中的某些文件可能被其它 MQL 程序、外部应用程序或终端本身锁定。我们打开要读取的文件，然后调用 **FolderClean**。

```
int handle = PRTF(FileOpen(filename, FILE_READ)); // 1
PRTF(FolderClean("MQL5Book/ABC")); // false / CANNOT_CLEAN_DIRECTORY(5025)
```

该函数返回 **false** 并出现错误代码 5025 (CANNOT_CLEAN_DIRECTORY)。我们关闭文件后，对整个文件夹结构的清理和删除将会成功。

```
FileClose(handle);
PRTF(FolderClean("MQL5Book/ABC")); // true
PRTF(FolderDelete("MQL5Book/ABC")); // true
}
```

使用共享终端目录时更有可能出现潜在锁定情况，因为在这种目录下，同一个文件或文件夹可能会被不同的程序实例“占用”。但即使是在本地沙盒中，也别忘了可能的冲突（例如，如果 CSV 文件在 Excel 中打开）。应对处理文件夹的代码部分实现详细的诊断和错误输出功能，以使用户能够注意到问题并修复问题。

4.5.17 文件或文件夹选择对话框

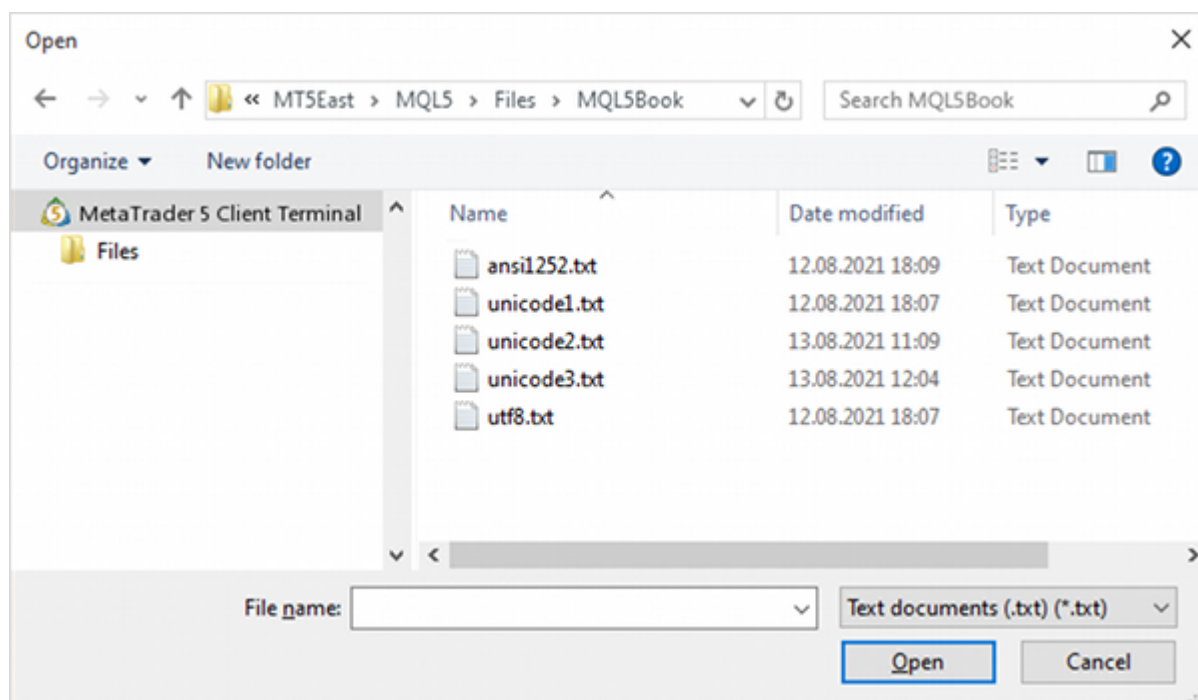
在处理文件和文件夹的函数组中，有一个函数允许从用户交互式请求文件或文件夹的名称，以及一组文件的名称，以便将该信息传递给一个 MQL 程序。调用 **FileSelectDialog** 函数可以使一个用于选择文件和文件夹的标准 Windows 窗口出现在终端中。

由于该对话框中断 MQL 程序的执行直至其关闭，该函数仅在以单独线程执行的两种类型的 MQL 程序中允许调用：EA 和脚本（参见 [MQL 程序类型](#)）。在指标和服务中禁止使用该函数：前者在终端的接口线程中执行（阻止它们将会冻结对应金融工具图表的更新），而后者在后台执行，不能访问用户接口。

该函数处理的文件系统的所有元素均位于沙盒中，即位于当前终端副本或测试程序（如果程序在测试程序中运行）的目录中的 **MQL5/Files** 子文件夹中。

如果 FSD_COMMON_FOLDER 标志出现在 **flags** 参数中（参见下文），则使用所有终端的公用沙盒 **Users/<user>...MetaQuotes/Terminal/Common/Files**。

对话框的外观取决于 Windows 操作系统。以下显示了其中一种可能的界面选项。



Windows 文件和文件夹选择对话框

```
int FileSelectDialog(const string caption, const string initDir, const string filter,
    uint flags, string &filenames[], const string defaultName)
```

该函数显示一个用于打开或创建文件或选择文件夹的标准 Windows 对话框。标题在 *caption* 参数中指定。如果值为 NULL，则使用标准标题：“打开”（用于读取）、“另存为”（用于写入文件）或者“选择文件夹”，取决于 *flags* 参数中的标志。

initDir 参数允许你设置将为之打开对话框的初始文件夹。如果设为 NULL，则将显示 *MQL5/Files* 文件夹的内容。如果在 *initDir* 中指定了不存在的路径，则同样会使用上述文件夹。

使用 *filter* 参数可以限制将在对话框中显示的文件扩展名集合。其它格式的文件将被隐藏。NULL 表示无限制。

filter 字符串的格式如下：

```
"<description 1>|<extension 1>|<description 2>|<extension 2>..."
```

可以使用任何字符串作为 *description*。可以编写带替换字符 ‘%’ 和 ‘?’ 的任何筛选器（这两个替换字符在 [查找文件和文件夹](#) 中介绍过）作为 *extensions*。定界符为 ‘|’ 符号。

由于相邻说明和扩展名形成一个逻辑相关对，因此行中的元素总数必须为偶数，而定界符的数量必须为奇数。

每个说明和扩展名组合在对话框的下拉列表中生成一个单独的选择。向用户显示的是说明，而扩展名用于筛选。

例如，“文本文档 (*.txt)|*.txt|所有文件 (*.*)|*.*”，其中第一个扩展名“文本文档 (*.txt)|*.txt”将被选择为默认文件类型。

在 *flags* 参数中，你可以指示位掩码，使用 ‘|’ 运算符指定操作模式。为其定义以下常量：

- `FSD_WRITE_FILE` - 文件写入模式（“另存为”）。在没有该标志的情况下，默认使用读模式（“打开”）。如果存在该标志，则允许输入任意新名称，无论 `FSD_FILE_MUST_EXIST` 标志如何。
- `FSD_SELECT_FOLDER` - 文件夹选择模式（仅一个并且仅当存在时）。有了该标志，除了 `FSD_COMMON_FOLDER` 以外的所有其它标志均被忽略，或者会导致错误。不能显式请求创建文件夹，但可以在对话框中交互式创建一个文件夹并立即选择该文件夹。
- `FSD_ALLOW_MULTISELECT` - 允许在读模式下选择多个文件。如果指定了 `FSD_WRITE_FILE` 或 `FSD_SELECT_FOLDER`，则该标志被忽略。
- `FSD_FILE_MUST_EXIST` - 选择的文件必须存在。如果用户试图指定一个任意名称，则该对话框将显示一个警告并保持打开。如果指定了 `FSD_WRITE_FILE` 模式，则该标志被忽略。
- `FSD_COMMON_FOLDER` - 对话框为所有客户终端的公用沙盒打开。

该函数将以选定的文件或文件夹的名称填充 *filenames* 字符串数组。如果该数组为动态数组，则其大小更改以适配实际数据量，尤其是如果没有选择任何内容，则扩展或截断至 0。如果数组为 `fixed` 数组，则其必须足够大以容纳预期数据。否则将发生错误 4007 (`ARRAY_RESIZE_ERROR`)。

defaultName 参数指定默认文件/文件夹名称，在打开对话框后，该名称会立即代入对应的输入字段中。如果参数为 `NULL`，则该字段将初始为空。

如果设置了 **defaultName** 参数，则在 MQL 程序的非可视化测试期间，*FileSelectDialog* 调用将返回 1，而 **defaultName** 值本身将被拷贝到 *filenames* 数组。

该函数将返回选择的项目数（如果用户未选择任何项目，则返回 0），如果出错，则返回 -1。

参见 *FileSelect.mq5* 脚本以了解该函数用法示例。在 *OnStart* 函数中，我们将以不同设置依次调用 *FileSelectDialog*。只要用户选择了某项内容（未单击对话框中的“取消”按钮），则测试将继续直至最后一步（即使函数执行时出现错误代码）。

```
void OnStart()
{
    string filenames[]; // a dynamic array suitable for any call
    string fixed[1]; // too small array if there are more than 1 files
    const string filter = // filter example
        "Text documents (*.txt)|*.txt"
        "|Files with short names|????.*"
        "|All files (*.*)|*.*";
```

首先，我们将要求用户从“MQL5Book”文件夹中选择一个文件。你可以选择一个现有文件或输入一个新文件名（因为没有 `FSD_FILE_MUST_EXIST` 标志）。

```
Print("Open a file");
if(PRTF(FileSelectDialog(NULL, "MQL5book", filter,
    0, filenames, NULL)) == 0) return; // 1
ArrayPrint(filenames); // "MQL5Book\utf8.txt"
```

假设该文件夹包含至少 5 个随本书交付提供的文件，在这里选择其中一个。

然后我们将在“写入”模式下作出一个类似的请求（使用 `FSD_WRITE_FILE` 标志）。

```
Print("Save as a file");
if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
    FSD_WRITE_FILE, filenames, NULL)) == 0) return; // 1
ArrayPrint(filenames); // "MQL5Book\newfile"
```

其中，用户也将能够选择一个现有文件或输入一个新名称。编程人员必须检查用户是否将要覆盖一个现有文件（对话框不会生成警告）。

现在我们在一个动态数组中检查多个文件的选择（FSD_ALLOW_MULTISELECT）。

```
if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
    FSD_FILE_MUST_EXIST | FSD_ALLOW_MULTISELECT, filenames, NULL)) == 0) return; //
ArrayPrint(filenames);
// "MQL5Book\ansi1252.txt" "MQL5Book\unicode1.txt" "MQL5Book\unicode2.txt"
// "MQL5Book\unicode3.txt" "MQL5Book\utf8.txt"
```

存在 FSD_FILE_MUST_EXIST 标志则表示如果你试图输入一个新名称，则对话框将会显示警告并保持打开。

如果我们试图以类似方式在一个固定大小数组中选择超过一个文件，则将会出错。

```
Print("Open multiple files (fixed, choose more than 1 file for error)");
if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
    FSD_FILE_MUST_EXIST | FSD_ALLOW_MULTISELECT, fixed, NULL)) == 0) return;
// -1 / ARRAY_RESIZE_ERROR(4007)
ArrayPrint(fixed); // null
```

最后，我们检查文件夹操作（FSD_SELECT_FOLDER）。

```
Print("Select a folder");
if(PRTF(FileSelectDialog(NULL, "MQL5book/nonexistent", NULL,
    FSD_SELECT_FOLDER, filenames, NULL)) == 0) return; // 1
ArrayPrint(filenames); // "MQL5Book"
```

在此情况下，不存在的子文件夹“nonexistent”被指定为开始路径，因此对话框将在沙盒 *MQL5/Files* 的根目录中打开。在那里，我们选择“MQL5book”。

如果我们组合无效的标志组合，则我们会遇到另一个错误。

```
if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
    FSD_SELECT_FOLDER | FSD_WRITE_FILE, filenames, NULL)) == 0) return;
// -1 / INTERNAL_ERROR(4001)
ArrayPrint(filenames); // "MQL5Book"
}
```

由于错误，函数不会修改传递的数组，并且旧的“MQL5Book”元素仍然保留在其中。

在此测试中，我们故意仅检查结果是否为 0 的情况，以演示所有选项，无视错误的存在。在真实程序中，在考虑错误的情况下检查函数的结果，即考虑三个结果的条件：用户作出选择（>0）、用户未作出选择（=0）以及错误（<0）。

4.6 客户终端全局变量

在前一章中，我们学习了处理文件的 MQL5 函数。它们为读写任意数据提供了广泛、灵活的选项。然而，有时一个 MQL 程序需要以更简便的方式在运行之间保存和还原属性状态。

例如，我们想要计算某些统计信息：程序启动了多少次，在不同图表上并行执行了程序的多少个实例等等。这些信息无法在程序本身内部累计。必须有某种外部长期存储。为此创建一个文件也是可行的，只不过代价很高。

很多程序设计用于彼此交互，即它们必须以某种方式交换信息。如果我们说的是与终端外部的程序集成，或者传输大量数据，那么不使用文件便很难实现。但是，若无需发送大量数据，并且所有程序均是以 MQL5 编写并在 MetaTrader 5 中运行，则使用文件似乎是多余的。对于这种情况，终端提供了更为简单的技术：全局变量。

全局变量是在终端的共享内存中的一个指定位置。全局变量可以由任何 MQL 程序创建、修改或删除，但并不会被任何程序独占，并且可以由所有其它 MQL 程序使用。全局变量的名称是在所有变量中唯一的字符串，长度不超过 63 个字符。该字符串不必满足 MQL5 中变量标识符的要求，因为终端的全局变量不是常规意义上的变量。编程人员无需在源代码中根据我们在 [变量](#) 章节学习的语法定义这些全局变量，它们不是 MQL 程序的一个组成部分，对它们的任何操作都只能通过调用本章将介绍的特殊函数实现。

全局变量允许你仅存储 `double` 类型的值。如果必要，你可以将其它类型的值包装/转换为 `double`，或者使用变量名称的一部分（例如，遵循某个前缀）存储字符串。

当终端正在运行时，全局变量存储在 RAM 中，几乎即时可用：唯一开销与函数调用有关。相比使用文件，全局变量无疑更具有优势，因为在使用文件时，获取句柄是一个相对缓慢的过程，而句柄本身也会消耗某些额外资源。

在终端会话结束时，全局变量被卸载进入一个特殊文件 (`gvariables.dat`)，并在下次运行终端时从这个文件中还原。

一个特定全局变量如果在 4 个星期内未被申领使用，则会被终端自动销毁。此行为依赖于跟踪并存储变量的上次使用时间，其中“使用”指的是设置新值或读取旧值（但不是检查是否存在，也不会获取上次使用时间）。

请注意，全局变量并不与账户、数据图或交易环境的任何其它特性绑定。因此，如果要将它们用于存储与环境相关的内容（例如特定账户的某些一般限制），则变量名称的构建应考虑影响算法和决定的所有因素。要区分同一 EA 交易的多个实例的全局变量，你可能需要通过 EA 设置向名称添加现有交易品种、时间范围或“魔法数字”。

除了 MQL 程序，全局变量也可由用户手动创建。现有全局变量的列表以及它们的交互式管理方式可在通过 **Tools -> Global Variables** 命令 (F3) 在终端中打开的对话框中找到。

使用对应按钮，你可以 **Add** 和 **Delete** 全局变量，还可以在 **Variable** 或 **Meaning** 列中双击以编辑特定变量的名称或值。可从键盘使用以下热键：F2 用于名称编辑，F3 用于值编辑，Ins 用于添加新变量，Del 用于删除选定变量。

稍后我们将了解两个主要类型的 MQL 程序：“EA 交易”和“指标”。它们的特殊功能是能够在测试程序中运行，用于全局变量的函数也同样有效。然而，全局变量由测试程序代理在测试程序中创建、存储和管理。换言之，终端全局变量列表在测试程序中不可用，由受测试程序创建的变量属于特定代理，它们的生命周期仅限于一次测试传递。也就是说，该代理的全局变量对其它代理不可见，并且在测试运行结束时将被移除。尤其是如果 EA 针对若干个代理进行了 [优化](#)，则它可操纵全局变量，从而与它在同一代理上下文中使用的指标通信，因为它们在该上下文中一起执行，对于并行代理，EA 的其它副本将形成它们自己的变量列表。

使用全局变量进行的 MQL 程序之间的数据交换不是唯一可用的方法，也并非始终是最适当的方法。尤其是 EA 和指标是交互型 MQL 程序的情况下，此时它们可以生成和接受 [图表上事件](#)。你可以在事件参数中传递各种类型的信息。此外，计算数据数组能以 [指标缓冲区](#) 的形式准备并提供给其它 MQL 程序。图表上的 MQL 程序可使用 UI [图形对象](#) 传输和存储信息。

从技术角度而言，全局变量的最大数量仅受限于操作系统的资源。但是，对于大量元素，建议使用更适当的方式：[文件](#) 或 [数据库](#)。

4.6.1 读写全局变量

MySQL API 提供了 2 个函数用于读写全局变量：*GlobalVariableSet* 和 *GlobalVariableGet* (两个版本)。

datetime GlobalVariableSet(const string name, double value)

该函数将新的 *value* 设置到 'name' 全局变量。如果在调用该函数之前该变量不存在，则将创建该变量。如果该变量已存在，则先前值将被替换为 *value*。

如果成功，则函数返回变量修改时间 (计算机的当前本地时间)。如果出错，则返回 0。

double GlobalVariableGet(const string name)

bool GlobalVariableGet(const string name, double &value)

该函数返回 'name' 全局变量的值。调用该函数第一个版本的结果仅包含变量的值 (如果成功) 或 0 (如果出错)。由于变量可能包含 0 值 (与错误指示相同)，如果接收到零，该选项要求解析内部错误代码 *_LastError*，以区分标准版本和非标准版本。尤其是如果试图读取一个并不存在的变量，则产生内部错误 4501 (GLOBAL_VARIABLE_NOT_FOUND)。

该函数版本适用于以下算法场景：获取零能等效模拟先前不存在的变量的默认初始化 (参见下面的示例)。如果变量缺失需要以特殊方式 (尤其是计算某些其它起始值) 处理，则你应首先使用 *GlobalVariableCheck* 函数检查该变量的存在性，然后根据检查结果执行不同的代码分支。或者，你可以使用第二个版本。

该函数的第二个版本根据执行的成功与否返回 *true* 或 *false*。如果成功，则终端的全局变量的值被放入接收 *value* 变量中，通过引用作为第二参数传递。如果没有变量，则返回 *false*。

在测试脚本 *GlobalsRunCount.mq5* 中，我们使用一个全局变量统计其运行次数。变量的名称是源文件的名称。

```
const string gv = __FILE__;
```

别忘了，内置宏 *__FILE__* (参见 [预定义常量](#)) 由编译器扩展到被编译文件的名称中，即在本例中是 "GlobalsRunCount.mq5"。

在 *OnStart* 函数中，我们将尝试读取给定的全局变量，并将结果保存在本地 *count* 变量中。如果还没有全局变量，则返回 0，这对我们来说正常 (我们从零开始计数)。

在将该值保存到 *count* 中之前，需要将其类型强制转换为 (*int*)，因为 *GlobalVariableGet* 函数返回 *double*，如果没有强制转换，则编译器生成关于潜在数据丢失的警告 (它不知道我们打算仅存储整数)。

```

void OnStart()
{
    int count = (int)PRTF(GlobalVariableGet(gv));
    count++;
    PRTF(GlobalVariableSet(gv, count));
    Print("This script run count: ", count);
}

```

然后我们将计数器增加 1，并使用 *GlobalVariableSet* 将其写回到全局变量中。如果我们运行脚本若干次，我们将得到类似下面的日志条目（你的时间戳将不同）：

```

GlobalVariableGet(gv)=0.0 / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariableSet(gv,count)=2021.08.29 16:04:40 / ok
This script run count: 1
GlobalVariableGet(gv)=1.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:05:00 / ok
This script run count: 2
GlobalVariableGet(gv)=2.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:05:21 / ok
This script run count: 3

```

务必要注意，在第一次运行时，我们收到 0 值，并产生内部错误标志 4501。所有后续调用均正常执行，因为变量存在（可以在终端的“全局变量”窗口中看到）。感兴趣者可以关闭终端，重新启动它并再次执行该脚本：计数器将继续从先前的值增加。

4.6.2 检查是否存在和上次活动时间

我们在前一节已经知道，可以通过尝试读取其值来检查全局变量的存在性：如果未导致 *_LastError* 中出现错误代码，则全局变量存在，我们已经获取到其值并将其用于算法中。然而，如果在某些情况下你仅需要检查其存在性而不需要读取全局变量，使用另一个专门为此设计的函数会更方便，即：*GlobalVariableCheck*。

还有另一种检查方式，就是使用 *GlobalVariableTime* 函数。顾名思义，该函数可以找出上次使用变量的时间。但如果该变量不存在，则其使用时间不存在，即等于 0。

bool GlobalVariableCheck(const string name)

该函数使用指定名称检查全局变量的存在性，并返回结果：*true*（变量存在）或者 *false*（变量不存在）。

datetime GlobalVariableTime(const string name)

该函数返回具有指定名称的全局变量的上次使用时间。“使用”这一事实可通过变量值的修改或读取来表示。

使用 *GlobalVariableCheck* 检查变量存在性或通过 *GlobalVariableTime* 获取其时间并不会更改使用时间。

在 *GlobalsRunCheck.mq5* 脚本中，我们对 *GlobalsRunCount.mq5* 的代码进行了小幅补充，以便在函数 *OnStart* 的最开始检查变量的存在性及其使用时间。

```

void OnStart()
{
    PRTF(GlobalVariableCheck(gv));
    PRTF(GlobalVariableTime(gv));
    ...
}

```

下面的代码不变。同时，请注意，通过 `__FILE__` 定义的 `gv` 变量这次将包含新的脚本名称“GlobalsRunCheck.mq5”作为全局变量的名称（即每个脚本有其自己的全局计数器）。

除了第一次运行外，后续所有运行都将在 `GlobalVariableCheck` 函数显示 `true`（变量存在）以及该变量前一次运行的时间。下面是一个示例日志：

```

GlobalVariableCheck(gv)=false / ok
GlobalVariableTime(gv)=1970.01.01 00:00:00 / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariableGet(gv)=0.0 / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariableSet(gv,count)=2021.08.29 16:59:35 / ok
This script run count: 1
GlobalVariableCheck(gv)=true / ok
GlobalVariableTime(gv)=2021.08.29 16:59:35 / ok
GlobalVariableGet(gv)=1.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:59:45 / ok
This script run count: 2
GlobalVariableCheck(gv)=true / ok
GlobalVariableTime(gv)=2021.08.29 16:59:45 / ok
GlobalVariableGet(gv)=2.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:59:56 / ok
This script run count: 3

```

4.6.3 获取全局变量列表

通常，需要 MQL 程序来浏览现有全局变量并选择满足某些条件的全局变量。例如，如果一个程序使用变量名称的一部分来存储文本信息，则仅前缀为事先已知。该前缀的目的是识别“其自己的”变量，而附加到前缀的“有效载荷”不允许按确切名称来搜索变量。

MQL5 API 有两个函数可用于枚举全局变量。

int GlobalVariablesTotal()

该函数返回全局变量总数。

string GlobalVariableName(int index)

该函数按其在全局变量列表中的索引号返回全局变量的名称。带有请求变量编号的 `index` 参数其范围必须在 0 到 `GlobalVariablesTotal() - 1` 之间。

如果出错，则该函数返回 NULL，错误代码可从服务变量 `_LastError` 或从 `GetLastError` 函数获取。

我们使用脚本 `GlobalsList.mq5` 来测试这对函数。


```

void OnStart()
{
    PRTF(GlobalVariableName(1000000));
    int n = PRTF(GlobalVariablesTotal());
    for(int i = 0; i < n; ++i)
    {
        const string name = GlobalVariableName(i);
        PrintFormat("%d %s=%f", i, name, GlobalVariableGet(name));
    }
}

```

第一个字符串故意请求一个大编号变量的名称，而该变量很可能不存在，这样应会导致错误。接下来，请求真实变量数量以及贯穿所有变量的循环，并输出名称和值。下面的日志包括由先前的测试脚本创建的变量以及一个第三方变量。

```

GlobalVariableName(1000000)= / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariablesTotal()=3 / ok
0 GlobalsRunCheck.mq5=3.000000
1 GlobalsRunCount.mq5=4.000000
2 abracadabra=0.000000

```

未定义终端按索引返回变量的顺序。

4.6.4 删除全局变量

如果必要，MQL 程序能够删除已变为冗余的一个或一组全局变量。全局变量列表消耗一些计算机资源，而良好编程风格建议应尽量释放资源。

bool GlobalVariableDel(const string name)

该函数删除 'name' 全局变量。如果成功，该函数返回 *true*，否则返回 *false*。

int GlobalVariablesDeleteAll(const string prefix = NULL, datetime limit = 0)

该函数删除在名称中具有指定前缀的全局变量以及使用时间超过 *limit* 参数值的全局变量。

如果指定了 NULL 前缀（默认）或空字符串，则同时满足“按日期”删除条件（如果已设置）的所有全局变量均满足删除条件。

如果 *limit* 参数为零（默认），则将删除所有符合前缀条件的全局变量（不考虑日期因素）

如果同时指定这两个参数，则将删除既匹配前缀又匹配时间条件的全局变量。

注意：在没有参数的情况下调用 *GlobalVariablesDeleteAll* 将移除所有变量。

该函数返回删除变量的数量。

了解脚本 *GlobalsDelete.mq5*，探索两个新功能。

```

void OnStart()
{
    PRTF(GlobalVariableDel("#123%"));
    PRTF(GlobalVariablesDeleteAll("#123%"));
    ...
}

```

在开始，尝试按其确切名称和前缀删除不存在的全局变量。二者均对现有变量无影响。

若调用 `GlobalVariablesDeleteAll` 时筛选了过去时间 (超过 4 个星期) 也会得到零结果, 因为终端自动删除这些旧变量 (这些变量不可能存在)。

```
PRTF(GlobalVariablesDeleteAll(NULL, D'2021.01.01'));
```

然后, 我们创建一个名称为 “abracadabra” 的变量 (如果不存在) 并立即删除。这些调用应会成功。

```
PRTF(GlobalVariableSet(abracadabra, 0));
PRTF(GlobalVariableDel(abracadabra));
```

最后, 我们删除以 “GlobalsRun” 前缀开头的变量: 它们应该是根据有关文件名的前两个章节中的测试脚本 (分别为 “GlobalsRunCount.mq5” 和 “GlobalsRunCheck.mq5”) 而创建。

```
PRTF(GlobalVariablesDeleteAll("GlobalsRun"));
PRTF(GlobalVariablesTotal());
}
```

该脚本应将如下字符串系列的内容输出到日志 (某些指标取决于外部条件和启动时间)。

```
GlobalVariableDel(#123%)=false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariablesDeleteAll(#123%)=0 / ok
GlobalVariablesDeleteAll(NULL,D'2021.01.01')=0 / ok
GlobalVariableSet(abracadabra,0)=2021.08.30 14:02:32 / ok
GlobalVariableDel(abracadabra)=true / ok
GlobalVariablesDeleteAll(GlobalsRun)=2 / ok
GlobalVariablesTotal()=0 / ok
```

最后, 我们打印剩余全局变量的总数量 (在本例中, 我们得到 0, 即没有变量)。如果全局变量是由其它 MQL 程序或由用户创建, 则你的结果可能不同。

4.6.5 临时全局变量

在终端的全局变量子系统中, 能够将某些变量设为临时变量: 这些变量仅存储在内存中, 当终端关闭时不写入到磁盘。

由于它们的特定性质, 临时全局变量仅用于 MQL 程序之间的数据交换, 不适合用于保存 MetaTrader 5 启动间隔的状态。临时变量最明显的用途之一就是存储运行活动的各种指标 (例如, 程序副本运行计数器), 这些指标应在每次启动时动态重新计算, 而不是从磁盘还原。

一个全局变量应在为其赋予任何值之前使用 `GlobalVariableTemp` 函数将其声明为临时变量。

遗憾的是, 无法通过全局变量的名称确定其是否为临时变量: MQL5 未提供这方面的方法。

临时变量只能使用 MQL 程序来创建。临时变量与普通 (持续性) 全局变量一起显示在 “全局变量” 窗口中, 但是用户无法从 GUI 自行添加临时变量。

`bool GlobalVariableTemp(const string name)`

该函数以指定名称创建一个新的全局变量, 仅存在周期仅限于当前终端会话的生命周期。

如果具有相同名称的变量已存在, 则不会将其转换为一个临时变量。

然而, 如果这样的变量还不存在, 则其初始值为 0。之后, 你可以将其作为常规变量使用, 特别是可以使用 `GlobalVariableSet` 函数为其赋予其它值。

我们将结合下一节的函数演示此函数的用法示例。

4.6.6 使用全局变量同步程序

由于全局变量是在 MQL 程序之外，它们可用于组织控制同一程序的多个副本或在不同程序之间传递信号的外部标志。最简单的例子是限制可以运行的程序的副本数。这一功能很有必要，可以防止在不同图表上意外复制 EA 交易（交易订单可能会因此翻倍），或者为了实施演示版本。

乍一看，此类检查可能可以在源代码中以如下方式完成。

```
void OnStart()
{
    const string gv = "AlreadyRunning";
    // if the variable exists, then one instance is already running
    if(GlobalVariableCheck(gv)) return;
    // create a variable as a flag signaling the presence of a working copy
    GlobalVariableSet(gv, 0);

    while(!IsStopped())
    {
        // work cycle
    }
    // delete variable before exit
    GlobalVariableDel(gv);
}
```

这里显示了一个最简单的版本，以脚本作为示例。对于其它类型的 MQL 程序，总体检查概念将是相同的，尽管指令位置可能不同：EA 交易和指标并没有采用无休止的工作循环，而是使用由终端反复调用的特性事件处理程序。我们将稍后探讨这些问题。

呈现的代码存在的问题是没有考虑 MQL 程序的并行执行。

一个 MQL 程序通常在其自己的线程中运行。对于四种 MQL 程序中的三种（EA 交易、脚本和服务），系统肯定分配单独的线程。至于指标，对于处理相同的操作交易品种和时间范围组合的指标，将为其所有实例分配一个公用线程。但是不同组合的指标仍然属于不同线程。

几乎始终是有很多线程在终端中运行 - 大大超过处理器核心数。因此，每个线程不定期被系统中止，以允许其它线程运行。由于在线程之间的所有这种切换非常迅速，作为用户的我们并不会注意到这一“内部组织过程”。然而，每个中止可能影响不同线程访问共享资源的顺序。全局变量就是此类资源。

从程序角度而言，暂停可能在任何相邻指令之间发生。如果知道这一点，我们再回到示例，就不难看到处理全局变量的逻辑可能被破坏的环节。

实际上，第一个副本（线程）可能执行检查并且未找到变量，但立即被中止。因此，在它有时间使用其下一个指令创建变量之前，执行上下文切换到了第二个副本。第二个副本也将没有找到变量，并将决定象第一个副本一样继续工作。为便于清晰理解，两个副本的相同源代码以其交错执行顺序显示为两列指令。

副本 1

```

void OnStart()
{
    const string gv = "AlreadyRunning";

    if(GlobalVariableCheck(gv)) return;
    // no variable

    GlobalVariableSet(gv, 0);
    // "I am the first and only"
    while(!IsStopped())
    {
        ;
    }
    GlobalVariableDel(gv);
}

```

副本 2

```

void OnStart()
{
    const string gv = "AlreadyRunning";

    if(GlobalVariableCheck(gv)) return;
    // still no variable

    GlobalVariableSet(gv, 0);
    // "No, I'm the first and only one"
    while(!IsStopped())
    {
        ;
    }
    GlobalVariableDel(gv);
}

```

当然，该线程切换方案具有很大的传统性。但在本例中，关键在于很有可能会破坏程序逻辑，即使是在一个字符串中。当有多个程序（线程）时，对于公用资源的不可预见性操作的概率会增加。这可能足以使 EA 在最意想不到的时刻亏损，或者导致获得的技术分析估计失真。

最令人沮丧的是此类错误非常难以检测到。这些错误无法由编译器无法检测到，但在运行时偶尔就会表现出来。但是如果该错误长时间未表现出来，并不意味着没有错误。

为解决此类问题，有必要以某种方式同步所有程序副本对共享资源（在此情况下是全局变量）的访问。

在计算机科学中，有一个特殊概念 - 互斥体，是指一个用于为并行程序中的共享资源提供独占访问权的对象。互斥体可防止数据因异步变更而丢失或损坏。通常情况下，访问互斥体可同步不同的程序，因为这些程序中只有一个能够通过特定时刻捕获互斥体来编辑受保护的数据，而其余程序被迫等待直至互斥体被释放。

在 MQL5 中没有现成的纯形态的互斥体。但对于全局变量，可通过以下函数获得类似的效果，我们将探讨该函数。

bool GlobalVariableSetOnCondition(const string name, double value, double precondition)

该函数设置现有全局变量 *name* 的一个新 *value*，前提是其当前值等于 *precondition*。

如果成功，该函数返回 *true*。否则返回 *false*，并且错误代码将在 *_LastError* 中提供。尤其如果该变量不存在，则函数将生成一个 ERR_GLOBAL_VARIABLE_NOT_FOUND (4501) 错误。

该函数提供对一个全局变量的原子访问，即，该函数分别执行两个操作：检查其当前值，如果与条件匹配，则其向变量赋予一个新 *value*。

等效函数代码可近似表示如下（为什么是“近似”，我们将稍后解释）：

```

bool GlobalVariableSetOnCondition(const string name, double value, double precondition)
{
    bool result = false;
    { /* enable interrupt protection */ }
    if(GlobalVariableCheck(name) && (GlobalVariableGet(name) == precondition))
    {
        GlobalVariableSet(name, value);
        result = true;
    }
    { /* disable interrupt protection */ }
    return result;
}

```

无法实现类似这样按预期工作的代码，原因有二。首先，在纯 MQL5 中，没有实现启用和禁用中断保护的块的功能（在内置 **GlobalVariableSetOnCondition** 函数中，这是由内核本身来提供）。其次，**GlobalVariableGet** 函数调用会更改变量上次使用时间，而如果前提条件未满足，**GlobalVariableSetOnCondition** 函数不会更改它。

为了说明如何使用 **GlobalVariableSetOnCondition**，我们将了解一种新的 MQL 程序类型：服务。我们将在单独的 章节中详细了解它们。目前需要注意的是，它们的结构与脚本十分相似：对于二者，仅有一个主函数（入口点）**OnStart**。唯一的主要差别是，脚本在图表上运行，而服务独立运行（在后台运行）。

需要将脚本替换为服务的原因是，我们在其中使用了 **GlobalVariableSetOnCondition** 的任务的实际意义在于统计程序的运行实例数量，并可以设置一个限值。在此情况下，只有启动多个程序时，才会造成与共享计数器同时修改的冲突。然而，对于脚本，难以在相对较短的时间内在不同图表上运行脚本的多个副本。相反，对于服务，终端接口具有用于批量（群组）启动的便利机制。此外，在下次终端启动时，所有激活的服务将自动启动。

我们拟定的用于统计副本数的机制也适用于其它类型的 MQL 程序。由于 EA 交易和指标即使在终端关闭后也仍然连接到图表，下次终端打开时，所有程序几乎同时读取它们的设置和共享资源。因此，如果对于副本数的限值内置于某些 EA 交易和指标中，则基于全局变量同步计数至关重要。

首先，我们了解一种没有使用 **GlobalVariableSetOnCondition**，以不成熟模式实现拷贝控制的服务，确保计数器失败问题的真实性。该服务位于总源代码目录中的一个专用子目录，其扩展路径为 **MQL5/Services/MQL5Book/p4/GlobalsNoCondition.mq5**。

在服务文件的开头应有一个指令：

```
#property service
```

在该服务中，我们将提供 2 个输入变量以设置允许并行运行的副本量上限，以及设置用于模拟因计算机磁盘和 CPU 高负载导致的执行中断（这在终端启动时经常发生）的延迟。这样更容易复现问题，无需为了中断同步而多次重启终端。因此，我们将捕获一个仅会偶然发生但一旦发生却伴随严重后果的漏洞问题。

```

input int limit = 1;          // Limit
input int startPause = 100; // Delay(ms)

```

延迟模拟是基于 **Sleep** 函数。

```

void Delay()
{
    if(startPause > 0)
    {
        Sleep(startPause);
    }
}

```

首先，在 *OnStart* 函数内声明一个临时全局变量。由于它设计用于统计运行的程序副本数量，没有必要使其为持续性的：每次启动终端，均需要重新计数。

```

void OnStart()
{
    PRTF(GlobalVariableTemp(__FILE__));
    ...
}

```

为了避免用户事先创建一个同名变量并向其赋予一个负值的情况，我们引入了保护机制。

```

int count = (int)GlobalVariableGet(__FILE__);
if(count < 0)
{
    Print("Negative count detected. Not allowed.");
    return;
}

```

接下来，主要功能片段开始。如果计数器已经大于或等于最大允许数量，则我们中断程序启动。

```

if(count >= limit)
{
    PrintFormat("Can't start more than %d copy(s)", limit);
    return;
}

```

否则，我们将计数器增加 1，并将其写入到全局变量。事先，我们模拟延迟，诱发另一个程序可能对在我们的程序中读取变量和写入变量之间进行干涉的情况。

```

Delay();
PRTF(GlobalVariableSet(__FILE__, count + 1));

```

如果确实发生了这种情况，我们的程序副本将递增并赋予一个已经过时的不正确的值。这将导致这样一种情况：在与我们的程序并行运行的另一个程序副本中，同一 *count* 值已经被处理过或者将被再次处理。

该服务的有用工作由以下循环表示：

```

int loop = 0;
while(!IsStopped())
{
    PrintFormat("Copy %d is working [%d]...", count, loop++);
    // ...
    Sleep(3000);
}

```

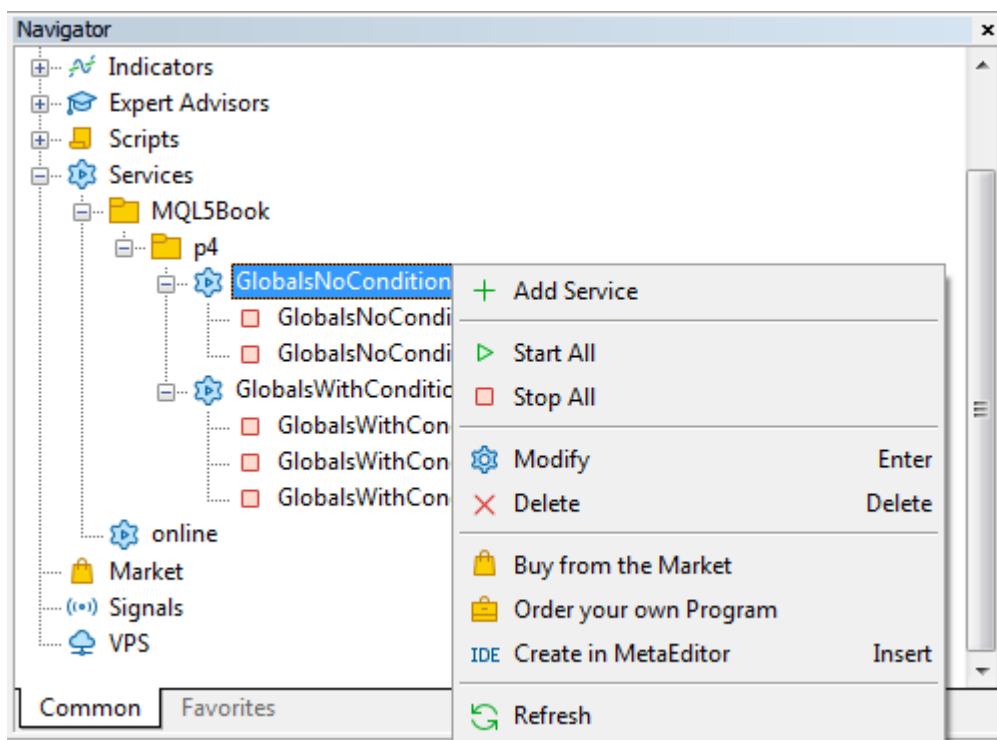
用户停止服务后 (为此, 接口有一个上下文菜单; 稍后将进一步详述), 循环将结束, 我们需要将计数器递减。

```

int last = (int)GlobalVariableGet(__FILE__);
if(last > 0)
{
    PrintFormat("Copy %d (out of %d) is stopping", count, last);
    Delay();
    PRTF(GlobalVariableSet(__FILE__, last - 1));
}
else
{
    Print("Count underflow");
}
}

```

编译的服务属于“导航器”的对应分支。



“导航器”中的服务及其上下文菜单

单击右键, 我们将打开上下文菜单并通过调用 `Add service` 命令两次来创建 `GlobalsNoCondition.mq5` 服务的两个实例。在此情况下, 每次对话框打开时都会显示服务设置, 其中你应保留参数的默认值。

请务必注意，**Add service** 命令会立即启动创建的服务。但我们不需要这样。因此，启动每个副本之后，我们必须立即再次调用上下文菜单并执行 **Stop** 命令（如果选择了特定实例），或者执行 **Stop everything**（如果选择了程序，即整组生成的实例）。

服务的第一个实例的默认名称与服务文件（“GlobalsNoCondition”）完全一致，并且在所有后续实例中，将自动添加一个增量编号。尤其是第二个实例将被列为 “GlobalsNoCondition 1”。终端允许你使用 **Rename** 命令将实例重命名为任意文本，但我们将不那样做。

现在一切就绪，可以进行试验。我们尝试同时运行两个实例。为此，我们为对应的 **GlobalsNoCondition** 分支运行 **Run All** 命令。

我们别忘了，参数中设置了 1 个实例的限值。然而，根据日志显示，该限制并未生效。

```
GlobalsNoCondition GlobalVariableTemp(GlobalsNoCondition.mq5)=true / ok
GlobalsNoCondition 1 GlobalVariableTemp(GlobalsNoCondition.mq5)=false / GLOBALVARIABLE
GlobalsNoCondition GlobalVariableSet(GlobalsNoCondition.mq5,count+1)=2021.08.31 17:47
GlobalsNoCondition Copy 0 is working [0]...
GlobalsNoCondition 1 GlobalVariableSet(GlobalsNoCondition.mq5,count+1)=2021.08.31 17:
GlobalsNoCondition 1 Copy 0 is working [0]...
GlobalsNoCondition Copy 0 is working [1]...
GlobalsNoCondition 1 Copy 0 is working [1]...
GlobalsNoCondition Copy 0 is working [2]...
GlobalsNoCondition 1 Copy 0 is working [2]...
GlobalsNoCondition Copy 0 is working [3]...
GlobalsNoCondition 1 Copy 0 is working [3]...
GlobalsNoCondition Copy 0 (out of 1) is stopping
GlobalsNoCondition GlobalVariableSet(GlobalsNoCondition.mq5,last-1)=2021.08.31 17:47:
GlobalsNoCondition 1 Count underflow
```

两个副本均“认为”它们是 0 号（从工作循环中输出 “Copy 0”），并且它们的总数错误地等于 1，因为这是两个副本存储在计数器变量中的值。

正因如此，当服务停止 **Stop everything** 命令时，我们收到一条关于不正确状态的消息（“Count underflow”）：毕竟，每个副本均试图将计数器减少 1，结果，第二个执行的副本收到一个负值。

为解决这一问题，需要使用 **GlobalVariableSetOnCondition** 函数。基于前一个服务的源代码，准备了改进版本的 **GlobalsWithCondition.mq5**。总体而言，它复现了其前身的逻辑，但有重大区别。

相比仅调用 **GlobalVariableSet** 来增加计数器，必须编写一个更复杂的结构体。

```

const int maxRetries = 5;
int retry = 0;

while(count < limit && retry < maxRetries)
{
    Delay();
    if(PRTF(GlobalVariableSetOnCondition(__FILE__, count + 1, count))) break;
    // condition is not met (count is obsolete), assignment failed,
    // let's try again with a new condition if the loop does not exceed the limit
    count = (int)GlobalVariableGet(__FILE__);
    PrintFormat("Counter is already altered by other instance: %d", count);
    retry++;
}

if(count == limit || retry == maxRetries)
{
    PrintFormat("Start failed: count: %d, retries: %d", count, retry);
    return;
}
...

```

如果旧值已过时，*GlobalVariableSetOnCondition* 函数可能会因此而无法写入一个新的计数器值，我们再次在循环中读取全局量并重新尝试将其递增，直至超过允许的最大计数器值。循环条件还限制尝试次数。如果循环因违反任一条件而结束，则计数器更新失败，程序不应继续运行。

同步策略

理论上，有若干个实施共享资源获取的标准策略。

第一个策略是软检查资源是否空闲，然后仅当资源在检查时刻空闲时才将其锁定。如果资源忙碌，则算法计划在一段时间后再次尝试，而此时资源在处理其它任务（这就是为什么具有若干活动/职责领域的程序会首选这种方法）。在 *GlobalVariableSetOnCondition* 函数的转录中对这一行为方案的模拟是一个单次调用，没有循环，失败则退出当前块。变量更改被延迟“直至更好的时机”。

第二个策略更具持续性，并应用在我们的脚本中。这是一个对资源请求重复给定次数或重复预定义时间（资源允许的超时时间）的循环。如果循环到期并且未获得正结果（调用 *GlobalVariableSetOnCondition* 函数从未返回 true），则同样退出当前块并可能计划稍后再试。

最后，第三个策略，是最坚韧不拔的一个，它“坚持不懈”地请求资源。可以将它视为一个具有函数调用的无限循环。该方法适合用于着眼于一个特定任务并且没有取得资源就无法继续工作的程序。在 MQL5 中，将 *while(!IsStopped())* 循环用于此目的，并且别忘了在内部调用 *Sleep*。

重要的是，需注意潜在的激烈抢夺多个资源的问题。想象一个 MQL 程序修改若干全局变量（理论上这是常见情况）。如果该程序的一个副本捕获一个变量，第二个副本捕获另一个变量，两个副本均等待变量释放，结果将导致它们相互阻塞（思索）。

基于前述原因，全局变量及其它资源（例如文件）的共享应针对锁定和所谓的“竞争情况”仔细设计和分析，在这些情况下程序的并行执行会导致未定义的结果（取决于它们的工作顺序）。

当新版本服务中的工作循环完成之后，计数器递减算法已经以类似的方式更改。

```

retry = 0;
int last = (int)GlobalVariableGet(__FILE__);
while(last > 0 && retry < maxRetries)
{
    PrintFormat("Copy %d (out of %d) is stopping", count, last);
    Delay();
    if(PRTF(GlobalVariableSetOnCondition(__FILE__, last - 1, last))) break;
    last = (int)GlobalVariableGet(__FILE__);
    retry++;
}

if(last <= 0)
{
    PrintFormat("Unexpected exit: %d", last);
}
else
{
    PrintFormat("Stopped copy %d: count: %d, retries: %d", count, last, retry);
}

```

作为试验，我们为新服务创建三个实例。在每个实例的设置中，在“Limit”参数中，我们指定 2 个实例（在条件变化的情况下进行测试）。别忘了，创建每个实例会立即启动它，我们并不需要这样，因此，应停止每个新创建的实例。

这些实例将会获得默认名称“GlobalSwitchCondition”、“GlobalSwitchCondition 1”，以及“GlobalSwitchCondition 2”。

当一切就绪，我们一次性运行所有实例，并在日志中得到类似如下内容。


```

GlobalsWithCondition 2 GlobalVariableTemp(GlobalsWithCondition.mq5)= »
» false / GLOBALVARIABLE_EXISTS(4502)
GlobalsWithCondition 1 GlobalVariableTemp(GlobalsWithCondition.mq5)= »
» false / GLOBALVARIABLE_EXISTS(4502)
GlobalsWithCondition GlobalVariableTemp(GlobalsWithCondition.mq5)=true / ok
GlobalsWithCondition GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,co
» true / ok
GlobalsWithCondition 1 GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,
» false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalsWithCondition 2 GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,
» false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalsWithCondition 1 Counter is already altered by other instance: 1
GlobalsWithCondition Copy 0 is working [0]...
GlobalsWithCondition 2 Counter is already altered by other instance: 1
GlobalsWithCondition 1 GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,
GlobalsWithCondition 1 Copy 1 is working [0]...
GlobalsWithCondition 2 GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,
» false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalsWithCondition 2 Counter is already altered by other instance: 2
GlobalsWithCondition 2 Start failed: count: 2, retries: 2
GlobalsWithCondition Copy 0 is working [1]...
GlobalsWithCondition 1 Copy 1 is working [1]...
GlobalsWithCondition Copy 0 is working [2]...
GlobalsWithCondition 1 Copy 1 is working [2]...
GlobalsWithCondition Copy 0 is working [3]...
GlobalsWithCondition 1 Copy 1 is working [3]...
GlobalsWithCondition Copy 0 (out of 2) is stopping
GlobalsWithCondition GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,last-1,las
GlobalsWithCondition Stopped copy 0: count: 2, retries: 0
GlobalsWithCondition 1 Copy 1 (out of 1) is stopping
GlobalsWithCondition 1 GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,last-1,l
GlobalsWithCondition 1 Stopped copy 1: count: 1, retries: 0

```

首先要注意并行运行程序的上下文切换效应:其表现是随机的,但又能直观展示我们描述的现象。创建了一个临时变量的第一个实例是没有编号的 “GlobalsWithCondition”:这可以从 **GlobalVariableTemp** 函数的结果 **true** 看出。然而,在日志中,该行仅占用第三个位置,而前两个包含调用具有编号 1 和 2 的名称的副本中的同一函数的结果;其中函数 **GlobalVariableTemp** 返回 **false**。这表示这些副本后来检查了变量,尽管它们的线程当时超越了无编号的 “GlobalsWithCondition” 线程并更早出现在日志中。

我们回到我们的主程序计数算法。实例 “GlobalsWithCondition” 首先通过了检查,并以内部标识符 “Copy 0” 开始工作 (我们不能从服务代码获知用户是如何命名了该实例的:在 MQL5 API 中没有此类函数,至少目前没有)。

利用 **GlobalVariableSetOnCondition** 函数,在实例 1 和 2 (“GlobalsWithCondition 1”, “GlobalsWithCondition 2”) 中,修改计数器的事实被检测到:一开始其为 0,但 GlobalsWithCondition 将其增加了 1。两个模板实例均输出消息 “计数器已被其它实例更改:1”。在 2 号前面的实例之一 (“GlobalsWithCondition 1”) 成功从变量获得新值 1 并将其增加到 2。**GlobalVariableSetOnCondition** 调用成功表明了这一点 (其返回 **true**)。对此,会显示一条提示其开始工作的消息:“副本 1 正在工作”。

内部计数器的值与外部实例编号相同这一情况完全是巧合。完全有可能 “GlobalsWithCondition 2” 是在 “GlobalsWithCondition 1” 之前启动 (或者以其它顺序启动,如果有三个副本的话)。如果是那样的话,则外部和内部编号将有所不同。你可以重复试验启动和停止所有服务很多次,实

例增加计数器变量的顺序将很可能是不同的。但在任何情况下，对于总数的限值将截断一个额外实例。

当“GlobalsWithCondition 2”的最后一个实例获准访问全局变量时，变量中已经存储了值 2。由于这是我们设置的限值，因此程序未启动。

```
GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,count)= »
» false / GLOBALVARIABLE_NOT_FOUND(4501)
Counter is already altered by other instance: 2
Start failed: count: 2, retries: 2
```

此外，“GlobalsWithCondition”和“GlobalsWithCondition 1”的副本在工作循环中“回旋”，直至服务停止。

你可以尝试仅停止一个实例。这样便能启动另一个之前因超过配额而被禁止执行的实例。

当然，上述防止并行修改的版本保护机制仅对于协调你自己的程序的行为有效，无法限制演示版本的单个副本，因为用户可以直接删除全局变量。为此，全局变量可以以一种不同的方式使用 - 与图表 ID 相关：仅当 MQL 程序创建的全局变量包含其图表 [图形对象](#)。要以其他方式控制共享数据（计数器及其它信息），还可以使用 [资源](#) 和 [数据库](#)。

4.6.7 将全局变量刷写到磁盘

为优化性能，当终端运行时，全局变量驻留在内存中。然而，我们知道，变量在会话之间被存储到一个特殊文件中。这适用于所有全局变量，[临时](#) 变量除外。通常，当终端关闭时，会将变量写入到文件。不过若是你的计算机突然崩溃，数据可能丢失。因此，为了保证任何无法预料情况下的数据安全，强制发起写入操作很有用。为此，MQL5 API 提供了 **GlobalVariablesFlush** 函数。

void GlobalVariablesFlush()

该函数强制将全局变量的内容写入到磁盘。该函数没有参数且不会返回任何内容。

最简单的示例在 **GlobalsFlush.mq5** 脚本中提供。

```
void OnStart()
{
    GlobalVariablesFlush();
}
```

利用该函数，你可以在必要时随时将变量刷写到磁盘。你可以使用你偏爱的文件管理器并确保 **gvariables.dat** 文件的日期和时间在脚本运行后立即更改。但请注意，仅当全局变量自前次保存后已经过任何方式的编辑或只是被读取（这会更改访问时间），该文件才会被更新。

该脚本对于保持终端长时间开机运行并且修改全局变量的程序在终端中执行的场景很有用。

4.7 时间处理函数

时间在大多数流程中都是一个基本要素，对于交易也扮演着重要应用角色。

我们知道，交易中的主要系统基于两个维度：价格和时间。它们分别沿纵轴和横轴显示在图表上。稍后我们将了解另一个重要的轴，其可表示为垂直于前两条轴，并在图表中体现更深的维度，在这个轴上标记交易量。但是现在我们先来了解时间。

时间度量对所有图表通用，使用相同的测量单位，虽然听起来有些奇怪，但它的特征在于恒定性（时间进程是可以预测的）。

终端内置了大量与时间计算和分析相关的工具。随着我们学习本书的各个章节，我们将由浅入深逐渐熟悉这些工具。

在本章中，我们将学习可控制时间以及将程序活动暂停指定间隔时间的函数。

在 [日期和时间](#) 一章中关于数据转换的章节，我们已经接触了与时间相关的几个函数：*TimeToStruct* 和 *StructToTime*。它们分别用于将 *datetime* 类型的值拆分为分量，以及从单独字段构建 *datetime*：别忘了，在 *MqlDateTime* 结构体中总结过这两个函数。

```
struct MqlDateTime
{
    int year;           // year (1970 - 3000)
    int mon;            // month (1 - 12)
    int day;            // day (1 - 31)
    int hour;           // hours (0 - 23)
    int min;            // minutes (0 - 59)
    int sec;            // seconds (0 - 59)
    int day_of_week;    // day of the week, numbered from 0 (Sunday) to 6 (Saturday)
                        // according to enum ENUM_DAY_OF_WEEK
    int day_of_year;    // ordinal number of the day in the year, starting from 0 (January)
};
```

但 MQL 程序可以从哪里获取到 *datetime* 值呢？

例如，历史价格和时间反映在报价中，而当前实时数据记为分时报价。二者均有时间戳，我们将在关于 [时间序列](#) 和 [终端事件](#) 的相关章节学习如何获取。然而，MQL 程序可以使用若干函数自行查询当前时间（不含价格或其它交易信息）。

需要多于函数，因为系统是分布式的：由位于全球任意位置的客户端终端和经纪人服务器组成，它们很可能属于不同时区。

任何时区的特征都是相对于全球参考时间格林威治标准时间 (GMT) 的临时偏移。原则上，时区偏移值是一个整数小时数 N （尽管也有半小时梯度的特异时区），因此可以表示为 $GMT + N$ 或 $GMT - N$ ，取决于时区是在子午线以东还是以西。例如位于伦敦以东的欧洲大陆使用中欧时间 (CET) 等于 $GMT + 1$ ，或者东欧时间 (EET) 等于 $GMT + 2$ ，而在美洲则有“负”时区，诸如东部标准时间 (EST) 或 $GMT - 5$ 。

需要注意的是，GMT 对应于天文（太阳）时间，该时间因地球旋转速度逐渐减慢而稍微呈现出非线性特征。最近数十年，实际上一直在向更为精确的计时系统（基于原子钟）过渡，其中全球时间称为协调世界时 (UTC)。在包括交易在内的很多应用领域，GMT 和 UTC 的差异不大，因此新的 UTC 和旧的 GMT 格式的时区名称应视为类似名称。例如，很多经纪人在其规范中已经使用 UTC 指定交易时段，而 MQL5 API 一直以来使用 GMT 标记法。

MQL5 API 支持查找终端的当前时间（即计算机的本机时间）以及服务器时间：它们分别由 *TimeLocal* 和 *TimeCurrent* 函数返回。此外，MQL 程序能够基于 Windows 时区设置获取当前 GMT 时间（*TimeGMT* 函数）。因此，交易员和编程人员获得本地时间与全球时间的绑定，通过本地时间和服务器时间之间的差异，即可判定服务器和报价的“时区”。但这里还有几个有趣的点。

首先，在很多国家，有实行切换到夏令时 (DST) 的习惯。通常，这表示大概从三月/四月到十月/十一月期间对标准（冬季）时间加 1 小时（在北半球，南半球则减去 1 小时）。同时，GMT/UTC 时间始终恒定，即其不进行 DST 修正，因此，客户和服务器时间之间的趋同/差异的各种选项成为潜在可能：

- 各个国家的夏令时生效日期有所不同
- 有些国家不实行夏令时

因此，如果算法是基于对当日时间的引用（例如新闻发布）而不是基于对价格变动或交易量密度的引用，一些 MQL 程序需要跟踪此类时区变化。

并且如果用户的计算机上的时间转换很容易确定（使用 *TimeDaylightSavings* 函数），则对于服务器时间并没有现成类似转换。

其次，遗憾的是，常规 MetaTrader 5 测试程序（用于调试或评估诸如 EA 交易和指标等类型的 MQL 程序）无法模拟交易服务器的时间。相反，所有上述三个函数 *TimeLocal*、*TimeGMT* 以及 *TimeCurrent* 将返回相同的时间，即时区始终是虚拟 GMT。

绝对和相对时间。

算法中的时间核算如现实生活一样，可以以绝对或相对坐标执行。在过去、现在和将来的每个时刻均用一个绝对值描述，我们可以引用该值以指示会计期间的开始或者指示经济新闻发布时间。在 MQL5 中，我们正是以 *datetime* 类型存储该时间。同时，也经常需要从当前时刻起往将来延伸或向过去回退给定的时间单位。在此情况下，我们关注的不是绝对值，而是时间间隔。

尤其是算法有超时概念，即在一段时间内必须执行某一操作，如果因任何原因未执行，则取消操作并停止等待结果（很明显是因为出错了）。可以以不同单位测量时间间隔：小时、秒、毫秒，甚至微秒（毕竟现在的计算机速度极快）。

在 MQL5 中，某些时间相关函数处理绝对值（例如，*TimeLocal*、*TimeCurrent*）以及时间间隔部分（例如，*GetTickCount*、*GetMicrosecondCount*）。

然而，时间间隔的测量以及程序在指定间隔时间的激活不仅可以通过本节的函数实现，还可以使用根据众所周知的闹钟原理工作的内置计时器来实现。启用计时器后，计时器会使用特殊事件来通知 MQL 程序以及我们实现的用于处理这些事件的函数 - *OnTimer*（它们类似于 *OnStart*）。我们将在单独章节中探讨时间管理的这个方面，在学习了 MQL5 中的一般事件概念之后再探讨它们（参见 [事件处理函数概述](#)）。

4.7.1 本地时间和服务器时间

在 MetaTrader 5 平台上始终有两种类型的时间：本地（客户）时间和服务器（经纪人）时间。

本地时间对应于运行终端的计算机时间，该时间以与真实世界相同的速率持续增加。

服务器时间的流动方式则不同。服务器时间的依据按经纪人的计算机上的时间设定，然而，客户端仅与下次价格变动一同接收到关于服务器时间的信息，这种价格变动被包装进特殊结构，称为分时报价（参见关于 *MqlTick* 的章节），并使用 [事件](#) 将其传递给 MQL 程序。

因此，仅当市场上的至少一个金融工具（即“市场报价”窗口中选择的金融工具之一）的价格变动之后，终端才能知道更新的服务器时间。最后已知的服务器时间显示在该窗口的标题栏中。如果没有分时报价，则终端中的服务器时间保持静止。这在周末和节假日尤其明显，这时候所有交易所和外汇平台均关闭。

尤其是在星期日，服务器时间将很可能显示为星期五晚上。唯一的例外是 MetaTrader 5 实例，其提供持续交易金融工具（诸如加密数字货币）。然而，即使在这种情况下，在低波动期间，服务器时间也可能明显滞后于本机时间。

本节中所有函数以精确到秒（*datetime* 类型的时间表示精度）的精度处理时间。

为获取本地和服务器时间，MQL5 API 提供了三个函数：*TimeLocal*、*TimeCurrent* 以及 *TimeTradeServer*。三个函数均有两个原型版本：第一个版本以 *datetime* 类型的值返回时间，第二个版本还通过引用接受和填充带时间分量的 *MqlDateTime* 结构体。

datetime TimeLocal()

datetime TimeLocal(MqlDateTime &dt)

该函数以 *datetime* 格式返回本地计算机时间。

务必要注意的是，时间包括夏令时（如果启用）。即 *TimeLocal* 等于计算机的时区的标准时间，减去修正 *TimeDaylightSavings*。该公式可条件性地表示如下：

```
TimeLocal summer() = TimeLocal winter() - TimeDaylightSavings()
```

其中 *TimeDaylightSavings* 通常等于 -3600，也就是将时钟前移 1 小时（损失 1 小时）。因此，相对于 UTC，*TimeLocal* 的夏季时间值大于冬季时间值（在天文时间相等的情况下）。例如，如果在冬季 *TimeLocal* 等于 UTC+2，则在夏季等于 UTC+3。UTC 可使用 *TimeGMT* 函数获得。

datetime TimeCurrent()

datetime TimeCurrent(MqlDateTime &dt)

该函数以 *datetime* 格式返回最后已知服务器时间。这是“市场报价”中所有金融工具列表的上次报价抵达时间。唯一例外是 EA 交易中的 *OnTick* 事件处理程序，其中该函数将返回已处理的分时报价的时间（即使时间更接近的分时报价已出现在“市场报价”中）。

另外，应注意，MetaTrader 5 中所有图表水平轴上的时间对应于服务器时间（历史习惯）。最后（当前，最右边）条柱包含 *TimeCurrent*。详情参见 图表 一节。

datetime TimeTradeServer()

datetime TimeTradeServer(MqlDateTime &dt)

该函数返回交易服务器的估计当前时间。不同于 *TimeCurrent*（如果没有新的报价，其结果可能不会改变），*TimeTradeServer* 允许你获取持续增加的服务器时间的估计值。计算是基于客户端时区与服务器时区之间的最后已知差值，该差值与当前本地时间相加。

在测试程序中，*TimeTradeServer* 值始终等于 *TimeCurrent*。

该函数用法示例在 *TimeCheck.mq5* 脚本中提供。

主函数具有持续不间断记录的所有类型时间的无限循环，直至用户停止该脚本。

```
void OnStart()
{
    while(!IsStopped())
    {
        PRTF(TimeLocal());
        PRTF(TimeCurrent());
        PRTF(TimeTradeServer());
        PRTF(TimeTradeServerExact());
        Sleep(1000);
    }
}
```

除了标准函数外，这里还应用了自定义函数 *TimeTradeServerExact*。

```

datetime TimeTradeServerExact()
{
    enum LOCATION
    {
        LOCAL,
        SERVER,
    };
    static datetime now[2] = {}, then[2] = {};
    static int shiftInHours = 0;
    static long shiftInSeconds = 0;

    // constantly detect the last 2 timestamps here and there
    then[LOCAL] = now[LOCAL];
    then[SERVER] = now[SERVER];
    now[LOCAL] = TimeLocal();
    now[SERVER] = TimeCurrent();

    // at the first call we don't have 2 labels yet,
    // needed to calculate the stable difference
    if(then[LOCAL] == 0 && then[SERVER] == 0) return 0;

    // when the time course is the same on the client and on the server,
    // and the server is not "frozen" due to weekends/holidays,
    // updating difference
    if(now[LOCAL] - now[SERVER] == then[LOCAL] - then[SERVER]
    && now[SERVER] != then[SERVER])
    {
        shiftInSeconds = now[LOCAL] - now[SERVER];
        shiftInHours = (int)MathRound(shiftInSeconds / 3600.0);
        // debug print
        PrintFormat("Shift update: hours: %d; seconds: %lld", shiftInHours, shiftInSecc
    }

    // NB: The built-in function TimeTradeServer calculates like this:
    //          TimeLocal() - shiftInHours * 3600
    return (datetime)(TimeLocal() - shiftInSeconds);
}

```

该函数是必需的，因为内置 *TimeTradeServer* 函数的算法不一定适合所有情况。内置函数找到本地时间和服务器时间之间以小时表示的差值（即时区差值），然后获取服务器时间，以根据该差值对本地时间进行修正。因此，如果客户端和服务器的分钟数和秒数不同步（这是极有可能的），则服务器时间的标准近似值将显示客户端的分钟数和秒数，而不是服务器的分钟数和秒数。

理想情况下，所有计算机的本地时钟应与全球时间同步，但在现实中会发生偏差。因此，即使二者之间只是出现很小偏差，*TimeTradeServer* 也无法再以最高精度重现服务器上的时间。

我们在 MQL5 中对同一函数的实现中，不将客户端与服务器时间之间的差值舍入至小时时区。相反，在计算中使用以秒数表示的准确差值。这就是为什么 *TimeTradeServerExact* 返回的时间其分钟数和秒数象服务器上一样显示。

下面是由该脚本生成的一个日志的示例。

```
TimeLocal()=2021.09.02 16:03:34 / ok
TimeCurrent()=2021.09.02 15:59:39 / ok
TimeTradeServer()=2021.09.02 16:03:34 / ok
TimeTradeServerExact()=1970.01.01 00:00:00 / ok
```

可以看到，客户端和服务器的时区相同，但有若干分钟的不同步（为了清晰展示）。在第一次调用中，**TimeTradeServerExact** 返回 0。此外，用于计算差值的数据将已到达，我们将会看到所有四个时间类型以几秒钟的间隔一致“走动”。

```
TimeLocal()=2021.09.02 16:03:35 / ok
TimeCurrent()=2021.09.02 15:59:40 / ok
TimeTradeServer()=2021.09.02 16:03:35 / ok
Shift update: hours: 0; seconds: 235
TimeTradeServerExact()=2021.09.02 15:59:40 / ok
TimeLocal()=2021.09.02 16:03:36 / ok
TimeCurrent()=2021.09.02 15:59:41 / ok
TimeTradeServer()=2021.09.02 16:03:36 / ok
Shift update: hours: 0; seconds: 235
TimeTradeServerExact()=2021.09.02 15:59:41 / ok
TimeLocal()=2021.09.02 16:03:37 / ok
TimeCurrent()=2021.09.02 15:59:41 / ok
TimeTradeServer()=2021.09.02 16:03:37 / ok
TimeTradeServerExact()=2021.09.02 15:59:42 / ok
TimeLocal()=2021.09.02 16:03:38 / ok
TimeCurrent()=2021.09.02 15:59:43 / ok
TimeTradeServer()=2021.09.02 16:03:38 / ok
TimeTradeServerExact()=2021.09.02 15:59:43 / ok
```

4.7.2 夏令时（本地）

为确定本地时钟是否已切换至夏令时，MQL5 提供了 **TimeDaylightSavings** 函数。该函数会获取你的操作系统的设置。

确定服务器上的夏令时时间并非易事。为此，你需要对 [报价](#)、[财经日历](#) 事件或展期交割/掉期时间（在 [账户交易历史记录](#) 中）实施 MQL5 分析。在下面的示例中，我们将显示其中一种方案。

int TimeDaylightSavings()

该函数返回秒数修正（如果已应用了夏令时）。冬季时间是每个时区的标准时间，因此该期间的修正为零。条件性形式的获取修正的公式可编写如下：

```
TimeDaylightSavings() = TimeLocal winter() - TimeLocal summer()
```

例如，如果标准时区（**winter**）等于 UTC+3（即时区时间比 UTC 早 3 小时），则在转换到夏令时（**summer**）的过程中，我们加 1 小时，得到 UTC+4。其中 **TimeDaylightSavings** 将返回 -3600。

该函数的一个使用示例在 **TimeSummer.mq5** 脚本中提供，其中还给出了一种识别服务器上适当模式的可能实证方法。


```

void OnStart()
{
    PRTF(TimeLocal());           // local time of the terminal
    PRTF(TimeCurrent());         // last known server time
    PRTF(TimeTradeServer());     // estimated server time
    PRTF(TimeGMT());             // GMT time (calculation from local via time zone shif
    PRTF(TimeGMTOffset());       // time zone shift compare to GMT, in seconds
    PRTF(TimeDaylightSavings()); // correction for summer time in seconds
    ...
}

```

首先，我们显示所有类型的时间以及 MQL5 提供的修正（函数 *TimeGMT* 和 *TimeGMTOffset* 将在关于 世界时间的下一节中探讨，但从前面的描述中我们应该已经大致清楚它们的含义了）。

该脚本应在交易日运行。日志中的条目将对应于你的计算机和经纪商的服务器的设置。

```

TimeLocal()=2021.09.09 22:06:17 / ok
TimeCurrent()=2021.09.09 22:06:10 / ok
TimeTradeServer()=2021.09.09 22:06:17 / ok
TimeGMT()=2021.09.09 19:06:17 / ok
TimeGMTOffset()=-10800 / ok
TimeDaylightSavings()=0 / ok

```

在本例中，客户的时区与 GMT 相差 3 小时 (UTC+3)，没有夏令时调整。

现在我们看看服务器的情况。基于 *TimeCurrent* 函数的值，我们可以确定服务器的当前时间，但不能确定其标准时区，因为该时间可能涉及夏令时转换（MQL5 不提供关于是否使用夏令时以及是否当前已启用夏令时的信息）。

为确定服务器的真实时区和夏令时，我们将利用服务器时间转换会影响报价这一事实。与大部分用于解决问题的实证方法一样，这一实证方法在某些情况下可能无法给出完全正确的结果。如果与其它来源的比较显示不一致，则应选择不同的方法。

外汇市场在 UTC 时间星期六 22:00 点开盘（对应于亚太地区早盘开始），在星期五 22:00 点关闭（美洲交易收盘）。这表示在 UTC+2 时区（东欧）的服务器上，第一柱线将在星期一 0 时 0 分准时出现。根据中欧时间（对应于 UTC+1），交易周在星期六 23:00 点开始。

在计算了每周末暂停后第一柱线 H1 的当天偏移统计之后，我们将获得服务器时区的估计结果。当然，为此最好使用流动性最强的外汇金融工具 EURUSD。

如果在年度期间的统计中找到两个最大当日偏移，并且它们的位置彼此相邻，这就表示经纪商正在切换到夏令时，反之亦然。

请注意，夏季和冬季时间周期不相等。因此，若在三月初切换到夏季时间而在十一月初返回冬季时间时，我们得到约 8 个月的夏季时间。这将影响统计中最大值的比率。

有了两个时区之后，我们就可以轻松确定当前处于活动状态的时区，进而确定当前是否存在夏令时修正。

将时钟切换到夏令时，经纪商的时区将从 UTC+2 变更为 UTC+3，将把周开始时间从 22:00 点偏移到 21:00 点。这将影响 H1 柱线的结构：在图表上，我们将看到在星期天晚上有三个柱线，而不是两个。



EURUSD H1 图表上从冬季时间 (UTC+2) 到夏季 (UTC+3) 时间的变化

为实现这点，我们有一个单独函数 `ServerTimeZone`。对内置函数 `CopyTime` 的调用负责获取报价，或者更准确的说，获取柱线时间戳（我们将在关于 [访问时间序列](#) 的章节学习该函数）。

```
ServerTime ServerTimeZone(const string symbol = NULL)
{
    const int year = 365 * 24 * 60 * 60;
    datetime array[];
    if(PRTF(CopyTime(symbol, PERIOD_H1, TimeCurrent() - year, TimeCurrent(), array)) >
    {
        // here we get about 6000 bars in the array
        const int n = ArraySize(array);
        PrintFormat("Got %d H1 bars, ~%d days", n, n / 24);
        // (-V-) loop through H1 bars
        ...
    }
}
```

`CopyTime` 函数接收工作金融工具、H1 时间范围以及去年的日期范围作为参数。代替金融工具的 NULL 值表示脚本放置位置的当前图表交易品种，因此，建议选择有 EURUSD 的窗口。你可能猜到了，PERIOD_H1 常量对应于 H1。我们已经熟悉了 `TimeCurrent` 函数：返回当前最新的已知服务器时间。如果我们从其减去一年中的秒数（该秒数放在 `year` 变量中），我们将得到正好一年前的时期和时间。结果将进入 `array` 中。

为统计每周某跟柱线在特定小时开盘的次数，我们保留了 `hours[24]` 数组。将通过遍历生成的 `array` 来执行计算，即，按从过去到现在的柱线计算。每次迭代时，被查看的该周的开盘时间将存储在 `current` 变量中。当循环结束时，服务器的当前时区将保持在 `current` 中，因为当前周将最后处理。

```

// (-v-) cycle through H1 bars
int hours[24] = {};
int current = 0;
for(int i = 0; i < n; ++i)
{
// (-V-) processing of the i-th bar H1
    ...
}

Print("Week opening hours stats:");
ArrayPrint(hours);

```

在天数循环中，我们将使用来自头文件 *MQL5Book/DateTime.mqh* 的 *datetime* 类（参见 [日期和时间](#)）。

```

// (-v-) processing the i-th bar H1
// find the day of the week of the bar
const ENUM_DAY_OF_WEEK weekday = TimeDayOfWeek(array[i]);
// skip all days except Sunday and Monday
if(weekday > MONDAY) continue;
// analyze the first bar H1 of the next trading week
// find the hour of the first bar after the weekend
current = _TimeHour();
// calculate open hours statistics
hours[current]++;

// skip next 2 days
// (because the statistics for the beginning of this week have already been u
i += 48;

```

建议的算法不是最优的，但不要求我们理解时间序列组织方式的技术细节，这些内容我们尚未学习。

有些周没有设置格式（在节假日之后开始）。如果此情况发生在最后一周，则 *current* 变量将包含异常偏移。这可通过统计来验证：对于得到的小时数，记录的每周“开盘”次数将非常少。在此情况下，在测试脚本中，会直接在日志中显示一条消息。在实践中，应核实前一到两周的标准开盘时间。

```

// (-V-) cycle through H1 bars
...
if(hours[current] <= 52 / 4)
{
// TODO: check for previous weeks
Print("Extraordinary week detected");
}

```

如果经纪商未切换到夏令时，则统计数据中将出现一个最大值，其将包含所有或几乎所有的周。如果经纪商实行时区更改，则在统计中将会有两个高值。

```
// find the most frequent time shift
int max = ArrayMaximum(hours);
// then check if there is another regular shift
hours[max] = 0;
int sub = ArrayMaximum(hours);
```

我们需要确定第二个极高值的重要性 (即区别于可能导致周初开盘时间偏移的随机假期)。为此, 我们评估一年的一个季度 (52 周/4) 的统计数据。如果超过了该限值, 则经纪商支持夏令时。

```
int DST = 0;
if(hours[sub] > 52 / 4)
{
    // basically, DST is supported
    if(current == max || current == sub)
    {
        if(current == MathMin(max, sub))
            DST = fabs(max - sub); // DST is enabled now
    }
}
```

如果当前周的开盘偏移 (存储在当前变量中) 与两个极高值之一重合, 则当前周正常开盘, 可据此推断时区 (这一保护性条件是必要的, 因为我们对于非标准周没有修正, 而是仅会发出一个警告)。

现在一切就绪, 可以确定我们的函数回应: 服务器时区和启用夏令时的标志。

```
current += 2 + DST; // +2 to get offset from UTC
current %= 24;
// timezones are always in the range [UTC-12, UTC+12]
if(current > 12) current = current - 24;
```

由于我们需要从函数返回两个特性 (*current* 和 *DST*), 并且除此之外, 我们可以告知调用代码, 经纪商是否使用夏令时 (即使现在是冬季), 因此声明一个具有所有要求字段的特殊结构体 *ServerTime* 是合理的。

```
struct ServerTime
{
    int offsetGMT; // timezone in seconds relative to UTC/GMT
    int offsetDST; // DST correction in seconds (included in offsetGMT)
    bool supportDST; // DST correction detected in quotes in principle
    string description; // result description
};
```

然后, 在 *ServerTimeZone* 函数中, 我们可以填充和返回由此得到的这样一个结构体。

```
ServerTime st = {};
st.description = StringFormat("Server time offset: UTC%d, including DST%d", cu
st.offsetGMT = -current * 3600;
st.offsetDST = -DST * 3600;
return st;
```

如果因某些原因该函数不能获取到报价, 则返回一个空结构体。

```
ServerTime ServerTimeZone(const string symbol = NULL)
{
    const int year = 365 * 24 * 60 * 60;
    datetime array[];
    if(PRTF(CopyTime(symbol, PERIOD_H1, TimeCurrent() - year, TimeCurrent(), array)) >
    {
        ...
        return st;
    }
    ServerTime empty = {-INT_MAX, -INT_MAX, false};
    return empty;
}
```

我们来以实操方式检查新函数，我们为该新函数在 *OnStart* 中添加以下指令：

```
...
ServerTime st = ServerTimeZone();
Print(st.description);
Print("ServerGMTOffset: ", st.offsetGMT);
Print("ServerTimeDaylightSavings: ", st.offsetDST);
}
```

我们来看看可能的结果。

```
CopyTime(symbol,PERIOD_H1,TimeCurrent()-year,TimeCurrent(),array)=6207 / ok
Got 6207 H1 bars, ~258 days
Week opening hours stats:
52 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Server time offset: UTC+2, including DST+0
ServerGMTOffset: -7200
ServerTimeDaylightSavings: 0
```

根据为 H1 柱线收集的统计数据,该经纪商的周开盘严格在星期一 00:00。因此,真实时区等于 UTC+2,, 并且没有夏季时间修正,即服务器时间必须匹配 EET (UTC+2)。然而在实践中,如同我们在日志第一部分中看到的,服务器上的时间与 GMT 相差 3 小时。

这里我们可以假定我们碰到了一个全年以夏季时间工作的服务器。在此情况下，函数 `ServerTimeZone` 便无法区分修正和“时区”中的额外小时：结果，DST 模式将等于零，而根据服务器报价计算得出的 GMT 时间将比真实时间向右偏移一小时。我们最初假设的“报价从星期天 22:00 点开始”与该服务器的运行模式不符。此类问题应通过经纪商的支持服务进行澄清。

4.7.3 世界时

在 MQL5 中，你可以基于计算机的本地时间及其时区确定全球 GMT (UTC)。

```
datetime TimeGMT()  
datetime TimeGMT(MqlDateTime &dt)
```

该函数返回 *datetime* 格式的 GMT, 从计算机的本机时间开始计数, 并考虑到了冬令时和夏令时的转换。

概括性计算公式:

`TimeGMT() = TimeLocal() + TimeGMTOffset()`

因此，世界时的表示精度取决于本地计算机上时钟的正确设置。理想情况下，取回的值应与服务器已知的值保持一致。

对于基于外部经济新闻的交易策略，使用 GMT 时区的日历是最容易的：这样可以可以通过 *TimeGMT* 跟踪即将发生的事件。要将一个事件与图表上的服务器时间绑定，你应针对服务器时区与 GMT 之间的差值 (*TimeTradeServer() - TimeGMT()*) 对事件进行校正。但请记住，MQL5 有其自己内置的日历。

int TimeGMTOffset()

该函数返回以秒数表示的 GMT 与计算机的本地时间之间的当前差值（基于 Windows 中的时区设置并考虑当前夏令时）。在大多数情况下，时区以相对于 GMT 的整数小时数给出，因此 *TimeGMTOffset* 等于时区乘以 -3600（转换为秒）。例如，在冬季，时区可能等于 UTC + 2，因此偏移量为 -7200，而在夏季可能为 UTC + 3，因此偏移量为 -10800。减号是必需的，因为在将其时间转换为 GMT 时，正时区需要减去上述秒数，而负时区需要加上。

包含 *TimeGMT* 和 *TimeGMTOffset* 用法的脚本见 [上一节](#)。

4.7.4 暂停程序

如之前示例中所示，程序有时候需要定期重复某些操作，按照简单的计划执行或是在先前的尝试失败之后执行。在一个循环中执行这类操作时，建议定期暂停程序以防止过于频繁的请求和对 CPU 的不必要负载，同时可以给外部参与者留出时间来处理它们的工作（例如，如果我们正在等待从另一个程序传来数据，加载报价历史记录等等）。

为此，MQL5 提供了 *Sleep* 函数。本节提供该函数的正式描述，其用法示例将在下一节与用于时间间隔测量的函数一同给出。

void Sleep(int milliseconds)

该函数将 MQL 程序执行暂停指定毫秒数。暂停时间结束后，*Sleep* 调用后的指令将继续被执行。

该函数主要适用于脚本和服务，因为这些类型的程序没有其它的等待方式。

在 EA 交易和指标中，建议使用计时器和 *OnTimer* 事件。在该方案中，MQL 程序将控制权返回给终端，并将在指定间隔后被再次调用。

此外，无法从指标调用 *Sleep* 函数，因为指标是在终端接口线程中执行，其执行的中止将影响图表的渲染。

如果用户在 MQL 程序等待调用完成 *Sleep* 时从终端接口中断 MQL 程序，则会立即（在 100 毫秒内）退出该函数，即暂停比原计划提前结束。这将会设置停止标志 *_StopFlag*（也可以通过函数 *IsStopped* 完成），程序应会尽快并尽可能正确地停止执行。

4.7.5 时间间隔计数器

要检测精确到秒的时间间隔，只需计算使用 *TimeLocal* 获得的两个 *datetime* 值之间的差值。然而，有时候我们需要更高的精度。为此，MQL5 提供了用于获得系统毫秒 (*GetTickCount*, *GetTickCount64*) 或微秒 (*GetMicrosecondCount*) 的计数器。

`uint GetTickCount()`

`ulong GetTickCount64()`

这两个函数返回自操作系统加载以来已经过去的毫秒数。计时精度受限于标准系统计时器（约 10 到 15 毫秒）。要获得更精确的间隔测量值，可使用 *GetMicrosecondCount* 函数。

对于 *GetTickCount* 函数，返回类型 *uint* 预先确定了计数器溢出的时间周期：约 49.7 天。换言之，如果超过该时间段后计算机仍未关闭，则计数又将从 0 开始。

相比之下，*GetTickCount64* 函数返回 *ulong* 值，该计数器在可预见的未来（684'942'417 年）之内将不会溢出。

`ulong GetMicrosecondCount()`

该函数返回自从 MQL 程序启动以来已经过去的微秒数。

该计数器函数和 *Sleep* 的使用示例在 *TimeCount.mq5* 脚本中汇总。

```
void OnStart()
{
    const uint startMs = GetTickCount();
    const ulong startMcs = GetMicrosecondCount();

    // loop for 5 seconds
    while(PRTF(GetTickCount()) < startMs + 5000)
    {
        PRTF(GetMicrosecondCount());
        Sleep(1000);
    }

    PRTF(GetTickCount() - startMs);
    PRTF(GetMicrosecondCount() - startMcs);
}
```

下面是该脚本日志输出的大致内容。

```
GetTickCount()=12912811 / ok
GetMicrosecondCount()=278 / ok
GetTickCount()=12913903 / ok
GetMicrosecondCount()=1089845 / ok
GetTickCount()=12914995 / ok
GetMicrosecondCount()=2182216 / ok
GetTickCount()=12916087 / ok
GetMicrosecondCount()=3273823 / ok
GetTickCount()=12917179 / ok
GetMicrosecondCount()=4365889 / ok
GetTickCount()=12918271 / ok
GetTickCount()-startMs=5460 / ok
GetMicrosecondCount()-startMcs=5458271 / ok
```

4.8 用户交互

程序与“外部世界”的连接始终是双向的，组织这一连接的方式可根据条件分为数据输入和输出的类别。在经典版本中，用户为程序提供某些设置，然后从程序收到结果。如果程序与某些外部程序或服务集成，则原则上讲，输入和输出会绕过用户界面，采用特别交换协议（通过文件、网络、共享内存等）执行。

MQL 程序执行环境允许你以多种方式组织与 MetaTrader 5 用户的交互。

在本章中，我们将了解其中最简单的方式：在日志或图形中显示消息，显示简单对话框以及发出声音提示。

别忘了，MQL 程序中的数据输入标准方式是 [输入变量](#)。然而，输入变量只能在程序初始化时设置。通过设置对话框更改程序特性表示以新值“重启”程序（稍后我们将探讨与一种 MQL 程序类型关联的某些特殊情况，正是因此，*restart* 才加了双引号）。

更灵活的交互关系意味着无需停止程序即可控制程序的行为。在简单情况下，我们在后文讨论的 MessageBox 对话框等即可满足要求，但对于大多数实践应用而言，这并不够。

因此，在本书的后续部分中，我们将大幅扩展用于实现用户界面的工具，并学习如何基于接口对象创建交互式程序，在 [指标](#) 或 [资源](#) 中显示图形信息，向用户移动设备推送通知等等。

4.8.1 日志消息

日志是将有关程序运行的当前信息告知用户的最常用的方式。这些信息可能是常规完成状态、长计算过程期间的进度指示，或者用于查找和复制错误的调试数据。

遗憾的是，任何编程人员在写代码时都难免犯错。因此，开发人员通常都尽量留下所谓的“面包屑痕迹”：记录程序执行的主要阶段（至少记录函数调用顺序）。

我们已经熟悉了一个日志函数，分别是 *Print* 和 *PrintFormat*。我们在前面章节的示例中使用过。我们不得不以简化模式提前“将它们投入使用”，因为少了它们几乎是无法进行的。

原则上，一次函数调用生成一条记录。然而，如果在输出字符串中碰到一个换行符字符（'\n'），则其将会把信息分为两个部分。

请注意，所有 *Print* 和 *PrintFormat* 调用都会转换为 Toolbox 窗口的 *Experts* 选项卡上的日志条目。虽然该选项卡名为 *Experts*，但是它收集的是所有打印指令的结果，无论何种 MQL 程序类型。

日志存储在根据“一天一文件”原则组织的文件中：它们的名称为 YYYYMMDD.log（Y 为年，M 为月，D 为天）。文件位于 `<data directory>/MQL5/Logs`（不要与终端系统日志文件夹 `<data directory>/Logs` 混淆了）。

请注意，在批量日志记录期间（如果 *Print* 函数调用在短时间内生成大量信息），终端在窗口中仅显示一部分条目。这是为了优化性能。此外，用户在任何情况下都无法实时查看所有消息。要看到完整版本的日志，你需要运行上下文菜单中的 *View* 命令。结果，将会打开一个包含日志的窗口。

切记，日志信息在写入到磁盘时会进行缓存，也就是说，它以懒模式大块写入到文件，这就是为什么原则上在任何给定时间，日志文件均不包含最近条目的原因（虽然它们在窗口中可见）。要发起将缓存刷新到磁盘的操作，可以在日志上下文菜单中运行命令 *View* 或 *Open*。

每个日志条目前面都是精确到最近毫秒数的时间，以及生成或导致该消息的程序的名称（及其图形）。

void Print(argument, ...)

该函数将一个或多个值以一行（如果输出数据不包含字符 '\n'）打印到“专家”日志。

自变量可以是任何 [内置类型](#)。它们以逗号分隔。参数的数量不能超过 64。原型中的省略号表示参数数量是可变的，但 MQL5 不允许你以类似特性描述你自己的函数：仅一些内置 API 函数可以具有可变数量的参数（具体来说，就是 [StringFormat](#)、[Print](#)、[PrintFormat](#)，以及 [Comment](#)）。

对于结构体和类，你应实现一种内置打印方法，或者单独显示它们的字段。

此外，该函数不能够处理数组。你可以逐元素显示它们，或者使用函数 [ArrayPrint](#)。

double 类型的值由该函数以 16 位有效数位的精度输出（尾数和小数部分的总和）。一个数字可以以传统或科学格式（采用指数）显示，取更紧凑的方式。**float** 类型的值以 7 个小数位的精度显示。要以不同的精度显示实数或者要明确指定格式，必须使用 [PrintFormat](#) 函数。

bool 类型的值输出为字符串 “true” 或 “false”。

显示的日期包含日和時間，并取最大精度（精确到秒）以 “YYYY.MM.DD hh:mm:ss” 格式显示。要以不同的格式显示日期，使用 [TimeToString](#) 函数（参见 [日期和时间](#) 章节）。

枚举值显示为整数。要显示元素名称，使用 [EnumToString](#) 函数（参见 [枚举](#) 章节）。

单字节和双字节字符也输出为整数。要将符号显示为字符或字母，使用函数 [CharToString](#) 或 [ShortToString](#)，参见章节 [处理符号和代码页](#)）。

color 类型的值显示字符串，其中包含数字三元组，分别表示每种颜色分量（“R, G, B”）强度，或者显示为颜色名称（如果该颜色出现在颜色集合中）。

有关将不同类型的值转换为字符串的更多信息，参见：[内置类型数据转换](#)（尤其是以下章节 [数字转换为字符串](#) 以及 [相反转换](#)、[日期和时间](#)、[颜色](#)）。

以单传递模式在策略测试程序中工作时（[测试 EA 交易或指标](#)），[Print](#) 函数的结果输出到测试代理日志中。

以 [优化](#) 模式在策略测试程序中工作时，出于性能原因会抑制日志记录，因此 [Print](#) 函数没有直观效果。但是给定为自变量的所有表达式均被求值。

所有自变量在转换为字符串表示后被连接为一个没有定界符的公用字符串。如果需要，这些字符必须显式写在自变量列表中。例如，

```
int x;
bool y;
datetime z;
...
Print(x, ", ", y, ", ", z);
```

其中，3 个变量被记录到日志，以逗号分隔。如果没有中间字面量 “, ”，变量的值将会在日志条目中粘连在一起。

从本书最初章节开始，就有很多应用 [Print](#) 的情况（例如，[第一个程序](#)、[赋值和初始化](#)，[表达式和数组](#) 以及其它章节）。

作为一种新的使用 *Print* 的方式，我们将实现一个简单类，该简单类将允许你显示一系列任意值而无需在每个相邻值之间指定一个分隔符。我们使用 ‘<<’ 运算符重载法，类似于在 C++ I/O 流中使用的方法 (std::cout)。

类定义将放在单独的头文件 *OutputStream.mqh* 中。下面显示了简化形式的类。

```
class OutputStream
{
protected:
    ushort delimiter;
    string line;

    // add the next argument, separated by a separator (if any)
    void appendWithDelimiter(const string v)
    {
        line += v;
        if(delimiter != 0)
        {
            line += ShortToString(delimiter);
        }
    }

public:
    OutputStream(ushort d = 0): delimiter(d) { }

    template<typename T>
    OutputStream *operator<<(const T v)
    {
        appendWithDelimiter((string)v);
        return &this;
    }

    OutputStream *operator<<(OutputStream &self)
    {
        if(&this == &self)
        {
            print(line); // output of the composed string
            line = NULL;
        }
        return &this;
    }
};
```

它的意义是在字符串变量 *line* 中积累使用 ‘<<’ 运算符传递的任何自变量的字符串表示。如果在类构造函数中指定了分隔符，则其将被自动插入在自变量之间。由于重载运算符返回一个对象指针，我们可以连续传递一系列自变量：

```
OutputStream out(',',');
out << x << y << z << out;
```

作为数据收集结束的属性，以及为了让 *line* 内容实际输出到日志，使用对象本身的同一运算符的重载。

真实类一定程度上更为复杂。具体来说，真实类不仅支持设置分隔符，还支持设置实数显示精度以及用于选择日期和时间值字段的标志。此外，该类支持字符形式（而不是整数代码形式）的 *ushort* 字符打印、简化的数组输出（输出到单独字符串）、十六进制格式的单值颜色（而不是以逗号分隔的数字三元组，由于逗号通常用作分隔符字符，因此在日志中的颜色分量看起来象是 3 个不同的变量）。

OutputStream.mq5 脚本中提供该类的用法演示。

```
void OnStart()
{
    OutputStream os(5, ',');

    bool b = true;
    datetime dt = TimeCurrent();
    color clr = C'127, 128, 129';
    int array[] = {100, 0, -100};
    os << M_PI << "text" << clrBlue << b << array << dt << clr << '@' << os;

    /*
       output example

       3.14159,text,clrBlue,true
       [100,0,-100]
       2021.09.07 17:38,clr7F8081,@
    */
}
```

void PrintFormat(const string format, ...)≡ void printf(const string format, ...)

该函数基于指定格式字符串记录一系列自变量。*format* 参数不仅提供了“按原样”显示的自由文本输出字符串模板，还可以包含描述如何设置特定自变量格式的转义系列。

包括格式字符串在内的参数总数不能超过 64。参数类型限制与 *print* 函数类似。

PrintFormat 工作原理和格式化原则与针对 *StringFormat* 函数的描述相同（参见章节 [数据到字符串的通用格式化输出](#)）。唯一差异是 *StringFormat* 将形成的字符串返回到调用代码，而 *print format* 将其发送到日志。我们可以说，*PrintFormat* 具有以下条件性等效内容：

```
Print(StringFormat(<list of arguments as is, including format>))
```

除了全称 *PrintFormat*，你还可以使用短别名 *printf*。

像 *Print* 函数一样，在测试程序中以优化模式工作时，*PrintFormat* 有一些特殊特性：其日志输出被禁止以提高性能。

我们已在很多章节遇到了使用 *PrintFormat* 的脚本，例如，[返回转换](#)、[颜色](#)、[动态数组](#)、[文件描述符管理](#)、[获取全局变量列表](#)。

4.8.2 警报

在本节中，信号指 *Alert* 函数向终端用户发出警告。

“警报”一词在 MetaTrader 5 中有多个含义。有 2 种语境：

- **Toolbox** 面板中的 **Alerts** 选项卡中的用户可配置（手动）警报。使用这种警报，你可以跟踪超过价格、交易量或时间设定值的简单条件的触发情况，并以各种方式发出通知。
- 由 **Alert** 函数从 MQL 代码生成的程序“警报”。它们与前一种警报无任何关系。

void Alert(argument, ...)

该函数在非模态对话框中显示一条消息，伴随标准声音信号（根据在终端中 **Options** 对话框中 **Events** 选项卡上的选择）。如果该窗口被隐藏，其将显示在主终端窗口上面（这样在使用主窗口时即可将其关闭、最小化或移开）。该消息也被添加到“专家”日志，标记为“警报”。

在 MetaTrader 5 界面中，没有用于手动打开警报窗口的命令（如果先前被关闭）。要想重新看到警告列表（纯形式，无需筛选日志），你将需要以某种方式生成一个新的信号。

该函数的自变量传递方式、信息显示以及一般原则，与针对 **Print** 函数的规定完全相同。

有关 **Alert** 函数的演示及截屏，见第一章中 **数据输出** 一节的介绍性欢迎示例。

如果需要用户特别注意显示的信息，应使用 **Alert** 而不是 **Print**。但不应滥用，因为窗口频繁出现会妨碍用户的工作，迫使他们忽略消息或停止 MQL 程序。在你的程序中提供一种算法以限制可能的消息生成频率。

4.8.3 在图表窗口中显示消息

我们在前面章节中已经知道，MQL5 支持将消息输出到日志或警报窗口。第一种方法主要用于技术信息，不能保证用户将会注意到消息（因为日志窗口可能被隐藏）。同时，如果使用第二种方法显示经常变化的程序状态，可能看起来干扰性过大。一种折衷的选择是 **Comment** 函数。

void Comment(argument, ...)

该函数在图表左上角显示一条由传递的所有自变量组成的消息。该消息会一直保留，直至该程序或其它程序将其消除或以另一条消息替换。

该窗口可能仅包含一条注释：每次调用 **Comment** 时，旧内容（如有）被新内容替换。

要清理注释，只需以空字符串调用该函数：**Comment("")**。

参数的数量不得超过 64。仅支持内置类型的自变量。根据传递值形成结果字符串的概念类似于函数 **Print**。

显示的消息的总长度限制为 2045 个字符。如果超过该限值，则行末将被截断。

注释的当前内容是图表的字符串特性之一，可以通过调用 **ChartGetString(NULL, CHART_COMMENT)** 函数找到。我们将在单独 **章节** 中探讨图表的这一特性及其它特性（不仅是字符串特性）。

与在 **Print**、**PrintFormat** 以及 **Alert** 函数中一样，字符串自变量可能包含换行符字符（`\n` 或 `\r\n`），这会将消息被拆分为适当数量的字符串。对于 **Comment**，这是显示多行消息的唯一方法。如果使用 **print** 和 **signal** 函数，可以通过多次调用来实现相同效果，但使用 **Comment** 则无法实现，因为每次调用将以新字符串替换旧字符串。

关于 **Comment** 函数的工作示例，可参见第一章的 **数据输出** 一节中包含欢迎脚本的窗口截图。

此外，我们将开发一个类和一些简化函数以基于给定大小的环形缓冲区显示多行注释。本书中包含了测试脚本（**OutputComment.mqh**）和具有类代码的头文件（**Comments.mqh**）。

```
class Comments
{
    const int capacity; // maximum number of strings
    const bool reverse; // display order (new ones on top if true)
    string lines[];      // text buffer
    int cursor;          // where to put the next string
    int size;            // actual number of strings saved

public:
    Comments(const int limit = N_LINES, const bool r = false):
        capacity(limit), reverse(r), cursor(0), size(0)
    {
        ArrayResize(lines, capacity);
    }

    void add(const string line);
    void clear();
};
```

主要工作由 *add* 方法完成。

```

void Comments::add(const string line)
{
    ...
    // if the passed text contains multiple strings,
    // split it into elements by newline character
    string inputs[];
    const int n = StringSplit(line, '\n', inputs);

    // add all new elements to the ring buffer
    // overwriting the oldest entries at the cursor
    // cursor increases by capacity module (reset to 0 on overflow)
    for(int i = 0; i < n; ++i)
    {
        lines[cursor] = inputs[reverse ? n - i - 1 : i];
        cursor = (cursor + 1) % capacity;
        if(size < capacity) size++;
    }
    // concatenate all text entries in forward or reverse order
    // gluing with newline characters
    string result = "";
    for(int i = 0, k = size == capacity ? cursor % capacity : 0;
        i < size; ++i, k = ++k % capacity)
    {
        if(reverse)
        {
            result = lines[k] + "\n" + result;
        }
        else
        {
            result += lines[k] + "\n";
        }
    }

    // output the result
    Comment(result);
}

```

如果必要，注释以及文本缓冲区可通过 *clear* 方法或通过调用 *add(NULL)* 进行清理。

```

void Comments::clear()
{
    Comment("");
    cursor = 0;
    size = 0;
}

```

有了这一个类，你可以以所需的缓冲区容量和输出方向定义对象，然后使用其方法。

```
Comments c(30/*capacity*/, true/*order*/);
```

```
void function()
{
    ...
    c.add("123");
}
```

但为简化常规函数风格中的注释生成 (类似 *Comment* 函数), 我们实现了若干个辅助函数。

```
void MultiComment(const string line = NULL)
{
    static Comments com(N_LINES, true);
    com.add(line);
}
```

```
void ChronoComment(const string line = NULL)
{
    static Comments com(N_LINES, false);
    com.add(line);
}
```

区别仅在于缓冲区输出的方向。*MultiComment* 以逆向时间顺序显示行, 即最新内容显示在顶部, 就象在公告牌上一样。建议将该函数用于长期持续显示信息并保留历史记录的场景。

ChronoComment 以正向顺序显示行, 即新行被添加到底部。建议将该函数用于多行消息的批量输出。

缓冲区行数默认为 `N_LINES` (10)。如果你在包括头文件之前以一个不同的值定义该宏, 则将调整大小。

测试脚本包含一个循环, 其中定期生成消息。

```
void OnStart()
{
    for(int i = 0; i < 50 && !IsStopped(); ++i)
    {
        if((i + 1) % 10 == 0) MultiComment();
        MultiComment("Line " + (string)i + ((i % 3 == 0) ? "\n (details)" : ""));
        Sleep(1000);
    }
    MultiComment();
}
```

每十次迭代清理一次注释。每三次迭代生成一条两行消息 (其余消息为一行)。有 1 秒的延迟, 可让你观察动态效果。

下面是该脚本正在运行时的窗口示例 (‘新消息在上’模式)。



图表上的多行注释

在一个注释中显示多行信息的能力十分有限。如果你需要按列组织数据输出，以颜色或不同字体来突出显示，实现鼠标单击反应或在图表上的任意位置，你应使用图形 [对象](#)。

4.8.4 消息对话框

MQL5 API 提供了 **MessageBox** 函数，用于以交互方式提示用户确认操作或选择用于处理特定情况的选项。

int MessageBox(const string message, const string caption = NULL, int flags = 0)

该函数打开一个具有给定消息 (*message*)、标题 (*caption*) 和设置 (*flags*) 的无模式对话框。该窗口保持在主终端窗口上面可见，直至用户单击可用按钮之一将其关闭 (参见下文)。

该消息同时也以 “Message” 标记显示在 “专家” 日志中。

如果 *caption* 参数为 NULL，则 MQL 程序的名称被用作标题。

flags 参数必须包含一个位标志与一个 OR (|) 运算的组合。支持标志的总集合分为 3 组，分别定义了：

- ⌚ 对话框中的一组按钮
- ⌚ 对话框中的图标图像
- ⌚ 默认选中的活动按钮

下表列出了用于定义对话框按钮的常量和标志值。

常量	值	说明
MB_OK	0x0000	1 个 “确定”按钮（默认）
MB_OKCANCEL	0x0001	2 个按钮：“确定”和“取消”
MB_ABORTRETRYIGNORE	0x0002	3 个按钮：“终止”、“重试”和“忽略”
MB_YESNOCANCEL	0x0003	3 个按钮：“是”、“否”和“取消”
MB_YESNO	0x0004	2 个按钮：“是”和“否”
MB_RETRYCANCEL	0x0005	2 个按钮：“重试”和“取消”
MB_CANCELTRYCONTINUE	0x0006	3 个按钮：“取消”、“再试一次”和“继续”

下表列出了可用图像（显示在消息左边）

常量	值	说明
MB_ICONSTOP MB_ICONERROR MB_ICONHAND	0x0010	“停止”符号 
MB_ICONQUESTION	0x0020	问号 
MB_ICONEXCLAMATION MB_ICONWARNING	0x0030	感叹号 
MB_ICONINFORMATION MB_ICONASTERISK	0x0040	信息符号 

所有图标取决于操作系统版本。你的计算机上显示的示例可能不同。

以下值保留用于选择活动按钮。

常量	值	说明
MB_DEFBUTTON1	0x0000	第一个按钮，即默认按钮（如果未选择任何其它常量）
MB_DEFBUTTON2	0x0100	第二个按钮
MB_DEFBUTTON3	0x0200	第三个按钮
MB_DEFBUTTON4	0x0300	第四个按钮

你可能会有疑问，如果上面的常量只允许你设置不超过三个按钮，那么这第四个按钮是怎么回事。事实是，在标志当中还有 MB_HELP（0x00004000）。该标志指示在对话框中显示“帮助”按钮。这样，如果有三个主按钮，“帮助”按钮会变成一排中的第四个。然而，与其它按钮不同，单击“帮助”按钮不会关闭对话框。根据 Windows 标准，帮助文件可与程序关联，当按下“帮助”按钮时，应该打开必要的帮助信息。然而，MQL 程序目前不支持这一技术。

该函数根据对话框被关闭的方式 (按了哪个按钮) 而返回其中一个预定义值。

常量	值	说明
IDOK	1	确定”按钮
IDCANCEL	2	取消”按钮
IDABORT	3	终止”按钮
IDRETRY	4	重试”按钮
IDIGNORE	5	忽略”按钮
IDYES	6	是”按钮
IDNO	7	否”按钮
IDTRYAGAIN	10	再试一次”按钮
IDCONTINUE	11	继续”按钮

如果消息框有“取消”按钮，则在用户按 `ESC` 键时 (除了“取消”按钮之外)，该函数返回 `IDCANCEL`。如果消息框没有“取消”按钮，则按 `ESC` 无效果。

调用 `MessageBox` 会暂停当前 MQL 程序的执行，直至用户关闭对话框。因此，`MessageBox` 在 指标 中禁止使用，因为指标是在终端的接口线程中执行，等待用户回应将会拖慢图表的更新。

同样，该函数也不能用在 服务 中，因为它们与用户界面无关联，而其它类型的 MQL 程序是在图表上下文中执行。

在策略测试程序中运行时，`MessageBox` 函数没有效果并返回 0。

从函数调用获得结果之后，你可以以你想要的方式处理结果，例如：

```
int result = MessageBox("Continue?", NULL, MB_YESNOCANCEL);
// use 'switch' or 'if' as needed
switch(result)
{
case IDYES:
    // ...
    break;
case IDNO:
    // ...
    break;
case IDCANCEL:
    // ...
    break;
}
```

`MessageBox` 函数可以使用 `OutputMessage.mq5` 脚本测试，用户可使用输入变量选择对话框的参数，并实际观察其效果。

按钮、图标和默认选定按钮的设置组以及返回代码在特殊枚举中描述：ENUM_MB_BUTTONS、ENUM_MB_ICONS、ENUM_MB_DEFAULT、ENUM_MB_RESULT。这通过下拉列表提供可视化输入，并使用 *EnumToString* 简化了它们向字符串的转换。

例如，下面是前两个枚举的定义方式。

```
enum ENUM_MB_BUTTONS
{
    _OK = MB_OK, // Ok
    _OK_CANCEL = MB_OKCANCEL, // Ok | Cancel
    _ABORT_RETRY_IGNORE = MB_ABORTRETRYIGNORE, // Abort | Retry | Ignore
    _YES_NO_CANCEL = MB_YESNOCANCEL, // Yes | No | Cancel
    _YES_NO = MB_YESNO, // Yes | No
    _RETRY_CANCEL = MB_RETRYCANCEL, // Retry | Cancel
    _CANCEL_TRYAGAIN_CONTINUE = MB_CANCELTRYCONTINUE, // Cancel | Try Again | Continue
};

enum ENUM_MB_ICONS
{
    _ICON_NONE = 0, // None
    _ICON_QUESTION = MB_ICONQUESTION, // Question
    _ICON_INFORMATION_ASTERISK = MB_ICONINFORMATION, // Information (Asterisk)
    _ICON_WARNING_EXCLAMATION = MB_ICONWARNING, // Warning (Exclamation)
    _ICON_ERROR_STOP_HAND = MB_ICONERROR, // Error (Stop, Hand)
};
```

其余可以在源代码中找到。

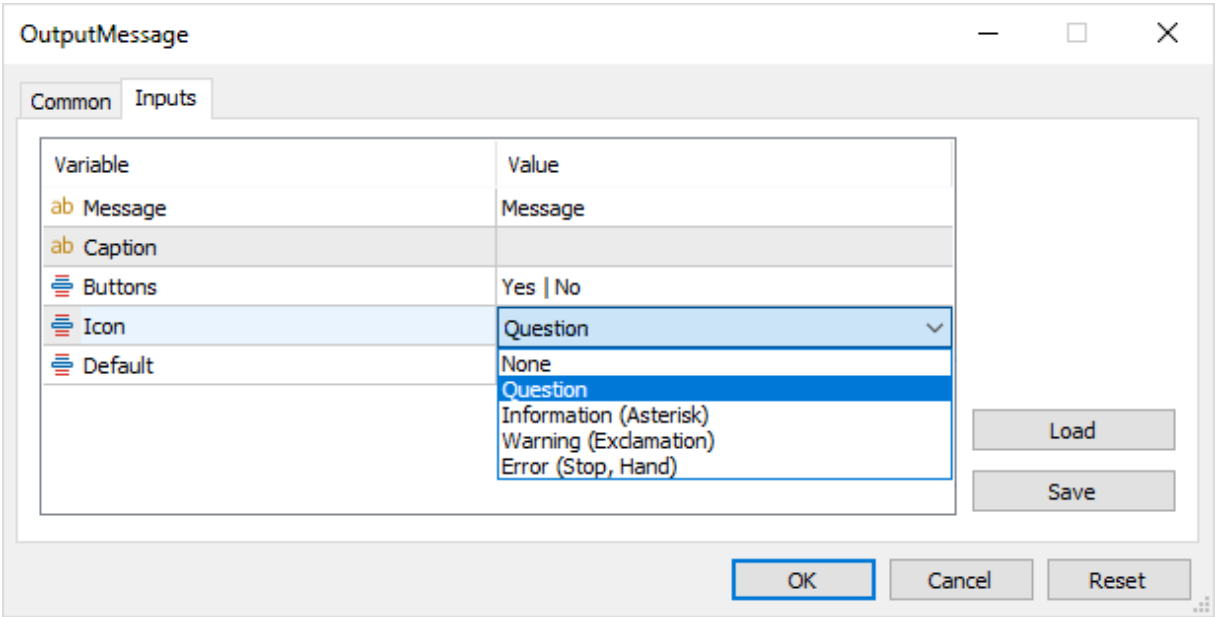
然后它们被用作输入变量类型（元素注释在用户界面中提供更为用户友好的呈现方式）。

```
input string Message = "Message";
input string Caption = "";
input ENUM_MB_BUTTONS Buttons = _OK;
input ENUM_MB_ICONS Icon = _ICON_NONE;
input ENUM_MB_DEFAULT Default = _DEF_BUTTON1;

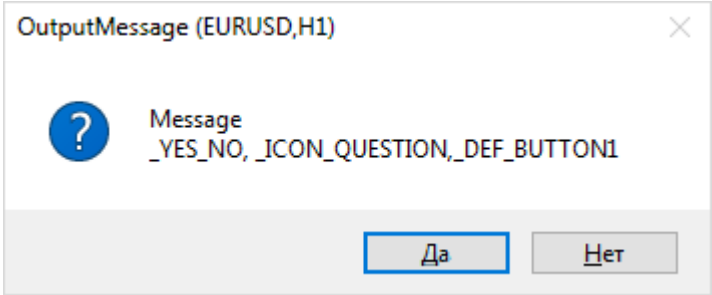
void OnStart()
{
    const string text = Message + "\n"
        + EnumToString(Buttons) + ", "
        + EnumToString(Icon) + ", "
        + EnumToString(Default);
    ENUM_MB_RESULT result = (ENUM_MB_RESULT)
        MessageBox(text, StringLen(Caption) ? Caption : NULL, Buttons | Icon | Default)
    Print(EnumToString(result));
}
```

该脚本在窗口中显示指定消息以及指定对话框设置。对话框结果显示在日志中。

下图显示了选项选择界面和生成对话框的截屏。



窗口特性”对话框



接收到的消息”对话框

4.8.5 声音警报

为处理声音，MQL5 API提供了一个函数：*PlaySound*。

`bool PlaySound(const string soundfile)`

该函数播放 *wav* 格式的指定声音文件。

如果指定了文件名称而没有路径（如“Ring.wav”），则该文件必须位于终端安装目录中的 *Sounds* 文件夹中。如果需要，你可以整理 *Sounds* 文件夹内的子文件夹。在这种情况下，*soundfile* 参数中的文件名应在前面带有一个相对路径。例如，“Example/Ring.wav”指终端安装目录中的文件夹和文件 *Sounds/Example/Ring.wav*。

此外，你可以使用位于终端的数据目录中的任何其它 MQL5 子文件夹中的声音文件。该路径必须在前面有一个前导斜杠（单斜杠 '/' 或双反斜杠 '\\'），这是用于文件系统中相邻文件夹层级之间的定界符。例如，如果声音文件 *Demo.wav* 位于 *terminal_data_directory/MQL5/Files* 中，则在 *PlaySound* 调用中，我们将写入路径“/Files/Demo.wav”。

以 NULL 参数调用该函数将会停止声音播放。当旧文件正在播放时以新文件调用函数将会导致旧文件播放中断，新文件开始播放。

除了位于文件系统中的文件之外，还可以将指向资源（嵌入 MQL 程序中的数据块）的路径传递给该函数。具体说，开发人员可在编译时，根据沙盒内本地可用的文件创建声音资源。所有资源均位于 ex5 文件中，这确保用户拥有这些资源，并简化了程序作为单个模块的分发。

有关资源（不仅包括声音，还有图像、任意二进制文件和文本数据以及独立程序（指标））的所有使用方法的详细描述，可参见本书第七章中的对应章节。

如果找到文件，**PlaySound** 函数返回 **true**，否则返回 **false**。请注意，即使该文件不是音频文件而无法播放，该函数也将返回 **true**。

声音回放与后续程序指令的执行并行异步执行。换言之，该函数在调用后立即将控制权交回给调用代码，而不会等待音频效果完成。

在策略测试程序中，不执行 **PlaySound** 函数。

OutputSound.mq5 脚本允许你测试该函数的操作。

```
void OnStart()
{
    PRTF(PlaySound("new.txt"));
    PRTF(PlaySound("abracadabra.wav"));
    const uint start = GetTickCount();
    PRTF(PlaySound("request.wav"));
    PRTF(GetTickCount() - start);
}
```

程序尝试播放多个文件。文件 “new.txt” 存在（特意为测试创建），文件 “abracadabra.wav” 不存在，“request.wav” 文件包括在 MetaTrader 5 的标准分发中。通过两次调用 **GetTickCount** 来测量上次函数调用时间。

运行该脚本后，我们得到以下日志条目：

```
PlaySound(new.txt)=true / ok
PlaySound(abracadabra.wav)=false / FILE_NOT_EXIST(5019)
PlaySound(request.wav)=true / ok
GetTickCount()-start=0 / ok
```

找到了 “new.txt” 文件，因此函数返回 **true**，尽管该文件不能播放声音。对第二个不存在的文件的调用返回了 **false**，并且 **_LastError** 中的错误代码为 5019 (FILE_NOT_EXIST)。最后，最后一个文件（假设它存在）的播放从各方面来说应会成功：函数将返回 **true**，终端将播放音频。调用处理时间几乎为零（不考虑声音持续时间）。

4.9 MQL 程序执行环境

我们知道，MQL 程序的源文本在编译为 **ex5** 格式的二进制可执行代码之后，即可在终端或测试代理中运行。因此，终端或测试程序提供了 MQL 程序在其中“居住”的公用环境。

别忘了，内置测试程序仅支持 2 种类型的 MQL 程序：EA 交易和指标。我们将在本书第五章中详述 MQL 程序的类型以及它们的功能。同时，在本章中，我们将着眼于对所有类型的程序通用的 MQL5 API 函数，可以通过这些函数来分析执行环境以及一定程度上控制环境。

大多数环境特性可通过以下函数进行只读访问：**TerminalInfoInteger**、**TerminalInfoDouble**、**TerminalInfoString**、**MQLInfoInteger** 和 **MQLInfoString**。从函数名称可以知道，每个函数返回某种类型的值。这样的架构导致在一个函数中组合特性的应用含义可能有很大区别。另一种分组可以

通过在 MQL5 中实施你自己的对象层来提供（稍后将在关于使用 [特性绑定到程序环境](#) 的章节提供示例）。

指定的函数集具有明确的逻辑划分，分为一般终端特性（带“Terminal”前缀）和单独 MQL 程序特性（带“MQL”前缀）。然而，在很多情况下，需要联合分析终端和程序的类似特性。例如，使用 DLL 或者执行交易操作的权限既发放给整个终端，也发放给特定程序。所以从整体上来综合考虑这些函数更加合理。

某些仅与错误代码关联的环境特性是可写入的，尤其是重置先前错误（*ResetLastError*）以及设置用户错误（*SetUserError*）。

此外，在本章中，我们将了解用于在程序中关闭终端的函数（*TerminalClose*，*SetReturnError*）以及在调试程序中暂停程序的函数（*Debug Break*）。

4.9.1 获取终端和程序特性的总列表

可用于获取环境特性的内置函数采用普通方法：每种特定类型的特性被组合到一个单独函数中，该单独函数具有一个指定所请求特性的自变量。定义了一些用于识别特性的枚举：每个元素描述一个特性。

我们将在下面看到，该方法通常用于 MQL5 API 以及其它领域中，包括应用程序领域。尤其是类似的函数集用于获取 [交易账户](#) 和 [金融工具](#) 的特性。

三个简单类型的特性（*int*、*double* 和 *string*）即足以描述环境。然而，不仅使用 *int* 类型的值呈现了整数特性，还有逻辑标志（尤其是权限/禁止、网络连接存在性等）以及其它内置枚举（如 MQL 程序类型以及许可证类型）。

根据终端特性和特定 MQL 程序特性的条件性划分，有以下描述环境的函数。

```
int MQLInfoInteger(ENUM_MQL_INFO_INTEGER p)
int TerminalInfoInteger(ENUM_TERMINAL_INFO_INTEGER p)
double TerminalInfoDouble(ENUM_TERMINAL_INFO_DOUBLE p)
string MQLInfoString(ENUM_MQL_INFO_STRING p)
string TerminalInfoString(ENUM_TERMINAL_INFO_STRING p)
```

这些原型将值类型映射到枚举类型。例如，*int* 类型的终端特性在 `ENUM_TERMINAL_INFO_INTEGER` 中概括，而其 *double* 类型的特性列在 `ENUM_TERMINAL_INFO_DOUBLE` 中。可用枚举及其元素的列表参见关于 [终端特性](#) 和 [MQL 程序](#) 章节。

在以下章节中，我们将了解基于其用途而分组的所有特性。但在这里我们先探讨获取所有现有特性以及它们的值的总列表的问题。通常来说这是有必要的，可识别 MQL 程序在特定终端实例上的运行“瓶颈”或特性。一种常见情况是一个 MQL 程序在一台计算机上可正常运行，但在另一台计算机上完全无法运行，或者能运行但表现出某些问题。

特性列表随着平台开发而持续更新，因此最好不要基于源代码中的固定列表作出请求，而是自动请求。

在 [枚举](#) 一节中，我们创建了模板函数 *EnumToArray* 用于获取枚举元素的完整列表（*EnumToArray.mqh* 文件）。同样在该章节中，我们介绍了脚本 *ConversionEnum.mq5*，该脚本使用指定的头文件。在该脚本中，实施了辅助函数 *process*，该函数接受具有枚举元素的数组，并将它们输出到日志。我们将把取得的这些进展作为进一步改进的起点。

我们需要修改 *process* 函数，使得我们不仅能够获取特定枚举的元素的列表，而且能够使用内置特性函数之一查询对应特性。

我们赋予新版本脚本一个名称：*Environment.mq5*。

由于环境的特性分散在若干个不同的函数中（在本例中，五个函数），你需要学会如何向新版本的函数 *process* 传递需要的内置函数的指针（参见 [函数指针 \(typedef\)](#) 章节）。然而，MQL5 不允许将一个内置函数的地址指派给一个函数指针。只能通过 在 MQL5 中实施的一个应用函数完成。因此，我们将创建包装器函数。例如：

```
int _MQLInfoInteger(const ENUM_MQL_INFO_INTEGER p)
{
    return MQLInfoInteger(p);
}
// example of pointer type description
typedef int (*IntFuncPtr)(const ENUM_MQL_INFO_INTEGER property);
// initialization of pointer variables
IntFuncPtr ptr1 = _MQLInfoInteger; // ok
IntFuncPtr ptr2 = MQLInfoInteger;  // compilation error
```

上面显示了 *MQLInfoInteger* 的一个“分身”（很明显，它应具有一个不同但最好类似的名称）。其它函数以类似方式“包装”。总共将有五个。

如果在旧版本的 *process* 中仅有一个指定一个枚举的模板参数，则在新版本中我们还需要传递返回值的类型（因为 MQL5 不“理解”枚举名称中的字词）：即使结尾的“INTEGER”出现在名称 `ENUM_MQL_INFO_INTEGER` 中，编译器也不能够将它与类型 *int* 关联）。

然而，除了链接返回值的类型和枚举，我们还需要以某种方式将恰当的包装器函数（我们早前定义的五個函数中的一个）的指针传递给 *process* 函数。毕竟，编译器本身不能够通过例如一个 `ENUM_MQL_INFO_INTEGER` 类型的自变量确定 *MQLInfoInteger* 需要被调用。

为解决这个问题，创建了一个将所有三个因素组合起来的特殊模板结构。

```
template<typename E, typename R>
struct Binding
{
public:
    typedef R (*FuncPtr)(const E property);
    const FuncPtr f;
    Binding(FuncPtr p): f(p) { }
};
```

两个模板参数允许你以所需的结果和输入参数的组合来指定函数指针（*FuncPtr*）的类型。该结构体实例具有用于该新定义类型的指针的 *f* 字段。

现在，一个新版本的 *process* 函数可描述如下。

```

template<typename E, typename R>
void process(Binding<E, R> &b)
{
    E e = (E)0; // turn off the warning about the lack of initialization
    int array[];
    // get a list of enum elements into an array
    int n = EnumToArray(e, array, 0, USHORT_MAX);
    Print(typename(E), " Count=", n);
    ResetLastError();
    // display the name and value for each element,
    // obtained by calling a pointer in the Binding structure
    for(int i = 0; i < n; ++i)
    {
        e = (E)array[i];
        R r = b.f(e); // call the function, then parse _LastError
        const int snapshot = _LastError;
        PrintFormat("% 3d %s=%s", i, EnumToString(e), (string)r +
            (snapshot != 0 ? E2S(snapshot) + " (" + (string)snapshot + ")" : ""));
        ResetLastError();
    }
}

```

输入自变量为 *Binding* 结构体。它包含用于获取特性的特定函数的指针（该字段将由调用代码填充）。

该版本的算法将系列号、特性标识符及其值记入日志。同样需注意的是，每个条目中的第一个数字将包含枚举中元素的序数，而不是值（值可以间隔赋予给元素）。或者你可以在 *print format* 指令中添加一个“纯形式”的变量 *e* 的输出。

此外，还可以修改该过程，使之将生成的特性值收集进一个数组（或其它容器，如映射），并“向外”返回这些特性值。

若在 *print format* 指令中直接引用函数指针并结合 *_LastError* 错误代码分析，可能会造成潜在错误。问题在于，这样一来，表达式中的函数自变量（参见 *参数和自变量* 章节）和操作数（参见 *基本概念* 章节）的求值顺序未定义。因此，当在调用 *_LastError* 的同一行中调用指针时，编译器的执行顺序可能会出错。结果，我们将看到一个不相关的错误代码（例如，来自一个先前函数调用的错误代码）。

但不仅如此。在表达式的求值中，如果任何运算失败，内置变量 *_LastError* 可能在任何地方更改其值。尤其是如果一个值作为不在枚举范围内自变量被传递，则函数 *EnumToString* 可能产生错误代码。在这个代码片段中，我们不存在此问题，因为我们的函数 *EnumToArray* 返回一个只包含校验过的（有效）枚举元素的数组。然而，在一般情况下，在任何“复合”指令中，可能很多情况下 *_LastError* 会被更改。对此，最好在我们感兴趣的目标操作（在这里是由指针进行的函数调用）完成后立即修复错误代码，将其保存到一个中间变量 *snapshot*。

我们回到主要问题。我们终于可以组织对新函数 *process* 的调用，从而获取软件环境的各种特性。

```

void OnStart()
{
    process(Binding<ENUM_MQL_INFO_INTEGER, int>(_MQLInfoInteger));
    process(Binding<ENUM_TERMINAL_INFO_INTEGER, int>(_TerminalInfoInteger));
    process(Binding<ENUM_TERMINAL_INFO_DOUBLE, double>(_TerminalInfoDouble));
    process(Binding<ENUM_MQL_INFO_STRING, string>(_MQLInfoString));
    process(Binding<ENUM_TERMINAL_INFO_STRING, string>(_TerminalInfoString));
}

```

下面是生成的日志条目的一个片断。

```

ENUM_MQL_INFO_INTEGER Count=15
 0 MQL_PROGRAM_TYPE=1
 1 MQL_DLLS_ALLOWED=0
 2 MQL_TRADE_ALLOWED=0
 3 MQL_DEBUG=1
...
 7 MQL_LICENSE_TYPE=0
...
ENUM_TERMINAL_INFO_INTEGER Count=50
 0 TERMINAL_BUILD=2988
 1 TERMINAL_CONNECTED=1
 2 TERMINAL_DLLS_ALLOWED=0
 3 TERMINAL_TRADE_ALLOWED=0
...
 6 TERMINAL_MAXBARS=100000
 7 TERMINAL_CODEPAGE=1251
 8 TERMINAL_MEMORY_PHYSICAL=4095
 9 TERMINAL_MEMORY_TOTAL=8190
10 TERMINAL_MEMORY_AVAILABLE=7813
11 TERMINAL_MEMORY_USED=377
12 TERMINAL_X64=1
...
ENUM_TERMINAL_INFO_DOUBLE Count=2
 0 TERMINAL_COMMUNITY_BALANCE=0.0 (MQL5_WRONG_PROPERTY,4512)
 1 TERMINAL_RETRANSMISSION=0.0
ENUM_MQL_INFO_STRING Count=2
 0 MQL_PROGRAM_NAME=Environment
 1 MQL_PROGRAM_PATH=C:\Program Files\MT5East\MQL5\Scripts\MQL5Book\p4\Environment.ex
ENUM_TERMINAL_INFO_STRING Count=6
 0 TERMINAL_COMPANY=MetaQuotes Software Corp.
 1 TERMINAL_NAME=MetaTrader 5
 2 TERMINAL_PATH=C:\Program Files\MT5East
 3 TERMINAL_DATA_PATH=C:\Program Files\MT5East
 4 TERMINAL_COMMONDATA_PATH=C:\Users\User\AppData\Roaming\MetaQuotes\Terminal\Common
 5 TERMINAL_LANGUAGE=Russian

```

这些特性以及其它特性将在以下章节中介绍。

值得说明的是，一些特性是继承自平台开发的前期阶段，仅为了兼容性而保留。尤其是 *TerminalInfoInteger* 中的 `TERMINAL_X64` 特性返回关于终端是否是 64 位的指示。如今，32 位版本的开发已经停止，因此该特性始终等于 1 (*true*)。

4.9.2 终端内部版本号

由于终端持续改进，新版本提供新功能，因此 MQL 程序可能需要分析当前版本，以应用不同的算法选择。此外，任何程序均免不了错误，包括终端本身。因此，如果出现问题，你应提供包括终端当前版本信息在内的诊断输出结果。这可帮助重现和修复漏洞。

可以使用 `ENUM_TERMINAL_INFO_INTEGER` 中的 `TERMINAL_BUILD` 获取终端的内部版本号。

```
if(TerminalInfoInteger(TERMINAL_BUILD) >= 3000)
{
    ...
}
```

别忘了，构建程序的编译器的内部版本号可通过宏定义 `__MQLBUILD__` 或 `__MQL5BUILD__` 在源代码中获得（参见 预定义常量）。

4.9.3 程序类型和许可证

相同源代码可以某种方式包括在不同类型的 MQL 程序中。除了选择在编译阶段在通用产品中包括源代码（预处理指令 `#include`）外，还可以组装库，即在执行阶段连接到主程序的二进制程序模块。

然而，某些函数只允许在特定类型的程序中使用。例如，`OrderCalcMargin` 函数不能用于指标。虽然这一限制似乎根本上不合理，但是用于计算抵押资金的通用算法（该算法不仅能够内置于 EA 交易中，还能内置于指标中）的开发人员应将这一细微差别纳入考虑，为指标供一种替代计算方法。

完整程序类型限制列表将在每一章的适当章节提供。在所有这些情况下，重要的是要知道“父”程序的类型。

要确定程序类型，可使用 `ENUM_MQL_INFO_INTEGER` 中的 `MQL_PROGRAM_TYPE` 属性。可能的属性值在 `ENUM_PROGRAM_TYPE` 枚举中描述。

标识符	值	说明
PROGRAM_SCRIPT	1	脚本
PROGRAM_EXPERT	2	EA 交易
PROGRAM_INDICATOR	4	指标
PROGRAM_SERVICE	5	服务

在前一节中的日志片断中，我们看到 `PROGRAM_SCRIPT` 属性设置为 1，因为我们使用脚本测试。要获取字符串说明，可以使用 `EnumToString`。

```
ENUM_PROGRAM_TYPE type = (ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE);
Print(EnumToString(type));
```

另一个便于分析以启用/禁用某些功能的 MQL 程序特性是许可证类型。我们知道，MQL 程序可免费分发，也可以在 MQL5 市场中分发。此外，商店中的程序可购买或作为演示版本下载。这些因素易于检查，并且可根据需要调整算法以适配。为此，可使用 `ENUM_MQL_INFO_INTEGER` 中的 `MQL_LICENSE_TYPE` 特性，该特性将 `ENUM_LICENSE_TYPE` 枚举用作一个类型。

标识符	值	说明
LICENSE_FREE	0	免费无限制版本
LICENSE_DEMO	1	来自市场的付费产品的演示版本，仅可在策略测试程序中工作
LICENSE_FULL	2	购买的许可版本，允许至少 5 次激活（激活次数可由卖家增加）
LICENSE_TIME	3	有时间限制的版本（尚未实现）

务必要注意，许可证指二进制 ex5 模块，使用 `MQLInfoInteger(MQL_LICENSE_TYPE)` 向该模块提交请求。在库中，该函数将返回该库自身的许可证，而不是该库所链接到的主程序的许可证。

作为测试本节所述两个函数的示例，本书提供了简单服务 `EnvType.mq5`。它不包含工作循环，因此在 `OnStart` 中执行两个指令后将立即终止。

```
#property service

void OnStart()
{
    Print(EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE)));
    Print(EnumToString((ENUM_LICENSE_TYPE)MQLInfoInteger(MQL_LICENSE_TYPE)));
}
```

为简化其启动流程（即无需创建一个服务实例并通过终端中的导航器来运行），建议使用调试器：只需在 MetaEditor 中打开源代码并执行命令 `Debugging -> Start on real data`（按 F5 或工具栏上的按钮）。

我们应会得到以下日志条目：

```
EnvType (debug)      PROGRAM_SERVICE
EnvType (debug)      LICENSE_FREE
```

其中你可以清楚看到，程序类型为服务，并且实际上没有许可证（免费使用）。

4.9.4 终端和程序运行模式

MetaTrader 5 环境为交易与编程交互时涉及的各种任务提供了解决方案，这种交互需要涉及终端本身和特定程序的若干种运行模式。

使用 MQL5 API，你可以区分正常在线活动与回测活动，区分源代码调试（以识别潜在错误）与性能分析（查找代码瓶颈），以及区分终端的本地副本和云副本（MetaTrader VPS）。

这些模式以标志描述，每个标志包含一个布尔值：`true` 或 `false`。

标识符	说明
MQL_DEBUG	程序正在以调试模式运行
MQL_PROFILER	程序正在以概要分析模式运行

标识符	说明
MQL_TESTER	程序在测试程序中运行
MQL_FORWARD	程序在前向测试过程中执行
MQL_OPTIMIZATION	程序正在优化过程中运行
MQL_VISUAL_MODE	程序正在可视化测试模式下运行
MQL_FRAME_MODE	EA 交易在图表上以收集优化结果帧的模式执行
TERMINAL_VPS	终端在虚拟服务器 MetaTrader Virtual Hosting (MetaTrader VPS) 上运行

MQL_FORWARD、MQL_OPTIMIZATION 以及 MQL_VISUAL_MODE 标志表明已经设置了 MQL_TESTER 标志。

某些成对标志组合相互排斥，即这些标志不能同时启用。

尤其是 MQL_FRAME_MODE 与 MQL_TESTER 相互排斥。MQL_OPTIMIZATION 排斥 MQL_VISUAL_MODE，而 MQL_PROFILER 排斥 MQL_DEBUG。

我们将学习与测试相关的所有标志 (MQL_TESTER、MQL_VISUAL_MODE) (在专门探讨 EA 交易 以及部分涉及 指标的章节中)。与 EA 交易优化相关的所有内容 (MQL_OPTIMIZATION、MQL_FORWARD、MQL_FRAME_MODE)将在 单独章节中讨论。

现在我们使用调试 (MQL_DEBUG) 和分析 (MQL_PROFILER) 模式来熟悉标志读取原理。同时，我们回顾如何从 MetaEditor 激活这些模式 (有关详细信息，参见文档中的 调试 和 概要分析 章节)。

我们将使用 *EnvMode.mq5* 脚本。

```
void OnStart()
{
    PRTF(MQLInfoInteger(MQL_TESTER));
    PRTF(MQLInfoInteger(MQL_DEBUG));
    PRTF(MQLInfoInteger(MQL_PROFILER));
    PRTF(MQLInfoInteger(MQL_VISUAL_MODE));
    PRTF(MQLInfoInteger(MQL_OPTIMIZATION));
    PRTF(MQLInfoInteger(MQL_FORWARD));
    PRTF(MQLInfoInteger(MQL_FRAME_MODE));
}
```

在运行程序之前，你应检查调试/分析设置。为此，在 MetaEditor 中运行命令 *Tools -> Options* 并检查 *Debugging/Profiling* 选项卡中的字段值。如果启用了选项 *Use specified settings*，则将影响程序启动所在金融工具图表和时间范围的，是相关字段的值。如果禁用了该选项，则将使用 *Market Watch* 中的第一个金融工具和 H1 时间范围。

在此阶段，选择哪个选项并不重要。

准备就绪后，使用命令 *Debug -> Start on Real Data* (F5) 运行脚本。由于该脚本仅将请求的特性打印到日志 (并且其中我们不需要断点)，其执行将瞬时完成。如果需要分步调试，我们可以在源

代码中的任何语句上放置一个断点 (F9)，脚本执行将在该处冻结我们需要的任何时长，从而能够研究 MetaEditor 中所有变量的内容，并且可以在算法中逐行移动 (F10)。

在 MetaTrader 5 日志中 (“Expert”选项卡)，我们将看到以下内容：

```
MQLInfoInteger(MQL_TESTER)=0 / ok
MQLInfoInteger(MQL_DEBUG)=1 / ok
MQLInfoInteger(MQL_PROFILER)=0 / ok
MQLInfoInteger(MQL_VISUAL_MODE)=0 / ok
MQLInfoInteger(MQL_OPTIMIZATION)=0 / ok
MQLInfoInteger(MQL_FORWARD)=0 / ok
MQLInfoInteger(MQL_FRAME_MODE)=0 / ok
```

所有模式的标志均被重置，MQL_DEBUG 除外。

现在我们在 MetaTrader 5 中从 **Navigator** 运行相同脚本 (只需用鼠标将其拖到任何图表)。我们将得到几乎相同的一系列标志，但这次 MQL_DEBUG 将等于 0 (因为程序以正常方式执行，而不是以调试器模式执行)。

请注意，以调试模式启动的程序之前，会先进行特殊模式的重新编译，在此其间，会将允许进行调试的服务信息添加到可执行文件。该二进制文件比一般的要更大更慢。因此，调试完成后，在用于真实交易、传输给客户或上传到市场之前，应使用 **File -> Compile** 命令 (F7) 重新编译该程序。

编译方法不直接影响 MQL_DEBUG 特性。我们可以看到，该程序的调试版本可以在没有调试器的终端中启动，在此情况下，MQL_DEBUG 将被重置。两个内置宏可用于确定编译方法：_DEBUG 和 _RELEASE (参见 [预定义常量](#) 章节)。它们是常量而不是函数，因为该属性是在编译时被“固定”在程序中的，因此无法更改 (不同于运行时环境)。

现在我们在 MetaEditor 中执行命令 **Debug -> Start Profiling on Real Data**。当然，分析这样一个简单脚本没有特定意义，但是我们现在的任务是确保正确标志在环境特性中打开。实际上，现在 MQL_PROFILER 的值为 1。

```
MQLInfoInteger(MQL_TESTER)=0 / ok
MQLInfoInteger(MQL_DEBUG)=0 / ok
MQLInfoInteger(MQL_PROFILER)=1 / ok
...
```

带分析的程序启动也是首先以另一种特殊模式重新编译，在此期间，向二进制文件中添加测量指令执行速度所需的其他服务信息。在分析了分析器报告并修复了瓶颈之后，你应以常规方式重新编译程序。

原则上，调试和分析既可在线执行，也能在测试程序 (MQL_TESTER) 中基于历史数据执行，但测试程序仅支持 EA 交易和指标。因此，无法在脚本示例中看到设置的 MQL_TESTER 或 MQL_VISUAL_MODE 标志。

我们知道，MetaTrader 5 支持以快速模式 (没有图表) 和可视化模式 (在单独图表上) 测试交易程序。在第二种情况下才会启用 MQL_VISUAL_MODE 特性。有必要进行检查，尤其是在没有可视化的情况下，可禁用对 **图形对象** 的操作。

要使用历史数据以可视化模式调试，必须首先在 MetaEditor 设置对话框中启用 **Use visual mode for debugging on history** 选项。分析程序 (指标) 始终在可视化模式下测试。

记住，在线调试对于交易性 EA 交易并不安全。

4.9.5 权限

出于安全原因，MetaTrader 5 提供了用于限制 MQL 程序执行某些操作的功能。某些限制是两个层级的，即为别针对整个终端以及为特定程序设置。终端设置优先，或者作为任何 MQL 程序设置的默认值。例如，交易员可以通过选中 MetaTrader 5 设置对话框中的对应选框来禁用所有自动交易。在此情况下，早前在相关对话框中为特定机器人设置的私有交易权限变为无效。

在 MQL5 API 中，这种限制（或者反面，权限）适用于通过 *TerminalInfoInteger* 和 *MQLInfoInteger* 函数进行的读取。由于它们对 MQL 程序的效果相同，因此程序对于总体和特定禁止项均需仔细检查（以避免尝试执行非法操作时出错）。因此，本节提供了不同层级所有选项的列表。

所有权限均为布尔标志，即它们存储 *true* 或 *false* 值。

标识符	说明
TERMINAL_DLLS_ALLOWED	使用 DLL 的权限
TERMINAL_TRADE_ALLOWED	在线自动交易的权限
TERMINAL_EMAIL_ENABLED	发送电子邮件的权限（必须在终端设置中指定 SMTP 服务器和登录信息）
TERMINAL_FTP_ENABLED	通过 FTP 向指定服务器发送文件的权限（包括为在终端设置中指定的交易账户提供的报告）
TERMINAL_NOTIFICATIONS_ENABLED	向智能手机发送推送通知的权限
MQL_DLLS_ALLOWED	将 DLL 用于该程序的权限
MQL_TRADE_ALLOWED	自动交易程序的权限
MQL_SIGNALS_ALLOWED	信号处理程序的权限

在终端层级使用 DLL 的权限表示当运行一个包含指向某些动态库的链接的 MQL 程序时，“依赖项”选项卡上的“启用 DLL 导入”标志将在其属性对话框中默认启用。如果该标志在终端设置中被清除，则在 MQL 程序的属性中的选项将默认禁用。在任何情况下，用户必须允许单个程序的导入（对于脚本有一个例外，将在下文讨论）。否则，程序将不会运行。

换言之，TERMINAL_DLLS_ALLOWED 和 MQL_DLLS_ALLOWED 标志既可由绑定到 DLL 的程序检查，也可以由绑定到 DLL 的程序检查，但对于后者，MQL_DLLS_ALLOWED 必须明确设置为 true（因为已经启动）。因此，作为需要 DLL 的软件系统的一部分，可能需要提供一个独立实用工具，用于监测标志状态并在标志突然被关闭时向用户显示诊断信息。例如，一个 EA 交易可能需要一个使用 DLL 的指标。然后，在尝试加载指标并获得其句柄之前，EA 可能检查 TERMINAL_DLLS_ALLOWED 标志并生成一个警告（如果标志被重置）。

对于脚本，该行为可能稍有不同，因为仅当源代码中存在 *#property script_show_inputs* 指令时，脚本设置对话框才会打开。如果不存在，则当 TERMINAL_DLLS_ALLOWED 标志在终端设置中被重置时（并且用户必须启用该标志以让脚本运行），该对话框才会出现。当普通标志 TERMINAL_DLLS_ALLOWED 被启用后，该脚本无需用户确认即运行，即假定 MQL_DLLS_ALLOWED 值为 true（根据 TERMINAL_DLLS_ALLOWED）。

在测试程序中运行时，TERMINAL_TRADE_ALLOWED 和 MQL_TRADE_ALLOWED 标志始终等于 true。然而，在 指标 中，无论这些标志如何设置，均禁止对所有交易函数的访问。测试程序不允许测试具有 DLL 依赖项的 MQL 程序。

TERMINAL_EMAIL_ENABLED、TERMINAL_FTP_ENABLED 以及 TERMINAL_NOTIFICATIONS_ENABLED 标志对于 *send mail*、*SendFTP* 和 *send notification* 函数很关键，这些函数在 [网络函数](#) 一节中介绍。MQL_SIGNALS_ALLOWED 标志会影响一组管理 mq5.com 交易信号订阅 (不在本书讨论范围内) 的函数的可用性。其状态对应于 MQL 程序特性的 *Common* 选项卡中的 *Allow changing signal settings* 选项。

由于检查某些特性需要额外工作，因此最好将标志封装在一个类中，该类通过方法隐藏对各种系统函数的多次调用。这是十分必要的，因为某些权限不仅限于上述选项。例如，交易权限不仅可以在终端或 MQL 程序层级设置 (或移除)，而且可以为单个金融工具设置，比如根据你的经纪人和交易所时段为其指定的规范来设置。因此，在这一步，我们将提供“权限”类的草案，其中仅包含我们熟知的元素，今后我们将针对特定应用程序 API 进行改善。

有了作为程序层的类，编程人员无需记住哪些权限是为 *TerminalInfo* 函数定义的，哪些是为 *MqlInfo* 函数定义。

源代码在 *EnvPermissions.mq5* 文件中。

```

class Permissions
{
public:
    static bool isTradeEnabled(const string symbol = NULL, const datetime session = 0)
    {
        // TODO: will be supplemented by applied checks of the symbol and sessions
        return PRTF(TerminalInfoInteger(TERMINAL_TRADE_ALLOWED))
            && PRTF(MQLInfoInteger(MQL_TRADE_ALLOWED));
    }
    static bool isDllsEnabledByDefault()
    {
        return (bool)PRTF(TerminalInfoInteger(TERMINAL_DLLS_ALLOWED));
    }
    static bool isDllsEnabled()
    {
        return (bool)PRTF(MQLInfoInteger(MQL_DLLS_ALLOWED));
    }

    static bool isEmailEnabled()
    {
        return (bool)PRTF(TerminalInfoInteger(TERMINAL_EMAIL_ENABLED));
    }

    static bool isFtpEnabled()
    {
        return (bool)PRTF(TerminalInfoInteger(TERMINAL_FTP_ENABLED));
    }

    static bool isPushEnabled()
    {
        return (bool)PRTF(TerminalInfoInteger(TERMINAL_NOTIFICATIONS_ENABLED));
    }

    static bool isSignalsEnabled()
    {
        return (bool)PRTF(MQLInfoInteger(MQL_SIGNALS_ALLOWED));
    }
};

```

所有类方法均为静态的，在 *OnStart* 中调用。

```

void OnStart()
{
    Permissions::isTradeEnabled();
    Permissions::isDllsEnabledByDefault();
    Permissions::isDllsEnabled();
    Permissions::isEmailEnabled();
    Permissions::isPushEnabled();
    Permissions::isSignalsEnabled();
}

```

生成的日志示例如下所示。

```
TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)=1 / ok
MQLInfoInteger(MQL_TRADE_ALLOWED)=1 / ok
TerminalInfoInteger(TERMINAL_DLLS_ALLOWED)=0 / ok
MQLInfoInteger(MQL_DLLS_ALLOWED)=0 / ok
TerminalInfoInteger(TERMINAL_EMAIL_ENABLED)=0 / ok
TerminalInfoInteger(TERMINAL_NOTIFICATIONS_ENABLED)=0 / ok
MQLInfoInteger(MQL_SIGNALS_ALLOWED)=0 / ok
```

为方便自学，该脚本内置（但注释掉）了以下功能：连接系统 DLL 以读取 Windows 剪贴板内容。我们将在本书第七章探讨库的创建和使用，尤其是 `#import` 指令，具体在 [库](#) 章节中。

我们假定全局 DLL 导入选项在终端中被禁用（出于安全考虑，这是建议设置）。然后，如果 DLL 连接到脚本，则可以仅通过在其单个设置对话框中允许导入来运行脚本，此时 `MQLInfoInteger(MQL_DLLS_ALLOWED)` 将返回 1 (`true`)。如果给定了 DLL 的全局权限，则我们得到 `TerminalInfoInteger(TERMINAL_DLLS_ALLOWED)=1`，并且 `MQL_DLLS_ALLOWED` 继承此值。

4.9.6 检查网络连接

我们知道，MetaTrader 5 平台是一个包含多个环节的分布式系统。除了客户端终端和经纪人服务器，还包括 MQL5 社区、市场，云服务等。事实上，客户端部分也是分布式的，由终端和测试代理组成，可以部署在一个本地网络的多台计算机上。在此情况下，任何环节之间的连接可能因种种原因而断开。虽然 MetaTrader 5 基础架构会尝试自动恢复其功能，但不一定总能快速恢复。

因此，在 MQL 程序中，应当考虑连接丢失的可能性。MQL5 API 允许你控制最重要的连接：与交易服务器和 MQL5 社区的连接。以下属性在 `TerminalInfoInteger` 中可用。

标识符	说明
TERMINAL_CONNECTED	与交易服务器的连接
TERMINAL_PING_LAST	与交易服务器的最后已知 ping 值（微秒）
TERMINAL_COMMUNITY_ACCOUNT	MQL5.community 授权数据在终端中的可用性
TERMINAL_COMMUNITY_CONNECTION	与 MQL5.community 的连接
TERMINAL_MQID	用于发送推送通知的 MetaQuotes ID 的可用性

除了 `TERMINAL_PING_LAST` 以外的所有属性均为布尔标志。`TERMINAL_PING_LAST` 包含一个 `int` 类型的值。

除了连接，MQL 程序通常还需要确保其数据是最新的。特别要注意，检查过的 `TERMINAL_CONNECTED` 标志不一定意味着你感兴趣的报价已与服务器同步。为此，你还需要检查 `SymbolsSynchronized` 或 `SeriesInfoInteger(..., SERIES_SYNCHRONIZED)`。这些功能将在关于 [时间序列](#) 的章节讨论。

`TerminalInfoDouble` 函数支持另一个有趣的特性：`TERMINAL_RETRANSMISSION`。它表示此计算机上所有正在运行的应用程序和服务在 TCP/IP 协议中重新传输的网络数据包百分比。即使是速度最快、配置最完善的网络，也难免发生丢包的情况，导致在接收者和发送者之间无法确认包传递。在此类情况下，丢失的包被重新发送。终端本身不统计 `TERMINAL_RETRANSMISSION` 指标，但是会在操作系统中每分钟请求一次。

若该度量的值很大，则表明可能存在外部问题（互联网连接、你的网络提供商、本地网络或计算机问题），这可能降低终端连接的质量。

如果确认已经与社区建立连接（`TERMINAL_COMMUNITY_CONNECTION`），则 MQL 程序可以通过调用 `TerminalInfoDouble(TERMINAL_COMMUNITY_BALANCE)` 查询用户的当前余额。这可让你使用对付费交易信号的自动订阅（API文档见 mq5.com 网站）。

我们使用 `EnvConnection.mq5` 脚本检查列出的特性。

```
void OnStart()
{
    PRTF(TerminalInfoInteger(TERMINAL_CONNECTED));
    PRTF(TerminalInfoInteger(TERMINAL_PING_LAST));
    PRTF(TerminalInfoInteger(TERMINAL_COMMUNITY_ACCOUNT));
    PRTF(TerminalInfoInteger(TERMINAL_COMMUNITY_CONNECTION));
    PRTF(TerminalInfoInteger(TERMINAL_MQID));
    PRTF(TerminalInfoDouble(TERMINAL_RETRANSMISSION));
    PRTF(TerminalInfoDouble(TERMINAL_COMMUNITY_BALANCE));
}
```

下面是一个日志示例（值将匹配你的设置）。

```
TerminalInfoInteger(TERMINAL_CONNECTED)=1 / ok
TerminalInfoInteger(TERMINAL_PING_LAST)=49082 / ok
TerminalInfoInteger(TERMINAL_COMMUNITY_ACCOUNT)=0 / ok
TerminalInfoInteger(TERMINAL_COMMUNITY_CONNECTION)=0 / ok
TerminalInfoInteger(TERMINAL_MQID)=0 / ok
TerminalInfoDouble(TERMINAL_RETRANSMISSION)=0.0 / ok
TerminalInfoDouble(TERMINAL_COMMUNITY_BALANCE)=0.0 / ok
```

4.9.7 计算资源：内存、磁盘和 CPU

与所有程序一样，MQL 应用程序也会消耗计算机资源，包括内存、磁盘以及 CPU。鉴于终端本身是资源密集型的（尤其因为可能下载具有长期历史记录多个金融工具的报价和分时报价），有时候就需要从可用资源限制的接近程度角度分析和控制情况。

MQL5 API 提供了若干属性，允许你估计最大可用资源以及已使用的资源。这些特性汇总在 `ENUM_MQL_INFO_INTEGER` 和 `ENUM_TERMINAL_INFO_INTEGER` 枚举中。

标识符	说明
MQL_MEMORY_LIMIT	用于 MQL 程序的最大可能动态内存数（单位 KB）
MQL_MEMORY_USED	MQL 程序使用的内存（单位 MB）
MQL_HANDLES_USED	类对象数量
TERMINAL_MEMORY_PHYSICAL	系统中的物理 RAM（单位 MB）
TERMINAL_MEMORY_TOTAL	终端（代理）进程可用的内存（物理内存+交换文件，即虚拟内存）（单位 MB）

标识符	说明
TERMINAL_MEMORY_AVAILABLE	终端 (代理) 进程的空闲内存 (单位 MB), TOTAL 的一部分
TERMINAL_MEMORY_USED	终端 (代理) 使用的内存 (单位 MB), TOTAL 的一部分
TERMINAL_DISK_SPACE	空闲磁盘, 考虑终端 (代理) 的 MQL5/Files 文件夹的配额 (单位 MB)
TERMINAL_CPU_CORES	系统中的处理器核心数
TERMINAL_OPENCL_SUPPORT	支持的 OpenCL 版本, 表示为 0x00010002 = 1.2; "0" 表示不支持 OpenCL

MQL 程序可用的最大内存由 MQL_MEMORY_LIMIT 特性描述。这是列出的特性中唯一使用千字节数 (KB) 作为单位的。所有其它特性均以兆字节数 (MB) 返回。原则上, MQL_MEMORY_LIMIT 等于 TERMINAL_MEMORY_TOTAL, 即默认情况下计算机上的所有可用内存可分配给一个 MQL 程序。然而, 终端 (尤其是其用于 MetaTrader VPS 的云实施) 以及云测试代理可能会限制单个 MQL 程序使用的内存。这样 MQL_MEMORY_LIMIT 将大大少于 TERMINAL_MEMORY_TOTAL。

由于 Windows 通常创建一个等于物理内存 (RAM) 大小的交换文件, 因此 TERMINAL_MEMORY_TOTAL 特性大小最多可能是 TERMINAL_MEMORY_PHYSICAL 的两倍。

所有可用虚拟内存 TERMINAL_MEMORY_TOTAL 分为已使用 (TERMINAL_MEMORY_USED) 内存和仍空闲 (TERMINAL_MEMORY_AVAILABLE) 内存。

本书提供了 *EnvProvision.mq5* 脚本, 该脚本将所有指定特性记录到日志。

```
void OnStart()
{
    PRTF(MQLInfoInteger(MQL_MEMORY_LIMIT)); // Kb!
    PRTF(MQLInfoInteger(MQL_MEMORY_USED));
    PRTF(TerminalInfoInteger(TERMINAL_MEMORY_PHYSICAL));
    PRTF(TerminalInfoInteger(TERMINAL_MEMORY_TOTAL));
    PRTF(TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE));
    PRTF(TerminalInfoInteger(TERMINAL_MEMORY_USED));
    PRTF(TerminalInfoInteger(TERMINAL_DISK_SPACE));
    PRTF(TerminalInfoInteger(TERMINAL_CPU_CORES));
    PRTF(TerminalInfoInteger(TERMINAL_OPENCL_SUPPORT));

    uchar array[];
    PRTF(ArrayResize(array, 1024 * 1024 * 10)); // allocate 10 Mb
    PRTF(MQLInfoInteger(MQL_MEMORY_USED));
    PRTF(TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE));
    PRTF(TerminalInfoInteger(TERMINAL_MEMORY_USED));
}
```

首次输出特性后, 我们为数组分配 10 MB, 然后再次检查内存。结果示例如下所示 (你将有自己的值)。

```
MQLInfoInteger(MQL_MEMORY_LIMIT)=8388608 / ok
MQLInfoInteger(MQL_MEMORY_USED)=1 / ok
TerminalInfoInteger(TERMINAL_MEMORY_PHYSICAL)=4095 / ok
TerminalInfoInteger(TERMINAL_MEMORY_TOTAL)=8190 / ok
TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE)=7842 / ok
TerminalInfoInteger(TERMINAL_MEMORY_USED)=348 / ok
TerminalInfoInteger(TERMINAL_DISK_SPACE)=4528 / ok
TerminalInfoInteger(TERMINAL_CPU_CORES)=4 / ok
TerminalInfoInteger(TERMINAL_OPENCL_SUPPORT)=0 / ok
ArrayResize(array,1024*1024*10)=10485760 / ok
MQLInfoInteger(MQL_MEMORY_USED)=11 / ok
TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE)=7837 / ok
TerminalInfoInteger(TERMINAL_MEMORY_USED)=353 / ok
```

注意总虚拟内存 (8190) 是物理内存的 (4095) 的两倍。脚本可用的内存数是 8388608 KB，几乎等于全部内存 8190 MB。空闲 (7842) 和已使用 (348) 系统内存总和也等于 8190。

如果在为数组分配内存之前，MQL 程序占用 1 Mb，则分配之后，其已经占用 11 Mb。同时，终端占用的内存数仅增加了 5 MB (从 348 增加到 353)，因为一些资源已事先预留。

4.9.8 屏幕规格

由函数 *TerminalInfoInteger* 提供的若干特性涉及计算机的视频子系统。

标识符	说明
TERMINAL_SCREEN_DPI	屏幕输出信息的分辨率以每线性英寸点数 (DPI 每英寸点数) 计量
TERMINAL_SCREEN_LEFT	虚拟屏幕左坐标
TERMINAL_SCREEN_TOP	虚拟屏幕上坐标
TERMINAL_SCREEN_WIDTH	虚拟屏幕宽度
TERMINAL_SCREEN_HEIGHT	虚拟屏幕高度
TERMINAL_LEFT	相对于虚拟屏幕的终端左坐标
TERMINAL_TOP	相对于虚拟屏幕的终端上坐标
TERMINAL_RIGHT	相对于虚拟屏幕的终端右坐标
TERMINAL_BOTTOM	相对于虚拟屏幕的终端下坐标

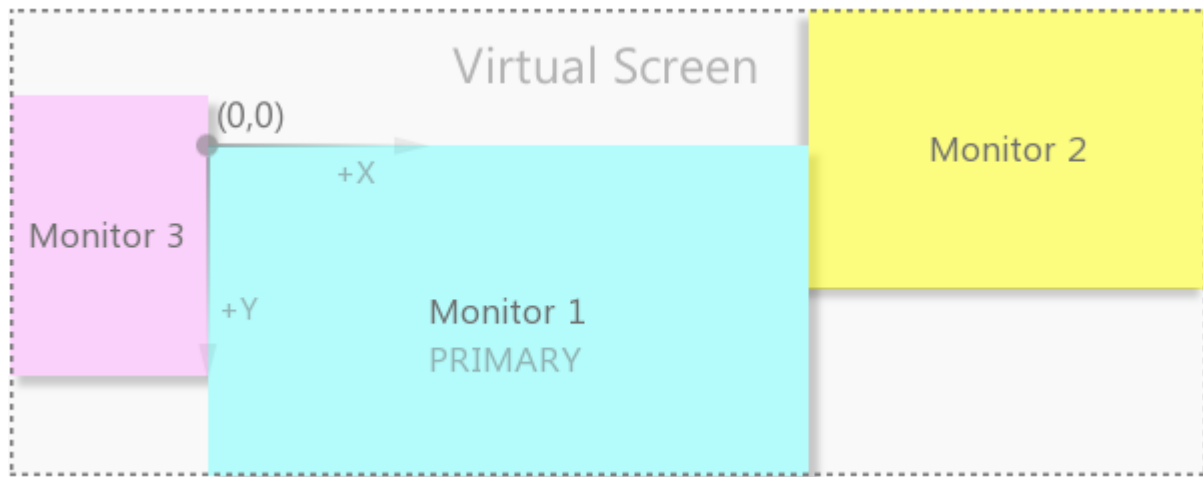
知道 TERMINAL_SCREEN_DPI 参数后，你可以设置 **图形对象** 的尺寸，以便这些图形对象在不同分辨率的显示器保持一致外观。例如，如果你想创建一个视觉大小为 X 厘米的按钮，则可以使用以下函数以屏幕点数 (像素数) 指定：

```

int cm2pixels(const double x)
{
    static const double inch2cm = 2.54; // 1 inch equals 2.54 cm
    return (int)(x / inch2cm * TerminalInfoInteger(TERMINAL_SCREEN_DPI));
}

```

虚拟屏幕是所有显示器的边界框。如果系统中有多个显示器，并且排列顺序不同，严格从左到右，则虚拟屏幕的左坐标可能是负值，而中间（参考点）将处于两个显示器的边框上（位于主显示器的左上角）。



多显示器虚拟屏幕

如果系统有一个显示器，则虚拟屏幕的大小完全与之对应。

终端坐标不考虑当前窗口是否为最大化（即，如果主窗口最大化，则该特性返回未最大化大小，尽管终端被扩展至整个显示器）。

可在 *EnvScreen.mq5* 脚本中检查读取屏幕属性。

```

void OnStart()
{
    PRTF(TerminalInfoInteger(TERMINAL_SCREEN_DPI));
    PRTF(TerminalInfoInteger(TERMINAL_SCREEN_LEFT));
    PRTF(TerminalInfoInteger(TERMINAL_SCREEN_TOP));
    PRTF(TerminalInfoInteger(TERMINAL_SCREEN_WIDTH));
    PRTF(TerminalInfoInteger(TERMINAL_SCREEN_HEIGHT));
    PRTF(TerminalInfoInteger(TERMINAL_LEFT));
    PRTF(TerminalInfoInteger(TERMINAL_TOP));
    PRTF(TerminalInfoInteger(TERMINAL_RIGHT));
    PRTF(TerminalInfoInteger(TERMINAL_BOTTOM));
}

```

下面是生成的日志条目的一个示例。

```
TerminalInfoInteger(TERMINAL_SCREEN_DPI)=96 / ok
TerminalInfoInteger(TERMINAL_SCREEN_LEFT)=0 / ok
TerminalInfoInteger(TERMINAL_SCREEN_TOP)=0 / ok
TerminalInfoInteger(TERMINAL_SCREEN_WIDTH)=1440 / ok
TerminalInfoInteger(TERMINAL_SCREEN_HEIGHT)=900 / ok
TerminalInfoInteger(TERMINAL_LEFT)=126 / ok
TerminalInfoInteger(TERMINAL_TOP)=41 / ok
TerminalInfoInteger(TERMINAL_RIGHT)=1334 / ok
TerminalInfoInteger(TERMINAL_BOTTOM)=836 / ok
```

除了屏幕和终端窗口的总体大小外，MQL 程序经常需要分析图表（终端内的子窗口）的当前大小。为此，有一组特殊函数（尤其是 *ChartGetInteger*），我们将在 [图表](#) 一节中讨论。

4.9.9 终端和程序字符串特性

MQLInfoString 和 *TerminalInfoString* 函数可用于找出终端和 MQL 程序的若干字符串特性。

标识符	说明
MQL_PROGRAM_NAME	正在运行的 MQL 程序的名称
MQL_PROGRAM_PATH	正在运行的 MQL 程序的路径
TERMINAL_LANGUAGE	终端语言
TERMINAL_COMPANY	公司（经纪人）名称
TERMINAL_NAME	终端名称
TERMINAL_PATH	从中启动终端的文件夹
TERMINAL_DATA_PATH	存储终端数据的文件夹
TERMINAL_COMMONDATA_PATH	安装在计算机上的所有客户端终端的共享文件夹

正在运行的程序的名称（MQL_PROGRAM_NAME）通常与主模块（mq5 文件）的名称一致，但也可能不同。尤其是如果你的源代码编译为一个 [库](#)，而该库被导入到另一个 MQL 程序中（EA 交易、指标，脚本或服务），则 MQL_PROGRAM_NAME 特性将返回主程序的名称，而不是该库的名称（库不是独立程序，无法运行）。

我们在 [处理文件](#) 中讨论过工作终端文件夹的安排。使用列出的特性，你可以找出终端安装位置（TERMINAL_PATH），还可以找到当前终端实例（TERMINAL_DATA_PATH）以及所有实例（TERMINAL_COMMONDATA_PATH）的工作数据。

有一个简单脚本 *EnvDescription.mq5* 可将所有这些特性记录在日志中。

```
void OnStart()
{
    PRTF(MQLInfoString(MQL_PROGRAM_NAME));
    PRTF(MQLInfoString(MQL_PROGRAM_PATH));
    PRTF(TerminalInfoString(TERMINAL_LANGUAGE));
    PRTF(TerminalInfoString(TERMINAL_COMPANY));
    PRTF(TerminalInfoString(TERMINAL_NAME));
    PRTF(TerminalInfoString(TERMINAL_PATH));
    PRTF(TerminalInfoString(TERMINAL_DATA_PATH));
    PRTF(TerminalInfoString(TERMINAL_COMMONDATA_PATH));
}
```

下面是一个结果示例。

```
MQLInfoString(MQL_PROGRAM_NAME)=EnvDescription / ok
MQLInfoString(MQL_PROGRAM_PATH)= »
» C:\Program Files\MT5East\MQL5\Scripts\MQL5Book\p4\EnvDescription.ex5 / ok
TerminalInfoString(TERMINAL_LANGUAGE)=Russian / ok
TerminalInfoString(TERMINAL_COMPANY)=MetaQuotes Software Corp. / ok
TerminalInfoString(TERMINAL_NAME)=MetaTrader 5 / ok
TerminalInfoString(TERMINAL_PATH)=C:\Program Files\MT5East / ok
TerminalInfoString(TERMINAL_DATA_PATH)=C:\Program Files\MT5East / ok
TerminalInfoString(TERMINAL_COMMONDATA_PATH)= »
» C:\Users\User\AppData\Roaming\MetaQuotes\Terminal\Common / ok
```

终端界面语言不仅可以通过 `TERMINAL_LANGUAGE` 特性中的一个字符串获取，还可以通过代码页编号获取（参见下一节中的 `TERMINAL_CODEPAGE` 特性）。

4.9.10 自定义属性：柱线数限制和界面语言

在终端的特性中，有两个特别特性是用户可以交互式更改的。这包括显示在每个图表上的默认最大柱线数（其对应于 *Options* 对话框中 *Max. bars in the window* 字段的值），以及界面语言（使用 *View -> Languages* 命令选择）。

标识符	说明
TERMINAL_MAXBARS	图表上的最大柱线数
TERMINAL_CODEPAGE	客户端终端中选择的语言代码页编号

请注意，`TERMINAL_MAXBARS` 值设置柱线显示上限，但事实上，如果可用报价历史记录深度在任何时间范围上不够充足，则其数字可能要更小。另一方面，历史记录的长度也可能超过指定限值 `TERMINAL_MAXBARS`。在此情况下，你可以使用 [时间序列](#) 特性组的函数找出潜在可用柱线数：具有 `SERIES_BARS_COUNT` 特性的 *SeriesInfoInteger*。请注意，`TERMINAL_MAXBARS` 值直接影响 RAM 消耗。

4.9.11 将程序绑定到运行时特性

在了解了前面章节中描述的特性后，我们通过一个例子来了解这样一个常见任务：将一个 MQL 程序绑定到一个硬件环境以防止其被复制。当程序通过 MQL5 市场分发时，绑定由服务本身提供。然而，如果程序是定制开发，则可能要链接到账号、客户端名称或者终端（计算机）的可用

特性。第一种方式不一定总是很方便，因为很多交易员有若干个活动账户（可能有不同经纪商），更不用说还有有效期有限的演示账户了。第二种方式可能有不真实的感觉或太普通。因此，我们将实施一种原型算法，用于将程序绑定到选定的一组环境特性。更严谨的安全方案可能会使用 DLL 并直接从 Windows 读取设备硬件标签，但并非每个客户都会同意运行潜在不安全的库。

我们的保护方案见 *EnvSignature.mq5* 脚本。该脚本从环境给定特性计算哈希值，并基于这些哈希值创建一个唯一签名（指纹）。

哈希处理是一种对任意信息的特殊处理，其结果是创建一个具有以下特性的新的数据块（由使用的算法保证）：

- ① 两个原始数据集的哈希值匹配意味着数据几乎百分百是相同的（随机匹配的概率可忽略）。
- ① 如果原始数据更改，则其哈希值也将更改。
- ① 无法以数学方式从哈希值还原原始数据（它们保持加密），除非穷举所有可能的初始值（如果它们的初始大小增加，并且没有关于它们的结构的信息，则问题在可预见的将来无解）。
- ① 哈希大小是固定的（与初始数据量无关）。

假设环境特性其中的一个由该字符串描述：“`TERMINAL_LANGUAGE=German`”。可获得类似下面的一个简单语句（简化）：

```
string language = EnumToString(TERMINAL_LANGUAGE) +
    "=" + TerminalInfoString(TERMINAL_LANGUAGE);
```

实际语言将匹配设置。利用一个假设性的 *Hash* 函数，我们可以计算签名。

```
string signature = Hash(language);
```

有多个特性时，我们可以直接为所有特性重复步骤，或者从组合字字符串请求哈希值（目前这是伪码，不是真实程序的一部分）。

```
string properties[];
// fill in the property lines as you wish
// ...
string signature;
for(int i = 0; i < ArraySize(properties); ++i)
{
    signature += properties[i];
}
return Hash(signature);
```

收到的签名可由用户报告给程序开发人员，开发人员在收到仅适合于该签名的验证字符串时以特殊方式签署。该签名也是基于哈希算法，需要知道某个密钥（密码短语），仅开发人员知道这个密钥，并硬编码到程序中（用于验证阶段）。

开发人员将把验证字符串传递给用户，用户然后将能够通过参数中指定该字符串来运行程序。

如果没有验证字符串的情况下启动，程序应会针对当前环境生成一个新的签名，将其打印到日志，然后退出（该信息应传递给开发人员）。如果验证字符串无效，程序应会显示一条错误消息，然后退出。

可为开发人员提供若干种启动模式：有签名，但没有验证字符串（生成最后一个），或者有签名和验证字符串（程序将重新签署签名，并将其与指定验证字符串比较以进行校验）。

我们评估一下该保护的选择性。毕竟这里的绑定未对任何形式的唯一标识符执行。

下表列出了关于两个特性的统计数据：屏幕尺寸和 RAM。很明显，值将随时间推移而改变，但大致分布将保持不变：一些特性值将是最普遍的，而一些“新款”高端配置和“旧款”值将构成频率递减的“尾部”。

屏幕	1920x1080	1536x864	1440x900	1366x768	800x600
RAM	21%	7%	5%	10%	4%
4Gb 20%	4.20	1.40	1.00	2.0	0.8
8Gb 20%	4.20	1.40	1.00	2.0	0.8
16Gb 15%	3.15	1.05	0.75	1.5	0.6
32Gb 10%	2.10	0.70	0.50	1.0	0.4
64Gb 5%	1.05	0.35	0.25	0.5	0.2

注意具有最大值的单元格，因为它们表示相同的签名（除非我们对它们引入随机性元素，这将在下文讨论）。在本例中，左上角的两个特性组合是最可能的，各为 4.2%。但这些仅仅是两个特性。如果你向评估的环境添加界面语言、时区、核心数以及工作数据路径（首选共享路径，因为它包含 Windows 用户名），则潜在匹配数将明显减少。

对于哈希加密，我们使用内置 *CryptEncode* 函数（将在 加密 一节中描述），该函数支持 SHA256 哈希加密方法。顾名思义，它生成 256 位长度（即 32 个字节）的哈希值。如果需要将其显示给用户，则我们将其转换为十二进制表示的文本，并获得一个长度为 64 字符的字符串。

为使签名更短，我们使用 Base64 编码将其转换 *CryptEncode* 函数及其对应函数 *CryptDecode* 也支持这种编码），转换结果为一个长度为 44 字符的字符串。不同于单向哈希运算，Base64 编码是可逆的，即可以从该编码恢复原始数据。

主要运算由 *EnvSignature* 类实现。它定义 *data* 字符串，该字符串应累积描述环境的某些片段。公共接口由 *append* 函数的若干重载版本组成，以添加具有环境特性的字符串。本质上，它们使用由虚拟 'pepper' 方法返回的某个抽象元素作为链接，将请求的特性名称及其值连接起来。派生类将其定义为特定字符串（但不能为空）。


```

class EnvSignature
{
private:
    string data;
protected:
    virtual string pepper() = 0;
public:
    bool append(const ENUM_TERMINAL_INFO_STRING e)
    {
        return append(EnumToString(e) + pepper() + TerminalInfoString(e));
    }
    bool append(const ENUM_MQL_INFO_STRING e)
    {
        return append(EnumToString(e) + pepper() + MQLInfoString(e));
    }
    bool append(const ENUM_TERMINAL_INFO_INTEGER e)
    {
        return append(EnumToString(e) + pepper()
            + StringFormat("%d", TerminalInfoInteger(e)));
    }
    bool append(const ENUM_MQL_INFO_INTEGER e)
    {
        return append(EnumToString(e) + pepper()
            + StringFormat("%d", MQLInfoInteger(e)));
    }
}

```

为将任意字符串添加到对象，使用普通方法 *append*，在上述方法中已经调用过。

```

bool append(const string s)
{
    data += s;
    return true;
}

```

或者，开发人员可向哈希化数据中添加所谓的“盐”。这是一个具有随机生成数据的数组，进一步复杂化哈希逆转。每次签名生成都将不同于前一次，即使环境保持不变。该功能的实现以及其他更具体的保护措施（如使用对称加密和密钥的动态计算）留待自学。

由于环境由我们熟知的特性（特性列表由 MQL5 API 常量限制）组成，并且它们并非都充分唯一，因此，如果不使用盐，我们的防护机制（如我们计算的那样）会为不同的用户生成相同的签名。如果发生许可证泄露，签名匹配将无法识别泄露源。

因此，你可以通过为每个客户更改实施哈希加密之前提供特性这一方法来增加保护的有效性。当然，该方法本身不应被披露。在研究的示例中，这意味着更改 *pepper* 方法的内容以及重新编译产品。这可能代价高昂，但可避免使用随机盐。

填充了特性字符串之后，我们便能够生成签名。这使用 *emit* 方法完成。

```

string emit() const
{
    uchar pack[];
    if(StringToArray(data + secret(), pack, 0,
        StringLen(data) + StringLen(secret()), CP_UTF8) <= 0) return NULL;

    uchar key[], result[];
    if(CryptEncode(CRYPT_HASH_SHA256, pack, key, result) <= 0) return NULL;
    Print("Hash bytes:");
    ArrayPrint(result);

    uchar text[];
    CryptEncode(CRYPT_BASE64, result, key, text);
    return CharArrayToString(text);
}

```

该方法向数据中添加某个密钥（一个字节序列，只有开发人员知道，且位于程序中），并计算共享字符串的哈希值。该密钥从虚拟 *secret* 方法获得，该方法也将定义派生类。

生成的带有哈希值的字节数组使用 Base64 编码为字符串。

现在我们了解最重要的类函数：*check*。正是该函数实现了“由开发人员签名并在用户端检查签名”这一机制。

```

bool check(const string sig, string &validation)
{
    uchar bytes[];
    const int n = StringToArray(sig + secret(), bytes, 0,
        StringLen(sig) + StringLen(secret()), CP_UTF8);
    if(n <= 0) return false;

    uchar key[], result1[], result2[];
    if(CryptEncode(CRYPT_HASH_SHA256, bytes, key, result1) <= 0) return false;

    /*
    WARNING
    The following code should only be present in the developer utility.
    The program supplied to the user must compile without this if.
    */
    #ifdef I_AM_DEVELOPER
    if(StringLen(validation) == 0)
    {
        if(CryptEncode(CRYPT_BASE64, result1, key, result2) <= 0) return false;
        validation = CharArrayToString(result2);
        return true;
    }
    #endif
    uchar values[];
    // the exact length is needed to not append terminating '0'
    if(StringToArray(validation, values, 0,
        StringLen(validation)) <= 0) return false;
    if(CryptDecode(CRYPT_BASE64, values, key, result2) <= 0) return false;

    return ArrayCompare(result1, result2) == 0;
}

```

在正常运行期间 (对于用户), 该方法计算接收签名的哈希值 (需用密钥补充), 并将其与来自验证字符串的值进行比较 (验证字符串必须首先从 Base64 解码为哈希值的原始二进制表示)。如果两个哈希值匹配, 则验证成功: 验证字符串匹配特性集。很明显, 空验证字符串 (或者随机输入的字符串) 将不能通过测试。

在开发人员的机器上, 必须在源代码中为签名实用工具定义 `I_AM_DEVELOPER` 宏, 这导致空验证字符串被以不同的方式处理。在此情况下, 生成的哈希值是 Base64 编码的, 该字符串通过 **validation** 参数传递。这样, 实用工具将能够向开发人员显示给定签名的现成验证字符串。

要创建一个对象, 你需要使用密钥和盐值来定义字符串的某个派生类。

```

// WARNING: change the macro to your own set of random bytes
#define PROGRAM_SPECIFIC_SECRET "<PROGRAM-SPECIFIC-SECRET>"
// WARNING: choose your characters to link in pairs name='value
#define INSTANCE_SPECIFIC_PEPPER "=" // obvious single sign is selected for demo
// WARNING: the following macro needs to be disabled in the real product,
//          it should only be in the signature utility
#define I_AM_DEVELOPER
#ifdef I_AM_DEVELOPER
#define INPUT input
#else
#define INPUT const
#endif

INPUT string Signature = "";
INPUT string Secret = PROGRAM_SPECIFIC_SECRET;
INPUT string Pepper = INSTANCE_SPECIFIC_PEPPER;

class MyEnvSignature : public EnvSignature
{
protected:
    virtual string secret() override
    {
        return Secret;
    }
    virtual string pepper() override
    {
        return Pepper;
    }
};

```

我们快速挑选几个特性以填充签名。

```

void FillEnvironment(EnvSignature &env)
{
    // the order is not important, you can mix
    env.append(TERMIAL_LANGUAGE);
    env.append(TERMIAL_COMMONDATA_PATH);
    env.append(TERMIAL_CPU_CORES);
    env.append(TERMIAL_MEMORY_PHYSICAL);
    env.append(TERMIAL_SCREEN_DPI);
    env.append(TERMIAL_SCREEN_WIDTH);
    env.append(TERMIAL_SCREEN_HEIGHT);
    env.append(TERMIAL_VPS);
    env.append(MQL_PROGRAM_TYPE);
}

```

现在一切就绪，可以在 **OnStart** 函数中测试我们的保护方案了。但首先我们看看输入变量。由于同一程序将以两个版本编译，即供最终用户使用的版本和供开发人员使用的版本，因此有两组输入变量：一组供用户输入注册数据，另一组基于开发人员的签名生成该数据。预期由开发人员使用的输入变量已在上文使用 **INPUT** 宏进行了描述。用户只能获得验证字符串。

```
input string Validation = "";
```

当该字符串为空时，程序将收集环境数据，生成新的签名，并将其打印到日志。此时脚本的工作便已完成，因为对有用代码的访问尚未被确认。

```
void OnStart()
{
    MyEnvSignature env;
    string signature;
    if(StringLen(Signature) > 0)
    {
        // ... here will be the code to be signed by the author
    }
    else
    {
        FillEnvironment(env);
        signature = env.emit();
    }

    if(StringLen(Validation) == 0)
    {
        Print("Validation string from developer is required to run this script");
        Print("Environment Signature is generated for current state...");
        Print("Signature:", signature);
        return;
    }
    else
    {
        // ... check the validation string here
    }
    Print("The script is validated and running normally");
    // ... actual working code is here
}
```

如果变量 *Validation* 已填充，我们检查其与签名的符合性，如果失败，终止工作。

```

    if(StringLen(Validation) == 0)
    {
        ...
    }
    else
    {
        validation = Validation; // need a non-const argument
        const bool accessGranted = env.check(Signature, validation);
        if(!accessGranted)
        {
            Print("Wrong validation string, terminating");
            return;
        }
        // success
    }
    Print("The script is validated and running normally");
    // ... actual working code is here
}

```

如果不存在不一致性，算法继续进行程序的工作代码。

在开发人员侧（在使用 `I_AM_DEVELOPER` 宏构建的程序版本中），将引入一个签名。我们使用签名还原 *MyEnvSignature* 对象的状态，并计算验证字符串。

```

void OnStart()
{
    ...
    if(StringLen(Signature) > 0)
    {
        #ifdef I_AM_DEVELOPER
        if(StringLen(Validation) == 0)
        {
            string validation;
            if(env.check(Signature, validation))
                Print("Validation:", validation);
            return;
        }
        signature = Signature;
        #endif
    }
    ...
}

```

开发人员不仅可以指定签名，而且可以验证前面：在此情况下，代码执行将以用户模式继续（用于调试目的）。

如果你希望，你可以通过以下方式模拟环境中的变化：

```

FillEnvironment(env);
// artificially make a change in the environment (add a time zone)
// env.append("Dummy" + (string)(TimeGMTOffset() - TimeDaylightSavings()));
const string update = env.emit();
if(update != signature)
{
    Print("Signature and environment mismatch");
    return;
}

```

我们看几个测试日志。

第一次运行 *EnvSignature.mq5* 脚本时，“用户”将看到类似以下日志的内容（值将因环境差异而不同）：

```

Hash bytes:
 4 249 194 161 242 28 43 60 180 195 54 254 97 223 144 247 216 103 238 245 244 2
Validation string from developer is required to run this script
Environment Signature is generated for current state...
Signature:BPnCofIckZy0wzb+Yd+Q99hn7vX04AdEZf34hhtmypk=

```

它将生成的签名发送给“开发人员”（在测试期间没有实际的用户，因此所有“用户”和“开发人员”角色均加了引号），开发人员在 *Signature* 参数中将其输入到使用 `I_AM_DEVELOPER` 宏编译的签名实用工具。结果，程序将生成验证字符串：

```
Validation:YBpYpQ0tLIpUhBsliw+AsPhtPG48b0qut9igJ+Tk1fQ=
```

“开发人员”将其发回给“用户”，“用户”将其输入到 *Validation* 参数中，将得到激活的脚本：

```

Hash bytes:
 4 249 194 161 242 28 43 60 180 195 54 254 97 223 144 247 216 103 238 245 244 2
The script is validated and running normally

```

为证实保护的有效性，我们将该脚本复制为服务：为此，我们将该文件复制到文件夹 *SQL5/Services/SQL5Book/p4/*，并将源代码中的以下行：

```
#property script_show_inputs
```

替换为以下行：

```
#property service
```

我们编译该服务，创建并运行其实例，然后在输入参数中指定先前接收到的验证字符串。结果，服务将中止（在到达具有要求的代码的语句之前）并显示以下消息：

```

Hash bytes:
147 131 69 39 29 254 83 141 90 102 216 180 229 111 2 246 245 19 35 205 223 1
Wrong validation string, terminating

```

重点是在环境的特性中，我们已经使用了字符串 `SQL_PROGRAM_TYPE`。因此，为某一类型程序颁发的许可证不适用于另一类型的程序，即使该程序运行在同一用户的计算机上。

4.9.12 检查键盘状态

TerminalInfoInteger 函数可用于找出控制键的状态，控制键也称为虚拟键。这些键具体包括 *Ctrl*、*Alt*、*Shift*、*Enter*、*Ins*、*Del*、*Esc*、箭头键等等。它们被称为虚拟键，因为原则上，键盘提供了

多种方式来实现同一控制操作。例如，空格键左侧和右侧都有相同的 *Ctrl*、*Shift* 和 *Alt* 键，而光标既可以使用专用按键移动，也可以在按下 *Fn* 键的同时按主按键来移动。因此，该函数不能区分物理层面的控制方式（例如左右 *Shift* 键）。

API 定义了以下按键的常量：

标识符	说明
TERMINAL_KEYSTATE_LEFT	左箭头
TERMINAL_KEYSTATE_UP	上箭头
TERMINAL_KEYSTATE_RIGHT	右箭头
TERMINAL_KEYSTATE_DOWN	下箭头
TERMINAL_KEYSTATE_SHIFT	Shift
TERMINAL_KEYSTATE_CONTROL	Ctrl
TERMINAL_KEYSTATE_MENU	Windows
TERMINAL_KEYSTATE_CAPSLOCK	CapsLock
TERMINAL_KEYSTATE_NUMLOCK	NumLock
TERMINAL_KEYSTATE_SCROLLLOCK	ScrollLock
TERMINAL_KEYSTATE_ENTER	Enter
TERMINAL_KEYSTATE_INSERT	Insert
TERMINAL_KEYSTATE_DELETE	Delete
TERMINAL_KEYSTATE_HOME	Home
TERMINAL_KEYSTATE_END	End
TERMINAL_KEYSTATE_TAB	Tab
TERMINAL_KEYSTATE_PAGEUP	PageUp
TERMINAL_KEYSTATE_PAGEDOWN	PageDown
TERMINAL_KEYSTATE_ESCAPE	Escape

该函数返回一个使用一对位元报告请求的键的当前状态的两字节整数值。

最低有效位跟踪自从上次函数调用以来的键击。例如，如果 *TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE)* 在某个点返回 0，然后用户按了 *Escape*，则在下次调用时，*TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE)* 将返回 1。如果再次按该键，值将回到 0。

对于负责切换输入模式的按键，诸如 *CapsLock*、*NumLock* 以及 *ScrollLock*，位的位置指示对应模式是已启用还是已禁用。

如果在当前时刻该按键被按下（且未松开），则第二字节（0x8000）的最高有效位被置位。

该功能不能用于跟踪字母数字键和功能键的按键情况。为此，有必要实现 [OnChartEvent](#) 处理程序，并以程序中的 `CHARTEVENT_KEYDOWN` 代码截取消息。请注意，事件在图表上生成，并且仅对 EA 交易和指标可用。其它类型的程序（脚本和服务）不支持事件编程模型。

EnvKeys.mq5 脚本包括一个贯穿所有 `TERMINAL_KEYSTATE` 常量的循环。

```
void OnStart()
{
    for(ENUM_TERMINAL_INFO_INTEGER i = TERMINAL_KEYSTATE_TAB;
        i <= TERMINAL_KEYSTATE_SCRLOCK; ++i)
    {
        const string e = EnumToString(i);
        // skip values that are not enum elements
        if(StringFind(e, "ENUM_TERMINAL_INFO_INTEGER") == 0) continue;
        PrintFormat("%s=%4X", e, (ushort)TerminalInfoInteger(i));
    }
}
```

你可以通过敲击键盘来试验，并启用/禁用键盘模式来查看值在日志中如何变化。

例如，如果默认禁用大写锁定，则我们将看到以下日志：

```
TERMINAL_KEYSTATE_SCRLOCK= 0
```

如果我们按住 `ScrollLock` 键不放，再次运行脚本，我们将得到以下日志：

```
TERMINAL_KEYSTATE_CAPSLOCK=8001
```

也就是说，该模式已经启用，并且按键已被按下。我们松开按键，下次脚本运行将返回：

```
TERMINAL_KEYSTATE_SCRLOCK= 1
```

模式保持启用，但按键已松开。

TerminalInfoInteger 不适用于检查由以下调用创建的依赖性指标中按键的状态 (`TERMINAL_KEYSTATE_XYZ`): [iCustom](#) 或 [IndicatorCreate](#)。在这两个函数中，该函数始终返回 0，即使该指标是使用 [ChartIndicatorAdd](#) 添加到图表。

此外，当 MQL 程序图表未处于活动状态时（用户已切换至另一个），该函数无效。MQL5 不提供对键盘永久控制的方法。

4.9.13 检查 MQL 程序状态和终止原因

我们已经在本书不同示例中接触了 *IsStopped* 函数。当 MQL 程序执行较长的计算时，必须不时调用该函数。这样可以检查用户是否发起了程序关闭操作（即，用户是否尝试将其从图表移除）。

bool IsStopped() = bool _StopFlag

如果程序被用户中断，则该函数返回 *true* 例如，在上下文菜单中的“Expert 列表”打开的对话框中，用户按下“删除”按钮）。

程序有 3 秒钟正常暂停计算，必要时保存中间结果，并完成其工作。若未完成，则将程序从图表强制移除。

如果不使用 *IsStopped* 函数，你可以检查内置 *_StopFlag* 变量的值。

EnvStop.mq5 测试脚本可模拟循环执行的长计算：搜索素数。退出 *while* 循环的条件是使用 *IsStopped* 函数编写的。因此，当用户删除脚本时，循环以正常方式中断，日志显示找到的素数日志统计（脚本还可以将这些数字保存到文件）。

```
bool isPrime(int n)
{
    if(n < 1) return false;
    if(n <= 3) return true;
    if(n % 2 == 0) return false;
    const int p = (int)sqrt(n);
    int i = 3;
    for( ; i <= p; i += 2)
    {
        if(n % i == 0) return false;
    }

    return true;
}

void OnStart()
{
    int count = 0;
    int candidate = 1;

    while(!IsStopped()) // try to replace it with while(true)
    {
        // emulate long calculations
        if(isPrime(candidate))
        {
            Comment("Count:", ++count, ", Prime:", candidate);
        }
        ++candidate;
        Sleep(10);
    }
    Comment("");
    Print("Total found:", count);
}
```

如果我们以 *true*（无限循环）替换循环条件，则脚本将停止回应用户的停止请求，并被强制卸载（从图表中）。因此，我们将在日志中看到“异常终止”错误，而窗口左上角的注释仍会残留。这样，本示例中所有表示保存数据和清除忙碌资源的指令（例如，这可能是将你自己的图形对象从窗口删除）均被忽略。

在停止请求被发送到程序之后（值 *_StopFlag* 等于 *true*），可使用 *UninitializeReason* 函数找到终止原因。

遗憾的是，此功能仅适用于 EA 交易和指标。

int UninitializeReason() ≡ int _UninitReason

该函数返回描述反初始化原因的预定义代码之一。

常量	值	说明
REASON_PROGRAM	0	<i>ExpertRemove</i> 调用了仅适用于 EA 交易和脚本的函数
REASON_REMOVE	1	程序已从图表中移除
REASON_RECOMPILE	2	程序已重新编译
REASON_CHARTCHANGE	3	图表交易品种或期间更改
REASON_CHARTCLOSE	4	图表关闭
REASON_PARAMETERS	5	程序输入参数更改
REASON_ACCOUNT	6	另一个账户连接到交易服务器，或者与交易服务器重新连接
REASON_TEMPLATE	7	应用了另一个图表模板
REASON_INITFAILED	8	<i>OnInit</i> 事件处理程序返回错误标志
REASON_CLOSE	9	终端关闭

若不使用函数，也可以访问内置全局变量 *_UninitReason*。

反初始化原因代码还将作为一个参数传递给 *OnDeinit* 事件处理程序函数。

稍后在学习 [程序启动和停止功能](#) 时，我们将看到一个指标 (*Indicators/MQL5Book/p5/LifeCycle.mq5*) 和一个 EA 交易 (*Experts/MQL5Book/p5/LifeCycle.mq5*)，它们将反初始化的原因记录到日志，可让你根据用户操作了解程序行为。

4.9.14 以编程方式关闭终端并设置返回码

MQL5 API 包含若干个不仅用于读取而且用于修改程序环境的函数。其中最为激进的是 *TerminalClose*。使用该函数，MQL 程序可以在无用户确认的情况下关闭终端！

bool TerminalClose(int retcode)

该函数有一个参数 *retcode*，这是由 `terminal64.exe` 进程向 Windows 操作系统返回的代码。要分析这些代码，可使用批处理文件 (`*.bat` 和 `*.cmd`) 以及 shell 脚本 (Windows Script Host，即 WSH，其支持 VBScript 和 JScript) 或 Windows PowerShell (`*.ps1` 文件) 以及其它自动化工具 (例如 Windows 计划程序、Windows 下 Linux 支持子系统 (`*.sh` 文件) 等)。

该函数不会立即停止终端，而是向终端发送一个终止命令。

如果调用结果为 *true*，则表示该命令已被成功“接受进行考虑”，终端将尝试尽快但正确地关闭 (生成一个通知并停止其它正在运行的 MQL 程序)。在调用代码中，当然，也必须为立即终止工作作好所有准备 (尤其是所有此前打开的文件应关闭)，并且在函数调用之后，将控制权交还给终端。

与该进程返回码关联的另一个函数是 *SetReturnError*。它允许你预指派该代码而无需发送立即关闭命令。

void SetReturnError(int retcode)

该函数设置终端进程关闭后将返回给 Windows 系统的代码。

请注意，终端不需要被 *TerminalClose* 函数强制关闭。用户也可以用指定代码正常关闭终端。如果终端因非预期关键错误关闭，该代码也将进入系统。

如果 *SetReturnError* 函数被重复调用和/或从不同的程序调用，则终端将返回上次设置的代码。

我们使用 *EnvClose.mq5* 脚本来测试这些函数。

```
#property script_show_inputs

input int ReturnCode = 0;
input bool CloseTerminalNow = false;

void OnStart()
{
    if(CloseTerminalNow)
    {
        TerminalClose(ReturnCode);
    }
    else
    {
        SetReturnError(ReturnCode);
    }
}
```

要以实际操作方式来测试，我们还需要 *envrun.bat* 文件（位于 *MQL5/Files/MQL5Book/* 文件夹中）。

```
terminal64.exe
@echo Exit code: %ERRORLEVEL%
```

事实上，它仅启动终端，完成后便向控制台显示生成的代码。该文件应被放在终端文件夹中（或者系统中安装的若干个 MetaTrader 5 实例中的当前实例应在 PATH 系统变量中注册）。

例如，如果我们使用批处理文件启动终端，并执行脚本 *EnvClose.mq5*，例如以参数 *ReturnCode=100, CloseTerminalNow=true* 执行，则我们将在控制台中看到类似这样的内容：

```
Microsoft Windows [Version 10.0.19570.1000]
(c) 2020 Microsoft Corporation. All rights reserved.
C:\Program Files\MT5East>envrun
C:\Program Files\MT5East>terminal64.exe
Exit code: 100
C:\Program Files\MT5East>
```

顺便提醒一下，从命令行启动时，MetaTrader 5 支持各种选项（详情参见 [运行交易平台](#)）。因此，可以安排批量测试各种 EA 交易或设置以及数千个被监测账户之间的顺序切换，在一台计算机上通过大量实例持续并行操作实现这些需求并不现实。

4.9.15 处理运行时错误

任何程序即使在编写和编译时没有出现任何问题，仍然无法避免运行时错误。这些错误可能因为开发人员的疏忽或者因为在软件环境中出现的不可预见的情况（诸如互联网连接丢失，内存不足等等）。但同样并不少见的是，因程序应用不当而导致错误发生的情况。在所有这些情况下，程序必须能够分析问题的本质，并充分处理问题。

每个 MQL5 语句都是一个潜在的运行时错误来源。如果发生此类错误，终端会将一个描述代码保存到特殊 `_LastError` 变量。确保在每个语句后立即分析该代码，因为后续语句中的潜在错误会覆写该值。

请注意，有很多关键错误一旦发生，它们将立即终止程序执行：

- 零除
- 索引超出范围
- 对象指针不正确

有关错误代码及其含义的完整列表，参见 [文档](#)。

在 [打开和关闭文件](#) 一节中，在编写实用 `PRTF` 宏的过程中，我们已经讨论了诊断错误的问题。尤其我们已经了解了辅助头文件 `MQL5/Include/MQL5Book/MqlError.mqh`，其中 `MQL_ERROR` 枚举允许使用 `EnumToString` 将数字错误代码轻松转换为名称。

```
enum MQL_ERROR
{
    SUCCESS = 0,
    INTERNAL_ERROR = 4001,
    WRONG_INTERNAL_PARAMETER = 4002,
    INVALID_PARAMETER = 4003,
    NOT_ENOUGH_MEMORY = 4004,
    ...
    // start of area for errors defined by the programmer (see next section)
    USER_ERROR_FIRST = 65536,
};
#define E2S(X) EnumToString((MQL_ERROR)(X))
```

在这里，作为 `E2S` 宏的 `X` 参数，我们应具有 `_LastError` 变量或等效的 `GetLastError` 函数。

`int GetLastError() ≡ int _LastError`

该函数返回 MQL 程序语句中上次发生错误的代码。一开始还没有错误时，值为 0。读取 `_LastError` 和调用 `GetLastError` 函数只是语法上不同而已（根据偏爱的风格选择适当选项）。

应记住，语句的正常无错误执行不会重置错误代码。调用 `GetLastError` 也不会重置错误代码。

因此，如果有一系列操作，其中仅一个操作将设置一个错误标志，则该标志将由函数或后续（成功）操作返回。例如，

```
// _LastError = 0 by default
action1; // ok, _LastError doe not change
action2; // error, _LastError = X
action3; // ok, _LastError does not change, i.e. is still equal to X
action4; // another error, _LastError = Y
action5; // ok, _LastError does not change, that is, it is still equal to Y
action6; // ok, _LastError does not change, that is, it is still equal to Y
```

此行为将导致难以定位问题区域。为避免这一情况，有一个单独的将 `_LastError` 变量重置为 0 的 `ResetLastError` 函数。

void ResetLastError()

该函数将内置 `_LastError` 变量的值设置为零。

若你认为任何操作可能引发错误并且想在发生错误后使用 `GetLastError` 来分析错误，则建议在该操作前调用该函数。

同时使用这两个函数的一个典型例子就是前面提到的 PRTF 宏 (PRTF.mqh 文件)。它的代码如下所示：

```
#include <MQL5Book/MqlError.mqh>

#define PRTF(A) ResultPrint(#A, (A))

template<typename T>
T ResultPrint(const string s, const T retval = NULL)
{
    const int snapshot = _LastError; // recording _LastError at input
    const string err = E2S(snapshot) + "(" + (string)snapshot + ")";
    Print(s, "=", retval, " / ", (snapshot == 0 ? "ok" : err));
    ResetLastError(); // clear the error flag for the next calls
    return retval;
}
```

该宏以及包装在宏中的 `ResultPrint` 函数的目的是将传递的值 (当前错误代码) 记录到日志并立即清除错误代码。这样，对多个语句连续应用 PRTF 便始终可以确保，打印到日志的错误 (或成功指示) 对应于获取到 `retval` 参数的值的最后一个语句。

我们需要将 `_LastError` 保存在中间本地变量 `snapshot` 中，因为在表达式求值过程中，如果任何运算失败，`_LastError` 的值几乎可能在任何位置被修改。在该特定示例中，`E2S` 宏使用 `EnumToString` 函数，如果枚举之外的值被传递为自变量，则该函数可能生成自己的错误代码。然后，在同一表达式的后续部分中，当形成一个字符串时，我们将看到的不是初始错误，而是生成的那个错误。

在任何语句中可能在若干个地方导致 `_LastError` 突然改变。对此，最好在所需操作后立即记录错误代码。

4.9.16 用户定义的错误

开发人员可将内置的 `_LastError` 变量用于他们自身的应用目的。`SetUserError` 函数为这种应用提供了便利。

void SetUserError(ushort user_error)

该函数将内置 `_LastError` 变量设为 `ERR_USER_ERROR_FIRST + user_error` 值，其中 `ERR_USER_ERROR_FIRST` 为 65536。所有低于该值的代码均为系统错误预留。

MQL5 中不支持异常，但利用上述机制，你可以在一定程度上绕过这种局限性。

很多时候，函数使用返回值作为错误标志。然而，在某些算法中，函数必须返回应用程序类型值。我们以 `double` 为例。如果该函数的定义范围是从负到正无穷，则我们选择用于指示错误的任何值（例如 0）将与实际计算结果混淆。当然，对于 `double` 来说，有一个返回特别构造的 NaN 值的选项 `NaN` 表示非数字，参见章节 [检查实数正常性](#)。但如果该函数返回一个结构体或一个类对象怎么办？一种可能的解决方案是通过一个参数按引用或按指针返回结果，但这种方式无法将函数用作表达式的操作数。

在类上下文中，我们来了解名为“构造函数”的特殊函数。构造函数可返回对象的新实例。然而，有时候情况不允许你构造整个对象，在此情况下，调用代码看似获得了对象但却无法使用。如果类能够提供一种额外的方法来检查对象是否可用，这当然不错。但是作为一种统一的替代方法（例如，涵盖所有类），我们可以使用 `SetUserError`。

在 [运算符重载](#) 一节，我们认识了 `Matrix` 类。我们将为其补充用于计算行列式和逆矩阵的方法，然后将其用于演示用户错误（参见文件 `Matrix.mqh`）。为矩阵定义了重载运算符，允许它们组合到一个表达式中的运算符链中，因此在其中实现潜在错误检查就不是很方便。

我们的 `Matrix` 类是对最近添加的 MQL5 内置对象类型 [矩阵](#) 的自定义替代实现。

我们首先验证 `Matrix` 主类构造函数中的输入参数。如果有人试图创建一个零大小的矩阵，我们设置一个自定义错误 `ERR_USER_MATRIX_EMPTY`（提供的若干自定义错误之一）。

```
enum ENUM_ERR_USER_MATRIX
{
    ERR_USER_MATRIX_OK = 0,
    ERR_USER_MATRIX_EMPTY = 1,
    ERR_USER_MATRIX_SINGULAR = 2,
    ERR_USER_MATRIX_NOT_SQUARE = 3
};

class Matrix
{
    ...
public:
    Matrix(const int r, const int c) : rows(r), columns(c)
    {
        if(rows <= 0 || columns <= 0)
        {
            SetUserError(ERR_USER_MATRIX_EMPTY);
        }
        else
        {
            ArrayResize(m, rows * columns);
            ArrayInitialize(m, 0);
        }
    }
}
```

这些新操作仅为方矩阵定义，因此，我们创建一个具有适当大小约束的派生类。

```

class MatrixSquare : public Matrix
{
public:
    MatrixSquare(const int n, const int _ = -1) : Matrix(n, n)
    {
        if(_ != -1 && _ != n)
        {
            SetUserError(ERR_USER_MATRIX_NOT_SQUARE);
        }
    }
    ...
}

```

构造函数中的第二个参数应空缺（假定它等于第一个），但是我们需要它，因为 *Matrix* 类具有一种模板换置方法，其中所有 T 类型必须支持一个具有两个整数参数的构造函数。

```

class Matrix
{
    ...
    template<typename T>
    T transpose() const
    {
        T result(columns, rows);
        for(int i = 0; i < rows; ++i)
        {
            for(int j = 0; j < columns; ++j)
            {
                result[j][i] = this[i][(uint)j];
            }
        }
        return result;
    }
}

```

由于在 *MatrixSquare* 函数中有两个参数，我们还必须检查它们的强制相等性。如果它们不相等，则我们设置 ERR_USER_MATRIX_NOT_SQUARE 错误。

最后，在逆矩阵计算期间，我们会发现矩阵退化（行列式为 0）。预留错误 ERR_USER_MATRIX_SINGULAR 便是用于这种情况。


```

class MatrixSquare : public Matrix
{
public:
    ...
    MatrixSquare inverse() const
    {
        MatrixSquare result(rows);
        const double d = determinant();
        if(fabs(d) > DBL_EPSILON)
        {
            result = complement().transpose<MatrixSquare>() * (1 / d);
        }
        else
        {
            SetUserError(ERR_USER_MATRIX_SINGULAR);
        }
        return result;
    }

    MatrixSquare operator!() const
    {
        return inverse();
    }
    ...

```

对于可视化错误输出，必须向日志添加静态方法，返回 `ENUM_ERR_USER_MATRIX` 枚举，可将其轻松传给 *EnumToString*：

```

static ENUM_ERR_USER_MATRIX lastError()
{
    if(_LastError >= ERR_USER_ERROR_FIRST)
    {
        return (ENUM_ERR_USER_MATRIX)(_LastError - ERR_USER_ERROR_FIRST);
    }
    return (ENUM_ERR_USER_MATRIX)_LastError;
}

```

所有方法的完整代码参见随附文件。

我们将在 *EnvError.mq5* 测试脚本中检查应用程序错误代码。

首先，我们确保该类可正常工作：逆反矩阵，确保原始矩阵和逆矩阵的乘积等于单位矩阵。

```

void OnStart()
{
    Print("Test matrix inversion (should pass)");
    double a[9] =
    {
        1, 2, 3,
        4, 5, 6,
        7, 8, 0,
    };

    ResetLastError();
    Matrix SquaremA(a); // assign data to the original matrix
    Print("Input");
    mA.print();
    MatrixSquare mAinv(3);
    mainv = !mA; // invert and store in another matrix
    Print("Result");
    mAinv.print();

    Print("Check inverted by multiplication");
    Matrix Squaretest(3); // multiply the first by the second
    test = mA * mAinv;
    test.print(); // get identity matrix
    Print(EnumToString(Matrix::lastError())); // ok
    ...
}

```

该代码片断生成以下日志条目。

```

Test matrix inversion (should pass)
Input
1.00000 2.00000 3.00000
4.00000 5.00000 6.00000
7.00000 8.00000 0.00000
Result
-1.77778 0.88889 -0.11111
1.55556 -0.77778 0.22222
-0.11111 0.22222 -0.11111
Check inverted by multiplication
1.00000 +0.00000 0.00000
-0.00000 1.00000 +0.00000
0.00000 0.00000 1.00000
ERR_USER_MATRIX_OK

```

注意在单位矩阵中，由于浮点错误，一些零元素实际上是接近零的极小值，因此它们有符号。

然后，我们看看算法如何处理退化矩阵。

```

Print("Test matrix inversion (should fail)");
double b[9] =
{
    -22, -7, 17,
    -21, 15, 9,
    -34, -31, 33
};

MatrixSquare mB(b);
Print("Input");
mB.print();
ResetLastError();
Print("Result");
(!mB).print();
Print(EnumToString(Matrix::lastError())); // singular
...

```

结果呈现如下。

```

Test matrix inversion (should fail)
Input
-22.000000 -7.000000 17.000000
-21.000000 15.000000 9.000000
-34.000000 -31.000000 33.000000
Result
0.0 0.0 0.0
0.0 0.0 0.0
0.0 0.0 0.0
ERR_USER_MATRIX_SINGULAR

```

在本例中，我们只是显示了错误说明。但在真实程序中，应能够选择一个继续选项，取决于问题的性质。

最后，我们将模拟两个剩余应用错误的情况。

```

Print("Empty matrix creation");
MatrixSquare m0(0);
Print(EnumToString(Matrix::lastError()));

Print("'Rectangular' square matrix creation");
MatrixSquare r12(1, 2);
Print(EnumToString(Matrix::lastError()));
}

```

这里我们描述一个空矩阵和一个假定方矩阵（但大小不同）。

```

Empty matrix creation
ERR_USER_MATRIX_EMPTY
'Rectangular' square matrix creation
ERR_USER_MATRIX_NOT_SQUARE

```

在这些情况下，我们无法避免地要创建对象，因为编译器会自动创建。

当然，此测试显然违反约定（数据和操作规范中关于类和方法“视为”有效的约定）。然而，在实践中，自变量通常是从代码的其它部分获取的，在处理大型“第三方”数据的过程中，检测与期望的偏差并非易事。

程序能否“消化”不正确数据而不引发致命后果，这一点与为正确输入数据生成正确结果一样，都是程序质量最重要的指标。

4.9.17 调试管理

MetaEditor 中的内置调试器允许在源代码中设置断点，即程序执行应暂停的行。有时候调试系统会失效（即暂停未生效），在此情况下，你可以使用 **DebugBreak** 函数显式执行停止。

void DebugBreak()

调用该函数会暂停程序并以调试模式激活编辑器窗口，具有用于查看变量和调用堆栈以及用于分步继续进一步执行的所有工具。

仅当程序是从编辑器以调试模式启动时（通过命令 **Debug -> Start on Real Data** 或 **Start in History Data**），程序执行才能被中断。在所有其它模式下，包括正常启动（在终端中）和概要分析模式，该函数均无效。

4.9.18 预定义变量

每个 MQL 程序具有某些由终端提供的普通全局变量集：其中大部分已在前面章节中介绍过，下面是摘要表。绝大部分变量都是只读的。**_LastError** 变量例外，该变量可以由 **ResetLastError** 函数重置。

变量	值
_LastError	上次错误值，类似于 GetLastError 函数
_StopFlag	程序停止标志，类似于 IsStopped 函数
_UninitReason	程序反初始化原因代码，类似于 UninitializeReason 函数
_RandomSeed	伪随机整数生成器 的当前内部状态
_IsX64	64 位终端的标志，类似于 TerminalInfoInteger （适于 TERMINAL_X64 特性）

此外，对于在图表上下文中运行的 MQL 程序，诸如 EA 交易、脚本和指标，语言为预定义变量提供了图表特性（这些变量也无法从程序中更改）。

变量	值
_Symbol	当前图表交易品种的名称，类似于 Symbol 函数
_Period	当前图表的 时间范围 ，类似于 Period 函数
_Digits	当前图表交易品种的价格中的小数位数，类似于 Digits 函数
_Point	当前交易品种的价格中的点值（以报价货币表示），类似于 Point 函数
_AppliedTo	计算 指标 所依据的数据类型（仅限指标）

4.9.19 MQL5 语言的预定义常量

本节介绍由运行时环境为任何程序定义的所有常量。在前面章节中我们已经接触过其中一些常量。某些常量涉及应用的 MQL5 编程要素，将在后面章节中介绍。

常量	说明	值
CHARTS_MAX	最大可能数量：同时打开的 图表	100
clrNONE	无 颜色	-1 (0xFFFFFFFF)
EMPTY_VALUE	指标缓冲区中的空值	DBL_MAX
INVALID_HANDLE	句柄无效	-1
NULL	任何类型 null	0
WHOLE_ARRAY	直至数组末尾的元素数量，即整个数组将被处理	-1
WRONG_VALUE	一个常量可被显式强制转换为任何枚举类型	-1

如 [文件](#) 一章中所示，INVALID_HANDLE 常量不能用于验证文件描述符。

WHOLE_ARRAY 常量适用于处理如下 [数组](#) 的函数：其要求指定处理数组中的元素数量：如果有必要处理从指定位置到末尾的所有数组值，则指定 WHOLE_ARRAY 值。

EMPTY_VALUE 常量通常被指派给 [指标缓冲区](#)中不应在图表上绘出的元素。换言之，该常量表示一个默认 [空值](#)。稍后我们将介绍对于具有另一个值（例如 0）的特定指标缓冲区如何进行替换。

WRONG_VALUE 常量适用于需要指定一个不正确的 [枚举](#) 值的情况。

此外，两个常量具有不同的值，取决于编译方法。

常量	说明
<code>IS_DEBUG_MODE</code>	表示以调试模式运行 <code>mql5</code> 程序的属性：在调试模式下不为零，在其他模式下为 0。
<code>IS_PROFILE_MODE</code>	表示以分析模式运行 <code>mql5</code> 程序的属性：在分析模式下不为零，在其他模式下为 0。

`IS_PROFILE_MODE` 常量允许你更改程序的运行，以便在 [概要分析](#) 模式下正确收集信息。分析可让你测量单个程序片段（函数和单个行）的执行时间。

在编译期间，编译器设置 `IS_PROFILE_MODE` 常量值。一般情况下设置为 0。当程序以分析模式启动时，执行特殊编译，在此情况下，使用非零值而不是 `IS_PROFILE_MODE`。

`IS_DEBUG_MODE` 常量以类似方式工作：原生编译时它等于 0，调试编译后大于 0。在出于验证目的而需要小幅更改 `MQL` 程序运行的情况下，这个常量很有用：例如，向日志输出额外信息，或者在图表上创建辅助图形对象。

预处理器定义含义类似的 `_DEBUG` 和 `_RELEASE` 常量（参见 [预定义预处理器常量](#)）。

可使用 *`MQLInfoInteger`* 函数在运行时获取有关程序运行模式的更详细信息（参见 [终端和程序运行模式](#)）。特别要注意的是，程序的调试构建版本可在没有调试器的情况下运行。

4.10 矩阵和向量

`MQL5` 语言提供了特殊对象数据类型：矩阵和向量。矩阵和向量可用于解决多种数学问题。通过这些类型，可以编写接近线性或微分方程数学符号的简明易懂代码。

所有编程语言都支持数组的概念，数组是多个元素的集合。大多数算法（尤其是在算法交易中）均基于数值型数组（*int*, *double*）或结构体而构建。数组元素可通过索引访问，索引使得我们能够在循环内实施运算。我们知道，数组可能是一维、二维或多维。

相对简单的数据存储和处理任务通常可通过使用数组来实现。但是对于复杂的数学问题，大量嵌套循环会导致数组操作在编程和代码可读性方面变得困难。即使最简单的线性代数运算也需要编写大量代码和对数学的良好理解。这一任务可通过编程的 [函数范式](#) 来简化，体现在矩阵和向量方法函数的使用。这些操作执行很多“幕后”例行操作。

诸如机器学习、神经网络以及三维图形等现代技术广泛应用于线性代数问题解决，而求解线性代数就要用到向量和矩阵运算。`MQL5` 新增的数据类型正是为了快速方便地处理这些对象。

在编写本书时，正在积极开发用于处理矩阵和向量的函数集，因此，很多有趣的新项目在这里可能未提到。敬请关注 mql5.com 网站上的发行说明和文章版块。

在本章中，我们将提供简要描述。有关矩阵和向量的更多详细信息，参见相应的帮助章节 [矩阵和向量方法](#)。

同时，我们假定读者熟悉线性代数理论。如果必要，你随时可以上网查阅参考文献和手册。

4.10.1 矩阵和向量的类型

向量是实数或复数类型的一维数组，而矩阵是实数或复数类型的二维数组。因此，这些对象元素的有效数值类型列表包括 *double*（视为默认类型）、*float* 和 *complex*。

从线性代数的角度（不是从编译器的角度！），一个素数也是一个最小向量，而反过来，一个向量可被视为一种特殊情况下的矩阵。

根据元素类型，可使用任一 **vector**（带或不带后缀）关键字描述向量：

- **vector** 是一个具有如下类型元素的向量：**double**
- **vectorf** 是一个具有如下类型元素的向量：**float**
- **vectorc** 是一个具有如下类型元素的向量：**complex**

虽然向量可以分为垂直和水平向量，但是 MQL5 不作此区分。要求的向量朝向由向量在表达式中的位置确定（暗示）。

对向量定义了以下运算：加法和乘法，以及范数（使用相关 **norm** 方法），范数运算可获取向量长度或模。

你可以将矩阵视为数组，其中第一个索引是行号，第二个索引是列号。然而，行和列的编号方式不同于线性代数，而是像数组一样从零开始。

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

矩阵的两个维度也称为轴，按如下方式编号：0 为横轴（沿行），而 1 为纵轴（沿列）。很多矩阵函数中都会使用轴编号。尤其是当我们谈到将矩阵拆分时，横拆分表示在行之间切割，纵拆分表示在列之间切割。

根据元素类型，可使用任一 **matrix**（带或不带后缀）关键字描述描述矩阵：

- **matrix** 是一个具有以下类型元素的矩阵：**double**
- **matrixf** 是一个具有以下类型元素的矩阵：**float**
- **matrixc** 是一个具有以下类型元素的矩阵：**complex**

对于模板函数中的应用，可以使用 **matrix<double>**，**matrix<float>**，**matrix<complex>**，**vector<double>**，**vector<float>**，**vector<complex>** 标记法代替对应类型。

```
vectorf v_f1 = {0, 1, 2, 3,};
vector<float> v_f2 = v_f1;
matrix m = {{0, 1}, {2, 3}};
```

```
void OnStart()
{
    Print(v_f2);
    Print(m);
}
```

当被记录到日志时，矩阵和向量以逗号分隔的数字序列形式输出，并用方括号括起。

```
[0,1,2,3]
[[0,1]
 [2,3]]
```

为矩阵定义了以下代数运算：

- 相同大小矩阵的加法
- 适配大小矩阵的乘法，即第一矩阵中的列数必须等于第二矩阵中的行数。
- 根据矩阵乘法规则，矩阵与列向量相乘、行向量与矩阵相乘（向量可视为特殊矩阵）
- 矩阵乘以数字

此外，*matrix* 和 *vector* 类型内置了与 NumPy 库（Python 中流行的机器学习库）类似的方法，因此你可以在文档和库示例中获得更多提示。完整方法列表参见 [MQL5 帮助](#) 的对应章节。

遗憾的是，MQL5 未提供从一种类型的矩阵和向量到另一种类型的强制转换（例如，从 *double* 到 *float*）。此外，在需要矩阵的表达式中，编译器不会自动将向量视为矩阵（一列或一行的矩阵）。这意味着尽管在这些结构之间存在明显关系，但是矩阵和向量之间不存在继承概念（OOP 的特性）。

4.10.2 创建和初始化矩阵和向量

有若干种方式声明和初始化矩阵和向量。可以根据用途将其分为若干种类别。

- ① 声明时不指定大小
- ① 声明时指定大小
- ① 声明时包含初始化
- ① 静态创建方法
- ① 非静态（重新）配置和初始化方法

最简单的创建方法是不指定大小的声明，即不为数据分配内存。对于这种情况，只需指定变量的类型和名称：

```
matrix          matrix_a;   // matrix of type double
matrix<double>  matrix_a1;  // double type matrix inside function or class templates
matrix<float>    matrix_a2;  // float matrix
vector          vector_v;   // vector of type double
vector<double>  vector_v1;  // another notation of a double-type vector creation
vector<float>   vector_v2;  // vector of type float
```

然后你可以更改创建的对象的大小，并以所需的值填充。它们也可用于内置矩阵和向量方法以获取计算结果。所有这些方法将在本章的各节中分组讨论。

你可以在指定大小的情况下声明矩阵或向量。这将分配内存并不进行任何初始化。对于这种情况，在圆括号中的变量名称后面，指定大小（对于矩阵，第一个是行数，第二个是列数）：


```

matrix          matrix_a(128, 128);      // you can specify as parameters
matrix<double>  matrix_a1(nRows, nCols); // both constants and variables
matrix<float>   matrix_a2(nRows, 1);     // analog of column vector
vector          vector_v(256);
vector<double>  vector_v1(nSize);
vector<float>   vector_v2(nSize + 16);   // expression as a parameter

```

创建对象的第三种方法是带初始化的声明。在此情况下，矩阵和向量的大小由花括号中指示的初始化系列确定：

```

matrix          matrix_a = {{0.1, 0.2, 0.3}, {0.4, 0.5, 0.6}};
matrix<double>  matrix_a1 = matrix_a;    // must be matrices of the same type
matrix<float>   matrix_a2 = {{1, 2}, {3, 4}};
vector          vector_v = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
vector<double>  vector_v1 = {1, 5, 2.4, 3.3};
vector<float>   vector_v2 = vector_v1;   // must be vectors of the same type

```

还有一些静态方法可创建指定大小的矩阵和向量并以某种方式进行初始化（特别是针对某种规范形式）。以下列出了所有这些方法，且具有类似的原型（向量和矩阵的唯一差别是没有第二维度）。

```

static matrix<T> matrix<T>::EyefTri(const ulong rows, const ulong cols, const int diagonal = 0);
static matrix<T> matrix<T>::IdentityfOnesfZeros(const ulong rows, const ulong cols);
static matrix<T> matrix<T>::Full(const ulong rows, const ulong cols, const double value);

```

- 🕒 **Eye** 构造一个在指定对角线上为 1、其余位置为 0 的矩阵
- 🕒 **Tri** 构造一个在指定对角线及其下方为 1、其余位置为 0 的矩阵
- 🕒 **Identity** 构造一个指定大小的单位矩阵
- 🕒 **Ones** 构造一个以 1 填充的矩阵（或向量）
- 🕒 **Zeros** 构造一个以 0 填充的矩阵（或向量）
- 🕒 **Full** 构造一个所有元素均以给定值填充的矩阵（或向量）

如果必要，你可以将任何现有矩阵转换为一个单位矩阵，并应为该矩阵应用一个非静态方法 **Identity**（无参数）。

我们来以实操的方式演示这些方法：

```

matrix          matrix_a = matrix::Eye(4, 5, 1);
matrix<double>  matrix_a1 = matrix::Full(3, 4, M_PI);
matrixf         matrix_a2 = matrixf::Identity(5, 5);
matrixf<float>  matrix_a3 = matrixf::Ones(5, 5);
matrix          matrix_a4 = matrix::Tri(4, 5, -1);
vector          vector_v = vector::Ones(256);
vectorf         vector_v1 = vector<float>::Zeros(16);
vector<float>   vector_v2 = vectorf::Full(128, float_value);

```

此外，还有以下非静态方法可以用给定值初始化矩阵/向量：**Init** 和 **Fill**。

```

void matrix<T>::Init(const ulong rows, const ulong cols, func_reference rule = NULL, ...)
void matrix<T>::Fill(const T value)

```

Init 方法的一个重要优势（构造函数也具备）是能够在参数中指定一个初始化函数，用于根据给定规则填充矩阵/向量的元素（参见下面的示例）。

可通过在 *rules* 参数中指定不带引号的标识符 (这不是 *typedef (*pointer)(...)* 意义上的指针, 也不是具有名称的字符串), 将该函数的引用传递到大小之后。

该初始化函数的第一个参数必须包含对被填充对象的引用, 并且还可以具有附加参数: 在此情况下, 它们的值被传递给 *Init* 或传递给函数引用之后的一个构造函数。如果 *rule* 链接未指定, 则将直接创建一个指定维数的矩阵。

Init 方法还允许更改矩阵配置。

我们使用几个简短例子来了解上述内容。

```
matrix m(2, 2);
m.Fill(10);
Print("matrix m \n", m);
/*
    matrix m
    [[10,10]
    [10,10]]
*/
m.Init(4, 6);
Print("matrix m \n", m);
/*
    matrix m
    [[10,10,10,10,10,0.0078125,32.00000762939453]
    [0,0,0,0,0,0]
    [0,0,0,0,0,0]
    [0,0,0,0,0,0]]
*/
```

这里的 *Init* 方法用于调整已经初始化的矩阵的大小, 结果新元素被填充了随机值。

下面的函数以指数级增加的数字填充矩阵。

```
template<typename T>
void MatrixSetValues(matrix<T> &m, const T initial = 1)
{
    T value = initial;
    for(ulong r = 0; r < m.Rows(); r++)
    {
        for(ulong c = 0; c < m.Cols(); c++)
        {
            m[r][c] = value;
            value *= 2;
        }
    }
}
```

然后它可用于创建矩阵。

```
void OnStart()
{
    matrix M(3, 6, MatrixSetValues);
    Print("M = \n", M);
}
```

执行结果为：

```
M =
[[1,2,4,8,16,32]
 [64,128,256,512,1024,2048]
 [4096,8192,16384,32768,65536,131072]]
```

在此情况下，构造函数调用时未在函数标识符后指定初始化函数的参数值，因此使用默认值 (1)。但是我们可以为相同的 *MatrixSetValues* 传递一个初始值 -1，其将以一个负行填充矩阵。

```
matrix M(3, 6, MatrixSetValues, -1);
```

4.10.3 复制矩阵、向量和数组

要复制矩阵和向量，最简单最常见的方法是使用赋值运算符 '='。

```
matrix a = {{2, 2}, {3, 3}, {4, 4}};
matrix b = a + 2;
matrix c;
Print("matrix a \n", a);
Print("matrix b \n", b);
c.Assign(b);
Print("matrix c \n", c);
```

该代码段生成以下日志条目：

```
matrix a
[[2,2]
 [3,3]
 [4,4]]
matrix b
[[4,4]
 [5,5]
 [6,6]]
matrix c
[[4,4]
 [5,5]
 [6,6]]
```

Copy 和 *Assign* 方法也可用于复制矩阵和向量。*Assign* 和 *Copy* 的区别在于，*Assign* 不仅允许复制矩阵，也可以复制数组。

```
bool matrix<T>::Copy(const matrix<T> &source)
bool matrix<T>::Assign(const matrix<T> &source)
bool matrix<T>::Assign(const T &array[])
```

类似方法和原型也对向量可用。

通过 *Assign*，能够将一个向量写入到矩阵：结果将为一行矩阵。

```
bool matrix<T>::Assign(const vector<T> &v)
```

你也可以将一个矩阵指派给一个向量：其将为未包装，即矩阵的所有行将排为一行（等效于调用 *Flat* 方法）。

```
bool vector<T>::Assign(const matrix<T> &m)
```

截止本章撰稿时，MQL5 中没有用于将矩阵或向量导出到数组的方法，虽然有助于“转移”数据的机制（详见 *Swap* 方法）。

下面的例子显示了整数数组 *int_arr* 如何被复制到 *double* 类型的矩阵。在此情况下，生成的矩阵自动调整为复制数组的大小。

```
matrix double_matrix = matrix::Full(2, 10, 3.14);
Print("double_matrix before Assign() \n", double_matrix);
int int_arr[5][5] = {{1, 2}, {3, 4}, {5, 6}};
Print("int_arr: ");
ArrayPrint(int_arr);
double_matrix.Assign(int_arr);
Print("double_matrix after Assign(int_arr) \n", double_matrix);
```

我们在日志中获得以下输出。

```
double_matrix before Assign()
[[3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14]
 [3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14]]

int_arr:
    [,0][,1][,2][,3][,4]
[0,]  1  2  0  0  0
[1,]  3  4  0  0  0
[2,]  5  6  0  0  0
[3,]  0  0  0  0  0
[4,]  0  0  0  0  0

double_matrix after Assign(int_arr)
[[1,2,0,0,0]
 [3,4,0,0,0]
 [5,6,0,0,0]
 [0,0,0,0,0]
 [0,0,0,0,0]]
```

因此，*Assign* 方法可用于从数组切换到矩阵，并自动调整大小和转换类型。

一种更高效的在矩阵、向量和数组之间转移数据的方法是使用 *Swap* 方法。

```

bool matrix<T>::Swap(vector<T> &vec)
bool matrix<T>::Swap(matrix<T> &vec)
bool matrix<T>::Swap(T &arr[])
bool vector<T>::Swap(vector<T> &vec)
bool vector<T>::Swap(matrix<T> &vec)
bool vector<T>::Swap(T &arr[])

```

它们的工作方式类似于 *ArraySwap*: 在两个对象内有数据的缓冲区内部指针被交换。结果, 矩阵或向量的元素在源对象中消失而出现在接收数组中, 或者反过来, 它们从数组移动到矩阵或向量。

Swap 方法可以处理动态数组, 包括多维数组。对于多维数组 (*array[][N1][N2]...*) 的最高维的恒定大小, 需满足以下条件: 这些维度的乘积必须是矩阵或向量的大小的倍数。这样, 一个 `int[2][3]` 数组被重新分布在 6 个元素的块中。因此, 它与大小为 6、12、18 (以此类推) 的矩阵和向量可互换。

4.10.4 将时间序列复制到矩阵和向量

matrix<T>::CopyRates 方法将具有报价历史记录的时间序列直接复制到矩阵或向量。该方法的工作方式类似于我们将在第五章关于 *时间序列* 章节中提到的函数, 即: *CopyRates* 和单独的 *Copy* 函数, 用于 *MqlRates* 结构体的每个字段。

```

bool matrix<T>::CopyRates(const string symbol, ENUM_TIMEFRAMES tf, ulong rates_mask,
    ulong start, ulong count)
bool matrix<T>::CopyRates(const string symbol, ENUM_TIMEFRAMES tf, ulong rates_mask,
    datetime from, ulong count)
bool matrix<T>::CopyRates(const string symbol, ENUM_TIMEFRAMES tf, ulong rates_mask,
    datetime from, datetime to)

```

在这些参数中, 你需要指定交易品种、时间范围以及请求的柱线的范围: 按编号和数量, 或按日期范围。复制数据时, 将最旧的元素放在矩阵/向量的开头。

rates_mask 参数指定带有一组可用字段的来自 *ENUM_COPY_RATES* 枚举的标志组合。该标记组合允许你在一个请求中从历史记录获取若干时间序列。在此情况下, 矩阵中的行顺序将对应于 *ENUM_COPY_RATES* 枚举中的值顺序, 尤其是矩阵中具有 *High* 数据的行将始终在具有 *Low* 数据的行上面。

复制到向量时, 仅可指定一个来自 *ENUM_COPY_RATES* 枚举的值。否则将会出错。

标识符	值	说明
COPY_RATES_OPEN	1	<i>Open</i> 价格
COPY_RATES_HIGH	2	<i>High</i> 价格
COPY_RATES_LOW	4	<i>Low</i> 价格
COPY_RATES_CLOSE	8	<i>Close</i> 价格
COPY_RATES_TIME	16	柱线打开次数
COPY_RATES_VOLUME_TICK	32	分时报价交易量
COPY_RATES_VOLUME_REAL	64	真实交易量
COPY_RATES_SPREAD	128	点差
		组合
COPY_RATES_OHLC	15	<i>Open, High, Low, Close</i>
COPY_RATES_OHLCT	31	<i>Open, High, Low, Close, Time</i>

我们将在 [解方程](#) 一节查看该函数的使用示例。

4.10.5 将分时报价历史记录复制到矩阵或向量

与柱线一样，你可以将分时报价复制到向量或矩阵。通过 `CopyTicks` 和 `CopyTicksRange` 方法重载即可实现。它们的工作原理依据类似于 `CopyTicks` 和 `CopyTicksRange` 函数，但它们将数据接收到调用方中。这些函数将在第五章中关于 [真实分时报价数组（MqlTick 结构体）](#)一节中详述。在这里我们将仅描述原型和要点。

```

bool matrix<T>::CopyTicks(const string symbol, uint flags, ulong from_msc, uint count)
bool vector<T>::CopyTicks(const string symbol, uint flags, ulong from_msc, uint count)
bool matrix<T>::CopyTicksRange(const string symbol, uint flags, ulong from_msc, ulong to_msc)
bool matrix<T>::CopyTicksRange(const string symbol, uint flags, ulong from_msc, ulong to_msc)

```

`symbol` 参数设置为其请求分时报价的金融工具的名称。分时报价范围可以不同的方式指定：

- ⌚ 在 `CopyTicks` 中，可以指定为分时报价数（`count` 参数），从毫秒单位的某一刻（`from_msc`）开始。
- ⌚ 在 `CopyTicksRange` 中，它可以是两个时间点的范围（从 `from_msc` 到 `to_msc`）。

至于复制的数据要包含每个分时报价的哪些组成部分，可在 `flags` 参数中以 `ENUM_COPY_TICKS` 枚举值的位掩码指定。

标识符	值	说明
COPY_TICKS_INFO	1	由 <i>Bid</i> 和 /或 <i>Ask</i> 变化生成的分时报价
COPY_TICKS_TRADE	2	由 <i>Last</i> 和 <i>Volume</i> 变化生成的分时报价
COPY_TICKS_ALL	3	所有分时报价
COPY_TICKS_TIME_MS	1 << 8	以毫秒为单位的时间
COPY_TICKS_BID	1 << 9	<i>Bid</i> 价格
COPY_TICKS_ASK	1 << 10	<i>Ask</i> 价格
COPY_TICKS_LAST	1 << 11	<i>Last</i> 价格
COPY_TICKS_VOLUME	1 << 12	交易量
COPY_TICKS_FLAGS	1 << 13	分时报价标志

前三位 (低字节) 确定请求的分时报价集, 而其余位 (高字节) 确定这些分时报价的特性。

高字节标志只能用于矩阵组合, 因为仅具有来自所有分时报价的特定字段的值的一行放在向量中。因此, 应仅选择最高有效字节的一位来填充向量。

在填充矩阵的过程中选择分时报价的若干特性时, 其中的行顺序将对应于枚举中元素的顺序。例如, *Bid* 价格将始终出现在高于具有 *Ask* 价格的行的具有更低索引的行中。

处理分时报价和向量的一个示例将在关于 [机器学习](#) 的章节中提供。

4.10.6 含矩阵和向量的表达式的求值

你可以逐元素 (使用运算符) 对矩阵和向量执行数学运算, 如加减乘除。对于这些运算, 两个对象必须为相同的类型并具有相同的维数。矩阵/向量的每个成员与第二矩阵/向量的对应元素交互。

作为第二项 (乘数、减数或除数), 你还可以使用对应类型 (*double*、*float* 或 *complex*) 的标量。在此情况下, 将在考虑该标量的情况下处理矩阵或向量的每个元素。

```

matrix matrix_a = {{0.1, 0.2, 0.3}, {0.4, 0.5, 0.6}};
matrix matrix_b = {{1, 2, 3}, {4, 5, 6}};
matrix matrix_c1 = matrix_a + matrix_b;
matrix matrix_c2 = matrix_b - matrix_a;
matrix matrix_c3 = matrix_a * matrix_b;    // Hadamard product (element-by-element)
matrix matrix_c4 = matrix_b / matrix_a;
matrix_c1 = matrix_a + 1;
matrix_c2 = matrix_b - double_value;
matrix_c3 = matrix_a * M_PI;
matrix_c4 = matrix_b / 0.1;
matrix_a += matrix_b;                      // operations "in place" are possible
matrix_a /= 2;

```

原位运算通过将结果放入其中而修改原始矩阵 (或向量), 不同于常规二进制运算, 后者其中操作数保持不变, 并为结果创建一个新对象。

此外, 矩阵和向量可作为参数传递给大部分 [数学函数](#)。在此情况下, 矩阵或向量逐元素处理。例如:

```

matrix a = {{1, 4}, {9, 16}};
Print("matrix a=\n", a);
a = MathSqrt(a);
Print("MatrSqrt(a)=\n", a);
/*
matrix a=
[[1,4]
 [9,16]]
MatrSqrt(a)=
[[1,2]
 [3,4]]
*/

```

对于 *MathMod* 和 *MathPow*, 第二个参数可以是标量或矩阵, 或者是适当大小的向量。

4.10.7 操作矩阵和向量

在处理矩阵和向量时, 基础操作可无需任何计算。矩阵专属方法提供在列表开头, 而最后四个方法也适用于向量。

- ① **Transpose**: 矩阵转置
- ① **Col, Row, Diag**: 按编号提取和设置行、列和对角线
- ① **TriL, TriU**: 按对角线编号获取上下三角矩阵
- ① **SwapCols, SwapRows**: 按编号指示重新排列行和列
- ① **Flat**: 按穿透索引设置及获取矩阵元素
- ① **Reshape**: 原位调整矩阵形状
- ① **Split, Hsplit, Vsplit**: 将矩阵拆分为若干子矩阵
- ① **resize**: 原位调整矩阵或向量的大小;
- ① **Compare, CompareByDigits**: 以给定实数精度比较两个矩阵或两个向量
- ① **Sort**: 原位排序 (元素置换) 以及通过获取向量或矩阵索引来排序

🕒 **clip**: 原位限制元素值范围

请注意，未提供向量拆分。

下面是用于矩阵的原型方法。

```
matrix<T> matrix<T>::Transpose()
vector matrix<T>::ColfRow(const ulong n)
void matrix<T>::ColfRow(const vector v, const ulong n)
vector matrix<T>::Diag(const int n = 0)
void matrix<T>::Diag(const vector v, const int n = 0)
matrix<T> matrix<T>::TriLfTriU(const int n = 0)
bool matrix<T>::SwapColsfSwapRows(const ulong n1, const ulong n2)
T matrix<T>::Flat(const ulong i)
bool matrix<T>::Flat(const ulong i, const T value)
bool matrix<T>::Resize(const ulong rows, const ulong cols, const ulong reserve = 0)
void matrix<T>::Reshape(const ulong rows, const ulong cols)
ulong matrix<T>::Compare(const matrix<T> &m, const T epsilon)
ulong matrix<T>::CompareByDigits(const matrix &m, const int digits)
bool matrix<T>::Split(const ulong nparts, const int axis, matrix<T> &splitted[])
void matrix<T>::Split(const ulong &parts[], const int axis, matrix<T> &splitted[])
bool matrix<T>::HsplitfVsplit(const ulong nparts, matrix<T> &splitted[])
void matrix<T>::HsplitfVsplit(const ulong &parts[], matrix<T> &splitted[])
void matrix<T>::Sort(func_reference compare = NULL, T context)
void matrix<T>::Sort(const int axis, func_reference compare = NULL, T context)
matrix<T> matrix<T>::Sort(func_reference compare = NULL, T context)
matrix<T> matrix<T>::Sort(const int axis, func_reference compare = NULL, T context)
bool matrix<T>::Clip(const T min, const T max)
```

对于向量，有更小的方法集。

```
bool vector<T>::Resize(const ulong size, const ulong reserve = 0)
ulong vector<T>::Compare(const vector<T> &v, const T epsilon)
ulong vector<T>::CompareByDigits(const vector<T> &v, const int digits)
void vector<T>::Sort(func_reference compare = NULL, T context)
vector vector<T>::Sort(func_reference compare = NULL, T context)
bool vector<T>::Clip(const T min, const T max)
```

矩阵转置示例：

```

matrix a = {{0, 1, 2}, {3, 4, 5}};
Print("matrix a \n", a);
Print("a.Transpose() \n", a.Transpose());
/*
matrix a
[[0,1,2]
 [3,4,5]]
a.Transpose()
[[0,3]
 [1,4]
 [2,5]]
*/

```

使用 *Diag* 方法设置不同对角线的若干示例：

```

vector v1 = {1, 2, 3};
matrix m1;
m1.Diag(v1);
Print("m1\n", m1);
/*
m1
[[1,0,0]
 [0,2,0]
 [0,0,3]]
*/

matrix m2;
m2.Diag(v1, -1);
Print("m2\n", m2);
/*
m2
[[0,0,0]
 [1,0,0]
 [0,2,0]
 [0,0,3]]
*/

matrix m3;
m3.Diag(v1, 1);
Print("m3\n", m3);
/*
m3
[[0,1,0,0]
 [0,0,2,0]
 [0,0,0,3]]
*/

```

使用 *Reshape* 更改矩阵配置：

```

matrix matrix_a = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};
Print("matrix_a\n", matrix_a);
/*
    matrix_a
    [[1,2,3]
     [4,5,6]
     [7,8,9]
     [10,11,12]]
*/

matrix_a.Reshape(2, 6);
Print("Reshape(2,6)\n", matrix_a);
/*
    Reshape(2,6)
    [[1,2,3,4,5,6]
     [7,8,9,10,11,12]]
*/

matrix_a.Reshape(3, 5);
Print("Reshape(3,5)\n", matrix_a);
/*
    Reshape(3,5)
    [[1,2,3,4,5]
     [6,7,8,9,10]
     [11,12,0,3,0]]
*/

matrix_a.Reshape(2, 4);
Print("Reshape(2,4)\n", matrix_a);
/*
    Reshape(2,4)
    [[1,2,3,4]
     [5,6,7,8]]
*/

```

在 [解方程](#) 时，我们将在一个示例中应用矩阵拆分为子矩阵。

Col 和 **Row** 方法不仅允许按其编号获取矩阵的列或行，而且允许原位将它们插入先前定义的矩阵。在此情况下，矩阵的维数或列向量（对于 **Col** 的情况）或行向量（对于 **Row** 的情况）以外元素的值均将不变。

如果这两个方法中的任一个应用于一个尚未设置其维数的矩阵，则将创建一个大小为 $[N * M]$ 的空矩阵，其中以不同方式为 **Col** 和 **Row** 定义 N 和 M （基于向量的长度以及给定的列或行索引）：

- ⌚ 对于 **Col**， N 是列向量的长度， M 比插入列的指定索引大 1
- ⌚ 对于 **Row**， N 比插入行的指定索引大 1，而 M 是行向量的长度

截止本章撰稿时，MQL5 还未提供用于成熟可靠的带后续元素扩展的行和列插入的方法，以及用于排除指定行和列的方法。

4.10.8 矩阵和向量的乘积

矩阵乘法是各种数值方法中的基础运算之一。例如，在神经网络层中实施前向和后向传播方法时经常用到。

各种卷积也可归因于矩阵乘积的类别。MQL5 中的此类函数组类似于：

- **MatMul**: 两个矩阵的矩阵乘积
- **Power**: 将一个方矩阵自乘到指定整数次幂
- **Inner**: 两个矩阵的内积
- **Outer**: 两个矩阵或两个向量的外积
- **Kron**: 两个矩阵、一个矩阵和一个向量、一个向量和一个矩阵或者两个向量的克罗内克积
- **CorrCoef**: 计算一个矩阵的行之间或列之间，或向量之间的皮尔逊相关性
- **Cov**: 计算一个矩阵的行或列的或两个向量之间的协方差矩阵
- **Correlate**: 计算两个向量的互相关性（交叉相关性）
- **Convolve**: 计算两个向量的离散线性卷积
- **Dot**: 两个向量的标量积

为给出关于如何管理这些方法的一个总体概念，我们将给出他们的原型（按以下顺序：从矩阵开始，经过混合矩阵-向量，到向量）。

```
matrix<T> matrix<T>::MatMul(const matrix<T> &m)
matrix<T> matrix<T>::Power(const int power)
matrix<T> matrix<T>::Inner(const matrix<T> &m)
matrix<T> matrix<T>::Outer(const matrix<T> &m)
matrix<T> matrix<T>::Kron(const matrix<T> &m)
matrix<T> matrix<T>::Kron(const vector<T> &v)
matrix<T> matrix<T>::CorrCoef(const bool rows = true)
matrix<T> matrix<T>::Cov(const bool rows = true)
matrix<T> vector<T>::Cov(const vector<T> &v)
T vector<T>::CorrCoef(const vector<T> &v)
vector<T> vector<T>::Correlate(const vector<T> &v, ENUM_VECTOR_CONVOLVE mode)
vector<T> vector<T>::Convolve(const vector<T> &v, ENUM_VECTOR_CONVOLVE mode)
matrix<T> vector<T>::Outer(const vector<T> &v)
matrix<T> vector<T>::Kron(const matrix<T> &m)
matrix<T> vector<T>::Kron(const vector<T> &v)
T vector<T>::Dot(const vector<T> &v)
```

下面的简单示例演示了如何使用 **MatMul** 方法计算两个矩阵的矩阵乘积：

```

matrix a = {{1, 0, 0},
            {0, 1, 0}};
matrix b = {{4, 1},
            {2, 2},
            {1, 3}};
matrix c1 = a.MatMul(b);
matrix c2 = b.MatMul(a);
Print("c1 = \n", c1);
Print("c2 = \n", c2);
/*
    c1 =
    [[4,1]
    [2,2]]
    c2 =
    [[4,1,0]
    [2,2,0]
    [1,3,0]]
*/

```

$A[M,N] * B[N,K] = C[M,K]$ 形式的矩阵可以相乘，即，第一矩阵中的列数必须等于第二矩阵中的行数。如果维数不一致，则结果为空矩阵。

当矩阵和向量相乘时，允许两个选项：

- 横向量（行）乘以右边的矩阵，向量的长度等于矩阵行数
- 矩阵乘以右边的纵向量（列），向量的长度等于矩阵的列数

向量也可以彼此相乘。在 `MatMul` 中，这始终等效于一个行向量乘以一个列向量的点积（`Dot` 法），并且“当一个列向量乘以一个行向量并得到一个矩阵”这一选择由另一个方法支持：`Outer`。

我们演示向量 `v5` 乘以向量 `v3` 的 `Outer` 积，并以相反顺序演示。在两种情况下，列向量指示在左边，行向量指示在右边。

```

vector v3 = {1, 2, 3};
vector v5 = {1, 2, 3, 4, 5};
Print("v5 = \n", v5);
Print("v3 = \n", v3);
Print("v5.Outer(v3) = m[5,3] \n", v5.Outer(v3));
Print("v3.Outer(v5) = m[3,5] \n", v3.Outer(v5));
/*
v5 =
[1,2,3,4,5]
v3 =
[1,2,3]
v5.Outer(v3) = m[5,3]
[[1,2,3]
 [2,4,6]
 [3,6,9]
 [4,8,12]
 [5,10,15]]
v3.Outer(v5) = m[3,5]
[[1,2,3,4,5]
 [2,4,6,8,10]
 [3,6,9,12,15]]
*/

```

4.10.9 矩阵变换 (分解)

矩阵变换是在处理数据时最常用的运算。然而，很多复杂的变换不能分析性地以绝对精度执行。

矩阵变换 (或者称为分解)是将一个矩阵分解为若干组成部分的方法，这样更容易进行比较复杂的矩阵运算。矩阵分解方法也称为矩阵因式分解法，它是线性代数算法 (诸如求解线性方程系统以及计算矩阵或行列式的倒数等)的依据。

尤其是奇异值分解 (SVD) 广泛用于机器学习中，它允许你将原始矩阵表示为三个其它矩阵的积。SVD 分解用于解决各种问题，从最小二乘拟合到压缩以及图像识别等。

可用方法列表。

- 🕒 **Cholesky**: 计算乔里斯基分解
- 🕒 **Eig**: 计算方矩阵的本征值及右本征向量
- 🕒 **Eig Vals**: 计算通用矩阵的本征值
- 🕒 **LU**: 以下三角矩阵和上三角矩阵的积实施矩阵的 LU 因式分解
- 🕒 **LUP**: 以部分旋转实施 LUP 因式分解，这是仅有行置换的 LU 因式分解: $PA=LU$
- 🕒 **QR**: 实施矩阵的 QR 因式分解
- 🕒 **SVD**: 奇异值分解

下面是方法原型。

```

bool matrix<T>::Cholesky(matrix<T> &L)
bool matrix<T>::Eig(matrix<T> &eigen_vectors, vector<T> &eigen_values)
bool matrix<T>::EigVals(vector<T> &eigen_values)
bool matrix<T>::LU(matrix<T> &L, matrix<T> &U)
bool matrix<T>::LUP(matrix<T> &L, matrix<T> &U, matrix<T> &P)
bool matrix<T>::QR(matrix<T> &Q, matrix<T> &R)
bool matrix<T>::SVD(matrix<T> &U, matrix<T> &V, vector<T> &singular_values)

```

我们来看一个使用 SVD 方法的奇异值分解示例（参见文件 *MatrixSVD.mq5*）。首先，我们初始化原始矩阵。

```

matrix a = {{0, 1, 2, 3, 4, 5, 6, 7, 8}};
a = a - 4;
a.Reshape(3, 3);
Print("matrix a \n", a);

```

现在我们进行一个 SVD 分解：

```

matrix U, V;
vector singular_values;
a.SVD(U, V, singular_values);
Print("U \n", U);
Print("V \n", V);
Print("singular_values = ", singular_values);

```

我们检查展开式：必须保持以下相等性： $U * \text{"singular diagonal"} * V = A$ 。

```

matrix matrix_s;
matrix_s.Diag(singular_values);
Print("matrix_s \n", matrix_s);
matrix matrix_vt = V.Transpose();
Print("matrix_vt \n", matrix_vt);
matrix matrix_usvt = (U.MatMul(matrix_s)).MatMul(matrix_vt);
Print("matrix_usvt \n", matrix_usvt);

```

我们比较生成的矩阵和原始矩阵之间是否有误差。

```

ulong errors = (int)a.Compare(matrix_usvt, 1e-9);
Print("errors=", errors);

```

日志应会类似这样：

```

matrix a
[[-4,-3,-2]
 [-1,0,1]
 [2,3,4]]
U
[[-0.7071067811865474,0.5773502691896254,0.408248290463863]
 [-6.827109697437648e-17,0.5773502691896253,-0.8164965809277256]
 [0.7071067811865472,0.5773502691896255,0.4082482904638627]]
V
[[0.5773502691896258,-0.7071067811865474,-0.408248290463863]
 [0.5773502691896258,1.779939029415334e-16,0.8164965809277258]
 [0.5773502691896256,0.7071067811865474,-0.408248290463863]]
singular_values = [7.348469228349533,2.449489742783175,3.277709923350408e-17]

matrix_s
[[7.348469228349533,0,0]
 [0,2.449489742783175,0]
 [0,0,3.277709923350408e-17]]
matrix_vt
[[0.5773502691896258,0.5773502691896258,0.5773502691896256]
 [-0.7071067811865474,1.779939029415334e-16,0.7071067811865474]
 [-0.408248290463863,0.8164965809277258,-0.408248290463863]]
matrix_usvt
[[-3.999999999999997,-2.999999999999999,-2]
 [-0.9999999999999981,-5.977974170712231e-17,0.9999999999999974]
 [2,2.999999999999999,3.999999999999996]]
errors=0

```

应用 *Convolve* 方法的另一个实际案例包括在 [机器学习方法](#) 一节的示例中。

4.10.10 获取统计数据

下面列出的方法旨在用于获取矩阵和向量的描述性统计。它们全部适用于向量或整个矩阵，也适用于给定矩阵轴（横轴或纵轴）。当完全应用到一个对象时，这些函数返回一个标量（奇异）。沿任何轴应用到矩阵时，返回一个向量。

原型的总体外观：

```

T vector<T>::Method(const vector<T> &v)
T matrix<T>::Method(const matrix<T> &m)
vector<T> matrix<T>::Method(const matrix<T> &m, const int axis)

```

方法列表：

- ⌚ *ArgMax*, *ArgMin*: 查找最大和最小值索引
- ⌚ *Max*, *Min*: 查找最大和最小值
- ⌚ *Ptp*: 查找一系列值
- ⌚ *Sum*, *Prod*: 计算元素的和或积
- ⌚ *CumSum*, *CumProd*: 计算元素的累加和或积
- ⌚ *Median*, *Mean*, *Average*: 计算中位数、算术平均值或加权算术平均值
- ⌚ *Std*, *Var*: 计算标准差和方差

⌚ **Percentile, Quantile**: 计算百分位数和分位数

⌚ **RegressionMetric**: 计算预定义回归指标之一，诸如矩阵/向量数据的回归线偏差等

计算当前交易品种和时间范围的系列柱线（以点数为单位）的标准差和百分数的示例在 *MatrixStdPercentile.mq5* 文件中提供。

```
input int BarCount = 1000;
input int BarOffset = 0;

void OnStart()
{
    // getting current chart quotes
    matrix rates;
    rates.CopyRates(_Symbol, _Period, COPY_RATES_OPEN | COPY_RATES_CLOSE,
        BarOffset, BarCount);
    // calculating price increments on bars
    vector delta = MathRound((rates.Row(1) - rates.Row(0)) / _Point);
    // debug print of initial bars
    rates.Resize(rates.Rows(), 10);
    Normalize(rates);
    Print(rates);
    // printing increment metrics
    PRTF((int)delta.Std());
    PRTF((int)delta.Percentile(90));
    PRTF((int)delta.Percentile(10));
}
```

日志：

```
(EURUSD,H1)  [[1.00832,1.00808,1.00901,1.00887,1.00728,1.00577,1.00485,1.00652,1.005
(EURUSD,H1)  [1.00808,1.00901,1.00887,1.00728,1.00577,1.00485,1.00655,1.00537,1.004
(EURUSD,H1)  (int)delta.Std()=163 / ok
(EURUSD,H1)  (int)delta.Percentile(90)=170 / ok
(EURUSD,H1)  (int)delta.Percentile(10)=-161 / ok
```

4.10.11 矩阵和向量的特征

下面几种方法可用于获取矩阵的主要特征：

⌚ **Rows, Cols**: 矩阵中的行数和列数。

⌚ **Norm**: 预定义矩阵范数之一（ENUM_MATRIX_NORM）

⌚ **Cond**: 矩阵的条件数

⌚ **Det**: 方形非退化矩阵的行列式

⌚ **SLogDet**: 计算矩阵行列式的符号和算法

⌚ **Rank**: 矩阵的秩

⌚ **Trace**: 矩阵对角线的元素总和（迹）

⌚ **Spectrum**: 矩阵谱（作为其本征值的集合）

此外，为向量定义了以下特征：

⌚ **Size**: 向量的长度

⌚ **Norm**: 预定义向量范数之一 (ENUM_VECTOR_NORM)

对象大小 (以及其中元素的索引) 使用 *ulong* 类型的值。

```
ulong matrix<T>::Rows()
ulong matrix<T>::Cols()
ulong vector<T>::Size()
```

大部分其它特征为实数。

```
double vector<T>::Norm(const ENUM_VECTOR_NORM norm, const int norm_p = 2)
double matrix<T>::Norm(const ENUM_MATRIX_NORM norm)
double matrix<T>::Cond(const ENUM_MATRIX_NORM norm)
double matrix<T>::Det()
double matrix<T>::SLogDet(int &sign)
double matrix<T>::Trace()
```

秩和谱分别为整数和向量。

```
int matrix<T>::Rank()
vector matrix<T>::Spectrum()
```

矩阵秩计算示例：

```
matrix a = matrix::Eye(4, 4);
Print("matrix a (eye)\n", a);
Print("a.Rank()=", a.Rank());

a[3, 3] = 0;
Print("matrix a (defective eye)\n", a);
Print("a.Rank()=", a.Rank());

matrix b = matrix::Ones(1, 4);
Print("b \n", b);
Print("b.Rank()=", b.Rank());

matrix zeros = matrix::Zeros(4, 1);
Print("zeros \n", zeros);
Print("zeros.Rank()=", zeros.Rank());
```

下面是脚本执行结果：

```

matrix a (eye)
[[1,0,0,0]
 [0,1,0,0]
 [0,0,1,0]
 [0,0,0,1]]
a.Rank()=4

matrix a (defective eye)
[[1,0,0,0]
 [0,1,0,0]
 [0,0,1,0]
 [0,0,0,0]]
a.Rank()=3

b
[[1,1,1,1]]
b.Rank()=1

zeros
[[0]
 [0]
 [0]
 [0]]
zeros.Rank()=0

```

4.10.12 解方程

在机器学习方法和优化问题中，通常要求解线性方程系统。MQL5 包含四个方法可根据矩阵类型求解该等方程。

- **Solve** 求解线性矩阵方程或线性代数方程系统
- **LstSq** 近似求解线性代数方程系统（对于非方形矩阵或退化矩阵）
- **Inv** 使用约当-高斯法计算相对于方形非奇异矩阵的乘法逆元矩阵
- **Plnv** 使用摩尔-彭罗斯法计算伪逆矩阵

下面是方法原型。

```

vector<T> matrix<T>::Solve(const vector<T> b)
vector<T> matrix<T>::LstSq(const vector<T> b)
matrix<T> matrix<T>::Inv()
matrix<T> matrix<T>::Plnv()

```

Solve 和 **LstSq** 方法表示求解 $A \cdot X = B$ 形式的方程系统，其中 A 是矩阵， B 是使用函数的值（“因变量”）通过一个参数传递的向量。

我们尝试应用 **LstSq** 方法求解方程系统，该系统是一个理想组合交易模型（在本例中，我们将分析主要外汇货币组合）。为此，对于给定数量的“历史”柱线，我们需要找出均线趋向于一条持续增长直线时的每种货币的手数。

我们将第 i 个货币对表示为 S_i 。其在 k 索引柱线处的报价等于 $S_i[k]$ 。柱线编号从过去到将来编号，如同由 **CopyRates** 方法填充的矩阵和向量中那样。因此，收集的用于训练模型的报价开头

对应于标记为编号 0 的柱线,但在时间线上,它将是 (根据算法设置处理的)最老的历史柱线。它右边的柱线 (将来方向)编号为 1、2,依此类推,直至用户要求计算的总柱线数。

在第 0 条柱线 and 第 N 条柱线之间的交易品种价格的变化决定截至第 N 条柱线时的益/损。

考虑到货币对,我们得到第 1 柱线的以下利润方程:

$$(S1[1] - S1[0]) * X1 + (S2[1] - S2[0]) * X2 + \dots + (Sm[1] - Sm[0]) * Xm = B$$

其中 m 是总字符数, X_i 是每个交易品种的手数, B 是浮动利润 (若锁定利润则为条件性余额)。

为简洁起见,我们缩短表示法。我们从绝对值转到价格增量 ($A_i[k] = S_i[k] - S_i[0]$)。考虑跨柱线的移动,我们将获得若干虚拟余额曲线的表达式:

$$\begin{aligned} A1[1] * X1 + A2[1] * X2 + \dots + Am[1] * Xm &= B[1] \\ A1[2] * X1 + A2[2] * X2 + \dots + Am[2] * Xm &= B[2] \\ &\dots \\ A1[K] * X1 + A2[K] * X2 + \dots + Am[K] * Xm &= B[K] \end{aligned}$$

成功交易的特征为每条柱线的利润恒定,即,右手边向量 B 的模型是是一个单增函数,理想情况下是一条直线。

我们实施该模型并基于报价为其选择 X 系数。由于我们尚不知道应用程序 API,因此无法编写一个成熟的交易策略。我们只是使用来自标准头文件 *Graphic.mqh* 的 *GraphPlot* 函数构建一个虚拟余额图 (我们已经使用它演示过 [数学函数](#))。

新示例的完整源代码在 *MatrixForexBasket.mq5* 脚本中。

在输入参数中,让用户选择用于数据采样的总柱线数 (*BarCount*),以及该选择范围内的柱线偏移量 (*BarOffset*),该偏移量对应“假设过去”结束和“假设未来”开始的分界柱线编号。

将对“假设过去”构建一个模型 (将求解上述线性方程系统),并对“假设未来”执行前向测试。

```
input int BarCount = 20; // BarCount (known "history" and "future")
input int BarOffset = 10; // BarOffset (where "future" starts)
input ENUM_CURVE_TYPE CurveType = CURVE_LINES;
```

为了以一个理想的余额填充向量,我们编写 *ConstantGrow* 函数:稍后初始化期间将使用这个函数。

```
void ConstantGrow(vector &v)
{
    for(ulong i = 0; i < v.Size(); ++i)
    {
        v[i] = (double)(i + 1);
    }
}
```

交易的金融工具 (主要外汇对)列表在 *OnStart* 函数的开头硬设定,需进行编辑以适合你的要求和交易环境。

```

void OnStart()
{
    const string symbols[] =
    {
        "EURUSD", "GBPUSD", "USDJPY", "USDCAD",
        "USDCHF", "AUDUSD", "NZDUSD"
    };
    const int size = ArraySize(symbols);
    ...

```

我们创建将在其中添加交易品种报价的 *rates* 矩阵、具有所需余额曲线的 *model* 向量以及用于逐交易品种请求柱线收盘价的辅助 *close* 向量（它包含的数据将被复制到 *rates* 矩阵的列中）。

```

matrix rates(BarCount, size);
vector model(BarCount - BarOffset, ConstantGrow);
vector close;

```

在一个交易品种循环中，我们将收盘价复制到 *close* 向量，计算价格增量，并将它们写入到 *rates* 矩阵的对应列中。

```

for(int i = 0; i < size; i++)
{
    if(close.CopyRates(symbols[i], _Period, COPY_RATES_CLOSE, 0, BarCount))
    {
        // calculate increments (profit on all and on each bar in one line)
        close -= close[0];
        // adjust the profit to the pip value
        close *= SymbolInfoDouble(symbols[i], SYMBOL_TRADE_TICK_VALUE) /
            SymbolInfoDouble(symbols[i], SYMBOL_TRADE_TICK_SIZE);
        // place the vector in the matrix column
        rates.Col(close, i);
    }
    else
    {
        Print("vector.CopyRates(%d, COPY_RATES_CLOSE) failed. Error ",
            symbols[i], _LastError);
        return;
    }
}
...

```

我们将在第五章中探讨一个价格点值（以存款货币计）的计算。

还有一点务必要注意，具有相同索引的柱线可能对不同的金融工具有不同的时间戳，例如，如果某个国家因而节假日而休市（外汇以外的交易品种在理论上可能具有不同的交易时段计划表）。为解决这一问题，我们需要对报价更深度的分析，在将它们插入 *rates* 矩阵之前，需考虑到柱线时间及其同步。在这里，为了简单起见，同时也因为外汇市场大多数时间根据相同规则运作，因此我们不这么做。

我们将矩阵拆分为两部分：初始部分将用于求解（模拟历史优化），后续部分将用于前向测试（计算后续余额变化）。

```

matrix split[];
if(BarOffset > 0)
{
    // training on BarCount - BarOffset bars
    // check on BarOffset bars
    ulong parts[] = {BarCount - BarOffset, BarOffset};
    rates.Split(parts, 0, split);
}

// solve the system of linear equations for the model
vector x = (BarOffset > 0) ? split[0].LstSq(model) : rates.LstSq(model);
Print("Solution (lots per symbol): ");
Print(x);
...

```

获得解后，我们来构建样本的所有柱线的余额曲线（理想的“历史”部分将在开头，然后是“将来”部分，后者不会用于调整模型）。

```

vector balance = vector::Zeros(BarCount);
for(int i = 1; i < BarCount; ++i)
{
    balance[i] = 0;
    for(int j = 0; j < size; ++j)
    {
        balance[i] += (float)(rates[i][j] * x[j]);
    }
}
...

```

我们根据 R2 标准估算解的质量。

```

if(BarOffset > 0)
{
    // make a copy of the balance
    vector backtest = balance;
    // select only "historical" bars for backtesting
    backtest.Resize(BarCount - BarOffset);
    // bars for the forward test have to be copied manually
    vector forward(BarOffset);
    for(int i = 0; i < BarOffset; ++i)
    {
        forward[i] = balance[BarCount - BarOffset + i];
    }
    // compute regression metrics independently for both parts
    Print("Backtest R2 = ", backtest.RegressionMetric(REGRESSION_R2));
    Print("Forward R2 = ", forward.RegressionMetric(REGRESSION_R2));
}
else
{
    Print("R2 = ", balance.RegressionMetric(REGRESSION_R2));
}
...

```

要在图表上显示余额曲线，你需要将数据从向量转移到数组。

```

double array[];
balance.Swap(array);

// print the values of the changing balance with an accuracy of 2 digits
Print("Balance: ");
ArrayPrint(array, 2);

// draw the balance curve in the chart object ("backtest" and "forward")
GraphPlot(array, CurveType);
}

```

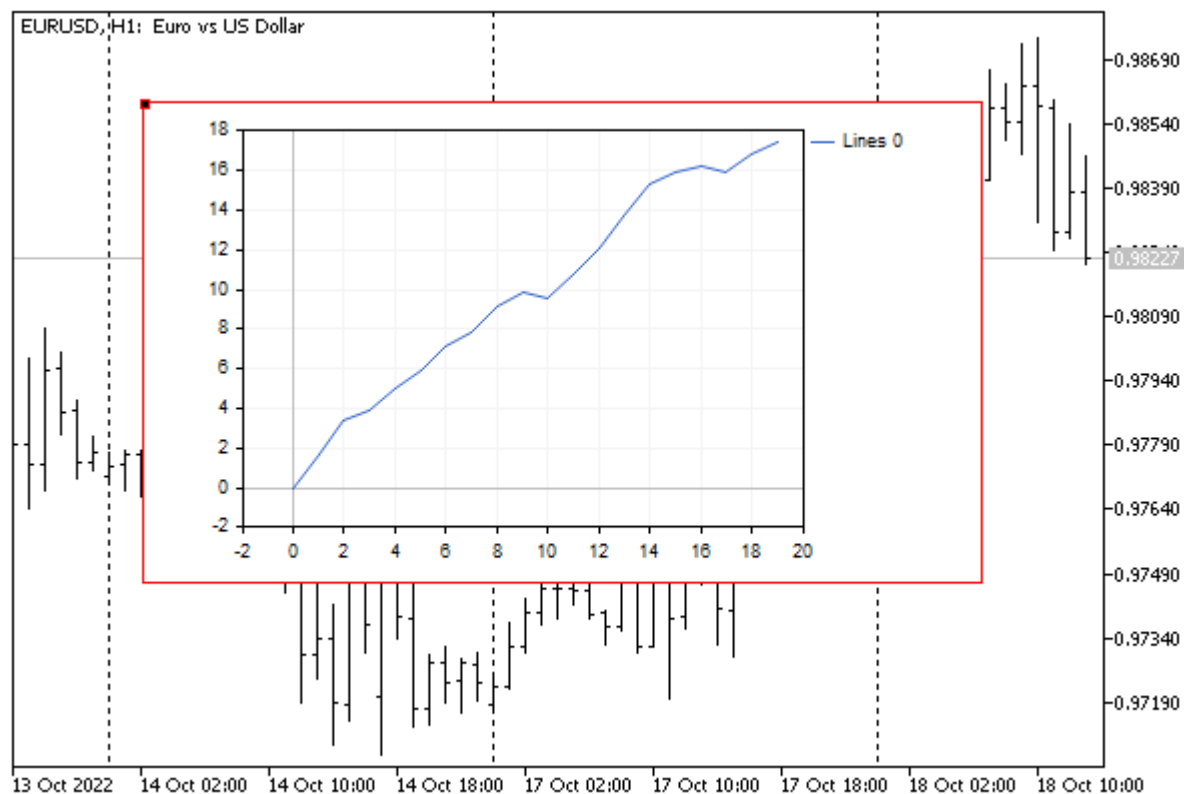
下面是通过在 EURUSD,H1 上运行该脚本获得的一个日志的示例。

```

Solution (lots per symbol):
[-0.0057809334,-0.0079846876,0.0088985749,-0.0041461736,-0.010710154,-0.0025694175,0.
Backtest R2 = 0.9896645616246145
Forward R2 = 0.8667852183780984
Balance:
0.00  1.68  3.38  3.90  5.04  5.92  7.09  7.86  9.17  9.88
9.55 10.77 12.06 13.67 15.35 15.89 16.28 15.91 16.85 16.58

```

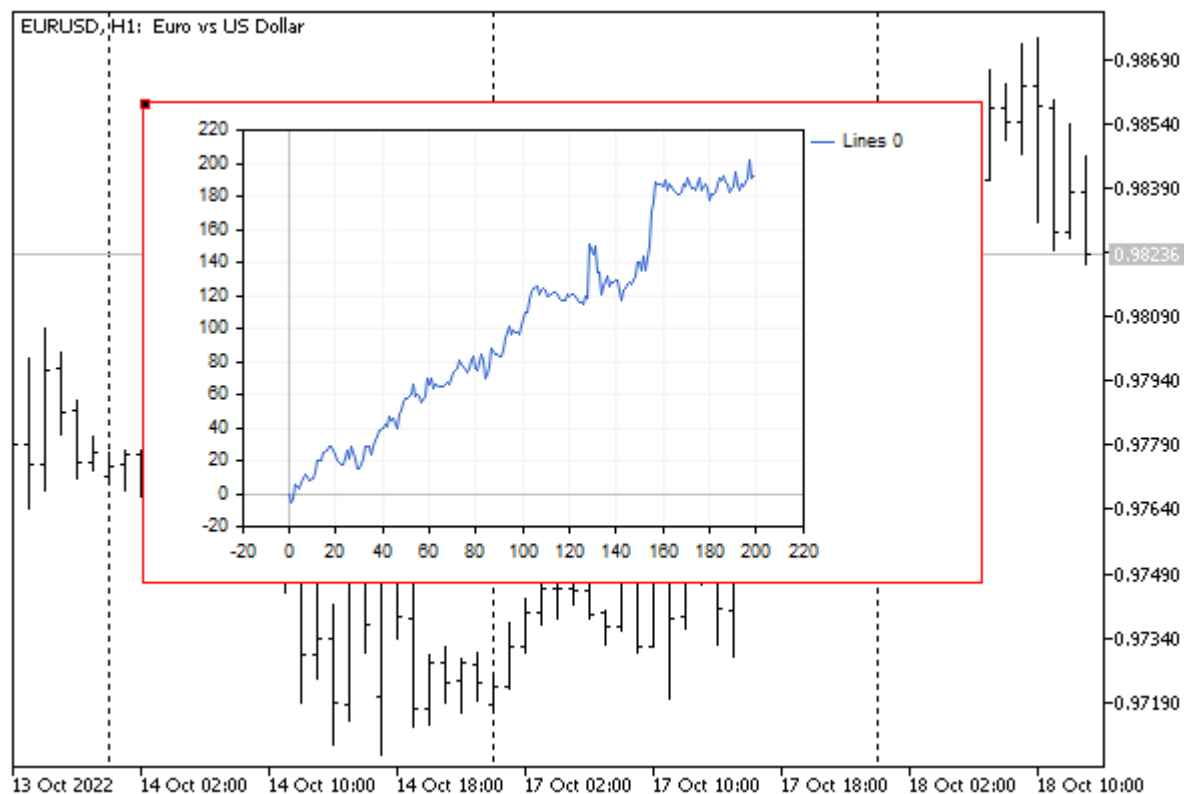
下面是虚拟余额曲线的示例。



根据决定按“手数”交易货币组合的虚拟余额

左半部分的形状更均匀且 R^2 更高，这并不奇怪，因为模型 (α 变量) 专门针对该部分进行调整。

出于兴趣，我们将训练和验证深度增加 10 倍，即我们将在参数中设置 $BarCount = 200$ 和 $BarOffset = 100$ 。我们将得到一个新图片。



根据决定按“手数”交易货币组合的虚拟余额

“将来”部分看起来更平滑，我们甚至可以说，它能够继续增长只是凭运气，尽管这是一个如此简单的模型。原则上，在前向测试期间，虚拟余额曲线会大大退化，开始下滑。

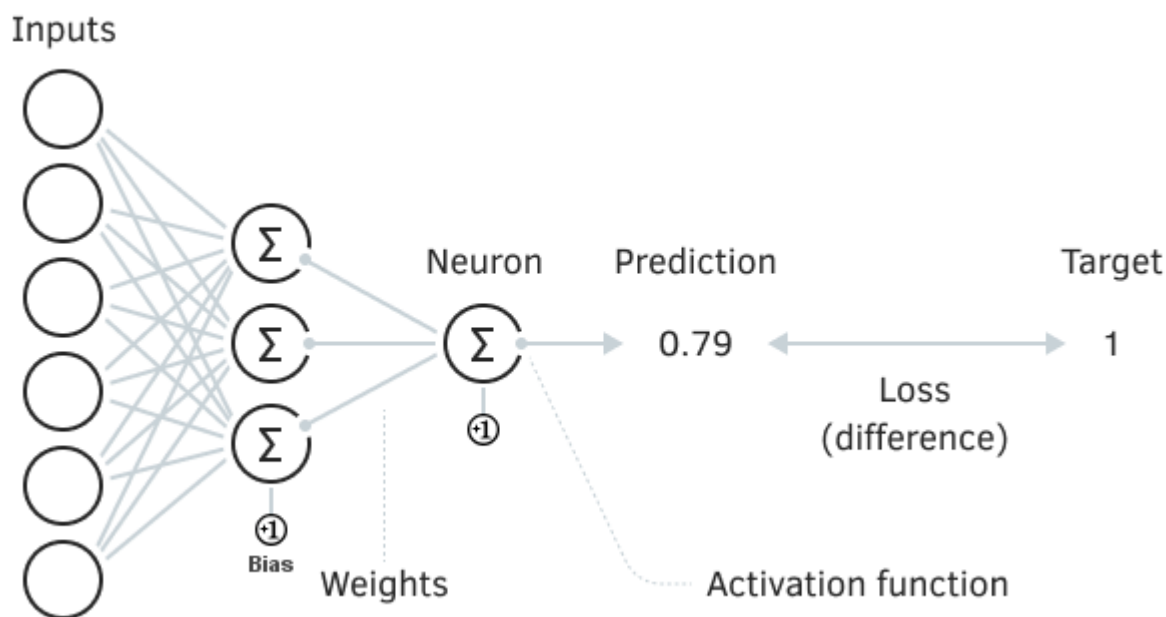
务必需要注意的是，为了测试该模型，我们采用从系统的“原样”解获得的 X 值，而在实践中，我们需要将它们标准化为最小手数和手梯度，这将对结果造成负面影响，使它们更接近现实。

4.10.13 机器学习方法

在矩阵和向量的内置方法当中，有几个是在机器学习任务中所需要的，尤其是在神经网络实现中。

顾名思义，神经网络是诸多神经元（即基本计算单元）的集合。称它们为“基本”，是因为它们执行比较简单的计算：通常一个神经元有一系列应用于某些输入信号的权重系数，此后信号的加权和被馈入函数（即非线性转换器）。

使用激活函数放大弱信号并限制过强的信号，防止进入饱和状态（实数计算溢出）。然而，最重要的在于，非线性赋予网络新的计算能力，能够解决更复杂的问题。

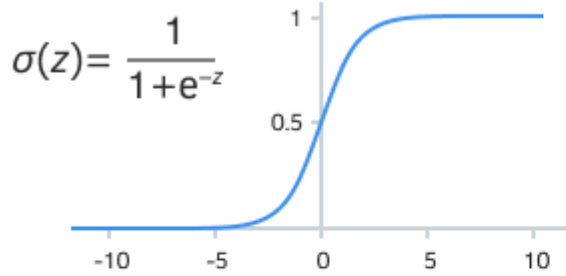


基础神经网络

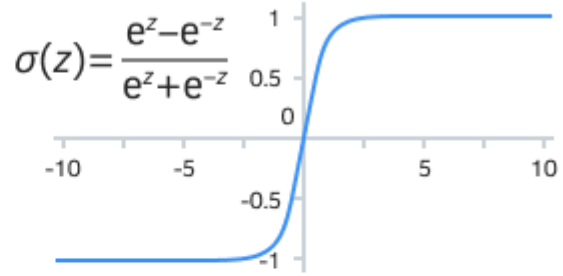
神经网络的强大之处体现为将大量神经元组合起来并在它们之间建立连接。通常，神经元被组织为层（可类比矩阵或向量），包括具有递归（循环）连接的层，并且还可能有效果各个相同的激活函数。这就能够使用各种算法分析体积性数据，尤其是通过找出它们当中隐藏的模式来分析。

请注意，若每个神经元中没有非线性特性，多层神经网络可等效表示为单层网络，其系数由所有层的矩阵积获得 $W_{total} = W_1 * W_2 * \dots * W_L$ ，其中，1..L 是层的编号）。这将是一个简单的线性加法器。因此，激活函数的重要性是有数学依据的。

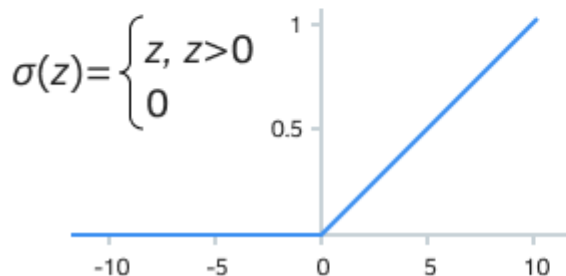
Sigmoid



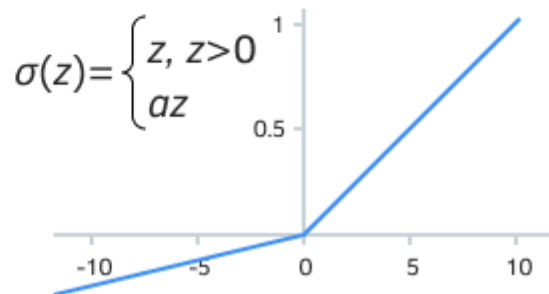
Tanh



ReLU



LeakyReLU



一些最著名的激活函数

神经网络的主要分类标准之一是根据使用的学习算法分为有监督学习网络和无监督学习网络。有监督学习网络要求人类专家向原始数据集提供所需的输出（例如交易系统状态的离散标记，或者暗示价格增量的数字指标）。无监督网络自行标识数据中的群集。

在任何情况下，训练神经网络的任务是找出使训练和测试样本误差最小化的参数，为此就需要使用损失函数：该函数可定性和定量估计目标网络应答和接收到的网络应答之间的误差。

神经网络之所以能广泛应用，最关键因素包括：选择对于信息性和相互独立的预测因子（经分析的特征）、根据学习算法具体细节进行数据转换（规范化和清理）以及网络架构和规模优化。请注意，使用机器学习算法并不保证成功。

在这里，我们不会涉及神经网络的理论、它们的分类以及要解决的典型任务。这一主题太过于广泛。感兴趣的读者可以参考 mql5.com 函数及其它来源的文章。

MQL5 提供三种机器学习方法，这些方法已成为矩阵和向量 API 的一部分。

- **Activation** 计算激活函数值
- **Derivative** 计算激活函数的导数值
- **Loss** 计算损失函数值

激活函数的导数能够基于在学习过程中变化的模型误差来高效更新模型参数。

前两个方法将结果写到传递的向量/矩阵，并返回成功指示器 (*true* 或 *false*)，而损失函数返回一个数字。我们提供它们的原型（我们将二者均标记在 *object<T>* 类型下，*matrix<T>* 和 *vector<T>*）：

```
bool object<T>::Activation(object<T> &out, ENUM_ACTIVATION_FUNCTION activation)
bool object<T>::Derivative(object<T> &out, ENUM_ACTIVATION_FUNCTION loss)
T object<T>::Loss(const object<T> &target, ENUM_LOSS_FUNCTION loss)
```

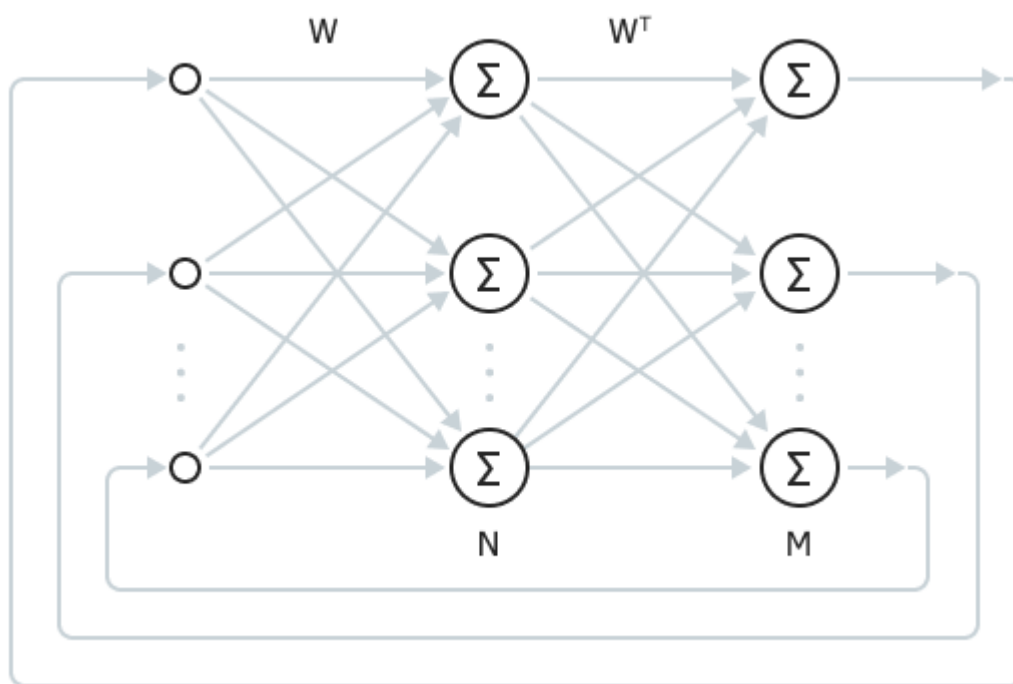
某些激活函数允许使用第三个自变量（可选）来设置参数。

请参阅 `MQL5` 文档,了解 `ENUM_ACTIVATION_FUNCTION` 枚举中支持的激活函数和 `ENUM_LOSS_FUNCTION` 枚举中损失函数列表。

作为介绍性示例,我们来探讨分析真实分时报价流的问题。有些交易员将分时报价视为垃圾噪声,而有些交易员却实行基于分时报价的高频交易。有一种假设是高频算法原则上对大资本玩家有优势,因为高频算法完全基于软件对价格信息的处理。基于此,我们将提出这样一个假设:由于做市商的当前活动的机器人的作用,在分时报价流中有一种短期记忆效应。然后,可以利用机器学习法找出这一依赖性,并预测若干将来分时报价。

机器学习始终涉及提出假设,为假设合成模型以及在实践中试验这些假设。显然,并非总是能够获得富有成效的假设。这是一个漫长的试错过程,失败是改进和新思路的源泉。

我们将使用最简单的神经网络类型之一:双向联想记忆 (BAM)。这样一种网络只有两层:输入和输出层。在输出中会形成某种响应 (关联) 以响应输入信号。层大小可以各不相同。当大小相同时,结果是霍普菲尔德网络。



全连接双向联想记忆

使用此类网络,我们将比较 N 个最近前面分时报价和 M 个后面预测的分时报价,从近期历史到给定深度构建训练样本。分时报价将以价格涨跌幅的正负值形式馈入网络,并转换为二进制值 $[+1, -1]$ (二进制信号是 BAM 和霍普菲尔德网络中简洁形式的编码)。

BAM 最重要的优势是几乎即时 (相比大多数其它迭代方法) 的学习过程,该过程包括计算权重矩阵。我们将在下面给出公式。

然而,这种简洁性也有劣势:BAM 容量 (它能记住的图像数量) 限于最小的层大小,且前提是满足训练样本向量中 $+1$ 和 -1 的特殊分布条件。

因此,对于我们的情况来说,网络会对训练样本中的所有报价序列进行泛化,随后在常规运行过程中,根据新输入的分时报价序列,回滚到某个存储的模式。实际效果取决于很多因素,包括网络规模和设置、当前分时报价流的特征等等。

由于假定分时报价流仅有短期记忆,因此最好是实时或接近实时重新训练网络,因为训练实际上会简化为若干个矩阵运算。

因此，为了使网络记住关联图像（对我们情况来说，指分时报价流的过去和将来），需要以下方程：

$$\mathbf{W} = \sum_i (\mathbf{A}_i^T \mathbf{B}_i)$$

其中 $\mathbf{W}$ 是网络的权重矩阵。对输入向量 $\mathbf{A}_i$ 和对应输出向量 $\mathbf{B}_i$ 的所有成对积求和。

然后，当网络正在运行时，我们将输入图像提供给第一层，对其应用将 $\mathbf{W}$ 矩阵，从而激活第二层，在第二层中计算每个神经元的激活函数。然后，使用转置的 $\mathbf{W}^T$ 矩阵，信号传播回第一层，也在神经元中应用激活函数。此刻，输入图像不再到达第一层，即自由振荡过程在网络中继续。该过程持续到网络神经元中的信号变化稳定下来（即变得小于某个预定义值）。

在此状态中，网络的第二层包含找到的关联输出图像 - 预测。

我们在 *MatrixMachineLearning.mq5* 脚本中实施这一机器学习场景。

在输入参数中，你可以设置从历史记录中请求的上次分时报价总数 (*TicksToLoad*)，以及设置分配其中多少用于测试 (*TicksToTest*)。相应地，模型（权重）将基于 (*TicksToLoad* - *TicksToTest*) 分时报价。

```
input int TicksToLoad = 100;
input int TicksToTest = 50;
input int PredictorSize = 20;
input int ForecastSize = 10;
```

此外，在输入变量中，选择输入向量（已知分时报价数 *PredictorSize*）和输出向量（将来分时报价数 *ForecastSize*）的大小。

在 *OnStart* 函数的开头请求分时报价。在这里，我们只处理 *Ask* 价格。但是你也可以随交易量一起添加 *Bid* 和 *Last* 过程。

```
void OnStart()
{
    vector ticks;
    ticks.CopyTicks(_Symbol, COPY_TICKS_ALL | COPY_TICKS_ASK, 0, TicksToLoad);
    ...
}
```

我们将分时报价拆分为训练集和测试集。

```
vector ask1(n - TicksToTest);
for(int i = 0; i < n - TicksToTest; ++i)
{
    ask1[i] = ticks[i];
}

vector ask2(TicksToTest);
for(int i = 0; i < TicksToTest; ++i)
{
    ask2[i] = ticks[i + TicksToLoad - TicksToTest];
}
...
}
```

为计算价格增量，我们使用具有附加向量 {+1, -1} 的 *Convolve* 方法。请注意，具有增量的向量将比原始向量少 1 个元素。

```
vector differentiator = {+1, -1};
vector deltas = ask1.Convolve(differentiator, VECTOR_CONVOLVE_VALID);
...
```

根据 VECTOR_CONVOLVE_VALID 算法的卷积意味着仅完全向量重叠才会被纳入考虑范围 (即较小的向量随较大向量顺序偏移, 不超出其边界)。其它类型的卷积允许向量仅与一个元素或元素的一半重叠 (在此情况下, 其余元素超出对应向量, 卷积值表现出边界效应)。

为将增量的连续值转换为单位脉冲 (正和负, 取决于向量的初始符号), 我们使用辅助函数 *Binary* (这里未显示): 它返回向量的新副本, 其中每个元素为 +1 或 -1。

```
vector inputs = Binary(deltas);
```

基于收到的输入系列, 我们使用 *TrainWeights* 函数计算 W 神经网络权重矩阵。我们将稍后了解该函数的结构。目前我们需要注意的是, 向该函数传递了 *PredictorSize* 和 *ForecastSize* 参数, 从而能够分别根据输入和输出 BAM 层的大小将连续分时报价系列拆分为成对输入和输出向量集。

```
matrix W = TrainWeights(inputs, PredictorSize, ForecastSize);
Print("Check training on backtest: ");
CheckWeights(W, inputs);
...
```

网络训练完成后, 我们立即在训练集上检查其精度: 只是为了确保网络已经过训练。这由 *CheckWeights* 函数实现。

然而, 更重要的是检查网络如何处理不熟悉的测试数据。为此, 我们求第二个向量 *ask2* 的微分并将其二值化, 然后将其发送至 *CheckWeights*。

```
vector test = Binary(ask2.Convolve(differentiator, VECTOR_CONVOLVE_VALID));
Print("Check training on forwardtest: ");
CheckWeights(W, test);
...
}
```

现在我们来熟悉 *TrainWeights* 函数, 我们在其中定义矩阵 A 和 B , 以将向量从传递的输入系列 (从 *data* 向量) “切下”。

```

template<typename T>
matrix<T> TrainWeights(const vector<T> &data, const uint predictor, const uint response,
    const uint start = 0, const uint _stop = 0, const uint step = 1)
{
    const uint sample = predictor + response;
    const uint stop = _stop <= start ? (uint)data.Size() : _stop;
    const uint n = (stop - sample + 1 - start) / step;
    matrix<T> A(n, predictor), B(n, response);

    ulong k = 0;
    for(ulong i = start; i < stop - sample + 1; i += step, ++k)
    {
        for(ulong j = 0; j < predictor; ++j)
        {
            A[k][j] = data[start + i * step + j];
        }
        for(ulong j = 0; j < response; ++j)
        {
            B[k][j] = data[start + i * step + j + predictor];
        }
    }
    ...
}

```

每个连续的 *A* 模式从数量等于 *predictor* 的连续分时报价获得，而对应的将来模式从以下 *response* 元素获得。只要总数据量允许，该窗口就会一次一个元素地向右移动，形成更多新的图像对。图像按行编号，而其中的分时报价按列编号。

接下来，我们应为权重矩阵 *W* 分配内存，并使用矩阵方法填充：我们使用 *Outer* 按顺序从 *A* 到 *B* 乘以行，然后执行矩阵求和。

```

matrix<T> W = matrix<T>::Zeros(predictor, response);

for(ulong i = 0; i < k; ++i)
{
    W += A.Row(i).Outer(B.Row(i));
}

return W;
}

```

CheckWeights 函数可为神经网络执行类似操作，其权重系数以现成形式在第一个 *W* 自变量中传递。训练向量的大小提取自 *W* 矩阵本身。

```

template<typename T>
void CheckWeights(const matrix<T> &W,
    const vector<T> &data,
    const uint start = 0, const uint _stop = 0, const uint step = 1)
{
    const uint predictor = (uint)W.Rows();
    const uint response = (uint)W.Cols();
    const uint sample = predictor + response;
    const uint stop = _stop <= start ? (uint)data.Size() : _stop;
    const uint n = (stop - sample + 1 - start) / step;
    matrix<T> A(n, predictor), B(n, response);

    ulong k = 0;
    for(ulong i = start; i < stop - sample + 1; i += step, ++k)
    {
        for(ulong j = 0; j < predictor; ++j)
        {
            A[k][j] = data[start + i * step + j];
        }
        for(ulong j = 0; j < response; ++j)
        {
            B[k][j] = data[start + i * step + j + predictor];
        }
    }

    const matrix<T> w = W.Transpose();
    ...
}

```

矩阵 A 和 B 在此情况下不是用于计算 W，而是充当用于测试的向量的“供应者”。我们还需要一个 W 的转置副本，用以计算从第二个网络层到第一个网络层的返回信号。

网络允许的瞬态过程迭代次数（超过该次数后进行收敛）取决于 *limit* 常量。

```

const uint limit = 100;

int positive = 0;
int negative = 0;
int average = 0;

```

需要变量 *positive*、*negative* 和 *average* 来计算成功和不成功预测的统计数据，以评估训练质量。

此外，网络在测试模式对的循环中激活，并采用其最终响应。每个下一输入向量被写入向量 **a**，并且输出层 **b** 以零填充。之后，为从 **a** 到 **b** 的信号传输使用矩阵 W 以及应用激活函数 AF_TANH 启动迭代，并且为从 **b** 到 **a** 的反馈信号同样使用 AF_TANH 启动迭代。该过程一直持续到 *limit* 循环（这不太可能），或者直至满足收敛条件，在该条件下，**a** 和 **b** 神经元状态向量几乎不会改变（这里我们使用 *Compare* 法以及来自前一迭代的 **x** 和 **y** 向量）。

```

for(ulong i = 0; i < k; ++i)
{
    vector a = A.Row(i);
    vector b = vector::Zeros(responce);
    vector x, y;
    uint j = 0;

    for( ; j < limit; ++j)
    {
        x = a;
        y = b;
        a.MatMul(W).Activation(b, AF_TANH);
        b.MatMul(w).Activation(a, AF_TANH);
        if(!a.Compare(x, 0.00001) && !b.Compare(y, 0.00001)) break;
    }

    Binarize(a);
    Binarize(b);
    ...
}

```

达到稳定状态后，我们使用 *Binarize* 函数（其类似于先前提到的 *Binary* 函数，但会原位更改向量的状态）将神经元的状态从连续（实数）转为二进制 +1 和 -1。

现在，我们只需要统计具有目标向量的输出层中的匹配项数量。为此，执行向量的标量乘法。结果为正数，表示正确猜想的分时报价数超过不正确数。总命中计数累积在“平均数”中。

```

const int match = (int)(b.Dot(B.Row(i)));
if(match > 0) positive++;
else if(match < 0) negative++;

average += match; // 0 in match means 50/50 precision (i.e. random guessing)
}

```

所有测试样本的循环完成后，我们显示统计数据。

```

float skew = (float)average / k; // average number of matches per vector

PrintFormat("Count=%d Positive=%d Negative=%d Accuracy=%.2f%%",
    k, positive, negative, ((skew + responce) / 2 / responce) * 100);
}

```

该脚本还包括 *RunWeights* 函数，该函数表示神经网络针对来自最后 *predictor* 分时报价的在线向量的一次工作运行（由其权重矩阵 *W* 运行）。该函数将返回一个带有将来分时报价估计数的向量。


```

template<typename T>
vector<T> RunWeights(const matrix<T> &W, const vector<T> &data)
{
    const uint predictor = (uint)W.Rows();
    const uint response = (uint)W.Cols();
    vector a = data;
    vector b = vector::Zeros(response);

    vector x, y;
    uint j = 0;
    const uint limit = LIMIT;
    const matrix<T> w = W.Transpose();

    for( ; j < limit; ++j)
    {
        x = a;
        y = b;
        a.MatMul(W).Activation(b, AF_TANH);
        b.MatMul(w).Activation(a, AF_TANH);
        if(!a.Compare(x, 0.00001) && !b.Compare(y, 0.00001)) break;
    }

    Binarize(b);

    return b;
}

```

在 *OnStart* 结束时，我们暂停执行 1 秒钟（以便等待一定概率的新分时报价），请求最后 *PredictorSize + 1* 分时报价数（别忘了 +1 以便区分），然后在线作出它们的预测。

```

void OnStart()
{
    ...
    Sleep(1000);
    vector ask3;
    ask3.CopyTicks(_Symbol, COPY_TICKS_ALL | COPY_TICKS_ASK, 0, PredictorSize + 1);
    vector online = Binary(ask3.Convolve(differentiator, VECTOR_CONVOLVE_VALID));
    Print("Online: ", online);
    vector forecast = RunWeights(W, online);
    Print("Forecast: ", forecast);
}

```

以默认设置对星期五晚上的 EURUSD 运行该脚本得到以下结果。

```
Check training on backtest:  
Count=20 Positive=20 Negative=0 Accuracy=85.50%  
Check training on forwardtest:  
Count=20 Positive=12 Negative=2 Accuracy=58.50%  
Online: [1,1,1,1,-1,-1,-1,1,-1,1,1,-1,1,1,-1,-1,1,1,-1,-1]  
Forecast: [-1,1,-1,1,-1,-1,1,1,-1,1]
```

交易品种和时间未提及，因为市场情况会极大地影响算法适用性以及特定网络配置。当市场开放时，每次运行脚本时，你将获得新的结果，因为越来越多的分时报价进来。这是与短期记忆形式假设一致的预期行为。

我们可以看到，训练精度可接受，但测试数据方面明显降低，可能低于 50%。

到这里，我们顺利从编程进入了科研领域。MQL5 中内置的机器学习工具套件允许你实施神经网络和分析器的很多其它的配置，有不同的交易策略和原理用于准备初始数据。

第五章在 MQL5 中创建应用程序

在这一部分，我们将深入研究 API 中与解决算法交易应用问题相关的各个部分：金融数据分析和处理、使用图形对象实现可视化对象和标记、常规操作自动化以及交互式用户交互。

我们首先介绍创建 MQL 程序的一般原则、程序类型、特点以及终端中的事件模型。然后，我们将探讨如何访问时间序列、处理图表和图形对象。最后，我们将分别分析每种 MQL 程序的创建原则和使用原则。

MetaTrader 5 的活跃用户一定记得，终端支持五种类型的程序：

- 技术指标，用于以时间序列形式计算任意指标，可在主图表窗口或单独面板（子窗口）中可视化；
- EA 交易，提供自动或半自动交易功能；
- 脚本，用于按需执行一次性辅助任务；
- 服务，用于以连续模式在后台执行任务；
- 库，具有特定独立功能的编译模块，在加载过程中动态地连接到其他类型的 MQL 程序（这与在编译阶段静态包含的头文件有本质区别）。

在本书的前几部分中，在我们掌握编程基础和常见内置函数的过程中，我们已经以脚本和服务为例进行过说明。选择这两类程序作为示例，是因为它们比其他程序更为简单。现在，我们将深入地讲解，并在其中添加更实用、更常见的指标。

借助指标和图表，我们将学习一些同样适用于 EA 交易的技术。但考虑到 EA 交易本质上更为复杂，我们将其实际开发内容移至单独的第 6 章，其中不仅包括订单的自动执行和交易策略的形式化，还包括这些策略的回测和优化。

就指标而言，众所周知，MetaTrader 5 自带一系列内置标准指标。在这一部分中，我们将学习如何通过编程方式使用这些指标，以及如何从零开始创建我们自己的指标，还学习如何基于其他指标创建指标。

所有已编译的指标、EA 交易、脚本和服务都会显示在 MetaTrader 5 的导航器中。库不是独立程序，因此在层级结构中并没有专门的分支，尽管从统一管理所有二进制模块的角度来看，这样的设计原本会更为便捷。我们稍后就会看到，如果程序依赖于特定库，那么如果相应的库不存在，程序就无法运行。但目前只能在文件管理器中查看库是否存在。

 面向交易员的 MQL5 编程 - 本书中的源代码第五章

 本书中的示例也可在 [公共项目](#) \MQL5\Shared Projects\MQL5Book 中获取

5.1 执行 MQL 程序的一般原则

所有 MQL 程序都可以根据其功能和特点大致分为几组。

大多数程序是在图表的环境中运行，如 EA 交易、指标和脚本。换言之，只有在通过 **Navigator** 中的 **Attach to Chart** 上下文菜单命令，或者通过从 **Navigator** 拖放到图表上，被附加到某个打开的图表后，程序才开始执行。

相比之下，服务不能放置在图表上，因为它们是为了在后台执行长时间的循环操作。例如，在服务中，你可以创建一个 **自定义交易品种**，然后使用网络函数接收其数据，并在无限循环中

持续更新数据。服务的另一个符合逻辑的应用是监控交易账户和网络连接，作为向用户通知通信问题的解决方案的一部分。

务必要注意的是，指标和 EA 交易在终端工作会话之间会被保存在图表上。换言之，若用户在图表上运行一个指标，然后在没有显式删除该指标的情况下关闭 MetaTrader 5，则下次启动终端时，该指标将连同图表一起恢复，包括其所有设置。

顺便说一下，将指标和 EA 交易关联到图表是模板的基础（请参阅 [文档](#)）。用户可以创建一组要在图表上使用的程序，对它们进行配置，并将该组程序保存到扩展名为 *tpl* 的特殊文件中。此操作通过上下文菜单命令 **Templates -> Save** 完成的。之后，你可以将模板应用到任何新图表上（命令 **Templates -> Upload**）并运行所有关联的程序。模板默认存储在 **MQL5/Profiles/Templates/** 目录中。

与图表关联的另一个结果是，关闭图表会导致卸载之前对图表运行的所有 MQL 程序。不过，MetaTrader 5 会以特定方式保存所有已关闭的图表（至少保存一段时间），因此，如果图表被意外关闭，可以恢复该图表及所有程序（以及 [图形对象](#)），方法是使用 **File -> Open Remote** 命令。

如果终端因某种原因无法加载图表文件，则 MQL 程序的整个状态（设置和位置）将丢失。基本上，这一原则同样适用于 [图形对象](#)，程序可以根据自身需要添加这些对象，并期望它们会保留在图表上。备份图表。每个图表都是扩展名为 *chr* 的文件。这些文件默认存储在 **MQL5/Profiles/Charts/Default/** 目录中。这是安装平台时创建的标准配置文件。你可以使用菜单命令 **File -> Profiles** 创建其他配置文件并在它们之间切换（请参阅 [文档](#)）。

如需停止 EA 交易并将其从图表移除，可使用上下文菜单命令 **Expert list**（在图表窗口点击右键调用）进行操作。将打开 **Experts** 对话框，其中列出了终端中正在运行的所有 EA 交易。在此列表中，选择不再需要的 EA 交易，然后选择 **Remove**。

指标也可以通过类似上下文菜单命令 **Indicator List** 来显式移除。将打开一个对话框，其中列出了当前图表上正在运行的指标，在此对话框中，你可以选择特定指标，然后点击 **Remove** 按钮。此外，大多数指标会在图表上显示各种图形结构，例如线条和柱状图，这些图形结构也可以使用相关上下文菜单命令来删除。

与指标和 EA 交易不同，脚本不会永久附加到图表上。在标准模式下，如果脚本执行的是一次性操作，则当所分配的任务完成后，脚本会自动从图表中移除。如果脚本包含定期重复执行操作的循环，脚本当然会持续运行，直到循环以某种方式中断，但不会超过会话结束时间。关闭终端会使脚本从图表上分离。重新启动 MetaTrader 5 后，脚本不会在图表上恢复。

请注意，如果将图表切换到另一个交易品种或时间范围，该图表上运行的脚本将会被卸载。但指标和 EA 交易会继续运行，不过，它们会重新初始化。这两者的初始化规则有所不同。具体细节将在 [各类程序启动和停止的特征](#) 章节中详细讨论。

每个图表上可放置一个 EA 交易、一个脚本，但可以放置任意数量的指标。EA 交易、脚本以及所有指标将并行（同时）运行。

至于服务，其已创建并正在运行的实例在终端加载后自动恢复。可以使用 **Navigator** 窗口中 **Services** 部分的上下文菜单来停止或删除服务实例。

下表以汇总形式概述了上述特性。

程序类型	链接到图表	在图标上的数量	会话恢复
指标	需要	多个	通过图表或模板
EA 交易	需要	最多 1 个	通过图表或模板
脚本	需要	最多 1 个	不支持
服务	不支持	0	通过终端

所有 MQL 程序都在客户端终端中执行，因此只有在终端打开时才能运行。如需实现对账户的持续程序监控，请使用 VPS。

5.1.1 设计各种类型的 MQL 程序

程序类型是 MQL5 中的基本特性。在 C++ 等通用编程语言中，任何程序都可以以任意方向进行开发，例如添加图形界面或者通过网络从服务器上传数据，但 MQL 程序与此不同，而是依据其用途被划分为特定类别。例如，通过指标实现带可视化功能的技术性时间序列分析，但指标本身无法用于交易。反过来，EA 交易可以使用交易 API 函数，但它们缺少指标缓冲区（用于绘制线条的数组）。

因此，在解决某个具体应用问题时，开发者应将问题分解为多个部分，并且每一部分的功能都应 与特定类型的专长相符。当然，简单情况下，单个 MQL 程序即可满足需求，但有时最优技术方案并不显而易见。例如，你会怎样实现砖形图的绘制：是作为 指标，还是作为服务生成的 自定义交易品种 亦或是直接在 EA 交易作为特定计算？所有选项都可行。

MQL 程序的类型由多个因素决定。

首先，每种类型的程序在 MQL5 工作目录中都有独立的文件夹。我们在 第一章 的引言中已经提到过这一点，并列出了相关文件夹。因此，对于指标、EA 交易、脚本和服务，指定文件夹分别为 *Indicators*、*Experts*、*Scripts* 和 *Services*。Libraries子文件夹为 MQL5 文件夹中的库保留。在这些文件夹中，你可以按任意配置方式组织嵌套文件夹树形结构。

编译 mql5 文件生成的二进制文件（即扩展名为 *ex5* 的成品程序）会与源 mql5 文件存放在同一目录下。然而，我们还应提及 MetaEditor 中的项目（扩展名为 *mqlproj* 的文件），这部分内容我们将在 项目一章进行分析。当开发一个项目时，成品会在项目旁边的目录中创建。当在 MetaEditor 中通过 MQL5 向导创建程序时（通过 *File -> New*），源文件默认会放置在与程序类型相对应的文件夹中。如果不小心将程序复制到了错误的目录，也不会造成严重问题：例如，EA 交易不会变成指标，指标也不会变成 EA 交易。可以直接在编辑器中、在 *Navigator*窗口内或者在外部文件管理器中将其移动到所需位置。在 *Navigator*中，每种程序类型都会显示一个特殊图标。

程序在 MQL5 目录中特定类型子文件夹内的存放位置，并不能决定该特定 MQL 程序的类型。程序类型是根据可执行文件的内容确定的，而可执行文件反过来又是由编译器根据源代码中的特性指令和语句生成的。

按程序类型划分的文件夹层级结构是为了便于使用。建议遵循这种层级结构，除非是处理一组相关项目（包含不同类型的程序），这种情况下将它们存储在单独的目录中更符合逻辑。

第二，每种类型的程序都有如下特征：支持一组有限的、特定的系统事件，这些事件会激活该程序。我们将在单独章节中了解 [事件处理函数概述](#)。要在程序中接收特定类型的事件，需要描述一个具有预定义原型（名称、参数列表、返回值）处理程序函数。

例如，我们已经看到，在脚本和服务中，工作在 **OnStart** 函数中开始，由于它是其中唯一的函数，因此可以将其称为主要的“入口点”，终端通过此入口点将控制权移交给应用程序代码。在其他类型的程序中，情况会稍微复杂一些。总的来说，我们注意到，程序类型的特征取决于一组特定的处理程序，其中一些处理程序可能是必需的，而另一些是可选的（但同时，对于其他类型的程序来说是不可接受的）。具体来说，指标需要 **OnCalculate** 函数（如果没有它，指标将无法编译，编译器会生成错误）。然而 EA 交易中并不使用此函数。

第三，某些类型的程序需要特殊的 **#property** 指令。在 [程序通用特性](#) 一章中，我们已经了解了可以在所有类型的程序中使用的指令。然而，还有其他一些专门的指令。例如，在我们提到的与服务相关的任务中，我们遇到了 **#property service** 指令，该指令使程序成为一个服务。如果没有这个指令，即使将程序放在 **Services** 文件夹中，它也无法在后台运行。

同样，**#property library** 指令在创建库时起着决定性作用。所有这些指令特性将在相应类型程序的章节中详细讨论。

在按以下顺序（从上到下，直到找到第一个匹配项）确定 MQL 程序类型时，将考虑指令和事件处理程序的组合：

- ⌚ 指标：存在 **OnCalculate** 处理程序
- ⌚ 库：**#property library**
- ⌚ 脚本：存在 **OnStart** 处理程序且不存在 **#property service**
- ⌚ 服务：存在 **OnStart** 处理程序且 **#property service**
- ⌚ EA 交易：存在任何其他处理程序

关于这些特性对编译器的影响示例，将在 [事件处理函数概述](#) 章节中给出。

对于以上提到的所有要点，还需额外考虑一点：程序类型由主编译模块决定，即扩展名为 **m q5** 的文件，在主编译模块中，可使用 **#include** 指令来包含其他目录中的源文件。通过这种方式引入的所有函数，其级别与直接编写在主 **.m q5** 文件中的函数完全一致。

另一方面，**#property** 指令仅在编译的主 **m q5** 文件中生效。如果这些指令出现在通过 **#include** 引入的程序文件中，则将被忽略。

主 **m q5** 文件并不需要实际包含事件处理程序函数。将部分或全部算法放置在 **m qh** 头文件中，然后将这些头文件包含在一个或多个程序中，这样做是完全可以接受的。例如，我们可以在 **m qh** 文件中实现一个包含一组有用操作的 **OnStart** 处理程序，然后通过 **#include** 在两个独立程序（脚本和服务）中使用该处理程序。

同时要注意，存在通用事件处理程序并不是将通用算法片段分离到头文件中的唯一原因。例如，你可以在指标和 EA 交易中使用相同的计算公式，而将它们的事件处理程序留在主程序模块中。

虽然习惯上将包含文件称为头文件，并给它们加上 **m qh** 扩展名，但从技术层面来讲这并不是必需的。在技术上允许（但不推荐）在一个 **m q5** 文件中包含另一个 **m q5** 文件或 **txt** 文件。这些文件可能包含一些遗留代码，或者说可以说包含以常量对数组进行初始化的操作。被包含的 **m q5** 文件并不会因此成为主文件。

应该确保程序中仅包含特定程序类型的事件处理函数，且避免函数重复（如你所知，函数通过名称和参数列表的组合来识别：[函数重载](#) 仅在参数集不同的情况下才被允许）这通常通过各种预处理器指令来实现。例如，在某些程序中包含第三方 **m q5** 脚本文件前，通过定义

`#define OnStart OnStartPrevious`宏，我们实际上会将其中描述的 `OnStart` 函数转变为 `OnStartPrevious`，并且我们可以从我们自己的事件处理程序中正常调用它。

然而，只有在特殊情况下由于某些原因无法修改所包含的 `mql5` 文件的源代码时，这种方法才有意义，特别是，当无法通过将相关算法选择到单独的头文件中的函数或类来对其进行结构化处理时。

根据与用户交互的原则，MQL 程序可以分为交互式程序和实用程序。

交互式程序（指标和 EA 交易）可以处理事件（即在软件环境中因响应用户按下键盘上的按钮、移动鼠标、更改窗口大小等操作而触发的事件），以及与接收报价数据或计时器操作相关的许多其他事件。

而实用程序（服务和脚本）仅依据启动时设置的输入变量来运行，并且不对系统中的事件做出响应。

除了所有这些类型的程序之外，还有库。库始终作为其他类型的 MQL 程序（四种主要类型之一）的一部分来执行，因此不具备任何独特的特征或行为。特别是，库不能直接从终端接收事件，也没有自己的线程（请参阅下一节）。同一个库可以连接到许多程序，并且这种连接是在每个父程序启动时动态发生的。在库章节中，我们将学习如何描述库的导出 API 并将其导入到父程序中。

5.1.2 线程

简单来说，程序可以看作是开发者为计算机生成的一系列语句。计算机中语句的主要执行者是中央处理器。现代计算机通常配备多核处理器，相当于拥有多个独立处理器。然而，用户可能想要并行运行的程序数量几乎是无限的。因此，程序的数量始终比可用内核/处理器数量多很多倍。为此，每个内核实际上会将其工作时间分配给多个不同程序：先分配 1 毫秒执行一个程序的语句，再分配 1 毫秒执行另一个程序的语句，接着是第三个程序，依此类推，循环往复。由于这种切换速度非常快，用户不会察觉到这一点，好像所有程序都是同时并行执行的。

为了让处理器能够暂停一个程序的语句执行，并在之后（当它悄悄切换到其他“并行”程序的语句后）从之前的位置恢复执行，它必须能够以某种方式保存和恢复每个程序的中间状态：包括当前语句、变量、可能已打开的文件、网络连接等。程序正常运行所需的全部资源和数据，及其在语句序列中的当前位置，统称为程序的执行上下文。实际上，操作系统的设计目的是根据用户（或其他程序）的请求为每个程序创建这样的上下文。每一个这样的活动上下文都被称为线程。许多程序自身需要多个线程，因为其功能涉及并行维持多个活动。MetaTrader 5 同样需要多个线程来加载多个交易品种的报价、绘制图表以及响应用户操作。此外，系统也会向 MQL 程序分配单独的线程。

MQL 程序执行环境为每个程序分配的线程数不超过一个。每个 EA 交易、脚本和服务都严格获得一个线程。至于指标，同一金融工具上运行的所有指标会分配一个线程。而且，同一个线程负责显示相应交易品种的图表，所以不建议让它执行密集型计算。否则，用户界面将变得无响应：用户操作的处理会被延迟，或者窗口甚至会变得无响应。所有其他类型的 MQL 程序的线程都不与界面绑定，因此可以让处理器执行任何复杂的任务。

线程的一个重要特性源自其定义和用途：它仅支持按顺序依次执行语句的语句。在一个线程中，一次只执行一条语句。如果在程序中编写了一个无限循环，线程将卡在这条指令上，并且永远不会执行其下面的指令。冗长的计算也可能产生类似无限循环的效果：它们会占用处理器，阻碍执行用户可能期望的其他操作。因此，指标中保持高效计算对图形界面的流畅运行至关重要。

然而，对于其他类型的 MQL 程序，应重视线程安排。在接下来的章节中，我们将详细了解作为 MQL 程序入口点的特殊事件处理函数。单线程模型意味着在处理一个事件的过程中，程序对可能同时发生的其他事件是无响应的。因此，终端会为每个程序维护一个事件队列。在下一章节中，我们会更详细地探讨这一点。

为了在实践中体验单线程的效果，我们将 [指标的优势与局限 \(IndBarIndex.mq5\)](#) 章节中查看一个简单的示例。我们选择指标作为示例，是因为它们不仅针对每个交易品种共享一个线程，还直接在图表上显示结果，这使得潜在的问题能够最直观地显现出来。

5.1.3 事件处理函数概述

终端或测试代理通过调用特定函数将控制权转移给 MQL 程序（即执行这些程序），这些函数由 MQL 开发者在应用程序代码中定义，用于处理预定义事件。此类函数必须有一个指定的原型，包括函数名、参数列表（数量、类型和顺序）以及返回类型。

每个函数的名称都与事件的含义相对应，并添加前缀 **On**。例如，**OnStart** 是启动（Start）脚本和服务的主函数；当脚本放置在图表上或服务实例启动时，终端会调用该函数。

就本书而言，我们将用相同名称指代一个事件及其相应的处理程序。

下表列出了所有事件类型以及支持这些事件类型的程序  - 指标、 - EA 交易、 - 脚本、 - 服务。对这些事件的详细说明将在相应程序类型的章节中展开阐述。许多因素会引发初始化和反初始化事件，例如：将程序放置在图表上、更改程序设置、更改图表的交易品种/时间框架（或模板、配置文件）、更改账户以及其他情况（请参阅 [各类程序启动和停止的特征一章](#)）。

程序类型 事件/处理程序					说明
OnStart	-	-	●	●	启动/执行
OnInit	+	+	-	-	加载后初始化 (请参见 各类程序启动和停止的特征章节中的详细说明)
OnDeinit	+	+	-	-	停止和卸载前初始化
OnTick	-	+	-	-	获取新的价格 (分时报价)
OnCalculate	●	-	-	-	由于收到新价格或同步旧价格而触发的指标重新计算请求
OnTimer	+	+	-	-	以指定频率触发计时器
OnTrade	-	+	-	-	在服务器上完成交易操作
OnTradeTransaction	-	+	-	-	更改交易账户状态 (订单、交易、头寸)
OnBookEvent	+	+	-	-	在订单簿中更改
OnChartEvent	+	+	-	-	图表上的用户或 MQL 程序操作
OnTester	-	+	-	-	单次测试周期结束
OnTesterInit	-	+	-	-	优化前初始化
OnTesterDeinit	-	+	-	-	优化后反初始化
OnTesterPass	-	+	-	-	从测试代理接收优化数据

必需的处理程序以符号 “●” 标识, 可选的处理程序以 “+” 标记。

虽然处理程序函数主要由运行时环境调用, 但你也可在从你自己的源代码中调用它们。例如, 当 EA 交易需要在启动后立即基于现有报价进行计算, 并且即使在没有分时报价 (例如在周末) 的情况下也需如此, 可以在退出 **OnInit** 之前调用 **OnTick**。或者, 将计算分离到一个单独的函数中, 并从 **OnInit** 和 **OnTick** 中调用它, 这也是合乎逻辑的。但需快速执行初始化函数的工作, 如果计算耗时较长, 应该通过 **计时器** 执行该计算。

所有 MQL 程序 (库除外) 必须包含至少一个事件处理程序。否则, 编译器将生成 “未找到事件处理函数” 错误。

在没有设置其他类型的 #property 指令的情况下, 某些处理程序函数的存在决定了程序的类型。例如, 拥有 **OnCalculate** 处理程序会导致生成指标 (即使它位于另一文件夹中, 例如脚本或 EA 交易)。存在 **OnStart** 处理程序 (如果没有 **OnCalculate** 处理程序) 意味着创建一个脚本。同时, 如果

指标除有 *OnCalculate* 外, 还存在 *OnStart*, 我们会得到一个编译器警告 “在非脚本程序中定义了 *OnStart* 函数”。

本书包含两个文件: *AllInOne.mq5* 和 *AllInOne.mqh*。头文件描述了所有主要事件处理程序几乎为空的模板。除了将处理程序的名称输出到日志以外, 它们不包含其他内容。我们将在特定类型的 MQL 程序的章节中考虑使用每个处理程序的语法和详情。此文件的意义在于提供一个实验领域, 根据某些处理程序和特性指令 (*#property*) 的存在情况来编译不同类型的程序。

某些组合可能导致编译错误或警告。

如果编译成功, 程序加载后将自动通过以下代码行记录生成的程序类型:

```
const string type =
    PRTF(EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE)));
```

关于枚举 *ENUM_PROGRAM_TYPE* 和函数 *MQLInfoInteger*, 我们在 [程序类型和许可证](#) 学习。

AllInOne.mq5 文件 (包括 *AllInOne.mqh*) 最初存放于 *MQL5Book/Scripts/p5/* 目录中, 但可复制至任意其他文件夹, 包括 *Navigator* 中的相邻分支 (例如 EA 交易或指标的文件夹)。在该文件的注释中, 保留了用于连接某些程序汇编配置的选项。默认情况下, 如果不编辑该文件, 就会生成 EA 交易。

```
//+-----+
//| Uncomment the following line to get the service |
//| NB: also activate #define _OnStart OnStart |
//+-----+
//#property service

//+-----+
//| Uncomment the following line to get a library |
//+-----+
//#property library

//+-----+
//| Uncomment the following line to get a script or |
//| service (#property service must be enabled) |
//+-----+
//#define _OnStart OnStart

//+-----+
//| Uncomment one of the following two lines for the indicator |
//+-----+
//#define _OnCalculate1 OnCalculate
//#define _OnCalculate2 OnCalculate

#include <MQL5Book/AllInOne.mqh>
```

如果将该程序附加到图表上, 我们将会在日志中得到一条记录:

```
EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE))=PROGRAM_EXPERT / ok
OnInit
OnChartEvent
OnTick
OnTick
OnTick
...
```

此外，如果市场处于交易时段，很可能会从 *OnTick* 处理程序生成一系列记录。

如果你用不同的名称复制该 *m q5* 文件，例如，取消对 *#property service* 指令的注释，编译器将生成服务，但会返回一些警告。

```
no OnStart function defined in the script
OnInit function is useless for scripts
OnDeinit function is useless for scripts
```

其中第一个警告是关于缺少 *OnStart* 函数的，这实际上很重要，因为创建服务实例时，系统不会调用任何函数，仅初始化全局变量。但由于这个原因，日志（终端中的 *Experts* 选项卡）仍会显示 *PROGRAM_SERVICE* 类型。但通常，在服务以及脚本中都会假定存在 *OnStart* 函数。

另外两个警告出现是因为我们的头文件包含了适用于各种情况的处理程序，编译器提醒我们 *OnInit* 和 *OnDeinit* 是没有意义的（终端不会调用它们，甚至不会将它们包含在程序的二进制镜像中）。当然，在实际程序中不应该有这样的警告，即应调用所有处理程序，同时通过预处理指令进行条件编译，从源代码中移除所有冗余内容（无论是物理删除还是逻辑删除）。

如果创建了 *AllInOne.m q5* 的另一个副本，同时不仅激活了 *#property service* 指令，还激活了 *#define _OnStart OnStart* 宏，则在编译了副本后，你会获得一个完全可用的服务。该副本在启动后，不仅会显示其类型名称，还会显示被触发的处理程序 *OnStart* 的名称。

如果需要启用或禁用标准处理程序 *OnStart*，这个宏是必需的。在 *AllInOne.mqh* 文本中，此函数的描述如下：

```
void _OnStart() // "extra" underline makes the function customized
{
    Print(__FUNCTION__);
}
```

以下划线开头的名称表明它不是标准处理程序，而只是具有相似原型的用户自定义函数。当我们包含宏时，在编译过程中，编译器会将 *_OnStart* 替换为 *OnStart*，从而将其转换为标准处理程序。如果我们显式命名了 *OnStart* 函数，则根据确定 MQL 程序类型的特征的优先级（请参阅 [各类程序的特征](#) 章节），你无法得到 EA 交易模板（因为 *OnStart* 将程序标识为脚本或服务）。

同理，需要使用 *_OnCalculate1* 或 *_OnCalculate2* 宏进行自定义编译，是为了可以选择“隐藏”具有标准名称 *OnCalculate* 的处理程序：否则，如果该处理程序存在，我们将始终获得指标。

如果你在程序的下一个副本中激活 *#define _OnCalculate1 OnCalculate* 宏，则会得到指标示例（即使为空，不执行任何操作）。正如我们稍后将会看到的，对于指标而言，*OnCalculate* 处理程序有两种不同形式，因此以编号名称（*_OnCalculate1* 和 *_OnCalculate2*）呈现。如果你在图表上运行指标，可以在日志中看到 *OnCalculate* 事件（在分时报价到达时）和 *OnChartEvent* 事件（例如在鼠标点击时）的名称。

在编译指标时，编译器将生成两条警告：

```
no indicator window property is defined, indicator_chart_window is applied
no indicator plot defined for indicator
```

这是因为指标作为数据可视化工具，其代码中需要一些特定设置（当前示例没有这些设置）。不过现阶段只是初步了解不同类型程序，这一点并不重要。但在后续章节中，我们将讲解如何在指标中描述它们的特性和数组，这些特性和数组决定了图表可视化内容和可视化形式。届时这些警告信息将消失。

事件队列

当一个新事件发生时，系统必须将其传递给在相应图表上运行的所有 MQL 程序。由于 MQL 程序的单线程执行模型（请参阅 [线程](#) 章节），可能会出现这样的情况：当前事件尚未处理完毕时，下一个事件就已到达。对于这种情况，终端会为每个交互式 MQL 程序维护一个事件队列。其中的所有事件会按照接收顺序依次进行处理。

事件队列的大小有限制。因此，程序编写不合理可能会因操作缓慢而导致其事件队列溢出。发生溢出时，新事件将被丢弃，不会进入队列排队。

如果不能足够快速地处理事件，可能会对用户体验或数据质量产生负面影响（试想在记录 [市场深度](#) 变化时错过了几条消息）。为解决这一问题，可以寻找更高效的算法，或者让多个相互关联的 MQL 程序并行运行（例如，将计算分配给一个指标，而在 EA 交易中仅读取已准备好的数据）。

需要记住的是，终端不会将所有事件都放入队列，而是有选择性地进行操作。某些事件类型的处理遵循“队列中同类型事件不超过一个”原则。例如，如果队列中已经有一个 **OnTick** 事件，或者正在处理该事件，则新的 **OnTick** 事件不会被加入队列。如果队列中已经存在 **OnTimer** 事件或图表更改事件，则新的同类型事件也同样会被丢弃（忽略）。这种情况仅针对程序的特定实例。其他不那么“繁忙”的程序仍会接收到这条消息。

我们并未提供此类事件类型的完整列表，因为终端开发者可能会更改这种跳过“重叠”事件的优化方式。

这种根据传入事件来组织程序运行的模式被称为事件驱动。也可以称为异步，因为在程序队列中对事件进行排队和事件提取（及处理）发生在不同的时刻（理想情况下，两者之间的间隔极短，但现实并非总是如此）。然而，在四种 MQL 程序中，只有指标和 EA 交易完全遵循这种模式。脚本和服务实际上只有主函数，调用主函数时，必须快速执行并完成所需操作，或者启动一个无限循环以维持某些活动（例如从网络读取数据），直到用户停止操作。我们已经见过这样的循环示例：

```
while(!IsStopped())
{
    useful code
    ...
    Sleep(...);
}
```

在这种循环中，务必记得以一定周期使用 **Sleep**，以便与其他程序共享 CPU 资源。周期值的选择取决于所实施活动的预期强度。

这种模式可以称为循环式或同步式，甚至可视为实时，因为你可以选择休眠周期来提供恒定的数据处理频率，例如：

```

int rhythm = 100; // 100 ms, 10 times per sec
while(!IsStopped())
{
    const int start = (int)GetTickCount();
    useful code
    ...
    Sleep(rhythm - ((int)GetTickCount() - start));
}

```

当然，“有用代码”必须在分配的时间范围内完成执行。

相比之下，在事件模式下，我们无法预知何时执行下一段代码（处理程序）。例如，在快速波动的市场中，在新闻发布期间，分时报价可能会成批到达，而在夜间则可能几秒甚至更长时间都没有变动。极端情况下，周五晚间最后一笔分时报价后，某些金融工具的下一次价格变动可能要到周一早盘才会触发，这意味着 **OnTick** 事件可能两天都不会出现。换句话说，事件（以及事件处理程序的启动时刻）既无规律性，也无明确时间表。

但如有必要，可以将这两种模式结合使用。尤其是，计时器事件（**OnTimer**）提供了规律性，开发者可以在循环中定期为图表生成 **自定义事件**（例如闪烁警告标签）。

5.1.4 各类程序启动和停止的特征

在编程领域，“初始化”这一术语被用于多种不同上下文。在 MQL5 中，该术语同样具有多重性。在 **初始化** 章节中，我们曾用该术语表示设置变量的初始值。随后，我们讨论了指标和 EA 交易中的初始化事件 **OnInit**。尽管这两种初始化的含义相近（均使程序进入可工作状态），但实际上它们代表了 MQL 程序启动准备的两个不同准备阶段：系统初始化和应用初始化。

一个完整的 MQL 程序的生命周期可以概括为以下主要阶段：

1. 加载 - 将程序从文件读取到终端内存中：包括指令、预定义数据（字面量）、**资源**和**库**。正是在这一阶段，**#property**指令发挥作用。
2. 为全局变量分配内存并设置其初始值 - 这是由运行时执行的系统初始化。回顾 **初始化** 章节内容，当我们通过调试器逐步分析程序启动过程时，我们看到 `@global_initializations` 条目出现在堆栈中。这是针对该项目的代码块，由编译器隐式创建。如果程序使用类/结构体的全局对象，则其 **构造函数** 将在该阶段进行调用。
3. 调用 **OnInit** 事件处理程序（如果存在）：它将执行更高级别的应用层面初始化操作，因此每个程序会根据需要独立执行此操作。例如，可能需要为对象数组动态分配内存，出于某种原因，需要使用参数化构造函数，而不是默认构造函数。如我们所知，数组的自动内存分配仅使用默认构造函数，因此无法在前面的步骤（2）中完成初始化。还可能包括打开文件、调用内置 API 函数以启用必要的图表模式等操作。
4. 保持循环，直至用户关闭程序/终端，或执行需要重新初始化的任何其他操作（详见后文）：
 - ⌚ 根据发生的事件调用相应的处理程序。
5. 当检测到用户或程序尝试关闭程序时（仅 EA 交易和脚本中可使用对应函数 **ExpertRemove**），调用 **OnDeinit** 事件处理程序（如果存在）。
6. 最终清理：释放程序员未在 **OnDeinit** 中处理的已分配内存和其他资源。如果程序使用了 OOP，全局对象和静态对象的析构函数将在该阶段进行调用。
7. 下载程序。

脚本和服务默认不具备 *OnInit*和 *OnDeinit* 处理程序，因此它们不存在步骤 3 和步骤 5，且步骤 4 简化为单次调用 *OnStart*。

系统初始化 (步骤 2)与加载阶段不可分割，即系统初始化总是紧随加载阶段之后执行。最终清理始终在卸载前执行。然而，指标和 EA 交易在不同场景下的加载和卸载机制存在差异。因此，*OnInit*和 *OnDeinit* 调用 (步骤 3 和 5)是能够为 EA 交易和指标提供一致的应用初始化和反初始化的参考点。

在以下场景下执行指标和 EA 交易加载：

场景		
用户在图表上启动程序	+	+
启动终端 (如果程序在终端上次关闭前已在图表上运行)	+	+
加载模板 (如果模板包含已附加到图表的程序)	+	+
配置文件变更 (如果程序附加到其中一个配置文件图表)	+	+
成功重新编译后 (如果程序之前已附加到图表)	+	+
更改活动账户	+	+
更改指标所附加的图表的交易品种或周期	+	-
更改指标的输入参数	+	-
连接账户 (授权)，无论账户号码是否发生变化	-	+

可以更简洁的形式制定以下规则：EA 交易不会经历完整的生命周期，即当图表的交易品种/时间范围发生变更时，以及当输入参数变更时，它们不会重新加载。

因此，在卸载程序时可以观察到类似不对称性。卸载指标和 EA 交易的原因如下：

场景		
从图表中移除程序	+	+
关闭终端 (当程序附加到图表时)	+	+
在正在运行程序的图表上加载模板	+	+
关闭正在运行程序的图表	+	+
更改配置文件 (如果程序已附加到配置文件的其中一个图表)	+	+
更改终端所连接到的账户	+	+
更改指标所附加的图表的交易品种和/或周期	+	-
更改指标的输入参数	+	-
将另一个或同一个 EA 附加到当前 EA 已在运行的图表上	-	+
调用 <code>ExpertRemove</code> 函数	-	+

反初始化的原因可以在程序中通过函数 `UninitializeReason` 或标志 `_UninitReason` 来查找 (请参阅 [检查停止 MQL 程序的状态和原因](#) 章节)。

请注意, 更改图表的交易品种或时间范围时, 以及更改输入参数时, EA 交易将保留在内存中, 即不会执行步骤 6 至 7 (最终清理和卸载) 以及步骤 1 至 2 (加载和主内存分配), 因此全局变量和静态变量的值不会进行重置。在这种情况下, 会分别针对旧的和新的交易品种/时间范围 (或者旧的和新的设置) 依次调用 `OnDeinit` 和 `OnInit` 处理程序。

由于 EA 交易中的全局变量不会被清除, 因此反初始化代码 `_UninitReason` 在 `OnInit` 处理程序中保持不变, 可供分析使用。只有在下一个事件发生时, 且在调用 `OnDeinit` 之前, 新代码才会被写入变量中。

在 `OnInit` 函数结束之前接收到的针对 EA 交易的所有事件都会被跳过。

当 MQL 程序首次启动时, 设置对话框会在步骤 1 和步骤 2 之间显示。当更改输入参数时, 设置对话框会根据程序类型以不同方式插入到主循环中: 对于指标, 对话框仍会在步骤 2 之前显示, 对于 EA 交易, 则会在步骤 3 之前显示。

本书附带名为 `LifeCycle.mq5` 的指标和 EA 交易模板。它会在 `OnInit/OnDeinit` 处理程序中记录全局初始化/最终清理步骤。将程序放置在图表上, 观察以下各种用户操作会触发哪些事件: 加载/卸载、更改参数、切换交易品种/时间范围。

脚本仅在添加到图表时加载一次。如果脚本正在循环运行, 重新编译不会导致其重启。

服务通过终端界面上上下文菜单命令进行加载和卸载。若某个已在运行的服务被重新编译, 该服务将重新启动。请记住, 处于活动状态的服务实例将在终端启动时自动加载, 并在终端关闭时自动卸载。

在接下来两个章节，我们将从事件处理程序的层面探讨启动不同 MQL 程序的特性。

5.1.5 指标和 EA 交易的参考事件：OnInit 和 OnDeinit

在交互式 MQL 程序（指标和 EA 交易）中，环境会生成两个事件，用于准备启动（*OnInit*）和停止（*OnDeinit*）。脚本和服务中不存在这些事件，因为它们不接受异步事件：当控制权交给其唯一的事件处理程序 *OnStart* 后，直到工作结束前，脚本/服务线程的执行上下文将始终处于该 MQL 程序代码中。相比之下，对于指标和 EA 交易，正常工作流程假定，环境会反复调用其特定的事件处理函数（我们将在指标和 EA 交易相关章节中讨论这些函数），并且每次完成必要操作后，程序会将控制权返回给终端，进入空闲状态等待新事件。

int OnInit()

*OnInit*函数是同名事件的处理程序，该事件在加载 EA 交易或指标后生成。该函数仅在需要时定义。

该函数必须返回 ENUM_INIT_RETCODE 枚举值之一。

标识符	说明
INIT_SUCCEEDED	初始化成功，可继续执行程序；对应值 0
INIT_FAILED	初始化失败，由于致命错误（如无法创建文件或辅助指标）无法继续执行；值为 1
INIT_PARAMETERS_INCORRECT	输入参数组错误，无法执行程序
INIT_AGENT_NOT_SUITABLE	在 测试程序中工作的特定代码：由于某些原因，当前代理不适合测试（例如 RAM 不足、不支持 OpenCL 等）

如果 *OnInit*返回任何非零返回码，则表示初始化失败，随后会生成 *Deinit* 事件，其反初始化原因代码为 REASON_INITFAILED（见下文）。

*OnInit*函数可以声明为返回类型 *void*：在这种情况下，初始化始终被视为成功。

在 *OnInit*处理程序中，重要的是确保所有必要的环境信息已就绪，如果不可用，则将延迟下一个分时报价或计时器到达事件的准备操作。关键在于，当终端启动时，*OnInit*事件通常在与服务器建立连接之前触发，因此许多金融工具和交易账户的特性仍然未知。特别是，特定交易品种的点值可能返回为零。

void OnDeinit(const int reason)

当 EA 交易或指标被反初始化时，将调用 *OnDeinit*函数（如果已定义）。该函数为可选函数。

*reason*参数包含反初始化原因代码。可能的值如下表所示。

常量	值	说明
REASON_PROGRAM	0	EA 交易通过调用 <i>ExpertRemove</i> 函数停止运行
REASON_REMOVE	1	程序从图表中移除
REASON_RECOMPILE	2	程序被重新编译
REASON_CHARTCHANGE	3	图表交易品种或周期发生变更
REASON_CHARTCLOSE	4	图表关闭
REASON_PARAMETERS	5	输入参数发生变更
REASON_ACCOUNT	6	激活了另一个账户，或由于账户设置变更导致重新连接到交易服务器
REASON_TEMPLATE	7	应用了不同的图表模板
REASON_INITFAILED	8	<i>OnInit</i> 处理程序返回非零值
REASON_CLOSE	9	终端关闭

在 MQL 程序中，如果设置了停止标志 *_StopFlag*，可以在程序的任何位置使用 *UninitializeReason* 函数获取相同的代码。

*AllInOne.mqh*文件有 *Finalizer* 类，允许通过 *UninitializeReason* 调用在析构函数中“挂钩”反初始化代码。我们必须在 *OnDeinit*处理程序中获取相同的值。

```
class Finalizer
{
    static const Finalizer f;
public:
    ~Finalizer()
    {
        PRTF(EnumToString((ENUM_DEINIT_REASON)UninitializeReason()));
    }
};

static const Finalizer Finalizer::f;
```

为了便于使用 *EnumToString*将代码转换为字符串表示形式（原因名称），*Uninit.mqh* 文件中定义了包含上述表格常量的 *ENUM_DEINIT_REASON* 枚举。日志将显示类似以下的条目：

```
OnDeinit DEINIT_REASON_REMOVE
EnumToString((ENUM_DEINIT_REASON)UninitializeReason())=DEINIT_REASON_REMOVE / ok
```

当更改指标所在图表的交易品种或时间范围时，指标会被卸载并重新加载。在这种情况下，旧副本中触发 *OnDeinit*事件顺序与新副本中触发 *OnInit* 事件的顺序没有明确定义。这是由于终端处理异步事件的特性所致。换言之，新副本可能会在旧副本完全卸载之前就被加载并初始化，这可能不太符合逻辑。如果指标在 *OnInit*中执行了某些图表调整操作（例如创建 *图形对象*），而没有采取特殊措施，则被卸载的副本可能会立即“清理”图表（删除该对象，误认为是自己创建的）。对于图形对象这一特定情况，有一个特殊的解决方案：可以为对象命名时包含交易品种和时间范围前缀（以及输入变量值的校验和），但该方案不适用于

一般情况。要实现这一问题的通用解决方案，应实现某种同步机制，例如基于 [全局变量](#) 或 [资源](#)。

在测试程序中测试指标时，MetaTrader 5 开发者决定不生成 *OnDeinit* 事件。他们的初衷是，指标会创建一些图形对象，这些对象通常会 *OnDeinit* 处理程序中被移除，但用户可能希望在测试结束后仍能查看这些对象。实际上，如果有需要，MQL 程序开发者可通过主动检查 *MQLInfoInteger(MQL_TESTER)* 模式来实现类似行为和保留对象。这一设计存在矛盾之处，因为在 EA 交易测试后，会调用 *OnDeinit* 处理程序，而 EA 交易同样可以在 *OnDeinit* 中删除对象。目前，仅针对指标，测试程序中无法保证 *OnDeinit* 处理程序的正常执行。此外，其他终止操作同样不会执行，例如不会调用全局对象的析构函数。

因此，如需在测试运行结束后执行统计计算、文件保存或其他原本计划在指标的 *OnDeinit* 中完成的操作，则必须将指标算法迁移至 EA 交易。

5.1.6 脚本和服务的主要函数：OnStart

实用程序（脚本和服务）在终端中通过调用其唯一的事件处理函数 *OnStart* 来执行。

void OnStart()

该函数没有参数，也不返回任何值。它仅作为从终端侧进入应用程序的入口点。

脚本通常用于在图表上执行一次性操作（稍后我们将探讨图表 API 提供的所有功能）。例如，脚本可用于执行以下操作：设置订单网格，反过来关闭所有盈利的未平仓头寸，自动应用图形对象标记或临时隐藏所有对象。

在脚本中，可以使用封装在无限循环中的持续性操作，但如前所述，必须始终检查停止标志 (*_StopFlag*) 并定期释放处理器 (*Sleep*)。这里需要记住的是，在关闭并重新启动终端后，脚本也必须重新运行。

因此，对于这类持续性活动，如果其与时间表无直接关联，则最好使用服务。服务的标准实现方式正是通过“无限”循环来完成。

在本书之前的章节中，几乎所有示例都是以脚本的形式实现。服务的其中一个示例是 *GlobalsWithCondition.mq5* 程序，详见 [使用全局变量同步程序](#) 章节。在下一节关于使用 *ExpertRemove* 函数停止 EA 交易和脚本的内容中，我们还会看到另一个示例。

5.1.7 以编程方式移除 EA 交易和脚本：ExpertRemove

如有必要，开发者可以组织对以下两种类型 MQL 程序的停止和卸载：EA 交易和脚本。这可以通过 *ExpertRemove* 函数来实现。

void ExpertRemove()

该函数没有参数，也不返回值。它向 MQL 程序执行环境发送删除当前程序的请求。实际上，这会导致设置 *_StopFlag* 标志，并停止接收（和处理）所有后续事件。之后，程序会有 3 秒钟的时间来正常完成其工作：释放资源、中断算法循环等。如果程序未能完成这些操作，系统将强制卸载该程序，并可能导致中间数据丢失。

该函数在指标和服务中无效（程序会继续运行）。

每次调用该函数时，日志中都将包含“ExpertRemove() 函数已被调用”条目。

该函数主要用于无法通过任何其他方式中断的 EA 交易。对于脚本而言，通常使用 *break* 语句来中断循环（如果存在）会更容易。但如果循环是嵌套的，或算法存在多个函数相互调用的情

况，则在继续计算的条件下，在不同级别考虑停止标志会更容易，并且在出现错误情况时，可以使用 *ExpertRemove* 设置此标志。如果不使用此内置标志，无论如何，都需要引入一个具有相同用途的全局变量。

脚本 *ScriptRemove.mq5* 提供了 *ExpertRemove* 用法示例。

可通过 *ProblemSource* 类进行模拟算法运行中可能导致需要卸载脚本的潜在问题。 *ExpertRemove* 会在其构造函数中随机进行调用。

```
class ProblemSource
{
public:
    ProblemSource()
    {
        // simulating a problem during object creation, for example,
        // with the capture of some resources, such as a file, etc.
        if(rand() > 20000)
        {
            ExpertRemove(); // will set _StopFlag to true
        }
    }
};
```

接下来，会在全局级别和辅助函数内部会创建该类的对象。

```
ProblemSource global; // object may throw an error

void SubFunction()
{
    ProblemSource local; //object may throw an error
    // simulate some work (we need to check the integrity of the object!)
    Sleep(1000);
}
```

现在，我们在 *OnStart* 操作中使用 *SubFunction*，并在带有 *IsStopped* 条件的循环内部调用它。

```
void OnStart()
{
    int count = 0;
    // loop until stopped by the user or the program itself
    while(!IsStopped())
    {
        SubFunction();
        Print(++count);
    }
}
```

下面是一个日志示例（由于随机性，每次运行的结果都会不同）：

```

1
2
3
ExpertRemove() function called
4

```

请注意，如果在创建全局对象时发生错误，循环将完全不会执行。

由于 EA 交易可以在 [测试程序](#) 中运行，因此 `ExpertRemove` 函数同样可以在测试程序中使用。其效果取决于函数调用的位置。如果在 `OnInit` 处理程序中执行此操作，该函数将取消测试，即终止测试程序针对当前 EA 交易参数集的单次执行。这种终止将被视为初始化错误。当在算法的其他任何位置调用 `ExpertRemove` 时，EA 交易的测试将提前中断，但会按照常规方式进行处理，即会调用 `OnDeinit` 和 `OnTester`。在这种情况下，所积累的交易统计数据 and 优化标准值将保留，但需注意，模拟服务器时间 `TimeCurrent` 并未达到测试程序设置中的结束日期。

5.2 脚本和服务

在本章中，我们将系统梳理并完整呈现关于脚本和服务的所有技术信息，这些内容我们在本书前几部分已开始有所了解。

脚本和服务在程序代码的组织 and 执行方面遵循相同的原则。我们知道，它们的主函数 `OnStart` 也是唯一的函数。脚本和服务无法处理 [其他事件](#)。

然而，仍存在一些显著差异。脚本在图表上下文中执行，并能通过内置变量（如 `_Symbol`、`_Period`、`_Point` 等）直接访问图表特性。我们将在 [图表特性](#) 章节中详细研究这些内容。而服务是独立运行的，不与任何窗口绑定，尽管它们能够使用特殊函数来分析所有图表（同样的 [图表函数](#) 也可用于其他程序类型：脚本、指标和 EA 交易）。

另一方面，所创建的服务实例会被终端在后续会话中自动恢复。换言之，服务一旦启动就会一直运行，除非用户将其停止。相比之下，脚本会在终端关闭或图表关闭时被删除。

请注意，服务与其他所有 MQL 程序类型一样，都是在终端中执行的，因此关闭终端也会使服务停止。已激活的服务会在你下次启动终端时恢复运行。只有在终端持续运行的情况下（例如在 VPS 上运行的终端），才能确保 MQL 程序的不间断运行。

在脚本和服务中，可以使用 `#property` 指令设置 [程序通用特性](#)。除了这些特性外，脚本和服务程序还有一些特有的特性；我们将在后续两个章节中分别详述。

当前在图表上运行的脚本会与正在运行的 EA 交易显示在同一列表中，该列表可通过图表上下文菜单中的 `Expert List` 命令打开的 `Experts` 对话框查看。可以从该对话框中强制将它们从图表中移除。

服务只能从 `Navigator` 窗口进行管理。

5.2.1 脚本

脚本是一种仅包含处理程序 `OnStart` 的 MQL 程序，前提是不存在 `#property service` 指令（否则就会变成服务，详见下一章节）。

默认情况下，脚本被加载到图表时会立即开始执行。开发者可通过在文件开头添加 `#property script_show_confirm` 指令，要求用户确认启动。在这种情况下，终端将显示一条消息，其中包含问题“你是否确定要在图表 交易品种，时间范围’上运行 程序’？”，以及 `Yes` 和 `No` 按钮。

脚本和其他程序一样，可以包含 输入变量。不过对于脚本来说，即使脚本定义了 `input`，默认也不会显示参数输入对话框。为确保在运行脚本之前打开特性对话框，应使用 `#property script_show_inputs` 指令。该指令优先级高于 `script_show_confirm`，即对话框弹出时将禁用确认请求（因为对话框本身已具备类似的确认功能）。即使没有输入变量，该指令也会调用对话框。用于向用户展示产品描述和版本信息（这些信息会显示在 `Common`选项卡中）。

下表展示了 `#property`指令的组合选项及其对程序的影响。

影响	指令	<code>script_show_confirm</code>	<code>script_show_inputs</code>
立即启动		否	否
确认请求		是	否
打开特性对话框		不相关	是

`ScriptNoComment.mq5`文件中提供了一个含指令的脚本简单示例。该脚本的用途如下。有时，MQL 程序会在图表左上角留下不必要的注释。由于注释会与图表一同存储在 `chr` 文件中，因此即使重启终端后，这些注释仍会恢复。此脚本允许你清除注释或将其设置为任意值。如果你使用 `Navigator`上下文菜单命令为脚本 `Assign hotkey`，则只需点击一下，即可清除当前图表的注释。

最初，`script_show_confirm`和`script_show_inputs` 指令通过转换为内联注释而处于禁用状态。可以通过逐一取消或一次性全部取消这些指令的注释，来试验不同的指令组合。

```
//#property script_show_confirm
//#property script_show_inputs

input string Text = "";

void OnStart()
{
    Comment(""); // clean up the comment
}
```

5.2.2 服务

服务是一种具有单个 `OnStart`处理程序和 `#property service` 指令的 MQL 程序。

回顾一下，在服务成功编译之后，需要使用 `Navigator`窗口上下文菜单中的 `Add Service` 命令来创建和配置其实例（一个或多个）。

作为一个服务示例，我们来解决一个 MQL 程序开发者经常遇到的小应用问题。大多数开发者会将其程序与用户账户号码绑定。这不一定是针对付费产品，也可能涉及在亲友间分发，用于收集统计数据或优化设置。与此同时，除了有效的真实账户之外，用户还可以注册模拟账户。此类账户的有效期通常有限，因此每隔几周就为它们更新绑定信息相当不便。为此，你需要编辑源代码、重新编译并再次发布程序。

不过我们可以改为开发一个服务，将从给定终端成功连接的账户号码注册到全局变量（或文件）中。

绑定技术基于账户号码的成对加密 (或者采用哈希方式): 即旧的登录账户和新的登录账户。前一个账户必须是主账户 (即获得“已签发”条件性授权), 这样这对账户的共同签名才能将产品的使用权扩展到新账户。密钥是只有程序内部才知道的机密信息 (假定所有程序都是以封闭编译后形式提供)。运算结果将是一个 **Base64** 格式的字符串。该实现使用 MQL5 API 函数, 其中一些函数我们还未讲解, 特别是通过 **AccountInfoInteger** 和 **CryptEncode** 加密函数获取账号。使用 **TerminalInfoInteger** 函数检查与服务器的连接 (请参阅 [检查网络连接](#))。

该服务无需了解哪些账户是主账户, 哪些是附属账户。它只需以特定方式为任意连续登录的账户对进行“签名”验证。但特定应用程序应补充其“许可证”检查过程: 除了将当前账户与主账户进行比较之外, 还应重复执行服务算法, 即创建一个 [主账户; 当前账户] 的账户对, 计算该账户对的加密签名, 并检查该签名是否在全局变量之中。

只有在交易模式 (而非投资者模式) 下连接到同一账户时, 才有可能通过将许可证转移到另一台计算机上来盗用此类许可证。当然, 不诚信的用户可能会为他人创建模拟账户。因此, 需要增强保护措施。在当前实现中, 全局变量仅设置为临时变量, 即在终端会话结束时会被删除, 但仍存在被复制的潜在风险。

作为补充措施, 例如可以在签名中对其创建时间进行加密, 并将权限有效期设置为每日 (或其他频率)。另一种选择是在服务启动时生成随机数, 并将其与账户号码一起添加到签名信息中。该随机数仅服务内部可知, 但可以通过 **EventChartCustom** 函数将其传输给图表上的相关 MQL 程序。因此, 签名将在终端的该实例持续有效, 直至会话结束。每个会话都会生成并发送一个新的随机数, 因此它不适用于其他终端。最后, 最简单且最便捷的选项可能是在签名中加入系统启动时间: $(TimeLocal() - GetTickCount()) / 1000$ 或其派生值。

在各类 MQL 程序中, 仅部分程序能在账户切换期间持续运行, 并支持实施这种保护方案。由于需要以统一方式保护所有类型的 MQL 程序, 包括指标和 EA 交易 (在账户变更时重新加载), 因此将此任务委托给服务是合理的。然后, 服务将控制登录并生成授权签名, 并且该服务从终端加载开始持续运行, 直至终端关闭。

服务的源代码位于文件 **MQL5/Services/MQL5Book/p5/ServiceAccount.mq5** 中。输入参数用于指定主账户以及存储签名的全局变量前缀。在实际程序中, 主账户列表应在源代码中进行硬编码, 而且最好使用 **Common** 文件夹中的文件来替代全局变量, 这样也能覆盖测试程序场景。

```
#property service
```

```
input long MasterAccount = 123456789;
input string Prefix = "!A_";
```

服务的主函数按以下方式执行工作: 在一个每秒暂停一次的无限循环中, 跟踪账户变更并保存最新账号, 为账户对创建签名, 然后将其写入全局变量。签名由 **Cipher** 函数生成。

```

void OnStart()
{
    static long account = 0; // previous login

    for(; !IsStopped(); )
    {
        // require connection, successful login and full access (not investor)
        const bool c = TerminalInfoInteger(TERMINAL_CONNECTED)
            && AccountInfoInteger(ACCOUNT_TRADE_ALLOWED);
        const long a = c ? AccountInfoInteger(ACCOUNT_LOGIN) : 0;

        if(account != a) // account changed
        {
            if(a != 0) // current account
            {
                if(account != 0) // previous account
                {
                    // transfer authorization from one to another
                    const string signature = Cipher(account, a);
                    PrintFormat("Account %I64d registered by %I64d: %s",
                        a, account, signature);
                    // saving a record about the connection of accounts
                    if(StringLen(signature) > 0)
                    {
                        GlobalVariableTemp(Prefix + signature);
                        GlobalVariableSet(Prefix + signature, account);
                    }
                }
                else // the first account is authorized, now waiting for the second one
                {
                    PrintFormat("New account %I64d detected", a);
                }
                // remember the last active account
                account = a;
            }
        }
        Sleep(1000);
    }
}

```

*Cipher*函数使用特殊联合体 *ByteOverlay2* 将一对账号 (*long* 型)表示为字节数组,并将其传递给 *CryptEncode* 进行加密 (此处选择 *CRYPT_DES* 加密方法,但如果不需要从“签名”中恢复信息,也可替换为 *CRYPT_AES128*、*CRYPT_AES256* 或仅使用 *CRYPT_HASH_SHA256* 哈希算法 (以密钥作为“盐值”))。


```

template<typename T>
union ByteOverlay2
{
    T values[2];
    uchar bytes[sizeof(T) * 2];
    ByteOverlay2(const T v1, const T v2) { values[0] = v1; values[1] = v2; }
};

string Cipher(const long data1, const long data2)
{
    // TODO: replace the secret with your passphrase
    // TODO: CRYPT_AES128/CRYPT_AES256 methods require 16/32 byte arrays
    const static uchar secret[] = {'S', 'E', 'C', 'R', 'E', 'T', '0'};
    ByteOverlay2<long> bo(data1, data2);
    uchar result[];
    if(CryptEncode(CRYPT_DES, bo.bytes, secret, result) > 0)
    {
        uchar dummy[], text[];
        if(CryptEncode(CRYPT_BASE64, result, dummy, text) > 0)
        {
            return CharArrayToString(text);
        }
    }
    return NULL;
}

```

然后，终端中的任何程序都可以检查全局变量中是否存在当前账户的“许可证”。这可以通过 *CheckAccounts* 和 *IsCurrentAccountAuthorizedByMaster* 函数来实现。它们在服务中仅作演示用途。

CheckAccounts 函数会对硬编码的所有主账户进行检查，以找到与当前账户匹配的主账户。

```

bool CheckAccounts()
{
    const long accounts[] = {MasterAccount}; // TODO: to fill array with constants
    for(int i = 0; i < ArraySize(accounts); ++i)
    {
        if(IsCurrentAccountAuthorizedByMaster(accounts[i])) return true;
    }
    return false;
}

```

IsCurrentAccountAuthorizedByMaster 接收一个主账号作为参数，为其与当前账户的配对重新创建“签名”，并分析匹配度。


```
bool IsCurrentAccountAuthorizedByMaster(const long data)
{
    const long a = AccountInfoInteger(ACCOUNT_LOGIN);
    if(a == data) return true; // direct match
    const string s = Cipher(data, a); // recalculating "signature"
    if(a != 0 && GlobalVariableGet(Prefix + s) == a)
    {
        Print("Sub-License is active: ", s);
        return true;
    }
    return false;
}
```

假设允许程序在账号 123456789 上运行，且该账号当前处于活动状态。启动时，服务将立即响应，并显示日志条目：

```
New account 123456789 detected
```

如果随后我们更改账号（例如更改为 5555555），将获得以下签名：

```
Account 5555555 registered by 123456789: jdVKxUswBiNlZzDAnV3yxw==
```

如果我们停止并重新启动服务，将看到正在验证账户 5555555（在 *OnStart* 函数开始时调用内置函数 *CheckAccounts* 进行演示）。

```
Sub-License is active: jdVKxUswBiNlZzDAnV3yxw==
```

```
Account 123456789 registered by 5555555: ZWcwwJ1d8seN1UrFSzAGIw==
```

新账户的许可证验证通过。如果切换回原账户，会生成一个从当前账户到前一账户的“通行证”这是因为服务无法区分主账户和临时账户，而实际程序中通常不需要此类“签名”。

如需间接授权新账户，需先重新登录主账户，再切换至新账户：这将创建一个新的全局变量，其中包含加密后的账户对 [主账户; 新账户]。

该版本的服务不会检查主账户是否为真实账户以及从属账户是否为模拟账户。以上各项限制都可以添加。

5.2.3 脚本和服务的局限性

在脚本和服务中，禁止使用指标操作组包含的所有函数。将在相应 [章节](#) 中介绍这些函数：

🕒 *SetIndexBuffer*

🕒 *IndicatorSetDouble*

🕒 *IndicatorSetInteger*

🕒 *IndicatorSetString*

🕒 *PlotIndexSetDouble*

🕒 *PlotIndexSetInteger*

🕒 *PlotIndexSetString*

🕒 *PlotIndexGetInteger*

此外，在脚本和服务中，使用 *OnTimer* 处理程序（与其他任何处理程序一样）及 [计时器](#) 函数没有意义。

⌚ *EventSetMillisecondTimer*

⌚ *EventSetTimer*

⌚ *EventKillTimer*

由于脚本和服务不受测试程序支持，因此它们无法使用 *测试程序函数*；它们会引发错误 ERR_FUNCTION_NOT_ALLOWED (4014)。

5.3 时间序列

时间序列是一种数据数组，其中元素的索引对应于有序的时间样本。由于终端的应用特性，交易者所需的几乎所有信息都是以时间序列的形式提供的。具体而言，这些信息包括报价数组、分时报价、技术指标读数等等。绝大多数 MQL 程序也处理这些数据，因此 MQL5 API 中专门为它们分配了一组函数，本节将重点探讨这些函数。

MQL5 中访问数组的方式使开发者可以设置以下两种索引方向之一：

⌚ 正向（向前）-元素编号从数组的开始到结束（从旧到新）

⌚ 反向（时间序列）-编号从数组的结束到开始（从新到旧）

我们已经在 *时间序列中的数组索引方向* 章节中讨论过此问题。

可使用 *ArraySetAsSeries* 函数来更改索引模式，并且这一操作不会影响数组在内存中的物理结构。只有通过编号访问元素的方式会发生变化：在正向索引索引模式下，我们通过 *array[i]* 获取第 *i* 个元素；而在时间序列模式下，等效的公式为 *array[N - i - 1]*，其中 *N* 是数组的大小（之所以称为“等效”，是因为如果为数组设置了时间序列索引模式，终端会自动执行这种重新计算，应用开发者无需在各处都进行这样的计算）。下表对此进行了说明（以一个包含 10 个元素的字符数组为例）。

数组元素	A	B	C	D	E	F	G	H	I	J
常规索引	0	1	2	3	4	5	6	7	8	9
时间序列中的索引	9	8	7	6	5	4	3	2	1	0

请记住，数组索引始终从零开始。

当涉及报价数组和其他持续更新的数据时，新元素会按物理顺序追加到数组末尾。但从交易角度来看，分析历史数据时应考虑最新数据并将其作为起点。正因如此，将当前（最后一个）柱线始终置于索引 0 的位置，而之前的柱线则从该索引开始按过去顺序依次计数，这种方式更为便捷。由此，我们便有了时间序列索引。

默认情况下，数组按从左到右的顺序建立索引。如果我们设想这样的数组显示在标准的 MetaTrader 5 图表上，从视觉上看，索引为 0 的元素会位于最左侧，最后一个元素位于最右侧。在反向索引的时间序列中，第 0 个元素对应最右侧位置，最后一个元素对应最左侧位置。由于时间序列按照时间顺序存储金融工具的价格历史数据，其中最新数据始终位于旧数据的右侧。

时间序列数组中索引为 0 的元素包含有关交易品种最新报价的信息。第 0 根柱线通常是不完整的，因为它仍在形成中。

报价时间序列的另一个特征是其周期，即相邻读数之间的时间间隔。该周期也称为“时间范围”，可以更精确地重新表述。时间范围是形成一根报价柱线的周期，其起始和结束在绝对时间上以相同的步长对齐。例如，在“1 小时”(H1) 时间范围下，柱线严格从每天整点的 0 分钟开始。每个这样的周期的起点属于当前柱线，终点则属于下一根柱线。

在 [交易品种和时间范围](#) 一章中，提供了标准时间范围的完整列表。

通常，在时间序列概念框架内，技术 [指标](#) 的缓冲也会起作用，但我们将在后面详细讨论它们的特性。

在任何 MQL 程序中，如有需要，均可请求获取任意交易品种和时间范围的时间序列数值，以及为任意交易品种和时间范围计算的指标数值。这些数据可通过 [Copy函数](#) 获取，其中包括分别多个读取不同类型价格数组的函数（例如 [Open、High、Low、Close](#)），以及读取包含每根柱线全部特征的 [MqlRates](#) 结构体数组的函数。

柱线和分时报价

除了带报价的柱线外，MetaTrader 5 还为用户和 MQL 程序提供了分析“分时报价”的能力，它们是构成柱线的基本价格变动单位。每个分时报价包含精确到毫秒的时间、多种价格类型 ([Bid、Ask、Last](#))、描述变动性质的标志以及交易的成交量。我们将在稍后的 [使用真实分时报价数组](#) 一章中学习相应的 [MqlTick](#) 结构。

根据交易金融工具的类型，柱线可以基于 [Bid](#) 或 [Last](#) 价格构建。特别是对于交易所交易金融工具，可使用 [Last](#) 价格，这些金融工具同时会实时推送 [市场深度价格](#)。对于外汇或 CFD 等非交易所金融工具，则使用 [Bid](#) 价格。

没有价格变动的周期不会生成柱线。这就是 MetaTrader 5 中价格的呈现方式。例如，如果时间范围为 1 天 (D1)，则周末通常不会生成柱线，周五的柱线之后会直接显示周一的柱线。

只有当相应时间间隔内至少出现一个分时报价时，才会生成一根报价柱线。同时，柱线的开盘时间始终严格与时间周期边界对齐，即使第一个分时报价到达的时间较晚（通常就是这种情况）。例如，如果午夜后 4 分钟内没有分时报价，而在 00:05:15（即第 5 分钟的第 15 秒）发生了价格变动，则当天的第一根 M1 柱线将在 00:05 形成。因此，分时报价是否属于某根柱线，取决于以下时间戳关系： $T_{\text{open}} \leq T_{\text{tick}} < T_{\text{open}} + P$ ，其中 T_{open} 是柱线开盘时间， T_{tick} 是分时报价时间， $T_{\text{open}} + P$ 是周期 P 结束后下一个潜在柱线的开盘时间（称为“潜在”柱线，是因为其是否存在取决于是否有其他分时报价）。

5.3.1 交易品种和时间范围

包含报价的时间序列由两个参数进行标识：交易品种名称（金融工具）和时间范围（周期）。

用户可以在 [Market Watch](#) 窗口中查看交易品种列表，并根据经纪商提供的总列表（在 [Symbols](#) 对话框中）对其进行编辑。对于 MQL 程序，有一组函数可用于执行相同的操作：在所有交易品种中搜索、查询其特性，以及在 [Market Watch](#) 窗口中添加或移除交易品种。这些功能将作为独立章节的主题进行详细讲解。

然而，要请求时间序列数据，只需知道交易品种名称即可，其名称是一个包含现有金融工具代码的字符串。例如，用户可以在输入变量中设置该名称。此外，当前图表的交易品种可通过内置变量 [_Symbol](#)（或 [Symbol](#) 函数）获取，但为方便起见，所有时间序列函数均支持以下约定：NULL 值也对应当前图表的交易品种。

现在让我们转向时间范围。系统中定义了 21 个标准时间范围：每个时间范围由特殊枚举 [ENUM_TIMEFRAMES](#) 中的一个元素指定。

标识符	说明	说明
PERIOD_CURRENT	0	当前周期
PERIOD_M1	1 (0x1)	1 分钟
PERIOD_M2	2 (0x2)	2 分钟
PERIOD_M3	3 (0x3)	3 分钟
PERIOD_M4	4 (0x4)	4 分钟
PERIOD_M5	5 (0x5)	5 分钟
PERIOD_M6	6 (0x6)	6 分钟
PERIOD_M10	10 (0xA)	10 分钟
PERIOD_M12	12 (0xC)	12 分钟
PERIOD_M15	15 (0xF)	15 分钟
PERIOD_M20	20 (0x14)	20 分钟
PERIOD_M30	30 (0x1E)	30 分钟
PERIOD_H1	16385 (0x4001)	1 小时
PERIOD_H2	16386 (0x4002)	2 小时
PERIOD_H3	16387 (0x4003)	3 小时
PERIOD_H4	16388 (0x4004)	4 小时
PERIOD_H6	16390 (0x4006)	6 小时
PERIOD_H8	16392 (0x4008)	8 小时
PERIOD_H12	16396 (0x400C)	12 小时
PERIOD_D1	16408 (0x4018)	1 天
PERIOD_W1	32769 (0x8001)	1 周
PERIOD_MN1	49153 (0xC001)	1 个月

正如我们在 [预定义变量](#) 章节中所见，程序可以通过内置变量 `_Period` (或 `Period` 函数) 获取当前图表的时间周期。从值列可以明显看出，向接收时间范围的内置函数传递零，即表示引用当前图表的时间周期。

对于分钟时间范围，其值与其中包含的分钟数一致 (例如 30 表示 M30)。对于小时时间范围，会设置 0x4000 位，同时低字节包含小时数 (例如 H3 对应 0x4003)。日周期 D1 按 24 小时编码，即 0x4018 (其中 0x18 等于 24)。最后，周线和月线时间范围分别有自己的特征位 0x8000 和 0xC000 作为单位标识，且两种情况下的计数值 (在低字节中) 均为 1。

为便于枚举元素与字符串相互转换，本书附带了头文件 *Periods.mqh* (我们已在文件操作示例中使用过该文件，后续示例还将继续使用)。这个头文件的其中一个函数 *StringToPeriod* 在算法中使用了前文所述的枚举元素内部位表示特性。

```
#define PERIOD_PREFIX_LENGTH 7 // StringLen("PERIOD_")

// getting the abbreviated name of the period without the "PERIOD_" prefix
string PeriodToString(const ENUM_TIMEFRAMES tf = PERIOD_CURRENT)
{
    const static int prefix = StringLen("PERIOD_");
    return StringSubstr(EnumToString(tf == PERIOD_CURRENT ? _Period : tf),
        PERIOD_PREFIX_LENGTH);
}

// get the period value by full (PERIOD_H4) or short (H4) name
ENUM_TIMEFRAMES StringToPeriod(string name)
{
    if(StringLen(name) < 2) return 0;
    // converting full name "PERIOD_TN" to short "TN" if needed
    if(StringLen(name) > PERIOD_PREFIX_LENGTH)
    {
        name = StringSubstr(name, PERIOD_PREFIX_LENGTH);
    }
    // convert the digital ending "N" to a number, skip "T"
    const int count = (int)StringToInteger(StringSubstr(name, 1));
    // clear possible error WRONG_STRING_PARAMETER(5040)
    // for example, if the input string is "MN1", then N1 is not a number for StringTo
    ResetLastError();
    switch(name[0])
    {
        case 'M':
            if(!count) return PERIOD_MN1;
            return (ENUM_TIMEFRAMES)count;
        case 'H':
            return (ENUM_TIMEFRAMES)(0x4000 + count);
        case 'D':
            return PERIOD_D1;
        case 'W':
            return PERIOD_W1;
    }
    return 0;
}
```

请注意，*_Symbol*和*_Period* 变量仅在基于图表运行的 MQL 程序 (包括脚本、EA 交易和指标) 中包含实际数据。在服务中，这些变量为空，因此如果要访问时间序列，必须显式设置交易品种名称和时间周期，或通过外部途径获取它们。

时间范围的决定性特性是其持续时间 (柱线持续时间)。MQL5 允许你使用 *PeriodSeconds*函数获取构成特定时间范围的一个柱线所对应的秒数。

```
int PeriodSeconds(ENUM_TIMEFRAMES period = PERIOD_CURRENT)
```

period参数以 `ENUM_TIMEFRAMES` 枚举元素的形式指定时间周期。如果未指定该参数，则返回程序运行所在的当前图表时间周期的秒数。

我们将在 [等待数据和管理可见性](#) 章节的 `IndDeltaVolume.mq5` 指标中以及 [使用内置指标](#) 章节的指标 `UseM1MA.mq5` 中，介绍该函数的用法示例。

为了生成未包含在指定列表中的非标准持续时间的时间范围，MQL5 API 提供了 [自定义交易品种](#)，但如果不修改 EA 交易，这些品种则无法像在标准图表上那样进行交易操作。

此外，需要重点注意的是，在 MetaTrader 5 中，特定时间序列内或图表上的柱线的持续时间始终恒定。因此，要构建不按时间形成、而是根据其他参数累积（尤其是成交量，如等成交量图表）或按固定步长的单向价格变动（砖形图）形成的图表，可基于指标（例如使用 `DRAW_CANDLES` 或 `DRAW_BARS` 渲染类型）或通过 [自定义交易品种](#)。

5.3.2 时间序列组织与存储的技术要点

在探讨 MQL5 中用于处理时间序列的 API 函数的实际应用问题之前，我们需要先了解 MetaTrader 5 平台从服务器接收报价数据并进行存储的技术基础。

在价格数据被终端用于图表显示和传递给 MQL 程序之前，这些数据会从服务器下载并经过特殊预处理。无论数据请求是由用户在图表中导航时发起，还是通过 MQL5 语言以编程方式发起，服务器数据访问机制都是相同的。

从服务器传输的数据采用压缩格式传输：这些是经过高效打包的分钟柱线数据块，但并非通常的 M1 柱线。

从服务器接收的数据会自动解压缩，并以特殊的 HCC 中间格式保存。每个交易品种的数据会写入独立文件夹：`{terminal_dir}/bases/{server_name}/history/{symbol_name}`。例如，来自 MetaQuotes-Demo 交易服务器的 EURUSD 数据可能存储在以下文件夹中：`C:/Program Files/MetaTrader 5/bases/MetaQuotes-Demo/history/EURUSD/`。

数据被写入扩展名为 *.hcc 的文件中：每个文件存储为期一年的一分钟柱线数据。例如，EURUSD 文件夹中的 2021.hcc 文件包含 2021 年的 EURUSD 分钟柱线数据。这些文件用于为所有时间范围准备价格数据，不适合直接访问。

HCC 格式的服务文件充当绘制特定时间范围价格数据的数据源。这些文件仅在图表或 MQL 程序请求时创建，并以 *.hc 扩展名保存以供后续在文件中使用。

每个时间范围的数据都是独立生成的，与其他时间范围无关。所有时间范围（包括 M1）的数据生成和可用性规则均相同。也就是说，虽然 HCC 格式的数据存储单元是分钟柱线，但这些数据的存在并不意味着 HC 格式中也有同等数量的 M1 时间范围数据可用。

为节省资源，时间范围数据仅在需要时才会被加载并存储到 RAM 中：如果长时间没有数据访问，它们会从 RAM 中卸载（但仍保留在文件中）。这可能导致如果某个时间序列长时间未被使用，下一次请求时的执行时间延长。如果计算机资源充足，所有常用时间序列（特别是已打开图表对应的数据）几乎可以立即访问。

从服务器接收新数据后，会自动更新所有时间范围的 HC 格式价格数据，并重新计算所有依赖的指标。

当 MQL 程序访问特定交易品种和时间周期的数据时，可能出现所需的时间序列尚未生成或尚未与交易服务器同步的情况（例如，服务器上已出现更新的价格）。在这种情况下，应通过某种形式实现数据就绪等待机制。

对于脚本，唯一的解决方案是使用循环，因为它们缺乏事件处理机制，没有其他选择。对于指标，强烈不建议使用此类算法（如任何等待循环），因为这会导致该交易品种的所有指标计算和其他价格数据处理暂停。

对于 EA 交易和指标，最好使用事件处理模型。如果在处理 *OnTick* 或 *OnCalculate* 事件时未能获取所需时间序列的全部数据，应退出事件处理程序，并在下一次调用处理程序时等待数据出现。

最大柱线数量

需要注意的是，为每个请求的交易品种/时间范围对计算的最大柱线数量，不会超过终端 *Options* 对话框中 **Max. bars in chart** 参数的值。因此，该参数不仅会限制任何时间范围的图表显示，也对所有 MQL 程序产生约束。

该限制主要用于节省系统资源。若将该参数设置为较大的值，需记住，如果较低时间范围存在足够深度的历史数据，存储时间序列和指标缓冲区的内存消耗量可能达到数百兆字节，甚至占满全部 RAM。

修改柱线限制仅在重启客户端终端后生效。该参数会影响从服务器请求的数据量，以构建工作时间范围所需数量的柱线。

参数设置的限制并非绝对，在特定情况下可能会被突破。例如，在会话开始时，如果特定时间范围的报价历史足以满足整个限制条件，则随着新柱线的形成，其数量可能会超过该参数的当前值。实际可用柱线数量可通过 *Bars/iBars* 函数获取。

5.3.3 获取价格数组的特性

在读取时间序列数组之前，我们应确保这些数组可用且具备所需特性。*SeriesInfoInteger* 函数可检索基本特性，例如终端和服务器上可用历史数据的深度、特定交易品种/时间范围组合已构建的柱线数量，以及终端与服务器之间的报价是否存在差异。

该函数具有两种形式：第一种形式直接返回请求的特性值（类型为 *long*），第二种形式使用按引用传递的第四个参数 *result*。在这种情况下，第二种形式会返回一个表示成功（*true*）或错误（*false*）的标志。无论哪种情况，都可以使用 *GetLastError* 函数找到错误代码。

```
long SeriesInfoInteger(const string symbol, ENUM_TIMEFRAMES timeframe,
ENUM_SERIES_INFO_INTEGER property)
bool SeriesInfoInteger(const string symbol, ENUM_TIMEFRAMES timeframe,
ENUM_SERIES_INFO_INTEGER property, long &result)
```

该函数允许你查询指定交易品种和时间范围的时间序列特性，或整个交易品种历史数据的相关特性。所请求的特性由第三个参数（类型为 *ENUM_SERIES_INFO_INTEGER*）标识。此枚举包括所有可用特性：

标识符	说明	特性类型
SERIES_BARS_COUNT	按交易品种/周期统计的柱线数量，请参阅 Bars	long
SERIES_FIRSTDATE	按交易品种/周期统计的首根柱线日期	datetime
SERIES_LASTBAR_DATE	按交易品种/周期统计的最近一根柱线开盘时间	datetime
SERIES_SYNCHRONIZED	终端和服务器的按交易品种/周期统计的数据同步标志	bool
SERIES_SERVER_FIRSTDATE	服务器上按交易品种统计的历史最早日期（与周期无关）	datetime
SERIES_TERMINAL_FIRSTDATE	客户端终端上按交易品种统计的历史最早日期（与周期无关）	datetime

根据特性的本质，生成的值应转换为特定类型的值（请参阅 *Property type* 这一列）。

所有特性数据均实时返回当前最新值。

SeriesInfo.mq5 脚本提供了一个查询所有特性的示例。

```
void OnStart()
{
    PRTF(SeriesInfoInteger(NULL, 0, SERIES_BARS_COUNT));
    PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_FIRSTDATE));
    PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_LASTBAR_DATE));
    PRTF((bool)SeriesInfoInteger(NULL, 0, SERIES_SYNCHRONIZED));
    PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_SERVER_FIRSTDATE));
    PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_TERMINAL_FIRSTDATE));
    PRTF(SeriesInfoInteger("ABRACADABRA", 0, SERIES_BARS_COUNT));
}
```

以下是 MQ 演示服务器上针对 EURUSD、H1 所获得的结果示例：

```
SeriesInfoInteger(NULL,0,SERIES_BARS_COUNT)=10001 / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_FIRSTDATE)=2020.03.02 10:00:00 / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_LASTBAR_DATE)=2021.10.08 14:00:00 / ok
(bool)SeriesInfoInteger(NULL,0,SERIES_SYNCHRONIZED)=false / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_SERVER_FIRSTDATE)=1971.01.04 00:00:00 / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_TERMINAL_FIRSTDATE)=2016.06.01 00:00:00 / c
SeriesInfoInteger(ABRACADABRA,0,SERIES_BARS_COUNT)=0 / MARKET_UNKNOWN_SYMBOL(4301)
```

5.3.4 可用柱线数量 (Bars/iBars)

要按交易品种/周期查找时间序列中的柱线总数，有一个更简便的方法是使用函数 *Bars* 和 *iBars*（这两个函数没有区别，*iBars* 的存在是为了与 MQL4 兼容）。


```
int Bars(const string symbol, ENUM_TIMEFRAMES timeframe)
int iBars(const string symbol, ENUM_TIMEFRAMES timeframe)
```

这些函数返回针对给定交易品种和周期，可供 MQL 程序使用的柱线数量。该值会受到终端的 *Options* 中 *Max. bars in chart* 参数的影响（请参阅 [时间序列的组织与存储技术特性](#) 章节中的相关注释）。例如，如果下载到终端的历史数据针对某个特定时间范围有 20,000 根柱线，但在设置中限制为 10,000 根，则以设置中的限制值为准。终端启动后，函数将立即返回 10,000 根柱线的数量。但随着新柱线生成，返回数量将逐渐增加（如果内存足够）。在 MQL5 中，可通过调用 *TerminalInfoInteger(TERMINAL_MAXBARS)* 函数获取该限制值。

此外，*Bars* 函数还提供另一个选项，允许你查找两个日期之间的柱线数量。

```
int Bars(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, datetime stop)
```

此类请求仅查询那些开盘时间在 *start* 到 *stop* 范围内（含起始和结束时间）的柱线。无论 *start* 和 *stop* 的指定顺序如何：该函数都将按照时间从早到晚的顺序来分析报价。

如果在调用 *Bars/iBars* 函数时，具有指定参数的时间序列数据尚未生成，或尚未与交易服务器同步，则该函数将返回空值。此时，*_LastError* 中的错误特性也将为 0（不存在错误，因为数据尚未下载或准备就绪）。收到 0 后，使用 *SeriesInfoInteger(..., SERIES_SYNCHRONIZED)* 检查特定时间范围的同步情况，或者使用特殊的 *SymbolsSynchronized* 函数检查交易品种的同步情况。

关于如何使用这些函数的示例，将在下一章节的脚本 *SeriesBars.mq5* 中展示，同时还会展示与之相关的 *iBarShift* 函数。

5.3.5 按时间查找柱线索引 (iBarShift)

iBarShift 函数用于获取指定时间对应的柱线编号。这里的柱线编号始终采用时间序列中的编号规则，即索引 0 对应最右侧最新的柱线，随着从右向左移动（向过去回溯），索引值逐渐递增。

```
int iBarShift(const string symbol, ENUM_TIMEFRAMES timeframe, datetime time, bool exact = false)
```

该函数返回指定 *symbol/timeframe* 参数对的时间序列中包含 *time* 参数值的柱线索引。每个柱线由开盘时间和该时间序列柱线统一的持续时间（即周期）来界定。例如，在小时时间范围上，开盘时间标记为 13:00 的柱线从 13:00:00 持续到 13:59:59（包括整个最后一分钟和最后一秒）。

如果指定时间没有对应的柱线（例如时间处于非交易时段或非交易日），则该函数将根据参数值采取不同处理方式：如果 *precise = true*，函数将返回 -1；如果 *exact=false*，函数将返回开盘时间早于指定时间的最邻近柱线索引。若不存在这样的柱线，即指定时间之前没有历史数据，函数将返回 -1。但这里有一个细微差别。

注意！如果 *iBarShift* 函数返回一个具体的柱线编号（即非 -1 的值），这并不意味着后续通过该索引访问时间序列时一定能获取该柱线的价格或其他特性。特别是当请求的柱线索引超过终端窗口中的柱线限制（*TerminalInfoInteger(TERMINAL_MAXBARS)*）时，就可能发生这种情况。这种情况可能在新柱线形成时出现：较早的柱线可能会向左移动超出限制范围，位于可见窗口之外，尽管在名义上它们仍可能会在内存中保留一段时间。开发者必须始终检查此类情况。

我们将使用脚本 *SeriesBars.mq5* 检查 *Bars/iBars* 和（见 [上一节](#)）*iBarShift* 函数的性能。

```

void OnStart()
{
    const datetime target = PRTF(ChartTimeOnDropped());
    PRTF(iBarShift(NULL, 0, target));
    PRTF(iBarShift(NULL, 0, target, true));
    PRTF(iBarShift(NULL, 0, TimeCurrent()));
    PRTF(Bars(NULL, 0, target, TimeCurrent()));
    PRTF(Bars(NULL, 0, TimeCurrent(), target));
    PRTF(iBars(NULL, 0));
    PRTF(Bars(NULL, 0));
    PRTF(Bars(NULL, 0, 0, TimeCurrent()));
    PRTF(Bars(NULL, 0, TimeCurrent(), TimeCurrent()));
}

```

在这里我们遇到了另一个新函数 `ChartTimeOnDropped` (后续将详细说明): 该函数可返回 (活动图表中) 在 `Navigator` 中通过鼠标将脚本拖放到特定柱线的时间。首先, 让我们将脚本拖放至图表上有报价的区域。

日志中将创建如下条目 (具体数值将根据你的设置、操作及当前时间而有所不同):

```

ChartTimeOnDropped()=2021.10.01 09:00:00 / ok
iBarShift(NULL,0,target)=125 / ok
iBarShift(NULL,0,target,true)=125 / ok
iBarShift(NULL,0,TimeCurrent())=0 / ok
Bars(NULL,0,target,TimeCurrent())=126 / ok
Bars(NULL,0,TimeCurrent(),target)=126 / ok
iBars(NULL,0)=10004 / ok
Bars(NULL,0)=10004 / ok
Bars(NULL,0,0,TimeCurrent())=10004 / ok
Bars(NULL,0,TimeCurrent(),TimeCurrent())=0 / ok

```

在本例中, 脚本被拖放至时间为 2021.10.01 09:00 的柱线 (使用的是小时时间范围)。根据 `iBarShift`, 该时间对应柱线编号 125。

从鼠标所在柱线到最新柱线 (当前时间) 的柱线数量为 126 根。这一点需要结合柱线编号 125 来考虑, 因为编号是从 0 开始的。

通过不同方式获取的图表上的柱线总数 (`iBars`、不带日期范围的 `Bars` 以及包含从 0 到当前时刻 `TimeCurrent` 完整时间段的 `Bars`) 均为 10004 根。终端设置的柱线限制为 10000 根, 但在会话期间额外形成了 4 根小时柱线。

对于已存在的交易品种和时间范围, 当 `exact = false` 时, 当前时间对应的柱线编号 `iBarShift(..., TimeCurrent())` 始终为 0。如果 `exact = true`, 则有时可能返回 -1, 因为当所有市场金融工具的分时报价到来时服务器时间会增加, 而当前交易品种可能暂时没有交易。此时服务器时间可能会提前多个柱线周期, 导致 `TimeCurrent` 没有完全对应的新柱线。

如果将脚本拖放至最后一根柱线右侧的空白区域 (即未来时间区域), 将出现以下情况:

```
ChartTimeOnDropped()=2021.10.09 02:30:00 / ok
iBarShift(NULL,0,target)=0 / ok
iBarShift(NULL,0,target,true)=-1 / ok
Bars(NULL,0,target,TimeCurrent())=0 / ok
Bars(NULL,0,TimeCurrent(),target)=0 / ok
iBars(NULL,0)=10004 / ok
Bars(NULL,0)=10004 / ok
Bars(NULL,0,0,TimeCurrent())=10004 / ok
Bars(NULL,0,TimeCurrent(),TimeCurrent())=0 / ok
```

*iBarShift*函数在搜索任意前一根柱线的模式下 (*exact = false*) 返回 0, 因为当前柱线是距离未来时间点最近的。然而, 精确搜索 (*exact = true*) 时会返回结果 -1。此外, 用于计算从当前时间到“目标”未来时间范围内柱线数量的 *Bars*函数此时返回 0 (该区域尚无柱线)。

*iBarShift*函数在编写多货币 MQL 程序时特别有用。由于不同金融工具的交易时间表往往不一致, 因此对于特定时间点, 某个交易品种可能存在对应的柱线, 而另一个交易品种可能不存在。通过*iBarShift*函数的最近 (前一个)柱线搜索模式, 你始终可以获取到同一时刻不同交易品种的有效价格对应的柱线索引。通常, 即使是外汇交易品种, 同一时间的历史柱线索引也可能不同。

例如, 以下指令将在小时时间范围 (MQ 模拟服务器)下, 记录三个交易品种 (EURUSD、XAUUSD、USDRUB)在同一日期范围内的不同柱线数量及其编号:

```
PRTF(Bars("EURUSD", PERIOD_H1, D'2021.05.01', D'2021.09.01')); // 2087
PRTF(Bars("XAUUSD", PERIOD_H1, D'2021.05.01', D'2021.09.01')); // 1991
PRTF(Bars("USDRUB", PERIOD_H1, D'2021.05.01', D'2021.09.01')); // 694
PRTF(iBarShift("EURUSD", PERIOD_H1, D'2021.09.01')); // 671
PRTF(iBarShift("XAUUSD", PERIOD_H1, D'2021.09.01')); // 638
PRTF(iBarShift("USDRUB", PERIOD_H1, D'2021.09.01')); // 224
```

5.3.6 用于获取报价数组的 Copy 函数概述

MQL5 API包含多个用于将报价时间序列读取到数组中的函数。它们的名称如下表所示。

函数	操作
CopyRates	将报价历史数据获取到 <i>MqlRates</i> 结构体数组中
CopyTime	将柱线开盘时间历史数据获取到以下类型的数组中: <i>datetime</i>
CopyOpen	将柱线开盘价历史数据获取到以下类型的数组中: <i>double</i>
CopyHigh	将柱线最高价历史数据获取到以下类型的数组中: <i>double</i>
CopyLow	将柱线最低价历史数据获取到以下类型的数组中: <i>double</i>
CopyClose	将柱线收盘价历史数据获取到以下类型的数组中: <i>double</i>
CopyTickVolume	将分时报价成交量历史数据获取到以下类型的数组中: <i>long</i>
CopyRealVolume	将成交量历史数据获取到以下类型的数组中: <i>long</i>
CopySpread	将点差历史数据获取到以下类型的数组中: <i>int</i>

所有函数的前两个参数均为所需交易品种的名称和周期，这可以通过以下伪代码有条件地表示：

```
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, ...)
```

此外，所有函数的原型均有三种变体，其区别在于请求范围的设置方式：

- 起始柱线索引和柱线数量：*Copy***(..., int offset, int count, ...)*
- 范围起始时间和柱线数量：*Copy***(..., datetime start, int count, ...)*
- 范围起始时间和结束时间：*Copy***(..., datetime start, datetime stop, ...)*

同时，参数表示法意味着所请求的数据具有时间序列式的索引方向，即索引为 0 的 *offset* 位置存储当前未完成柱线的数据，索引增大对应向价格历史的更早期移动。因此，特别是对于第二种选项，所指定的柱线数量 *count* 将从 *offset* 范围起始点开始反向计数，即沿时间递减的方向计数。

第三种选项提供了更高的灵活性：起始日期和结束日期的指定顺序 (*start/stop*) 无关紧要，因为函数在任何情况下都会返回从较小日期到较大日期范围内的数据。符合条件的柱线将按以下方式选择：其开盘时间位于 *start/stop* 时间点之间，或等于其中之一，即区间 [*start*; *stop*] 为包含边界的闭区间。

选择哪种函数选项取决于以下哪个因素更重要：是获取有保证的元素数量（例如用于机器学习算法），还是覆盖特定日期区间（例如针对预先确定的统一市场行为时段）。

datetime 类型的时间表示精度为 1 秒。*start/stop* 值无需按周期大小取整。例如，从 14:59 到 16:01 的时间范围将允许你在 H1 时间范围上选择 15:00 和 16:00 两个柱形。具有相等且取整标签的退化范围（例如 H1 报价中的 15:00）对应一个柱形。

即使起始/终止参数中存在非零的小时/分钟/秒（尽管 D1 时间范围的柱线标签时间为 00:00），你仍可请求日时间范围的柱线。在这种情况下，仅开盘时间位于 *start/stop* 最小值之后且截止到 *start/stop* 最大值的 D1 柱线（由于所需时间包含小时/分钟/秒，因此此时不可能与日柱线的标签完全一致）。例如，在 D'2021.09.01 12:00' 至 D'2021.09.03 07:00' 之间，存在两根 D1 柱线的开盘时间 - D'2021.09.02' 和 D'2021.09.03'。这些柱线将包含在结果中。D'2021.09.01' 柱线的开盘时间为 00:00，早于时间范围的起始点，因此会被排除。D'2021.09.03' 柱线会被包含在结果中，尽管当天只有早上 7 小时属于该时间范围。另一方面，如果请求的是一天内的几个小时，例如在 D'2021.09.01 12:00' 至 D'2021.09.01 15:00' 之间，该时间段将不覆盖任何单个日柱线（D'2021.09.01' 柱线的开盘时间不在此范围内），因此接收数组将为空。

表格中所有函数之间的唯一区别在于接收数据的数组类型，该数组作为最后一个参数通过引用传递。例如，*CopyRates* 函数将请求的数据放入结构体数组 *MqlRates* 中，而 *CopyTime* 函数则将柱线开盘时间放入 *datetime* 类型的数组中，依此类推。

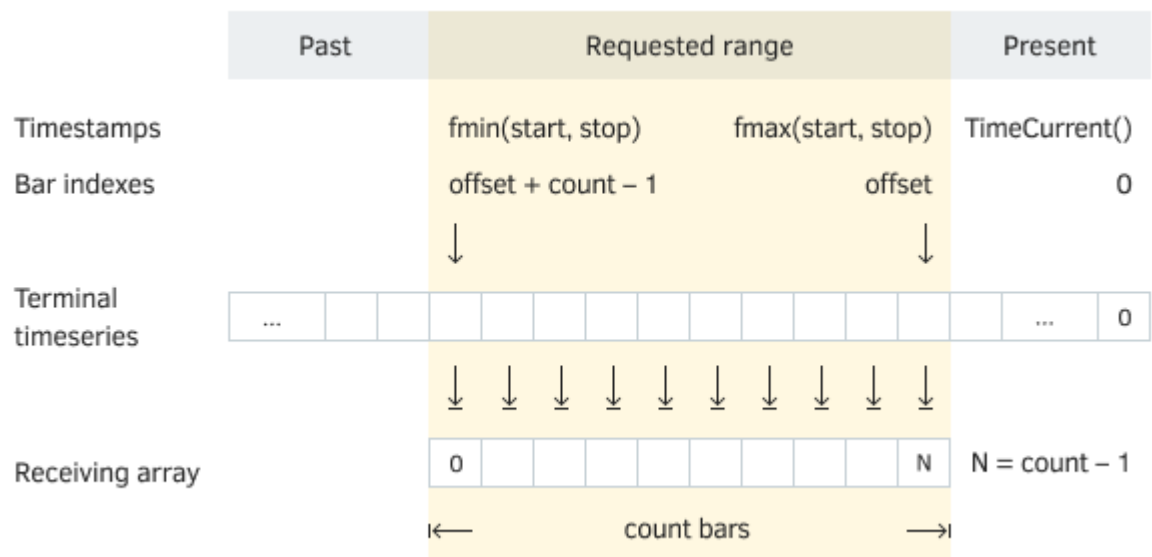
因此，常见函数原型可表示如下：

```
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, int offset, int count, type &result[])
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, int count, type &result[])
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, datetime stop, type &result[])
```

此处，*type* 与 *MqlRates*、*datetime*、*double*、*long* 或 *int* 中的任意一种类型匹配，具体取决于函数本身。

这些函数会返回复制到数组中的元素数量；如果发生错误，则返回 -1。具体来说，如果服务器上没有请求时间间隔内的数据，或该时间间隔超出了图表上的最大柱线数（通过 *(TerminalInfoInteger(TERMINAL_MAXBARS))* 获取），我们将得到 -1。

需要特别注意的是，在接收数组中，所接收的数据始终按物理顺序按时间先后排列（从过去到未来）。因此，如果对接收数组使用标准索引（即通过 `ArraySetAsSeries` 函数设置），则索引 0 处的元素为最早数据，而最后一个元素为最新数据。如果对数组执行了 `ArraySetAsSeries(result, true)` 指令，则编号将按时间序列的相反顺序进行：索引 0 处的元素将是时间范围内的最新数据，而最后一个元素将是最旧的数据。这一点如下图所示。



终端时间序列和接收数组

如果成功，终端自身（内部）时间序列中指定数量的元素将被复制到目标数组。当按日期范围（`start/stop`）请求数据时，结果数组中的元素数量将基于该范围内的历史内容间接确定。因此，要复制数量未知的值，建议使用动态数组：复制函数会独立分配目标数组所需的容量（大小可以增大或减小）。

如果你需要复制已知数量的元素，或者需要频繁进行复制操作（例如在 EA 交易的 `OnTick` 函数或指标的 `OnCalculate` 函数每次调用时），则最好使用静态分配的数组。事实上，动态数组的内存分配操作需要额外时间，并且可能影响性能，尤其是在测试和优化期间。

如果请求的数据尚未准备好，不同类型的 MQL 程序对时间序列的访问方式也会有所不同。例如，在自定义 `指标`，`Copy` 中，函数会立即返回错误，因为指标在终端的公共界面线程中执行，无法等待数据接收（通常假设指标会在其事件处理程序的下一次调用时请求数据，而此时时间序列已完成下载和构建）。此外，在关于指标的章节中，我们将了解到，要访问指标所在的“原生”图表的报价，指标无需使用 `Copy` 函数，因为所有时间序列都会通过处理程序 `OnCalculate` 的数组参数自动传递。

当从 EA 交易和脚本访问时，会尝试多次接收数据，并在每次尝试之间进行短暂暂停（函数内部会进行等待），以便系统有时间加载和计算缺失的时间序列。该函数将返回超时到期时已准备好的数据量，但历史数据加载会持续进行，下一次类似请求将返回更多数据。

无论如何情况，你都应做好准备，即 `Copy` 函数可能会返回错误而非数据（原因多种多样：连接失败、请求的数据不存在、并行请求大量新时间序列导致处理器负载过高等）：在代码中分析问题原因（`LastError`），稍后重试，修正设置，或通知用户。

交易品种是否显示在 `Market Watch` 中并非使用 `Copy` 函数请求其时间序列数据的必要条件，不过，对于已添加到该窗口的交易品种，相关查询通常会执行得更快，因为部分数据可能已从服务器下载完毕，并且可能已针对请求的周期完成计算。关于如何通过编程方式向 `Market Watch` 添加交易品种，我们将在 `编辑市场观察列表` 章节中详细介绍。

为了在实践中解释这些函数的工作原理，我们以脚本 *SeriesCopy.mq5* 为例进行说明。该脚本多次调用 *CopyTime* 函数，这使你能够直观地看到时间戳与柱线编号之间的关联。

该脚本定义了一个动态数组 *times* 来接收数据。所有请求均针对 “EURUSD” 交易品种和 H1 时间范围。

```
void OnStart()
{
    datetime times[];
```

首先，请求从 2021 年 9 月 5 日开始，向过去方向的 10 根柱线数据。由于这一天是周日，因此前一个交易日的柱线是 3 日周五（见下方日志）。

```
PRTF(CopyTime("EURUSD", PERIOD_H1, D'2021.09.05', 10, times)); // 10 / ok
ArrayPrint(times);
/*
[0] 2021.09.03 14:00 2021.09.03 15:00 2021.09.03 16:00 2021.09.03 17:00 2021.09.03 18:00
[5] 2021.09.03 19:00 2021.09.03 20:00 2021.09.03 21:00 2021.09.03 22:00 2021.09.03 23:00
*/
```

数组的输出默认按时间顺序进行（尽管函数参数是在反向坐标系中设置的：如在时间序列中）。我们来更改接收数组中的索引顺序，然后重新输出。

```
PRTF(ArraySetAsSeries(times, true)); // true / ok
ArrayPrint(times);
/*
[0] 2021.09.03 23:00 2021.09.03 22:00 2021.09.03 21:00 2021.09.03 20:00 2021.09.03 19:00
[5] 2021.09.03 18:00 2021.09.03 17:00 2021.09.03 16:00 2021.09.03 15:00 2021.09.03 14:00
*/
```

对于后续实验，我们将恢复常规顺序。

```
PRTF(ArraySetAsSeries(times, false)); // true / ok
```

现在，我们来请求两个时间点之间的不确定数量的柱线（数量未知，例如因为该时间段内可能存在假期）。我们将通过两种方式实现：第一种情况，指定从未来到过去的时间范围；第二种情况，指定从过去到未来的时间范围。结果两者一致。

```
//                                FROM                                TO
PRTF(CopyTime("EURUSD", PERIOD_H1, D'2021.09.06 03:00', D'2021.09.05 03:00', times
ArrayPrint(times) //                                FROM                                TO
PRTF(CopyTime("EURUSD", PERIOD_H1, D'2021.09.05 03:00', D'2021.09.06 03:00', times
ArrayPrint(times);
/*
CopyTime(EURUSD,PERIOD_H1,D'2021.09.06 03:00',D'2021.09.05 03:00',times)=4 / ok
2021.09.06 00:00 2021.09.06 01:00 2021.09.06 02:00 2021.09.06 03:00
CopyTime(EURUSD,PERIOD_H1,D'2021.09.05 03:00',D'2021.09.06 03:00',times)=4 / ok
2021.09.06 00:00 2021.09.06 01:00 2021.09.06 02:00 2021.09.06 03:00
*/
```

通过输出数组，我们可以看到它们是相同的。让我们回到时间序列索引模式，并讨论另一个要点。

```
PRTF(ArraySetAsSeries(times, true)); // true / ok
ArrayPrint(times);
```



```
// 2021.09.06 03:00 2021.09.06 02:00 2021.09.06 01:00 2021.09.06 00:00
```

尽管两个时间戳相隔 24 小时，这意味着数组中应包含 25 个元素（请记住起始和结束时间都包含在内），但结果仅包含 4 根柱线。原因是 9 月 5 日是周日，因此在整个时间范围内，仅在 6 日上午有交易活动。

另外，请注意接收数组的大小已自动从 10 个元素缩减为 4 个。

最后，我们将从第 100 根柱线开始请求 10 根柱线（具体结果将取决于你当前的时间和可用历史数据）。

```
PRTF(CopyTime("EURUSD", PERIOD_H1, 100, 10, times)); // 10 / ok
ArrayPrint(times);
/*
[0] 2021.10.04 19:00 2021.10.04 18:00 2021.10.04 17:00 2021.10.04 16:00 2021.10.04
[5] 2021.10.04 14:00 2021.10.04 13:00 2021.10.04 12:00 2021.10.04 11:00 2021.10.04
*/
}
```

由于采用时间序列式的索引方式，数组将按逆时间顺序显示。

5.3.7 作为 MqlRates 结构体的数组获取报价

如果要请求包含所有柱线特征的报价数组，可使用具备多种重载形式的 *CopyRates* 函数。

```
int CopyRates(const string symbol, ENUM_TIMEFRAMES timeframe, int offset, int count, MqlRates &rates[])
int CopyRates(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, int count, MqlRates &rates[])
int CopyRates(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, datetime stop, MqlRates &rates[])
```

该函数会将指定参数的历史数据填入 *rates* 数组：交易品种、时间范围以及由柱线编号或 *datetime* 类型的 *start/stop* 值所确定的时间范围。

该函数返回已复制的数组元素数量；如果出现错误则返回 -1，错误代码可通过 *_LastError* 函数找到。具体来说，在以下情况下会出现错误：指定了不存在的交易品种；时间区间内服务器无可用的数据；或者时间区间超出了图表最大柱线数量限制（可通过 *TerminalInfoInteger(TERMINAL_MAXBARS)* 获取该限制值）。

此函数的基本使用方法与所有 *Copy* 函数一致，相关内容已在 *获取报价数组的 Copy 函数概述* 章节中详细说明。

内联类型结构体 *MqlRates* 的描述如下：

```

struct MqlRates
{
    datetime time;           // bar opening time
    double open;             // opening price
    double high;             // maximum price per bar
    double low;              // minimum price per bar
    double close;            // closing price
    long tick_volume;        // tick volume per bar
    int spread;              // minimum spread per bar in points
    long real_volume;        // exchange volume per bar
};

```

让我们尝试在脚本 *SeriesStats.mq5* 中应用该函数来计算柱线的平均大小。在输入变量中，我们将提供选择交易品种、时间范围、分析柱线数量以及向历史初始偏移量的功能（0 表示从当前柱线开始分析）。

```

input string WorkSymbol = NULL; // Symbol (leave empty for current)
input ENUM_TIMEFRAMES TimeFrame = PERIOD_CURRENT;
input int BarOffset = 0;
input int BarCount = 10000;

void OnStart()
{
    MqlRates rates[];
    double range = 0, move = 0; // calculate the range and price movement in bars

    PrintFormat("Requesting %d bars on %s %s",
        BarCount, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol,
        EnumToString(TimeFrame == PERIOD_CURRENT ? _Period : TimeFrame));

    // request all information about BarCount bars to the MqlRates array
    const int n = PRTF(CopyRates(WorkSymbol, TimeFrame, BarOffset, BarCount, rates));

    // in the loop we calculate the average for the range and movement
    for(int i = 0; i < n; ++i)
    {
        range += (rates[i].high - rates[i].low) / n;
        move += (fmax(rates[i].open, rates[i].close)
            - fmin(rates[i].open, rates[i].close)) / n;
    }

    PrintFormat("Stats per bar: range=%f, movement=%f", range, move);
    PrintFormat("Dates: %s - %s",
        TimeToString(rates[0].time), TimeToString(rates[n - 1].time));
}

```

将该脚本应用于 EURUSD,H1 图表时，我们可以得到大致如下结果。


```
Requesting 100000 bars on EURUSD PERIOD_H1
CopyRates(WorkSymbol,TimeFrame,BarOffset,BarCount,rates)=20018 / ok
Stats per bar: range=0.001280, movement=0.000621
Dates: 2018.07.19 15:00 - 2021.10.11 17:00
```

由于终端设置了 20,000 根柱线的限制，即使请求获取 100,000 根柱线数据，实际也只能返回 20018 根（柱线上限加上会话开始后新形成的柱线数量）数组的第一个元素（索引为 0）对应的是 2018.07.19 15:00 的柱线，最后一个元素对应 2021.10.11 17:00 的柱线。

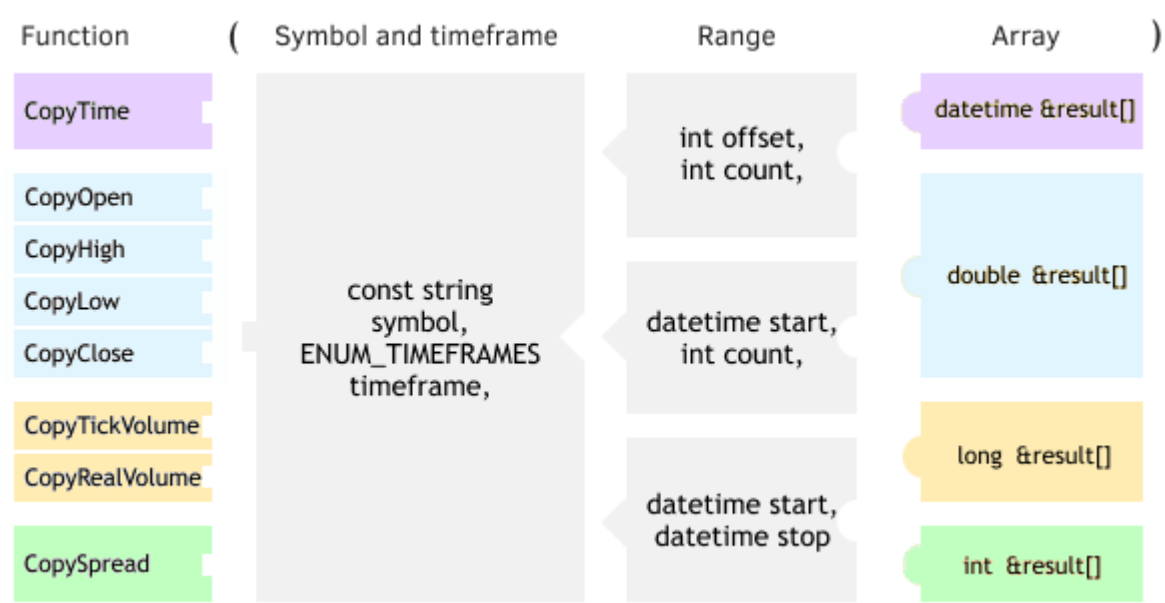
根据统计数据，在此期间柱线的平均波幅为 128 点，开盘价与收盘价之间的波动为 62 点。

当使用开始和结束日期（*start/stop*）请求信息时，请记住两个边界均被视为包含在内。因此，如果要设置一个与更高时间范围柱线对应的时间区间，应当从右边界减去 1 秒。我们将在 [按柱线索引读取价格、成交量、点差和时间](#) 章节的 *SeriesSpread.mq5* 示例中应用这一技巧。

5.3.8 单独请求价格、交易量、点差、时间的数组

可以将特定字段（价格、成交量、点差或时间）的数据读取到单独的数组中，而不必将所有编码特征作为 *MqlRates* 数组进行查询。为此，定义了几个函数，其操作原理基于 [获取报价数组的 Copy 函数概述](#) 章节中讨论的通用原则。

下图汇总了所有函数原型的描述。



Copy 函数原型图

SeriesRates.mq5 脚本使用复制 OHLC 价格的函数，将它们与调用 *CopyRates* 的结果进行比较。

```

void OnStart()
{
    const int N = 10;
    MqlRates rates[];

    // request and display all information about N bars from the MqlRates array
    PRTF(CopyRates("EURUSD", PERIOD_D1, D'2021.10.01', N, rates));
    ArrayPrint(rates);

    // now request OHLC prices separately
    double open[], high[], low[], close[];
    PRTF(CopyOpen("EURUSD", PERIOD_D1, D'2021.10.01', N, open));
    PRTF(CopyHigh("EURUSD", PERIOD_D1, D'2021.10.01', N, high));
    PRTF(CopyLow("EURUSD", PERIOD_D1, D'2021.10.01', N, low));
    PRTF(CopyClose("EURUSD", PERIOD_D1, D'2021.10.01', N, close));

    // compare prices obtained by different methods
    for(int i = 0; i < N; ++i)
    {
        if(rates[i].open != open[i]
           || rates[i].high != high[i]
           || rates[i].low != low[i]
           || rates[i].close != close[i])
        {
            // we shouldn't be here
            Print("Data mismatch at ", i);
            return;
        }
    }

    Print("Copied OHLC arrays match MqlRates array"); // success: there is a match
}

```

运行脚本后，我们将会在日志中获得以下条目。

```
CopyRates(EURUSD,PERIOD_D1,D'2021.10.01',N,rates)=10 / ok
[time] [open] [high] [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.09.20 00:00:00 1.17272 1.17363 1.17004 1.17257 58444 0 0
[1] 2021.09.21 00:00:00 1.17248 1.17486 1.17149 1.17252 58514 0 0
[2] 2021.09.22 00:00:00 1.17240 1.17555 1.16843 1.16866 72571 0 0
[3] 2021.09.23 00:00:00 1.16860 1.17501 1.16835 1.17381 68536 0 0
[4] 2021.09.24 00:00:00 1.17379 1.17476 1.17007 1.17206 51401 0 0
[5] 2021.09.27 00:00:00 1.17255 1.17255 1.16848 1.16952 57807 0 0
[6] 2021.09.28 00:00:00 1.16940 1.17032 1.16682 1.16826 64793 0 0
[7] 2021.09.29 00:00:00 1.16825 1.16901 1.15894 1.15969 68964 0 0
[8] 2021.09.30 00:00:00 1.15963 1.16097 1.15626 1.15769 68517 0 0
[9] 2021.10.01 00:00:00 1.15740 1.16075 1.15630 1.15927 66777 0 0
CopyOpen(EURUSD,PERIOD_D1,D'2021.10.01',N,open)=10 / ok
CopyHigh(EURUSD,PERIOD_D1,D'2021.10.01',N,high)=10 / ok
CopyLow(EURUSD,PERIOD_D1,D'2021.10.01',N,low)=10 / ok
CopyClose(EURUSD,PERIOD_D1,D'2021.10.01',N,close)=10 / ok
复制的 OHLC 数组与 MqlRates 数组相匹配
```

请记住，**tick_volume** 字段中的分时报价成交量是某个周期的分时报价的简单计数器。对于非交易所金融工具（以及本例中的 EURUSD），**real_volume** 字段中的交易所成交量为零。

在 [获取报价数组的 Copy 函数概述](#) 章节的 *SeriesCopy.mq5* 脚本中，提供了使用 **CopyTime** 函数的另一个示例。

5.3.9 通过柱线索引读取价格、成交量、点差和时间

有时，你需要查询的不是一系列柱线的信息，而仅是某一根柱线的信息。从理论上讲，这可以通过使用前面讨论过的 **Copy** 函数来实现，只需在函数中指定数量 (**count** 参数) 为 1，但这种方式并不够便捷。以下函数提供了更简单的方法，它们可以根据柱线在时间序列中的编号，返回该柱线的某个特定类型的单一值。

所有函数都具有相似的原型，但名称和返回类型不同。从历史上看，这些函数名称均以前缀 **i** 开头，形式为 **iValue**（这些函数属于内置技术指标的大类：毕竟报价的各种特性是技术分析的主要数据来源，而几乎所有指标都是其衍生指标，因此使用字母 **i**）。

type iValue(const string symbol, ENUM_TIMEFRAMES timeframe, int offset)

其中，**type** 对应 **datetime**、**double**、**long** 或 **int** 中的一种类型，具体取决于函数本身。**symbol** 和 **timeframe** 用于指定要查询的时间序列。所需柱线的索引 **offset** 采用时间序列的表示方法：0 表示最新的、最右侧的柱线（通常尚未完成），数值越大，表示越早期的柱线。与 **Copy** 函数的情况类似，可以使用 **NULL** 和 0 来指定交易品种和时间周期，使其与当前图表的特性保持一致。

由于 **i** 函数等效于调用 **Copy** 函数，因此在 [获取报价数组的 Copy 函数概述](#) 章节中描述的所有从不同类型程序请求时间序列的特性，同样适用于这些函数。

函数	说明
iTime	柱线开盘时间
iOpen	柱线开盘价
iHigh	柱线最高价
iLow	柱线最低价
iClose	柱线收盘价
iTickVolume	柱线的分时报价成交量 (与 iVolume 类似)
iVolume	柱线的分时报价成交量 (与 iTickVolume 类似)
iRealVolume	柱线的真实交易量
iSpread	柱线的最小点差 (以点为单位)

这些函数会返回请求的值，如果出错则返回 0 (遗憾的是，在某些情况下 0 可能是一个有效值)。如需获取详细的错误信息，可调用 `GetLastError` 函数。

这些函数不会缓存结果。每次调用时，它们都会从指定交易品种/时间周期的时间序列中返回实时数据。这意味着，在没有现成数据的情况下 (首次调用时或同步丢失后)，函数可能需要一些时间来准备结果。

举例来说，我们尝试获取每根柱线点差大小的近似真实值估算。由于报价中存储的是最小点差值，这可能会导致在设计交易策略时产生不切实际的过高预期。要获取每根柱线的平均点差、中位数点差或最大点差的绝对精确值，就必须分析实际的分时报价数据，但目前我们还未学习如何处理这类数据。此外，这也会是一个资源消耗极大的过程。更合理的方法是分析较低的 M1 时间范围的点差：对于较高时间范围的柱线，只需在其内部包含的 M1 柱线中查找最大点差即可。当然严格来说，这并非真正的最大值，而是最小点差值中的最大值，但考虑到分钟数据的瞬时特性，我们至少有望在某些 M1 柱线上检测到典型点差扩大情况，这足以在分析精度和速度之间取得可接受的平衡。

脚本 `SeriesSpread.mq5` 中实现了该算法的一个版本。在输入变量中，你可以设置用于分析的交易品种、时间范围和柱线数量。默认情况下，处理当前图表的交易品种及其时间周期 (必须大于 M1)。

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input ENUM_TIMEFRAMES TimeFrame = PERIOD_CURRENT;
input int BarCount = 100;
```

由于每个柱线只需关注其时间和点差信息，因此定义了一个包含两个字段的特殊结构体。我们本可以使用标准的 `MqlRates` 结构体，并将“最大”点差数据添加到某个未使用的字段中 (例如外汇交易品种的 `real_volume`)，但这样会复制大部分字段的数据，造成内存浪费。

```
struct SpreadPerBar
{
    datetime time;
    int spread;
};
```

通过使用新的结构体类型，我们准备了一个 `peaks` 数组，用于计算指定数量的柱线数据。

```

void OnStart()
{
    SpreadPerBar peaks[];
    ArrayResize(peaks, BarCount);
    ZeroMemory(peaks);
    ...
}

```

随后,算法的主体部分在柱线循环中执行。对于每个柱线,我们使用*iTime*函数确定定义该柱线边界的两个时间戳。实际上,这两个时间戳分别是第 *i* 根柱线和相邻的第 *i+1* 根柱线的开盘时间。根据索引原则,可以认为第 *i+1* 根柱线是前一根柱线 (更早,见变量 *prev*),而第 *i* 根柱线是后一根柱线 (更新,见变量 *next*)。柱线的开盘时间仅属于该柱线,即标签 *prev* 属于第 *i+1* 根柱线,而标签 *next* 属于第 *i* 根柱线。因此,在处理每根柱线时,应将其右边界排除在区间 [*prev*; *next*) 之外。

我们关注的是一分钟时间周期上的点差,因此将使用 *CopySpread* 函数并指定 *PERIOD_M1* 参数。在这种情况下,通过将 *start/stop* 参数分别设置为精确的 *prev* 值和减少 1 秒的 *next* 值,可实现半开区间的设置。点差信息被复制到动态数组 *spreads* 中 (内存由函数自动分配)。

```

for(int i = 0; i < BarCount; ++i)
{
    int spreads[]; // receiving array for M1 spreads inside the i-th bar
    const datetime next = iTime(WorkSymbol, TimeFrame, i);
    const datetime prev = iTime(WorkSymbol, TimeFrame, i + 1);
    const int n = CopySpread(WorkSymbol, PERIOD_M1, prev, next - 1, spreads);
    const int m = ArrayMaximum(spreads);
    if(m > -1)
    {
        peaks[i].spread = spreads[m];
        peaks[i].time = prev;
    }
}

```

然后,我们在该数组中找出最大值,并将其与柱线时间一起保存在相应的 *SpreadPerBar* 结构体中。请注意,零号柱线不完整,未包含在分析中 (如有需要,可对算法进行补充)。

循环结束后,我们将结构体数组输出到日志中。

```

PrintFormat("Maximal speeds per intraday bar\nProcessed %d bars on %s %s",
    BarCount, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol,
    EnumToString(TimeFrame == PERIOD_CURRENT ? _Period : TimeFrame));
ArrayPrintM(peaks);

```

在 EURUSD,H1 图表上运行该脚本后,我们将获得小时柱线内的点差统计数据 (节选):

```

Maximal speeds per intraday bar
Processed 100 bars on EURUSD PERIOD_H1
[ 0] 2021.10.12 14:00      1
[ 1] 2021.10.12 13:00      1
[ 2] 2021.10.12 12:00      1
[ 3] 2021.10.12 11:00      1
[ 4] 2021.10.12 10:00      0
[ 5] 2021.10.12 09:00      1
[ 6] 2021.10.12 08:00      2
[ 7] 2021.10.12 07:00      2
[ 8] 2021.10.12 06:00      1
[ 9] 2021.10.12 05:00      1
[10] 2021.10.12 04:00      1
[11] 2021.10.12 03:00      1
[12] 2021.10.12 02:00      4
[13] 2021.10.12 01:00     16
[14] 2021.10.12 00:00     65
[15] 2021.10.11 23:00     15
[16] 2021.10.11 22:00      2
[17] 2021.10.11 21:00      1
[18] 2021.10.11 20:00      1
[19] 2021.10.11 19:00      2
[20] 2021.10.11 18:00      1
[21] 2021.10.11 17:00      1
[22] 2021.10.11 16:00      1
[23] 2021.10.11 15:00      2
[24] 2021.10.11 14:00      1

```

夜间点差明显扩大：例如临近午夜时，报价中的点差为 7-15 点，而在我们的测量中达到 15-65 点。虽然小时柱线的指标通常显示为零，但在其他时段也能发现非零值。

5.3.10 在时间序列中查找最大值和最小值

在处理报价时间序列的函数组中，有两个提供最简单聚合处理的函数：用于查找给定时间区间内序列的最大值和最小值，分别为 *iHighest* 和 *iLowest*。

```

int iHighest(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_SERIESMODE type, int count
= WHOLE_ARRAY, int offset = 0)
int iLowest(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_SERIESMODE type, int count =
WHOLE_ARRAY, int offset = 0)

```

这两个函数将返回特定时间序列类型中最大值/最小值的索引，时间序列类型由 *symbol/timeframe* 参数对以及 `ENUM_SERIESMODE` 枚举元素（用于描述我们已熟知的报价字段）共同指定。

标识符	说明
MODE_OPEN	开盘价
MODE_LOW	最高价
MODE_HIGH	最低价
MODE_CLOSE	收盘价
MODE_VOLUME	分时报价成交量
MODE_REAL_VOLUME	实际成交量
MODE_SPREAD	点差

*offset*参数用于指定开始搜索的索引位置。编号方式与时间序列一致，即*offset*增大意味着向历史方向移动，而 0 号索引代表当前柱线（此为默认值）。分析的柱线数量通过 *count*参数指定（默认值为 `W HOLE_ARRAY`）。

如果出现错误，函数将返回 -1。使用 *GetLastError* 获取错误代码。

为了演示其中一个函数（*iHighest*）的工作原理，我们将修改上一节中通过柱线估算点差实际大小的示例，并对结果进行比较。当然，它们必须匹配。脚本的新版本附在文件 *SeriesSpreadHighest.mq5*中。

这些变化影响了 *SpreadPerBar*结构体以及 *OnStart* 内部的工作循环。

该结构体中添加了一些字段，这些字段允许你了解新函数的工作原理。由于算法的特性，这些字段不是必填字段。

```
struct SpreadPerBar
{
    datetime time;
    int spread;
    int max; // through index of the M1 bar with a spread, the value of which is maximum
            // among all M1-bars within the current bar of the higher timeframe
    int num; // number of M1 bars in the current bar of the higher timeframe
    int pos; // initial index of the M1 bar within the current bar of the higher timeframe
};
```

主要修改影响了 *OnStart*，但这些修改都位于循环内部（所有其他代码片段保持不变）。

```
for(int i = 0; i < BarCount; ++i)
{
    const datetime next = iTime(WorkSymbol, TimeFrame, i);
    const datetime prev = iTime(WorkSymbol, TimeFrame, i + 1);
    ...
}
```

当前柱线的边界（*prev*和*next*）的定义与之前相同。不过，我们不再将这些标签之间的时间序列元素复制到自有数组 *spreads*中，也不再随后对其调用*ArrayMaximum* 函数，而是确定构成更高时间范围当前柱线的 M1 柱线的索引及数量。这可以按以下方式完成。

*iBarShift*函数允许你确定 M1 历史数据中的偏移量（变量 *p*），该偏移量指向时间为*next - 1* 的柱线右边界所在位置。*bars*函数用于计算介于 *prev* 和 *next - 1* 之间的 M1 柱线数量（变量*n*）这两个

值将作为参数传入 *iHighest* 函数调用中, 用于在从索引 *p* 开始的 *n* 个 M1 柱线中, 找出 MODE_SPREAD 类型的最大值。如果能够顺利找到最大值 (*m* > -1), 我们只需使用 *iSpread* 获取相应值并将其存入结构体即可。

```
const int p = iBarShift(WorkSymbol, PERIOD_M1, next - 1);
const int n = Bars(WorkSymbol, PERIOD_M1, prev, next - 1);
const int m = iHighest(WorkSymbol, PERIOD_M1, MODE_SPREAD, n, p);
if(m > -1)
{
    peaks[i].spread = iSpread(WorkSymbol, PERIOD_M1, m);
    peaks[i].time = prev;
    peaks[i].max = m;
    peaks[i].num = n;
    peaks[i].pos = p;
}
}
```

将包含结果的数组输出到日志时, 我们现在还会看到 M1 柱线的索引, 其中包括更高时间范围柱线“起始”的位置, 以及在该柱线中找到的最大点差的位置。“起始”一词之所以加上引号, 是因为随着新的 M1 柱线不断生成, 这些索引会逐渐递增, 每条柱线的虚拟“起始位置”也会随之移动, 不过历史柱线的开盘时间当然是保持不变的。


```

Maximal speeds per intraday bar
Processed 100 bars on EURUSD PERIOD_H1
      [time] [spread] [max] [num] [pos]
[ 0] 2021.10.12 15:00      0    7   60    7
[ 1] 2021.10.12 14:00      1   89   60   67
[ 2] 2021.10.12 13:00      1  181   60  127
[ 3] 2021.10.12 12:00      1  213   60  187
[ 4] 2021.10.12 11:00      1  248   60  247
[ 5] 2021.10.12 10:00      0  307   60  307
[ 6] 2021.10.12 09:00      1  385   60  367
[ 7] 2021.10.12 08:00      2  469   60  427
[ 8] 2021.10.12 07:00      2  497   60  487
[ 9] 2021.10.12 06:00      1  550   60  547
[10] 2021.10.12 05:00      1  616   60  607
[11] 2021.10.12 04:00      1  678   60  667
[12] 2021.10.12 03:00      1  727   60  727
[13] 2021.10.12 02:00      4  820   60  787
[14] 2021.10.12 01:00     16  906   60  847
[15] 2021.10.12 00:00     65  956   60  907
[16] 2021.10.11 23:00     15  967   60  967
[17] 2021.10.11 22:00      2 1039   60 1027
[18] 2021.10.11 21:00      1 1090   60 1087
[19] 2021.10.11 20:00      1 1148   60 1147
[20] 2021.10.11 19:00      2 1210   60 1207
[21] 2021.10.11 18:00      1 1313   60 1267
[22] 2021.10.11 17:00      1 1345   60 1327
[23] 2021.10.11 16:00      1 1411   60 1387
[24] 2021.10.11 15:00      2 1461   60 1447
[25] 2021.10.11 14:00      1 1526   60 1507
...

```

例如，在脚本启动时，标签为 2021.10.12 14:00 的柱线是从第 67 根 M1 柱线开始的（即它在 67 分钟前开盘），而在该 H1 柱线内具有最大点差的 M1 柱线的索引为 89。显然，该索引应小于前一个 H1 柱线起始位置的 M1 柱线编号：2021.10.12 13:00 - 其标记为 127 分钟前开盘。在该 H1 柱线中，又发现索引 181 处存在最大点差。而该数值小于更早的 2021.10.12 12:00 柱线所对应的索引 187。

*pos*和*max* 列中的索引值会持续递增，这是因为我们按照从当前到过去的时间顺序遍历柱线数据。*num*列中的值几乎均为 60，因为大多数 H1 柱线都是由 60 个 M1 柱线构成。但实际情况并非总是如此。例如，以下是不完整的小时柱线，由不足一小时的数据构成：这可能是由于节假日导致市场提前休市，或是交易活动出现缺口（流动性不足）所致。

```

...
[38] 2021.10.11 01:00     20  2346   60  2287
[39] 2021.10.11 00:00     85  2404   58  2347
[40] 2021.10.08 23:00     15  2406   55  2405
[41] 2021.10.08 22:00      2  2463   60  2460
...

```

5.3.11 使用 MqlTick 结构处理实时分时报价数组

MetaTrader 5 不仅提供处理报价历史 (柱线) 的功能, 还支持处理真实分时报价历史。在用户界面中, 所有历史数据均可通过 *Symbols* 对话框获取。该对话框包含三个选项卡: *Specification*、*Bars* 和 *Ticks*。当在第一个选项卡的树状交易品种列表中选中某个具体品种后, 切换到 *Bars* 和 *Ticks* 选项卡时, 即可分别请求该品种的柱线或分时报价形式的报价。

在 MQL 程序中, 也可以通过 *CopyTicks* 和 *CopyTicksRange* 函数访问实时分时报价历史。

```
int CopyTicks(const string symbol, MqlTick &ticks[], uint flags = COPY_TICKS_ALL, ulong from = 0,
uint count = 0)
int CopyTicksRange(const string symbol, MqlTick &ticks[], uint flags = COPY_TICKS_ALL, ulong from
= 0, ulong to = 0)
```

这两个函数将向通过引用传递的数组 *ticks* 请求指定金融工具 *symbol* 的分时报价。*MqlTick* 结构包含了单个分时报价的全部信息, 其在 MQL5 中的描述如下:

```
struct MqlTick
{
    datetime time;           // time of this price update
    double    bid;           // current Bid price
    double    ask;           // current Ask price
    double    last;          // Last trade price
    ulong     volume;        // volume for Last price
    long      time_msc;      // time of this price update in milliseconds
    uint      flags;         // flags (which fields of the structure have changed)
    double    volume_real;   // volume for the Last price with increased accuracy
};
```

flags 字段用于存储标志掩码, 指示分时报价结构中哪些字段的值发生了变化。

常量	值	说明
TICK_FLAG_BID	2	买价已变更
TICK_FLAG_ASK	4	卖价已变更
TICK_FLAG_LAST	8	最新成交价已变更
TICK_FLAG_VOLUME	16	成交量已变更
TICK_FLAG_BUY	32	分时报价由买入交易生成
TICK_FLAG_SELL	64	分时报价由卖出交易生成

这一设计是必要的, 因为每个分时报价数据都会填充所有字段, 无论这些数据相较前一个分时报价是否发生变化。这样, 你可以随时获取最新的价格状态, 而无需在分时报价历史中查找先前的值。例如, 某个分时报价可能仅买价发生变化, 但结构中除了新的价格外, 还会显示其他参数: 之前的 *Ask*、*Last*、成交量等。

同时, 需要注意, 根据金融工具类型的不同, 分时报价中的某些字段可能始终为零 (相应的掩码位也不会进行设置)。特别是对于外汇金融工具, 通常, *last*、*volume*、*volume_real* 字段留空。

接收 *ticks* 数组可以是固定大小,也可以是动态变化的。无论请求的时间间隔内实际有多少分时报价 (由 *CopyTicksRange* 函数的 *from/to* 参数指定), 或者 *CopyTicks* 函数的 *count* 参数指定的数量如何, 函数复制到固定大小数组中的分时报价数量不会超过数组的大小。在 *ticks* 数组中, 最早的分时报价排在最前, 最新的分时报价排在最后。

在这两个函数的参数中, 时间值均以 01.01.1970 00:00:00 至今的毫秒数表示。在 *CopyTicks* 函数中, 请求的分时报价范围由起始时间 *from* 和分时报价数量 *count* 设置, 而在 *CopyTicksRange* 函数中则由 *from* 和 *to* 设置 (包含两端时间点)。

换言之, *CopyTicksRange* 用于获取特定时间间隔内的分时报价, 其数量事先未知。 *CopyTicks* 保证返回不超过 *count* 个分时报价, 但无法预先确定这些分时报价覆盖的时间间隔。

CopyTicksRange 中 *from* 和 *to* 值的时间顺序并不重要: 无论如何情况, 函数都会从两个值中的最小值开始, 到最大值结束, 返回分时报价数据。

CopyTicks 函数将 *from* 参数视为最小时间的左边界, 并从此处开始向未来方向计数 *count* 个分时报价数据。然而, 这里有一个重要的例外: 当 *from* = 0 (默认值) 时, 会将其视为当前时刻, 分时报价将从当前时刻向过去方向计数。这使得我们始终可以获取最近的指定数量的分时报价。当 *count* = 0 (默认值) 时, 函数最多复制 2000 个分时报价。

两个函数均返回复制的分时报价数量, 如果发生错误, 则返回 -1。具体来说, *GetLastError* 可能返回以下错误代码:

- *ERR_HISTORY_TIMEOUT* - 分时报价同步超时, 函数返回已获取的所有数据。
- *ERR_HISTORY_SMALL_BUFFER* - 静态缓冲区过小, 仅返回数组能容纳的数据量。
- *ERR_NOT_ENOUGH_MEMORY* - 无法分配足够的内存来将指定范围内的分时报价历史加载到动态数组中。

flags 参数定义请求的分时报价类型。

常量	值	说明
<i>COPY_TICKS_INFO</i>	1	由以下参数变动引起的分时报价: <i>Bid</i> 和 / 或 <i>Ask</i> (<i>TICK_FLAG_BID</i> , <i>TICK_FLAG_ASK</i>)
<i>COPY_TICKS_TRADE</i>	2	包含 <i>Last</i> 和 <i>Volume</i> 变动的分时报价 (<i>TICK_FLAG_LAST</i> , <i>TICK_FLAG_VOLUME</i> , <i>TICK_FLAG_BUY</i> , <i>TICK_FLAG_SELL</i>)
<i>COPY_TICKS_ALL</i>	3	所有分时报价

对于任何请求类型, *MqlTick* 结构体中与标志不匹配的剩余字段将包含先前的实际值。例如, 如果仅请求信息类分时报价 (*COPY_TICKS_INFO*), 其余字段仍会被填充。这意味着, 如果只有 *Bid* 价格发生变更, *ask* 和 *volume* 字段将写入最后已知值。要确定分时报价中哪些内容发生了变更, 分析其 *flags* 字段 (可能为 *TICK_FLAG_BID*、*TICK_FLAG_ASK* 或两者的组合)。如果某分时报价的 *Bid* 和 *Ask* 价格均为零, 且标志表明这些价格已变更 (*flags* == *TICK_FLAG_BID* | *TICK_FLAG_ASK*), 则表示订单簿已清空。

类似地, 如果请求交易类分时报价 (*COPY_TICKS_TRADE*), *bid* 和 *ask* 字段将记录最后已知价格。此时, *flags* 字段可能包含 *TICK_FLAG_LAST*、*TICK_FLAG_VOLUME*、*TICK_FLAG_BUY*、*TICK_FLAG_SELL* 的组合。

当请求 *COPY_TICKS_ALL* 时, 将返回所有分时报价。

调用 *CopyTicks/CopyTicksRange* 函数时，系统会检查硬盘中存储的指定交易品种的分时报价库的同步状态。如果本地数据库中的分时报价不足，将自动从交易服务器下载缺失的分时报价。此时，分时报价将从查询参数中的最早日期开始同步至当前时刻。此后，该交易品种的所有传入分时报价将被写入分时报价数据库，以保持其同步更新。

分时报价数据量远大于分钟级报价数据。第一次请求分时报价历史或启动 *真实分时报价* 测试时，下载过程可能耗时较长。分时报价历史数据以内部 *TKC* 格式存储在以下目录：*{terminal_dir}/bases/{server_name}/ticks/{symbol_name}*。每个文件包含一个月的信息。

在指标中，这些函数会立即返回结果，即复制当前可用的按交易品种分类的分时报价，并在数据不足时启动分时报价库的后台同步进程。由于同一交易品种的所有指标共享一个 *线程*，因此它们无权等待同步操作完成。同步结束后，再次调用该函数将返回所有请求的分时报价。

在 EA 交易和脚本中，函数最多可等待 45 秒获取结果：与指标不同，每个 EA 交易和脚本在独立线程中运行，因此可在超时范围内等待同步完成。如果在此期间仍未同步所需数量的分时报价，将仅返回可用分时报价，并继续在后台进行同步。

请记住，实时分时报价是以事件形式向图表广播的：指标通过 *OnCalculate* 处理程序接收新分时报价的通知，而 EA 交易则通过 *OnTick* 处理程序接收。需要注意的是，系统不保证传递所有事件。如果程序正在处理当前的 *OnCalculate/OnTick* 事件时，终端收到新的分时报价，新事件可能不会被添加到这一“忙碌”程序的队列中（请参阅 *事件处理函数概述* 章节）。此外，终端可能同时收到多个分时报价，但每个 MQL 程序只会生成一个事件：当前市场状态事件。在这种情况下，你可以使用 *CopyTicks* 函数请求自上次处理事件以来收到的所有分时报价。以下是该算法的伪代码形式：

```
void processAllTicks()
{
    static ulong prev = 0;
    if(!prev)
    {
        MqlTick ticks[];
        const int n = CopyTicks(_Symbol, ticks, COPY_TICKS_ALL, prev + 1, 1000000);
        if(n > 0)
        {
            prev = ticks[n - 1].time_msc;
            ... // processing all missed ticks
        }
    }
    else
    {
        MqlTick tick;
        SymbolInfoTick(_Symbol, tick);
        prev = tick.time_msc;
        ... // processing the first tick
    }
}
```

此处使用的 *SymbolInfoTick* 函数会通过引用传递的方式，将最新的分时报价数据填充到单个 *MqlTick* 结构体中。我们将在单独的 *章节* 中对其进行研究。

需要注意，调用 *CopyTicks* 时，会在旧时间戳 *prev* 的基础上增加 1 毫秒。这确保不会重复处理前一个分时报价。但如果同 1 毫秒内有多个分时报价与 *prev* 对应，此算法会跳过它们。如果需要覆盖所有分时报价，应在更新 *prev* 变量时记录 *prev* 时间点的可用分时报价数量。在下次调用

*CopyTicks*时，从 *prev* 时刻开始查询分时报价，并在数组中跳过（忽略）相应数量的“旧”分时报价。

但请注意，并非每个 MQL 程序都需要上述算法。大多数程序不会分析每个分时报价，而与最新已知分时报价对应的当前价格状态会通过 **事件** 模型快速广播到图表，并可通过 **交易品种** 和 **图表** 特性获取。

演示这两个函数的用法，我们将通过两个示例分别说明。针对这两个示例，我们建立了一个共用头文件 *TickEnum.mqh*，其中将前文请求的分时报价标志和分时报价状态标志的常量总结为两个枚举。

```
enum COPY_TICKS
{
    ALL_TICKS = /* -1 */ COPY_TICKS_ALL,    // all ticks
    INFO_TICKS = /* 1 */ COPY_TICKS_INFO,    // info ticks
    TRADE_TICKS = /* 2 */ COPY_TICKS_TRADE,  // trade ticks
};

enum TICK_FLAGS
{
    TF_BID = /* 2 */ TICK_FLAG_BID,
    TF_ASK = /* 4 */ TICK_FLAG_ASK,
    TF_BID_ASK = TICK_FLAG_BID | TICK_FLAG_ASK,

    TF_LAST = /* 8 */ TICK_FLAG_LAST,
    TF_BID_LAST = TICK_FLAG_BID | TICK_FLAG_LAST,
    TF_ASK_LAST = TICK_FLAG_ASK | TICK_FLAG_LAST,
    TF_BID_ASK_LAST = TF_BID_ASK | TICK_FLAG_LAST,

    TF_VOLUME = /* 16 */ TICK_FLAG_VOLUME,
    TF_LAST_VOLUME = TICK_FLAG_LAST | TICK_FLAG_VOLUME,
    TF_BID_VOLUME = TICK_FLAG_BID | TICK_FLAG_VOLUME,
    TF_BID_ASK_VOLUME = TF_BID_ASK | TICK_FLAG_VOLUME,
    TF_BID_ASK_LAST_VOLUME = TF_BID_ASK | TF_LAST_VOLUME,

    TF_BUY = /* 32 */ TICK_FLAG_BUY,
    TF_SELL = /* 64 */ TICK_FLAG_SELL,
    TF_BUY_SELL = TICK_FLAG_BUY | TICK_FLAG_SELL,
    TF_LAST_VOLUME_BUY = TF_LAST_VOLUME | TICK_FLAG_BUY,
    TF_LAST_VOLUME_SELL = TF_LAST_VOLUME | TICK_FLAG_SELL,
    TF_LAST_VOLUME_BUY_SELL = TF_BUY_SELL | TF_LAST_VOLUME,
    ...
};
```

使用枚举可以让源代码中的类型检查更加严格，也能通过 *EnumToString* 更轻松地将值的含义以字符串形式显示出来。此外，TICK_FLAGS 枚举中添加了最常用的标志组合，以优化分时报价的可视化或过滤。由于会发生命名冲突，所以无法为枚举元素设置与内置常量相同的名称。

第一个脚本 *SeriesTicksStats.mq5* 利用 *CopyTicks* 函数统计在给定历史深度内，带不同标志的分时报价数量。

在输入参数里，你可以通过 COPY_TICKS 来设置交易品种（默认为图表交易品种）、分析的分时报价数量以及请求模式。

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input int TickCount = 10000;
input COPY_TICKS TickType = ALL_TICKS;
```

分时报价特性中每个标志（位掩码中的每个位）的出现统计信息会收集到 *TickFlagStats* 结构中。

```
struct TickFlagStats
{
    TICK_FLAGS flag; // mask with bit (one or more)
    int count;       // number of ticks with this bit in the flags field
    string legend;   // bit description
};
```

OnStart 函数定义了一个包含 8 个元素的 *TickFlagStats* 结构数组：其中 6 个（从 1 到 6，包含两端）用于对应的 *TICK_FLAG* 位，另外两个用于位组合（见下文）。通过一个简单的循环，数组中会填充各个标准位/标志的元素；循环结束后，会填充两个组合掩码（在第 0 个元素中，会统计 *Bid* 和 *Ask* 同时发生变化的分时报价；在第 7 个元素中，会统计同时有 *Buy* 和 *Sell* 交易的分时报价）。

```
void OnStart()
{
    TickFlagStats stats[8] = {};
    for(int k = 1; k < 7; ++k)
    {
        stats[k].flag = (TICK_FLAGS)(1 << k);
        stats[k].legend = EnumToString(stats[k].flag);
    }
    stats[0].flag = TF_BID_ASK; // combination of BID AND ASK
    stats[7].flag = TF_BUY_SELL; // combination of BUY AND SELL
    stats[0].legend = "TF_BID_ASK (COMBO)";
    stats[7].legend = "TF_BUY_SELL (COMBO)";
    ...
}
```

我们会将所有主要工作委托给辅助函数 *CalcTickStats*，并向其传递输入参数和预准备好的用于收集统计数据的数组。之后，只需在日志中显示统计数字即可。

```
const int count = CalcTickStats(TickType, 0, TickCount, stats);
PrintFormat("%s stats requested: %d (got: %d) on %s",
    EnumToString(TickType),
    TickCount, count, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol);
ArrayPrint(stats);
}
```

CalcTickStats 函数本身十分有趣：

```

int CalcTickStats(const string symbol, const COPY_TICKS type,
    const datetime start, const int count,
    TickFlagStats &stats[])
{
    MqlTick ticks[];
    ResetLastError();
    const int nf = ArraySize(stats);
    const int nt = CopyTicks(symbol, ticks, type, start * 1000, count);
    if(nt > -1 && _LastError == 0)
    {
        PrintFormat("Ticks range: %s'%03d - %s'%03d",
            TimeToString(ticks[0].time, TIME_DATE | TIME_SECONDS),
            ticks[0].time_msc % 1000,
            TimeToString(ticks[nt - 1].time, TIME_DATE | TIME_SECONDS),
            ticks[nt - 1].time_msc % 1000);

        // loop through ticks
        for(int j = 0; j < nt; ++j)
        {
            // loop through TICK_FLAGS (2 4 8 16 32 64) and combinations
            for(int k = 0; k < nf; ++k)
            {
                if((ticks[j].flags & stats[k].flag) == stats[k].flag)
                {
                    stats[k].count++;
                }
            }
        }
    }
    return nt;
}

```

它使用 *CopyTicks*, 从 *start* 日期开始, 按 *count* 数量请求特定 *type* 的指定 *symbol* 的分时报价。*start* 参数类型为 *datetime*, 传递给 *CopyTicks* 时需转换为毫秒。如果 *start = 0* (如 *OnStart* 函数中所示), 系统将从当前时间开始返回最近的分时报价。因此, 每次调用脚本时, 由于新分时报价的传入, 统计结果很可能会更新。唯一可能的例外是在周末发出的请求, 或针对低流动性金融工具的请求。

如果 *CopyTicks* 执行时没有出错, 我们的代码会记录所获取的分时报价覆盖的时间范围。

最后, 在循环中遍历所有分时报价, 将每个分时报价标志与预定义的统计结构体数组 *TickFlagStats* 中的元素掩码进行位运算匹配, 并统计匹配的数量。

建议在包含真实成交量和交易信息的金融工具上运行此脚本, 以便测试 *COPY_TICKS* 枚举中的所有模式 (请记住, 它们对应于 *CopyTicks* 中 *flags* 参数的常量: *COPY_TICKS_INFO*、*COPY_TICKS_TRADE* 和 *COPY_TICKS_ALL*)。

以下是请求 100000 个全类型 (*TickType = ALL_TICKS*) 分时报价统计时的日志条目示例:

```

Ticks range: 2021.10.11 07:39:53'278 - 2021.10.13 11:51:29'428
ALL_TICKS stats requested: 100000 (got: 100000) on YNDX.MM
  [flag] [count]           [legend]
[0]      6   11323 "TF_BID_ASK (COMBO)"
[1]      2   26700 "TF_BID"
[2]      4   33541 "TF_ASK"
[3]      8   51082 "TF_LAST"
[4]     16   51082 "TF_VOLUME"
[5]     32   25654 "TF_BUY"
[6]     64   28802 "TF_SELL"
[7]     96   3374  "TF_BUY_SELL (COMBO)"

```

若仅请求信息类分时报价 (*TickType = INFO_TICKS*), 结果如下:

```

Ticks range: 2021.10.07 07:08:24'692 - 2021.10.13 11:54:01'297
INFO_TICKS stats requested: 100000 (got: 100000) on YNDX.MM
  [flag] [count]           [legend]
[0]      6   23115 "TF_BID_ASK (COMBO)"
[1]      2   60860 "TF_BID"
[2]      4   62255 "TF_ASK"
[3]      8      0 "TF_LAST"
[4]     16      0 "TF_VOLUME"
[5]     32      0 "TF_BUY"
[6]     64      0 "TF_SELL"
[7]     96      0 "TF_BUY_SELL (COMBO)"

```

此处可以验证计算的准确性: TF_BID 和 TF_ASK 的数量之和, 减去 TF_BID_ASK (COMBO) 的匹配数, 恰好为 100000 (分时报价总数)。正如预期一样, 带成交量和 *Last* 价格的分时报价未包含在结果中。

现在, 我们再次运行脚本, 仅针对交易类分时报价 (*TickType = TRADE_TICKS*)。

```

Ticks range: 2021.10.06 20:43:40'024 - 2021.10.13 11:52:40'044
TRADE_TICKS stats requested: 100000 (got: 100000) on YNDX.MM
  [flag] [count]           [legend]
[0]      6      0 "TF_BID_ASK (COMBO)"
[1]      2      0 "TF_BID"
[2]      4      0 "TF_ASK"
[3]      8 100000 "TF_LAST"
[4]     16 100000 "TF_VOLUME"
[5]     32   51674 "TF_BUY"
[6]     64   55634 "TF_SELL"
[7]     96   7308  "TF_BUY_SELL (COMBO)"

```

所有分时报价均包含 TF_LAST 和 TF_VOLUME 标志, 且买卖方向混合的情况出现了 7308 次。同样, TF_BUY 和 TF_SELL 数量之和, 减去它们的组合数, 与分时报价总数一致。

第二个脚本 *SeriesTicksDeltaVolume.mq5* 使用 *CopyTicksRange* 函数计算每个柱线的成交量差值。我们知道, MetaTrader 5 的报价仅包含中性成交量 (每根柱线的买卖成交量合并为一个值)。然而, 借助真实分时报价历史, 我们可以分别计算买入和卖出成交量的总和, 以及它们之间的差值。这些特性是制定交易决策的重要辅助因素。

输入参数的设置与第一个脚本类似，特别是分析的交易品种和分时报价请求模式。不过，在这里，你需要额外指定一个时间范围，因为成交量差值需要逐柱线计算。默认使用当前图表的时间范围。**BarCount**参数用于指定计算的柱线数量。

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input ENUM_TIMEFRAMES TimeFrame = PERIOD_CURRENT;
input int BarCount = 100;
input COPY_TICKS TickType = INFO_TICKS;
```

每根柱线的统计数据存储在 **DeltaVolumePerBar** 结构体中。

```
struct DeltaVolumePerBar
{
    datetime time; // bar time
    ulong buy;     // net volume of buy operations
    ulong sell;    // net sell operations
    long delta;    // volume difference
};
```

OnStart函数定义了一个此类结构体的数组，其大小根据指定的柱线数量分配。

```
void OnStart()
{
    DeltaVolumePerBar deltas[];
    ArrayResize(deltas, BarCount);
    ZeroMemory(deltas);
    ...
}
```

主要算法如下。

```
for(int i = 0; i < BarCount; ++i)
{
    MqlTick ticks[];
    const datetime next = iTime(WorkSymbol, TimeFrame, i);
    const datetime prev = iTime(WorkSymbol, TimeFrame, i + 1);
    ResetLastError();
    const int n = CopyTicksRange(WorkSymbol, ticks, COPY_TICKS_ALL,
        prev * 1000, next * 1000 - 1);
    if(n > -1 && _LastError == 0)
    {
        ...
    }
}
```

在遍历柱线的循环中，我们获取每根柱线的时间范围：**prev**和**next**（第 0 根柱线不完整，因此不进行处理）。为该时间间隔调用 **CopyTicksRange**时，请记住，将**datetime** 转换为毫秒，并从右边界中减去 1 毫秒，因该时间属于下一根柱线。如果没有错误，我们将在循环中处理所获取的分时报价数组。

```

deltas[i].time = prev; // remember the bar time
for(int j = 0; j < n; ++j)
{
    // when real volumes can be available, take them from ticks
    if(TickType == TRADE_TICKS)
    {
        // separately accumulate volumes for buy and sell deals
        if((ticks[j].flags & TICK_FLAG_BUY) != 0)
        {
            deltas[i].buy += ticks[j].volume;
        }
        if((ticks[j].flags & TICK_FLAG_SELL) != 0)
        {
            deltas[i].sell += ticks[j].volume;
        }
    }
    // when there are no real volumes, we evaluate them by the price movement
    else
    if(TickType == INFO_TICKS && j > 0)
    {
        if((ticks[j].flags & (TICK_FLAG_ASK | TICK_FLAG_BID)) != 0)
        {
            const long d = (long)((((ticks[j].ask + ticks[j].bid)
                - (ticks[j - 1].ask + ticks[j - 1].bid)) / _Point);
            if(d > 0) deltas[i].buy += d;
            else deltas[i].sell += -d;
        }
    }
}
deltas[i].delta = (long)(deltas[i].buy - deltas[i].sell);

```

如果脚本设置中请求了按交易类分时报价进行分析 (TRADE_TICKS), 检查是否存在 TICK_FLAG_BUY 和 TICK_FLAG_SELL 标志, 如果至少设置了其中一个标志, 则将 *DeltaVolumePerBar* 结构体对应变量中的 **volume** 字段纳入统计。此模式仅适用于股票类金融工具。对于外汇金融工具, 不会填充成交量和交易方向标志, 因此需要采用不同的处理方法。

如果在设置中指定了适用于所有金融工具的信息类分时报价 (INFO_TICKS), 则算法基于以下经验规则: 如你所知, 买入推动价格上涨, 卖出推动价格下跌。因此, 我们可以假设, 如果新分时报价数据的平均价格 *Ask+Bid* 相对于上一个分时报价数据上涨, 则认为发生了买入操作; 如果价格下跌, 则认为发生了卖出操作。交易量可以通过所经过的点数 (*_Point* 粗略估算)。

计算结果会以结构数组的形式简洁呈现, 其中汇总了统计信息。

```

PrintFormat("Delta volumes per intraday bar\nProcessed %d bars on %s %s %s",
    BarCount, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol,
    EnumToString(TimeFrame == PERIOD_CURRENT ? _Period : TimeFrame),
    EnumToString(TickType));
ArrayPrint(deltas);
}

```

以下是针对 TRADE_TICKS 和 INFO_TICKS 模式的一些日志。

```
Delta volumes per intraday bar
Processed 100 bars on YNDX.MM PERIOD_H1 TRADE_TICKS
      [time] [buy] [sell] [delta]
[ 0] 2021.10.13 11:00:00  7912  14169  -6257
[ 1] 2021.10.13 10:00:00  8470  11467  -2997
[ 2] 2021.10.13 09:00:00 10830  13047  -2217
[ 3] 2021.10.13 08:00:00 23682  19478   4204
[ 4] 2021.10.13 07:00:00 14538  11600   2938
[ 5] 2021.10.12 20:00:00  2132   4786  -2654
[ 6] 2021.10.12 19:00:00  9173  13775  -4602
[ 7] 2021.10.12 18:00:00  1297   1719   -422
[ 8] 2021.10.12 17:00:00  3803   2995    808
[ 9] 2021.10.12 16:00:00  6743   7045   -302
[10] 2021.10.12 15:00:00 17286  37286 -20000
[11] 2021.10.12 14:00:00 33263  54157 -20894
[12] 2021.10.12 13:00:00 56060  52659   3401
[13] 2021.10.12 12:00:00 12832  10489   2343
[14] 2021.10.12 11:00:00  7530   6092   1438
[15] 2021.10.12 10:00:00  6268  25201 -18933
...
```

当然，这些数值存在显著差异，但重点不在于绝对值：在缺乏交易所成交量的情况下，即使是这种对拆分和差值动态的模拟，也能让我们从不同角度观察市场行为。

```
Delta volumes per intraday bar
Processed 100 bars on YNDX.MM PERIOD_H1 INFO_TICKS
      [time] [buy] [sell] [delta]
[ 0] 2021.10.13 11:00:00  1939   2548   -609
[ 1] 2021.10.13 10:00:00  2222   2400   -178
[ 2] 2021.10.13 09:00:00  2903   2909     -6
[ 3] 2021.10.13 08:00:00  4489   4060    429
[ 4] 2021.10.13 07:00:00  4999   4285    714
[ 5] 2021.10.12 20:00:00  1444   1556   -112
[ 6] 2021.10.12 19:00:00  5464   5867   -403
[ 7] 2021.10.12 18:00:00  2522   2653   -131
[ 8] 2021.10.12 17:00:00  2111   2017     94
[ 9] 2021.10.12 16:00:00  4617   6096  -1479
[10] 2021.10.12 15:00:00  5716   5411    305
[11] 2021.10.12 14:00:00 10044  10866   -822
[12] 2021.10.12 13:00:00 10893  11178   -285
[13] 2021.10.12 12:00:00  2822   2783     39
[14] 2021.10.12 11:00:00  2070   1936    134
[15] 2021.10.12 10:00:00  2053   2303   -250
...
```

当我们学习如何 [创建指标](#)时，可以将这一算法嵌入其中一个指标中（请参阅 [等待数据和管理可见性](#)章节中的 *IndDeltaVolume.mq5*，从而直接在图表上可视化展示增量数据。

5.4 创建自定义指标

指标是最常见的 MQL 程序之一。它们是简单而强大的技术分析工具。其主要使用机制是通过公式处理原始价格数据，从而创建派生时间序列。这样就可以对市场过程的特定特征进行评估和可视化。任何时间序列，包括通过指标计算获得的序列，都可以作为输入传递给另一个指标，依此类推。许多众所周知的指标的公式（例如 [MACD](#)）实际上包含对多个相关指标的调用。

终端用户无疑熟悉众多内置指标，同时也知道可以使用 MQL5 语言扩展可用指标列表。从用户角度来看，MetaTrader 5 平台内置的指标与使用 MQL5 语言编写的自定义指标在操作方式上完全相同。

通常，指标会以线条、直方图或其他图形形式在价格图表窗口中显示其计算结果。每个此类图表都是基于计算出的时间序列进行可视化的，这些时间序列存储在指标内部称为指标缓冲区的特殊数组中：可在终端的 **Data Window** 中与 OHLC 价格一同查看。然而，除缓冲区功能外，指标还可以提供额外功能，甚至可能完全不使用缓冲区。例如，指标经常用于解决需要创建 [图形对象](#) 管理图表及其 [特性](#) 以及与用户交互的问题（请参阅 [OnChartEvent](#) 函数）。

在本章中，我们将学习在 MQL5 中创建指标的基本原理。这类指标通常被称为“自定义指标”，因为用户可以从零开始编写，也可以基于现有的源代码进行编译。在下一章中，我们将探讨通过编程方式管理自定义指标和内置指标的相关问题，这将帮助我们构建更复杂的指标，并为基于指标的交易信号以及 EA 交易的过滤奠定基础。

再后面的内容中，我们将掌握以 [资源](#) 形式将指标集成到可执行 MQL 程序中的技术。

5.4.1 指标的主要特性

指标实现了一种特定的计算算法，该算法通过柱线应用于给定的初始时间序列或多个时间序列。所有此类时间序列均为 *terminal's own* 数组（请参阅函数 [ArraysSeries](#)）：终端为其分配内存，并在新柱形形成时添加新元素。自然地，不同时间范围下的交易品种报价数组具有基础性作用，因为这些数组由终端自动填充。但已启动的指标可显著扩展可供分析的时间序列集合。

指标通常将其运算结果存储在动态数组中，这些数组通过专用函数（[SetIndexBuffer](#)）注册为指标缓冲区，并同样成为终端的自有数组。除了为这些数组分配内存外，终端还将其作为新的时间序列提供公共访问权限，其他指标可基于这些时间序列进行计算。

指标计算部分的入口点是 [OnCalculate](#) 函数，一个同名的事件处理程序。在 [事件处理函数概述](#) 中，我们已提及该函数：只要源代码中包含该函数，就足以让终端将该 MQL 程序识别为指标。[OnCalculate](#) 函数将在 [下一节](#) 中详细介绍。特别是，[OnCalculate](#) 函数的主要特点是存在两种不同的形式。程序员应在指标设计的最开始就选择该选项，因为这决定了指标的用途和可能的应用场景。

[OnCalculate](#) 函数并非指标的唯一区别性特性。除此之外，一组特殊的预处理器指令 [#property](#) 专门用于指标，我们将在本章的几个相关章节中逐步介绍它们。此前我们已经了解了一些 [通用程序特性](#)，这些指令当然也适用于指标。

正如 MetaTrader 5 用户所知，每个指标都以自己的方式来显示其图形结构（时间序列）：可以显示在展示交易品种价格的主窗口中，也可以显示在单独的子窗口中。当某个指标（或一组指标）被添加到图表时，如果该指标设计为在子窗口中运行，则会在窗口下方创建一个子窗口。例如，标准移动平均线（MA）指标绘制在价格图表上，而威廉指标（WPR）则绘制在单独的子窗口中。

从开发者的角度来看，这意味着，需要在最初阶段确定指标是显示在主窗口还是子窗口中，因为这两种模式无法同时使用。此外，这一特性以及指标缓冲区数量只能通过 [#property](#) 指令一次

性设置 (请参见 [两种类型的指标](#) 和 [设置缓冲区和绘图的数量](#)), 之后无法通过 MQL5 API 函数调用进行修改, 因为根本不存在此类函数。与这些不可变特性不同, 大多数其他指标特性可以通过特殊函数进行动态调整。因此, 在研究指标编程的技术细节时, 我们将能够建立 `#property` 特性与 MQL5 函数之间的对应关系。

此外, 指标通常还会实现 `OnInit` 和 `OnDeinit` 处理程序 (请参阅 [指标和 EA 交易的参考事件](#))。 `OnInit` 函数尤为重要, 它用于分配将作为指标缓冲区的数组, 这些缓冲区用于存储中间及最终计算结果, 既可供用户查看, 也能被其他程序 (如 EA 交易) 调用。

指标是交互式 MQL 程序的一种, 必要时, 它可以处理计时器事件 (`OnTimer`) 以及用户或其他程序产生的图表变化 (`OnChartEvent`)。这些技术特征对于指标而言是可选的, 并且基于 [图表事件队列](#)。我们将在关于 [图表](#) 一章中单独讨论它们。

5.4.2 主要指标事件: OnCalculate

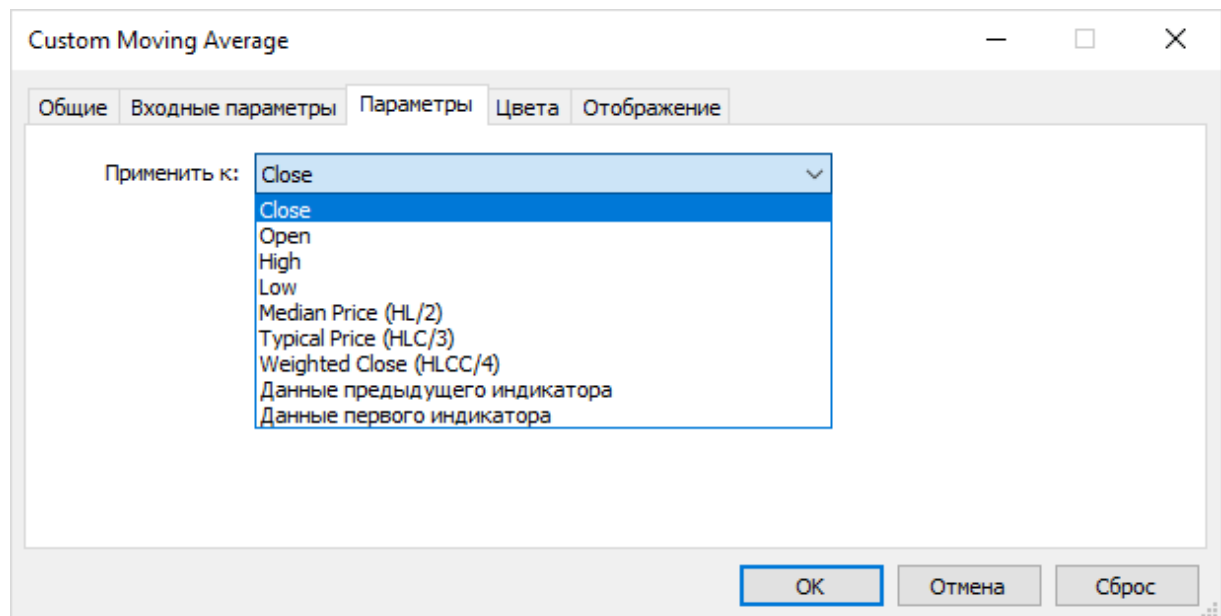
`OnCalculate` 函数是 MQL5 指标代码的主要入口点。每当发生 `OnCalculate` 事件时, 就会调用该函数, 而当价格数据发生变化时, 将触发该事件。例如, 当某个交易品种的新分时报价到达时, 或旧价格发生变动时 (填补历史数据缺口或从服务器下载缺失数据时), 都会触发该事件。

该函数有两种变体, 区别在于计算所依据的源数据不同:

- ① 完整版 - 通过参数提供一组标准价格时间序列 (OHLC 价、成交量、点差)
- ② 简化版 - 针对单一任意时间序列 (不一定是标准序列)

一个指标只能选择两种形式中的一种使用, 无法在同一指标中混用这两种形式。

使用简化版 `OnCalculate` 函数时, 将指标添加到图表上后, 其特性对话框中会多出一个选项卡。该选项卡提供了一个 `Apply to` 下拉列表, 你需要在此选择指标计算所基于的初始时间序列。如果未选择任何时间序列, 则默认以 `Close` 价格值作为计算依据。



使用简化版 `OnCalculate` 为指标选择初始时间序列

下拉列表始终提供标准价格类型, 但如果图表上已加载其他指标, 该设置允许你选择其中一个指标作为另一个指标的数据源, 从而构建指标处理链。我们将在 [跳过初始柱线的绘制](#) 章节尝试基于其他指标构建新指标。使用完整版 `OnCalculate` 时, 该选项不可用。

禁止将指标应用于以下内置指标：Fractals、Gator、Ichimoku 和 Parabolic SAR。

简化版 *OnCalculate* 具有以下原型：

```
int OnCalculate(const int rates_total, const int prev_calculated, const int begin,
const double &data[])
```

data 数组包含计算所需的初始数据。该数据可以是价格时间序列之一，或另一个指标的计算缓冲区。*rates_total* 参数指定 *data* 数组的大小。*ArraySize(data)* 或 *iBars(NULL, 0)* 调用应提供与 *rates_total* 相同的值。

prev_calculated 旨在让指标仅对少量新增柱线（通常是最后一根）进行有效重新计算，而非重新计算所有柱线。*prev_calculated* 值等于上一次调用 *OnCalculate* 函数时返回给运行时系统的结果。例如，当收到下一个分时报价时，如果指标已计算所有柱线的公式，则应从 *OnCalculate* 函数返回 *rates_total_A* 值（此处索引 *A* 表示初始时刻）。然后，在收到下一个分时报价并触发 *OnCalculate* 事件时，终端会将 *prev_calculated* 设置为先前返回的值 *rates_total_A*。然而，在此期间柱线数量可能已经发生变化，新的 *rates_total* 值会增大，我们将该值设为 *rates_total_B*。因此，只需计算从 *prev_calculated*（即 *rates_total_A*）到 *rates_total_B* 之间的柱线数据。

不过，最常见的情况新分时报价适合当前零柱线，即 *rates_total* 不会变化。因此在大多数 *OnCalculate* 调用中，会出现 *prev_calculated == rates_total* 等式。这种情况下是否需要重新计算？这取决于计算的性质。例如，如果指标是基于开盘价计算的（开盘价固定不变），则无需重新计算。但如果指标使用收盘价（实际上是最后一个已知分时报价的价格）或任何其他依赖于 *Close* 的汇总价格，则应始终重新计算最后一根柱线。

当第一次调用 *OnCalculate* 函数时，*prev_calculated* 的值为 0。

如果自上次调用 *OnCalculate* 函数以来，价格数据发生了变化（例如加载了更深层次的历史数据或填补了缺口），则终端也会将 *prev_calculated* 参数的值设为 0。因此，指标将收到信号，需要对整个可用历史数据进行完整重新计算。

如果 *OnCalculate* 函数返回空值，则不会绘制指标，且其缓冲区的名称和值将在 *Data window* 中隐藏。

请注意，返回全部柱线数量 *rates_total*，是向终端以及将使用该指标的其他 MQL 程序表明指标数据已准备就绪的唯一标准方式。即使指标仅用于计算和显示有限数据量，仍应返回 *rates_total*。

data 数组的索引方向可以通过调用 *ArraySetAsSeries* 函数选择（默认值为 *false*，可通过调用 *ArrayGetAsSeries* 验证）。同时，如果对该数组应用 *ArrayIsSeries* 函数，将返回 *true*。这意味着该数组是由终端管理的内部数组。指标无法以任何方式对它进行修改，只能读取，因为参数声明中有 *const* 修饰符。

begin 参数报告 *data* 数组中应从计算中排除的初始值数量。当用户将指标配置为接收其他指标的 *data* 时（见上图），系统会自动设置该参数。例如，如果所选数据源指标计算周期为 *N* 的移动平均线，根据定义，前 *N - 1* 根柱线不包含有效数据，因为无法对不足 *N* 根柱线进行平均计算。如果开发者在源指标中设置了特殊特性，该特性将通过 *begin* 参数正确传递给当前指标。我们很快将在实践中验证这一点（请参阅 [跳过初始柱线的绘制](#) 章节）。

我们来尝试创建一个简化版 *OnCalculate* 的空指标模板。该指标目前不执行任何计算，但可作为后续实验的准备。原始文件 *IndStub.mq5* 可在 *MQL5/Indicators/MQL5Book/p5/* 文件夹中找到。为确认指标正常运行，我们在 *OnCalculate* 中添加以下功能：能够将 *prev_calculated* 和 *rates_total* 值输出到日志，能够统计函数调用次数。


```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &data[])
{
    static int count = 0;
    ++count;
    // compare the number of bars on the previous call and the current one
    if(prev_calculated != rates_total)
    {
        // signal only if there is a difference
        PrintFormat("calculated=%d rates=%d; %d ticks",
                   prev_calculated, rates_total, count);
    }
    return rates_total; // return the number of processed bars
}

```

通过判断 *prev_calculated* 和 *rates_total* 是否不相等，可以确保仅在以下情况输出消息：当指标第一次添加到图表时输出消息，以及当新柱线形成时输出消息。在当前柱线形成过程中到达的所有分时报价不会改变柱线数量，因此 *prev_calculated* 和 *rates_total* 将保持相等。但我们仍会使用 *count* 变量统计所有分时报价的总数。

其余参数目前尚未启用，但我们将逐步使用全部功能。

这段源代码能够成功编译，但会生成两条警告。

```

no indicator window property is defined, indicator_chart_window is applied
no indicator plot defined for indicator

```

它们提示缺少某些 *#property* 指令，这些指令虽然并非强制要求，但用于设置指标的基本特性。具体来说，第一条警告表明尚未为指标选择绑定方式：主窗口还是子窗口，因此默认将使用主图表窗口。第二条警告与我们尚未设置显示的图表数量有关。如前所述，虽然某些指标特意设计为不带缓冲区，因为它们旨在执行其他操作，但当前警告提醒我们稍后添加可视化部分。

我们将在接下来的章节中解决这些警告问题，但现在，我们将在 EURUSD,M1 图表上启动该指标。我们使用 M1 时间范围，因为这样可以快速观察到新柱线的形成以及日志中消息的更新。

```

calculated=0 rates=10002; 1 ticks
calculated=10002 rates=10003; 30 ticks
calculated=10003 rates=10004; 90 ticks
calculated=10004 rates=10005; 167 ticks
calculated=10005 rates=10006; 240 ticks

```

因此，我们看到 *OnCalculate* 处理程序按预期调用，你可以在其中基于每个分时报价或基于每个柱线执行计算。如需从图表移除指标，可通过图表上下文菜单调用 *Indicator List* 对话框：选择目标指标，然后按 *Delete*。

现在，我们回到 *OnCalculate* 函数的另一种原型。我们已经在实践中试用了简化版，同样也可以为完整版创建完全相同的空白模板。

完整版用于基于标准价格时间序列进行计算，其原型如下：

```
int OnCalculate(const int rates_total, const int prev_calculated, const datetime &time[],
               const double &open[], const double &high[], const double &low[], const double &close[],
               const long &tick_volume[], const long &volume[], const int &spread[])
```

*rates_total*和 *prev_calculated* 参数的含义与简化版 *OnCalculate* 相同：*rates_total*设置传递的时间序列大小（所有数组长度相同，因为它对应图表的柱线总数），*prev_calculated* 包含上次调用时已处理的柱线数量（即 *OnCalculate* 函数之前通过 `return` 语句返回给终端的值）。

数组 *open*、*high*、*low* 和 *close* 包含当前图表柱线的相关价格：即操作中的交易品种和时间范围的价格时间序列。*time*数组包含每根柱线的开盘时间，*tick_volume* 和 *volume* 包含每根柱线的交易成交量（分时报价和交易所成交量）。

在上一章中，我们学习了终端通过一组函数为 MQL 程序提供的标准价格和成交量类型的 **时间序列**。为便于指标计算，这些时间序列将以数组引用的形式直接传递给 *OnCalculate* 处理程序。这样便无需调用这些函数并将报价复制（重复）到内部数组。当然，这种技术仅适用于基于特定条件进行计算的指标：即基于与当前图表匹配的单个交易品种和时间范围的组合。但 MQL5 允许创建多货币、多时间范围指标，以及适用于非当前图表交易品种和时间范围的指标。在所有这些情况下，都必须借助时间序列访问函数来实现。具体实现方法，我们稍后将详细探讨。

如果使用 *ArrayIsSeries* 检查所有传入的数组是否属于终端，该函数将返回 *true*。所有数组均为只读。参数说明中的 *const* 修饰符也强调了这一特性。

基于计算算法所需的数据，选择完整版或简化版。例如，要使用移动平均算法对数组进行平滑处理，仅需一个输入数组，因此，该指标可以构建为支持用户选择的任意价格类型。但诸如 *ParabolicSAR* 或 *ZigZag* 等经典指标需要使用 *High* 和 *Low* 价，因此必须使用完整版 *OnCalculate*。在接下来的章节中，我们将看到基于 *OnCalculate* 简化版和完整版的指标示例。

5.4.3 两种指标类型：主窗口指标与子窗口指标

如你所知，MetaTrader 5 中的指标可以在两个位置显示其线条：在主图表窗口的报价数据上方，或在价格图表下方创建的单独窗口中。这两种模式互斥：每个指标只能用于主窗口或子窗口，无法同时兼用两种模式。

当程序需要在两个窗口中同时可视化数据时，有几种替代解决方案。例如，项目可以通过两个相互交互的指标来实现（交互的技术实现方式不受限制，包括：[资源](#)、[文件](#)、[DBMS](#)或通过 DLL 访问的共享内存）。另一种方法是在其中一个窗口（例如底部面板）中使用指标缓冲区，并通过 **图形对象**在主图表上执行可视化。

主窗口和子窗口中均可应用多个指标。如果指标设计为在独立窗口中运行，则从导航器中用鼠标将其拖到主窗口时，系统会自动为该指标创建一个新窗口。然而，如果该窗口已经有一个包含其他指标的子窗口，则可以将新指标拖到同一位置，从而使两个或多个指标在同一个子窗口中对齐。在这种情况下，同一个窗口中可存在多种指标缩放模式。默认情况下，每个指标的图形会自动独立缩放，并自适应填充面板的整个高度，但这一行为可以进行修改（请参阅 [键盘事件](#)章节中的示例 *SubScaler.mq5*）。

指标的显示窗口通过以下两个编译指令之一进行选择。

```
#property indicator_chart_window // display the indicator in the chart window
#property indicator_separate_window // display the indicator in a separate window
```

指标开发者应在源代码起始位置插入其中一个指令。如果两个指令均未出现，系统将默认输出至主窗口，但编译器会生成警告提示。我们在上一节中已经了解了这一点。在以下示例中，我们一定会注明 *#property indicator_chart_window*或 *#property indicator_separate_window*。

第二个编译警告 *IndStub.mq5* 与缺少缓冲区和图表设置有关。我们将在下一节中讨论这些问题。

指标设置中 *Apply to* 下拉列表的操作取决于该指标设计的目标窗口。

为独立窗口设计的指标可以 *Applied* 子窗口中的指标，但不能应用于主窗口中的指标。

但为主窗口设计的指标可以 *Applied to* 任何指标，包括主窗口中的指标和子窗口中的指标。

5.4.4 设置缓冲区和绘图的数量

指标要在图表上显示计算结果，必须定义一个或多个数组并将其声明为指标缓冲区。缓冲区数量通过以下指令设置：

```
#property indicator_buffers N
```

这里 *N* 是 1 到 512 之间的整数。该指令设置代码中可用于计算指标的缓冲区数量。

N 必须是整数常量（字面量）或等效宏定义。由于这是预处理指令，在源代码预处理阶段尚不存在变量（即使带有 *const* 修饰符）。

然而，仅靠缓冲区不足以将计算数据可视化。在 MQL5 中，可视化系统采用两级结构。第一级由指标缓冲区构成，也就是动态数组，用于存储待显示的数据。第二级负责管理这些数据的显示方式。它基于名为图形构造（或图表、绘图）的新实体构建。关键在于，不同的数据显示方式可能需要不同数量的指标缓冲区。例如，移动平均线每根柱线仅需一个值，因此单个指标缓冲区即可满足此类折线图需求。然而，要显示蜡烛图，则每根柱线需要 4 个值（开盘价、最高价、最低价、收盘价）。因此，一个这样的图形绘图需要 4 个指标缓冲区。

图表数量 (*P*) 也必须在源代码中使用特殊指令定义。

```
#property indicator_plots P
```

在最简单的情况下，缓冲区和图表的数量相同。但我们很快会分析需要缓冲区数量多于图形构造的示例。除了特定类型的图形构造需要预定数量的缓冲区的情况外，我们有时还需要为中间计算分配一个或多个数组。此类数组不直接参与渲染，但包含用于构建渲染缓冲区的数据。当然，你可以将简单动态数组用于此类目的，而无需将其声明为缓冲区。但我们必须自行控制和调整它们的大小。将它们设置为缓冲区，从而指示终端分配内存，这样会更加便捷。

缓冲区和图形绘图的数量只能通过预处理指令设置；无法使用 MQL5 函数动态更改这些特性。

在确定缓冲区和图表的数量后，应在源代码中明确定义将作为指标缓冲区的数组。

我们来开始开发一个新的指标示例 *IndReplica1.mq5*，以演示源代码中的必要组成部分。该指标的本质很简单：我们将在其唯一的缓冲区中显示接收到的 *data* 参数数组的值。如前所述，用户在将指标应用于图表时选择要传递到 *data* 数组的特定时间序列；默认情况下将提供收盘价时间序列。

让我们添加声明一个缓冲区和一个图表的指令。

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
```

这些指令并不分配缓冲区本身，而只是设置指标的特性，并让运行时系统为程序进一步确定和配置指定数量的数组做好准备。接下来，我们将演示如何将数组注册为缓冲区。

5.4.5 将数组分配为缓冲区：SetIndexBuffer

在程序从启动到终止的整个运行周期内，任何 *double*类型 *动态数组* 均可充当指标缓冲区。定义这类数组的最常见方式是在全局层面声明。但在某些情况下，将数组设置为类成员，然后创建包含这些数组的全局对象会更加便捷。我们将在实现多货币对指标 (请参阅 *多货币和多时间范围指标* 章节的 *IndUnityPercent.mq5* 示例) 和增量成交量指标 (请参阅 *等待数据和管理可见性* 章节的 *IndDeltaVolume.mq5* 示例) 时探讨这种实现方式的示例。

因此，让我们在全局层面声明一个动态数组 *buffer* (无需指定大小)。

```
double buffer[];
```

可以通过终端中的特殊函数 *SetIndexBuffer* 将其注册为缓冲区。通常，该函数会与其他许多用于配置指标的函数一样，在 *OnInit* 处理程序中调用，我们将在后续章节中详细讨论。

```
bool SetIndexBuffer(int index, double buffer[],  
ENUM_INDEXBUFFER_TYPE mode = INDICATOR_DATA)
```

该函数在 *index* 指定的指标缓冲区与 *buffer* 动态数组之间建立关联。*index* 值必须介于 0 和 N-1 之间，其中 N 是由 *#property indicator_buffers* 指令指定的缓冲区数量。

绑定操作完成后，数组尚未准备好处理数据，甚至不会改变其大小，因此初始化和所有计算都应在 *OnCalculate* 函数中执行。将动态数组分配为指标缓冲区后，不可再更改其大小。对于指标缓冲区，所有大小调整操作均由终端本身自动完成。

将数组与指标缓冲区关联后，索引方向默认与普通数组一致。必要时，可使用 *ArraySetAsSeries* 函数进行更改。

SetIndexBuffer 函数成功时返回 *true*，失败时返回 *false*。

可选 *mode* 参数用于告知系统该缓冲区的使用方式。可能的值在 *ENUM_INDEXBUFFER_TYPE* 枚举中提供。

标识符	说明
INDICATOR_DATA	要渲染的数据
INDICATOR_COLOR_INDEX	渲染颜色
INDICATOR_CALCULATIONS	中间计算的内部结果

默认情况下，指标缓冲区用于绘制数据 (*INDICATOR_DATA*)。该值除了在图表上显示数组外，还有另一作用：鼠标光标所在柱线对应的每个缓冲区值会显示在 *Data window* 中。不过，这种行为可以通过某些指标设置更改 (请参阅 *图形绘图设置* 章节的 *PLOT_SHOW_DATA* 特性)。本章的大多数示例都采用 *INDICATOR_DATA* 模式。

如果指标计算需要存储每根柱线的中间结果，可以为它们分配一个辅助性的非显示缓冲区 (*INDICATOR_CALCULATIONS*)。相较于使用普通数组实现目的，这种方式更加便捷，因为使用普通数组时，程序员必须自行管理数组大小。本章将展示使用 *INDICATOR_CALCULATIONS* 的两个示例：*IndTripleEMA.mq5* (请参阅 *跳过初始柱线的绘制*) 和 *IndSubChartSimple.mq5* (请参阅 *多货币和多时间范围指标*)。

某些结构允许为每根柱线设置显示颜色。颜色缓冲区 (*INDICATOR_COLOR_INDEX*) 用于存储颜色信息。颜色由整数类型 *color* 表示，但所有指标缓冲区必须为 *double* 类型，在这种情况下，缓冲

区将存储由开发者设置的特殊调色板中的颜色编号 (请参阅 [图表逐元素着色](#) 章节以及其示例指标 *IndColorWPR.mq5*)。

颜色和辅助缓冲区的值不会显示在 *Data window* 中, 并且无法使用 *CopyBuffer* 函数获取这些值 (我们将在 [MQL5 内置指标与自定义指标使用](#) 章节中详细探讨该函数)。

指标缓冲区在初始化时不会填充任何数值。如果由于某种原因未计算其某些元素 (例如, 指标设置中存在最大柱线数量限制, 或者图形构建本身要求在重要元素之间保留间隔, 如锯齿线顶点之间), 则应显式将这些位置填充为特殊的“空值”。空值不会显示在图表上, 也不会显示在 *Data Window* 中。默认情况下, 它存在 `EMPTY_VALUE (DBL_MAX)` 常量, 但如有必要, 也可替换为其他值, 例如 0。这可以通过 *PlotIndexSetDouble* 函数实现。

基于对 *SetIndexBuffer* 函数的新认识, 让我们完善上一节开始的示例 *IndReplica1.mq5*。具体来说, 我们需要 *OnInit* 处理程序。

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1

#include <MQL5Book/PRTF.mqh>

double buffer[]; // global dynamic array

int OnInit()
{
    // register an array as an indicator buffer
    PRTF(SetIndexBuffer(0, buffer)); // true / ok
    // the second incorrect call is made here intentionally to show an error
    PRTF(SetIndexBuffer(1, buffer)); // false / BUFFERS_WRONG_INDEX(4602)
    // check size: still 0
    PRTF(ArraySize(buffer)); // 0
    return INIT_SUCCEEDED;
}
```

由于通过指令定义的缓冲区数量为 1, 因此为单个缓冲区分配数组时应使用索引 0 (即 *SetIndexBuffer* 的第一个参数)。第二个函数调用是错误示例, 仅用于演示问题: 由于索引 1 对应两个已声明的缓冲区, 这将产生 `BUFFERS_WRONG_INDEX (4602)` 错误。

在 *OnCalculate* 函数的最开始, 让我们再次打印输出数组的大小。此时, 数组已根据柱线数量分配完毕。

```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &data[])
{
    // after starting, check that the platform automatically manages the size of the a
    if(prev_calculated == 0)
    {
        PRTF(ArraySize(buffer)); // 10189 - actual number of bars
    }
    ...
}

```

现在，让我们讨论指标的计算逻辑。如前所述，我们暂不加入复杂公式，仅尝试将 *data* 参数传入的时间序列复制到缓冲区。这也反映在指标的命名中。

```

...
// on each new bar or set of bars (including the first calculation)
if(prev_calculated != rates_total)
{
    // fill in all new bars
    ArrayCopy(buffer, data, prev_calculated, prev_calculated);
}
else // ticks on the current bar
{
    // update the last bar
    buffer[rates_total - 1] = data[rates_total - 1];
}

// we report the number of processed bars to ourselves in the future
return rates_total;
}

```

现在，该指标可以正常编译，无警告。我们可以在图表上运行该指标，在默认设置下，它应在缓冲区中复制柱线收盘价的值。这基于 *OnCalculate* 的简化形式，我们在 [主要指标事件：OnCalculate](#) 中探讨过这一方面。

但存在一个特殊现象：*Data window* 中的缓冲区数值显示正确，但图表上却未出现对应线条。这是因为负责显示的是图形结构，而非缓冲区。在当前版本的指标中，我们仅配置了缓冲区。在下一章节中，我们将创建新版本的 *IndReplica2.mq5* 并补充必要的指令。

同时，上述效果可用于创建“隐藏指标”，这类指标不在图表上显示线条，但可被其他 MQL 程序通过编程方式读取。如果需要，开发者甚至可以在 *Data windows* 中隐藏指标缓冲区的显示（请参阅下一节的 *PLOT_SHOW_DATA*）。

关于如何通过 MQL5 代码管理指标，我们将在 [下一章](#) 中讨论

5.4.6 绘图设置：PlotIndexSetInteger

MQL5 API 提供了以下用于配置绘图的函数：*PlotIndexSetInteger*、*PlotIndexSetDouble* 和 *PlotIndexSetString*。整数特性也可以通过 *PlotIndexGetInteger* 读取。我们主要关注整数特性。

PlotIndexSetInteger 函数有两种形式。稍后，我们将介绍它们的区别。

```
bool PlotIndexSetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int value)
bool PlotIndexSetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int modifier,
int value)
```

该函数用于设置指定 *index* 处图形绘图的特性值。*index* 值必须介于 0 和 P - 1 之间，其中 P 表示由 *#property indicator_plots* 指令指定的绘图数量。特性本身由 *property* 参数标识：允许的值应从 ENUM_PLOT_PROPERTY_INTEGER 枚举选取（见下文）。特性值通过参数 *value* 传递。

该函数的第二种形式用于应用于多个组件的特性（尽管这些组件属于同一个特性）。特别是，对于某些类型的图表，可以分配一组颜色，而非单一颜色。在这种情况下，你可以使用 *modifier* 参数更改该组的任何颜色。

该函数成功时返回 *true*，否则返回 *false*。

下表提供可用的 ENUM_PLOT_PROPERTY_INTEGER 特性。

标识符	说明	特性类型
PLOT_ARROW	DRAW_ARROW 图表的 Wingdings 字体的箭头代码	uchar
PLOT_ARROW_SHIFT	DRAW_ARROW 图表的垂直箭头偏移量	int
PLOT_DRAW_BEGIN	数据起始的第一根柱线索引（从左到右）	int
PLOT_DRAW_TYPE	绘图（图表）类型	ENUM_DRAW_TYPE
PLOT_SHOW_DATA	在 <i>Data window</i> 中显示绘图值的标志（ <i>true</i> - 可见， <i>false</i> - 不可见）	bool
PLOT_SHIFT	指标图形沿时间轴的柱线偏移量（正值向右偏移，负值向左偏移）	int
PLOT_LINE_STYLE	线条绘制风格	ENUM_LINE_STYLE
PLOT_LINE_WIDTH	设置线条粗细（像素，范围 1-5）	int
PLOT_COLOR_INDEXES	颜色数量（1-64）	int
PLOT_LINE_COLOR	渲染颜色	颜色（ <i>modifier</i> - 颜色编号）

我们将逐步学习所有特性，但目前，我们将关注三个主要特性：PLOT_DRAW_TYPE、PLOT_LINE_STYLE 和 PLOT_LINE_COLOR。

MetaTrader 5 中的指标支持多种预定义的绘制类型。它们决定了可视化表示方式以及用于显示的初始数据缓冲区的所需结构。

总共有 10 种此类基本绘图类型，在 MQL5 层面，它们由 ENUM_DRAW_TYPE 枚举中的标识符描述。PLOT_DRAW_TYPE 特性应被赋予其中一个 ENUM_DRAW_TYPE 值。

可视化类型， 示例	说明	缓冲区 数量
DRAW_NONE <i>IndDeltaVolume.mq5</i>	图表上不显示任何内容，但相应缓冲区的值可在数据窗口中查看。	1
DRAW_LINE <i>IndLabelHighLowClose.mq5</i> , <i>IndWPR.mq5</i> , <i>IndUnityPercent.mq5</i>	根据缓冲区值绘制曲线（“空”元素会在线条中形成间隙）。	1
DRAW_SECTION	在“非空”缓冲区元素之间形成折线的直线段（如果没有间隙，则类似于 DRAW_LINE）。	1
DRAW_ARROW <i>IndReplica3.mq5</i> , <i>IndFractals.mq5</i>	字符（标签）	1
DRAW_HISTOGRAM <i>IndDeltaVolume.mq5</i>	从零线到缓冲区值的直方图	1
DRAW_HISTOGRAM2 <i>IndLabelHighLowClose.mq5</i>	两个指标缓冲区的配对元素值之间的直方图	2
DRAW_ZIGZAG <i>IndFractalsZigZag.mq5</i>	在两个缓冲区中连续出现的“非空”元素之间形成折线的直线段（类似于 DRAW_SECTION，但允许在单个柱线上有垂直线段）。	2
DRAW_FILLING	根据两个缓冲区中的配对值对两条线之间的通道进行颜色填充	2
DRAW_BARS <i>IndSubChartSimple.mq5</i>	以柱线显示：每个柱线的四个价格按 OHLC 顺序显示在四个相邻的缓冲区中。	4
DRAW_CANDLES <i>IndSubChartSimple.mq5</i>	以蜡烛图显示：每个柱线的四个价格按 OHLC 顺序显示在四个相邻的缓冲区中。	4

此表未列出所有 ENUM_DRAW_TYPE 元素。存在支持对单个元素（柱线）进行着色的同类绘图类型。我们将在单独章节 [图表逐元素着色](#) 中介绍。MQL5 文档提供了 [所有类型的示例](#)，而在本书范围内存在一些例外：类型名称旁会标注是否存在演示指标。

在所有情况下（包括 DRAW_NONE），其他程序均可通过 [CopyBuffer](#) 函数获取缓冲区数据。

DRAW_NONE 类型的一个额外特性是，此类缓冲区的值不参与图表的自动缩放，而默认情况下，显示在 [子窗口](#) 中的指标会启用自动缩放功能。

线条风格由 PLOT_LINE_STYLE 特性决定，该特性还具有包含有效 ENUM_LINE_STYLE 值的枚举。

标识符	说明
STYLE_SOLID	实线
STYLE_DASH	虚线
STYLE_DOT	点线
STYLE_DASHDOT	点划线
STYLE_DASHDOTDOT	双点划线

最后，线条颜色由 `PLOT_LINE_COLOR` 特性设置。在最简单的情况下，该特性包含应用于整个图表的单一颜色。对于某些图表类型（尤其是 `DRAW_CANDLES`），可以使用 `modifier` 参数指定多种颜色。我们将在后面讨论这一点（请参阅 [多货币和多时间范围指标](#) 章节的 *IndSubChartSimple.mq5* 示例）。

上述三个特性足以演示指标 *IndReplica2.mq5*。我们分别添加 `ENUM_DRAW_TYPE` 和 `ENUM_LINE_STYLE` 类型的两个输入参数 *DrawType* 和 *LineStyle*，然后在 *OnInit* 函数中多次调用 *PlotIndexSetInteger* 函数来设置指标的渲染特性。

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1

input ENUM_DRAW_TYPE DrawType = DRAW_LINE;
input ENUM_LINE_STYLE LineStyle = STYLE_SOLID;

double buffer[];

int OnInit()
{
    // register an array as an indicator buffer
    SetIndexBuffer(0, buffer);

    // set the properties of the chart numbered 0
    PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DrawType);
    PlotIndexSetInteger(0, PLOT_LINE_STYLE, LineStyle);
    PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrBlue);

    return INIT_SUCCEEDED;
}
```

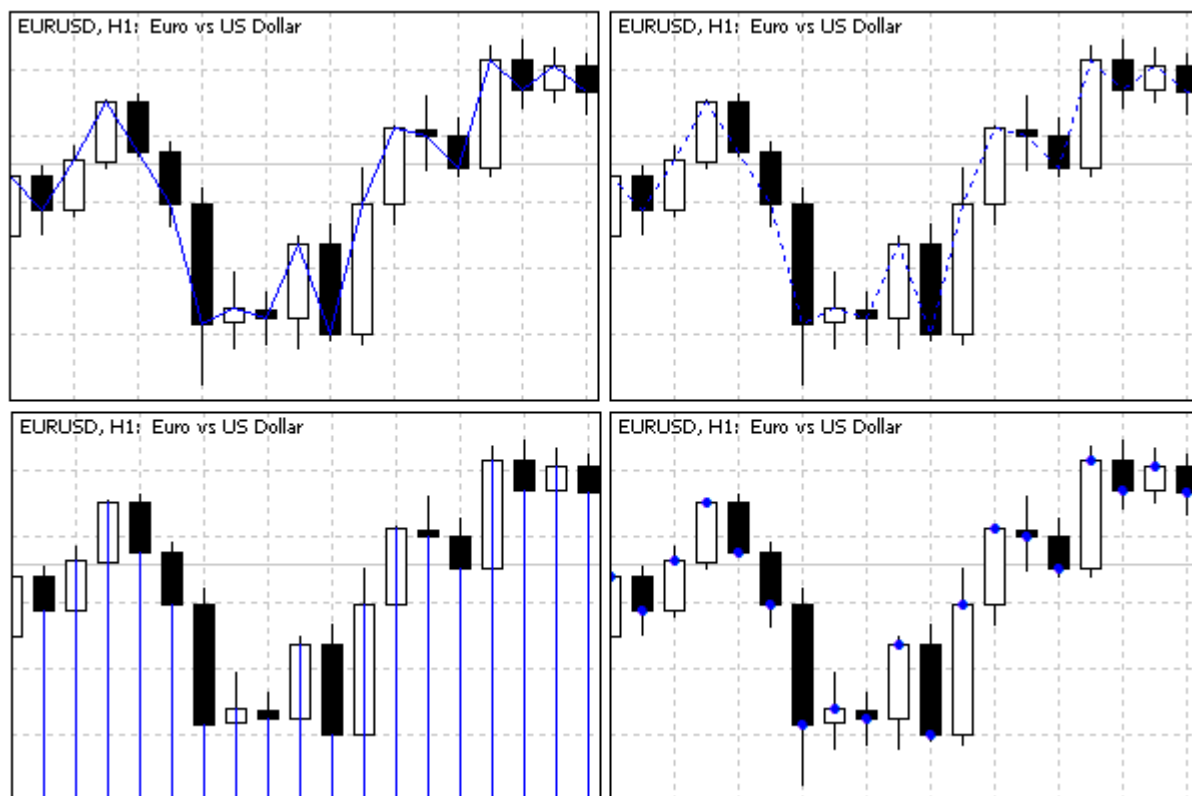
对于 `PLOT_LINE_COLOR` 特性，我们没有创建输入变量，因为该特性及其他一些特性可直接从任何指标的特性对话框中的 *Colors* 选项卡中获取。默认情况下，即指标启动后，线条颜色为蓝色。但颜色以及线条粗细和风格均可在对话框（指定选项卡）中更改。我们的 *LineStyle* 参数部分复制了 *Colors* 表格中的相应 *Style* 单元格。但它提供了额外的优势。对话框的标准控件不允许在线宽大于 1 时选择线型。使用输入变量 *LineStyle* 时，我们可以获得例如宽度为 3 像素的点划线。

与 *IndReplica1.mq5* 相比，*OnCalculate* 函数中向缓冲区填充数据的方式完全一致。

在编译指标并在图表上启动指标后，我们得到预期画面：图表上收盘价位置显示一条蓝色线条，*Data window*中显示相应柱线的收盘价。

通过更改 *DrawType* 输入参数，我们可以更改缓冲区数据的显示方式。在这种情况下，应仅选择需要单个缓冲区的类型。任何其他图形类型 (*DRAW_HISTOGRAM2*、*DRAW_ZIGZAG*、*DRAW_FILLING*、*DRAW_BARS*、*DRAW_CANDLES*) 在单个缓冲区上使用，因此不会显示任何内容。同样，选择带着色的结构类型 (名称以 “Color” 关键字开头) 也不可行，因为它们需要每个柱线都有一个包含颜色编号的额外缓冲区 (如前所述，我们将在 [图表逐元素着色](#) 章节中详细介绍该功能)。

下面展示了 *DRAW_LINE*、*DRAW_SECTION*、*DRAW_HISTOGRAM* 和 *DRAW_ARROW* 显示选项。



一个缓冲区的图表类型

如果不是特意选择了不同的线型 (*DRAW_LINE* 选择 *STYLE_SOLID*，*DRAW_SECTION* 选择 *STYLE_DOT*)，这些绘制类型将完全相同，因为我们的缓冲区中所有元素均为 “非空” 值。默认情况下，“空” 值指的是特殊常量 *EMPTY_VALUE* (本示例未使用)。*DRAW_SECTION* 中的线段会绕过 “空” 元素进行绘制，但这一特性仅在存在空元素时才会显现。我们将在 [数据间隙可视化](#) 章节中介绍空元素的设置。

从零线开始的直方图 *DRAW_HISTOGRAM* 通常用于拥有独立窗口的指标，但此处仅为演示用途而展示。我们将在 [等待数据和管理可见性](#) 章节中创建一个使用此渲染类型的子窗口指标 (请参阅 *IndDeltaVolume.mq5* 示例)。

对于 *DRAW_ARROW* 类型，系统默认使用实心圆字符 (代码 159)，但可通过调用 *PlotIndexSetInteger(index, PLOT_ARROW, code)* 将其更改为其他符号。

有关 *Wingdings* 的代码和外观，可在 MQL5 帮助中查找。

在 *IndReplica3.mq5* 指标的另一个修改版本中，我们新增了以下输入参数：用于选择 “箭头” 符号 (*ArrowCode*)，以及用于在图表上垂直移动这些标记 (*Arrow padding*) 和水平移动这些标记 (*TimeShift*)。


```

input uchar ArrowCode = 159;
input int ArrowPadding = 0;
input int TimeShift = 0;

```

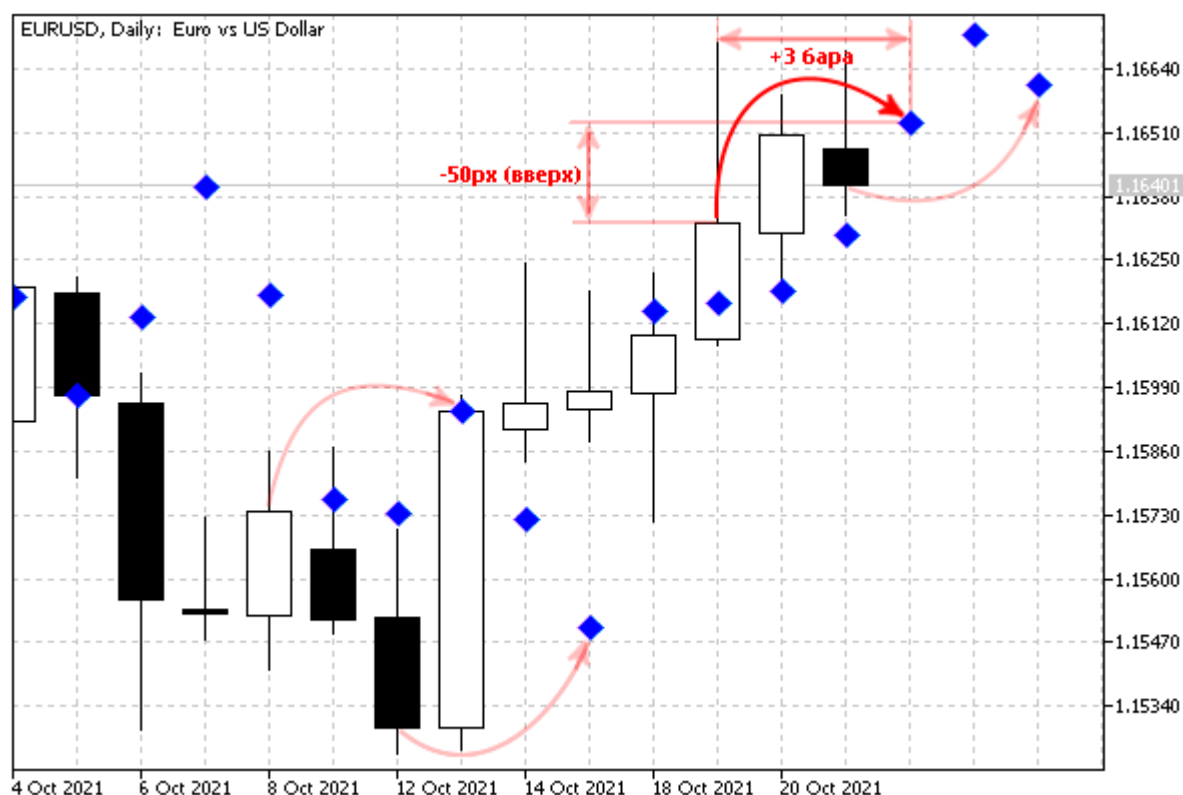
沿价格轴的垂直移动以像素为单位指定（正值表示向下移动，负值表示向上移动）。沿时间轴的水平移动以柱线为单位设置（正值表示向右移动，即向未来方向，负值表示向左移动，即向过去方向）。新的输入变量通过 *OnInit* 中的 *PlotIndexSetInteger* 调用传递。

```

int OnInit()
{
    ...
    PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_ARROW);
    PlotIndexSetInteger(0, PLOT_ARROW, ArrowCode);
    PlotIndexSetInteger(0, PLOT_ARROW_SHIFT, ArrowPadding);
    PlotIndexSetInteger(0, PLOT_SHIFT, TimeShift);
    ...
}

```

以下截图展示了 *IndReplica3.mq5* 在图表上的示例，其设置为 117（菱形标记）、-50（向上移动 50 点）、3（向右/向前移动 3 根柱线）。



带垂直和水平标签偏移的散点图

我们的默认指标基于 *Close* 价类型（但用户可以在特性对话框的 *Apply to* 下拉列表中更改此设置）。如有需要，你可以通过以下指令指定不同的初始配置：

```
#property indicator_applied_price PRICE_TYPE
```

此处请勿指定 *PRICE_TYPE*，而是应指定 *ENUM_APPLIED_PRICE* 枚举中的任意常量。它还包括 *PRICE_CLOSE*，对应于默认设置。例如，在源代码中添加以下指令将使指标默认基于典型价格：

```
#property indicator_applied_price PRICE_TYPICAL
```

再次强调，该设置仅指定默认值。内置 `_AppliedTo` 变量允许你确定指标所基于的实际价格类型。如果指标是根据另一个指标的描述符构建的，则只能确定这一事实，而无法获知提供数据的具体指标名称。

如果要在源代码中查询 `ENUM_PLOT_PROPERTY_INTEGER` 枚举中的特性当前状态，需使用 `PlotIndexGetInteger` 函数。

```
int PlotIndexGetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property)
int PlotIndexGetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int modifier)
```

该函数通常与 `PlotIndexSetInteger` 配合使用，用于将一条线的绘制特性复制到另一条线，或从各类指标源代码所包含的通用 `.mqh` 文件代码中读取特性。

遗憾的是，目前没有提供类似的 `PlotIndexGetDouble` 和 `PlotIndexGetString` 函数。

5.4.7 缓冲区和图表映射规则

在使用 `PlotIndexSetInteger(i, PLOT_DRAW_TYPE, type)` 注册图表时，每次调用会根据渲染 `type` 的需求数量，按顺序为第 `i` 个图表分配一定数量的缓冲区（请参阅 [上一节](#) 中的 `ENUM_DRAW_TYPE` 表格）。因此，在将缓冲区链接到后续图表时（在后续调用 `PlotIndexSetInteger` 时），这些已分配的缓冲区将不再参与分配。

例如，如果第一个绘图（索引为 0）是 `DRAW_CANDLES`，它需要 4 个指标缓冲区，则会精确分配 4 个缓冲区与之关联。因此，索引 0 至 3（含边界值）的缓冲区将被绑定，下一个可供绑定的空闲缓冲区索引为 4。

如果接下来注册一个简单折线图 `DRAW_LINE`（其在图表序列中的索引为 1），它将仅占用 1 个缓冲区，即索引为 4 的缓冲区。

如果进一步配置一个 `DRAW_ZIGZAG` 图表（下一个图表索引为 2），由于它需要使用两个缓冲区，因此索引为 5 和 6 的缓冲区将分配给它。

当然，缓冲区数量必须足够满足所有已注册绘图的需求。上述示例在下表中进行了说明。它仅有 7 个缓冲区和 3 个绘图（图表）。

SetIndexBuffer 中的 缓冲区索引	0	1	2	3	4	5	6
PlotIndexSetInteger 中的 图表索引	0				1	2	
渲染类型	DRAW_CANDLES				DRAW _LINE	DRAW_ZIGZAG	

缓冲区索引与图表索引是相互独立的，即缓冲区索引不必与图表索引相同。同时，随着图表索引的递增，绑定到这些图表的缓冲区索引也会递增，如果使用的渲染类型需要占用多个缓冲区，则索引之间的差异会越来越大。

虽然习惯上在调用 `PlotIndexSetInteger` 函数之前先调用 `SetIndexBuffer` 函数，但这并非强制要求。唯一重要的是缓冲区索引与图表索引的正确对应关系。当使用指令（请参阅 [下一节](#)）替代调用 `PlotIndexSetInteger` 时，无论如何，指令都会在 `OnInit` 处理程序之前执行。

为了演示缓冲区索引与图表索引的区别,我们让我们来看一个简单的示例

IndHighLowClose.mq5。在该文件中,我们将以 `DRAW_HISTOGRAM2` 类型直方图的形式绘制每个柱线的 *High*和 *Low* 之间的区间,并使用 `DRAW_LINE` 的简单线条标记 *Close* 价格。要访问不同类型的价格时间序列,我们还需要将 *OnCalculate*函数的形式从简化版改为完整版。

由于直方图需要 2 个缓冲区,再加上用于 *Close*线条的缓冲区,我们应该定义三个缓冲区。

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 2

double highs[];
double lows[];
double closes[];
```

在 *OnInit*中按优先级顺序注册这些缓冲区。

```
int OnInit()
{
    // arrays for buffers for 3 price types
    SetIndexBuffer(0, highs);
    SetIndexBuffer(1, lows);
    SetIndexBuffer(2, closes);

    // drawing a histogram between the High and Low candles under index 0
    PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_HISTOGRAM2);
    PlotIndexSetInteger(0, PLOT_LINE_WIDTH, 5);
    PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrBlue);

    // drawing the line Close at index 1
    PlotIndexSetInteger(1, PLOT_DRAW_TYPE, DRAW_LINE);
    PlotIndexSetInteger(1, PLOT_LINE_WIDTH, 2);
    PlotIndexSetInteger(1, PLOT_LINE_COLOR, clrRed);

    return INIT_SUCCEEDED;
}
```

同时,将直方图的宽度设置为 5 像素,线条宽度设置为 2 像素。未明确指定风格,因此将默认使用 `STYLE_SOLID`。

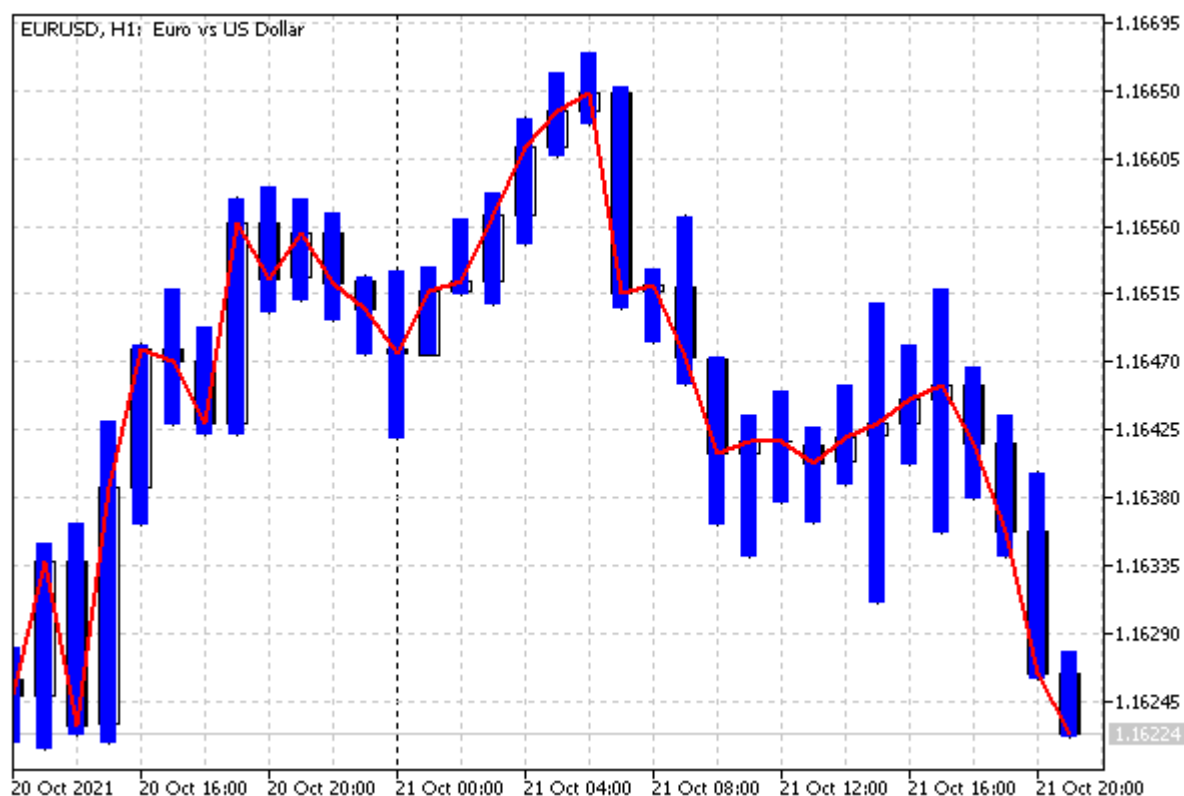
现在,让我们来看看实际的 *OnCalculate*函数。

```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const datetime &time[],
               const double &open[],
               const double &high[],
               const double &low[],
               const double &close[],
               const long &tick_volume[],
               const long &volume[],
               const int &spread[])
{
    // on each new bar or set of bars (including the first calculation)
    if(prev_calculated != rates_total)
    {
        // fill in all new bars
        ArrayCopy(highs, high, prev_calculated, prev_calculated);
        ArrayCopy(lows, low, prev_calculated, prev_calculated);
        ArrayCopy(closes, close, prev_calculated, prev_calculated);
    }
    else // ticks on the current bar
    {
        // update the last bar
        highs[rates_total - 1] = high[rates_total - 1];
        lows[rates_total - 1] = low[rates_total - 1];
        closes[rates_total - 1] = close[rates_total - 1];
    }
    // return the number of processed bars for the next call
    return rates_total;
}

```

该指标的效果如下图所示：



高低价直方图与收盘价线

请注意一个重要细节：图表将按照其索引顺序进行绘制，这可能导致某些图表在视觉上位于其他图表的上方（产生重叠）。在这种情况下，首先绘制索引为 0 的直方图，然后在其上方绘制索引为 1 的线条。有时，调整图表的注册顺序是有意义的，这样可以让较小的图形结构获得更好的可见性，因为它们可能会被较大（较宽）的图形覆盖。

沿着假想的 Z 轴（垂直于屏幕、向屏幕纵深方向）设置此类优先级的方式，称为 Z 轴顺序。我们在学习 [图形对象](#) 时再此遇到这种方法。

另外，请记住，指标默认显示在价格图表的上方，但可以通过以下设置更改此行为：**Chart Properties**对话框 **Common** 选项卡 **Chart on foreground** 选项。在软件界面中有一个类似选项（`ChartSetInteger(CHART_FOREGROUND)`），请参阅 [图表显示模式](#)。

5.4.8 应用指令以自定义绘图

到目前为止，我们一直使用 `PlotIndexSetInteger` 函数调用来自定义图形绘图。MQL5 允许你使用 `#property` 预处理器指令实现相同的功能。这两种方法的主要区别在于，指令在编译时处理，使用它们定义的特性在加载时从可执行文件中读取，甚至早于 `OnInit` 处理程序的执行（如果存在的话）换言之，指令提供了一些默认值，如果你不需要更改它们，则可以直接使用。

另一方面，`PlotIndexSetInteger` 函数调用允许你在程序执行期间动态更改特性。通过函数动态更改特性，你可以创建更灵活的指标使用场景。指令和相关 `PlotIndexSetInteger` 函数调用如下表所示。

指令	函数	说明
indicator_colorN	PlotIndexSetInteger(N-1, PLOT_LINE_COLOR, color)	设置绘图的颜色
indicator_styleN	PlotIndexSetInteger(N-1, PLOT_LINE_STYLE, type)	设置风格 (来自 ENUM_LNE_STYLE 枚举)
indicator_typeN	PlotIndexSetInteger(N-1, PLOT_DRAW_TYPE, type)	设置类型 (来自 ENUM_DRAW_TYPE 枚举)
indicator_widthN	PlotIndexSetInteger(N-1, PLOT_LINE_WIDTH, width)	设置线条粗细 (像素, 范围 1-5)

请注意，指令中的绘图编号从 1 开始，而函数中的编号从 0 开始。例如，指令 `#property indicator_type1 DRAW_ZIGZAG` 等效于调用 `PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_ZIGZAG)`。

此外，值得注意的是，通过函数可以设置的特性比通过指令多得多：

`ENUM_PLOT_PROPERTY_INTEGER` 枚举提供了十个元素。

通过指令定义的特性，即使在指标第一次添加到图表时，也会显示在指标设置对话框中（用户可查看和编辑）。具体而言，这些特性包括线条的粗细、颜色和风格（**Colors** 选项卡），以及水平层级的数量和位置（**Levels** 选项卡）。而通过函数设置的相同特性（但在指令中没有默认值的情况下），仅会在指标第二次及后续添加到图表时才会出现在对话框中。

让我们调整 `IndHighLowClose.mq5` 指标，改用指令。新版本文件名为 `IndPropHighLowClose.mq5`。使用指令简化了 `OnInit` 处理程序；`OnCalculate` 保持不变。

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 2

// High-Low histogram rendering settings (change index 0 to 1 in the directive)
#property indicator_type1 DRAW_HISTOGRAM2
#property indicator_style1 STYLE_SOLID // by default, can be omitted
#property indicator_color1 clrBlue
#property indicator_width1 5

// close line drawing settings (change index 1 to 2 in the directive)
#property indicator_type2 DRAW_LINE
#property indicator_style2 STYLE_SOLID // by default, can be omitted
#property indicator_color2 clrRed
#property indicator_width2 2

double highs[];
double lows[];
double closes[];

int OnInit()
{
    // arrays for buffers for 3 price types
    SetIndexBuffer(0, highs);
    SetIndexBuffer(1, lows);
    SetIndexBuffer(2, closes);

    return INIT_SUCCEEDED;
}
```

新指标的外观与旧版本完全一致。

5.4.9 设置绘图名称

在本章中前面的示例中，数据窗口中的指标缓冲区默认使用指标本身的名称进行标识。这种做法缺乏信息性。MQL5 API 提供了为每个缓冲区设置自定义名称的功能。这可通过以下两种已知方式实现：一是使用 *#property* 指令，二是调用特殊的 *PlotIndexSetString* 函数。

```
bool PlotIndexSetString(int index, ENUM_PLOT_PROPERTY_STRING property, string value)
```

该函数的原型与 *PlotIndexSetInteger* 类似，区别在于特性类型 (*value* 参数) 为 *string*。该函数仅支持一个 *PLOT_LABEL* 特性 (即 *ENUM_PLOT_PROPERTY_STRING* 枚举常量)。 *index* 索引参数中的自定义图表索引必须在 0 到 *N-1* 之间，其中 *N* 是由 *#property indicator_plots N* 指定的绘图总数。

使用指令时，图表索引需按 1 进行调整，因为指令中的绘图编号从 1 开始，而函数参数的编号从 0 开始。

指令	函数	说明
#property indicator_labelN	PlotIndexSetString(N-1, PLOT_LABEL, string)	指定在 <i>Data window</i> 和工具提示中显示的文本标签

对于需要多个指标缓冲区的图形系列（例如 `DRAW_CANDLES`、`DRAW_FILLING`、`DRAW_CANDLES` 等等），标签名称使用分号 “;” 分隔

当鼠标悬停在图表上时，标签也会显示在工具提示中。

在 `IndLabelHighLowClose.mq5` 示例中，我们添加了两条指令（与 `IndPropHighLowClose.mq5` 的区别）。

```
#property indicator_label1 "High;Low"
#property indicator_label2 "Close"
```

现在，*Data Window* 中显示指标时，其显示的值要容易理解得多。

5.4.10 可视化数据缺口（空元素）

在许多情况下，指标读数应仅显示在某些柱线上，其余柱线则保持原样（视觉上不出现额外的线条或标签）。例如，许多信号指标仅在出现买入或卖出建议的柱线上，才显示向上或向下箭头。但信号出现的频率通常很低。

使用 `PlotIndexSetDouble` 函数可以设置空值，该空值既不会在图表上显示，也不会出现在 *Data Window* 中。

`bool PlotIndexSetDouble(int index, ENUM_PLOT_PROPERTY_DOUBLE property, double value)`

该函数用于为指定 *index* 的绘图设置 *double* 特性。此类特性的集合在 `ENUM_PLOT_PROPERTY_DOUBLE` 枚举中列出，但目前该枚举仅包含一个元素：`PLOT_EMPTY_VALUE`。它还设置空值。实际空值通过最后一个参数 *value* 传递。

作为一个数值稀疏型指标的示例，我们将讨论分形检测器。该指标会在图表上标记满足以下条件的价格：高价（*High*）：高于左右两侧各 *N* 个相邻柱线的高点；低价（*Low*）：低于左右两侧各 *N* 个相邻柱线的低点。指标文件名为 `IndFractals.mq5`。

该指标将包含两个缓冲区和两个 `DRAW_ARROW` 类型的绘图。

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots 2

// rendering settings
#property indicator_type1 DRAW_ARROW
#property indicator_type2 DRAW_ARROW
#property indicator_color1 clrBlue
#property indicator_color2 clrRed
#property indicator_label1 "Fractal Up"
#property indicator_label2 "Fractal Down"

// indicator buffers
double UpBuffer[];
double DownBuffer[];
```

FractalOrder 输入变量允许你设置用于确定上极值或下极值的相邻柱线数量。

```
input int FractalOrder = 3;
```

为了提高可见性，我们将让箭头符号与极值保持 10 像素的间距。


```
const int ArrowShift = 10;
```

在 *OnInit* 函数中，声明数组作为缓冲器并将其绑定到绘图上。

```
int OnInit()
{
    // binding buffers
    SetIndexBuffer(0, UpBuffer, INDICATOR_DATA);
    SetIndexBuffer(1, DownBuffer, INDICATOR_DATA);

    // up and down arrow character codes
    PlotIndexSetInteger(0, PLOT_ARROW, 217);
    PlotIndexSetInteger(1, PLOT_ARROW, 218);

    // padding for arrows
    PlotIndexSetInteger(0, PLOT_ARROW_SHIFT, -ArrowShift);
    PlotIndexSetInteger(1, PLOT_ARROW_SHIFT, +ArrowShift);

    // setting an empty value (can be omitted, since EMPTY_VALUE is the default)
    PlotIndexSetDouble(0, PLOT_EMPTY_VALUE, EMPTY_VALUE);
    PlotIndexSetDouble(1, PLOT_EMPTY_VALUE, EMPTY_VALUE);

    return FractalOrder > 0 ? INIT_SUCCEEDED : INIT_PARAMETERS_INCORRECT;
}
```

请注意，默认的空值是特殊常量 `EMPTY_VALUE`，因此上述对 *PlotIndexSetDouble* 的调用是可选的。

在 *OnCalculate* 处理程序中，第一次调用时，使用 `EMPTY_VALUE` 初始化两个数组，随后随着柱线的形成，将该值赋给新的元素。这种填充是必要的，因为指标缓冲器分配的内存可能包含任意数据（垃圾数据）。

```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const datetime &time[],
               const double &open[],
               const double &high[],
               const double &low[],
               const double &close[],
               const long &tick_volume[],
               const long &volume[],
               const int &spread[])
{
    if(prev_calculated == 0)
    {
        // at the start, fill the arrays entirely
        ArrayInitialize(UpBuffer, EMPTY_VALUE);
        ArrayInitialize(DownBuffer, EMPTY_VALUE);
    }
    else
    {
        // on new bars we also clean the elements
        for(int i = prev_calculated; i < rates_total; ++i)
        {
            UpBuffer[i] = EMPTY_VALUE;
            DownBuffer[i] = EMPTY_VALUE;
        }
    }
    ...
}

```

在主循环中，逐根柱线执行以下操作：将 *high* 和 *low* 价格与相邻柱线的同类型价格进行对比，并在发现当前价格在左右两侧各 *FractalOrder* 根柱线范围内构成极值的位置设置标记。

```

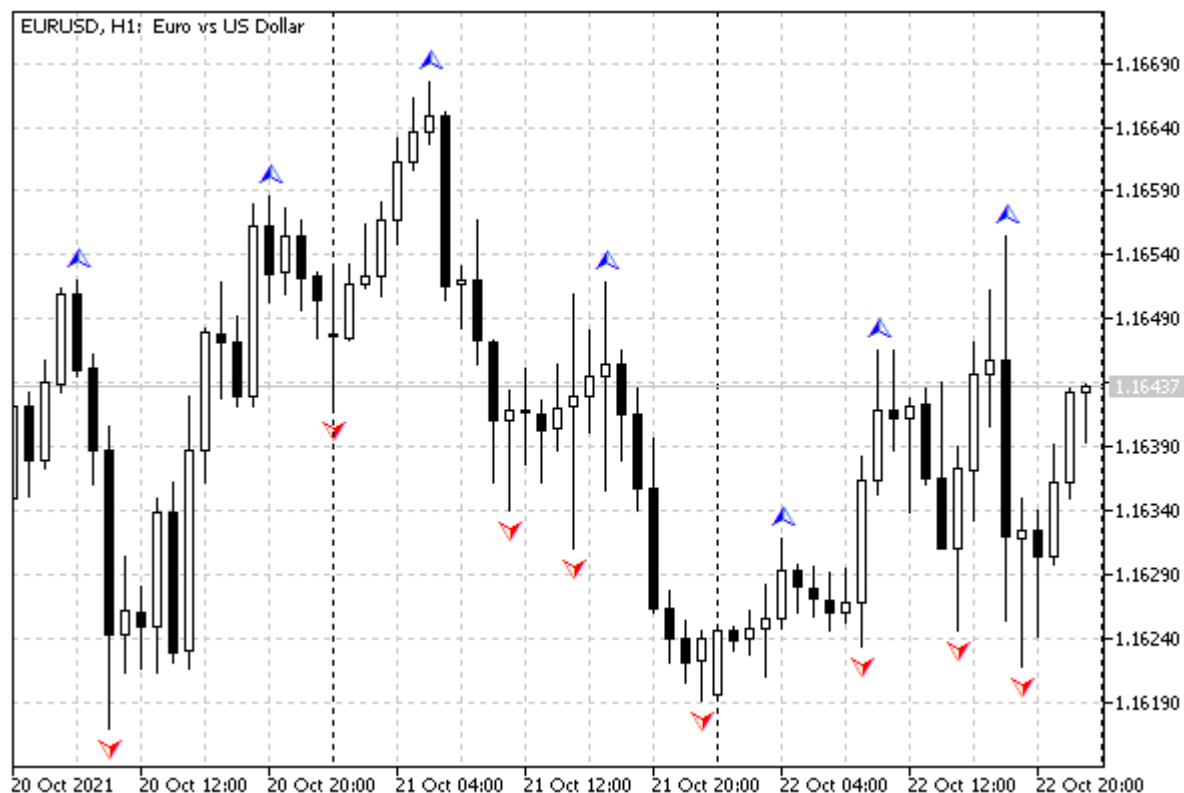
// view all or new bars that have bars in the FractalOrder environment
for(int i = fmax(prev_calculated - FractalOrder - 1, FractalOrder);
    i < rates_total - FractalOrder; ++i)
{
    // check if the upper price is higher than neighboring bars
    UpBuffer[i] = high[i];
    for(int j = 1; j <= FractalOrder; ++j)
    {
        if(high[i] <= high[i + j] || high[i] <= high[i - j])
        {
            UpBuffer[i] = EMPTY_VALUE;
            break;
        }
    }

    // check if the lower price is lower than neighboring bars
    DownBuffer[i] = low[i];
    for(int j = 1; j <= FractalOrder; ++j)
    {
        if(low[i] >= low[i + j] || low[i] >= low[i - j])
        {
            DownBuffer[i] = EMPTY_VALUE;
            break;
        }
    }
}

return rates_total;
}

```

让我们看看该指标在图表上的表现。



分形指标

现在我们将绘图类型从 `DRAW_ARROW` 改为 `DRAW_ZIGZAG`，并比较这两个选项在空值情况下的效果。结果应是基于分形的锯齿线。脚本的修改版本附在文件 *IndFractalsZigZag.mq5* 中。

主要变更之一与图表数量有关：现在只有一个图表，因为 `DRAW_ZIGZAG` 函数会“占用”两个缓冲器。

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots 1

// rendering settings
#property indicator_type1 DRAW_ZIGZAG
#property indicator_color1 clrMediumOrchid
#property indicator_width1 2
#property indicator_label1 "ZigZag Up;ZigZag Down"
...
```

所有与设置箭头相关的函数调用已从 *OnInit* 中移除。

```

int OnInit()
{
    SetIndexBuffer(0, UpBuffer, INDICATOR_DATA);
    SetIndexBuffer(1, DownBuffer, INDICATOR_DATA);

    PlotIndexSetDouble(0, PLOT_EMPTY_VALUE, EMPTY_VALUE);

    return FractalOrder > 0 ? INIT_SUCCEEDED : INIT_PARAMETERS_INCORRECT;
}

```

源代码其余部分保持不变。

下图展示了同时应用锯齿线及分形的图表，因此，你可以直观地比较它们的结果。这两个指标完全独立运行，但由于算法相同，所找到的极值是一样的。



基于分形的锯齿线指标

需要注意的是，如果同一类型的极值连续出现，锯齿线会采用第一个极值。这是由于分形被用作极值的必然结果。当然，这种情况在标准锯齿线中不会发生。如有必要，开发者可以通过先对分形序列进行细化处理，对算法进行优化。

还需注意，对于渲染 `DRAW_ZIGZAG` (以及 `DRAW_SECTION`)，可见线段连接的是非空元素，因此严格来说，每条柱形上都会绘制部分线段片段，包括值为 `EMPTY_VALUE` (或其他指定替代值) 的柱形。然而，在 *Data Window* 中可以看到，空元素确实为空：这些元素未显示任何值。

5.4.11 独立子窗口中的指标：大小与层级

到目前为止，我们仅讨论了在主图表窗口中运行的指标，即使用 `#property indicator_chart_window` 指令的指标。现在是时候研究位于价格图表下方独立子窗口中的指标。回顾一下，这类指标应使用 `#property indicator_separate_window` 指令来声明。

我们之前所学的一切同样适用于子窗口指标，具体包括描述和绑定缓冲区、设置绘制类型和样式以及选择使用完整或简化形式的 *OnCalculate* 函数。但子窗口指标也有一些特性和额外设置。

由于子窗口有其自身的值刻度，MQL5 允许设置其最大值和最小值（用户也可以在指标设置对话框的 *Scale* 选项卡中设置类似限制）。这可以通过以下原型的 *IndicatorSetDouble* 函数实现。

```
bool IndicatorSetDouble(ENUM_CUSTOMIND_PROPERTY_DOUBLE property, double value)
bool IndicatorSetDouble(ENUM_CUSTOMIND_PROPERTY_DOUBLE property, int modifier,
double value)
```

该函数用于为指标设置 *double* 特性值。需要两种形式的函数，因为某些特性可以是多个值，特别是水平层级（稍后会详细讨论）。可用特性已收集在 `ENUM_CUSTOMIND_PROPERTY_DOUBLE` 枚举中。

标识符	说明
INDICATOR_MINIMUM	纵轴最小值
INDICATOR_MAXIMUM	纵轴最大值
INDICATOR_LEVELVALUE	水平层级值（通过 <code>modifier</code> 参数设置数值）。

许多振荡指标（如 `WPR`）使用固定刻度范围。我们将通过示例演示本节涉及的所有函数（特性）。

该函数成功时返回 *true*，否则返回 *false*。

正如我们所了解的，除了控制刻度范围以外，子窗口指标还可以设置水平层级。要设置水平层级的数量和特性，需使用另一个函数 *IndicatorSetInteger*。用户可以在指标设置对话框的 *Levels* 选项卡中执行类似操作。

```
bool IndicatorSetInteger(ENUM_CUSTOMIND_PROPERTY_INTEGER property, int value)
bool IndicatorSetInteger(ENUM_CUSTOMIND_PROPERTY_INTEGER property, int modifier,
int value)
```

该函数也有两种形式，支持为指标设置 *int* 类型或等效类型（例如 *color* 或枚举值）的特性值。可用特性已收集在 `ENUM_CUSTOMIND_PROPERTY_INTEGER` 枚举中。除了与水平层级相关的特性外，该枚举还包含对所有类型指标通用的 `INDICATOR_DIGITS` 特性，我们将在下一节中介绍。

标识符	说明
INDICATOR_DIGITS	指标值的显示精度 (小数点后的位数)
INDICATOR_HEIGHT	指标窗口的固定高度, 以像素为单位 (预处理命令 <i>#property indicator_height</i>)
INDICATOR_LEVELS	指标窗口中水平层级的数量
INDICATOR_LEVELCOLOR	水平层级线条颜色 (类型为 <i>color</i> , <i>modifier</i> 参数设置水平层级编号)
INDICATOR_LEVELSTYLE	水平层级线条风格 (类型为 <i>ENUM_LINE_STYLE</i> , <i>modifier</i> 参数设置水平层级编号)
INDICATOR_LEVELWIDTH	水平层级线条粗细 (1-5) (<i>modifier</i> 参数设置水平层级编号)

水平层级可以是文本标签。要分配它们, 使用 *IndicatorSetString*函数。

```
bool IndicatorSetString(ENUM_CUSTOMIND_PROPERTY_STRING property, string value)
bool IndicatorSetString(ENUM_CUSTOMIND_PROPERTY_STRING property, int modifier,
string value)
```

ENUM_CUSTOMIND_PROPERTY_STRING 包含字符串指标参数列表。需注意 INDICATOR_SHORTNAME 特性, 它与水平层级无关: 它是所有指标共有的特性, 我们将在 下一节中介绍。

标识符	说明
INDICATOR_SHORTNAME	指标公开名称
INDICATOR_LEVELTEXT	水平层级说明 (通过 <i>modifier</i> 指定数值)

所有 *int*和 *double* 数值类型的函数都通过特殊指令复制 (下面是汇总表):

水平层级特性的指令	模拟函数	特性类型	说明
indicator_levelN	IndicatorSetDouble(INDICATOR_LEVELVALUE, N-1, value)	double	纵轴上第 N 个水平层级的数值
indicator_levelcolor	IndicatorSetInteger(INDICATOR_LEVELCOLOR, N-1, color)	color	水平层级颜色 (不同编号的水平层级需通过函数设置不同颜色)
indicator_levelwidth	IndicatorSetInteger(INDICATOR_LEVELWIDTH, N-1, width)	int	水平层级线条粗细, 以像素为单位 (不同编号的水平层级需通过函数设置不同粗细)
indicator_levelstyle	IndicatorSetInteger(INDICATOR_LEVELSTYLE, N-1, style)	ENUM_LINE_STYLE	水平层级线条风格 (不同编号的水平层级需通过函数设置不同风格)
indicator_minimum	IndicatorSetDouble(INDICATOR_MINIMUM, minimum)	double	纵轴固定最小值, 刻度下限
indicator_maximum	IndicatorSetDouble(INDICATOR_MAXIMUM, maximum)	double	纵轴固定最大值, 刻度上限

请注意, 使用 *#property*指令时, 特性实例 (修饰符) 的编号从 1 开始, 而函数中使用的编号从 0 开始。

细心的读者会注意到, 某些特性没有对应的指令。它们包括 INDICATOR_LEVELTEXT、INDICATOR_SHORTNAME、INDICATOR_DIGITS。这些特性应根据输入变量以及指标所依附的图表, 通过 MQL 代码动态填充。INDICATOR_LEVELS 可通过为水平层级指定多个指令间接设置。

最后, 子窗口指标的标志是程序能够“冻结”其窗口的垂直大小。

子窗口大小的指令	模拟函数	说明
indicator_height	IndicatorSetInteger(INDICATOR_HEIGHT, height)	指标子窗口的固定高度, 以像素为单位 (用户将无法更改高度)

固定子窗口高度通常仅用于包含通过 图形对象实现的控件 (按钮、标志或输入字段) 的控制面板。

遗憾的是, 特性设置函数没有对应的反函数 (*IndicatorGetInteger*、*IndicatorGetDouble*、*IndicatorGetString*)。除此之外, 这还导致无法实现某些功能, 例如, 当用户修改了水平层级的数量和值时, 无法获取这些信息。

作为使用固定刻度和水平层级的示例，我们来看看 *IndWPR.mq5* 指标。在该指标中，我们将使用标准的 WPR 算法：在指定的历史柱线（WPR 周期）内，找到价格的最高点 H 和最低点 L（即价格区间）。然后，计算当前价格 C 与最低点 L 的差值 $C-L$ （或差值 $-(H-C)$ ，带负号）与整个价格区间的比值，并将结果归一化到 0 至 -100 的范围内。下面是计算 WPR 的标准公式：

$$R\% = (-(H - C) / (H - L)) * 100$$

让我们在源代码开头添加一些指令。除了将指标放置在单独窗口的特性外，我们还将值的范围从 0 设置为 -100。

```
#property indicator_separate_window
#property indicator_maximum 0.0
#property indicator_minimum -100.0
```

一个缓冲区和一个折线图就足以存储数值并显示该指标。

```
#property indicator_buffers 1
#property indicator_plots 1
#property indicator_type1 DRAW_LINE
#property indicator_color1 clrDodgerBlue
```

在 WPR 指标中，通常会划分出两个水平层级：-20 和 -80，分别作为超买区和超卖区的边界。我们来为它们创建一对水平线。

```
#property indicator_level1 -20.0
#property indicator_level2 -80.0
#property indicator_levelstyle STYLE_DOT
#property indicator_levelcolor clrSilver
#property indicator_levelwidth 1
```

唯一的输入变量允许你设置 WPR 的计算周期。

```
input int WPRPeriod = 14; // Period
```

缓冲区数组在全局层面声明，并向 *OnInit* 注册。

```
double WPRBuffer[];

void OnInit()
{
    // check for correct input
    if(WPRPeriod < 1)
    {
        Alert(StringFormat("Incorrect Period value (%d). Should be 1 or larger",
            WPRPeriod));
    }

    // binding array as buffer
    SetIndexBuffer(0, WPRBuffer);
}
```

OnInit 处理程序声明为 *void* 类型，这隐含表示初始化成功。但是，如果周期设置小于 1，则无法进行计算，并向用户发出警告。

为了简化 *OnCalculate* 函数头，我们准备了适用于指标的头文件 *IndCommon.mqh*，其中包含两个宏，用于描述两种形式的事件处理程序的标准参数列表。

```

#define ON_CALCULATE_STD_FULL_PARAM_LIST \
const int rates_total, \
const int prev_calculated, \
const datetime &time[], \
const double &open[], \
const double &high[], \
const double &low[], \
const double &close[], \
const long &tick_volume[], \
const long &volume[], \
const int &spread[]

#define ON_CALCULATE_STD_SHORT_PARAM_LIST \
const int rates_total, \
const int prev_calculated, \
const int begin, \
const double &data[]

```

现在，我们可以该指标和其他指标中使用简洁的 *OnCalculate* 定义（前提是我们接受宏中提议的参数名称）。

```

#include <MQL5Book/IndCommon.mqh>

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
    if(rates_total < WPRPeriod || WPRPeriod < 1) return 0;
    ...
    return rates_total;
}

```

在 *OnCalculate* 的开始部分，我们会检查是否可以使用当前的 *WPRPeriod* 和 *rates_total* 值进行计算。如果数据不足或周期过短，函数将返回 0，这会使指标窗口保持空白。

接下来，我们会用空值填充前几个无法计算给定周期 WPR 的柱线。

```

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
    ...
    if(prev_calculated == 0)
    {
        ArrayFill(WPRBuffer, 0, WPRPeriod - 1, EMPTY_VALUE);
    }
    ...
}

```

最后，我们执行 WPR 计算并将结果存入缓冲区。注意，每笔分时报价都会更新最后一根柱线：这通过从 *prev_calculated - 1* 开始循环实现。

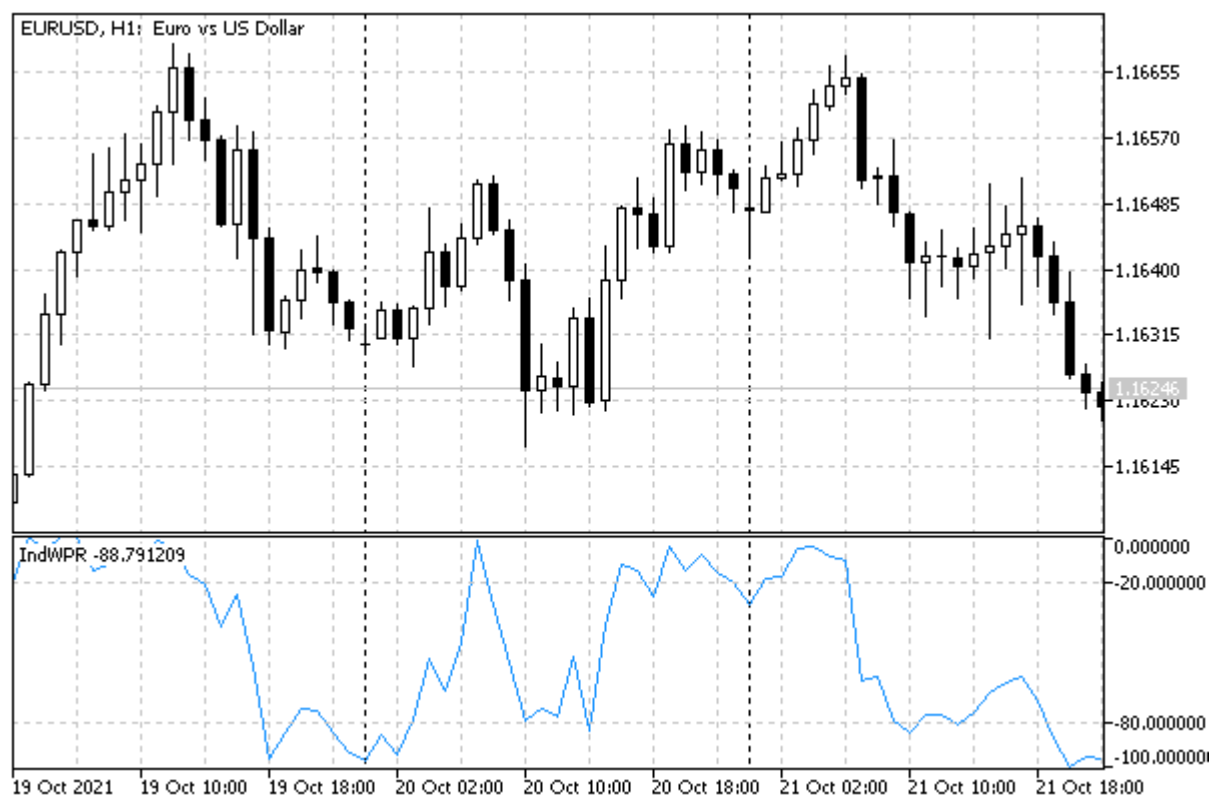
```

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
    ...
    for(int i = fmax(prev_calculated - 1, WPRPeriod - 1);
        i < rates_total && !IsStopped(); i++)
    {
        double max_high = high[fmax(ArrayMaximum(high, i - WPRPeriod + 1, WPRPeriod), 0)];
        double min_low = low[fmax(ArrayMinimum(low, i - WPRPeriod + 1, WPRPeriod), 0)];
        if(max_high != min_low)
        {
            WPRBuffer[i] = -(max_high - close[i]) * 100 / (max_high - min_low);
        }
        else
        {
            WPRBuffer[i] = WPRBuffer[i - 1];
        }
    }
    return rates_total;
}

```

`ArrayMaximum` 和 `ArrayMinimum` 函数允许搜索最高点 `high` 和最低点 `low` 的索引。

该指标会显示在一个独立窗口中，如下所示。



WPR 指标

在接下来的章节中，我们将继续完善该指标，逐步添加其他常用特性。

5.4.12 指标通用特性：标题与数值精度

所有指标都支持一些与计算无关但能提升用户体验的重要特性。在 *OnInit* 处理程序中正确设置这些特性已成为指标开发标准的一部分。

整数特性 `INDICATOR_DIGITS` 可通过之前讨论过的 *IndicatorSetInteger* 函数设置，它会影响图表和 *Data Window* 中实数的显示精度。默认情况下，终端会显示小数点后 6 位数字。如果指标读数与当前金融工具的价格相关，则应将此特性设置为与价格显示精度一致：

IndicatorSetInteger(INDICATOR_DIGITS, _Digits)。

对于 WPR，其值类似于百分比，因此将显示值限制为两位小数更为合理。

```
IndicatorSetInteger(INDICATOR_DIGITS, 2);
```

第二个常用特性是字符串 `INDICATOR_SHORTNAME`，它使用 *IndicatorSetString* 函数。这是指标在工具提示中显示的标题，并且如果指标有自己的窗口，该标题也会显示在子窗口的左上角。如果未明确指定，将使用指标的文件名。例如，在上一节的截图中，我们看到的标题是 `IndWPR`。

通常习惯在指标标题中显示主要输入变量和操作模式（如果有多种模式）。

例如，对于 WPR，通常会在标题中包含用户选择的周期。

此外，标题允许你对名称进行简化，这一点很重要，因为标题长度限制为 63 个字符。

对于 WPR 的更新版本，我们将使用以下设置：

```
IndicatorSetString(INDICATOR_SHORTNAME, "%R" + "(" + (string)WPRPeriod + ")");
```

在下一节中，当我们为超买和超卖区域设置不同颜色后（见示例 *IndColorWPR.mq5*），将检验这些改进的效果。

5.4.13 图表逐项着色

除了前面在 `ENUM_DRAW_TYPE` 中列出的标准绘图类型外，平台还提供了它们的变体，支持对每根柱线上的值进行单独着色。为此，使用一个额外的指标缓冲区来存储颜色编号。这些编号对应于程序员定义的一组颜色的特殊数组中的元素。最大颜色数量为 64。

下表列出了支持颜色功能的 `ENUM_DRAW_TYPE` 元素，以及绘制它们所需的缓冲区数量（包括 1 个用于存储颜色索引的缓冲区）。

可视化类型	说明	缓冲区数量
DRAW_COLOR_LINE	多色线条	1+1
DRAW_COLOR_SECTION	多色分段	1+1
DRAW_COLOR_ARROW	多色箭头	1+1
DRAW_COLOR_HISTOGRAM	零轴多色直方图	1+1
DRAW_COLOR_HISTOGRAM2	基于两个指标缓冲区配对值的多色直方图	2+1
DRAW_COLOR_ZIGZAG	多色锯齿线	2+1
DRAW_COLOR_BARS	多色柱线	4+1
DRAW_COLOR_CANDLES	多色蜡烛图	4+1

在将缓冲区绑定到图表时，请记住，必须在 *SetIndexBuffer* 的第一个参数中，在数据缓冲区编号之后紧接着的编号位置下，指定一个额外的颜色缓冲区。例如，如果要使用一个数据缓冲区和一个颜色缓冲区为线条着色，数据缓冲区编号为 0，其颜色缓冲区编号为 1：

```
double ColorLineData[];
double ColorLineColors[];

void OnInit()
{
    SetIndexBuffer(0, ColorLineData, INDICATOR_DATA);
    SetIndexBuffer(1, ColorLineColors, INDICATOR_COLOR_INDEX);
    PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_COLOR_LINE);
    ...
}
```

图表 N 的调色板中的初始颜色集可通过 *#property indicator_colorN* 指令指定。它要求以逗号分隔的颜色，可使用常量名或颜色字面值指定。例如，指标中的以下条目将为第 0 个图表选择 6 种标准颜色进行着色（在指令中编号从 1 开始）：

```
#property indicator_color1 clrRed,clrBlue,clrGreen,clrYellow,clrMagenta,clrCyan
```

在程序中，不是指定显示图形结构的颜色自身，而是指定颜色的索引。调色板中的编号方式与常规数组相同，从 0 开始。因此，如果需要为第 i 根柱线设置绿色，只需在颜色缓冲区中设置调色板中绿色的索引即可，在本例中为 2。

```
ColorLineColors[i]=2;// reference to element with color clrGreen
```

用于着色的颜色集并非固定不变，可使用 *PlotIndexSetInteger(index, PLOT_LINE_COLOR, color)* 函数动态更改。

例如，要将上述调色板中的 *clrGreen* 颜色替换为 *clrGray*，请使用以下调用：

```
PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrGray);
```

让我们在 WPR 指标中应用着色。新文件为 *IndColorWPR.mq5*。修改涉及以下方面：

缓冲区数量增加了 1 个。三种颜色替代单一颜色。

```
#property indicator_buffers      2
#property indicator_plots       1
#property indicator_type1       DRAW_COLOR_LINE
#property indicator_color1      clrDodgerBlue,clrGreen,clrRed
```

在 *OnInit* 中新增了颜色缓冲区数组及其注册。

```
double WPRColors[];

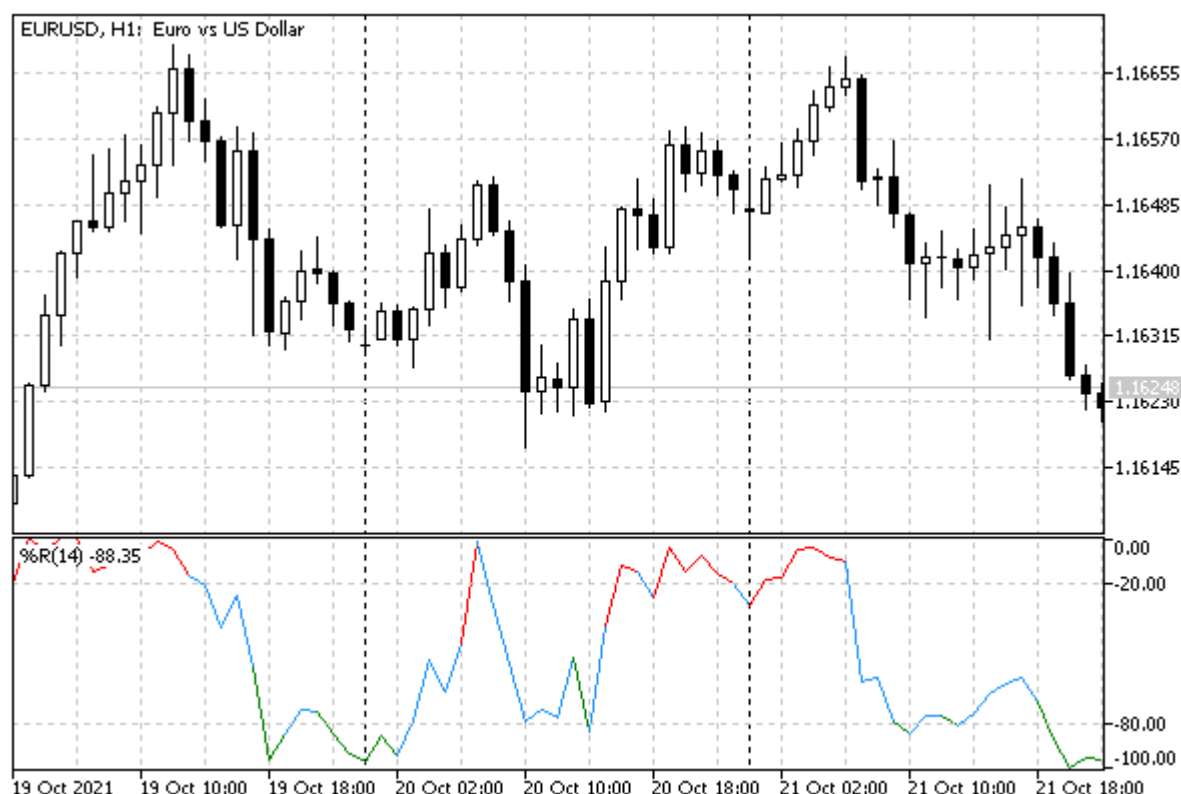
void OnInit()
{
    ...
    SetIndexBuffer(1, WPRColors, INDICATOR_COLOR_INDEX);
    ...
}
```

如果没有设置 *INDICATOR_COLOR_INDEX* 缓冲区类型 (即使用 *SetIndexBuffer(1, WPRColors)* 调用时, 会被默认视为 *INDICATOR_DATA* 类型), 它将在 *Data Window* 中可见。

在 *OnCalculate* 函数的工作循环内, 我们基于对第 *i* 根柱线值的分析添加着色。默认情况下, 我们使用索引为 0 的颜色, 即之前的 *clrDodgerBlue*。如果指标读数进入上方区域, 用颜色 2 (*clrRed*) 高亮显示; 如果进入下方区域, 用颜色 1 (*clrGreen*) 高亮显示。

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
    ...
    for(int i = fmax(prev_calculated - 1, WPRPeriod - 1);
        i < rates_total && !IsStopped(); i++)
    {
        ...
        WPRColors[i] = 0;
        if(WPRBuffer[i] > -20) WPRColors[i] = 2;
        else if(WPRBuffer[i] < -80) WPRColors[i] = 1;
    }
    return rates_total;
}
```

在屏幕上的显示效果如下。



WPR 指标，含彩色超买和超卖区域

请注意，如果线段的最后一个点（柱线）位于上方区域或下方区域，则该线段将以另一种颜色绘制。在这种情况下，先前的读数可能位于中间区域，这可能会让人误以为颜色有误。但这是正确的行为，符合当前实现，也与平台的颜色使用方式一致。

DRAW_COLOR_LINE 线型图表中两个相邻柱线之间的线段颜色由右侧（时间上更新的）柱线颜色决定。

如果你希望仅对相邻两根柱线均处于同一区域的线段进行颜色高亮显示，按以下方案修改代码：

```
WPRColors[i] = 0;
if(WPRBuffer[i] > -20 && WPRBuffer[i - 1] > -20) WPRColors[i] = 2;
else if(WPRBuffer[i] < -80 && WPRBuffer[i - 1] < -80) WPRColors[i] = 1;
```

此外，请记住，我们已在源代码中添加了标题设置和值显示精度（2 位小数）。将新图像与旧图像进行对比，你会注意到以下视觉差异：特别是，标题现在显示为“%R(14)”，且纵向数值轴的显示更为紧凑。

我们要在指标 *IndColorWPR.mq5* 中进行的最后一项修改是：跳过初始柱线的绘制。

5.4.14 跳过初始柱线的绘制

在许多情况下，根据算法的条件，指标值的计算无法从第一根（最左侧可用的）柱线开始，因为需要确保历史中存在指定的最小数量的前置柱线。例如，许多类型的平滑处理意味着，当前值是使用前 N 根柱线的价格数组计算得出的。

在这种情况下，可能无法在最初的几根柱线上计算指标值，或者不打算将这些值显示在图表上，仅作为计算后续值的辅助。

如果要禁止在历史数据的前 $N-1$ 根柱线上显示指标，为相应的图形绘制索引将 `PLOT_DRAW_BEGIN` 特性设置为 `N: PlotIndexSetInteger(index, PLOT_DRAW_BEGIN, N)`。默认情况下，该特性为 0，这意味着数据从最开始处开始显示。

此外，我们可以通过将必要的柱线设置为空值（默认值为 `EMPTY_VALUE`），禁止在这些柱线上显示线条。然而，调用 `PlotIndexSetInteger` 函数对 `PLOT_DRAW_BEGIN` 特性执行其他作用。我们通过这种方式向外部程序告知指标缓冲中前导无效值的数量。具体来说，其他可能基于我们指标的时间序列构建的指标，会在其 `OnCalculate` 处理程序的 `begin` 参数中收到 `PLOT_DRAW_BEGIN` 特性的值。因此，它们将有机会跳过柱线。

在 `IndColorWPR.mq5` 指标的示例中，我们向 `OnInit` 函数添加一个类似的设置。

```
input int WPRPeriod = 14; // Period

void OnInit()
{
    ...
    PlotIndexSetInteger(0, PLOT_DRAW_BEGIN, WPRPeriod - 1);
    ...
}
```

现在，在 `OnCalculate` 函数中，可以移除对最初几根柱线的强制清除，因为它们将始终处于隐藏状态。

```
if(prev_calculated == 0)
{
    ArrayFill(WPRBuffer, 0, WPRPeriod - 1, EMPTY_VALUE);
}
```

但这仅在用户手动将我们的指标选为另一个指标的时间序列来源时，才会正确生效。如果某个程序员决定在其开发中使用我们的指标，则存在另一种获取数据的机制（我们将在下一章讨论这一点），而该机制无法查找 `PLOT_DRAW_BEGIN` 特性。因此，最好使用显式缓冲区初始化。

为了演示如何在借助我们指标数据计算得出的另一个指标中使用该特性，我们来准备另一个指标。这将是封装在 `IndTripleEMA.mq5` 指标中的众所周知的三重指数移动平均线算法。完成后，它将能轻松应用于价格时间序列和任意指标（例如之前的 `IndColorWPR.mq5` 指标）。

此外，我们还将了解描述计算辅助缓冲区（`INDICATOR_CALCULATIONS`）的技术可能性。

三重 EMA 公式由多个计算步骤组成。对初始时间序列 T 进行周期为 P 的简单指数平滑可表示为：

$$K = 2.0 / (P + 1)$$

$$A[i] = T[i] * K + A[i - 1] * (1 - K)$$

其中， K 是用于考虑原始序列元素的权重因子，在给定周期 P 后计算得出； $(1 - K)$ 是应用于平滑序列 A 元素的惯性系数。为获得序列 A 的第 i 个元素，我们将原始序列 $T[i]$ 的第 K 部分与前一个元素 $A[i-1]$ 的 $(1-K)$ 部分相加。

如果我们将上述公式中的平滑处理记为 E 算子，则顾名思义，三重 EMA 包含三次 E 算子的应用，之后将 3 次平滑得到的序列以特殊方式组合。


```

EMA1 = E(A, P), for all i
EMA2 = E(EMA1, P), for all i
EMA3 = E(EMA2, P), for all i
TEMA = 3 * EMA1 - 3 * EMA2 + EMA3, for all i

```

与周期相同的常规 EMA 相比，三重 EMA 对原始序列的滞后更小。然而，其特点是具有更强的响应性，这可能导致生成的曲线出现不规则，并产生虚假信号。

EMA 平滑处理允许从序列的第二个元素开始即可获得平均值的粗略估计，且这不需要更改算法。这使 EMA 有别于其他平滑方法，后者需要 P 个前置元素，或者在可用元素少于 P 个时，需要对初始样本使用修改后的算法。一些开发者甚至在使用 EMA 时，也倾向于将平滑序列的前 $P-1$ 个元素设为无效值。然而，需要注意的是，EMA 公式中序列过去元素的影响并非局限于 P 个元素，只有当元素数量趋近于无穷大时，这种影响才会变得可以忽略不计（而在众所周知的其他 MA 算法中，恰好是前 P 个元素产生影响）。

在本书中，为了研究跳过初始数据的影响，我们不会禁用 EMA 初始值的输出。

要计算三重 EMA 级别，我们需要辅助缓冲区以及一个用于最终序列的缓冲区：后者将以折线图形式显示。

```

#property indicator_chart_window
#property indicator_buffers 4
#property indicator_plots 1

#property indicator_type1 DRAW_LINE
#property indicator_color1 Orange
#property indicator_width1 1
#property indicator_label1 "EMA3"

double TemaBuffer[];
double Ema[];
double EmaOfEma[];
double EmaOfEmaOfEma[];

void OnInit()
{
    ...
    SetIndexBuffer(0, TemaBuffer, INDICATOR_DATA);
    SetIndexBuffer(1, Ema, INDICATOR_CALCULATIONS);
    SetIndexBuffer(2, EmaOfEma, INDICATOR_CALCULATIONS);
    SetIndexBuffer(3, EmaOfEmaOfEma, INDICATOR_CALCULATIONS);
    ...
}

```

输入变量 *InpPeriodEMA* 允许设置平滑周期。第二个变量 *InpHandleBegin* 是一个模式切换开关，通过它，我们可以探究指标在 *OnCalculate* 处理程序中考虑或忽略 *begin* 参数时的反应。可用模式总结在 *BEGIN_POLICY* 枚举中，其含义如下（按排列顺序）：

- 根据初始数据的自定义验证据 *begin*
- 进行严格位移，不进行处理，*begin*
- 即忽略 *begin* 参数并直接计算所有数据

```

enum BEGIN_POLICY
{
    STRICT, // strict
    CUSTOM, // custom
    NONE,   // no
};

input int InpPeriodEMA = 14;           // EMA period:
input BEGIN_POLICY InpHandleBegin = STRICT; // Handle 'begin' parameter:

```

第二种 CUSTOM 模式基于将每个源元素与 EMPTY_VALUE 进行预先比较,并将其替换为适合算法的值。此模式仅适用于以下指标:那些如实初始化缓冲区未使用开头部分,且不遗留垃圾数据的指标。我们的指标 *IndColorWPR* 按要求填充缓冲区,因此在 STRICT 和 CUSTOM 模式下,你可以预期得到几乎相同的结果。

为基于 *InpPeriodEMA* 计算 EMA 准备 K 常数。

```
const double K = 2.0 / (InpPeriodEMA + 1);
```

EMA 函数本身相当简单 (此处省略了 CUSTOM 变体下检查 EMPTY_VALUE 的保护代码段)。

```

void EMA(const double &source[], double &result[], const int pos, const int begin = 0)
{
    ...
    if(pos <= begin)
    {
        result[pos] = source[pos];
    }
    else
    {
        result[pos] = source[pos] * K + result[pos - 1] * (1 - K);
    }
}

```

以下是 *OnCalculate* 中完整的三重平滑计算过程。

```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &price[])
{
    const int _begin = InpHandleBegin == STRICT ? begin : 0;
    // fresh start, or history update
    if(prev_calculated == 0)
    {
        Print("begin=", begin, " ", EnumToString(InpHandleBegin));

        // we can change chart settings dynamically
        PlotIndexSetInteger(0, PLOT_DRAW_BEGIN, _begin);

        // preparing arrays
        ArrayInitialize(Ema, EMPTY_VALUE);
        ArrayInitialize(EmaOfEma, EMPTY_VALUE);
        ArrayInitialize(EmaOfEmaOfEma, EMPTY_VALUE);
        ArrayInitialize(TemaBuffer, EMPTY_VALUE);
        Ema[_begin] = EmaOfEma[_begin] = EmaOfEmaOfEma[_begin] = price[_begin];
    }

    // main loop, taking into account the start from _begin
    for(int i = fmax(prev_calculated - 1, _begin);
        i < rates_total && !IsStopped(); i++)
    {
        EMA(price, Ema, i, _begin);
        EMA(Ema, EmaOfEma, i, _begin);
        EMA(EmaOfEma, EmaOfEmaOfEma, i, _begin);

        if(InpHandleBegin == CUSTOM) // protection from empty elements at the beginning
        {
            if(Ema[i] == EMPTY_VALUE
               || EmaOfEma[i] == EMPTY_VALUE
               || EmaOfEmaOfEma[i] == EMPTY_VALUE)
                continue;
        }

        TemaBuffer[i] = 3 * Ema[i] - 3 * EmaOfEma[i] + EmaOfEmaOfEma[i];
    }
    return rates_total;
}

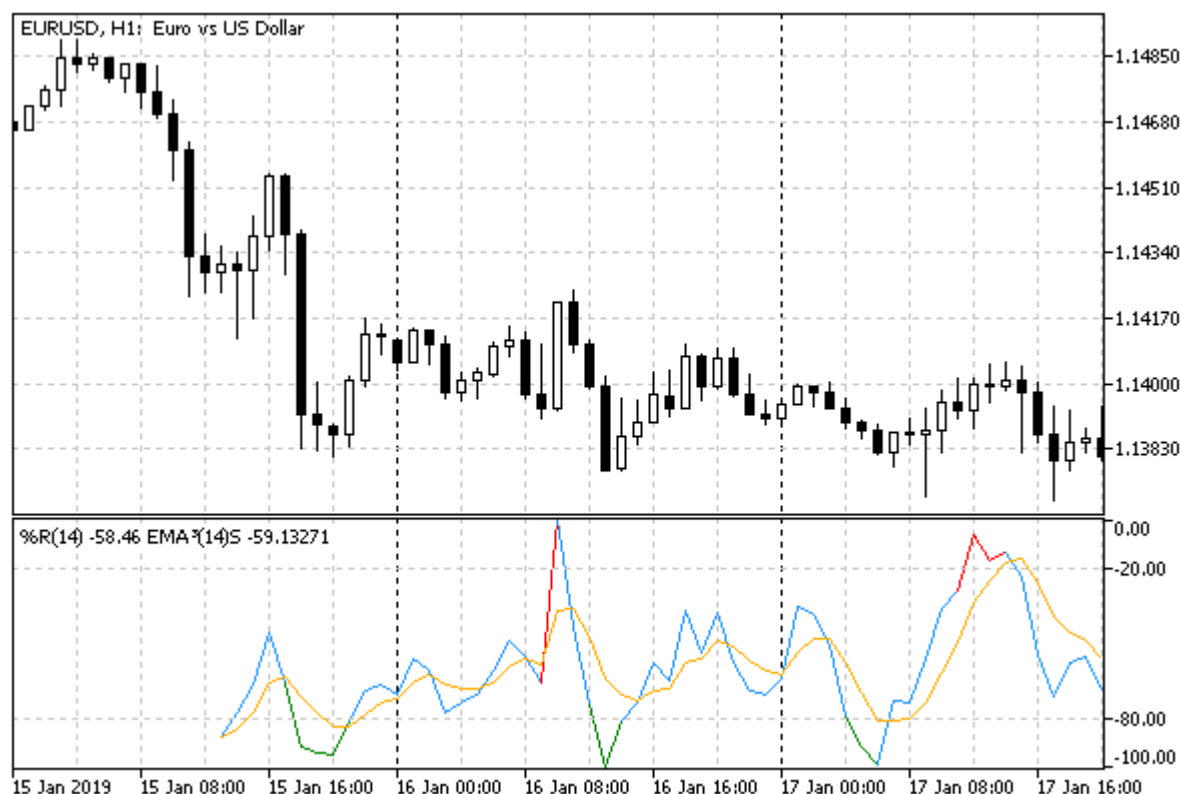
```

第一次启动或更新历史数据时，*begin*参数的接收值将与用户选择的处理模式一同写入日志。

成功编译后，即可开始进行各项实验。

首先，让我们运行*IndColorWPR*指标（默认情况下，其周期为 14，这意味着根据源代码，需将 PLOT_DRAW_BEGIN 特性设置为 13，因为索引从 0 开始，第 13 根柱线将是第一个显示计算值的位置）。然后将*IndTripleEMA*指标拖至显示 WPR 的子窗口中。在弹出的特性设置对话框的 *Options*选项卡上，在 *Apply to*下拉列表中选择 *Previous indicator data*。保留 *Inputs*选项卡的默认值。

下图显示了图表的起始部分。日志将记录以下条目：*begin=13 STRICT*。



考虑数据起点时应用于 WPR 的三重 EMA 指标

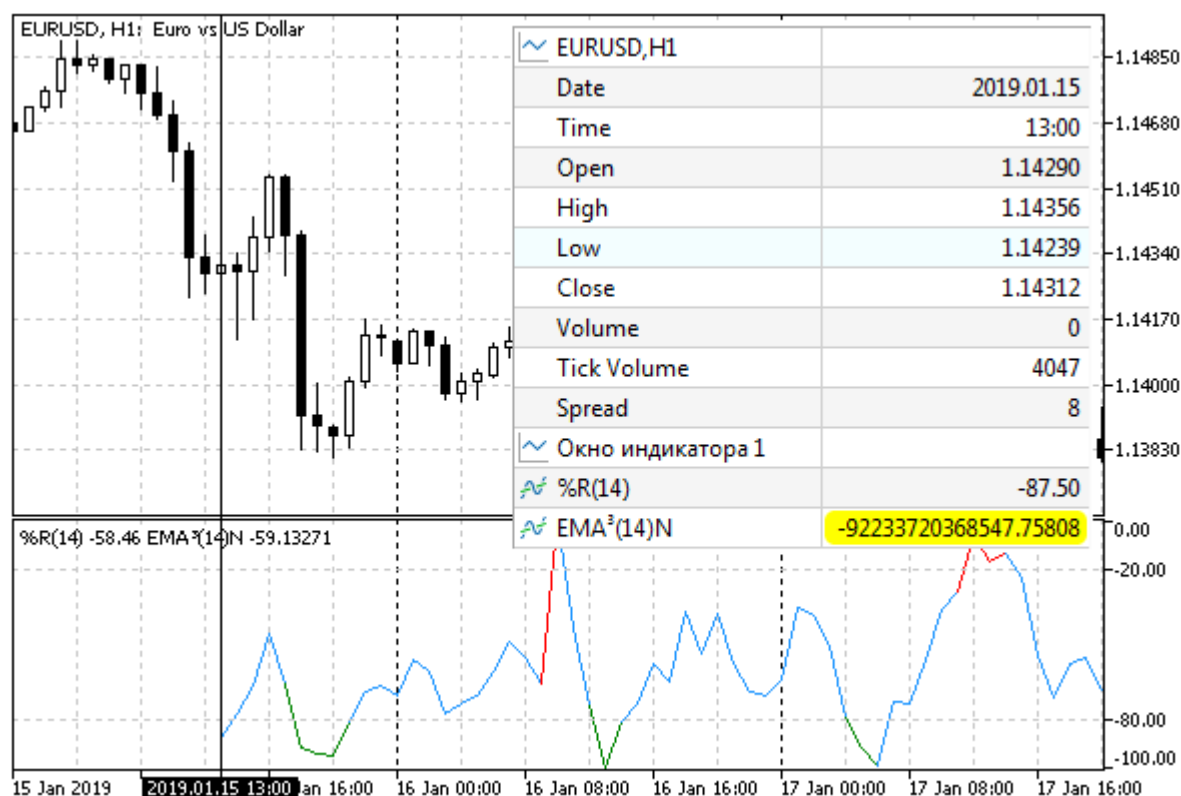
请注意，平均线与 WPR 指标一样，均从与起点有一定距离的位置开始。

注意！如果本地历史报价数据较长，用于指标计算的可用柱线数量 *rates_total* (或 *iBars(_Symbol, _Period)*) 可能会超过终端设置中允许显示的最大柱线数量。在这种情况下，WPR 线 (或任何跳过起始元素的指标，如 MA) 起始处的空值元素将变得不可见，它们将被隐藏在图表的左边界之后。如果要复现初始柱线上没有指标线的情况，你需要增加图表上显示的柱线数量，或关闭终端并删除特定交易品种的本地历史数据。

现在，我们来在 *IndTripleEMA* 指标设置中切换至 CUSTOM 模式 (日志将显示 *begin=0 CUSTOM*)。指标读数应无明显变化。

最后，我们激活 NONE 模式。日志将输出：*begin=0 NONE*。

此时，图表上的情况看起来有些异常，因为指标线实际上会消失。在数据窗口中，你可以看到元素值变得非常大。



未考虑数据起点时应用于 WPR 指标的三重 EMA 指标

这是因为 `EMPTY_VALUE` 的值等于最大实数 `DBL_MAX`。因此，如果不考虑 *begin* 参数，使用这些值进行计算时也会生成非常大的数值。根据计算的具体特性，溢出可能导致我们得到特殊值 `NaN`（非数值，请参阅 [检查实数有效性](#)）。其中一个值 `-nan(ind)` 在图中高亮显示 *Data Window* 已支持输出某些类型的 `NaN`，例如 `“inf”` 和 `“-inf”`，但目前尚不支持 `“nan(ind)”`。正如我们所知，此类 `NaN` 值具有危险性，因为涉及它们的计算将持续生成 `NaN`。如果未生成 `NaN`，则当你向右移动浏览柱线时，大数计算中的“瞬态过程”会逐渐减弱（由于 EMA 公式中的衰减因子 $(1 - K)$ ），结果将趋于稳定并变得合理。如果将图表滚动到当前时间，你会看到正常的三重 EMA 指标线。

考虑 *begin* 参数是一种很好的做法，但这并不能保证数据提供者（如果是第三方指标）正确填写该特性。因此，最好是在你的代码中提供一些保护措施。在该 *IndTripleEMA* 实现中，在初始层面实现了保护措施。

如果我们在价格图表上运行 *IndTripleEMA* 指标，你将始终收到 *begin = 0*，因为价格时间序列从一开始（即使是最早的柱线）就填写了真实数据。

5.4.15 等待数据和管理可见性 (`DRAW_NONE`)

在上一章的 [使用真实分时报价数组](#) 一节 (*MqlTick* 结构体) 中，我们使用了脚本 *SeriesTicksDeltaVolume.mq5* 来计算每根柱线的 *Delta* 成交量。当时，我们将结果显示到日志中，但分析此类技术信息的更便捷、更合理的方式是通过指标实现。在本节中，我们将创建一个这样指标，即 *IndDeltaVolume.mq5*。

在此过程中，我们需要处理指标开发中常见的但在之前示例未涉及的两个因素：

第一个因素是，分时报价数据不会引用终端通过 *OnCalculate* 参数传递给指标的标准价格时间序列。这意味着，指标必须自行请求这些数据，并在等到获取这些数据之后才能在窗口中显示内容。

第二个因素涉及到买入和卖出的成交量通常远大于其 Delta 值的情况，因此在同一窗口中显示时，Delta 值将难以区分。但 Delta 值本身是基准参考值，通常需要结合价格走势进行分析。例如，柱线与 Delta 值成交量可以形成 4 种最典型的组合：

- 阳线柱线 + 正 Delta 值 = 确认上升趋势
- 阴线柱线 + 负 Delta 值 = 确认下降趋势
- 阳线柱线 + 负 Delta 值 = 可能出现向下反转
- 阴线柱线 + 正 Delta 值 = 可能出现向上反转

要显示 Delta 值直方图，我们需要提供一种禁用“大型”直方图（买入量和卖出量）的模式，为此我们将使用 `DRAW_NONE` 类型。它会禁用特定绘图的绘制，并避免其对窗口自动选择的缩放比例产生影响（但会在 *Data Window* 中保留缓冲区）。因此，不考虑大型绘图，剩余的 Delta 值图表将获得更大的自动缩放空间。另一种通过将缓冲区标记为辅助类型（`INDICATOR_CALCULATIONS` 模式）来隐藏缓冲区的方法，将在下一节中讨论。

成交量 Delta 值设计用于分别计算分时报价中的买入量和卖出量，然后我们便可以求出两者的差值。相应地，我们将得到三个时间序列：买入量、卖出量及其差值。由于这些信息与价格刻度不兼容，该指标应显示在独立窗口中，并且我们将选择从零开始的直方图（`DRAW_HISTOGRAM`），以此作为显示这三个时间序列的方式。

根据这一点，让我们在指令中描述指标特性：位置、缓冲区和绘图的数量以及它们的类型。

```
#property indicator_separate_window
#property indicator_buffers 3
#property indicator_plots 3
#property indicator_type1 DRAW_HISTOGRAM
#property indicator_color1 clrBlue
#property indicator_width1 1
#property indicator_label1 "Buy"
#property indicator_type2 DRAW_HISTOGRAM
#property indicator_color2 clrRed
#property indicator_width2 1
#property indicator_label2 "Sell"
#property indicator_type3 DRAW_HISTOGRAM
#property indicator_color3 clrMagenta
#property indicator_width3 3
#property indicator_label3 "Delta"
```

我们将使用前一个脚本中的输入变量。由于分时报价数据量相当庞大，我们会限制历史计算的柱线数量（*BarCount*）。此外，根据特定金融工具的分时报价中是否存在真实成交量，我们可以通过两种不同方式计算 Delta 值，为此我们将使用 *tick type* 参数 `COPY_TICKS`（枚举在头文件 *TickEnum.mqh* 中定义，我们已在脚本中使用过该头文件）。

```
#include <MQL5Book/TickEnum.mqh>

input int BarCount = 100;
input COPY_TICKS TickType = INFO_TICKS;
input bool ShowBuySell = true;
```

在 *OnInit* 处理程序中，我们根据用户选择的 *ShowBuySell* 参数（默认值为 *true*，表示显示全部三个直方图）在 `DRAW_HISTOGRAM` 和 `DRAW_NONE` 模式之间切换前两个直方图的显示模式。请注意，通过 *PlotIndexSetInteger* 进行的动态配置会覆盖使用 *#property* 指令嵌入到可执行文件中的静态设置（在这种情况下，仅覆盖部分设置）。

```

int OnInit()
{
    PlotIndexSetInteger(0, PLOT_DRAW_TYPE, ShowBuySell ? DRAW_HISTOGRAM : DRAW_NONE);
    PlotIndexSetInteger(1, PLOT_DRAW_TYPE, ShowBuySell ? DRAW_HISTOGRAM : DRAW_NONE);

    return INIT_SUCCEEDED;
}

```

但指标缓冲区的注册在何处？我们将在后面几段讨论该问题。现在让我们开始准备 **OnCalculate** 函数。

```

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
    if(prev_calculated == 0)
    {
        // TODO(1): initialization, padding with zeros
    }

    // on each new bar or set of new bars on first run
    if(prev_calculated != rates_total)
    {
        // process all or new bars
        for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
            i < rates_total && !IsStopped(); ++i)
        {
            // TODO(2): try to get the data and calculate the i-th bar,
            // if it doesn't work, do something!
        }
    }
    else // ticks on the current bar
    {
        // TODO(3): updating the current bar
    }

    return rates_total;
}

```

主要技术问题出现在标记为 **TODO (2)** 的代码块中。分时报价请求算法（在脚本中使用，将以最小改动移植到指标中）通过 **CopyTicksRange** 函数请求分时报价。此类调用将返回分时报价数据库中可用的数据。但如果对给定历史柱线尚不可用，该请求将触发分时报价数据的异步下载和同步（即在后台模式下进行）。这种情况下，调用代码收到 0 条分时报价。在这方面，当指标收到这样的“空”响应时，应中断计算并返回失败（而非错误）标志，稍后再重新请求分时报价。在市场正常开市期间，我们会定期收到分时报价，因此 **OnCalculate** 函数可能很快会被再次调用，并基于更新后的分时报价库重新计算。但在周末没有分时报价的情况下该如何处理？

为正确处理这种情况，MQL5 提供了 **计时器**。我们将在后续章节中深入学习计时器，但目前我们将其作为一个“黑盒”使用。特殊函数 **EventSetTimer** 用于“请求”内核在指定秒数后调用我们的 MQL 程序。此类调用的入口点是预留的 **OnTimer** 处理函数，我们在 **事件处理函数概述** 章节的总表中已经见过它。因此，如果在接收分时报价数据时出现延迟，应使用 **EventSetTimer** 启动计时器（最小周期设为 1 秒即可），并从 **OnCalculate** 返回零。

```

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
    ...
    for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
        i < rates_total && !IsStopped(); ++i)
    {
        // TODO(2): try to get the data and calculate the i-th bar,
        if(/*if no data*/)
        {
            Print("No data on bar ", i, ", at ", TimeToString(time[i]),
                ". Setting up timer for refresh...");
            EventSetTimer(1); // please call us in 1 second
            return 0; // don't show anything in the window yet
        }
    }
    ...
}

```

在 *OnTimer* 处理程序中，我们使用 *EventKillTimer* 函数停止计时器（如果不这样做，系统会继续每秒调用我们的处理程序）。此外，我们需要以某种方式启动指标的重新计算。为此，我们将应用另一个函数即 *ChartSetSymbolPeriod*，我们在图表章节尚学习该函数（请参阅 [切换交易品种和时间范围](#) 章节）。该函数允许你为指定标识符的图表设置新的交易品种和时间范围组合（0 表示当前图表）。但如果通过传递 *_Symbol* 和 *_Period*（请参阅 [预定义变量](#)）保持它们不变，将仅执行图表更新（指标会重新计算）。

```

void OnTimer()
{
    EventKillTimer();
    ChartSetSymbolPeriod(0, _Symbol, _Period); // auto-updating of the chart
}

```

在这里，需要注意的另一点是，在开市期间，如果下一个分时报价出现在 *OnTimer* 之前，计时器事件和图表自动更新可能是多余的。因此，我们将创建一个全局变量（*calcDone*）来切换计算就绪的标志。在 *OnCalculate* 开始时，我们会将其重置为 *false*；当计算正常完成时，会将其设置为 *true*。

```

bool calcDone = false;

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
    calcDone = false;
    ...
    if(/*if no data*/)
    {
        ...
        return 0; // exit with calcDone = false
    }
    ...
    calcDone = true;
    return rates_total;
}

```

然后在 *OnTimer* 中，仅当 *calcDone* 等于 *false* 时，才会启动图表自动更新。


```

void OnTimer()
{
    EventKillTimer();
    if(!calcDone)
    {
        ChartSetSymbolPeriod(0, _Symbol, _Period);
    }
}

```

现在来看 *TODO(1,2,3)* 注释部分，我们需要在这里执行计算并填充指标缓冲区。我们将所有这些操作整合到一个名为 *CalcDeltaVolume* 的类中。因此，每个操作都将分配一个单独的方法，同时保持 *OnCalculate* 处理程序简洁（注释将被方法替代）。

在该类中，我们将提供成员变量来接收用户设置的已处理历史柱线数量、增量计算方法，同时还会包含三个指标缓冲区的数组。我们将在构造函数中对它们进行初始化。

```

class CalcDeltaVolume
{
    const int limit;
    const COPY_TICKS tickType;

    double buy[];
    double sell[];
    double delta[];

public:
    CalcDeltaVolume(
        const int bars,
        const COPY_TICKS type)
        : limit(bars), tickType(type), lasttime(0), lastcount(0)
    {
        // register internal arrays as indicator buffers
        SetIndexBuffer(0, buy);
        SetIndexBuffer(1, sell);
        SetIndexBuffer(2, delta);
    }
}

```

由于接下来我们将创建该类的全局对象，因此可以将成员数组指定为缓冲区。为确保数据正确显示，我们只需确保在绘图时附加到图表的数组存在即可。可以动态更改缓冲区绑定（请参阅下一节的 *IndSubChartSimple.mq5* 示例）。

请注意，指标缓冲区必须为 *double* 类型，而成交量为 *ulong* 类型。因此，对于非常大的值（例如在极大时间范围下），理论上可能存在精度损失。

已经创建了 *reset* 方法来初始化缓冲区。大多数数组元素会被填充为空值 *EMPTY_VALUE*，而最后 *limit* 根柱线会被填充为 0，因为我们会在这些位置分别累加买入量和卖出量。

```

void reset()
{
    // fill in the buys array and copy the rest from it
    // empty value in all elements except the last limit bars with 0
    ArrayInitialize(buy, EMPTY_VALUE);
    ArrayFill(buy, ArraySize(buy) - limit, limit, 0);

    // duplicate the initial state into other arrays
    ArrayCopy(sell, buy);
    ArrayCopy(delta, buy);
}

```

第 i 根历史柱线的计算由 `createDeltaBar` 方法执行。该方法的输入参数接收柱线编号以及一个指向包含柱线时间戳的数组引用（我们通过 `OnCalculate` 参数获取该数组）。第 i 个数组元素被初始化为 0。

```

int createDeltaBar(const int i, const datetime &time[])
{
    delta[i] = buy[i] = sell[i] = 0;
    ...
}

```

然后，我们需要确定第 i 根柱线的时间限制：`prev` 和 `next`，其中 `next` 是通过在 `prev` 的基础上累加 `PeriodSeconds` 函数的值来计算的，后者是我们新接触的函数。它返回当前时间范围的秒数。通过累加该数值，我们可以计算出下一根柱线的理论起始时间。在历史数据中，当 i 不等于最后一根柱线编号时，我们本可以用 `time[i + 1]` 来替代计算下一个时间戳。然而，该指标也应适用于仍在形成过程中且没有下一根柱线的最后一根柱线。因此，一般情况下禁止使用 `time[i + 1]`。

```

...
const datetime prev = time[i];
const datetime next = prev + PeriodSeconds();

```

当我们在脚本中进行类似计算时，无需使用 `PeriodSeconds` 函数，因为脚本不考虑最后一根（当前）柱线，因此可以直接通过 `iTime(WorkSymbol, TimeFrame, i)` 和 `iTime(WorkSymbol, TimeFrame, i + 1)` 分别获取 `next` 和 `prev`。

接下来，在 `createDeltaBar` 方法中，我们会在已找到的时间戳范围内请求分时报价（从右侧时间戳减去 1 毫秒，以免触及下一根柱线）。分时报价将到达 `ticks` 数组中，并通过辅助方法 `calc` 进行处理。该方法包含的脚本算法几乎无需修改。我们不得不将其分离为一个专门的方法，因为计算将在两种不同的情况下执行：使用历史柱线（记录注释 `TODO(2)`）和使用当前柱线上的分时报价（注释 `TODO(3)`）。下面我们来考虑第二种情况。

```

ResetLastError();
MqlTick ticks[];
const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL,
    prev * 1000, next * 1000 - 1);
if(n > -1 && _LastError == 0)
{
    calc(i, ticks);
}
else
{
    return _LastError;
}
return n;
}

```

如果请求成功，该方法将返回已处理的分时报价数量；如果发生错误，则返回带负号的错误代码。请注意，如果数据库中尚未有该柱线的分时报价（严格来说这并非错误，但会导致指标无法继续可视化运行），该方法将返回 0 0 的符号不会改变其数值）。因此，在 **OnCalculate** 函数中，我们需要检查该方法的返回值是否“小于或等于”0。

calc方法实际上包含了脚本 *SeriesTicksDeltaVolume.mq5* 中的有效代码行，此处不再赘述。需要回顾的读者可以查阅 *IndDeltaVolume.mq5* 文件了解具体实现方式。

要在不断更新的最后一根柱线上计算 Delta 值，我们需要以毫秒精度记录最后一个已处理分时报价的时间戳。然后，在下次调用 **OnCalculate** 时，我们可以查询该时间戳之后的所有分时报价。

请注意，系统无法保证在每个实时分时报价更新时都能及时调用我们的 **OnCalculate** 处理程序。如果我们执行大量计算，或者其他 MQL 程序使终端计算负载过重，又或者分时报价来得非常快（例如在重要新闻发布后），事件可能无法进入指标队列（队列中每种类型的事件最多存储一个，包括最多一个分时报价通知）。因此，如果程序希望获取所有分时报价，则必须使用 **CopyTicksRange** 或 **CopyTicks** 请求它们。

但仅最后一个已处理分时报价的时间戳是不够的。即使考虑到毫秒级，不同的分时报价也可能具有相同的时间戳。因此，我们不能简单地在时间戳上加 1 毫秒来排除“旧”分时报价：因为相同时间戳的“新”分时报价可能会在其后出现。

为此，你不仅需要记录时间戳，还记录具有该时间戳的最后几个分时报价的数量。然后，下次请求分时报价时，我们可以从记录的时间开始请求（即包括“旧”分时报价），但准确跳过上次已经处理过的数量。

为了实现该算法，类中声明了两个变量：**last time** 和 **last count**。

```

ulong last time; // millisecond marker of the last processed online tick
int last count; // number of ticks with this label at that moment

```

从系统接收的分时报价数组中，我们使用辅助方法 **updateLastTime** 来为这些变量找到合适的值。

```

void updateLastTime(const int n, const MqlTick &ticks[])
{
    lasttime = ticks[n - 1].time_msc;
    lastcount = 0;
    for(int k = n - 1; k >= 0; --k)
    {
        if(ticks[k].time_msc == ticks[n - 1].time_msc) ++lastcount;
    }
}

```

现在我们可以完善 `createDeltaBar` 方法: 在处理最后一根柱线时, 我们第一次调用 `updateLastTime` 方法。

```

int createDeltaBar(const int i, const datetime &time[])
{
    ...
    const int size = ArraySize(time);
    const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL,
        prev * 1000, next * 1000 - 1);
    if(n > -1 && _LastError == 0)
    {
        if(i == size - 1) // last bar
        {
            updateLastTime(n, ticks);
        }
        calc(i, ticks);
    }
    ...
}

```

获得最新的 *last time* 和 *last count* 值后, 我们可以实现一个在线计算当前柱线 Delta 值的方法。

```

int updateLastDelta(const int total)
{
    MqlTick ticks[];
    ResetLastError();
    const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL, lasttime);
    if(n > -1 && _LastError == 0)
    {
        const int skip = lastcount;
        updateLastTime(n, ticks);
        calc(total - 1, ticks, skip);
        return n - skip;
    }
    return _LastError;
}

```

为了实现该模式, 我们在 `calc` 方法中引入了一个额外的可选参数 *skip*。它允许跳过对指定数量的“旧”分时报价的计算。

```

void calc(const int i, const MqlTick &ticks[], const int skip = 0)
{
    const int n = ArraySize(ticks);
    for(int j = skip; j < n; ++j)
        ...
}

```

用于计算的类已准备就绪。现在，我们只需在 *OnCalculate* 中插入对三个公共方法的调用。

```

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
    if(prev_calculated == 0)
    {
        deltas.reset(); // initialization, padding with zeros
    }

    calcDone = false;

    // on each new bar or set of new bars on first run
    if(prev_calculated != rates_total)
    {
        // process all or new bars
        for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
            i < rates_total && !IsStopped(); ++i)
        {
            // try to get data and calculate the i-th bar,
            if((deltas.createDeltaBar(i, time)) <= 0)
            {
                Print("No data on bar ", i, ", at ", TimeToString(time[i]),
                    ". Setting up timer for refresh...");
                EventSetTimer(1); // call us in 1 second
                return 0; // don't show anything in the window yet
            }
        }
    }
    else // ticks on the current bar
    {
        if((deltas.updateLastDelta(rates_total)) <= 0)
        {
            return 0; // error
        }
    }

    calcDone = true;
    return rates_total;
}

```

我们来编译并运行该指标。首先，建议选择不高于 H1 的时间范围，并将 *BarCount* 中的柱线数量默认设置为 100。等待指标构建完成后，结果应大致如下：



包含所有直方图的 Delta 值成交量指标 (包括买入量和卖出量)

现在对比一下将 *ShowBuySell* 参数设置为 *false* 时的情况:



包含单一 Delta 值直方图的成交量指标 (单独的买入量和卖出量已隐藏)

因此，在该指标中，我们通过计时器实现了对当前金融工具分时报价数据加载的等待机制，因为处理分时报价可能需要大量资源。在下一节中，我们将讨论在报价层面运行的多货币指标，对于这类指标，使用 `ChartSetSymbolPeriod` 发起简化的异步图表更新请求即可满足需求。稍后，我们将需要实现另一种等待机制，以确保另一个指标的时间序列准备就绪。

5.4.16 多货币和多时间范围指标

到目前为止，我们讨论的都是处理当前图表交易品种的报价或分时报价的指标。但有时需要分析多个金融工具，或分析与当前金融工具不同的某个金融工具。在这种情况下，正如我们在分时报价分析中所看到的，通过 `OnCalculate` 参数传递给指标的标准时间序列是不够的。需要以某种方式请求“外部”报价，等待它们构建完成，然后才能基于它们计算指标。

请求和构建与当前图表时间范围不同的报价数据，其机制与处理其他交易品种没有区别。因此，在本节中，我们将讨论多货币指标的创建，而多时间范围指标也可以按照类似原则进行创建。

我们需要解决的问题之一是柱线在时间上的同步性。具体来说，不同交易品种可能有不同的交易时段、周末休市安排，总体而言，父图表上的柱线编号可能与“外部”交易品种的报价柱线编号有所不同。

首先，我们先简化任务，将范围限定为任意一个交易品种，该交易品种可能与当前交易品种不同。交易者常常需要同时查看不同交易品种的多个图表（例如相关对中的主导交易品种和跟随交易品种）。我们来创建一个名为 `IndSubChartSimple.mq5` 的指标，用于在子窗口中显示用户选择的交易品种的报价。

IndSubChartSimple

为了复制主图表的外观，在输入参数中，我们将不仅提供交易品种选择，还提供绘图模式：`DRAW_CANDLES`、`DRAW_BARS`、`DRAW_LINE`。前两种模式需要四个缓冲区，它们输出全部四种价格：`Open`、`High`、`Low` 和 `Close`（日式蜡烛图或柱线），而后者使用一个缓冲区显示 `Close` 价格的折线。为了支持所有模式，我们将使用所需的最大缓冲区数量。

```
#property indicator_separate_window
#property indicator_buffers 4
#property indicator_plots 1
#property indicator_type1 DRAW_CANDLES
#property indicator_color1 clrBlue,clrGreen,clrRed // border,bullish,bearish
```

缓冲区数组将根据价格类型命名。

```
double open[];
double high[];
double low[];
double close[];
```

默认情况下启用日式蜡烛图显示。在这种模式下，`MQ5` 允许指定多种颜色，而非单一颜色。在 `#property indicator_colorN` 指令中，这些颜色用逗号隔开。如果有两种颜色，则第一种颜色决定蜡烛图轮廓的颜色，第二种颜色决定填充颜色。如果像示例中一样有三种颜色，则第一种颜色决定轮廓的颜色，第二种和第三种颜色分别决定蜡烛图阳线和阴线的实体颜色。

在专门讨论 图表 的章节中，我们将了解 `ENUM_CHART_MODE` 枚举，它定义了三种可用的图表绘制模式。

ENUM_CHART_MODE 元素	ENUM_DRAW_TYPE 元素
CHART_CANDLES	DRAW_CANDLES
CHART_BARS	DRAW_BARS
CHART_LINE	DRAW_LINE

它们对应我们选择的绘图模式，因为我们特意选择了与标准模式一致的绘图方法。在这里，ENUM_CHART_MODE 使用便捷，因为它只包含我们需要的 3 种元素，与 ENUM_DRAW_TYPE 不同，它包含许多其他绘图方法。

因此，输入变量有以下定义。

```
input string SubSymbol = ""; // Symbol
input ENUM_CHART_MODE Mode = CHART_CANDLES;
```

通过实现一个简单的函数，将 ENUM_CHART_MODE 转换为 ENUM_DRAW_TYPE。

```
ENUM_DRAW_TYPE Mode2Style(const ENUM_CHART_MODE m)
{
    switch(m)
    {
        case CHART_CANDLES: return DRAW_CANDLES;
        case CHART_BARS: return DRAW_BARS;
        case CHART_LINE: return DRAW_LINE;
    }
    return DRAW_NONE;
}
```

SubSymbol 输入参数中的空字符串表示当前图表交易品种。但由于 MQL5 不允许编辑输入变量，我们需要添加一个全局变量来存储实际使用的交易品种，并在 **OnInit** 处理程序中对其赋值。

```
string symbol;
...
int OnInit()
{
    symbol = SubSymbol;
    if(symbol == "") symbol = _Symbol;
    else
    {
        // making sure the symbol exists and is selected in the Market Watch
        if(!SymbolSelect(symbol, true))
        {
            return INIT_PARAMETERS_INCORRECT;
        }
    }
    ...
}
```

我们还需要检查用户输入的交易品种是否存在并将其添加到 **Market SymbolSelect** 函数实现，我们将在关于 **交易品种** 的章节中学习该函数。

Watch: 这可以通过

为了统一管理缓冲区和图表设置，源代码有多个辅助函数：

- `InitBuffer` -设置一个缓冲区
- `InitBuffers` -设置完整的缓冲区集
- `InitPlot` -设置一个图表

单独的函数将注册相同实体时重复执行的多个操作组合在一起。它们也为该指标在 [图表一章](#) 中进一步开发铺平了道路：我们将支持绘图设置的交互式更改，以响应用户对图表的操作（请参阅 [图表显示模式一章](#) 中指标 `IndSubChart.mq5` 的完整版）。

```
void InitBuffer(const int index, double &buffer[],
    const ENUM_INDEXBUFFER_TYPE style = INDICATOR_DATA,
    const bool asSeries = false)
{
    SetIndexBuffer(index, buffer, style);
    ArraySetAsSeries(buffer, asSeries);
}

string InitBuffers(const ENUM_CHART_MODE m)
{
    string title;
    if(m == CHART_LINE)
    {
        InitBuffer(0, close, INDICATOR_DATA, true);
        // hiding all buffers not used for the line chart
        InitBuffer(1, high, INDICATOR_CALCULATIONS, true);
        InitBuffer(2, low, INDICATOR_CALCULATIONS, true);
        InitBuffer(3, open, INDICATOR_CALCULATIONS, true);
        title = symbol + " Close";
    }
    else
    {
        InitBuffer(0, open, INDICATOR_DATA, true);
        InitBuffer(1, high, INDICATOR_DATA, true);
        InitBuffer(2, low, INDICATOR_DATA, true);
        InitBuffer(3, close, INDICATOR_DATA, true);
        title = "# Open;# High;# Low;# Close";
        StringReplace(title, "#", symbol);
    }
    return title;
}
```

请注意，当开启折线图模式时，仅使用 `close` 数组。该数组被分配索引 0。由于 `INDICATOR_CALCULATIONS` 特性，剩余的三个数组对用户完全隐藏。所有四个数组均用于蜡烛图和柱线模式，且其编号遵循 OHLC 标准（这是 `DRAW_CANDLES` 和 `DRAW_BARS` 绘图类型的要求）。所有数组均分配“串行”特性，即索引按从右到左的顺序排列。

`InitBuffers` 函数返回 `Data Window` 中缓冲区的标题。

所有必需的绘图特性均在 `InitPlot` 函数中设置。

```

void InitPlot(const int index, const string name, const int style,
             const int width = -1, const int colorx = -1,
             const double empty = EMPTY_VALUE)
{
    PlotIndexSetInteger(index, PLOT_DRAW_TYPE, style);
    PlotIndexSetString(index, PLOT_LABEL, name);
    PlotIndexSetDouble(index, PLOT_EMPTY_VALUE, empty);
    if(width != -1) PlotIndexSetInteger(index, PLOT_LINE_WIDTH, width);
    if(colorx != -1) PlotIndexSetInteger(index, PLOT_LINE_COLOR, colorx);
}

```

单个图表 (索引为 0) 的初始设置通过 *OnInit* 处理程序中的新函数完成。

```

int OnInit()
{
    ...
    InitPlot(0, InitBuffers(Mode), Mode2Style(Mode));
    IndicatorSetString(INDICATOR_SHORTNAME, "SubChart (" + symbol + ")");
    IndicatorSetInteger(INDICATOR_DIGITS, (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS));

    return INIT_SUCCEEDED;
}

```

尽管在该指标版本中设置仅执行一次, 但它是动态完成的, 会考虑 *mode* 输入参数, 这与 *#property* 指令提供的静态设置不同。未来, 在指标的完整版中, 我们将能够多次调用 *InitPlot* 函数, 动态更改指标的外部显示形式。

缓冲区通过 *OnCalculate* 填充。在最简单的情况下, 当给定交易品种与图表一致时, 我们可以直接使用以下实现方式。

```

int OnCalculate(const int rates_total, const int prev_calculated,
    const datetime &time[],
    const double &op[], const double &hi[], const double &lo[], const double &cl[],
    const long &[], const long &[], const int &[]) // unused
{
    if(prev_calculated == 0) // needs clarification (see further)
    {
        ArrayInitialize(open, EMPTY_VALUE);
        ArrayInitialize(high, EMPTY_VALUE);
        ArrayInitialize(low, EMPTY_VALUE);
        ArrayInitialize(close, EMPTY_VALUE);
    }

    if(_Symbol != symbol)
    {
        // being developed
        ...
    }
    else
    {
        ArraySetAsSeries(op, true);
        ArraySetAsSeries(hi, true);
        ArraySetAsSeries(lo, true);
        ArraySetAsSeries(cl, true);
        for(int i = 0; i < MathMax(rates_total - prev_calculated, 1); ++i)
        {
            open[i] = op[i];
            high[i] = hi[i];
            low[i] = lo[i];
            close[i] = cl[i];
        }
    }

    return rates_total;
}

```

但在处理任意交易品种时，数组参数并不包含所需报价，且可用柱线总数可能有所不同。此外，当第一次将指标添加到图表时，如果没有提前在附近打开该“外部”交易品种的图表窗口，其报价可能根本尚未准备就绪。此外，第三方交易品种的报价将异步加载，因此新的一批柱线数据可能随时“到达”，这需要进行完整的重新计算。

因此，让我们创建用于控制另一个交易品种的柱线数量的变量 `(lastAvailable)`、常量参数 `prev_calculated` 的可编辑“宽隆”，以及报价就绪标志。

```

static bool initialized; // symbol quotes readiness flag
static int lastAvailable; // number of bars for a symbol (and the current timeframe)
int _prev_calculated = prev_calculated; // editable copy of prev_calculated

```

在 `OnCalculate` 的起始位置，我们要添加一项检查，用于判断是否同时出现多个柱线：我们将使用 `lastAvailable` 变量，该变量的值是基于函数上一次正常退出（即成功计算）时 `iBars(symbol, _Period)` 的值进行填充的。一旦加载额外的历史数据，我们应将 `_prev_calculated` 和柱线数量重置为 0，同时移除就绪标志，以重新计算指标。

```

int OnCalculate(const int rates_total, const int prev_calculated,
    const datetime &time[],
    const double &op[], const double &hi[], const double &lo[], const double &cl[],
    const long &[], const long &[], const int &[]) // unused
{
    ...
    if(iBars(symbol, _Period) - lastAvailable > 1)
    {
        // loading additional history or first start
        _prev_calculated = 0;
        initialized = false;
        lastAvailable = 0;
    }

    // then everywhere we use a copy of _prev_calculated
    if(_prev_calculated == 0)
    {
        ArrayInitialize(open, EMPTY_VALUE);
        ArrayInitialize(high, EMPTY_VALUE);
        ArrayInitialize(low, EMPTY_VALUE);
        ArrayInitialize(close, EMPTY_VALUE);
    }

    if(_Symbol != symbol)
    {
        // request quotes and "wait" till they are ready
        ...
        // main calculation (filling buffers)
        ...
    }
    else
    {
        ... // as is
    }
    lastAvailable = iBars(symbol, _Period);
    return rates_total;
}

```

注释中“等待”一词加上引号并非偶然。正如我们所知道的，在指标中我们无法真正地“等待”（以免拖慢终端的界面线程）。相反，如果数据不足，我们应直接退出函数。因此，“等待”意味着等待下一次计算事件：即新的分时报价到来时或响应图表更新请求时。

以下代码将检查报价是否就绪。

```

int OnCalculate(const int rates_total, const int prev_calculated,
    const datetime &time[],
    const double &op[], const double &hi[], const double &lo[], const double &cl[],
    const long &[], const long &[], const int &[]) // unused
{
    ...
    if(_Symbol != symbol)
    {
        if(!initialized)
        {
            Print("Host ", _Symbol, " ", rates_total, " bars up to ", (string)time[0]);
            Print("Updating ", symbol, " ", lastAvailable, " -> ", iBars(symbol, _Period)
                (iBars(symbol, _Period) > 0 ?
                    (string)iTime(symbol, _Period, iBars(symbol, _Period) - 1) : "n/a"),
                "... Please wait");
            if(QuoteRefresh(symbol, _Period, time[0]))
            {
                Print("Done");
                initialized = true;
            }
        }
        else
        {
            // asynchronous request to update the chart
            ChartSetSymbolPeriod(0, _Symbol, _Period);
            return 0; // nothing to show yet
        }
    }
    ...
}

```

主要工作由特殊的 *QuoteRefresh* 函数完成。该函数接收三个参数：所需交易品种、时间范围，以及当前图表上最早（最旧）柱线的时间，我们对更早日期不感兴趣，但请求的交易品种可能没有这么深的历史数据。正因如此，通过一个单独的函数隐藏检查的所有复杂性会很便捷。

该函数会在数据完成下载且同步到可用范围时立即返回 *true*。我们稍后将详细分析其内部结构。

同步完成后，我们使用 *iBarShift* 函数查找同步柱线，并复制它们的 OHLC 值（*iOpen*、*iHigh*、*iLow*、*iClose* 函数）。

```

ArraySetAsSeries(time, true); // go from present to past
for(int i = 0; i < MathMax(rates_total - _prev_calculated, 1); ++i)
{
    int x = iBarShift(symbol, _Period, time[i], true);
    if(x != -1)
    {
        open[i] = iOpen(symbol, _Period, x);
        high[i] = iHigh(symbol, _Period, x);
        low[i] = iLow(symbol, _Period, x);
        close[i] = iClose(symbol, _Period, x);
    }
    else
    {
        open[i] = high[i] = low[i] = close[i] = EMPTY_VALUE;
    }
}

```

表面上看，使用 Copy 函数复制整个价格数组的替代方案似乎更加高效，但这种方法在此处并不适用，因为不同交易品种中相同索引的柱线可能对应完全不同的时间戳。因此，复制后，还需分析日期并在缓冲区内部移动元素，使其与当前图表上的时间对齐。

由于在 *iBarShift* 函数中，*true* 作为最后一个参数传递，因此该函数将查找与柱线的时间精确匹配的值。如果另一个交易品种中没有柱线，我们将得到 -1，并在图表上显示空白值 (EMPTY_VALUE)。

成功完成完整计算后，新的柱线将以经济模式计算，即考虑 *_prev_calculated* 和 *rates_total*。

现在让我们来看看 *QuoteRefresh* 函数。这是一个通用且实用的函数，因此它被放在头文件 *QuoteRefresh.mqh* 中。

在函数的起始位置，我们将检查是否已从指标类型的 MQL 程序中请求当前交易品种和当前时间范围的时间序列。此类请求被禁止，因为指标运行所依赖的“原生”时间序列已由终端构建或已准备就绪：再次请求可能会导致循环或阻塞。因此，我们只需返回同步标志 (SERIES_SYNCHRONIZED)，如果尚未就绪，指标应在后续（下一个分时报价、通过计时器等）检查数据。

```

bool QuoteRefresh(const string asset, const ENUM_TIMEFRAMES period,
    const datetime start)
{
    if(MQL5InfoInteger(MQL5_PROGRAM_TYPE) == PROGRAM_INDICATOR
        && _Symbol == asset && _Period == period)
    {
        return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
    }
    ...
}

```

第二项检查涉及柱线数量：如果它已经等于图表上允许的最大值，则继续下载任何内容没有意义。

```

if(Bars(asset, period) >= TerminalInfoInteger(TERMINAL_MAXBARS))
{
    return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
}
...

```

接下来的代码部分会按顺序从终端请求可用报价的开始日期：

- 通过指定的时间范围 (SERIES_FIRSTDATE)
- 不与终端本地数据库中的时间范围建立关联 (SERIES_TERMINAL_FIRSTDATE)
- 不与服务器上的时间范围建立关联 (SERIES_SERVER_FIRSTDATE)

如果在任何阶段，请求的日期已存在于可用数据区域中，我们将获得 *true* 作为就绪标志。否则，会从终端本地数据库或服务器请求数据，随后构建时间序列（所有这些操作都是异步自动完成的，以响应我们的 *CopyTime* 调用；也可使用其他 *Copy* 函数）。

```

datetime times[1];
datetime first = 0, server = 0;
if(PRTF(SeriesInfoInteger(asset, period, SERIES_FIRSTDATE, first)))
{
    if(first > 0 && first <= start)
    {
        // application data exists, it is already ready or is being prepared
        return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
    }
    else
    if(PRTF(SeriesInfoInteger(asset, period, SERIES_TERMINAL_FIRSTDATE, first)))
    {
        if(first > 0 && first <= start)
        {
            // technical data exists in the terminal database,
            // initiate the construction of a timeseries or immediately get the desired
            return PRTF(CopyTime(asset, period, first, 1, times)) == 1;
        }
        else
        {
            if(PRTF(SeriesInfoInteger(asset, period, SERIES_SERVER_FIRSTDATE, server))
            {
                // technical data exists on the server, let's request it
                if(first > 0 && first < server)
                    PrintFormat(
                        "Warning: %s first date %s on server is less than on terminal ",
                        asset, TimeToString(server), TimeToString(first));
                // you can't ask for more than the server has - so fmax
                return PRTF(CopyTime(asset, period, fmax(start, server), 1, times)) ==
            }
        }
    }
}

return false;
}

```

指标已就绪。我们可以对它进行编译和运行，例如在 EURUSD, H1 图表上，指定 USDRUB 为附加交易品种。日志将显示类似以下内容：

```

Host EURUSD 20001 bars up to 2018.08.09 13:00:00
Updating USDRUB 0 -> 14123 / 2014.12.22 11:00:00... Please wait
SeriesInfoInteger(symbol,period,SERIES_FIRSTDATE,first)=false / HISTORY_NOT_FOUND(440
Host EURUSD 20001 bars up to 2018.08.09 13:00:00
Updating USDRUB 0 -> 14123 / 2014.12.22 11:00:00... Please wait
SeriesInfoInteger(symbol,period,SERIES_FIRSTDATE,first)=true / ok
Done

```

流程完成后（显示“Done”消息），子窗口将显示另一个图表的蜡烛图。



IndSubChartSimple 指标 -DRAW_CANDLES, 包含第三方交易品种报价

需要注意的是, 由于交易时段缩短, USDRUB 的有效柱线仅占据每个日时间间隔的日间部分。

IndUnityPercent

我们在本节要创建的第二个指标是一个真正意义的多货币 (多资产) 指标, 即 **IndUnityPercent.mq5**。其设计用于显示给定金融工具中包含的所有独立货币 (资产) 的相对强弱。例如, 当我们交易包含 EURUSD 和 XAUUSD 两个交易品种的投资组合时, 则美元、欧元和黄金都会会计入该组合的价值, 这些资产中的每种资产都与其他资产存在比值关系。

在每个时间点上, 都会有当前价格, 这些价格可用以下公式表示:

$$\text{EUR} / \text{USD} = \text{EURUSD}$$

$$\text{XAU} / \text{USD} = \text{XAUUSD}$$

其中, 变量 EUR、USD、XAU 分别代表各资产的独立“基准值”, 而 EURUSD 和 XAUUSD 则是常量 (已知报价)。

为求解这些变量, 我们在方程组中再添加一个方程, 将变量的平方和限制为 1 (因此该指标名称的第一个词为“Unity”):

$$\text{EUR} * \text{EUR} + \text{USD} * \text{USD} + \text{XAU} * \text{XAU} = 1$$

变量数量可能更多, 按逻辑可将其表示为 x_i 。请注意, x_0 是所有金融工具共有的必需基础货币。

一般来说, 变量的计算公式如下 (我们将省略其推导过程):

```

x0 = sqrt(1 / (1 + sum(C(xi, x0)2))), i = 1..n
xi = C(xi, x0) * x0, i = 1..n

```

其中, n 为变量总数, $C(x_i, x_0)$ 表示第 i 个交易对的报价。需注意, 变量数量比金融工具数量多 1 个。

由于计算中涉及的报价通常存在显著差异 (例如 EURUSD 和 XAUUSD 的情况), 且仅通过彼此相互表示 (即不参考任何稳定基准), 因此将绝对值转换为百分比变化值是合理的。因此, 当根据上述公式编写算法时, 我们未采用 $C(x_i, x_0)$, 而是采用比值 $C(x_i, x_0)[0] / C(x_i, x_0)[1]$, 其中方括号中的索引表示当前柱线 [0] 和前一柱线 [1]。此外, 为加快计算速度, 可省略平方与开方运算。

为了将线条可视化, 我们会规定货币和指标缓冲区的具体最大允许数量。当然, 如果用户输入的交易品种较少, 计算中可以仅使用其中一部分。但无法动态增大该限制: 需要修改指令并重新编译指标。

```

#define BUF_NUM 15
#property indicator_separate_window
#property indicator_buffers BUF_NUM
#property indicator_plots BUF_NUM

```

在实现该指标的过程中, 我们将一并解决一个棘手的问题。由于将存在许多同类缓冲区, 标准方法是通过“乘法”对其进行大量编码 (即应当避免的“复制粘贴”式编程方法)。

```

double buffer1[];
...
double buffer15[];

void OnInit()
{
    SetIndexBuffer(0, buffer1);
    ...
    SetIndexBuffer(14, buffer15);
}

```

该方法操作繁琐、效率低下, 且容易出错。相反, 我们将应用 OOP。我们将创建一个类, 用于存储指标缓冲区的数组, 并负责对其进行统一设置, 因为我们的缓冲区应保持一致 (颜色除外, 可能还需要为构成当前图表交易品种的货币进行加粗处理, 但这将在用户输入参数后进行调整)。

采用该类后, 我们只需分配其对象数组, 即可自动连接和配置所需数量的指标缓冲区。如图所示, 该方法可通过以下伪代码说明。

```
// "engine" code supporting an array of unified indicator buffers
class Buffer
{
    static int count; // global buffer counter
    double array[];   // array for this buffer
    int cursor;       // pointer of assigned element
public:
    // constructor sets up and connects the array
    Buffer()
    {
        SetIndexBuffer(count++, array);
        ArraySetAsSeries(array, ...);
    }
    // overload to set the number of the element of interest
    Buffer *operator[](int index)
    {
        cursor = index;
        return &this;
    }
    // overload to write value to selected element
    double operator=(double x)
    {
        buffer[cursor] = x;
        return x;
    }
    ...
};

static int Buffer::count;
```

通过运算符重载，我们仍可沿用为缓冲区对象元素赋值的熟悉语法：*buffer[i] = value*。

在指标代码中，无需编写许多行描述单独数组的代码，只需定义一个“多维数组”即可，

```
// indicator code
// construct 15 buffer objects with auto-registration and configuration
Buffer buffers[15];
...
```

实现该机制的类完整版可在文件 *IndBufArray.mqh* 中找到。请注意，它仅支持缓冲区，不支持图表。理想情况下，应通过新增类来扩展类集合，以便创建可直接使用的对象，这些对象可根据特定图表类型，在缓冲区数组中占用所需数量的缓冲区。我们建议你自行研究并补充该文件。特别说明，代码中包含一个管理指标缓冲区数组的类 *BufferArray*，用于创建具有相同特性值（如 *ENUM_INDEXBUFFER_TYPE* 类型、索引方向、空值）的“多维数组”。我们在新指标中的使用方式如下：

```
BufferArray buffers(BUF_NUM, true);
```

此处，构造函数的第一个参数传入所需的缓冲区数量，第二个参数传入时间序列中的索引指示符（下文将详细说明）。

完成此定义后，我们可以在代码任意位置使用便捷表示法，设置第 *i* 个缓冲区中第 *j* 根柱线的值（它使用了缓冲区对象及缓冲区数组中运算符 `[]` 的双重重载）：

```
buffers[i][j] = value;
```

在指标的输入变量中，我们将允许用户指定以逗号隔开的交易品种列表，并限制用于历史计算的柱线数量，以便控制潜在大量金融工具的加载和同步。如果你决定显示全部可用历史，则应识别并采用不同金融工具的最小可用柱线数量，并控制从服务器加载额外历史。

```
input string Instruments = "EURUSD,GBPUSD,USDCHF,USDJPY,AUDUSD,USDCAD,NZDUSD";
input int BarLimit = 500;
```

程序启动时，解析交易品种列表并形成一个大为 *SymbolCount* 的独立 *Symbols* 数组。

```
string Symbols[];
int direction[]; // direct(+1)/reverse(-1) rate to the common currency
int SymbolCount;
```

所有交易品种必须具有相同的基准货币（通常为 USD），以显示相互关联性。根据特定交易品种中的该基准货币是基础货币（即外汇货币对中的第一位货币）还是报价货币（即外汇货币对中的第二位货币），计算时将使用其直接报价或反向报价（1.0/汇率）。该方向将存储在 *Direction* 数组中。

现在，我们来看看执行上述操作的 *InitSymbols* 函数。如果成功解析列表，该函数将返回基准货币的名称。内置 *SymbolInfoString* 可以获取任何金融工具的基础货币和报价货币，我们将在 [金融工具](#) 章节详细学习该函数。

```

string InitSymbols()
{
    SymbolCount = fmin(StringSplit(Instruments, ','), BUF_NUM - 1);
    ArrayResize(Symbols, SymbolCount);
    ArrayResize(Direction, SymbolCount);
    ArrayInitialize(Direction, 0);

    string common = NULL; // common currency

    for(int i = 0; i < SymbolCount; i++)
    {
        // guarantee the presence of the symbol in the Market Review
        if(!SymbolSelect(Symbols[i], true))
        {
            Print("Can't select ", Symbols[i]);
            return NULL;
        }

        // get the currencies that make up the symbol
        string first, second;
        first = SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_BASE);
        second = SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_PROFIT);

        // count the number of inclusions of each currency
        if(first != second)
        {
            workCurrencies.inc(first);
            workCurrencies.inc(second);
        }
        else
        {
            workCurrencies.inc(Symbols[i]);
        }
    }
    ...
}

```

循环使用辅助模板类**MapArray**跟踪每种货币在所有金融工具中出现的次数。该对象在指标的全局层面描述，需要连接头文件 **MapArray.mqh**。

```

#include <MQL5Book/MapArray.mqh>
...
// array of pairs [name; number]
// to calculate currency usage statistics
MapArray<string,int> workCurrencies;
...
string InitSymbols()
{
    ...
}

```

由于该类起到支持作用，这里不做详细描述。你可以查看源代码获取更多细节。关键在于，当你为一个新的货币名称调用其 *inc*方法时，该货币将加入内部数组，且计数器初始值为 1，如果该名称已存在，则计数器加 1。

随后，我们将计数器大于 1 的货币确定为基准货币。在设置正确的情况下，其余货币应正好出现一次。以下是 *InitSymbols*函数的后续部分。

```

...
// find the common currency based on currency usage statistics
for(int i = 0; i < workCurrencies.GetSize(); i++)
{
    if(workCurrencies[i] > 1) // counter greater than 1
    {
        if(common == NULL)
        {
            common = workCurrencies.GetKey(i); // get the name of the i-th currency
        }
        else
        {
            Print("Collision: multiple common symbols");
            return NULL;
        }
    }
}

if(common == NULL) common = workCurrencies.GetKey(0);

// knowing the common currency, determine the "direction" of each symbol
for(int i = 0; i < SymbolCount; i++)
{
    if(SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_PROFIT) == common)
        Direction[i] = +1;
    else if(SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_BASE) == common)
        Direction[i] = -1;
    else
    {
        Print("Ambiguous symbol direction ", Symbols[i], ", defaults used");
        Direction[i] = +1;
    }
}

return common;
}

```

准备好 *InitSymbols* 函数后，我们可以编写 *OnInit* 函数（简化版）。

```

int OnInit()
{
    const string common = InitSymbols();
    if(common == NULL) return INIT_PARAMETERS_INCORRECT;

    string base = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_BASE);
    string profit = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_PROFIT);

    // setting up lines by the number of currencies (number of symbols + 1)
    for(int i = 0; i <= SymbolCount; i++)
    {
        string name = workCurrencies.getKey(i);
        PlotIndexSetString(i, PLOT_LABEL, name);
        PlotIndexSetInteger(i, PLOT_DRAW_TYPE, DRAW_LINE);
        PlotIndexSetInteger(i, PLOT_SHOW_DATA, true);
        PlotIndexSetInteger(i, PLOT_LINE_WIDTH, 1 + (name == base || name == profit));
    }

    // hide extra buffers in the Data Window
    for(int i = SymbolCount + 1; i < BUF_NUM; i++)
    {
        PlotIndexSetInteger(i, PLOT_SHOW_DATA, false);
    }

    // single level at 1.0
    IndicatorSetInteger(INDICATOR_LEVELS, 1);
    IndicatorSetDouble(INDICATOR_LEVELVALUE, 0, 1.0);

    // Name with parameters
    IndicatorSetString(INDICATOR_SHORTNAME,
        "Unity [" + (string)workCurrencies.getSize() + "]");

    // accuracy
    IndicatorSetInteger(INDICATOR_DIGITS, 5);

    return INIT_SUCCEEDED;
}

```

现在我们来了解主事件处理程序 *OnCalculate*。

需特别注意，主循环中柱线的遍历顺序是逆向的，即按照时间序列从当前到过去的顺序处理。这种方式对于多货币指标尤为方便，因为不同交易品种的历史数据深度可能不同，并且从当前柱线开始回溯计算，直至首次出现任一交易品种缺失数据的位置，这样处理更为合理。在这种情况下，循环提前终止不应被视为错误，我们应返回 *rates_total*，以便在图表上显示已计算的最有效的柱线的值。

但在这个简化版 *IndUnityPercent* 指标中，我们不采用上述方式，而是使用一种更简单、更严格的方式：用户必须使用 *BarLimit* 参数定义无条件的历史查询深度。换言之，对于所有交易品种，必须存在直至图表交易品种编号为 *BarLimit* 的柱线时间戳的数据。否则，指标将尝试下载缺失数据。


```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double& price[])
{
    if(prev_calculated == 0)
    {
        buffers.empty(); // delegate total cleanup to the BufferArray class
    }

    // main loop in the direction "as in a timeseries" from the present to the past
    const int limit = MathMin(rates_total - prev_calculated + 1, BarLimit);
    for(int i = 0; i < limit; i++)
    {
        if(!calculate(i))
        {
            EventSetTimer(1); // give 1 more second to upload and prepare data
            return 0; // let's try to recalculate on the next call
        }
    }

    return rates_total;
}

```

Calculate函数 (见下文) 会计算第 *i* 根柱线上所有缓冲区的值。如果缺失数据, 该函数将返回 **false**, 我们将启动一个计时器, 为构建所有所需金融工具的时间序列留出时间。在计时器处理程序中, 我们将以常规方式向终端发送更新图表的请求。

```

void OnTimer()
{
    EventKillTimer();
    ChartSetSymbolPeriod(0, _Symbol, _Period);
}

```

在 **Calculate**函数中, 我们先确定当前柱线和前一根柱线的日期范围, 基于此计算价格变化。

```

bool Calculate(const int bar)
{
    const datetime time0 = iTime(_Symbol, _Period, bar);
    const datetime time1 = iTime(_Symbol, _Period, bar + 1);
    ...
}

```

调用下一个 **CopyClose**函数的特定版本时需要两个日期, 用于指定日期区间。在该指标中, 我们不能使用基于柱线数量的选项, 因为任何交易品种都可能存在任意柱线缺口, 与其他交易品种的缺口情况不同。例如, 如果某个交易品种存在当前柱线 *t* 和前一根柱线 *t-1*, 则可以正确计算价格变化 $Close[t]/Close[t-1]$ 。然而, 在另一个交易品种中, 第 *t* 根柱线可能不存在, 此时请求两根柱线会返回左侧 “最近的” 柱线 (历史方向), 而该历史时间点可能与 “当前” 时间点相距较远 (例如, 如果该交易品种不是 24 小时交易, 可能对应前一交易日的交易时段)。

为避免这种情况, 指标会严格按时间区间请求报价, 如果某个交易品种在该区间内无数据, 则意味着无价格变动。

同时, 可能出现查询返回超过两根柱线的情况, 此时始终取最后两根 (右侧) 柱线作为最有效的柱线。例如, 当该指标加载到 USDRUB,H1 图表时, 会 “识别” 到在每个工作日 17:00 的柱线之后,

是下一个工作日 10:00 的柱线。但对于 EURUSD 等主要外汇货币对，这两个时间点之间会存在 16 根晚间、夜间和早间的 H1 柱线。

```
bool Calculate(const int bar)
{
    ...
    double w[]; // receiving array of quotes (by bar)
    double v[]; // character changes
    ArrayResize(v, SymbolCount);

    // find quote changes for each symbol
    for(int j = 0; j < SymbolCount; j++)
    {
        // try to get at least 2 bars for the j-th symbol,
        // corresponding to two bars of the symbol of the current chart
        int x = CopyClose(Symbols[j], _Period, time0, time1, w);
        if(x < 2)
        {
            // if there are no bars, try to get the previous bar from the past
            if(CopyClose(Symbols[j], _Period, time0, 1, w) != 1)
            {
                return false;
            }
            // then duplicate it as no change indication
            // (in principle, it was possible to write any constant 2 times)
            x = 2;
            ArrayResize(w, 2);
            w[1] = w[0];
        }

        // find the reverse course when needed
        if(Direction[j] == -1)
        {
            w[x - 1] = 1.0 / w[x - 1];
            w[x - 2] = 1.0 / w[x - 2];
        }

        // calculating changes as a ratio of two values
        v[j] = w[x - 1] / w[x - 2]; // last / previous
    }
    ...
}
```

收到变动后，算法将根据前文所述公式进行计算，并将值写入指标缓冲区。

```

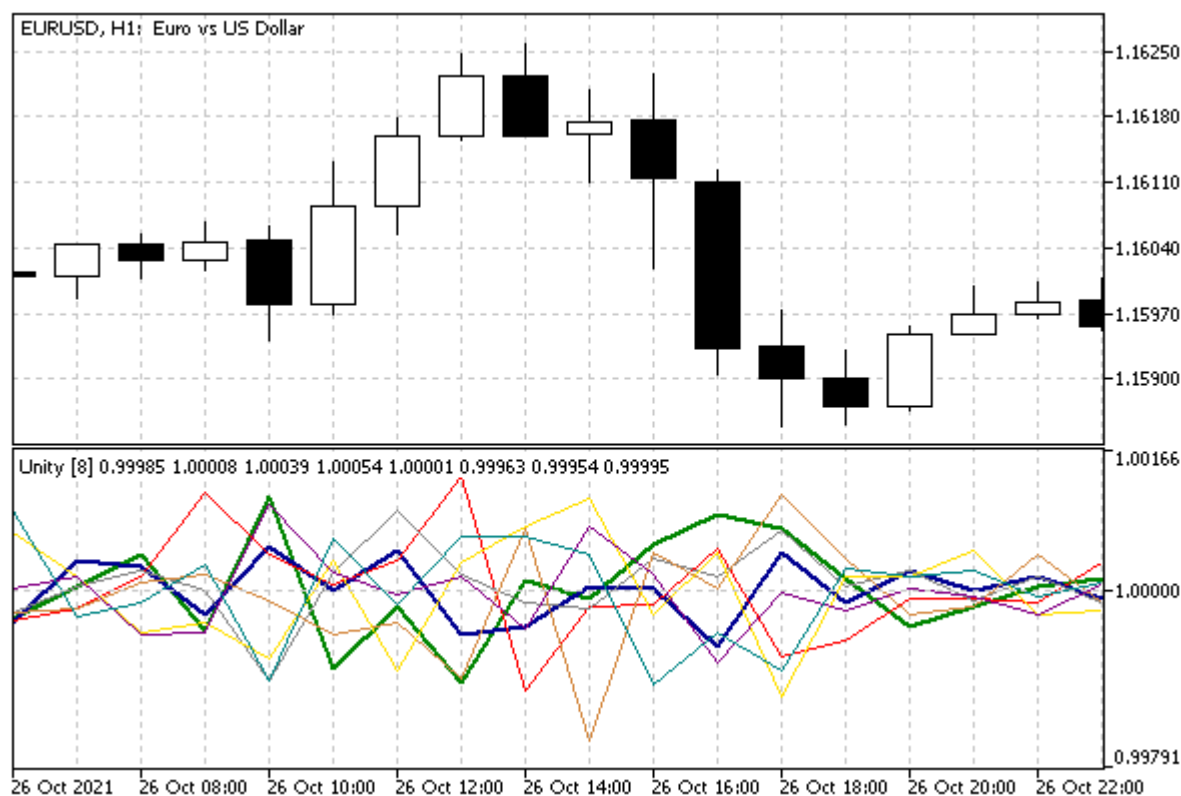
double sum = 1.0;
for(int j = 0; j < SymbolCount; j++)
{
    sum += v[j];
}

const double base_0 = (1.0 / sum);
buffers[0][bar] = base_0 * (SymbolCount + 1);
for(int j = 1; j <= SymbolCount; j++)
{
    buffers[j][bar] = base_0 * v[j - 1] * (SymbolCount + 1);
}

return true;
}

```

让我们看看该指标在默认设置下对一组基础外汇金融工具的表现（第一次加载时，如果尚未打开这些金融工具的图表，获取时间序列数据可能需要较长时间）。



多货币指标 IndUnityPercent (主要外汇币种)

在指标窗口中，两条货币线之间的距离等于对应报价的百分比变动率（两个相邻 *Close* 之间）。这正是指标名称中第二个词 “Percent” 的由来。

在下一章有关指标编程应用的内容中，我们将介绍进阶版的 *IndUnityPercentPro.mq5*，在该版本中，*Copy* 函数将被内置指标 *IMA* 调用替代，从而无需额外操作即可实现任意价格类型的平滑计算。

5.4.17 跟踪柱线形成

上一节讨论的 *IndUnityPercent.mq5* 指标由于使用 *Close* 价，会在每个分时报价时对最后一根柱线重新计算。某些指标和 EA 交易会特别设计为更高效的模式，即每根柱线仅计算一次。例如，我们可以使用开盘价来计算 Unity 公式，在这种情况下跳过分时报价是合理的。检测新柱线的方法有几种：

- ① 记录当前 0 号柱线的时间 通过 *OnCalculate* 的 *time* 参数 - *time[0]*, 或通用 *iTime(symbol, period, 0)*), 并等待其发生变化
- ① 记住柱线数量 *rates_total* (或通过 *iBars(symbol, period)*), 并在数量增加 1 时响应 (如果柱线数量增减变化异常, 可能意味着历史数据被修改)
- ① 等待出现分时报价成交量等于 1 的柱线 (即柱线的第一个分时报价)

然而, 由于该指标的多货币特性, 新柱线形成的概念变得不再那么明确。

在每个交易品种上, 新柱线会在其自身分时报价到达时生成, 而不同交易品种的分时报价到达时间通常不同。在这种情况下, 指标开发者必须确定如何处理: 是等待所有交易品种出现相同时间的柱线后再进行计算, 还是在任意一个交易品种出现新柱线后, 对最新柱线多次重新计算指标。

在本节中, 我们将介绍一个简单的类 *MultiSymbolMonitor* 请参阅文件 *MultiSymbolMonitor.mqh*), 用于根据给定的交易品种列表跟踪新柱线的形成。

所需的时间范围可通过类构造函数传入。默认情况下, 该类会跟踪程序运行所在的当前图表的时间范围。

```
class MultiSymbolMonitor
{
protected:
    ENUM_TIMEFRAMES period;

public:
    MultiSymbolMonitor(): period(_Period) {}
    MultiSymbolMonitor(const ENUM_TIMEFRAMES p): period(p) {}
    ...
}
```

要存储被跟踪交易品种的列表, 我们将使用上一节中的辅助类 *MapArray*。在该数组中, 我们将编写 [symbolname; timestamp of the last bar] 对, 即模板类型 *<string, datetime>*。attach 方法用于填充该数组。

```
protected:
    MapArray<string, datetime> lastTime;
    ...
public:
    void attach(const string symbol)
    {
        lastTime.put(symbol, NULL);
    }
}
```

对于给定数组, 该类可以在 *check* 方法中通过循环调用 *iTime* 函数来更新和检查交易品种的时间戳。

```

ulong check(const bool refresh = false)
{
    ulong flags = 0;
    for(int i = 0; i < lastTime.getSize(); i++)
    {
        const string symbol = lastTime.getKey(i);
        const datetime dt = iTime(symbol, period, 0);

        if(dt != lastTime[symbol]) // are there any changes?
        {
            flags |= 1 << i;
        }

        if(refresh) // update timestamp
        {
            lastTime.put(symbol, dt);
        }
    }
    return flags;
}

```

调用代码应自行决定何时调用 *check* 方法，通常是在分时报价到达时或通过计时器触发。严格来说，这两种方式都无法即时响应其他金融工具的分时报价（及新柱线形成），因为 *OnCalculate* 事件仅在图表当前交易品种的分时报价时触发，如果在两次触发之间出现其他交易品种的分时报价，我们只能等到下一次当前交易品种的分时报价时才会知晓。

我们将在交互式图表事件章节中讨论对多个金融工具分时报价的实时监控（请参阅 [自定义事件生成](#) 章节中的 *EventTickSpy.mq5* 监控指标）。

目前，我们将以可行精度检查柱线。接下来，我们继续讨论 *check* 方法。

每个时间点的特点在于数组中为所有交易品种设置的时间戳状态。例如，流动性最高的金融工具可能在 12:00 形成新柱线，而其他几个金融工具的分时报价可能在几毫秒甚至几秒后才出现。在该时间间隔内，数组中只有一个元素会进行更新，其余元素仍为旧值。随后，所有交易品种会逐渐生成 12:00 的柱线。

对于最后一根柱线的开盘时间与保存时间不一致的所有交易品种，该方法会设置对应交易品种编号的位，从而形成一个包含变化的位掩码。该列表最多只能包含 64 个交易品种。

如果返回值为零，则表示未检测到任何变化。

refresh 参数指定 *check* 方法是仅记录变化 (*false*)，还是根据当前市场情况更新状态 (*true*)。

describe 方法允许通过位掩码获取发生变化的交易品种列表。

```

string describe(ulong flags = 0)
{
    string message = "";
    if(flags == 0) flags = check();
    for(int i = 0; i < lastTime.getSize(); i++)
    {
        if((flags & (1 << i)) != 0)
        {
            message += lastTime.getKey(i) + "\t";
        }
    }
    return message;
}

```

接下来,我们将使用 *inSync* 方法来判断数组中所有交易品种的最后一根柱线时间是否一致。该方法仅适用于交易时段相同的一组货币。

```

bool inSync() const
{
    if(lastTime.getSize() == 0) return false;
    const datetime first = lastTime[0];
    for(int i = 1; i < lastTime.getSize(); i++)
    {
        if(first != lastTime[i]) return false;
    }
    return true;
}

```

利用上述类,我们实现了一个简单的多货币对指标 *IndMultiSymbolMonitor.mq5*, 其唯一任务是检测交易品种列表中的新柱线。

由于未提供该指标的绘图设置,其缓冲区和图表数量为 0。

```

#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots 0

```

金融工具列表由对应的输入变量指定,随后会被转换为数组并注册到 *monitor* 对象中。

```

input string Instruments = "EURUSD,GBPUSD,USDCHF,USDJPY,AUDUSD,USDCAD,NZDUSD";

#include <MQL5Book/MultiSymbolMonitor.mqh>

MultiSymbolMonitor monitor;

void OnInit()
{
    string symbols[];
    const int n = StringSplit(Instruments, ',', symbols);
    for(int i = 0; i < n; ++i)
    {
        monitor.attach(symbols[i]);
    }
}

```

OnCalculate处理程序会在每个分时报价时调用监控，并将状态变化输出到日志中。

```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &price[])
{
    const ulong changes = monitor.check(true);
    if(changes != 0)
    {
        Print("New bar(s) on: ", monitor.describe(changes),
              ", in-sync:", monitor.inSync());
    }
    return rates_total;
}

```

要检查该指标，我们需要在终端中花费大量在线时间。不过，MetaTrader 5 提供了更简便的方式，即借助测试程序。我们将在下一节中讨论这一点。

5.4.18 测试指标

MetaTrader 5 内置测试程序支持两种 MQL 程序：EA 交易和指标。指标始终在可视化窗口中进行测试。但这仅适用于单独测试某个指标的情况。如果指标是通过 EA 交易以编程方式创建和调用的，则用户可以自行决定是否在无可可视化对象的情况下测试该 EA 交易及其指标。我们将在下一章学习如何通过 MQL 代码使用指标的技术。同样的技术也将用于与 EA 交易集成。

同时，指标开发者应注意：在无可可视化对象的情况下，测试程序会对 EA 交易调用的指标使用加速计算方法。不会在每个分时报价出现时都计算数据，而是仅当用户在从指标缓冲区请求相关数据时才会触发计算（请参阅 [CopyBuffer](#) 函数）。

如果尚未根据当前分时报价计算指标，则在第一次访问其数据时会进行一次计算。如果在同一个分时报价内产生其他请求，则计算好的数据将以就绪状态返回。如果当前分时报价未读取指标缓冲区，则不会进行计算。按需计算指标可显著提升测试和优化的速度。

如果特定指标需要精确计算且不可跳过任何分时报价，MQL5 可通过指令要求测试程序对每个分时报价重新计算指标：这可以通过以下指令实现：

#property tester_everytick_calculate

该指令中的 **everytick** 关键词专门用于控制指标的计算方式，不影响分时报价生成模式。换言之，分时报价指的是由测试程序生成的价格变动（无论是分时报价、OHLC M1 价格还是柱线开盘价），该测试程序的设置始终保持有效。

对于本章所讨论的指标而言，该特性并非关键所在。需要特别说明的是，该特性仅适用于策略测试程序中的操作。在终端中，指标会在每个分时报价到来时触发 **OnCalculate** 事件（如果 **OnCalculate** 的计算耗时过长，导致在新分时报价到来前未能完成，则可能会跳过部分分时报价）。

在测试程序中，指标将在以下任一条件下按每个分时报价计算：

- 🕒 在可视化模式下
- 🕒 如果存在 **tester_everytick_calculate** 指令
- 🕒 如果存在 **EventChartCustom** 调用或 **OnChartEvent** 或 **OnTimer** 函数

我们来尝试测试上一节的 **IndMultiSymbolMonitor.mq5** 指标。

选择 EURUSD, H1 图表作为交易品种和时间范围。分时报价生成方法设置为“基于真实分时报价”。

启动测试后，我们应在可视化模式窗口的日志中看到以下条目：

```
2021.10.20 00:00:00 New bar(s) on: EURUSD USDCHF USDJPY , in-sync:false
2021.10.20 00:00:00 New bar(s) on: AUDUSD , in-sync:false
2021.10.20 00:00:00 New bar(s) on: GBPUSD , in-sync:false
2021.10.20 00:00:02 New bar(s) on: USDCAD , in-sync:false
2021.10.20 00:00:11 New bar(s) on: NZDUSD , in-sync:true
2021.10.20 01:00:04 New bar(s) on: EURUSD GBPUSD USDCHF USDJPY AUDUSD USDCAD NZDUSD ,
2021.10.20 02:00:00 New bar(s) on: EURUSD USDJPY NZDUSD , in-sync:false
2021.10.20 02:00:00 New bar(s) on: USDCHF , in-sync:false
2021.10.20 02:00:01 New bar(s) on: AUDUSD , in-sync:false
2021.10.20 02:00:15 New bar(s) on: GBPUSD USDCAD , in-sync:true
2021.10.20 03:00:00 New bar(s) on: EURUSD AUDUSD NZDUSD , in-sync:false
2021.10.20 03:00:00 New bar(s) on: GBPUSD USDJPY USDCAD , in-sync:false
2021.10.20 03:00:12 New bar(s) on: USDCHF , in-sync:true
```

正如你所看到的，不同的交易品种上会逐渐出现新的柱线。通常，在“in-sync”标志设置为 **true** 之前会发生多个事件。

你也可以对本章的其他指标进行测试。请注意：如果 MQL 程序查询分时报价历史数据，在测试程序中选择“基于真实分时报价”的生成方法。

“按开盘价”测试仅适用于开发时支持此模式的指标和 EA 交易，例如，仅通过 **Open** 价进行计算或从第一根柱线开始分析已完成柱线的程序。

注意！在测试程序中进行指标测试时，**OnDeinit** 事件无效。此外，其他终止操作同样不会执行，例如不会调用全局对象的析构函数。

5.4.19 指标的局限性和优点

本章讨论的所有专用函数仅可在指标源代码中使用。在其他类型的 MQL 程序中使用这些函数毫无意义：它们会返回错误。

此外，指标中还存在以下禁用函数：

- ⌚ *OrderCalcMargin*
- ⌚ *OrderCalcProfit*
- ⌚ *OrderCheck*
- ⌚ *OrderSend*
- ⌚ *SendFTP*
- ⌚ *WebRequest*
- ⌚ *Socket****
- ⌚ *Sleep*
- ⌚ *MessageBox*
- ⌚ *ExpertRemove*

其中一些函数（以 *Order-* 为前缀）与交易计算相关，仅允许在 EA 交易和脚本中使用。另一些函数则用于执行会阻塞线程的请求，直到返回结果为止，而指标禁止此类操作，因为它们在终端的界面线程中执行。出于类似原因，*Sleep* 和 *MessageBox* 函数也被禁止。

指标的主要职责是可视化数据，但确实不适合执行大规模运算任务，这确实挺奇怪的。特别是，如果你决定创建一个在运行过程中训练神经网络或决策树的指标，这很可能会对终端的正常运行产生负面影响。

IndBarIndex.mq5 指标展示了长时间计算的影响，该指标在正常模式下用于在其缓冲区元素中显示柱线编号。然而，通过将输入参数 *SimulateCalculation* 设置为 *true*，便可以启动一个基于计时器的无限循环。

```

// Setting to true will freeze the drawing of indicators
// on charts of the same working symbol
// Attention! Don't forget to remove the indicator after the experiment!
input bool SimulateCalculation = false;

void OnInit()
{
    ...
    if(SimulateCalculation)
    {
        EventSetTimer(1);
    }
}
...
void OnTimer()
{
    Comment("Calculation started at ", TimeLocal());
    while(!IsStopped())
    {
        // infinite loop to emulate calculations
    }
    Comment("");
}
}

```

在这种模式下，正如预期的一样，该指标会完全占用 1 个处理器核心，但同时也会出现另一个副作用。在放置了 *IndBarIndex* 指标的同一交易品种上，所有其他指标都会停止更新。例如，我们可以在 EURUSD (任意时间周期) 上运行 *IndBarIndex* 指标，然后在任何其他 EURUSD 图表上尝试应用普通移动平均线：该移动平均线将不会显示，直到从第一个图表中移除 *IndBarIndex* 指标。

因此，所有冗长的计算都应该放在单独的线程中执行，即通过脚本或非交易型 EA 交易进行，而指标中仅使用其计算结果。MQL5 API 允许你创建新的 图表 或 带有图表的对象，并在其中应用包含所需 EA 交易或脚本的 *tpl* 模板。

5.4.20 在 MQL 向导中创建指标草稿

至此，我们已经了解了指标的内部结构，并能够理解源代码中的特定语法结构如何影响指标的外部显示和计算逻辑。通过这一阶段的学习，你可以开始处理他人的代码并对其进行修改以满足自身需求。或者，你也可以尝试创建自己的指标。为了避免从零开始，你可以使用 MQL 向导。特别是，MQL 向导还可以用于创建指标草稿。

要启动向导，在 MetaEditor 的 *Navigator* 的 *Indicators* 分支中调用上下文菜单并运行 *New file* 命令 (Ctrl + N)。在本书第一章的 *MQL 向导和程序草稿* 一节中，我们使用向导创建了第一个脚本，并演示了相关操作流程。

在这种情况下 (从上下文菜单启动时)，向导的第一步将自动选择 *Custom indicator* 项目。

点击 *Next* 进入第二步，此处你需要指定文件名。在这里，你可以 *Add* 指标输入参数。这一步与创建脚本时的操作完全相同。

在第三步中，向导将提供多种 *OnCalculate* 处理程序形式供你选择，同时提供其他可选事件处理程序。

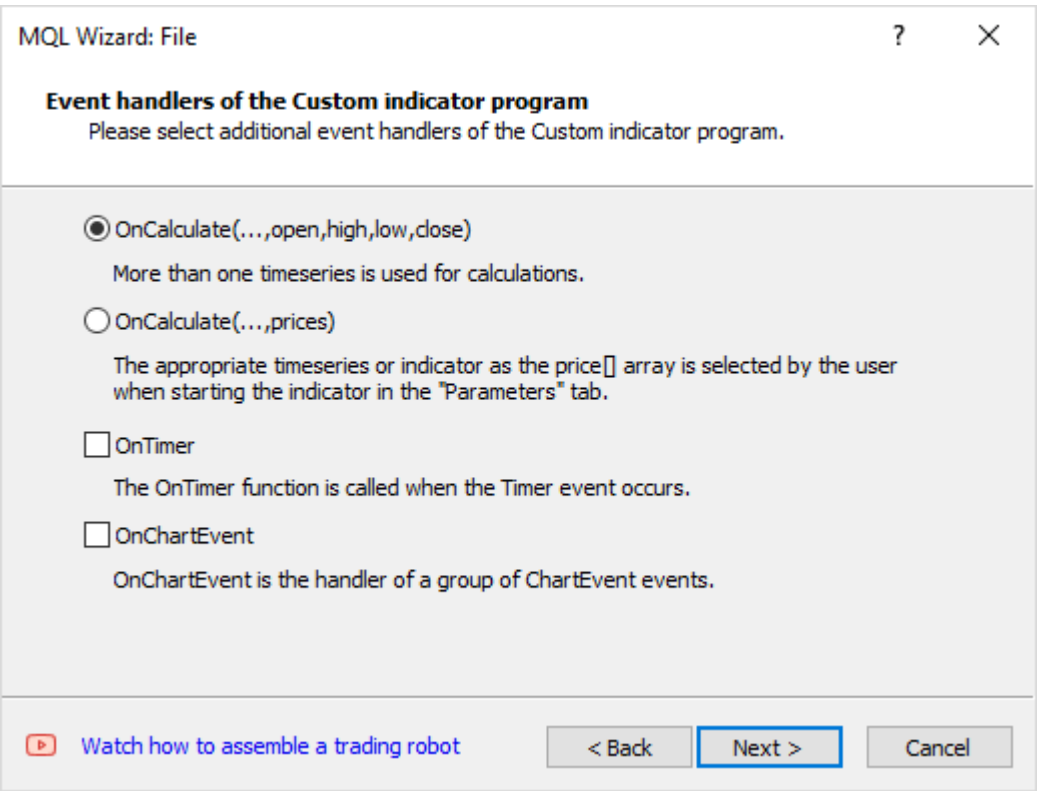


图 5.4 MQL 向导：创建指标时选择事件处理程序

最后一步允许你定义指标线在图表中的显示位置：可以是主窗口（默认选项），也可以是图表下方的独立子窗口（如果启用 *Indicator in a separate window* 标志）。

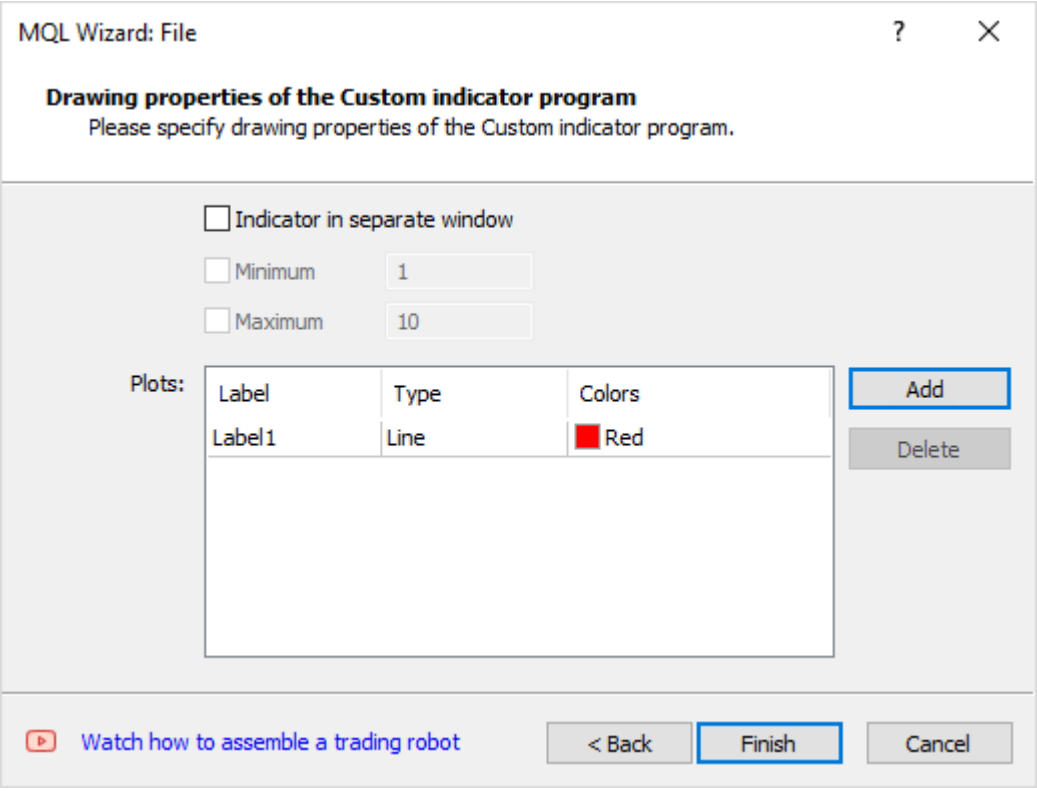


图 5.5 MQL 向导：创建指标时的窗口选择和图表列表

使用 **Add** 按钮，你可以列出多个图形结构并设置它们的基本特性。

所有这些术语我们已从“内部”层面充分理解，你可以自主选择所需选项。

尝试生成多个启用不同选项的指标版本，并评估它们对最终程序文本的影响。

当然，在收到源代码的草稿后，开发者可以自由进行任意修改，更改向导中设置的任何方面。这一点尤为重要，因为向导的设置范围非常有限。特别是，输入参数类型列表仅限于标准 MQL5 类型，没有层级结构、调色板等选项。至于额外的事件处理程序，向导仅提供 `OnTimer` 和 `OnChartEvent`，而将 `OnBookEvent` 和 `OnDeinit` 留作后台处理。但根据本章的内容，你可以逐步为草稿补充所需的所有内容。

5.5 使用来自 MQL 程序的现成指标

在上一章中，我们学习了如何开发自定义指标。用户可以将这些指标加载到图表上，并使用它们进行手动技术分析。但这并非使用指标的唯一方式。MQL5 允许通过编程方式创建指标实例并请求其计算数据。这一功能可通过以下两种途径实现：在将多个简单指标组合成更复杂的指标；在 EA 交易中基于指标信号实现自动或半自动交易。

只要了解指标参数及其计算数据在公共缓冲区中的位置和含义，即可构建这些新应用的时间序列并获取访问权限。

在本章中，我们将学习用于创建和删除指标以及读取其缓冲区的函数。这些知识不仅适用于用 MQL5 编写的自定义指标，还适用于大量内置指标。

与指标进行程序化交互的一般原则包括以下几个步骤：

- ① 创建指标描述符，指标描述符是系统响应特定函数调用（如 `iCustom` 或 `IndicatorCreate`）所分配的唯一标识号，MQL 代码通过指标描述符指定所需指标的名称和参数
- ② 使用 `CopyBuffer` 函数从描述符指定的指标缓冲区读取数据
- ③ 如果不再需要该指标，释放其句柄（`IndicatorRelease`）

创建和释放描述符通常分别在程序初始化和反初始化阶段执行，而缓冲区的读取和分析则会根据需要重复进行（例如在分时报价到达时）。

除了需在程序执行期间动态更改指标设置的特殊情况以外，建议在 `OnInit` 或全局对象类的构造函数中一次性获取指标描述符。

所有指标创建函数至少包含两个参数：交易品种和时间范围。可以传递 `NULL` 替代交易品种，这表示当前金融工具。此外，值 `0` 对应当前时间范围。也可以选择使用内置变量 `_Symbol` 和 `_Period`。如有必要，可设置与图表无关的任意交易品种和时间范围。因此，尤其可以实现多资产和多时间范围指标。

创建指标实例后无法立即访问指标数据，因为缓冲区的计算需要一定时间。读取数据前，应使用 `BarsCalculated` 函数检查数据是否就绪（该函数也接受描述符作为参数，并返回已计算柱线数量）。否则，将收到错误，而非数据。虽然这并非致命错误（不会导致程序停止或卸载），但缺失数据会使程序失去意义。

在本章后续内容中，为简洁起见，我们将创建指标实例并获取其描述符的操作简称为“创建指标”。需将该术语与“创建自定义指标”这一相似术语区分开来，在上一章，“创建自定义指标”指的是编写指标的源代码。

5.5.1 指标所有者的句柄和计数器

在程序中操作指标需要使用句柄进行控制。这与文件描述符的运行机制类似（请参阅 [打开和关闭文件](#) 章节）：通过 **File Open** 函数告知系统文件名和打开模式，之后该描述符将作为访问其他所有文件函数的“通行证”。

指标描述符系统有以下几项用途。

它允许提前向终端告知需要启动的指标和需要计算的时间序列。由于下载初始历史数据及计算指标需要一定时间（至少在第一次请求时需要），同时还需分配资源（内存、图形界面），因此指标创建时间点和就绪时间点并不相同。描述符是这两者之间的桥梁。它是一种指向终端内部对象的链接，该对象存储我们创建指标时设置的特性集合及其当前状态。

当然，为了处理描述符，终端需要维护一个记录所有请求指标及其特性的特定表格。但终端并不提供总表中的实际编号信息：相反，每个程序都会形成自己请求的指标私有列表。该列表中的条目指向总表中的元素，而描述符只是列表中的一个编号。

因此，不同程序中相同的描述符可能对应完全不同的指标。因此，在程序之间传递描述符的值毫无意义。

描述符是终端资源管理系统的一部分，因为它们会尽量避免重复创建具有相同特征的指标实例。换言之，所有内置指标和通过编程方式、手动方式或从 `tpl` 模板创建的自定义指标都会进行缓存。

在创建新的指标实例之前，终端会检查缓存中是否存在完全相同的指标。检查副本时会应用以下标准：

- 匹配交易品种和周期
- 匹配参数

对于自定义指标，还必须额外匹配以下内容：

- 磁盘路径（作为字符串，但不会规范化为绝对路径形式）
- 指标运行所在的图表（当从 MQL 程序创建指标时，被创建的指标会继承创建它的程序所在的图表）

内置指标会按交易品种进行缓存，因此它们的实例可供分配以在不同图表（相同交易品种/时间范围）上单独使用。

请注意，无法在同一图表上手动创建两个完全相同的指标。不同的程序实例可以请求同一个指标，此时只会创建该指标的一个副本，并提供给所有发出请求的程序。

对于每个独特的条件组合，终端会维护一个计数器：第一次请求创建某个特定指标后，其计数器值为 1；后续每请求一次，计数器值加 1（不会创建新的指标副本）。当某个指标被释放时，其计数器值减 1。只有当计数器重置时，即所有所有者明确不再使用指标时，指标才会进行卸载。

要注意的是，在同一个 MQL 程序中多次调用相同参数（包括交易品种/时间范围）的指标构造函数，不会导致引用计数器多次递增，计数器只会递增一次。因此，对于每个句柄值，调用一次释放函数（**IndicatorRelease**）即可。所有后续调用都是多余的，且会返回错误，因为已没有可释放的指标。

除在 MQL5 中通过 **iCustom** 和 **IndicatorCreate** 创建指标外，还可以获取第三方（已存在）指标的句柄。这可以通过 **ChartIndicatorGet** 函数实现，我们将在 [图表](#) 一章中学习该函数。需要注意的是，通过这种方式获取句柄也会增加其引用计数，并阻止指标卸载，除非后续释放该句柄。

如果程序创建了从属指标，则当该程序进行卸载时，这些指标的句柄将自动释放（计数器减1），即使没有调用 *IndicatorRelease* 函数。

5.5.2 创建指标实例的简单方法：iCustom

MQL5 提供了以下两个用于从程序中创建指标实例的函数：*iCustom*和*IndicatorCreate*。第一个函数涉及传递参数列表，这些参数必须在程序编译时就已知。第二个函数允许在程序执行过程中利用所调用指标的参数动态形成数组。这种高级模式将在 [指标创建的高级方式：IndicatorCreate](#) 章节中讨论。

int iCustom(const string symbol, ENUM_TIMEFRAMES timeframe, const string pathname, ...)

该函数为指定交易品种和时间范围创建一个指标。在 *symbol* 参数中，NULL 可用于表示当前图表的交易品种，而在 *timeframe* 中，0 用于设置当前周期。

在 *pathname* 参数中，指定指标名称（不含扩展名的 .ex5 文件名），也可选择性指定路径。路径的更多细节详见下文说明。

pathname 引用的指标必须已编译。

该函数将返回指标句柄，如果发生错误，则返回 INVALID_HANDLE。要调用本章所述且属于指标程序控制组的其他函数时，需要使用该句柄。该句柄是一个整数，用于在调用程序中唯一标识所创建的指标实例。

iCustom 函数原型中的省略号表示该指标的实际参数列表。这些参数的类型和顺序必须与（指标代码中）的形式参数一致。但允许从参数列表末尾开始省略部分值。对于调用代码中未指定的参数，创建的指标将使用对应 *input* 参数的默认值。

例如，如果指标接受两个输入变量：周期（*input int WorkPeriod = 14*）和价格类型（*input ENUM_APPLIED_PRICE WorkPrice = PRICE_CLOSE*），则可以通过以下不同详细程度调用 *iCustom*：

① *iCustom* (_Symbol, _Period, 21, PRICE_TYPICAL): 为整个列表的参数设置值

② *iCustom* (_Symbol, _Period, 21): 设置第一个参数，第二个参数被省略，其将接收值 PRICE_CLOSE

③ *iCustom* (_Symbol, _Period): 两个参数均被省略，将获取 14 和 PRICE_CLOSE

不能省略参数列表开头或中间的参数。

如果创建的指标采用简版 *OnCalculate*，则最后一个额外参数（除指标内部定义的输入变量列表外）可以是用于构建指标的价格类型。这类似于指标特性对话框中的 *Apply to* 下拉列表。此外，在该额外参数中，你可以传递一个先前创建的其他指标的句柄（见下文示例）。在这种情况下，新创建的指标将使用指定句柄的第一个指标缓冲区进行计算。换言之，程序员可以设置一个指标基于另一个指标进行计算。

MQL5 未提供编程方式来查询特定第三方指标是采用简版还是完整版 *OnCalculate* 实现，即在通过 *iCustom* 创建指标时是否允许传递额外句柄。此外，如果通过额外句柄指定的指标包含多个缓冲区，MQL5 不允许选择缓冲区编号。

我们回到 *pathname* 参数。

路径是包含至少一个反斜杠 (\) 或正斜杠 (/) 的字符串，它们是文件系统中用作文件夹和文件层级结构分隔符的特殊字符。你可以使用正斜杠或反斜杠，但反斜杠需要进行“转义”，即必须书写两次。这是因为反斜杠是一种控制字符，可构成许多服务代码，例如制表符 (\t)、换行符 (\n) 等等（请参阅 [字符类型](#) 章节）。

如果路径以斜杠开头,则称为绝对路径,其根目录为所有 MQL5 源代码的目录。例如,在 *pathname* 参数中指定字符串 `"/MyIndicator"`,将查找 `MQL5/MyIndicator.ex5` 文件,带目录 `"/Exercise/MyIndicator"` 的较长路径则对应 `MQL5/Exercise/MyIndicator.ex5`。

如果 *pathname* 参数包含一个或多个斜杠但不以斜杠开头,则该路径称为相对路径,因为此时它被视为相对于两个预定义位置之一。首先,会从调用 MQL 程序所在文件夹开始搜索指标文件。如果未找到,则继续在指标公共文件夹 `MQL5/Indicators` 中搜索。

在包含斜杠的路径中,最右侧斜杠右侧的部分被视为文件名,之前的部分构成文件夹层级结构。例如,路径 `"Folder/SubFolder/FileName"` 对应两个子文件夹: *Folder* 中的 *SubFolder*, 以及 *SubFolder* 中的 *FileName* 文件。

最简单的情况是 *pathname* 参数不包含任何斜杠。这种情况下,它仅指定文件名。它同样会基于上述两个搜索起点进行解析。

例如: *MyExpert.ex5EA* 交易位于文件夹 `MQL5/Experts/Examples` 中,其中包含 `iCustom(_Symbol, _Period, "MyIndicator")` 调用。此处相对路径简化为(空路径),仅包含文件名。因此,指标搜索将从 `MQL5/Experts/Examples/` 文件夹开始,结合文件名 *MyIndicator*, 构成路径 `MQL5/Experts/Examples/MyIndicator.ex5`。如果在此目录中未找到该指标,将继续在指标根文件夹中搜索,即通过具体路径和文件名 `MQL5/Indicators/MyIndicator.ex5` 查找。

如果在这两个位置均未找到指标,函数将返回 `INVALID_HANDLE`,并将错误码 `4802 (ERR_INDICATOR_CANNOT_CREATE)` 设置为 `_LastError`。

更复杂的情况是,如果 *pathname* 不仅包含文件名,还包含目录,例如 `"TradeSignals/MyIndicator"`。此时,指定路径会被附加到调用程序所在文件夹中,形成以下搜索路径:

`MQL5/Experts/Examples/TradeSignals/MyIndicator.ex5`。如果搜索失败,相同路径会被附加到 `MQL5/Indicators`,即搜索文件 `MQL5/Indicators/TradeSignals/MyIndicator.ex5`。请注意,如果使用反斜杠作为分隔符,应写成双反斜杠,例如 `iCustom(_Symbol, _Period, "TradeSignals\\MyIndicator")`。

要释放不再使用的指标占用的计算机内存,可使用 *IndicatorRelease* 数并传入该指标的句柄。

在测试使用指标的程序时需特别注意。如果在 *iCustom* 调用中, *pathname* 参数被指定为常量字符串,编译器将自动检测所需指标,并将其与被测试程序一并传递给测试程序。否则,如果该参数通过表达式计算或从外部获取(例如用户 *input*),则必须在源代码中通过 `#property tester_indicator` 指定特性:

```
#property tester_indicator "indicator_name.ex5"
```

这意味着程序中仅能测试预先已知的自定义指标。

以新指标 *UseWPR1.mq5* 为例,其 *OnInit* 处理程序会创建上一章讨论的 *IndWPR* 指标的句柄(务必编译 *IndWPR*, 因为 *iCustom* 会下载 `ex5` 文件)。*UseWPR1* 中收到的处理程序尚未使用,我们仅研究获取处理程序的可行性并验证操作是否成功。因此,新指标无需设置缓冲区。

```
#property indicator_separate_window
#property indicator_buffers 0
#property indicator_plots 0
```

该指标将创建空白子窗口,但暂时不显示任何内容。这属正常行为。

我们来测试几种获取描述符的方式,对应不同的 *pathname* 值:

1. 绝对路径,以斜杠开头且包含完整的文件夹层级结构(从 MQL5 根目录开始),此处以第五章指标为示例: `"/Indicators/MQL5Book/p5/IndWPR"`

2. 仅名称 “IndW PR”，用于在调用指标 *UseWPR1.mq5* 所在的同一文件夹中搜索（两个指标位于同一文件夹中）
3. 相对标准目录 *MT5/Indicators* 的指标示例的文件夹层级路径，即 “*MT5\Book\p5\IndW PR*”（注意开头无斜杠）
4. 仅名称，与第 2 点类似，但使用不存在的指标名称 “IndW PR NonExistent”
5. 绝对路径，与第 1 点类似，但使用未转义的反斜杠，即 “*\Indicators\MT5\Book\p5\IndW PR*”
6. 完全等同于第 2 点。

```
int OnInit()
{
    int handle1 = PRTF(iCustom(_Symbol, _Period, "/Indicators/MT5\Book\p5\IndWPR"));
    int handle2 = PRTF(iCustom(_Symbol, _Period, "IndWPR"));
    int handle3 = PRTF(iCustom(_Symbol, _Period, "MT5\Book\p5\IndWPR"));
    int handle4 = PRTF(iCustom(_Symbol, _Period, "IndWPR NonExistent"));
    int handle5 = PRTF(iCustom(_Symbol, _Period, "\Indicators\MT5\Book\p5\IndWPR"));
    int handle6 = PRTF(iCustom(_Symbol, _Period, "IndWPR"));
    return INIT_SUCCEEDED;
}
```

由于句柄变量未被使用，因此将其声明为局部变量。需特别说明，虽然局部 *handle* 变量在 *OnInit* 函数退出时被删除，但这并不影响句柄本身：只要“父”指标 *UseWPR* 在执行，句柄就会持续存在。我们只是在代码中丢失了这些句柄的值，但这没问题，因为此处未在任何地方使用它们。在后续将讨论的实际指标示例中，句柄一定会被存储（通常存储在全局变量中）并使用。

也无需担心资源泄漏：当从图表中删除 *UseWPR* 指标时，终端将自动清除其创建的所有句柄。关于句柄显式释放的原则及必要性，将在通过 *IndicatorRelease* 删除指标实例 章节中详细说明。

上述 *OnInit* 代码将生成以下日志条目：

```
iCustom(_Symbol,_Period,/Indicators/MT5\Book\p5\IndWPR)=10 / ok
iCustom(_Symbol,_Period,IndWPR)=11 / ok
iCustom(_Symbol,_Period,MT5\Book\p5\IndWPR)=12 / ok
cannot load custom indicator 'IndWPR NonExistent' [4802]
iCustom(_Symbol,_Period,IndWPR NonExistent)=-1 / INDICATOR_CANNOT_CREATE(4802)
iCustom(_Symbol,_Period,\Indicators\MT5\Book\p5\IndWPR)=13 / ok
iCustom(_Symbol,_Period,IndWPR)=11 / ok
```

如我们所看到，除了第 4 种情况（调用不存在的指标）外，其他情况均会收到有效句柄值 10、11、12 和 13。句柄值为 -1（INVALID_HANDLE）。

另外需要注意，第 5 行代码在编译时会生成多条“无法识别的字符转义序列”警告。这是由于我们未对反斜杠进行转义处理而导致的。我们也很幸运，指令执行成功，因为如果任何文件夹或文件的名称以支持的转义序列中的某个字母开头，则序列的解析将会违背名称的预期读取。例如，如果同一文件夹中有个名为“test”的指标，而我们尝试通过路径“*MT5\Book\p5\test*”创建该指标，我们将会收到 INVALID_HANDLE 和错误码 4802。这是因为“\t”是制表符，因此终端会将查找“*MT5\Book\p5<nbsp>est*”。正确条目应为“*MT5\Book\p5\\test*”。因此，使用正斜杠更为简便。

还需要注意的是，虽然所有成功的变体都引用同一个指标

MT5/Indicators/MT5\Book\p5\IndWPR.ex5，且实际上路径 1、2、3 和 5 是等效的，但终端仍将它们视为不同的字符串，这就是我们得到不同描述符值的原因。只有当选项 6 完全复制选项 2 时，才会返回相同的描述符值 11。

为什么句柄编号从 10 开始？更小的数值是为系统保留的。如上所述，对于使用简版 **OnCalculate** 的指标，最后一个参数可用于传递价格类型或另一个指标的句柄，该指标的缓冲区将被用于计算新创建的指标实例。由于 `ENUM_APPLIED_PRICE` 枚举元素拥有各自的常量值，它们占据 10 以下的区域。更多详情，请参阅 [定义指标的数据源](#)。

在接下来的 **UseWPR2.mq5** 示例中，我们将实现一个指标：该指标将创建 **IndWPR** 的实例，并通过句柄检查其计算进度。但在此之前，你需要先了解一个新函数 **BarsCalculated**。

5.5.3 检查计算得出柱线的数量: **BarsCalculated**

当我们通过调用 **iCustom** 或本章后续将介绍的其他函数创建第三方指标时，指标需要一定时间进行计算。我们知道，指标数据就绪状态的主要衡量标准是已计算柱线数量，这一数值由指标的 **OnCalculate** 函数返回。通过指标句柄，我们可以获取这一数值。

int BarsCalculated(int handle)

该函数返回由 **handle** 指定的指标中计算数据的柱线数量。如果发生错误，我们将收到 -1。

当数据尚未计算时，结果为 0。后续需将此数值与时间序列的大小进行比较（例如，如果调用指标在自身 **OnCalculate** 函数上下文校验 **BarsCalculated**，可对比 **rates_total**），以分析指标对新柱线的处理情况。

在 **UseWPR2.mq5** 指标中，我们将尝试通过在输入参数更改 WPR 周期来创建 **IndWPR**。

```
input int WPRPeriod = 0;
```

其默认值为 0，这是一个无效值。我们有意设计这种情况，为的就是演示异常处理情况。回顾 **IndWPR.mq5** 的源代码，**OnInit** 和 **OnCalculate** 函数中均包含校验。

```
// IndWPR.mq5
void OnInit()
{
    if(WPRPeriod < 1)
    {
        Alert(StringFormat("Incorrect Period value (%d). Should be 1 or larger",
            WPRPeriod));
    }
    ...
}

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
    if(rates_total < WPRPeriod || WPRPeriod < 1) return 0;
    ...
}
```

因此，当周期为零时，我们应收到错误消息，且 **BarsCalculated** 应始终返回 0。当输入正周期后，辅助指标应开始正常计算（考虑到 WPR 计算的简易性，几乎会立即完成），此时 **BarsCalculated** 将返回柱线总数。

现在我们来展示 **UseWPR2.mq5** 中创建指标句柄的源代码：

```
// UseWPR2.mq5
int handle; // handle to global variable

int OnInit()
{
    // passing name and parameter
    handle = PRTF(iCustom(_Symbol, _Period, "IndWPR", WPRPeriod));
    // next check is useless here because you have to wait,
    // when the indicator is loaded, run and calculate
    // (here it is for demonstration purposes only)
    PRTF(BarsCalculated(handle));
    // successful initialization depends on the descriptor
    return handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

在 *OnCalculate* 中，我们仅记录 *BarsCalculated* 和 *rates_total* 值。

```
int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &data[])
{
    // wait until the slave indicator is calculated on all bars
    if(PRTF(BarsCalculated(handle)) != PRTF(rates_total))
    {
        return prev_calculated;
    }

    // ... here is usually further work using handle

    return rates_total;
}
```

编译并运行 *UseWPR2*，先将参数设为 0，再设为有效值（如 21）。下面是周期为零时的日志记录：

```
iCustom(_Symbol,_Period,IndWPR,WPRPeriod)=10 / ok
BarsCalculated(handle)=-1 / INDICATOR_DATA_NOT_FOUND(4806)
Alert: Incorrect Period value (0). Should be 1 or larger
BarsCalculated(handle)=0 / ok
rates_total=20000 / ok
...
```

在句柄创建后立即检验，由于数据尚不可用，会显示 *INDICATOR_DATA_NOT_FOUND(4806)* 错误，且结果 *BarsCalculated* 为 -1。随后会出现输入参数错误的通知，证实 *IndWPR* 指标已成功加载并启动。在后续日志段中，我们将收到 *BarsCalculated* 值为 0

为使指标正常计算，我们将输入正确的输入参数。此时，*BarsCalculated* 等于 *rates_total*。

```

iCustom(_Symbol,_Period,IndWPR,WPRPeriod)=10 / ok
BarsCalculated(handle)=-1 / INDICATOR_DATA_NOT_FOUND(4806)
BarsCalculated(handle)=20000 / ok
rates_total=20000 / ok
...

```

在掌握从属指标就绪状态的检验方法后，我们可以开始读取其数据。让我们在接下来的示例 *UseWPR3.mq5* 中实践这一点，在该示例中我们将学习 *CopyBuffer* 函数的使用。

5.5.4 从指标获取时间序列数据: CopyBuffer

MQL 程序可以通过指标句柄读取其公共缓冲区中的数据。需要注意，在自定义指标中，此类缓冲区是源代码中通过 *SetIndexBuffer* 函数调用指定的数组。

MQL5 API 提供了 *CopyBuffer* 函数用于读取缓冲区，该函数有 3 种形式。

```

int CopyBuffer(int handle, int buffer, int offset, int count, double &array[])
int CopyBuffer(int handle, int buffer, datetime start, int count, double &array[])
int CopyBuffer(int handle, int buffer, datetime start, datetime stop, double &array[])

```

handle 参数用于指定通过 *iCustom* 或其他函数调用获取的句柄（有关详细信息，请参阅 *IndicatorCreate* 和 内置指标 章节）。*buffer* 参数用于设置要请求数据的指标缓冲区索引。编号从 0 开始。

所请求时间序列的接收元素进入通过引用设置的 *array*。

该函数的三种变体区别在于指定时间戳范围 (*start/stop*) 或获取数据的柱线的编号 (*offset*) 和数量 (*count*) 的方式不同。这些参数的基本用法与我们在 获取报价数组的 Copy 函数概述 中学习的内容完全一致。特别是，在 *offset* 和 *count* 中，复制数据的元素是按从当前向历史方向计数，即起始位置 0 表示当前柱线。接收 *array* 中的元素在物理上按从历史向当前方向的顺序排列（但可以通过 *ArraySetAsSeries* 在逻辑层面反转这种寻址方式）。

CopyBuffer 类似于读取 *Copy Open*, *Copy Close* 等类型内置时间序列的函数。其主要区别在于，报价时间序列由终端自身生成，而指标缓冲区中的时间序列由自定义指标或 内置指标 计算得出。此外，对于指标而言，我们在处理程序创建函数（如 *iCustom*）中预先设定了一组特定的交易品种和时间范围，以定义和标识时间序列；而在 *CopyBuffer* 中，这些信息是通过 *handle* 间接传递的。

当复制未知数量的数据为目标数组时，建议使用动态数组。在这种情况下，*CopyBuffer* 函数将根据被复制数据的大小分配接收数组的大小。如果需要重复复制已知数量的数据，则最好在静态分配的缓冲区中进行此操作（使用带有 *static* 修饰符的局部变量，或在全局上下文中使用固定大小的变量），以避免重复进行内存分配。

如果接收数组是一个指标缓冲区（即先前通过 *SetIndexBuffer* 函数在系统中注册的数组），则时间序列和接收缓冲区的索引方式是相同的（前提是请求相同的交易品种/时间范围）。在这种情况下，可轻松实现对接收方的局部填充（特别是用于更新最后几根柱线，详见下文示例）。如果所请求时间序列的交易品种或时间范围与当前图表的交易品种和/或时间范围不匹配，该函数返回的元素数量不会超过源数据和目标数据中最小柱线数量。

如果将普通数组（而非缓冲区）作为 *array* 参数传递，则该函数将从第一批元素开始完整填充（动态情况下）或部分填充（静态情况下，大小超限）。因此，如果需要将指标值部分复制到另一个数组的任意位置时，则需要使用中间数组：先将所需数量的元素复制到中间数组，再从中间数组转存至最终目标位置。

该函数将返回复制的元素数量，如果发生错误（包括暂时无可用数据）则返回 -1。

由于指标通常直接或间接依赖于价格时间序列，其计算将在报价同步完成后才开始。因此，需考虑终端中 [时间序列组织与存储的技术要点](#)，并做好请求数据不会即时出现的准备。具体来说，我们可能会收到 0 数据或少于请求数量的数据。所有此类情况都应根据实际情况处理，例如等待构建或向用户报告问题。

如果请求的时间序列尚未构建，或需要从服务器下载，则该函数的行为会因调用它的 MQL 程序类型不同而有所差异。

当从指标请求尚未准备就绪的数据时，该函数将立即返回 -1，但同时会启动时间序列的加载和构建流程。

当从 EA 交易或脚本请求数据时，如果数据可通过本地历史构建，则将启动从服务器的下载及/或开始构建所需的时间序列。该函数将在为函数分配的同步执行超时期限内（45 秒）返回就绪的数据量（调用代码会等待函数执行完成）。

请注意，*CopyBuffer* 函数可以从缓冲区读取数据，无论其操作模式是 `INDICATOR_DATA`、`INDICATOR_COLOR_INDEX` 还是 `INDICATOR_CALCULATIONS`，其中后两种模式对用户隐藏。

同样需要注意，在调用的指标中，时间序列的偏移量可以通过 `PLOT_SHIFT` 特性设置，这会影响到通过 *CopyBuffer* 读取数据的偏移量。例如，如果指标线向未来方向偏移了 `N` 根柱线，则在 *CopyBuffer* 参数（第一种形式）中必须设置 *offset* 为 `(-N)`，即带有负号，这是因为当前时间序列柱线索引为 0，而随着向未来偏移，未来柱线索引值会逐根递减 1。这种情况尤其常见于 *Gator* 指标，因为其空值图表会向前偏移 *TeethShift* 参数的值，而第一个图表则会偏移 *LipsShift* 参数的值。应基于这两个参数中的较大值进行修正。具体示例将在 [从存在偏移的图表中读取数据](#) 章节中详细说明。

MQL5 未提供获取第三方指标的 `PLOT_SHIFT` 特性的编程工具。因此必要时，需通过输入变量向用户请求该信息。

我们将在 [EA 交易](#) 章节中详述如何在 EA 交易代码中使用 *CopyBuffer* 函数，但目前我们先聚焦于指标。

接下来我们继续完善辅助指标 *IndWPR* 的示例。这次在 *UseWPR3.mq5* 版本中，我们将提供一个指标缓冲区，并通过 *CopyBuffer* 函数来为其填充来自 *IndWPR* 的数据。为此，我们使用的指令中包含设置缓冲区数量和渲染设置。

```
#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots 1

#property indicator_type1 DRAW_LINE
#property indicator_color1 clrBlue
#property indicator_width1 1
#property indicator_label1 "WPR"
```

在全局范围内，我们在描述输入参数时使用了 `WPR` 周期、缓冲区数组以及带描述符的变量。

```

input int WPRPeriod = 14;

double WPRBuffer[];

int handle;

```

*OnInit*处理程序基本保持不变：仅新增了 *SetIndexBuffer* 调用。

```

int OnInit()
{
    SetIndexBuffer(0, WPRBuffer);
    handle = iCustom(_Symbol, _Period, "IndWPR", WPRPeriod);
    return handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}

```

在 *OnCalculate*中，我们将直接复制数据而不做任何转换。

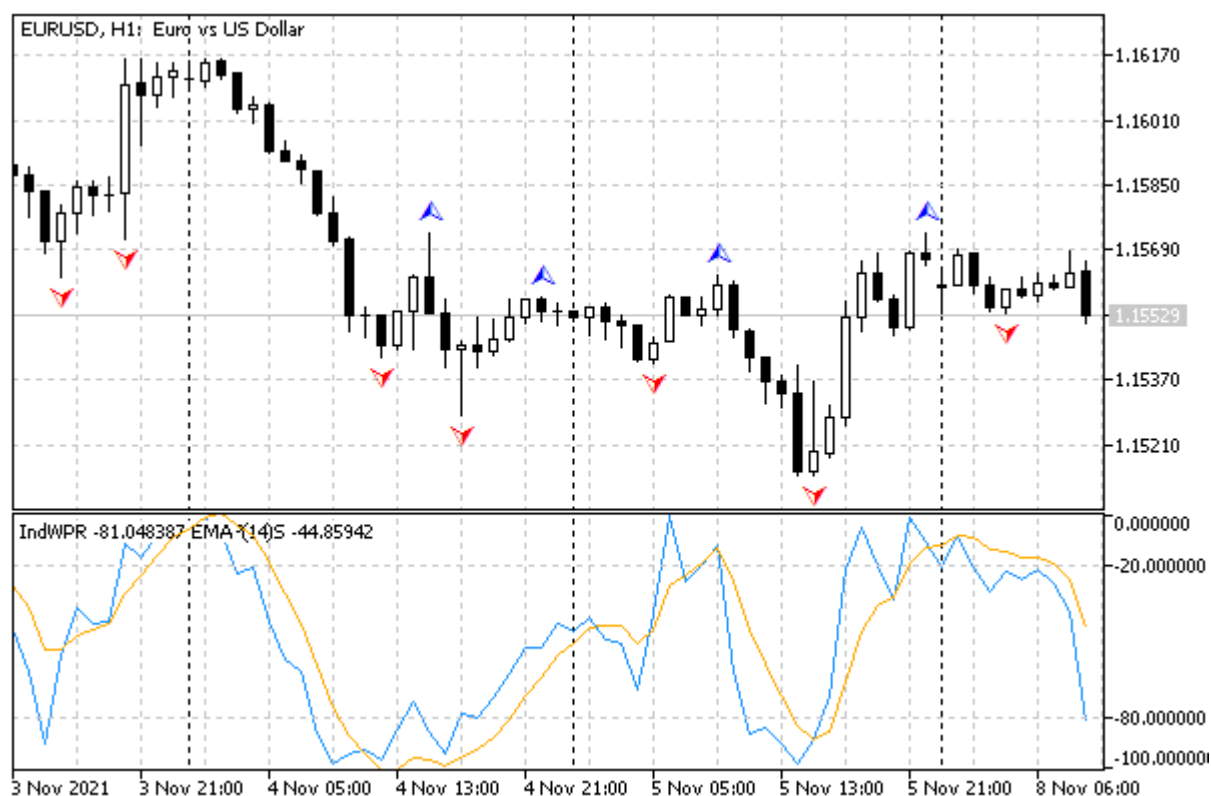
```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &data[])
{
    // waiting for the calculation to be ready for all bars
    if(BarsCalculated(Handle) != rates_total)
    {
        return prev_calculated;
    }

    // copy the entire timeseries of the subordinate indicator or on new bars to our b
    const int n = CopyBuffer(handle, 0, 0, rates_total - prev_calculated + 1, WPRBuffer);
    // if there are no errors, our data is ready for all bars rates_total
    return n > -1 ? rates_total : 0;
}

```

通过编译并运行 *UseWPR3*，我们实际上将获得一个原始 WPR 指标的副本，但水平位调整、数值精度和标题除外。这足以用于测试该机制，但通常基于一个或多个辅助指标的新指标会提供自己的分析思路和数据转换。因此，我们将开发另一个生成买卖交易信号的指标（从交易角度而言，这些信号不应被视为交易模型，因为这仅仅是一个编程任务）。该指标的设计思路如下图所示。



Indicators IndWPR, IndTripleEMA, IndFractals

我们将 WPR 从超买和超卖区域的退出，分别作为卖出和买入的建议信号。为避免信号对随机波动做出反应，我们对 WPR 应用三重移动平均线，并检查其值是否突破上下区域边界。

作为这些信号的过滤器，我们将检查在此前的最后一个分形形态：顶部分形意味着价格向下反转并确认卖出信号，而底部分形意味着价格向上反转并支持买入信号。分形出现延迟，其延迟的柱线数量等于分形的阶数。

新指标包含在 *UseWPRFractals.mq5* 文件中。

我们需要三个缓冲区：两个信号缓冲区、一个过滤缓冲区。我们本可以将后者设置为 INDICATOR_CALCULATIONS 模式。我们改为使用标准 INDICATOR_DATA，但使用 DRAW_NONE 风格，这样它不会在图表上显示，但数值将在数据窗口中可见。

信号将显示在主图表上（默认基于 *Close* 价），因此我们将使用 *indicator_chart_window* 指令。我们仍可调用 WPR 类型的指标（这类指标通常显示在独立窗口中），因为所有从属指标可以无需可视化即可进行计算。如有必要，我们可以绘制指标，但我们在图表章节中将讨论这一点（请参阅 *ChartIndicatorAdd*）。

```

#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 3
// buffer drawing settings
#property indicator_type1 DRAW_ARROW
#property indicator_color1 clrRed
#property indicator_width1 1
#property indicator_label1 "Sell"
#property indicator_type2 DRAW_ARROW
#property indicator_color2 clrBlue
#property indicator_width2 1
#property indicator_label2 "Buy"
#property indicator_type3 DRAW_NONE
#property indicator_color3 clrGreen
#property indicator_width3 1
#property indicator_label3 "Filter"

```

在输入变量中，我们将提供指定 WPR 周期、平均（平滑）周期和分形阶数的功能。这些是从属指标的参数。此外，我们引入 *offset* 变量，用于指定分析信号的柱线编号。值 0（默认值）表示当前柱线并以分时报价模式分析（注意：最后一根柱线的信号可能会重绘；部分交易者不喜欢这一点）。如果我们将 *offset* 设为 1，我们将分析已形成的柱线，此类信号不会变化。

```

input int PeriodWPR = 11;
input int PeriodEMA = 5;
input int FractalOrder = 1;
input int Offset = 0;
input double Threshold = 0.2;

```

Threshold 变量用于定义超买和超卖区域阈值，取值范围为 ± 1.0 （双向）。例如，如果遵循经典 WPR 设置（在 0 到 -100 的刻度上，水平位分别为 -20 和 -80），则 *Threshold* 应设为 0.4。

为指标缓冲区提供以下数组。

```

double UpBuffer[]; // upper signal means overbought, i.e. selling
double DownBuffer[]; // lower signal means oversold, i.e. buy
double filter[]; // fractal filter direction +1 (up/buy), -1 (down/sell)

```

指标句柄将保存在全局变量中。

```
int handleWPR, handleEMA3, handleFractals;
```

我们将像往常一样在 *OnInit* 中执行所有设置。由于 *CopyBuffer* 函数使用从现在到过去的索引方式，为保持数据读取统一性，我们将为所有数组设置“序列”标志 (*ArraySetAsSeries*)。

```

int OnInit()
{
    // binding buffers
    SetIndexBuffer(0, UpBuffer);
    SetIndexBuffer(1, DownBuffer);
    SetIndexBuffer(2, Filter, INDICATOR_DATA); // version: INDICATOR_CALCULATIONS
    ArraySetAsSeries(UpBuffer, true);
    ArraySetAsSeries(DownBuffer, true);
    ArraySetAsSeries(Filter, true);

    // arrow signals
    PlotIndexSetInteger(0, PLOT_ARROW, 234);
    PlotIndexSetInteger(1, PLOT_ARROW, 233);

    // subordinate indicators
    handleWPR = iCustom(_Symbol, _Period, "IndWPR", PeriodWPR);
    handleEMA3 = iCustom(_Symbol, _Period, "IndTripleEMA", PeriodEMA, 0, handleWPR);
    handleFractals = iCustom(_Symbol, _Period, "IndFractals", FractalOrder);
    if(handleWPR == INVALID_HANDLE
    || handleEMA3 == INVALID_HANDLE
    || handleFractals == INVALID_HANDLE)
    {
        return INIT_FAILED;
    }

    return INIT_SUCCEEDED;
}

```

在 *iCustom* 调用中，需特别注意 *handleEMA3* 的创建方式。由于该平均值是基于 WPR 计算的，我们将在调用指标 *IndTripleEMA* 的实际参数后，将 *handleWPR*（在上一次 *iCustom* 调用中获得）作为最后一个参数传入。在这一过程中，我们必须指定 *IndTripleEMA* 的完整输入参数列表（其中的参数包括 *int InpPeriodEMA* 和 *BEGIN_POLICY InpHandleBegin*；我们曾使用第二个参数来研究初始柱线的跳过问题，现在虽然不需要该参数，但仍必须传递该参数，因此只需将其设置为 0）。如果我们在调用时省略了第二个参数（因为在当前应用场景中为无关参数），则传递的 *handleWPR* 句柄将在被调用的指标中被解释为 *InpHandleBegin*。因此，*IndTripleEMA* 将被应用于常规 *Close* 价。

当我们无需传递额外句柄时，*iCustom* 调用语法允许省略末尾任意数量的参数，这些参数将从源代码中获取默认值。

在 *OnCalculate* 处理程序中，我们等待 WPR 指标和分形就绪，然后使用辅助函数 *MarkSignals* 为整个历史或最后一根柱线计算信号。


```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &data[])
{
    if(BarsCalculated(handleEMA3) != rates_total
    || BarsCalculated(handleFractals) != rates_total)
    {
        return prev_calculated;
    }

    ArraySetAsSeries(data, true);

    if(prev_calculated == 0) // first launch
    {
        ArrayInitialize(UpBuffer, EMPTY_VALUE);
        ArrayInitialize(DownBuffer, EMPTY_VALUE);
        ArrayInitialize(Filter, 0);

        // look for signals throughout history
        for(int i = rates_total - FractalOrder - 1; i >= 0; --i)
        {
            MarkSignals(i, Offset, data);
        }
    }
    else // online
    {
        for(int i = 0; i < rates_total - prev_calculated; ++i)
        {
            UpBuffer[i] = EMPTY_VALUE;
            DownBuffer[i] = EMPTY_VALUE;
            Filter[i] = 0;
        }

        // looking for signals on a new bar or each tick (if Offset == 0)
        if(rates_total != prev_calculated
        || Offset == 0)
        {
            MarkSignals(0, Offset, data);
        }
    }

    return rates_total;
}

```

我们重点关注 *MarkSignals* 中隐藏的 *CopyBuffer* 函数的操作。平滑处理后的 WPR 值将被读取到 *wpr[2]* 数组中，分形数据将被读取到 *peaks[1]* 和 *hollows[1]* 中。

```

int MarkSignals(const int bar, const int offset, const double &data[])
{
    double wpr[2];
    double peaks[1], hollows[1];
    ...

```

然后，我们通过三次 *CopyBuffer* 调用填充本地数组。请注意，我们不需要直接读取 *IndWPR* 指标，因为它已被用于 *IndTripleEMA* 的计算中。我们通过 *handleEMA3* 句柄将数据读取到 *wpr* 数组中。需特别注意，分形指标包含 2 个缓冲区，因此调用两次 *CopyBuffer* 函数，分别为 *peaks* 和 *hollows* 数组使用不同的索引 0 和 1。读取分形时需设置 *FractalOrder* 的偏移量，因为分形只能在左右两侧拥有特定数量柱线的柱线上形成。

```

if(CopyBuffer(handleEMA3, 0, bar + offset, 2, wpr) == 2
&& CopyBuffer(handleFractals, 0, bar + offset + FractalOrder, 1, peaks) == 1
&& CopyBuffer(handleFractals, 1, bar + offset + FractalOrder, 1, hollows) == 1)
{
    ...

```

接下来，我们从 *Filter* 缓冲区的上一根柱线中获取先前的过滤方向（在历史起始处为 0，但当出现向上或向下分形时，会在该位置写入 +1 或 -1，这可以在下面的源代码中看到），并在检测到任何新分形时对它进行相应的更改。

```

int filterdirection = (int)Filter[bar + 1];

// the last fractal sets the reversal movement
if(peaks[0] != EMPTY_VALUE)
{
    filterdirection = -1; // sell
}
if(hollows[0] != EMPTY_VALUE)
{
    filterdirection = +1; // buy
}

Filter[bar] = filterdirection; // remember the current direction

```

最后，我们分析平滑处理后的 *WPR* 从上部或下部区域向中间区域的过渡情况，同时考虑 *Threshold* 参数中指定的区域宽度。

```

// translate 2 WPR values into the range [-1,+1]
const double old = (wpr[0] + 50) / 50;    // +1.0 -1.0
const double last = (wpr[1] + 50) / 50;    // +1.0 -1.0

// bounce from the top down
if(filterdirection == -1
&& old >= 1.0 - Threshold && last <= 1.0 - Threshold)
{
    UpBuffer[bar] = data[bar];
    return -1; // sale
}

// bounce from the bottom up
if(filterdirection == +1
&& old <= -1.0 + Threshold && last >= -1.0 + Threshold)
{
    DownBuffer[bar] = data[bar];
    return +1; // purchase
}
}
return 0; // no signal
}

```

下面是该指标在图表上的截屏：



基于 WPR、EMA3 和分形的信号指标 UseWPRFractals

5.5.5 支持多个交易品种和时间范围

到目前为止，在所有指标示例中，我们都只为与当前图表相同的交易品种和时间范围创建描述符。但实际上并没有此类限制。我们可以在任何交易品种和时间范围上创建辅助指标。当然，在这种情况下，就像之前通过计时器那样，需要等待第三方时间序列就绪。

让我们实现一个 WPR 多时间范围指标 (请参阅文件 *UseWPRMTF.mq5*)，该指标还可以指定在任意交易品种 (不同于图表上的交易品种) 上进行计算。

我们将基于 ENUM_TIMEFRAMES 枚举中的所有标准时间范围，显示指定周期的 WPR 指标值。时间范围数量为 21，因此该指标将始终显示在最后 21 根柱线上。最右侧的 0 号柱线显示 M1 的 WPR，下一根柱线显示 M2 的 WPR，依此类推，直到第 20 根柱线显示月级时间范围的 WPR。为便于阅读，我们将绘图设置不同颜色：分钟级时间范围为红色，小时级时间范围为绿色，日线及更大时间范围为蓝色。

由于可以在指标中设置工作交易品种并在同一图表上为不同交易品种创建多个副本，我们将选择 DRAW_ARROW 绘制风格并提供一个用于指定交易品种的输入参数。通过这种方式，可以区分不同交易品种的指标显示。着色需要一个额外的缓冲区。

```
#property indicator_separate_window
#property indicator_buffers 2
#property indicator_plots 1

#property indicator_type1 DRAW_COLOR_ARROW
#property indicator_color1 clrRed,clrGreen,clrBlue
#property indicator_width1 3
#property indicator_label1 "WPR"
```

WPR 值将被转换到 [-1,+1] 范围。让我们为子窗口选择一个在该范围基础上留有一定裕度的刻度。值为 ±0.6 的水平位对应 WPR 转换前的标准 -20 和 -80。

```
#property indicator_maximum +1.2
#property indicator_minimum -1.2

#property indicator_level1 +0.6
#property indicator_level2 -0.6
#property indicator_levelstyle STYLE_DOT
#property indicator_levelcolor clrSilver
#property indicator_levelwidth 1
```

输入变量包括：WPR 周期、工作交易品种、显示箭头的代码。如果交易品种留空，则使用当前图表的交易品种。

```
input int WPRPeriod = 14;
input string WorkSymbol = ""; // Symbol
input int Mark = 0;

const string _WorkSymbol = (WorkSymbol == "" ? _Symbol : WorkSymbol);
```

为便于编码，时间范围集合列于数组 *TF* 中。

```

#define TFS 21

ENUM_TIMEFRAMES TF[TFS] =
{
    PERIOD_M1,
    PERIOD_M2,
    PERIOD_M3,
    ...
    PERIOD_D1,
    PERIOD_W1,
    PERIOD_MN1,
};

```

每个时间范围的指标描述符存储在数组 *Handle* 中。

```
int Handle[TFS];
```

我们将通过 *OnInit* 函数配置指标缓冲区并获取句柄。

```

double WPRBuffer[];
double Colors[];

int OnInit()
{
    SetIndexBuffer(0, WPRBuffer);
    SetIndexBuffer(1, Colors, INDICATOR_COLOR_INDEX);
    ArraySetAsSeries(WPRBuffer, true);
    ArraySetAsSeries(Colors, true);
    PlotIndexSetString(0, PLOT_LABEL, _WorkSymbol + " WPR");

    if(Mark != 0)
    {
        PlotIndexSetInteger(0, PLOT_ARROW, Mark);
    }

    for(int i = 0; i < TFS; ++i)
    {
        Handle[i] = iCustom(_WorkSymbol, TF[i], "IndWPR", WPRPeriod);
        if(Handle[i] == INVALID_HANDLE) return INIT_FAILED;
    }

    IndicatorSetInteger(INDICATOR_DIGITS, 2);
    IndicatorSetString(INDICATOR_SHORTNAME,
        "%Rmtf" + "(" + _WorkSymbol + "/" + (string)WPRPeriod + ")");

    return INIT_SUCCEEDED;
}

```

OnCalculate 函数的计算遵循常规方案：等待数据就绪，初始化，在新柱线填充数据。辅助函数 *IsDataReady* 和 *FillData* 用于执行与描述符的直接交互（见下文）。

```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &data[])
{
    // waiting for slave indicators to be ready
    if(!IsDataReady())
    {
        EventSetTimer(1); // if not ready, postpone the calculation
        return prev_calculated;
    }
    if(prev_calculated == 0) // initialization
    {
        ArrayInitialize(WPRBuffer, EMPTY_VALUE);
        ArrayInitialize(Colors, EMPTY_VALUE);
        // constant colors for the latest TFS bars
        for(int i = 0; i < TFS; ++i)
        {
            Colors[i] = i < 11 ? 0 : (i < 18 ? 1 : 2);
        }
    }
    else // preparing a new bar
    {
        for(int i = prev_calculated; i < rates_total; ++i)
        {
            WPRBuffer[i] = EMPTY_VALUE;
            Colors[i] = 0;
        }
    }

    if(prev_calculated != rates_total) // new bar
    {
        // clear the label on the oldest bar that moved to the left beyond TFS bars
        WPRBuffer[TFS] = EMPTY_VALUE;
        // update bar coloring
        for(int i = 0; i < TFS; ++i)
        {
            Colors[i] = i < 11 ? 0 : (i < 18 ? 1 : 2);
        }
    }

    // copy the data from the subordinate indicators to our buffer
    FillData();
    return rates_total;
}

```

如有必要，我们会通过计时器启动重新计算。

```

void OnTimer()
{
    ChartSetSymbolPeriod(0, _Symbol, _Period);
    EventKillTimer();
}

```

以下是 *IsDataReady* 和 *FillData* 函数的实现。

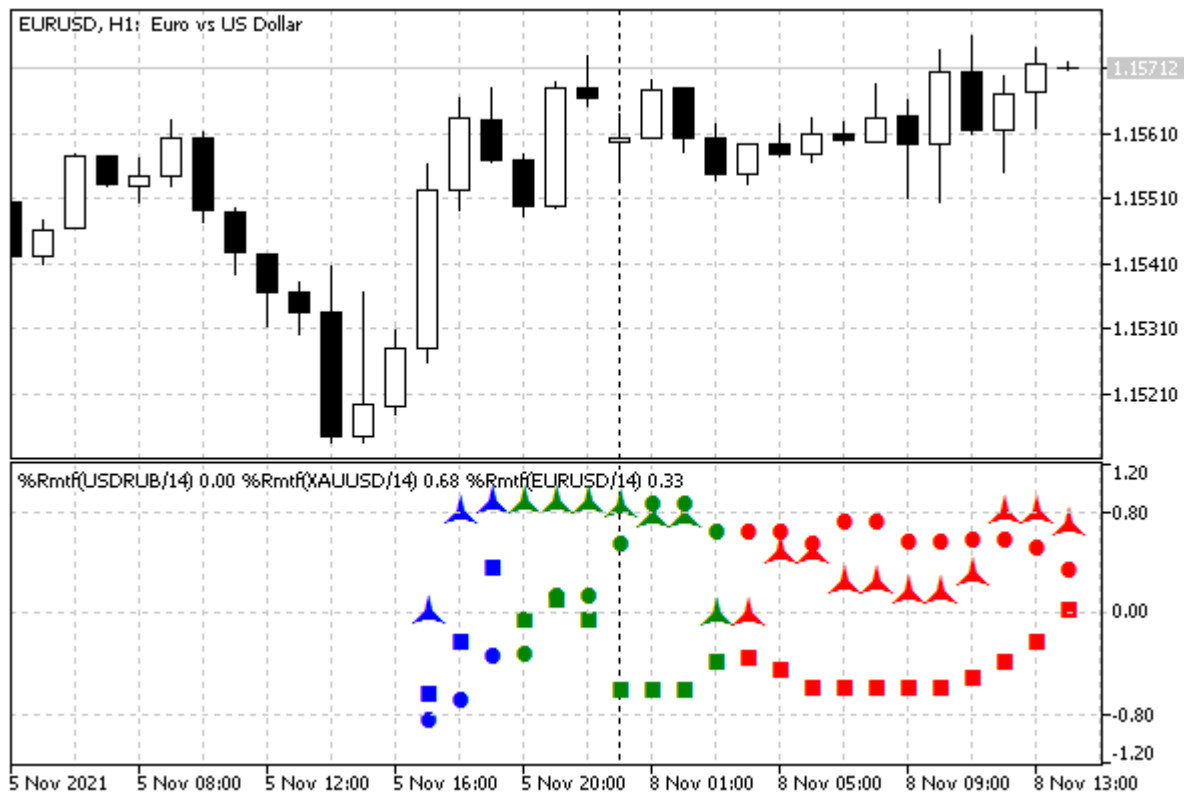
```

bool IsDataReady()
{
    for(int i = 0; i < TFS; ++i)
    {
        if(BarsCalculated(Handle[i]) != iBars(_WorkSymbol, TF[i]))
        {
            Print("Waiting for ", _WorkSymbol, " ", EnumToString(TF[i]));
            return false;
        }
    }
    return true;
}

void FillData()
{
    for(int i = 0; i < TFS; ++i)
    {
        double data[1];
        // taking the last actual value (buffer 0, index 0)
        if(CopyBuffer(Handle[i], 0, 0, 1, data) == 1)
        {
            WPRBuffer[i] = (data[0] + 50) / 50;
        }
    }
}

```

让我们编译该指标，观察其在图表上的显示效果。例如，我们可以为 EURUSD、USDRUB 和 XAUUSD 创建三个副本。

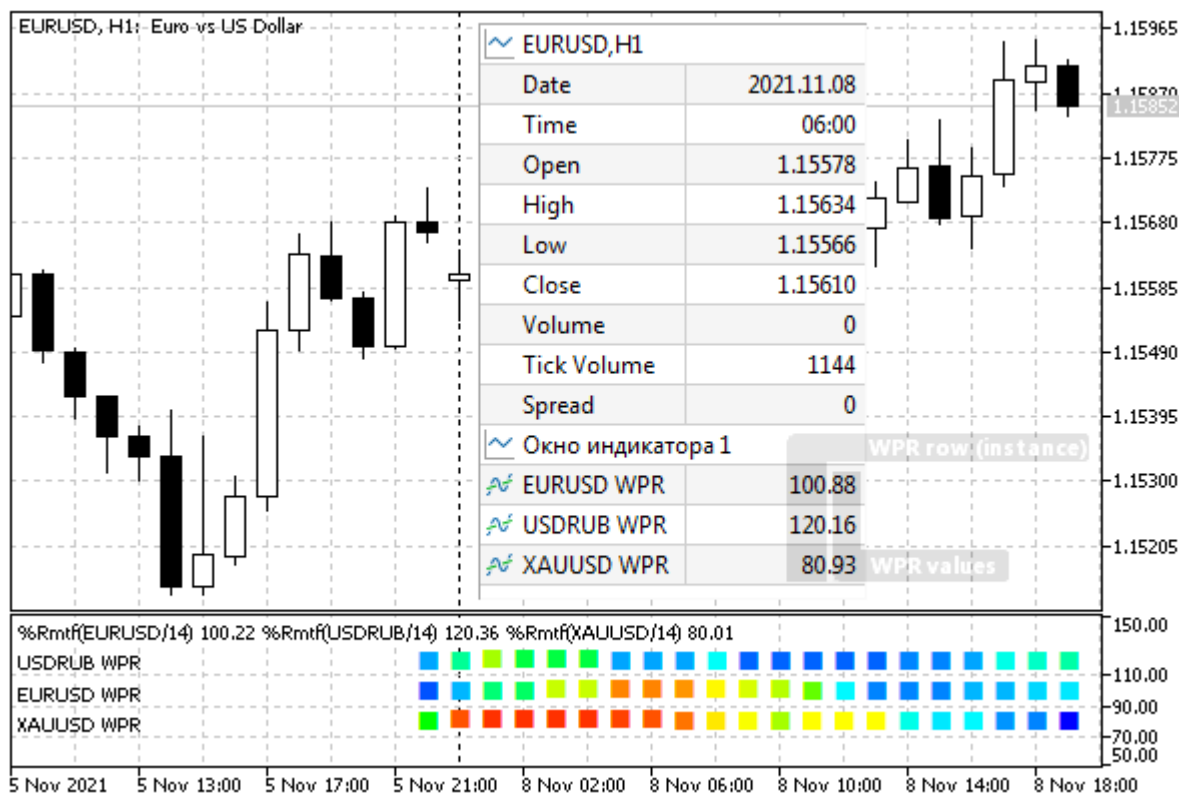


不同工作交易品种的多时间范围 WPR 的三个实例

在第一次计算时，指标可能需要大量时间来为所有时间范围准备时间序列。

在计算层面，完全相同的指标 *UseWPRMTFDashboard.mq5* 被设计为交易员常用的仪表盘形式。对于每个交易品种，我们通过指标的 *Level* 参数设置独立的垂直缩进。所有时间范围的 WPR 值将在此处以标记线的形式显示，并且这些值通过颜色编码区分。在该版本中，WPR 值被归一化到 [0..1] 区间，因此使用间隔数十个单位（例如在下面截图中为 20）的水平标尺，可在同一子窗口无重叠地布局多个指标实例（例如 80、100、120 等）。每个副本用于其各自的工作交易品种。此外，由于 *Level* 大于 1.0 且 WPR 值较小，它们在 *Data window* 中分别显示为：小数点左侧和小数点右侧。

标签标尺的标签由 *OnInit* 中动态添加的水平位提供。



不同工作交易品种的三线式多时间范围 WPR 面板

你可以研究 `UseWPRMTFDashboard.mq5` 的源代码，并将其与 `UseWPRMTF.mq5` 进行比较。为生成一系列渐变色，我们使用了 `ColorMix.mqh` 文件。

在完成对 内置指标 (包括 `iWPR`) 的学习后，我们可以将自定义 `IndWPR` 替换为内置 `iWPR`。

关于复合指标的效率和资源占用

上述通过生成多个辅助指标的方法在执行速度和资源占用方面并不高效。这主要是一个集成 MQL 程序并在这些程序之间交换数据的示例。但与任何技术一样，应合理使用方法。

所创建的两个指标均会计算时间序列中所有柱线的 WPR，但最终只有最后一个值被调用指标所采用。这样既浪费内存又消耗处理器时间。

如果辅助指标的源代码可用或其运算逻辑已知，最优方法是将计算算法嵌入主指标 (或 EA 交易) 内部，并仅针对最小所需深度的近期历史数据范围执行计算。

在某些情况下，你可以通过在当前时间范围上执行等效计算来避免引用更高时间范围：例如，无需 14 根日柱线的价格区间 (需要构建完整的 D1 时间序列)，你可以使用 $14 * 24$ H1 柱线上的价格区间，但前提是 24 小时交易且指标在 H1 图表上运行。

当交易系统中使用商用指标 (无源代码) 时，只能通过开放编程接口获取其数据。这种情况下，创建句柄并通过 `CopyBuffer` 从指标缓冲区读取数据，是唯一可行且兼具便利性与通用性的方法。需要注意的是，调用 API 函数的开销显著高于操作 MQL 程序内部数组或调用本地函数。如果你需要同时运行多个终端，且每个终端都可能运行一组未优化的 MQL 程序，在资源有限的情况下，系统性能可能会下降。

5.5.6 内置指标概述

终端提供了大量常用指标，这些指标也可以通过 API 访问。因此，你无需在 MQL5 中实现其算法。此类指标通过类似于 *iCustom* 的内置函数创建。例如，我们之前出于教学目的创建了自定义版本的 WPR 和 EMA。不过，可直接通过 *iWPR* 和 *iTEMA* 函数使用相应的指标。下表列出了所有可用指标。

所有内置指标均以字符串形式的工作交易品种和时间范围作为前两个参数，并返回一个整数作为指标描述符。通常，所有函数的原型如下：

```
int iFunction(const string symbol, ENUM_TIMEFRAMES timeframe, ...)
```

省略号部分将由特定指标的具体参数替代。它们的数量和类型各不相同。部分指标没有参数。

例如，WPR 与我们自建版本一样，只有一个参数，即周期：*int iWPR(const string symbol, ENUM_TIMEFRAMES timeframe, int period)*。与我们的版本不同，内置分形指标没有特殊参数：*int iFractals(const string symbol, ENUM_TIMEFRAMES period)*。在此情况下，分形的阶数采用硬编码，值为 2，即极值点（顶部或底部）前后必须分别至少有两根柱线的 *high* 和 *low* 价不超过该极值点。

允许将值设置为 NULL，而非交易品种。NULL 表示当前图表的工作交易品种，而 *timeframe* 参数中的值 0 对应于当前图表的时间范围，因为它也是 ENUM_TIMEFRAMES 枚举中的 PERIOD_CURRENT 值（请参阅 [交易品种和时间范围](#) 章节）。

还需要注意，不同类型的指标具有不同数量的缓冲区。例如，移动平均线或 WPR 只有一个缓冲区，而分形指标有两个。缓冲区数量也会在表格的单独一列中注明。

函数	指标名称	选项	缓冲区数量
iAC	Accelerator Oscillator	—	1*
iAD	Accumulation / Distribution	ENUM_APPLIED_VOLUME volume	1*
iADX	Average Directional Index	int period	3*
iADXWilder	Average Directional Index by Welles Wilder	int period	3*
iAlligator	Alligator	int jawPeriod, int jawShift, int teethPeriod, int teethShift, int lipsPeriod, int lipsShift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	3
iAMA	Adaptive Moving Average	int period, int fast, int slow, int shift, ENUM_APPLIED_PRICE price	1
iAO	Awesome Oscillator	—	1*
iATR	Average True Range	int period	1*

函 数	指 标 名 称	选 项	缓 冲 区 数 量
iBands	Bollinger Bands	int period, int shift, double deviation, ENUM_APPLIED_PRICE price	3
iBearsPower	Bears Power	int period	1*
iBullsPower	Bulls Power	int period	1*
iBWMFI	Market Facilitation Index by Bill Williams	ENUM_APPLIED_VOLUME volume	1*
iCCI	Commodity Channel Index	int period, ENUM_APPLIED_PRICE price	1*
iChaikin	Chaikin Oscillator	int fast, int slow, ENUM_MA_METHOD method, ENUM_APPLIED_VOLUME volume	1*
iDEMA	Double Exponential Moving Average	int period, int shift, ENUM_APPLIED_PRICE price	1
iDeMarker	DeMarker	int period	1*
iEnvelopes	Envelopes	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price, double deviation	2
iForce	Force Index	int period, ENUM_MA_METHOD method, ENUM_APPLIED_VOLUME volume	1*
iFractals	Fractals	—	2
iFrAMA	Fractal Adaptive Moving Average	int period, int shift, ENUM_APPLIED_PRICE price	1
iGator	Gator Oscillator	int jawPeriod, int jawShift, int teethPeriod, int teethShift, int lipsPeriod, int lipsShift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	4*
ilchimoku	Ichimoku Kinko Hyo	int tenkan, int kijun, int senkou	5
iMomentum	Momentum	int period, ENUM_APPLIED_PRICE price	1*
iMFI	Money Flow Index	int period, ENUM_APPLIED_VOLUME volume	1*

函数	指标名称	选项	缓冲区数量
iMA	Moving Average	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	1
iMACD	Moving Averages Convergence-Divergence	int fast, int slow, int signal, ENUM_APPLIED_PRICE price	2*
iOBV	On Balance Volume	ENUM_APPLIED_VOLUME volume	1*
iOsMA	Moving Average of Oscillator (MACD histogram)	int fast, int slow, int signal, ENUM_APPLIED_PRICE price	1*
iRSI	Relative Strength Index	int period, ENUM_APPLIED_PRICE price	1*
iRVI	Relative Vigor Index	int period	1*
iSAR	Parabolic Stop And Reverse System	double step, double maximum	1
iStdDev	Standard Deviation	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	1*
iStochastic	Stochastic Oscillator	int Kperiod, int Dperiod, int slowing, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	2*
iTEMA	Triple Exponential Moving Average	int period, int shift, ENUM_APPLIED_PRICE price	1
iTriX	Triple Exponential Moving Averages Oscillator	int period, ENUM_APPLIED_PRICE price	1*
iVIDyA	Variable Index Dynamic Average	int momentum, int smooth, int shift, ENUM_APPLIED_PRICE price	1
iVolumes	Volumes	ENUM_APPLIED_VOLUME volume	1*
iWPR	Williams Percent Range	int period	1*

在右侧列中，带有独立窗口的指标均已标注星号*（它们显示在主图表下方）。

最常用的参数是定义指标周期的参数 *period*、*fast*、*slow* 等变体），以及线条 *shift*：当为正数时，绘图向右移动；当为负数时，绘图向左移动指定柱线数量。

许多参数具有应用枚举类型：ENUM_APPLIED_PRICE、ENUM_APPLIED_VOLUME、ENUM_MA_METHOD。我们已经在 [枚举](#) 章节中介绍过 ENUM_APPLIED_PRICE。所有可用类型及其说明如下表所示。

标识符	说明	值
PRICE_CLOSE	柱线收盘价	1
PRICE_OPEN	柱线开盘价	2
PRICE_HIGH	柱线最高价	3
PRICE_LOW	柱线最低价	4
PRICE_MEDIAN	中间价，即 (最高价 + 最低价)/2	5
PRICE_TYPICAL	典型价，即 (最高价 + 最低价 + 收盘价)/3	6
PRICE_WEIGHTED	加权平均价，即 (最高价 + 最低价 + 收盘价 + 收盘价)/4	7

与成交量相关的指标可基于分时报价成交量 (实际为分时报价计数器)或实际成交量 (通常仅适用于交易所金融工具)运行。这两种类型都汇总在 ENUM_APPLIED_VOLUME 枚举中。

标识符	说明	值
VOLUME_TICK	分时报价成交量	0
VOLUME_REAL	交易成交量	1

许多技术指标会对时间序列进行平滑处理 (或求平均)。终端支持四种最常见的平滑方法，这些方法在 MQL5 中通过 ENUM_MA_METHOD 枚举的元素来指定。

标识符	说明	值
MODE_SMA	简单平均	0
MODE_EMA	指数平均	1
MODE_SMMA	平滑平均	2
MODE_LWMA	线性加权平均	3

对于 Stochastic 指标 (我们将在下一节讨论其示例)，有两个计算选项：基于 *Close* 价或基于 *High/Low* 价。这些值在专用枚举 ENUM_STO_PRICE 中提供。

标识符	说明	值
STO_LOWHIGH	基于最低价/最高价计算	0
STO_CLOSECLOSE	基于收盘价/收盘价计算	1

对于包含多个缓冲区的指标，其缓冲区用途和编号如下表所示。

指 标	常 量	说 明	值
ADX, ADXW			
	MAIN_LINE	主 线	0
	PLUSDI_LINE	+DI 线	1
	MINUSDI_LINE	-DI 线	2
iAlligator			
	GATORJAW_LINE	颚 线	0
	GATORTEETH_LINE	齿 线	1
	GATORLIPS_LINE	唇 线	2
iBands			
	BASE_LINE	主 线	0
	UPPER_BAND	上 轨	1
	LOWER_BAND	下 轨	2
iEnvelopes, iFractals			
	UPPER_LINE	上 轨 线	0
	LOWER_LINE	下 轨 线	1
iGator			
	UPPER_HISTOGRAM	上 直 方 图	0
	LOWER_HISTOGRAM	下 直 方 图	2
ilchimoku			
	TENKANSEN_LINE	Tenkan-sen 线	0
	KIJUNSEN_LINE	Kijun-sen 线	1
	SENKOUSPANA_LINE	Senkou Span A 线	2
	SENKOUSPANB_LINE	Senkou Span B 线	3
	CHIKOSPAN_LINE	Chikou span 线	4
iMACD, iRVI, iStochastic			
	MAIN_LINE	主 线	0
	SIGNAL_LINE	信 号 线	1

所有指标的计算公式在 [MetaTrader 5 文档](#) 中提供。

关于调用指标函数的完整技术信息（包括源代码示例）可以在 [MQL5 文档](#) 中找到。本书后续将探讨部分相关示例。

5.5.7 使用内置指标

作为使用内置指标的简单入门示例，我们将使用 *iStochastic* 调用。该指标函数的原型如下：

```
int iStochastic(const string symbol, ENUM_TIMEFRAMES timeframe,
int Kperiod, int Dperiod, int slowing,
ENUM_MA_METHOD method, ENUM_STO_PRICE price)
```

如我们所看到的，除了标准参数 *symbol* 和 *time frame* 外，随机指标还有几个特定参数：

- *Kperiod* — 计算 % K 线的柱线数量
- *Dperiod* — % D 线的主平滑周期
- *slowing* — 二次平滑周期（减速因子）
- *method* — 平均（平滑）方法
- *price* — 随机指标计算方法

让我们尝试创建一个自己的指标 *UseStochastic.mq5*，该指标会将随机指标的值复制到其缓冲区中。由于随机指标有两个缓冲区，我们同样预留两个缓冲区：分别是“主线”和“信号线”。

```
#property indicator_separate_window
#property indicator_buffers 2
#property indicator_plots 2

#property indicator_type1 DRAW_LINE
#property indicator_color1 clrBlue
#property indicator_width1 1
#property indicator_label1 "St'Main"

#property indicator_type2 DRAW_LINE
#property indicator_color2 clrChocolate
#property indicator_width2 1
#property indicator_label2 "St'Signal"
#property indicator_style2 STYLE_DOT
```

在输入变量中，我们会提供所有必需的参数。

```
input int KPeriod = 5;
input int DPeriod = 3;
input int Slowing = 3;
input ENUM_MA_METHOD Method = MODE_SMA;
input ENUM_STO_PRICE StochasticPrice = STO_LOWHIGH;
```

接下来，我们将介绍用于指标缓冲区的数组以及用于描述符的全局变量。

```
double MainBuffer[];
double SignalBuffer[];

int Handle;
```

我们将在 *OnInit* 中进行初始化。

```

int OnInit()
{
    IndicatorSetString(INDICATOR_SHORTNAME,
        StringFormat("Stochastic(%d,%d,%d)", KPeriod, DPeriod, Slowing));
    // binding of arrays as buffers
    SetIndexBuffer(0, MainBuffer);
    SetIndexBuffer(1, SignalBuffer);
    // getting the descriptor Stochastic
    Handle = iStochastic(_Symbol, _Period,
        KPeriod, DPeriod, Slowing, Method, StochasticPrice);
    return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}

```

现在，在 *OnCalculate* 中，一旦句柄准备就绪，我们就需要使用 *CopyBuffer* 函数读取数据。

```

int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
    // waiting for the calculation of the stochastic on all bars
    if(BarsCalculated(Handle) != rates_total)
    {
        return prev_calculated;
    }

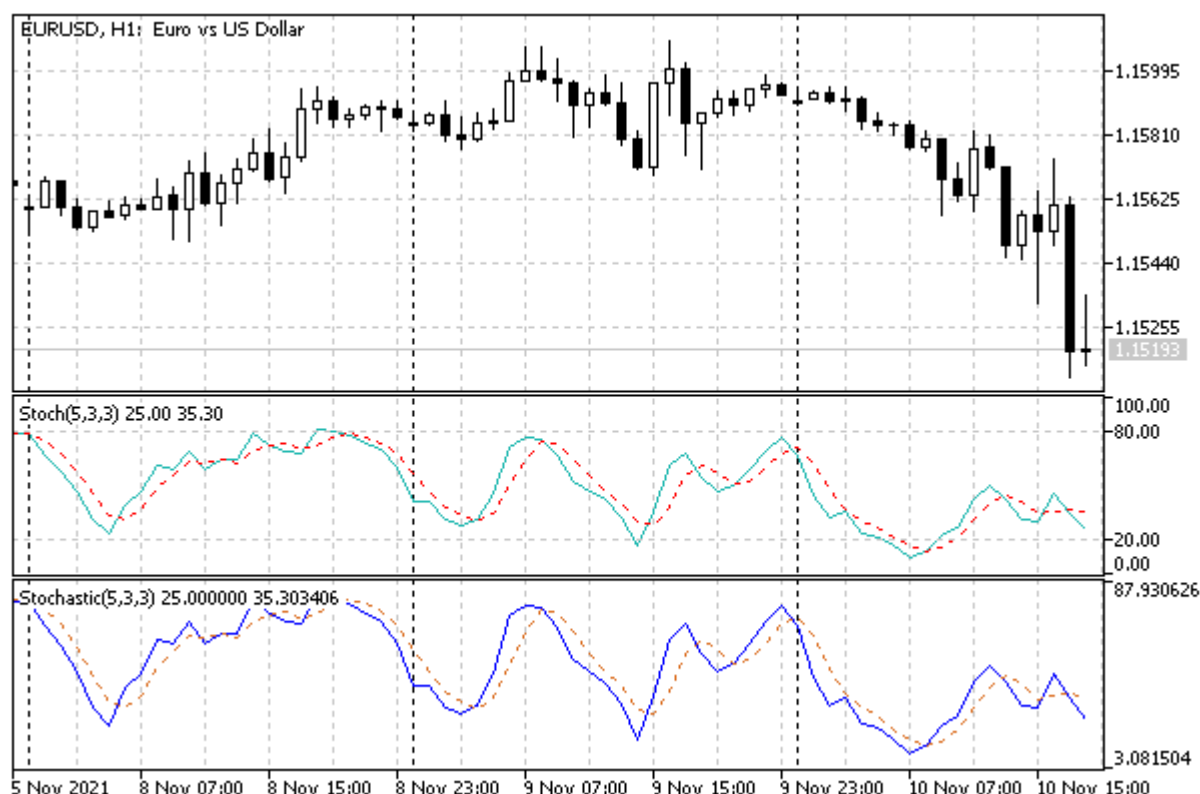
    // copy data to our two buffers
    const int n = CopyBuffer(Handle, 0, 0, rates_total - prev_calculated + 1,
        MainBuffer);
    const int m = CopyBuffer(Handle, 1, 0, rates_total - prev_calculated + 1,
        SignalBuffer);

    return n > -1 && m > -1 ? rates_total : 0;
}

```

请注意，我们要调用两次 *CopyBuffer*：分别针对每个缓冲区（第二个参数中的 0 和 1）。尝试读取一个不存在的缓冲区索引（例如 2），将会生成错误，且我们不会收到任何数据。

我们的指标并非特别有用，因为它没有为原始随机指标添加任何内容，也没有对其读数进行分析。另一方面，我们可以确保标准终端指标的线条与在 MQL5 中创建的指标线条完全一致（正如我们在完全自定义指标中所做的那样，也可以轻松添加水平位和精度设置，但这样一来就很难区分复制指标与原始指标）。



标准随机指标与基于 Stochastic 函数的自定义指标

为了演示终端对指标的缓存机制，在 `OnInit` 函数中添加了几行代码。

```
double array[];
Print("This is very first copy of iStochastic with such settings=",
      !(CopyBuffer(Handle, 0, 0, 10, array) > 0));
```

在这里，我们使用了一个与已知特性相关的技巧：在指标创建后需要一定时间进行计算，因此在获取句柄后无法立即从缓冲区读取数据。这种情况适用于“冷启动”，即当终端内存的缓存中尚未存在指定参数的指标时。如果存在现成的指标，则我们可以立即访问缓冲区。

编译新指标后，应在同一交易品种和时间范围的两个图表上加载该指标的两个副本。第一次加载时，日志中将显示标志为 `true` 的消息（这是第一个副本），而第二次（以及后续多次，如果有多个图表）则会显示为 `false`。你也可以在图表上手动添加标准“Stochastic Oscillator”指标（使用默认设置或后续将在 `Use Stochastic` 中应用的设置），然后再运行 `Use Stochastic`：此时我们同样应获得 `false`。

现在让我们尝试基于标准指标开发一些原创性内容。以下 `UseM1MA.mq5` 指标用于在 M5 及更高时间范围（主要面向日间交易）上计算每根柱线的平均价格。它会累加在工作（更高）时间范围上中每根特定柱线时间戳范围内的 M1 柱线价格。这使得你能够比标准价格类型（`Close`、`Open`、`Median`、`Typical`、`Weighted` 等）更准确地估算柱线的有效价格。此外，我们还将提供在特定周期内对这些价格进行平均的可能性，但需做好心理准备，生成的线条可能不会特别平滑。

该指标将显示在主窗口中，且包含一个单独的缓冲区。可以通过 3 个参数更改设置：

```
input uint _BarLimit = 100; // BarLimit
input uint BarPeriod = 1;
input ENUM_APPLIED_PRICE M1Price = PRICE_CLOSE;
```

BarLimit 设置用于计算的最近历史柱线数量。这一点很重要，因为与分钟级 M1 相比，时间范围很大的图表可能需要处理大量柱线（例如，在 24/7 交易中，一根日线 D1 包含 1440 根 M1 柱线）。这可能导致需要下载额外数据并等待同步完成。先用保守的默认设置（工作时间范围的 100 根柱线）进行测试，再将该参数设为 0，即表示无限制处理。

然而，即使将 **BarLimit** 设置为 0，指标也可能不会针对更早时间范围的全部可见历史进行计算：如果终端对图表中的柱线数量设有限制，则这也会影响对 M1 柱线的请求。换言之，分析深度取决于历史数据中允许的最大 M1 柱线数量所对应的时间范围。

BarPeriod 设置进行平均计算的更大时间范围的柱线数量。此处默认值为 1，这使得能够单独查看每根柱线的有效价格。

M1Price 参数指定用于 M1 柱线计算的价格类型。

在全局范围内，为缓冲区、描述符和自更新标志定义数组，我们需要这些来等待构建“外部”M1 时间范围的时间序列。

```
double Buffer[];

int Handle;
int BarLimit;
bool PendingRefresh;

const string MyName = "M1MA (" + StringSubstr(EnumToString(M1Price), 6)
    + ", " + (string)BarPeriod + "[" + (string)(PeriodSeconds() / 60) + "])";
const uint P = PeriodSeconds() / 60 * BarPeriod;
```

此外，还会在此处生成指标名称和平均周期 **P**。**PeriodSeconds** 函数可返回当前时间范围中单根柱线内的秒数，使用该函数能够计算当前柱线内包含的 M1 柱线数量：**PeriodSeconds() / 60**（60 秒为 M1 柱线的时长）。

通常通过 **OnInit** 执行初始化操作。

```
int OnInit()
{
    IndicatorSetString(INDICATOR_SHORTNAME, MyName);
    IndicatorSetInteger(INDICATOR_DIGITS, _Digits);

    SetIndexBuffer(0, Buffer);

    Handle = iMA(_Symbol, PERIOD_M1, P, 0, MODE_SMA, M1Price);

    return Handle != INVALID_HANDLE ? INIT_SUCCEEDED : INIT_FAILED;
}
```

为获取更高时间范围柱线的平均价格，我们采用简单移动平均线，调用 **iMA** 并设置为 **MODE_SMA** 模式。

下面给出的 **OnCalculate** 函数已进行简化处理。在第一次运行或历史数据发生更改时，我们会清空缓冲区并填充 **BarLimit** 变量（这是必需的，因为输入变量无法编辑，而我们希望将值 0 解释为

可用于计算的最大柱线数量)。在后续调用中, 仅从 `prev_calculated` 位置开始的最后几根柱线数量不超过 `BarLimit`) 清除缓冲区元素。

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
    if(prev_calculated == 0)
    {
        ArrayInitialize(Buffer, EMPTY_VALUE);
        if(_BarLimit == 0
           || _BarLimit > (uint)rates_total)
        {
            BarLimit = rates_total;
        }
        else
        {
            BarLimit = (int)_BarLimit;
        }
    }
    else
    {
        for(int i = fmax(prev_calculated - 1, (int)(rates_total - BarLimit));
            i < rates_total; ++i)
        {
            Buffer[i] = EMPTY_VALUE;
        }
    }
}
```

在从创建的 `iMA` 指标读取数据之前, 需要等待数据准备就绪: 为此我们将 `BarsCalculated` 与 `M1` 柱线数量进行比较。

```
if(BarsCalculated(Handle) != iBars(_Symbol, PERIOD_M1))
{
    if(prev_calculated == 0)
    {
        EventSetTimer(1);
        PendingRefresh = true;
    }
    return prev_calculated;
}
...
```

如果数据尚未就绪, 我们将启动一个计时器, 在 1 秒后再次尝试读取。

接下来, 进入算法的主要计算部分, 因此如果计时器仍在运行, 则必须停止计时器。如果下一个分时报价事件的发生速度快于 1 秒, 而 `iMA M1` 已经完成, 则可能出现这种情况。此时, 合理的做法是调用相应的函数 `EventKillTimer`。但它的行为存在一个细微差别: 它不会清空指标类型 `MQ1` 程序的事件队列, 如果队列中已经存在一个计时器事件, 则 `OnTimer` 处理程序仍会被调用一次。为避免图表的不必要更新, 我们使用自己的变量 `Pending Refresh` 控制这一过程, 在此处将其赋值为 `false`。

```
...
Pending Refresh =false;// data is ready, the timer will idle
...
```

以下是 *OnTimer* 处理程序中的实现方式：

```
void OnTimer()
{
    EventKillTimer();
    if(PendingRefresh)
    {
        ChartSetSymbolPeriod(0, _Symbol, _Period);
    }
}
```

让我们回到 *OnCalculate* 并展示主要工作流程。

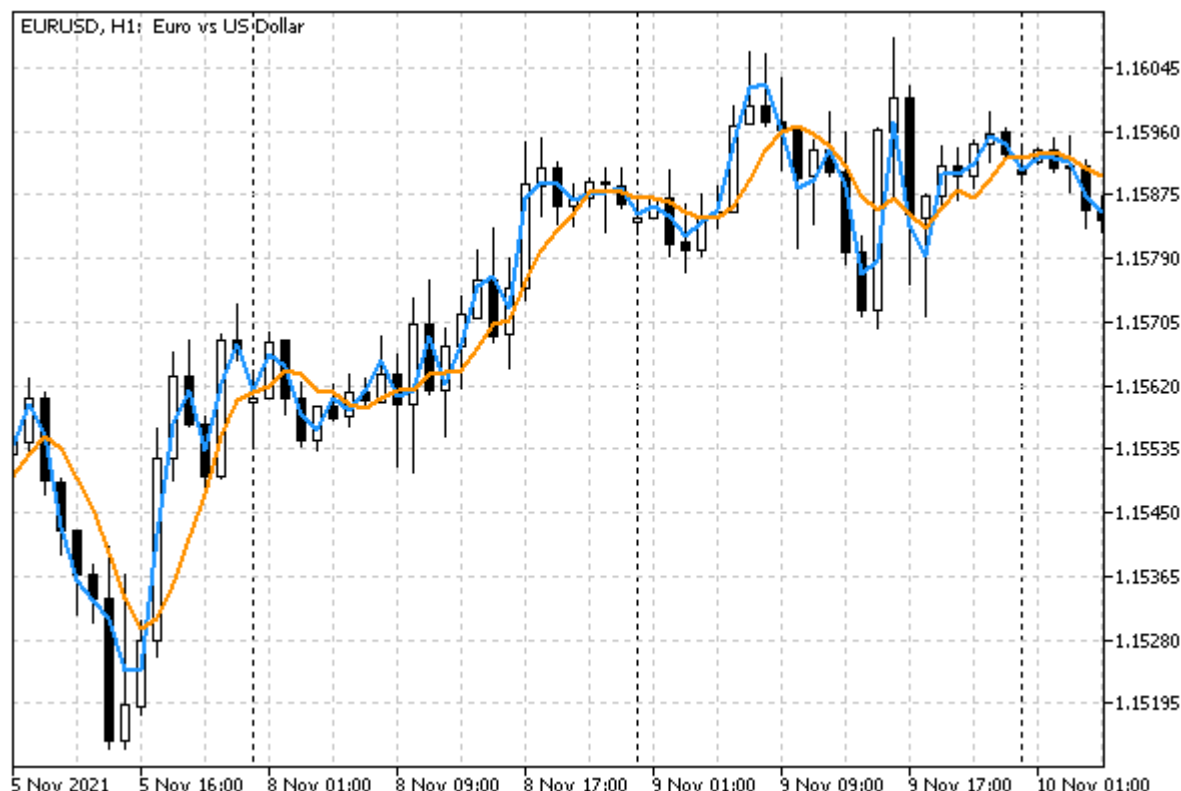
```
for(int i = fmax(prev_calculated - 1, (int)(rates_total - BarLimit));
    i < rates_total; ++i)
{
    static double result[1];

    // get the last bar M1 corresponding to the i-th bar of the current timeframe
    const datetime dt = time[i] + PeriodSeconds() - 60;
    const int bar = iBarShift(_Symbol, PERIOD_M1, dt);

    if(bar > -1)
    {
        // request MA value on M1
        if(CopyBuffer(Handle, 0, bar, 1, result) == 1)
        {
            Buffer[i] = result[0];
        }
        else
        {
            Print("CopyBuffer failed: ", _LastError);
            return prev_calculated;
        }
    }
}

return rates_total;
}
```

该指标的操作通过以下 EURUSD,H1 图表展示：蓝色线条对应默认设置。每个值通过对 60 根柱线 M1 的 PRICE_CLOSE 进行平均计算得出。橙色线条额外采用 5 根 H1 柱线的平滑处理，且基于 M1 PRICE_TYPICAL 计算得出。



EURUSD,H1 上的 UseM1MA 指标的两个实例

本书展示了 *UseM1MASimple.mq5* 的简化版。我们将以下细节留在后台处理：对最后一根（未完成的）柱线进行平均计算、空柱线（即没有对应 M1 数据的柱线）的处理、PLOT_DRAW_BEGIN 特性的正确设置，以及对新柱形出现时平均值计算短期延迟现象的控制。完整版可在文件 *UseM1MA.mq5* 中获取。

作为基于标准指标构建指标的最后一个示例，我们来分析对 *IndUnityPercent.mq5* 指标的改进，该指标曾在 *多货币和多时间范围指标* 章节中介绍过。第一个版本使用 *Close* 价格进行计算，通过 *CopyBuffer* 获取这些价格。在新版 *UseUnityPercentPro.mq5* 中，让我们将该方法替换为读取 *iMA* 指标数据。这将允许我们实现以下新功能：

- 某个给定周期的平均价格
- 选择平均值计算方法
- 选择用于计算的价格类型

源代码的改动是最小的。我们为 *iMA* 句柄添加了 3 个新参数和一个全局数组：

```
input ENUM_APPLIED_PRICE PriceType = PRICE_CLOSE;
input ENUM_MA_METHOD PriceMethod = MODE_EMA;
input int PricePeriod = 1;
...
int Handles[];
```

在辅助函数 *InitSymbols* 中（该函数从 *OnInit* 调用，用于解析包含工作交易品种列表的字符串），我们为新数组添加了内存分配（其 *SymbolCount* 大小由该列表确定）。

```

string InitSymbols()
{
    SymbolCount = StringSplit(Instruments, ',', Symbols);
    ...
    ArrayResize(Handles, SymbolCount);
    ArrayInitialize(Handles, INVALID_HANDLE);
    ...
    for(int i = 0; i < SymbolCount; i++)
    {
        ...
        Handles[i] = iMA(Symbols[i], PERIOD_CURRENT, PricePeriod, 0,
            PriceMethod, PriceType);
    }
}

```

在同一函数末尾，我们将创建所需附属指标的描述符。

在执行主要计算的 *Calculate* 函数中，我们会将以下形式的调用：

```
CopyClose(Symbols[j], _Period, time0, time1, w);
```

替换为：

```
CopyBuffer(Handles[j], 0, time0, time1, w); // j-th handle, 0-th buffer
```

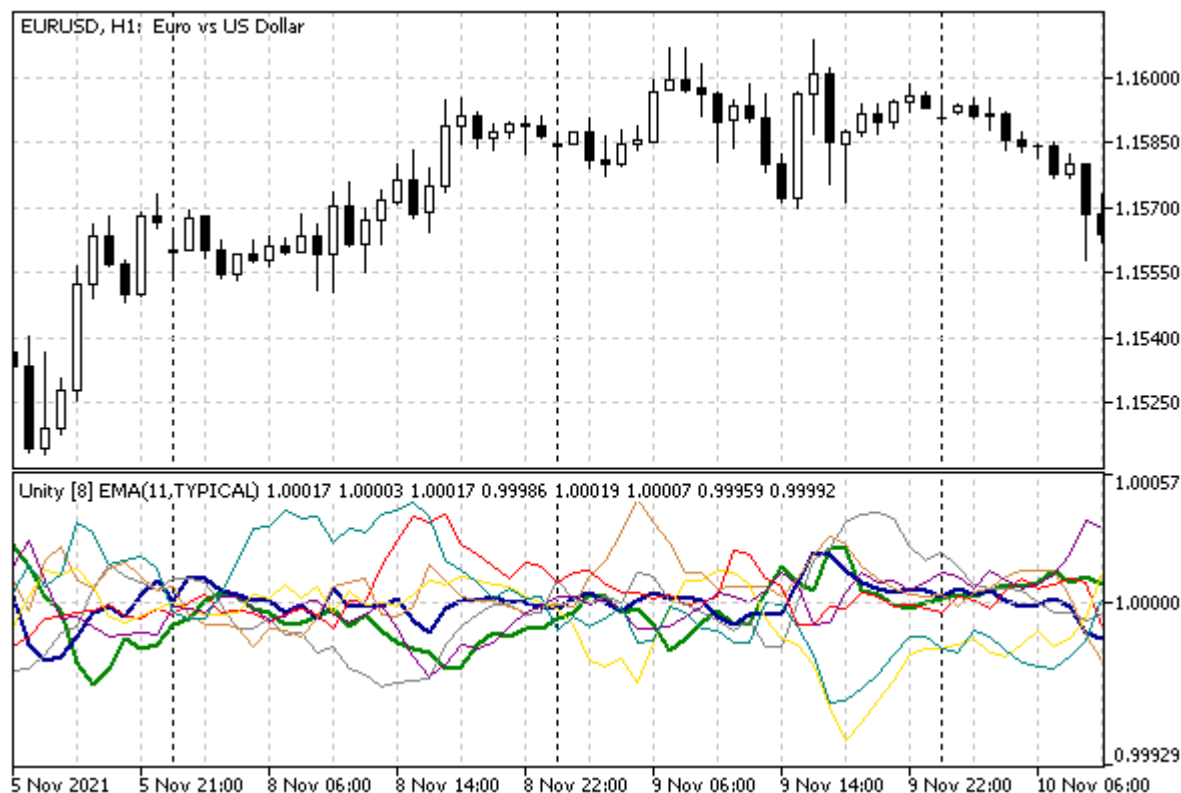
为清晰起见，我们还为指标的简称补充了三个新参数。

```

IndicatorSetString(INDICATOR_SHORTNAME,
    StringFormat("Unity [%d] %s(%d,%s)", workCurrencies.getSize(),
        StringSubstr(EnumToString(PriceMethod), 5), PricePeriod,
        StringSubstr(EnumToString(PriceType), 6)));

```

结果如下：



适用于主要外汇货币对的 UseUnityPercentPro 多交易品种指标

此处显示的是一篮子 8 种主要外汇货币 (默认设置), 采用 11 根柱线并基于 *typical* 价计算出的平均值。两条粗线对应当前图表中货币的相对价值: 欧元以蓝色标示, 美元以绿色标示。

5.5.8 指标创建的高级方式: IndicatorCreate

在创建指标时, 无论是使用 *iCustom* 函数还是任意内置指标函数, 都需要在编码阶段掌握参数列表的信息。但在实际应用中, 常常需要编写足够灵活的程序, 以便在指标之间进行替换。

例如, 在 [测试程序](#) 中优化 EA 交易时, 不仅需选择移动平均线的周期, 还需选择其计算算法, 这些都是合理的。当然, 如果我们基于单一指标 *iMA* 构建算法, 你可以在其方法设置中提供指定 `ENUM_MA_METHOD` 的能力。但有人可能希望能在双指数移动平均线、三指数移动平均线和分形移动平均线之间进行切换, 从而扩大选择范围。初看来, 这可以通过 *switch* 分别调用 *DEMA*、*iTEMA* 和 *iFrAMA* 来实现。但如何将自定义指标纳入该列表?

尽管在 *iCustom* 调用中可以轻松替换指标名称, 但参数列表可能存在显著差异。通常情况下, EA 交易可能需要基于任意技术指标组合生成信号, 而这些指标并非预先已知, 且不仅限于移动平均线。

针对此类情况, MQL5 提供了一种通用方法, 即通过 *IndicatorCreate* 函数创建任意技术指标。

```
int IndicatorCreate(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_INDICATOR indicator,
int count = 0, const MqlParam &parameters[] = NULL)
```

该函数为指定交易品种和时间范围创建指标实例。指标类型通过 *indicator* 参数设置。其类型为 `ENUM_INDICATOR` 枚举 (详见下文), 其中包含所有 [内置指标](#) 的标识符以及用于 *iCustom* 的选项。指标参数的数量及其描述分别通过 *count* 参数和 *MqlParam* 结构体数组 (见下文) 进行传递。

该数组的每个元素描述所创建指标的相应输入参数，因此元素的内容和顺序必须与以下内容相对应：对于内置指标函数，需与原型相对应，如果是自定义指标，则需与其源代码中的输入变量描述符相对应。

违反此规则可能导致程序执行阶段出错（见下文示例），且无法创建句柄。在最坏的情况下，传递的参数将被错误解释，指标表现异常，但由于缺少报错，这类问题难以察觉。例外情况是当传递空数组或完全不传递时（因为参数 *count*和*parameters* 为可选参数）：此时指标将使用默认设置创建。此外，对于自定义指标，你还可以从参数列表末尾省略任意数量的参数。

*MqlParam*结构体专门用于以下场景：通过 *IndicatorCreate* 创建指标时传递输入参数；通过 *IndicatorParameters* 获取（在图表上执行）第三方指标的参数信息。

```
struct MqlParam
{
    ENUM_DATATYPE type;           // input parameter type
    long          integer_value;  // field for storing an integer value
    double        double_value;   // field for storing double or float values
    string        string_value;   // field for storing a value of string type
};
```

参数的实际值必须根据第一个 *type*字段的值，在 *integer_value*、*double_value*、*string_value* 其中一个字段进行设置。而 *type*字段使用 *ENUM_DATATYPE* 枚举定义，该枚举包含所有 *MT5* 内置类型的标识符。

标识符	数据类型
TYPE_BOOL	bool
TYPE_CHAR	char
TYPE_UCHAR	uchar
TYPE_SHORT	short
TYPE_USHORT	ushort
TYPE_COLOR	color
TYPE_INT	int
TYPE_UINT	uint
TYPE_DATETIME	datetime
TYPE_LONG	long
TYPE_ULONG	ulong
TYPE_FLOAT	float
TYPE_DOUBLE	double
TYPE_STRING	string

如果指标参数为枚举类型，需在 *type*字段中使用 *TYPE_INT* 值进行描述。

第三个参数 *IndicatorCreate* 中用于指定指标类型的 `ENUM_INDICATOR` 枚举包含以下常量：

标识符	指标
IND_AC	Accelerator Oscillator
IND_AD	Accumulation/Distribution
IND_ADX	Average Directional Index
IND_ADXW	ADX by Welles Wilder
IND_ALLIGATOR	Alligator
IND_AMA	Adaptive Moving Average
IND_AO	Awesome Oscillator
IND_ATR	Average True Range
IND_BANDS	Bollinger Bands®
IND_BEARS	Bears Power
IND_BULLS	Bulls Power
IND_BWMFI	Market Facilitation Index
IND_CCI	Commodity Channel Index
IND_CHAIKIN	Chaikin Oscillator
IND_CUSTOM	自定义指标
IND_DEMA	Double Exponential Moving Average
IND_DEMARKER	DeMarker
IND_ENVELOPES	Envelopes
IND_FORCE	Force Index
IND_FRACTALS	Fractals
IND_FRAMA	Fractal Adaptive Moving Average
IND_GATOR	Gator Oscillator
IND_ICHIMOKU	Ichimoku Kinko Hyo
IND_MA	Moving Average
IND_MACD	MACD
IND_MFI	Money Flow Index
IND_MOMENTUM	Momentum

标识符	指标
IND_OBV	On Balance Volume
IND_OSMA	OsMA
IND_RSI	Relative Strength Index
IND_RVI	Relative Vigor Index
IND_SAR	Parabolic SAR
IND_STDDEV	Standard Deviation
IND_STOCHASTIC	Stochastic Oscillator
IND_TEMA	Triple Exponential Moving Average
IND_TRIX	Triple Exponential Moving Averages Oscillator
IND_VIDYA	Variable Index Dynamic Average
IND_VOLUMES	Volumes
IND_WPR	Williams Percent Range

需要注意的是，如果将 `IND_CUSTOM` 值作为指标类型传递，则参数数组的第一个元素必须包含 `type` 字段且其值为 `TYPE_STRING`，同时 `string_value` 字段必须包含自定义指标的名称（路径）。

如果成功，`IndicatorCreate` 函数将返回所创建指标的句柄；如果失败，则返回 `INVALID_HANDLE`。错误代码将通过 `_LastError` 提供。

需注意，要测试创建自定义指标且指标名称在编译阶段未知的 MQL 程序（使用 `IndicatorCreate` 时也通常如此），必须通过以下指令显式绑定指标：

```
#property tester_indicator "indicator_name.ex5"
```

这允许测试程序向测试代理发送所需的辅助指标，但该过程仅限于预先已知的指标。

我们来看看几个示例。我们从一个简单应用开始：使用 `IndicatorCreate` 替代已知函数，然后为了展示新方法的灵活性，我们将创建一个通用包装指标，用于可视化任意内置或自定义指标。

`UseEnvelopesParams1.mq5` 的第一个示例创建了 `Envelopes` 指标的嵌入式副本。为此，我们需要定义两个缓冲区、两个绘图、它们的数组、以及重复 `iEnvelopes` 参数的输入参数。

```

#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots 2

// drawing settings
#property indicator_type1 DRAW_LINE
#property indicator_color1 clrBlue
#property indicator_width1 1
#property indicator_label1 "Upper"
#property indicator_style1 STYLE_DOT

#property indicator_type2 DRAW_LINE
#property indicator_color2 clrRed
#property indicator_width2 1
#property indicator_label2 "Lower"
#property indicator_style2 STYLE_DOT

input int WorkPeriod = 14;
input int Shift = 0;
input ENUM_MA_METHOD Method = MODE_EMA;
input ENUM_APPLIED_PRICE Price = PRICE_TYPICAL;
input double Deviation = 0.1; // Deviation, %

double UpBuffer[];
double DownBuffer[];

int Handle; // handle of the subordinate indicator

```

如果你使用 *iEnvelopes* 函数，*OnInit* 处理程序可能如下所示。

```

int OnInit()
{
    SetIndexBuffer(0, UpBuffer);
    SetIndexBuffer(1, DownBuffer);

    Handle = iEnvelopes(WorkPeriod, Shift, Method, Price, Deviation);
    return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}

```

缓冲区绑定方式保持不变，但现在我们将采用另一种方式创建句柄。我们来描述 *MqlParam* 数组、填充该数组并调用 *IndicatorCreate* 函数。

```

int OnInit()
{
    ...
    MqlParam params[5] = {};
    params[0].type = TYPE_INT;
    params[0].integer_value = WorkPeriod;
    params[1].type = TYPE_INT;
    params[1].integer_value = Shift;
    params[2].type = TYPE_INT;
    params[2].integer_value = Method;
    params[3].type = TYPE_INT;
    params[3].integer_value = Price;
    params[4].type = TYPE_DOUBLE;
    params[4].double_value = Deviation;
    Handle = IndicatorCreate(_Symbol, _Period, IND_ENVELOPES,
        ArraySize(params), params);
    return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}

```

收到句柄后，我们将在 *OnCalculate* 中使用它来填充其两个缓冲区。

```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &data[])
{
    if(BarsCalculated(Handle) != rates_total)
    {
        return prev_calculated;
    }

    const int n = CopyBuffer(Handle, 0, 0, rates_total - prev_calculated + 1, UpBuffer);
    const int m = CopyBuffer(Handle, 1, 0, rates_total - prev_calculated + 1, DownBuffer);

    return n > -1 && m > -1 ? rates_total : 0;
}

```

我们来验证创建的 *UseEnvelopesParams1* 指标在图表上的显示效果。



UseEnvelopesParam s1 指标

上述方法虽然标准，但并非填充特性的最佳方法。由于许多项目中可能需要调用 *IndicatorCreate*，因此有必要简化调用代码的流程，这是合理的。为此，我们将开发一个名为 *MqlParamBuilder* 的类（请参阅文件 *MqlParamBuilder.mqh*）。它的任务是，通过特定方法接收参数值，判断参数类型，并向数组中添加合适的元素（正确填充的结构体）。

MQL5 并不完全支持运行时类型信息（RTTI）概念。通过该机制，程序可以在运行时查询其组成部分的描述性元数据，包括变量、结构、类、函数等。MQL5 中可归为 RTTI 的几个内置功能是运算符 *typename* 和 *offsetof*。由于 *typename* 运算符可以字符串形式返回类型名称，我们将基于字符串构建类型自动检测器（请参阅文件 *RTTI.mqh*）。

```

template<typename T>
ENUM_DATATYPE rtti(T v = (T)NULL)
{
    static string types[] =
    {
        "null",      //          (0)
        "bool",      // 0 TYPE_BOOL=1 (1)
        "char",      // 1 TYPE_CHAR=2 (2)
        "uchar",     // 2 TYPE_UCHAR=3 (3)
        "short",     // 3 TYPE_SHORT=4 (4)
        "ushort",    // 4 TYPE_USHORT=5 (5)
        "color",     // 5 TYPE_COLOR=6 (6)
        "int",       // 6 TYPE_INT=7 (7)
        "uint",      // 7 TYPE_UINT=8 (8)
        "datetime",  // 8 TYPE_DATETIME=9 (9)
        "long",      // 9 TYPE_LONG=10 (A)
        "ulong",     // 10 TYPE_ULONG=11 (B)
        "float",     // 11 TYPE_FLOAT=12 (C)
        "double",    // 12 TYPE_DOUBLE=13 (D)
        "string",    // 13 TYPE_STRING=14 (E)
    };
    const string t = typename(T);
    for(int i = 0; i < ArraySize(types); ++i)
    {
        if(types[i] == t)
        {
            return (ENUM_DATATYPE)i;
        }
    }
    return (ENUM_DATATYPE)0;
}

```

模板函数 *rtti* 使用 *typename* 获取模板类型参数名称的字符串，然后将其与一个数组的元素进行比较，该数组包含 *ENUM_DATATYPE* 枚举中的所有内置类型。在该数组中，名称的枚举顺序与枚举元素的值相对应，因此当找到匹配的字符串时，只需将索引转换为 (*ENUM_DATATYPE*) 类型并返回给调用代码即可。例如，调用 *rtti(1.0)* 或 *rtti<double>()* 将返回 *TYPE_DOUBLE* 值。

利用该工具，我们可以回到 *MqlParamBuilder* 的开发。在该类中，我们将定义 *MqlParam* 结构体数组以及 *n* 变量，该变量将包含最后一个待填充元素的索引。

```

class MqlParamBuilder
{
protected:
    MqlParam array[];
    int n;
    ...
}

```

我们将用于向参数列表添加下一个值的公共方法设计为模板方法。此外，我们将其实现为运算符 '<<' 的重载，并返回指向 "builder" 对象本身的指针。这将允许在一行中向数组写入多个值，例如，像这样：*builder << WorkPeriod << PriceType << SmoothingMode*。

正是在该方法中，我们将增大数组的大小，获取待填充的有效索引 *n*，并且立即重置第 *n*- 个结构。

```

...
public:
    template<typename T>
    MqlParamBuilder *operator<<(T v)
    {
        // expand the array
        n = ArraySize(array);
        ArrayResize(array, n + 1);
        ZeroMemory(array[n]);
        ...
        return &this;
    }

```

在省略号的位置，后续将是主要工作部分，即填充结构体的各个字段。可以假定我们将使用自制的 *rtti* 直接确定参数类型。但你应注意一个细微之处。如果我们编写指令 *array[n].type = rtti(v)*，则对于枚举，它将无法正确处理。每个枚举都是独立类型，拥有自己的名称，尽管它们的存储方式和整数相同。对于枚举类型，*rtti* 函数将返回 0，因此需要显式地将其替换为 *TYPE_INT*。

```

...
// define value type
array[n].type = rtti(v);
if(array[n].type == 0) array[n].type = TYPE_INT; // imply enum
...

```

现在，我们只需将值 *v* 放入结构体的三个字段之一：*integer_value* 类型为 *long* 的字段（注意，*long* 是长整数，因此该字段得名）、类型为 *double* 的 *double_value* 字段，或类型为 *string* 的 *string_value* 字段。同时，内置类型的数量更多，因此我们假定所有整数类型（包括 *int*、*short*、*char*、*color*、*datetime* 和枚举）必须归入 *integer_value* 字段，*float* 值必须归入 *double_value* 字段，而 *string_value* 字段只有一种明确解释：它始终为 *string*。

为完成该任务，我们将实现多个重载的 *assign* 方法：三个针对特定类型 *float*、*double* 和 *string* 的方法，以及一个适用于其他所有类型的方法。

```

class MqlParamBuilder
{
protected:
    ...
    void assign(const float v)
    {
        array[n].double_value = v;
    }

    void assign(const double v)
    {
        array[n].double_value = v;
    }

    void assign(const string v)
    {
        array[n].string_value = v;
    }

    // here we process int, enum, color, datetime, etc. compatible with long
    template<typename T>
    void assign(const T v)
    {
        array[n].integer_value = v;
    }
    ...

```

这就完成了填充结构体的过程，接下来的问题是将生成的数组传递给调用代码。该操作被分配给一个重载了运算符 `>>` 的公共方法，该方法有一个参数：对接收数组 *MqlParam* 的引用。

```

// export the inner array to the outside
void operator>>(MqlParam &params[])
{
    ArraySwap(array, params);
}

```

现在一切准备就绪，我们可以处理修改后的指标 *UseEnvelopesParams2.mq5* 的源代码。与第一版相比，改动仅涉及 *OnInit* 处理程序中对 *MqlParam* 数组的填充。在其中，我们描述 “builder” 对象，并通过 `<<` 和 return the finished array via `>>` 将参数发送给它。整个过程只需一行代码。


```

int OnInit()
{
    ...
    MqlParam params[];
    MqlParamBuilder builder;
    builder << WorkPeriod << Shift << Method << Price << Deviation >> params;
    ArrayPrint(params);
    /*
        [type] [integer_value] [double_value] [string_value]
    [0]      7              14          0.00000 null      <- "INT" period
    [1]      7              0           0.00000 null      <- "INT" shift
    [2]      7              1           0.00000 null      <- "INT" EMA
    [3]      7              6           0.00000 null      <- "INT" TYPICAL
    [4]     13              0           0.10000 null      <- "DOUBLE" deviation
    */
}

```

为了进行控制，我们将数组输出至日志（上方显示默认值的结果）

如果数组未完整填充，调用 *IndicatorCreate* 将返回错误。例如，当使用 *Envelopes* 指标时，如果只传入 5 个必需参数中的 3 个，将返回错误代码 4002，并得到一个无效句柄。

```

Handle = PRTF(IndicatorCreate(_Symbol, _Period, IND_ENVELOPES, 3, params));
// Error example:
// indicator Envelopes cannot load [4002]
// IndicatorCreate(_Symbol,_Period,IND_ENVELOPES,3,params)=
-1 / WRONG_INTERNAL_PARAMETER(4002)

```

不过，如果数组长度超过指标规范不会被视为错误：超出的值将被直接忽略。

需要注意的是，当值类型与预期参数类型不一致时，系统将执行隐式类型转换，这通常不会触发明显的错误，但生成的指标可能无法按预期运行。例如，如果我们向指标发送的不是 *Deviation* 而是字符串，该字符串将被解释为数字 0，导致“包络线”失效：两条线将与中线对齐，而中线的偏移量是由 *Deviation*（以百分比表示）的大小决定的。同样，如果在需要整数的参数中传递含小数部分的实数，该数值将被四舍五入。

当然，我们会保留正确版本的 *IndicatorCreate* 调用，从而获得有效的指标，就像第一版一样。

```

...
Handle = PRTF(IndicatorCreate(_Symbol, _Period, IND_ENVELOPES,
    ArraySize(params), params));
// success:
// IndicatorCreate(_Symbol,_Period,IND_ENVELOPES,ArraySize(params),params)=10 / ok
return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}

```

从运行表现来看，新指标与之前的版本没有区别。

5.5.9 使用 IndicatorCreate 灵活创建指标

在了解了创建指标的新方法后，我们来看看更贴近实际的任务。*IndicatorCreate* 通常用于被调用指标预先未知的情况。例如，在编写能够基于用户配置的任意信号进行交易的通用 EA 交易时，就会有此类需求。甚至连指标名称都可以由用户设置。

我们尚未准备好开发 EA 交易，因此将通过封装指标 *UseDemoAll.mq5* 的示例来研究这项技术，该指标能够显示任何其他指标的数据。

整个过程应如下所示：当我们在图表上运行 *UseDemoAll* 时，特性对话框中会显示一个列表，我们需要从中选择一个内置指标或自定义指标，如果选择后者，还需在输入字段中额外指定该指标名称。在另一个字符串参数中，我们可以输入用逗号分隔的参数列表。参数类型将根据其拼写自动确定。例如，带小数点的数字 (10.0) 会被视为双精度浮点数，不带小数点的数字 (15) 视为整数，用引号括起来的内容 ("text") 则视为字符串。

这些只是 *UseDemoAll* 的基础设置，并非全部可能的设置。我们稍后会考虑其他设置。

让我们将 `ENUM_INDICATOR` 枚举作为解决方案的基础：它已包含所有指标类型的元素，包括自定义指标 (`IND_CUSTOM`)。但坦白说，由于几个原因，这个枚举的原生形式不适合直接使用。首先，无法从中获取特定指标的元数据，例如自变量的数量和类型、缓冲区数量以及指标显示窗口（主窗口还是子窗口）。这些信息对于正确创建和可视化指标至关重要。其次，如果我们定义一个 `ENUM_INDICATOR` 类型的输入变量，以便用户可以选择所需指标，则在特性对话框中，该变量将通过下拉列表呈现，其中选项仅包含元素名称。实际上，最好能在该列表中为用户提供提示（至少是关于参数的提示）。因此，我们将描述自己的枚举 *IndicatorType*。回想一下，*MT5* 允许为每个元素在右侧指定注释，该注释将在界面中显示。

在 *IndicatorType* 枚举的每个元素中，我们不仅要编码来自 `ENUM_INDICATOR` 的相应标识符 (`ID`)，还要编码参数数量 (`P`)、缓冲区数量 (`B`) 和工作窗口编号 (`W`)。为此，我们开发了以下宏。

```
#define MAKE_IND(P,B,W,ID) ((int)((W << 24) | ((B & 0xFF) << 16) | ((P & 0xFF) << 8) |
#define IND_PARAMS(X)      ((X >> 8) & 0xFF)
#define IND_BUFFERS(X)     ((X >> 16) & 0xFF)
#define IND_WINDOW(X)      ((uchar)(X >> 24))
#define IND_ID(X)           ((ENUM_INDICATOR)(X & 0xFF))
```

`MAKE_IND` 宏将上述所有特征作为参数，并将它们打包到单一 4 字节整数的不同字节中，从而为新枚举的元素形成唯一代码。其余 4 个宏则允许执行反向操作，即使用该代码计算指标的所有特征。

此处，我们不会提供完整的 *IndicatorType* 枚举，只提供其中一部分。完整的源代码可在 *AutoIndicator.mqh* 文件中找到。

```

enum IndicatorType
{
    iCustom_ = MAKE_IND(0, 0, 0, IND_CUSTOM), // {iCustom}(...) [?]

    iAC_ = MAKE_IND(0, 1, 1, IND_AC), // iAC( ) [1]*
    iAD_volume = MAKE_IND(1, 1, 1, IND_AD), // iAD(volume) [1]*
    iADX_period = MAKE_IND(1, 3, 1, IND_ADX), // iADX(period) [3]*
    iADXWilder_period = MAKE_IND(1, 3, 1, IND_ADXW), // iADXWilder(period) [3]*
    ...
    iMomentum_period_price = MAKE_IND(2, 1, 1, IND_MOMENTUM), // iMomentum(period, price) [2]*
    iMFI_period_volume = MAKE_IND(2, 1, 1, IND_MFI), // iMFI(period, volume) [1]*
    iMA_period_shift_method_price = MAKE_IND(4, 1, 0, IND_MA), // iMA(period, shift, method, price) [4]*
    iMACD_fast_slow_signal_price = MAKE_IND(4, 2, 1, IND_MACD), // iMACD(fast, slow, signal, price) [4]*
    ...
    iTEMA_period_shift_price = MAKE_IND(3, 1, 0, IND_TEMA), // iTEMA(period, shift, price) [3]*
    iVolumes_volume = MAKE_IND(1, 1, 1, IND_VOLUMES), // iVolumes(volume) [1]*
    iWPR_period = MAKE_IND(1, 1, 1, IND_WPR) // iWPR(period) [1]*
};

```

注释将成为对用户可见的下拉列表元素，其中显示带命名参数的原型、方括号内的缓冲区数量，以及显示在独立窗口中的指标的星标标记。这些标识符本身也具有信息性，因为它们会被 `EnumToString` 函数转换为文本，该函数用于向日志输出消息。

参数列表尤为重要，因为用户需要将相应的逗号分隔值输入到为此预留的输入变量中。我们本可以同时显示参数类型，但为简洁起见，决定只保留具有含义的名称，而类型也可从名称中推断出来。例如，*period*、*fast*、*slow* 是表示周期（柱线数量）的整数，*method* 是均线方法 `ENUM_MA_METHOD`，*price* 是价格类型 `ENUM_APPLIED_PRICE`，*volume* 是成交量类型 `ENUM_APPLIED_VOLUME`。

为方便用户使用（无需记忆枚举元素的数值），程序将支持所有枚举的名称。特别是，标识符 *sma* 对应 `MODE_SMA`，*ema* 对应 `MODE_EMA`，依此类推。*close* 价会转换为 `PRICE_CLOSE`，*open* 会转换为 `PRICE_OPEN`，其他价格类型也遵循相同规则，即均根据枚举元素标识符中最后一个单词（下划线之后）。例如，对于 *MA* 指标的参数列表（*MA_period_shift_method_price*），可以输入以下值：*11,0,sma,close*。标识符无需加引号。但如有必要，也可以传递包含相同文本的字符串，例如列表 *1.5,"close"* 包含实数 1.5 和字符串 "close"。

指标类型、参数字符串列表以及可选的名称（如果指标是自定义指标）是 `AutoIndicator` 类构造函数的主要数据。

```

class AutoIndicator
{
protected:
    IndicatorType type;           // selected indicator type
    string symbols;              // working symbol (optional)
    ENUM_TIMEFRAMES tf;         // working timeframe (optional)
    MqlParamBuilder builder;     // "builder" of the parameter array
    int handle;                  // indicator handle
    string name;                 // custom indicator name
    ...
public:
    AutoIndicator(const IndicatorType t, const string custom, const string parameters,
        const string s = NULL, const ENUM_TIMEFRAMES p = 0):
        type(t), name(custom), symbol(s), tf(p), handle(INVALID_HANDLE)
    {
        PrintFormat("Initializing %s(%s) %s, %s",
            (type == iCustom_ ? name : EnumToString(type)), parameters,
            (symbol == NULL ? _Symbol : symbol), EnumToString(tf == 0 ? _Period : tf));
        // split the string into an array of parameters (formed inside the builder)
        parseParameters(parameters);
        // create and store the handle
        handle = create();
    }

    int getHandle() const
    {
        return handle;
    }
};

```

此处及下文省略了部分涉及校验输入数据正确性的代码片段完整源代码详见本书随附资料。

分析参数字符串的过程由 *parseParameters* 方法处理。它实现了上述方案，可识别值的类型并将其传递给 *MqlParamBuilder* 对象，我们在之前的示例中已见过该对象。

```

int parseParameters(const string &list)
{
    string sparams[];
    const int n = StringSplit(list, ',', sparams);

    for(int i = 0; i < n; i++)
    {
        // normalization of the string (remove spaces, convert to lower case)
        StringTrimLeft(sparams[i]);
        StringTrimRight(sparams[i]);
        StringToLower(sparams[i]);

        if(StringGetCharacter(sparams[i], 0) == '"'
        && StringGetCharacter(sparams[i], StringLen(sparams[i]) - 1) == '"')
        {
            // everything inside quotes is taken as a string
            builder << StringSubstr(sparams[i], 1, StringLen(sparams[i]) - 2);
        }
        else
        {
            string part[];
            int p = StringSplit(sparams[i], '.', part);
            if(p == 2) // double/float
            {
                builder << StringToDouble(sparams[i]);
            }
            else if(p == 3) // datetime
            {
                builder << StringToTime(sparams[i]);
            }
            else if(sparams[i] == "true")
            {
                builder << true;
            }
            else if(sparams[i] == "false")
            {
                builder << false;
            }
            else // int
            {
                int x = lookUpLiterals(sparams[i]);
                if(x == -1)
                {
                    x = (int)StringToInteger(sparams[i]);
                }
                builder << x;
            }
        }
    }

    return n;
}

```

```
}
```

辅助函数 *lookUpLiterals* 可将标识符转换为标准枚举常量。

```
int lookUpLiterals(const string &s)
{
    if(s == "sma") return MODE_SMA;
    else if(s == "ema") return MODE_EMA;
    else if(s == "smma") return MODE_SMMMA;
    else if(s == "lwma") return MODE_LWMA;

    else if(s == "close") return PRICE_CLOSE;
    else if(s == "open") return PRICE_OPEN;
    else if(s == "high") return PRICE_HIGH;
    else if(s == "low") return PRICE_LOW;
    else if(s == "median") return PRICE_MEDIAN;
    else if(s == "typical") return PRICE_TYPICAL;
    else if(s == "weighted") return PRICE_WEIGHTED;

    else if(s == "lowhigh") return STO_LOWHIGH;
    else if(s == "closeclose") return STO_CLOSECLOSE;

    else if(s == "tick") return VOLUME_TICK;
    else if(s == "real") return VOLUME_REAL;

    return -1;
}
```

在识别参数并将其保存到对象的内部数组 *MqlParamBuilder* 后，将调用 *create* 方法。其目的是将参数复制到本地数组，补充自定义指标的名称（如有），并调用 *IndicatorCreate* 函数。

```
int create()
{
    MqlParam p[];
    // fill 'p' array with parameters collected by 'builder' object
    builder >> p;

    if(type == iCustom_)
    {
        // insert the name of the custom indicator at the very beginning
        ArraySetAsSeries(p, true);
        const int n = ArraySize(p);
        ArrayResize(p, n + 1);
        p[n].type = TYPE_STRING;
        p[n].string_value = name;
        ArraySetAsSeries(p, false);
    }

    return IndicatorCreate(symbol, tf, IND_ID(type), ArraySize(p), p);
}
```

该方法将返回所获得的句柄。

特别值得注意的是，如何将包含自定义指标名称的额外字符串参数插入到数组的最起始位置。首先，将数组的索引顺序指定为“时间序列模式”（请参阅 [ArraySetAsSeries](#)），结果，最后一个元素（根据内存物理存储位置）的索引变为 0，且元素从右向左计数。随后，数组大小增大，并将指标名称写入新增元素中。由于采用反向索引，新增元素并非在现有元素的右侧，而是在现有元素的左侧。最后，将数组恢复为常规索引顺序，此时索引 0 位置存储的正是刚刚作为最后一个元素添加的字符串。

*AutoIndicator*类可以选择性地根据枚举元素名称形成内置指标的缩写名称。

```
...
string getName() const
{
    if(type != iCustom_)
    {
        const string s = EnumToString(type);
        const int p = StringFind(s, "_");
        if(p > 0) return StringSubstr(s, 0, p);
        return s;
    }
    return name;
}
};
```

现在，一切准备就绪，可以直接查看源代码 *UseDemoAll.mq5*。不过，让我们先从略微简化版 *UseDemoAllSimple.mq5* 开始。

首先，我们定义指标缓冲区的数量。由于内置指标中缓冲区最大数量为 5（对于 *Ichimoku*），因此我们以此为上限。我们会将注册这些数组为缓冲区的工作，交给我们已经熟悉的 *BufferArray* 类来完成（请参阅 [多货币和多时间范围指标](#) 章节，示例 *IndUnityPercent*）。

```
#define BUF_NUM 5

#property indicator_chart_window
#property indicator_buffers BUF_NUM
#property indicator_plots BUF_NUM

#include <MQL5Book/IndBufArray.mqh>

BufferArray buffers(5);
```

重要的是要记住，指标可以设计为在主窗口中显示或在独立窗口中显示。MQL5 不允许同时使用这两种模式。然而，由于我们无法预知用户将选择哪种指标，因此需要想出某种“变通方法”。目前，我们先将指标放置在主窗口中，稍后再处理独立窗口的问题。

从纯技术角度而言，将具有 *indicator_separate_window* 特性的指标缓冲区数据复制到主窗口显示的缓冲区中，不存在任何障碍。但需要注意的是，这类指标的数值范围往往与价格刻度不匹配，因此，即便数值仍会输出到 *Data window*，也很可能无法在图表上看到它们（线条将超出可见区域，位于顶部或底部某处）。

我们将通过输入变量来选择指标类型、自定义指标名称以及参数列表。我们还会添加用于渲染类型和线宽的变量。由于缓冲区将根据源指标的缓冲区数量动态连接，因此我们不会使用指令静态描述缓冲区央视，而是在 *OnInit* 中通过调用内置的 *Plot* 函数来完成。

```

input IndicatorType IndicatorSelector = iMA_period_shift_method_price; // Built-in In
input string IndicatorCustom = ""; // Custom Indicator Name
input string IndicatorParameters = "11,0,sma,close"; // Indicator Parameters (comma,s
input ENUM_DRAW_TYPE DrawType = DRAW_LINE; // Drawing Type
input int DrawLineWidth = 1; // Drawing Line Width

```

我们来定义一个全局变量用于存储指标描述符。

```
int Handle;
```

在 *OnInit* 处理程序中，我们使用前面介绍的 *AutoIndicator* 类来解析输入数据、准备 *MqlParam* 数组并基于此获取句柄。

```

#include <MQL5Book/AutoIndicator.mqh>

int OnInit()
{
    AutoIndicator indicator(IndicatorSelector, IndicatorCustom, IndicatorParameters);
    Handle = indicator.getHandle();
    if(Handle == INVALID_HANDLE)
    {
        Alert(StringFormat("Can't create indicator: %s",
            _LastError ? E2S(_LastError) : "The name or number of parameters is incorrec
        return INIT_FAILED;
    }
    ...
}

```

要自定义绘图，我们需要定义一组颜色并从 *AutoIndicator* 对象中获取指标的简称。我们还会使用 `IND_BUFFERS` 宏计算内置指标所使用的 *n* 个缓冲区数量；而对于任何自定义指标（预先未知），如果没有更好解决方案，我们将包括其所有缓冲区。此外，在数据复制过程中，不必要的 *CopyBuffer* 调用将直接返回错误，这类数组可填充为空值。

```

...
static color defColors[BUF_NUM] = {clrBlue, clrGreen, clrRed, clrCyan, clrMagenta}
const string s = indicator.getName();
const int n = (IndicatorSelector != iCustom_) ? IND_BUFFERS(IndicatorSelector) : E
...

```

在循环中，我们将设置图表特性，并考虑上限值 *n*：超出该值的缓冲区将被隐藏。


```

for(int i = 0; i < BUF_NUM; ++i)
{
    PlotIndexSetString(i, PLOT_LABEL, s + "[" + (string)i + "]");
    PlotIndexSetInteger(i, PLOT_DRAW_TYPE, i < n ? DrawType : DRAW_NONE);
    PlotIndexSetInteger(i, PLOT_LINE_WIDTH, DrawLineWidth);
    PlotIndexSetInteger(i, PLOT_LINE_COLOR, defColors[i]);
    PlotIndexSetInteger(i, PLOT_SHOW_DATA, i < n);
}

Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
        "(", IndicatorParameters, ")");

return INIT_SUCCEEDED;
}

```

图表左上角的注释将显示带参数的指标名称。

在 *OnCalculate* 处理程序中，当句柄准备就绪后，我们会将其读取到自己的数组中。

```

int OnCalculate(ON_CALCULATE_STD_SHORT_PARAM_LIST)
{
    if(BarsCalculated(Handle) != rates_total)
    {
        return prev_calculated;
    }

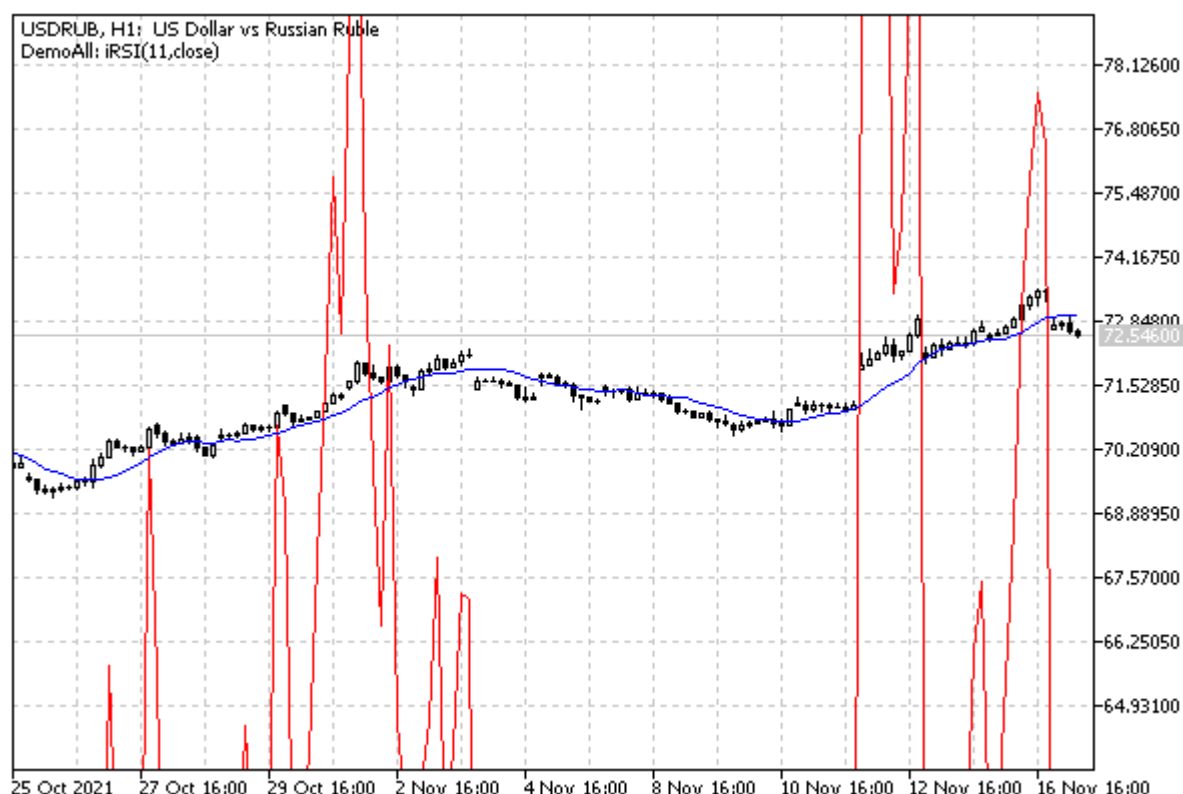
    const int m = (IndicatorSelector != iCustom_) ? IND_BUFFERS(IndicatorSelector) : E
    for(int k = 0; k < m; ++k)
    {
        // fill our buffers with data form the indicator with the 'Handle' handle
        const int n = buffers[k].copy(Handle,
            k, 0, rates_total - prev_calculated + 1);

        // in case of error clean the buffer
        if(n < 0)
        {
            buffers[k].empty(EMPTY_VALUE, prev_calculated, rates_total - prev_calculated
        }
    }

    return rates_total;
}

```

上述实现是简化版，与原始文件 *UseDemoAllSimple.mq5* 保持一致。我们将在后续处理其扩展功能，但现在先检查当前版本的运行情况。下图展示了该指标的两个实例：蓝色线条使用默认设置 *iMA_period_shift_method_price*，选项为 *"11,0,sma,close"*，红色线条为 *iRSI_period_price*，参数为 *"11 close"*。



包含 MA 和 iRSI 读数的 UseDemoAllSimple 指标的两个实例

为了演示，特意选择了 USDRUB 图表，因为此处的报价值与 RSI 指标的范围基本吻合（RSI 指标本应显示在独立窗口中）。在大多数其他交易品种图表上，我们可能看不到 RSI 指标的显示。如果你只关心通过编程方式访问值，则这不算什么大问题，但如果有可视化需求，这就是一个需要解决的问题。

因此，你应以某种方式为子窗口专用的指标提供独立显示。从根本上讲，MQL 开发者社区中存在一个常见需求：支持在主窗口和子窗口中同时显示图形。我们将介绍其中一种解决方案，但为此你需要先了解一些新功能。

5.5.10 图表中指标管理函数概述

我们已经了解到，指标是一类同时兼具计算和可视化的 MQL 程序。计算在内部执行，对用户不可见，而可视化则需要与图表进行关联。正因如此，指标与图表紧密相关，MQL5 API 甚至包含一组用于管理图表指标的函数。我们将在关于 [图表](#) 的章节详细讨论这些函数，在此仅列出其名称。

函数	用途
ChartWindowFind	返回包含当前指标或指定名称指标的子窗口编号
ChartIndicatorAdd	将具有指定句柄的指标添加到指定图表窗口中
ChartIndicatorDelete	从指定图表窗口移除指定名称的指标
ChartIndicatorGet	返回指定图表窗口中指定短名称的指标句柄
ChartIndicatorName	通过指定图表窗口中指标列表中的编号返回指标短名称
ChartIndicatorsTotal	返回附加到指定图表窗口的所有指标数量

在下一节关于主窗口和辅助窗口中 [组合信息输出](#) 的内容中，我们将看到一个示例 *UseDemoAll.mq5*，其中会应用上述部分函数。

5.5.11 将输出组合到主窗口和辅助窗口

让我们回到在主窗口和子窗口中显示同一指标图形的问题，因为我们在开发示例 *UseDemoAllSimple.mq5* 时遇到了该问题。我原本认为，独立窗口不适合在主图表上可视化，而主窗口指标又没有额外窗口。有几种替代方案：

- ① 为独立窗口实现一个父指标，在其中显示图表，并在主窗口中使用它来显示 [图形对象](#) 类型的数据。该方案存在缺陷，因为从对象读取数据的方式和从时间序列读取不同，且许多对象会消耗额外的资源。
- ② 为主窗口开发自定义虚拟面板（类），并在正确缩放的情况下，在其中显示本应在子窗口显示的时间序列数据。
- ③ 使用多个指标（至少一个用于主窗口，一个用于子窗口），并通过共享内存（需调用 [DLL](#)）、[资源](#) 或 [数据库](#) 在指标间交换数据。
- ④ 在主窗口和子窗口的指标中复用计算逻辑（使用公共源代码）。

我们将介绍一种超出单个 MQL 程序限制的解决方案：需要一个具备 *indicator_separate_window* 特性的额外指标。实际上我们已经有了，因为已经通过请求句柄创建了其计算部分。我们只需以某种方式将其显示在独立子窗口中即可。

在新版（完整版）的 *UseDemoAll.mq5* 中，我们将分析通过相应的 *IndicatorType* 枚举元素所请求创建的指标的元数据。回想一下，除其他因素外，每种内置指标的工作窗口类型也在其中进行了编码。当某个指标需要独立窗口时，我们将通过特殊的 MQL5 函数创建一个独立窗口，这些函数我们稍后会详细学习。

目前无法获取自定义指标的工作窗口信息。因此，我们添加一个输入变量 *IndicatorCustomSubwindow*，允许用户声明需要一个子窗口。

```
input bool IndicatorCustomSubwindow = false; // Custom Indicator Subwindow
```

在 *OnInit* 中，我们将隐藏用于子窗口的缓冲区。

```

int OnInit()
{
    ...
    const bool subwindow = (IND_WINDOW(IndicatorSelector) > 0)
        || (IndicatorSelector == iCustom_ && IndicatorCustomSubwindow);
    for(int i = 0; i < BUF_NUM; ++i)
    {
        ...
        PlotIndexSetInteger(i, PLOT_DRAW_TYPE,
            i < n && !subwindow ? DrawType : DRAW_NONE);
    }
    ...
}

```

完成该设置后，还需要调用一些函数，这些函数不仅适用于指标操作，也适用于图表操作。我们将在相应章节中详细学习这些函数，而 [上一节](#) 已对它们进行了介绍性概述。

其中，*ChartIndicatorAdd* 函数允许将句柄指定的指标添加到窗口中，不仅可添加到主要部分，还能添加到子窗口。关于图表标识符和窗口编号，我们将在 [图表](#) 章节中讨论，目前只需知道在下次调用 *ChartIndicatorAdd* 函数时，会将指定 *handle* 的指标添加到当前图表的新建子窗口中。

```

int handle = ... // get indicator handle, iCustom or IndicatorCreate

// set the current chart (0)
// |
// |      set the window number to the current number of windows
// |
// |      |
// |      | passing the descriptor
// |      |
// |      |
// v      v      v
ChartIndicatorAdd( 0, (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL), handle);

```

了解这一可能性后，我们可以考虑调用 *ChartIndicatorAdd*，并向其传递一个已创建的从属指标的句柄。

我们需要的第二个函数是 *ChartIndicatorName*。该函数通过指标句柄返回其简称。该名称对应指标代码中设置的 *INDICATOR_SHORTNAME* 特性，可能与文件名不同。该名称是必需的，用于清理自身的操作，即在删除或重新配置父指标后，移除辅助指标及其子窗口。

```

string subTitle = "";

int OnInit()
{
    ...
    if(subwindow)
    {
        // show a new indicator in the subwindow
        const int w = (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL);
        ChartIndicatorAdd(0, w, Handle);
        // save the name to remove the indicator in OnDeinit
        subTitle = ChartIndicatorName(0, w, 0);
    }
    ...
}

```

在 *OnDeinit* 处理程序中，我们使用保存的 *subTitle* 调用另一个函数 *ChartIndicatorDelete*（后续会详细讲解）。该函数通过最后一个参数指定的指标名称，从图表中移除对应指标。

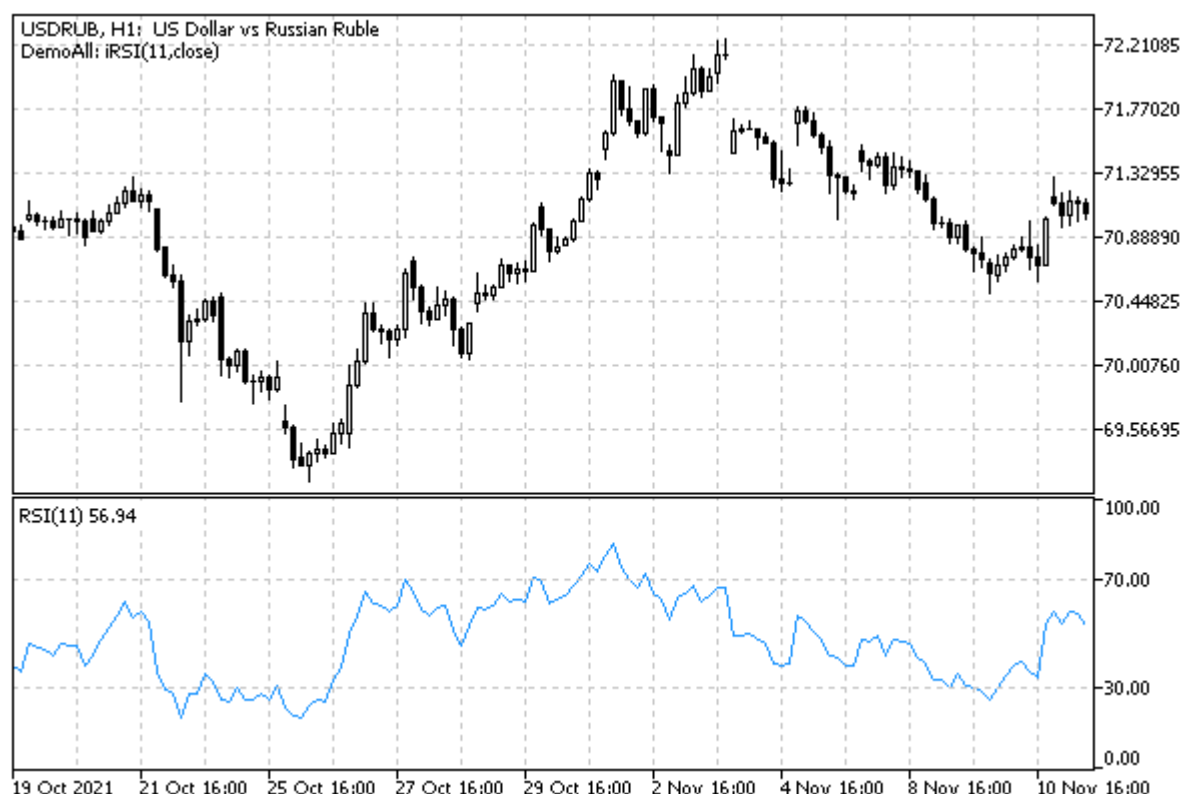
```

void OnDeinit(const int)
{
    Print(__FUNCSIG__, (StringLen(subTitle) > 0 ? " deleting " + subTitle : ""));
    if(StringLen(subTitle) > 0)
    {
        ChartIndicatorDelete(0, (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL) - 1,
            subTitle);
    }
}

```

这里假设图表中仅运行我们这一个指标，且仅存在一个实例。在更通用的场景中，应分析所有子窗口以确保正确删除，但这需要用到 [图表](#) 章节将介绍的更多函数，因此目前我们暂时采用简化版本。

现在，如果我们运行 *UseDemoAll* 并从列表中选择带星号标记的指标（即需要子窗口的指标），例如 RSI 指标，就能看到预期效果：RSI 在独立窗口中。



RSI在由 UseDemoAll指标创建的子窗口中

5.5.12 从存在偏移的图表中读取数据

我们的新指标 *UseDemoAll* 已基本就绪。我们只需再考虑一个细节。

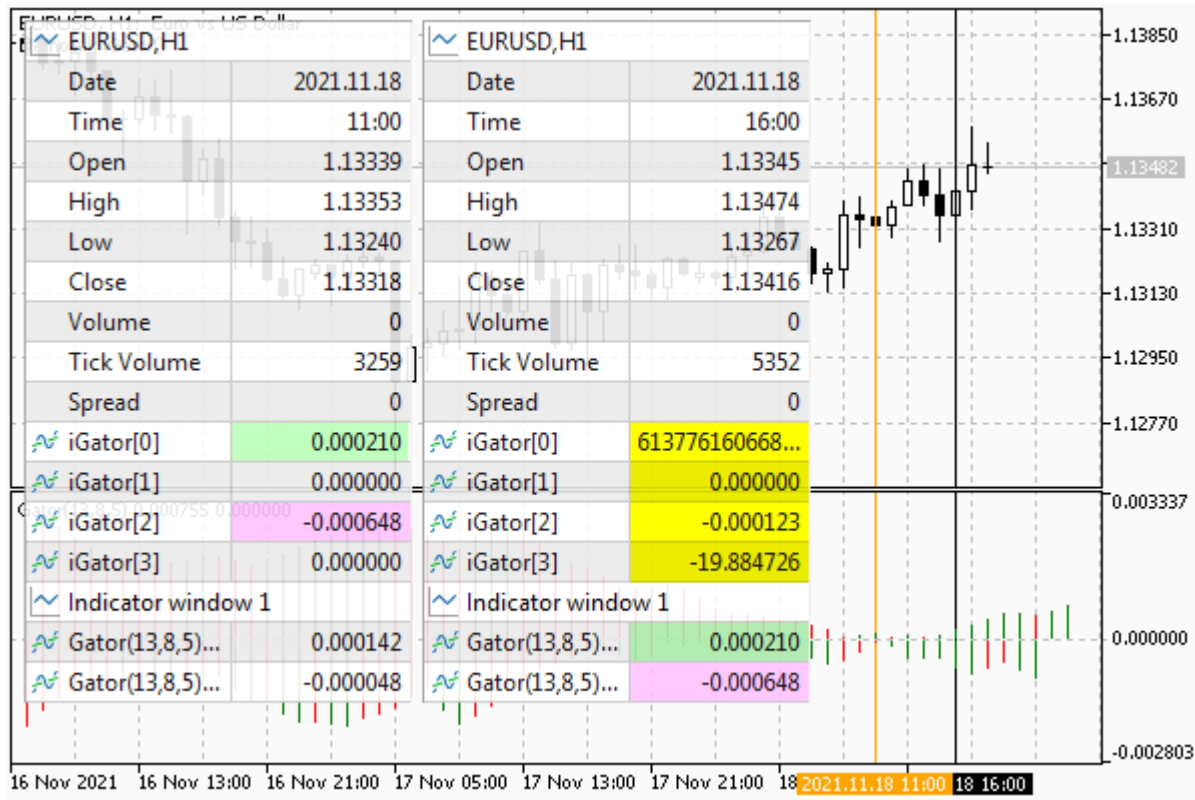
在从属指标中，部分图表可能通过 **PLOT_SHIFT** 特性设置了偏移。例如，使用正偏移，时间序列元素将移至未来方向，并显示在索引 0 柱线的右侧。奇怪的是，其索引值为负数。向右移动时，索引值逐渐减小：-1、-2、-3 等。这种寻址方式也会影响 *CopyBuffer* 函数。当我们使用第一种形式的 *CopyBuffer* 时，*offset* 参数设为 0 将指向时间序列中当前时间的元素。但如果时间序列本身向右偏移，我们将获得从第 N 个元素开始的数据，其中 N 为源指标的偏移值。同时，缓冲区中索引 N 右侧的元素将不会填充数据，而是保留原有“垃圾值”。

为演示该问题，让我们从无偏移的指标开始：*Awesome Oscillator* 正好符合这一要求。回想一下，*UseDemoAll* 会将所有值复制到其数组中，虽然由于价格刻度和指标读数不同，这些值在图表上不可见，但我们可以通过 *Data Window* 进行验证。无论将鼠标光标移动到图表的哪个位置，*Data Window* 中子窗口显示的指标值与 *UseDemoAll* 缓冲区中的值都将完全匹配。例如，在下图中，可以清晰看到，在 16:00 的小时柱线上，两个值均为 0.001797。



UseDemoAll缓冲区中的 AO 指标数据

现在，在 *UseDemoAll* 设置中，我们选择 *iGator (Gator Oscillator)* 指标。为简化操作，请清空 *Gator* 参数字段，使其使用默认参数构建。此时，可清晰观察到直方图（向前）偏移了 5 根柱线。



未修正未来偏移时 UseDemoAll缓冲区中的 Gator 指标数据

黑色垂直线标记了 16:00 的小时柱线。但 *Data Window* 中的 *Gator* 指标值与我们从同一指标读取到数组中的值存在差异。黄色 *UseDemoAll* 高亮显示包含垃圾值的缓冲区。

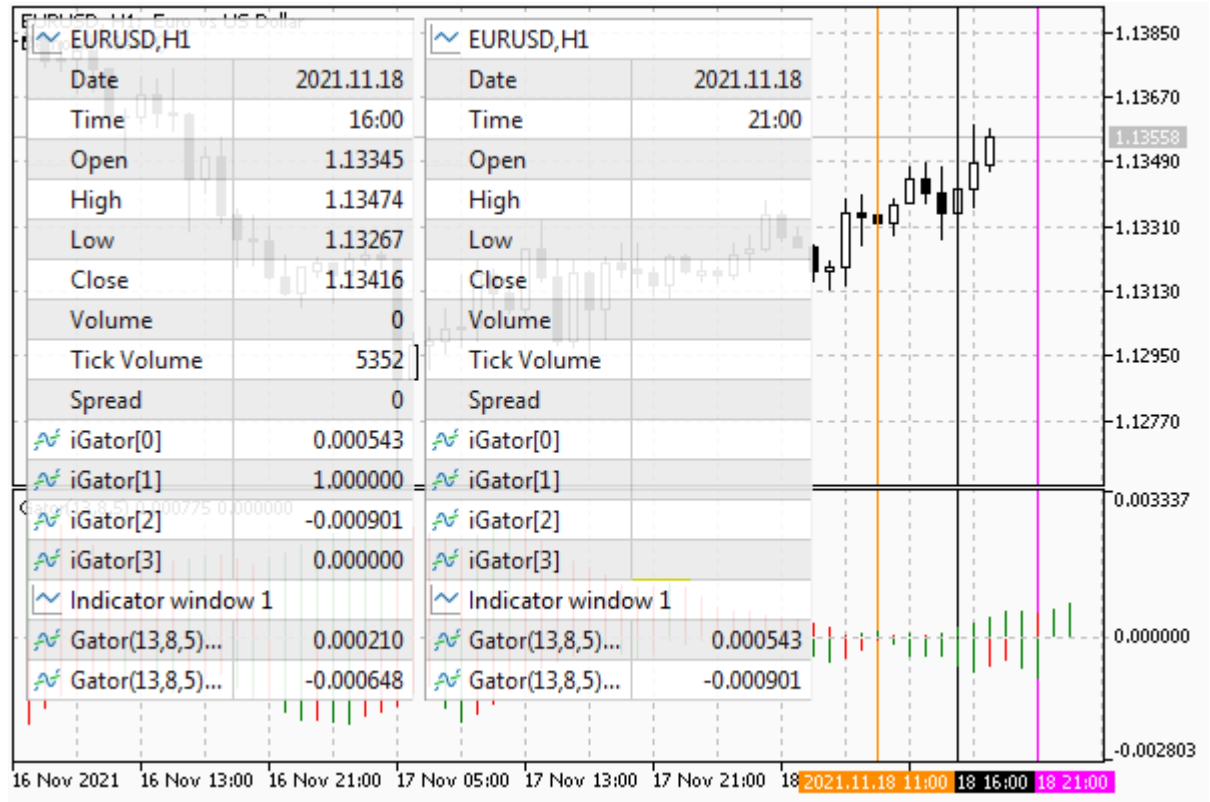
如果我们回溯移动 5 根柱线至 11:00 (橙色垂直线标注处) 查看数据, 会发现 *Gator* 指标在 16:00 输出的值出现在该位置。上下直方图的成对正确值分别用绿色和粉色高亮显示。

为解决该问题, 我们需要在 *UseDemoAll* 中添加一个输入变量, 让用户指定图表偏移量, 然后在调用 *CopyBuffer* 时对其进行修正。

```
input int IndicatorShift = 0; // Plot Shift
...
int OnCalculate(ON_CALCULATE_STD_SHORT_PARAM_LIST)
{
    ...
    for(int k = 0; k < m; ++k)
    {
        const int n = buffers[k].copy(Handle, k,
            -IndicatorShift, rates_total - prev_calculated + 1);
        ...
    }
}
```

遗憾的是, 在 MQL5 中无法直接获取第三方指标的 *PLOT_SHIFT* 特性值。

让我们验证, 引入偏移量 5 如何修复 *Gator* 指标 (使用默认设置) 的显示问题。



调整未来偏移后 *UseDemoAll* 缓冲区中的 *Gator* 指标数据

现在, *UseDemoAll* 在 16:00 柱线处的读数对应于 *Gator* 在虚拟未来 5 根柱线 (21:00 的淡紫色垂直线) 处的实际数据。

你可能会疑惑，为什么 *Gator* 指标窗口中只显示 2 个缓冲区，而我们的指标有 4 个。这是因为 *Gator* 的彩色直方图使用了一个额外的缓冲区进行颜色编码。但实际上只有两种颜色：红色和绿色，我们在数组中看到它们分别为 0 或 1。

5.5.13 删除指标实例：IndicatorRelease

如本章导言所述，终端为每个创建的指标维护一个引用计数器，只要至少有一个 MQL 程序或图表在使用该指标，终端就会保持运行状态。在 MQL 程序中，需要指标的标志就是有效的句柄。通常，我们在初始化时请求句柄，并在算法中持续使用它，直到程序结束。

当程序卸载时，所有创建的唯一句柄都会被自动释放，即它们的计数器减 1（如果计数器归零，相应指标也会从内存中卸载）。因此无需显式释放句柄。

但在程序运行过程中有时会出现子指标不再需要的情况。此时，无用的指标会继续消耗资源。因此，必须使用 *IndicatorRelease* 显式释放句柄。

bool IndicatorRelease(int handle)

该函数会删除指定指标句柄，并在没有其他对象使用该指标时将其卸载。卸载过程会有轻微延迟。

该函数返回操作状态标识：成功 (*true*) 或失败 (*false*)。

调用 *IndicatorRelease* 后，传入的句柄将立即失效，即使变量本身仍保留之前的值。如果尝试在其他指标函数（如 *CopyBuffer*）中使用此类句柄将失败，同时触发错误 4807 (ERR_INDICATOR_WRONG_HANDLE)。为避免混淆，建议在释放句柄后立即将对应变量赋值为 INVALID_HANDLE。

但如果程序后续请求一个新指标的句柄，该句柄很可能与之前释放的句柄值相同，但此时该句柄已关联到新指标的数据。

在策略测试程序中运行时，*IndicatorRelease* 函数不会执行。

为演示 *IndicatorRelease* 的应用，我们准备了 *UseDemoAllLoop.mq5* 的一个特殊版本，该版本会基于列表周期性循环重新创建辅助指标，该列表仅包含主窗口指标（为简化演示）。

```

IndicatorType MainLoop[] =
{
    iCustom_,
    iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_price,
    iAMA_period_fast_slow_shift_price,
    iBands_period_shift_deviation_price,
    iDEMA_period_shift_price,
    iEnvelopes_period_shift_method_price_deviation,
    iFractals_,
    iFrAMA_period_shift_price,
    iIchimoku_tenkan_kijun_senkou,
    iMA_period_shift_method_price,
    iSAR_step_maximum,
    iTEMA_period_shift_price,
    iVIDyA_momentum_smooth_shift_price,
};

const int N = ArraySize(MainLoop);
int Cursor = 0; // current position inside the MainLoop array

const string IndicatorCustom = "LifeCycle";

```

数组的第一个元素包含一个自定义指标作为例外，即来自各类 [程序启动和停止的功能](#) 章节的 **LifeCycle**。虽然该指标不显示任何线条，但它在这里很合适，因为当 *OnInit/OnDeinit* 处理程序被调用时，该指标会在日志中显示消息，这将使你能够跟踪该指标的生命周期。其他指标的生命周期与此类似。

在输入变量中，我们仅保留渲染设置。默认的 `DRAW_ARROW` 标签输出格式最适合展示不同类型的指标。

```

input ENUM_DRAW_TYPE DrawType = DRAW_ARROW; // Drawing Type
input int DrawLineWidth = 1; // Drawing Line Width

```

为了动态重建指标，我们在 *OnInit* 中启动了一个 5 秒的 [计时器](#)，并将之前的整个初始化过程（含下文所述的部分修改）移至 *OnTimer* 处理程序。

```

int OnInit()
{
    Comment("Wait 5 seconds to start looping through indicator set");
    EventSetTimer(5);
    return INIT_SUCCEEDED;
}

IndicatorType IndicatorSelector; // currently selected indicator type

void OnTimer()
{
    if(Handle != INVALID_HANDLE && ClearHandles)
    {
        IndicatorRelease(Handle);
        /*
        // descriptor is still 10, but is no longer valid
        // if we uncomment the fragment, we get the following error
        double data[1];
        const int n = CopyBuffer(Handle, 0, 0, 1, data);
        Print("Handle=", Handle, " CopyBuffer=", n, " Error=", _LastError);
        // Handle=10 CopyBuffer=-1 Error=4807 (ERR_INDICATOR_WRONG_HANDLE)
        */
    }
    IndicatorSelector = MainLoop[Cursor];
    Cursor = ++Cursor % N;

    // create a handle with default parameters
    // (because we pass an empty string in the third argument of the constructor)
    AutoIndicator indicator(IndicatorSelector,
        (IndicatorSelector == iCustom_ ? IndicatorCustom : ""), "");
    Handle = indicator.getHandle();
    if(Handle == INVALID_HANDLE)
    {
        Print(StringFormat("Can't create indicator: %s",
            _LastError ? E2S(_LastError) : "The name or number of parameters is incorrec
    }
    else
    {
        Print("Handle=", Handle);
    }

    buffers.empty(); // clear buffers because a new indicator will be displayed
    ChartSetSymbolPeriod(0, NULL, 0); // request a full redraw
    ...
    // further setup of diagrams - similar to the previous one
    ...
    Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
        "(default-params)");
}

```

主要区别在于，当前创建的指标类型 *IndicatorSelector* 现在不由用户设置，而是通过 *Cursor* 索引从 *MainLoop* 数组中按顺序选择。每次计时器触发时，该索引会循环递增，即到达数组末尾后自动跳回开头。

所有指标的参数行均为空。这是为了统一初始化流程。因此，每个指标都会使用其默认值创建。

在 *OnTimer* 处理程序开始时，我们调用 *IndicatorRelease* 释放前一个句柄。但我们提供了一个输入变量 *ClearHandles* 来禁用给定 *if* 分支，观察不清理句柄会出现的情况。

```
input bool ClearHandles = true;
```

默认情况下，*ClearHandles* 为 *true*，即指标会按预期被删除。

最后，还有两个额外设置：清空缓冲区和请求完整重绘图表。这两项操作都是必需的，因为我们替换了提供显示数据的从属指标。

OnCalculate 处理程序保持不变。

我们使用默认设置运行 *UseDemoAllLoop*。日志中将显示以下条目（仅显示开头部分）：

```
UseDemoAllLoop (EURUSD,H1) Initializing LifeCycle() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) Handle=10
LifeCycle (EURUSD,H1) Loader::Loader()
LifeCycle (EURUSD,H1) void OnInit() 0 DEINIT_REASON_PROGRAM
UseDemoAllLoop (EURUSD,H1) Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lips
UseDemoAllLoop (EURUSD,H1) iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_pric
UseDemoAllLoop (EURUSD,H1) Handle=10
LifeCycle (EURUSD,H1) void OnDeinit(const int) DEINIT_REASON_REMOVE
LifeCycle (EURUSD,H1) Loader::~~Loader()
UseDemoAllLoop (EURUSD,H1) Initializing iAMA_period_fast_slow_shift_price() EURUSD, P
UseDemoAllLoop (EURUSD,H1) iAMA_period_fast_slow_shift_price requires 5 parameters, 0
UseDemoAllLoop (EURUSD,H1) Handle=10
UseDemoAllLoop (EURUSD,H1) Initializing iBands_period_shift_deviation_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iBands_period_shift_deviation_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=10
...
```

需要注意的是，由于我们在创建新句柄前释放了旧句柄，因此每次都会得到相同的句柄“编号”(10)。

同样重要的是，*LifeCycle* 指标会在句柄释放后立即卸载（假设该指标未被单独添加到同一图表中，否则其引用计数器将不会归零）。

下图展示了我们的指标渲染 Alligator 数据时的状态。



Alligator演示步骤中的 UseDemoAllLoop

如果将 `ClearHandles` 参数设置为 *false*, 日志将呈现完全不同的情况: 此时, 句柄编号会不断递增, 这表明指标仍在终端持续运行, 造成资源被白白浪费。特别是, 不会收到来自 `LifeCycle` 的反初始化消息。

```

UseDemoAllLoop (EURUSD,H1) Initializing LifeCycle() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) Handle=10
LifeCycle (EURUSD,H1) Loader::Loader()
LifeCycle (EURUSD,H1) void OnInit() 0 DEINIT_REASON_PROGRAM
UseDemoAllLoop (EURUSD,H1) Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lips
UseDemoAllLoop (EURUSD,H1) iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_pric
UseDemoAllLoop (EURUSD,H1) Handle=11
UseDemoAllLoop (EURUSD,H1) Initializing iAMA_period_fast_slow_shift_price() EURUSD, P
UseDemoAllLoop (EURUSD,H1) iAMA_period_fast_slow_shift_price requires 5 parameters, 0
UseDemoAllLoop (EURUSD,H1) Handle=12
UseDemoAllLoop (EURUSD,H1) Initializing iBands_period_shift_deviation_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iBands_period_shift_deviation_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=13
UseDemoAllLoop (EURUSD,H1) Initializing iDEMA_period_shift_price() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) iDEMA_period_shift_price requires 3 parameters, 0 given
UseDemoAllLoop (EURUSD,H1) Handle=14
UseDemoAllLoop (EURUSD,H1) Initializing iEnvelopes_period_shift_method_price_deviation
UseDemoAllLoop (EURUSD,H1) iEnvelopes_period_shift_method_price_deviation requires 5
UseDemoAllLoop (EURUSD,H1) Handle=15
...
UseDemoAllLoop (EURUSD,H1) Initializing iVIDyA_momentum_smooth_shift_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iVIDyA_momentum_smooth_shift_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=22
UseDemoAllLoop (EURUSD,H1) Initializing LifeCycle() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) Handle=10
UseDemoAllLoop (EURUSD,H1) Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lips
UseDemoAllLoop (EURUSD,H1) iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_pric
UseDemoAllLoop (EURUSD,H1) Handle=11
UseDemoAllLoop (EURUSD,H1) Initializing iAMA_period_fast_slow_shift_price() EURUSD, P
UseDemoAllLoop (EURUSD,H1) iAMA_period_fast_slow_shift_price requires 5 parameters, 0
UseDemoAllLoop (EURUSD,H1) Handle=12
UseDemoAllLoop (EURUSD,H1) Initializing iBands_period_shift_deviation_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iBands_period_shift_deviation_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=13
UseDemoAllLoop (EURUSD,H1) Initializing iDEMA_period_shift_price() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) iDEMA_period_shift_price requires 3 parameters, 0 given
UseDemoAllLoop (EURUSD,H1) Handle=14
UseDemoAllLoop (EURUSD,H1) void OnDeinit(const int)
...

```

当循环遍历指标类型数组的索引到达最后一个元素并从头开始循环时，终端将开始向我们的代码返回已存在指标的句柄（相同数值：句柄 22 之后会再次出现 10）。

5.5.14 通过指标句柄获取指标设置

有时，MQL 程序需要获取运行中指标实例的参数。这些指标可以是图表上的第三方指标，也可以是从主程序传递到库或头文件的句柄。为此，MQL5 提供了 *IndicatorParameters* 函数。

```
int IndicatorParameters(int handle, ENUM_INDICATOR &type, MqlParam &params[])
```

该函数通过指定句柄返回指标输入参数数量及其类型和值。

成功时，该函数将填充传入的 *params* 数组，并将指标类型保存在 *type* 参数中。

如果出现错误，该函数将返回 -1。

作为使用该函数的示例，我们将改进 [删除指标实例](#) 章节中展示的 *UseDemoAllLoop.mq5* 指标。新版本命名为 *UseDemoAllParams.mq5*。

你应该还记得，我们曾在循环中依次创建列表中的内置指标，并将参数列表留空，这导致指标使用某些未知的默认值。为此，我们曾在图表注释中显示一个通用原型：仅包含指标名称，而没有具体参数值。

```
// UseDemoAllLoop.mq5
void OnTimer()
{
    ...
    Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
            "(default-params)");
    ...
}
```

现在，我们可以基于指标句柄获取其参数并向用户显示。

```
// UseDemoAllParams.mq5
void OnTimer()
{
    ...
    // read the parameters applied by the indicator by default
    ENUM_INDICATOR itype;
    MqlParam defParams[];
    const int p = IndicatorParameters(Handle, itype, defParams);
    ArrayPrint(defParams);
    Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
            "(" + MqlParamStringer::stringify(defParams) + ")");
    ...
}
```

将 *MqlParam* 数组转换为字符串的功能通过 *MqlParamStringer* 特殊类实现（请参阅文件 *MqlParamStringer.mqh*）。

```

class MqlParamStringer
{
public:
    static string stringify(const MqlParam &param)
    {
        switch(param.type)
        {
            case TYPE_BOOL:
            case TYPE_CHAR:
            case TYPE_UCHAR:
            case TYPE_SHORT:
            case TYPE_USHORT:
            case TYPE_DATETIME:
            case TYPE_COLOR:
            case TYPE_INT:
            case TYPE_UINT:
            case TYPE_LONG:
            case TYPE_ULONG:
                return IntegerToString(param.integer_value);
            case TYPE_FLOAT:
            case TYPE_DOUBLE:
                return (string)(float)param.double_value;
            case TYPE_STRING:
                return param.string_value;
        }
        return NULL;
    }

    static string stringify(const MqlParam &params[])
    {
        string result = "";
        const int p = ArraySize(params);
        for(int i = 0; i < p; ++i)
        {
            result += stringify(params[i]) + (i < p - 1 ? "," : "");
        }
        return result;
    }
};

```

编译并运行新指标后，你会发现当前渲染指标的特定参数列表显示在图表左上角。

对于列表中的单个自定义指标 (*LifeCycle*)，第一个参数将包含指标的路径和文件名。第二个参数在源代码中被描述为整数类型。但第三个参数有点不同，它隐式定义了所有使用简化版 *OnCalculate* 处理程序的指标所共有的“Apply to”特性。在此处，指标默认应用于 PRICE_CLOSE (值 1)。


```

Initializing LifeCycle() EURUSD, PERIOD_H1
Handle=10
[type] [integer_value] [double_value] [string_value]
[0] 14 0 0.00000 "Indicators\MQL5\Book\p5\LifeCycle.ex5"
[1] 7 0 0.00000 null
[2] 7 1 0.00000 null
Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_price() EURUSD, PE
iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_price requires 8 parameters, 0
Handle=10
[type] [integer_value] [double_value] [string_value]
[0] 7 13 0.00000 null
[1] 7 8 0.00000 null
[2] 7 8 0.00000 null
[3] 7 5 0.00000 null
[4] 7 5 0.00000 null
[5] 7 3 0.00000 null
[6] 7 2 0.00000 null
[7] 7 5 0.00000 null
Initializing iAMA_period_fast_slow_shift_price() EURUSD, PERIOD_H1
iAMA_period_fast_slow_shift_price requires 5 parameters, 0 given
Handle=10
[type] [integer_value] [double_value] [string_value]
[0] 7 9 0.00000 null
[1] 7 2 0.00000 null
[2] 7 30 0.00000 null
[3] 7 0 0.00000 null
[4] 7 1 0.00000 null

```

根据日志记录，内置指标的设置也与默认值相符。

5.5.15 定义指标的数据源

在 MQL 程序的 [内置变量](#) 中，有一个专门用于指标的变量。它是 *int* 类型的 `_AppliedTo` 变量，允许读取指标设置对话框中的 **Apply to** 特性。此外，如果指标是通过调用 `iCustom` 函数创建的，且该函数传入了第三方指标句柄，则 `_AppliedTo` 变量将包含该句柄。

下表介绍了 `_AppliedTo` 变量的可能取值。

值	计算数据说明
0	指标使用完整形式的 <i>OnCalculate</i> , 计算数据不由单一数据数组设置
1	收盘价
2	开盘价
3	最高价
4	最低价
5	平均价 = (最高价 + 最低价)/2
6	典型价 = (最高价 + 最低价 + 收盘价)/3
7	加权价 = (开盘价 + 最高价 + 最低价 + 收盘价)/4
8	先于本指标在图表上启动的指标数据
9	图表上最早启动的指标数据
10+	<i>_AppliedTo</i> ;所含句柄对应的指标数据;创建指标时,该句柄作为最后一个参数传递给 <i>iCustom</i> 函数。

为便于分析这些数值,本书附带了包含枚举的头文件 *AppliedTo.mqh*。

5.6 使用计时器

对于许多应用任务而言,能够按计划以指定时间间隔执行操作非常重要。在 MQL5 中,此功能由计时器提供,这是一个系统时间计数器,可配置为向 MQL 程序定期发送通知。

在 MQL5 API 中,有多个用于设置或取消计时器通知的函数: *EventSetTimer*、*EventSetMillisecondTimer*、*EventKillTimer*。这些通知本身以特殊类型的事件形式进入程序:在源代码中为它们保留了 *OnTimer* 处理程序。本章将讨论这组函数。

需要提醒的是,在 MQL5 中,只有在图表上运行的交互式程序(即指标和 EA 交易)才能接收事件。脚本和服务不支持任何事件,包括来自计时器的事件。

但在时间处理函数一章中,我们已经涉及了相关主题:

- ⌚ 获取当前本地或服务器时钟的时间戳 (*TimeLocal/TimeCurrent*)
- ⌚ 使用以下函数使程序按指定时间段暂停: *Sleep*
- ⌚ 获取计算机系统时间计数器的状态,该计数器可从操作系统启动时开始计数 (*GetTickCount*) 或从 MQL 程序启动时开始计数 (*GetMicrosecondCount*)

这些选项对所有类型的 MQL 程序完全开放。

在前面的章节中,我们已经多次使用了计时器函数,尽管直到现在才会对它们进行正式说明。由于计时器事件仅在指标或 EA 交易中可用,因此在讲解程序本身之前很难对其进行研究。在我们掌握了指标创建之后,计时器的主题将成为一个顺理成章的延续。

基本上,我们使用计时器来等待时间序列的构建。这类示例可以在以下章节中找到: 等待数据、多货币和多时间范围指标、支持多个交易品种和时间范围、使用内置指标。

此外，在 [删除指标实例](#) 一节的指标“动画”演示中，我们每隔 5 秒，对从属指标类型进行一次定时检测。

5.6.1 打开和关闭计时器：EventSetTimer/EventKillTimer

MQL5 允许启用或禁用标准计时器以执行任何预定操作。为此提供了两个函数：*EventSetTimer*和*EventKillTimer*。

bool EventSetTimer(int seconds)

该函数向客户端终端表明，对于此 EA 交易或指标，需要根据指定频率通过计时器生成事件，频率以秒为单位设置（参数为 *seconds*）。

该函数返回操作状态标识：成功 (*true*)或失败 (*false*)。错误代码可从 *_LastError*中获取。

为了处理计时器事件，EA 交易或指标的代码中必须包含 *OnTimer* 函数。第一次计时器事件不会在调用 *EventSetTimer*后立即发生，而是在经过 *seconds* 秒后发生。

对于每个调用*EventSetTimer*函数的 EA 交易或指标，它会创建自己的专用计时器。程序将仅接收来自该计时器的事件。不同程序中的计时器独立工作。

放置在图表上的每个交互式 MQL 程序都有一个独立的事件队列，收到的相关事件会被添加到该队列中。如果队列中已经有一个 *OnTimer*事件，或者正在处理该事件，则新的 *OnTimer* 事件不会被加入队列。

如果不再需要计时器，应使用 *EventKillTimer*函数将其禁用。

void EventKillTimer(void)

该函数用于停止此前通过 *EventSetTimer* (或即将讨论的 *EventSetMillisecondTimer*函数)启用的计时器。该函数也可以从*OnTimer*处理程序中调用。因此具体而言，可以执行一个延迟的单次操作。

在指标中调用 *EventKillTimer*函数不会清空队列，因此之后你仍可能收到最后一个残留的 *OnTimer* 事件。

当 MQL 程序终止时，如果计时器已被创建但未通过 *EventKillTimer*函数禁用，则该计时器将被强制销毁。

每个程序只能设置一个计时器。因此，如果你想以不同时间间隔调用算法的不同部分，应启用一个周期为所需周期最小公倍数的计时器 (在极限情况下，最小周期为 1 秒)，并在*OnTimer*处理程序中独立跟踪较长的周期。我们将在下一节中探讨该方法的示例。

MQL5 还允许创建周期小于 1 秒的计时器：为此提供了一个函数 *EventSetMillisecondTimer*。

5.6.2 计时器事件：OnTimer

*OnTimer*事件是 MQL5 程序支持的标准事件之一 (请参阅 [事件处理函数概述](#) 章节)。要在程序代码中接收计时器事件，应声明如下原型的函数。

void OnTimer(void)

对于通过 *EventSetTimer* 或 *EventSetMillisecondTimer* 函数激活计时器的 EA 交易或指标，客户端终端会定期生成 *OnTimer*事件 (请参阅下一节)。

注意！在通过其他程序调用 *iCustom*或 *IndicatorCreate* 创建的从属指标中，计时器无效，且不会生成 *OnTimer* 事件。这是 MetaTrader 5 架构方面的局限性所致。

需要理解的是，启用计时器和 *OnTimer* 处理程序不会使 MQL 程序成为多线程程序。每个 MQL 程序最多分配一个线程（同一交易品种上的指标甚至可以与其他指标共享一个线程），因此 *OnTimer* 和其他处理程序的调用始终按照事件队列的顺序依次进行。如果某个处理程序（包括 *OnTimer*）开始执行冗长的计算，这将暂停所有其他事件和程序代码段的执行。

如果需要组织并行数据处理，则应同时运行多个 MQL 程序（例如在不同 **图表** 或 **图表对象** 上运行同一程序的多个实例），并使用自定义协议（例如使用 **自定义事件**）在它们之间交换命令和数据。

作为示例，我们将创建能够在一个程序中组织多个逻辑计时器的类。所有逻辑计时器的周期将被设置为基本周期的整数倍，即单一硬件计时器向标准处理程序 *OnTimer* 提供事件的周期。在该处理程序中，我们必须调用新的 *MultiTimer* 类的某种方法，该类将管理所有逻辑计时器。

```
void OnTimer()
{
    // call the MultiTimer method to check and call dependent timers when needed
    MultiTimer::onTimer();
}
```

MultiTimer 类和各个计时器的相关类将组合在一个文件 *MultiTimer.mqh* 中。

工作计时器的基类将是 *TimerNotification*。严格来说，这可以是一个接口，但将通用实现的部分细节输出到其中会很方便：特别是存储计数器 *chronometer* 的读数，我们将使用它来确保计时器按照主计时器相对周期的特定倍数触发，同时采用 *isTimeCome* 方法用于校验计时器应触发时机。这是 *TimerNotification* 为什么是抽象类的原因。它缺少两个虚方法的实现：*notify* - 计时器触发时执行的操作，*getInterval* - 用于获取设定单个计时器周期相对于主计时器周期倍率的乘数。

```

class TimerNotification
{
protected:
    int chronometer; // counter of timer checks (isTimeCome calls)
public:
    TimerNotification(): chronometer(0)
    {
    }

    // timer work event
    // pure virtual method, it is required to be described in the heirs
    virtual void notify() = 0;
    // returns the period of the timer (it can be changed on the go)
    // pure virtual method, it is required to be described in the heirs
    virtual int getInterval() = 0;
    // check if it's time for the timer to fire, and if so, call notify
    virtual bool isTimeCome()
    {
        if(chronometer >= getInterval() - 1)
        {
            chronometer = 0; // reset the counter
            notify();         // notify application code
            return true;
        }

        ++chronometer;
        return false;
    }
};

```

所有逻辑都包含在 *isTimeCome* 方法中。每次调用该方法时，*chronometer* 计数器都会递增，如果根据 *getInterval* 方法达到最后一次迭代，则调用 *notify* 方法通知应用程序代码。

例如，如果主计时器以 1 秒为周期启动 *EventSetTimer(1)*，则当子类对象 *TimerNotification*，从 *getInterval* 返回 5 时，其 *notify* 方法将每 5 秒被调用一次。

如之前所述，这些计时器对象将由 *MultiTimer* 管理器对象管理。我们只需要一个这样的对象。因此，其构造函数被声明为受保护，且在类内部静态创建单个实例。

```

class MultiTimer
{
protected:
    static MultiTimer _mainTimer;

    MultiTimer()
    {
    }
    ...

```

在该类内部，我们设计了 *TimerNotification* 对象数组的存储（具体填充逻辑将在后续段落详述）。获得该数组后，我们可以轻松编写遍历所有逻辑计时器的 *checkTimers* 方法。为便于外部访问，该方法通过公共静态方法 *onTimer* 进行复制，该方法正是我们先前在全局 *OnTimer* 处理程序中看到的方法。由于唯一的管理器实例是静态创建的，因此我们可以从静态方法访问它。

```

...
TimerNotification *subscribers[];

void checkTimers()
{
    int n = ArraySize(subscribers);
    for(int i = 0; i < n; ++i)
    {
        if(CheckPointer(subscribers[i]) != POINTER_INVALID)
        {
            subscribers[i].isTimeCome();
        }
    }
}

public:
    static void onTimer()
    {
        _mainTimer.checkTimers();
    }
    ...

```

*TimerNotification*对象通过 *bind* 方法被添加到 *subscribers* 数组中。

```

void bind(TimerNotification &tn)
{
    int i, n = ArraySize(subscribers);
    for(i = 0; i < n; ++i)
    {
        if(subscribers[i] == &tn) return; // there is already such an object
        if(subscribers[i] == NULL) break; // found an empty slot
    }
    if(i == n)
    {
        ArrayResize(subscribers, n + 1);
    }
    else
    {
        n = i;
    }
    subscribers[n] = &tn;
}

```

该方法可防止对象被重复添加，并且在可能的情况下，如果数组中存在空元素，会将指针放置在该空元素中，从而避免数组扩容。当使用 *unbind*方法移除任意 *TimerNotification* 对象时，数组中可能会出现空元素（计时器可能仅偶尔使用）。

```

void unbind(TimerNotification &tn)
{
    const int n = ArraySize(subscribers);
    for(int i = 0; i < n; ++i)
    {
        if(subscribers[i] == &tn)
        {
            subscribers[i] = NULL;
            return;
        }
    }
}

```

请注意，管理器不会获取计时器对象的所有权，也不会尝试调用 *delete*。如果你要在管理器中注册动态分配的计时器对象，可以在置零前的 *if* 内添加以下代码：

```

if(CheckPointer(subscribers[i]) == POINTER_DYNAMIC) delete subscribers[i]

```

现在需要思考如何便捷地组织 *bind/unbind* 调用，以免让应用代码被这些工具性操作拖累。如果“手动”处理，很容易忘记在某处创建计时器，或者相反，忘记删除计时器。

我们来开发一个从 *TimerNotification* 派生的 *SingleTimer* 类，分别在构造函数和析构函数中实现 *bind* 和 *unbind* 调用。此外，我们在该类中定义了 *multiplier* 变量来存储计时器周期。

```

class SingleTimer: public TimerNotification
{
protected:
    int multiplier;
    MultiTimer *owner;

public:
    // creating a timer with the specified base period multiplier, optionally pause
    // automatically register the object in the manager
    SingleTimer(const int m, const bool paused = false): multiplier(m)
    {
        owner = &MultiTimer::_mainTimer;
        if(!paused) owner->bind(this);
    }

    // automatically disconnect the object from the manager
    ~SingleTimer()
    {
        owner->unbind(this);
    }

    // return timer period
    virtual int getInterval() override
    {
        return multiplier;
    }

    // pause this timer
    virtual void stop()
    {
        owner->unbind(this);
    }

    // resume this timer
    virtual void start()
    {
        owner->bind(this);
    }
};

```

构造函数的第二个参数 (*paused*) 允许创建对象，但不立即启动计时器。这种延迟启动的计时器后续可通过 *start* 方法激活。

某些对象订阅其他对象事件的机制是 OOP 中常见的设计模式之一，被称为“发布者/订阅者”模式。

要注意的是，该类也是抽象类，因为它未实现 *notify* 方法。基于 *SingleTimer*，我们来描述具有额外功能的计时器类。

首先从 *CountableTimer* 类开始。它允许指定计时器触发的次数，达到次数后会自动停止。借助该类，安排单次延迟操作会特别简单。*CountableTimer* 构造函数包含用于设置计时器周期、暂停标志和重试次数的参数。默认情况下，重复次数不受限制，因此该类将成为大多数应用计时器的基础。


```

class CountableTimer: public MultiTimer::SingleTimer
{
protected:
    const uint repeat;
    uint count;

public:
    CountableTimer(const int m, const uint r = UINT_MAX, const bool paused = false):
        SingleTimer(m, paused), repeat(r), count(0) { }

    virtual bool isTimeCome() override
    {
        if(count >= repeat && repeat != UINT_MAX)
        {
            stop();
            return false;
        }
        // delegate the time check to the parent class,
        // increment our counter only if the timer fired (returned true)
        return SingleTimer::isTimeCome() && (bool)++count;
    }
    // reset our counter on stop
    virtual void stop() override
    {
        SingleTimer::stop();
        count = 0;
    }

    uint getCount() const
    {
        return count;
    }

    uint getRepeat() const
    {
        return repeat;
    }
};

```

要使用 *CountableTimer*, 我们需要在程序中按以下方式描述派生类:

```
// MultipleTimers.mq5
class MyCountableTimer: public CountableTimer
{
public:
    MyCountableTimer(const int s, const uint r = UINT_MAX):
        CountableTimer(s, r) { }

    virtual void notify() override
    {
        Print(__FUNCSIG__, multiplier, " ", count);
    }
};
```

在 *notify* 方法的实现中，我们仅需记录计时器周期和触发次数。顺便提一下，这是 *MultipleTimers.mq5* 指标的一个片段，我们将把它作为实际示例。

我们将从 *SingleTimer* 派生的第二个类称为 *FunctionalTimer*。其目的是为偏好函数式编程风格、不希望编写派生类的开发者提供一个简单的计时器实现。*FunctionalTimer* 类的构造函数除了接收周期参数外，还将接收一个指向特殊类型函数 *TimerHandler* 的指针。

```
// MultiTimer.mqh
typedef bool (*TimerHandler)(void);

class FunctionalTimer: public MultiTimer::SingleTimer
{
    TimerHandler func;
public:
    FunctionalTimer(const int m, TimerHandler f):
        SingleTimer(m), func(f) { }

    virtual void notify() override
    {
        if(func != NULL)
        {
            if(!func())
            {
                stop();
            }
        }
    }
};
```

在 *notify* 方法的实现中，对象通过指针调用函数。借助这样的类，我们可以定义一个宏，若将这个宏放在由花括号括起来的语句块之前，就会将该语句块“变成”计时器函数的主体。

```
// MultiTimer.mqh
#define OnTimerCustom(P) OnTimer##P(); \
FunctionalTimer ft##P(P, OnTimer##P); \
bool OnTimer##P()
```

这样在应用代码中就可以这样编写：

```
// MultipleTimers.mq5
bool OnTimerCustom(3)
{
    Print(__FUNCSIG__);
    return true;          // continue the timer
}
```

这种结构声明了一个周期为 3 的计时器，圆括号内是一组指令（此处只是日志输出）。如果函数返回 *false*，该计时器将停止运行。

我们来深入分析 *MultipleTimers.mq5* 指标。由于该指标不涉及可视化，我们将图表数量设为零。

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots 0
```

要使用逻辑计时器类，我们需包含头文件 *MultiTimer.mqh*，并添加一个用于设置基础（全局）计时器周期的输入变量。

```
#include <MQL5Book/MultiTimer.mqh>

input int BaseTimerPeriod = 1;
```

基础计时器在 *OnInit* 中启动。

```
void OnInit()
{
    Print(__FUNCSIG__, " ", BaseTimerPeriod, " Seconds");
    EventSetTimer(BaseTimerPeriod);
}
```

需记住，所有逻辑计时器的运行均通过拦截全局 *OnTimer* 事件来保障。

```
void OnTimer()
{
    MultiTimer::onTimer();
}
```

除了前文提到的计时器应用类 *MyCountableTimer*，我们再描述另一个暂停计时器类 *MySuspendedTimer*。

```

class MySuspendedTimer: public CountableTimer
{
public:
    MySuspendedTimer(const int s, const uint r = UINT_MAX):
        CountableTimer(s, r, true) { }
    virtual void notify() override
    {
        Print(__FUNCSIG__, multiplier, " ", count);
        if(count == repeat - 1) // execute last time
        {
            Print("Forcing all timers to stop");
            EventKillTimer();
        }
    }
};

```

稍后我们将演示其启动方式。此处还需重点指出，在达到指定操作次数后，该计时器将通过调用 *EventKillTimer* 终止所有计时器。

下面我们来展示（在全局上下文中）如何定义这两类的不同计时器对象。

```

MySuspendedTimer st(1, 5);
MyCountableTimer t1(2);
MyCountableTimer t2(4);

```

MySuspendedTimer 类的 *st* 计时器周期为 1 ($1 * BaseTimerPeriod$)，并应在执行 5 次操作后停止。

MyCountableTimer 类的 *t1* 和 *t2* 计时器周期分别为 2 ($2 * BaseTimerPeriod$) 和 4 ($4 * BaseTimerPeriod$)。在默认值 *BaseTimerPeriod* = 1 的情况下，所有周期均以秒为单位。这两个计时器会在程序启动后立即开始运行。

我们还将以函数式样式创建两个计时器。

```

bool OnTimerCustom(5)
{
    Print(__FUNCSIG__);
    st.start();           // start delayed timer
    return false;         // and stop this timer object
}

bool OnTimerCustom(3)
{
    Print(__FUNCSIG__);
    return true;          // this timer keeps running
}

```

请注意，*OnTimerCustom5* 仅有一项任务：在程序启动 5 个周期后，它需要启动延迟计时器 *st* 并终止自身执行。考虑到延迟计时器应在 5 个周期后停用所有计时器，在默认设置下，程序将运行 10 秒。

在此期间，*OnTimerCustom3* 计时器应触发 3 次。

因此，我们有 5 个不同周期的计时器：1 秒、2 秒、3 秒、4 秒和 5 秒。

我们将分析一个输出到日志的示例（右侧以示意图形式显示时间戳）。

```

// time
17:08:45.174 void OnInit() 1 Seconds |
17:08:47.202 void MyCountableTimer::notify()2 0 |
17:08:48.216 bool OnTimer3() |
17:08:49.230 void MyCountableTimer::notify()2 1 |
17:08:49.230 void MyCountableTimer::notify()4 0 |
17:08:50.244 bool OnTimer5() |
17:08:51.258 void MyCountableTimer::notify()2 2 |
17:08:51.258 bool OnTimer3() |
17:08:51.258 void MySuspendedTimer::notify()1 0 |
17:08:52.272 void MySuspendedTimer::notify()1 1 |
17:08:53.286 void MyCountableTimer::notify()2 3 |
17:08:53.286 void MyCountableTimer::notify()4 1 |
17:08:53.286 void MySuspendedTimer::notify()1 2 |
17:08:54.300 bool OnTimer3() |
17:08:54.300 void MySuspendedTimer::notify()1 3 |
17:08:55.314 void MyCountableTimer::notify()2 4 |
17:08:55.314 void MySuspendedTimer::notify()1 4 |
17:08:55.314 Forcing all timers to stop |

```

两秒计时器的第一条消息如预期一样在启动后约 2 秒到达（我们说“约”是因为硬件计时器存在精度限制，此外，计算机的其他负载也会影响执行）。一秒后，三秒计时器首次触发。两秒计时器的第二次触发与四秒计时器的第一次输出同时发生。五秒计时器执行一次后，一秒计时器的消息开始定期出现在日志中（其计数器从 0 增大至 4）。在最后一次迭代中，它将停止所有计时器运行。

5.6.3 高精度计时器：EventSetMillisecondTimer

如果程序需要计时器的触发频率高于每秒一次，则应使用 *EventSetMillisecondTimer* 函数而非 *EventSetTimer* 函数。

不同单位的计时器不能同时启动：必须使用其中一个函数，而不是同时使用多个函数。实际运行的计时器类型取决于最后调用的函数。属于 **标准计时器** 固有的所有功能对高精度时器同样有效。

bool EventSetMillisecondTimer(int milliseconds)

该函数向客户端终端表明，需要为该 EA 交易或指标生成频率小于 1 秒的计时器事件。周期以毫秒为单位设置（参数为 *milliseconds*）。

该函数返回操作状态标识：成功 (*true*) 或失败 (*false*)。

在策略测试程序中运行时，需记住，计时器周期越短，测试耗时越长，因为计时器事件处理程序的调用次数会增加。

在正常运行期间，由于硬件限制，计时器事件的生成频率不超过每 10-16 毫秒一次。

为了演示如何使用毫秒级计时器，我们来扩展一下指标示例 *MultipleTimers.mq5*。由于全局计时器的激活由应用程序负责，我们可以在不修改逻辑计时器类的情况下轻松更改计时器类型。唯一的区别在于，它们的乘数将应用于我们在 *EventSetMillisecondTimer* 函数中指定的以毫秒为单位的基础周期。

为了选择计时器类型，我们将定义一个枚举并添加一个新的输入变量。

```
enum TIMER_TYPE
{
    Seconds,
    Milliseconds
};

input TIMER_TYPE TimerType = Seconds;
```

默认情况下，我们使用秒级计时器。在 *OnInit* 函数中，启动所需类型的计时器。

```
void OnInit()
{
    Print(__FUNCSIG__, " ", BaseTimerPeriod, " ", EnumToString(TimerType));
    if(TimerType == Seconds)
    {
        EventSetTimer(BaseTimerPeriod);
    }
    else
    {
        EventSetMillisecondTimer(BaseTimerPeriod);
    }
}
```

我们来观察选择毫秒级计时器时日志中将显示的内容。

```

// time ms
17:27:54.483 void OnInit() 1 Milliseconds |
17:27:54.514 void MyCountableTimer::notify()2 0 | +31
17:27:54.545 bool OnTimer3() | +31
17:27:54.561 void MyCountableTimer::notify()2 1 | +16
17:27:54.561 void MyCountableTimer::notify()4 0 |
17:27:54.577 bool OnTimer5() | +16
17:27:54.608 void MyCountableTimer::notify()2 2 | +31
17:27:54.608 bool OnTimer3() |
17:27:54.608 void MySuspendedTimer::notify()1 0 |
17:27:54.623 void MySuspendedTimer::notify()1 1 | +15
17:27:54.655 void MyCountableTimer::notify()2 3 | +32
17:27:54.655 void MyCountableTimer::notify()4 1 |
17:27:54.655 void MySuspendedTimer::notify()1 2 |
17:27:54.670 bool OnTimer3() | +15
17:27:54.670 void MySuspendedTimer::notify()1 3 |
17:27:54.686 void MyCountableTimer::notify()2 4 | +16
17:27:54.686 void MySuspendedTimer::notify()1 4 |
17:27:54.686 Forcing all timers to stop |
```

事件生成的顺序与我们在秒级计时器中看到的完全相同，但执行速度显著加快，几乎是瞬间完成。

由于系统计时器的精度限制在几十毫秒，实际事件间隔会远超无法达到的 1 毫秒。此外，每个“时间步长”存在波动。因此，即使使用毫秒级计时器，也建议不要设置小于几十毫秒的周期。

5.7 使用图表

脚本、指标、EA 交易等大多数 MQL 程序均是在图表上执行的。只有服务是在后台运行，且不与任何时间表绑定。系统提供了一组丰富的函数，可用于获取和更改图表特性、分析图表列表以及搜索其他正在运行的程序。

由于图表是指标运行的自然环境，因此我们在之前的指标章节中已经接触过部分此类功能。在本章中，我们将有针对性地研究所有这些函数。

在处理图表时，我们将使用“窗口”这一概念。窗口是专门用于显示价格图表和/或指标图表的区域。顶部窗口（通常也是最大的窗口）包含价格图表，其编号为 0 且始终存在。在放置指标时添加到下方的所有附加窗口的编号均从 1 开始依次递增（编号顺序为自上而下）。每个子窗口仅在包含至少一个指标时才会存在。

由于用户可以删除任意子窗口中的所有指标（包括非最下方窗口），剩余子窗口的索引可能随之减小。

关于图表的事件模型（涉及接收和处理图表事件通知以及生成自定义事件）将在[单独的章节](#)中讨论。

除了此处讨论的“窗口中的图表”外，MetaTrader 5 还允许创建“对象中的图表”。我们将在下一章讨论[图形对象](#)。

5.7.1 用于获取当前图表基本特性的函数

在本书的许多示例中，我们已经需要使用[预定义变量](#)，这些变量包含图表及其交易品种的主要特性。MQL 程序还可以访问用于返回其中某些变量值的函数。无论使用变量还是函数效果等同，因此你可以使用自己偏好的源代码风格。

每个图表均以其交易品种和时间范围为特征。它们可以分别通过 *Symbol* 和 *Period* 函数来获取。此外，MQL5 提供了对两个最常用交易品种特性的简化访问方式：价格点值 (*Point*) 和价格中小数点后的有效位数 (*Digits*)。

string Symbol()

Symbol 函数返回当前图表的交易品种名称，即系统变量 *_Symbol* 的值。要获取任意图表的交易品种，可以使用 *ChartSymbol* 函数，该函数基于图表标识符运行。我们稍后将讨论获取图表标识符的方法。

ENUM_TIMEFRAMES Period()

Period 函数返回当前图表的时间范围值 (*ENUM_TIMEFRAMES*)，该值对应 *_Period* 变量。要获取任意图表的时间范围，可使用 *ChartPeriod* 函数，该函数同样需要图表标识符作为参数。

double Point()

Point 函数返回当前金融工具的点值，以报价货币计算，该值与 *_Point* 变量的值相同。

int Digits()

Digits() 函数返回小数点后的小数位数，该值决定了当前图表交易品种价格的测量精度，与 *_Digits* 变量等效。

当前工具的其他特性允许获取 *SymbolInfo-functions*，这些函数在更普遍的情况下可对所有金融工具进行分析。

以下是 *ChartMainProperties.mq5* 脚本的简单示例，用于记录本节描述的各项特性。

```
void OnStart()
{
    PRTF(_Symbol);
    PRTF(Symbol());
    PRTF(_Period);
    PRTF(Period());
    PRTF(_Point);
    PRTF(Point());
    PRTF(_Digits);
    PRTF(Digits());
    PRTF(DoubleToString(_Point, _Digits));
    PRTF(EnumToString(_Period));
}
```

对于 EURUSD,H1 图表，我们将得到以下日志条目。

```
_Symbol=EURUSD / ok
Symbol()=EURUSD / ok
_Period=16385 / ok
Period()=16385 / ok
_Point=1e-05 / ok
Point()=1e-05 / ok
_Digits=5 / ok
Digits()=5 / ok
DoubleToString(_Point,_Digits)=0.00001 / ok
EnumToString(_Period)=PERIOD_H1 / ok
```

5.7.2 图表标识

在 MetaTrader 5 中，每个图表均在独立窗口中运行，并拥有唯一标识符。对于熟悉 Windows 操作原理的程序员，需要澄清的是，该标识符并非系统窗口句柄（但 MQL5 API 可通过 [CHART_WINDOW_HANDLE](#) 特性获取窗口句柄）。我们知道，除了显示报价的图表主工作区外，还有一些额外区域（子窗口）用于显示具有 *indicator_separate_window* 特性的指标。所有子窗口均属于图表组成部分，且属于同一个 Windows 窗口。

long ChartID()

该函数返回当前图表的唯一标识符。

我们将介绍的多数函数均需以图表 ID 作为参数，但你可以为当前图表指定 0，而无需调用 *ChartID* 函数。在 MQL 程序之间传递标识符时，使用 *ChartID* 是合理的，例如在同一个图表或不同图表上交换消息（[自定义事件](#)）时。指定无效的图表 ID 将导致 ERR_CHART_WRONG_ID (4101) 错误。

图表 ID 在会话之间通常保持不变。

我们将在研究获取 [图表列表](#) 的方法后，在示例脚本 *ChartList1.mq5* 中演示 *ChartID* 函数以及标识符的具体形态。

5.7.3 获取图表列表

MQL 程序可以使用 *ChartFirst* 和 *ChartNext* 函数获取终端中打开的图表列表（包括窗口和 图形对象）。

long ChartFirst()

long ChartNext(long chartId)

ChartFirst 函数返回客户端终端中第一个图表的标识符。MetaTrader 5 维护着所有图表的内部列表，其顺序可能与屏幕显示不同，例如在窗口标签页最大化时的顺序。特别是，该列表中的顺序可能会因拖动选项卡、分离窗口和固定窗口而发生变化。加载终端后，书签的可见顺序与内部列表视图保持一致。

ChartNext 函数返回具有指定 *chartId* 的图表之后的图表的 ID。

与其他用于操作图形的函数不同，*ChartId* 参数中的 0 值并非指向当前图表，而是表示列表起始位置。换言之，调用 *ChartNext(0)* 等同于调用 *ChartFirst*。

如果到达列表末尾，该函数返回 -1。

ChartList1.mq5 脚本会将图表列表输出到日志中。主要工作由 *ChartList* 函数完成，该函数在 *OnStart* 中被调用。在函数的起始部分，我们使用 *ChartID* 获取当前图表的标识符，然后在列表中用星号对其进行标记。最后会输出图表的总数。

```
void OnStart()
{
    ChartList();
}

void ChartList()
{
    const long me = ChartID();
    long id = ChartFirst();
    // long id = ChartNext(0); - analogue of calling ChartFirst()
    int count = 0, used = 0;
    Print("Chart List\nN, ID, *active");
    // keep iterating over charts until there are none left
    while(id != -1)
    {
        const string header = StringFormat("%d %lld %s",
            count, id, (id == me ? " *" : ""));

        // fields: N, id, label of the current chart
        Print(header);
        count++;
        id = ChartNext(id);
    }
    Print("Total chart number: ", count);
}
```

以下显示的是一个结果示例。

```

Chart List
N, ID, *active
0 132358585987782873
1 132360375330772909  *
2 132544239145024745
3 132544239145024732
4 132544239145024744
Total chart number: 5

```

5.7.4 获取任意图表的交易品种和时间范围

任何图表的两个基本特性是其工作交易品种和时间范围。如先前所述，当前图表的这些特性既可以通过内置变量 `_Symbol`和 `_Period` 访问，也可以通过相关函数 `Symbol`和 `Period`访问。以下函数可用于确定其他图表的相同特性：`ChartSymbol`和 `ChartPeriod`。

`string ChartSymbol(long chartId = 0)`

该函数返回具有指定标识符的图表的交易品种名称。如果参数为 0，则默认指向当前图表。

如果图表不存在，将返回空字符串（""），并且 `_LastError`将设置错误代码 `ERR_CHART_WRONG_ID` (4101)。

`ENUM_TIMEFRAMES ChartPeriod(long chartId = 0)`

该函数返回具有指定标识符的图表的时间范围值。

如果图表不存在，将返回 0。

脚本 `ChartList2.mq5`,与 `ChartList1.mq5` 类似，可生成显示交易品种和时间范围的图表列表。

```

#include <MQL5Book/Periods.mqh>

void OnStart()
{
    ChartList();
}

void ChartList()
{
    const long me = ChartID();
    long id = ChartFirst();
    int count = 0;

    Print("Chart List\nN, ID, Symbol, TF, *active");
    // keep iterating over charts until there are none left
    while(id != -1)
    {
        const string header = StringFormat("%d %lld %s %s %s",
            count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
            (id == me ? " *" : ""));

        // fields: N, id, symbol, timeframe, label of the current chart
        Print(header);
        count++;
        id = ChartNext(id);
    }
    Print("Total chart number: ", count);
}

```

以下是在 EURUSD, H1 图表上运行脚本后生成的日志内容示例 (见第二行记录)。

```

Chart List
N, ID, Symbol, TF, *active
0 132358585987782873 EURUSD M15
1 132360375330772909 EURUSD H1  *
2 132544239145024745 XAUUSD H1
3 132544239145024732 USDRUB D1
4 132544239145024744 EURUSD H1
Total chart number: 5

```

对于任何图表, MQL5 不仅可以识别交易品种和时间范围, 还可以 [切换交易品种和时间范围](#)。

5.7.5 用于处理完整图表特性集的函数概述

图表特性可通过 `ChartSet` 和 `ChartGet` 函数组进行读取和编辑, 每个函数组包含特定类型的特性: 实数 (`double`)、整数 (`long`、`int`、`datetime`、`color`、`bool`、枚举) 和字符串。

所有函数均将图表 ID 作为第一个参数。值 0 表示当前图表, 即相当于传递调用 `ChartID()` 的结果。然而, 这并不意味着当前图表的 ID 为 0。

描述所有特性的常量构成了三个枚举: `ENUM_CHART_PROPERTY_INTEGER`、`ENUM_CHART_PROPERTY_DOUBLE` 和 `ENUM_CHART_PROPERTY_STRING`, 它们被用作相应类型函数

的参数。所有特性的汇总表可在 MQL5 文档中的 [图表特性](#) 相关页面上找到。在本章的后面几节，我们将逐步介绍几乎所有特性，同时按用途对它们进行分组。唯一的例外是管理图表事件的特性 - 我们将在事件一章的 [相关章节](#) 中对它们进行介绍。

所有三个枚举的元素都被赋予值，使得它们构成一个没有交集 (重复) 的单一列表。因此你可以通过特定值确定枚举的类型。例如，给定一个常量，我们可以持续尝试将其转换为其中一个枚举名称的字符串，直到成功为止。

```
int value = ...;

ResetLastError(); // clear the error code if there was one
EnumToString((ENUM_CHART_PROPERTY_INTEGER)value); // resulting string is not important
if(_LastError == 0) // analyze if there is a new error
{
    // success is an element of ENUM_CHART_PROPERTY_INTEGER
    return ChartGetInteger(0, (ENUM_CHART_PROPERTY_INTEGER)value);
}

ResetLastError();
EnumToString((ENUM_CHART_PROPERTY_DOUBLE)value);
if(_LastError == 0)
{
    // success is an ENUM_CHART_PROPERTY_DOUBLE element
    return ChartGetDouble(0, (ENUM_CHART_PROPERTY_DOUBLE)value);
}

... // continue a similar check for ENUM_CHART_PROPERTY_STRING
```

稍后我们将在测试脚本中使用这种方法。

某些特性 (例如可见柱线数量) 为只读特性，无法更改。这些特性将进一步标记为 “f/o” (只读)。

特性读取函数有简写和完整两种形式：简写形式直接返回请求的值，完整形式返回成功 (*true*) 或错误 (*false*) 的布尔特性，而值本身会被放入通过引用传递的最后一个参数中。使用简写形式时，检查 `_LastError` 变量中的错误代码尤为重要，因为在出现问题时返回的 0 (NULL) 值可能通常会被误认为是正确的。

访问某些特性时，必须指定一个额外的参数 *window*，用于指示图表窗口/子窗口。0 表示主窗口。子窗口编号从 1 开始。某些特性适用于整个图表，因此其对应的函数变体不包含 *window* 参数。

以下是用于读取和写入整数型特性的函数原型。请注意，其中值的类型为 *long*。

```
bool ChartSetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, long value)
bool ChartSetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, int window, long value)
long ChartGetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, int window = 0)
bool ChartGetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, int window, long &value)
```

实数型特性的函数说明类似。子窗口没有可写的实数型特性，因此 `ChartSetDouble` 仅有一种形式，即不带 *window* 参数。

```
bool ChartSetDouble(long chartId, ENUM_CHART_PROPERTY_DOUBLE property, double value)
double ChartGetDouble(long chartId, ENUM_CHART_PROPERTY_DOUBLE property, int window = 0)
bool ChartGetDouble(long chartId, ENUM_CHART_PROPERTY_DOUBLE property, int window, double &value)
```

字符串特性同样遵循此规则，但需要注意一个细节：字符串长度不能超过 2045 个字符（超出部分将被截断）。

```
bool ChartSetString(long chartId, ENUM_CHART_PROPERTY_STRING property, string value)
string ChartGetString(long chartId, ENUM_CHART_PROPERTY_STRING property)
bool ChartGetString(long chartId, ENUM_CHART_PROPERTY_STRING property, string &value)
```

使用 *ChartGetInteger/ChartGetDouble* 简写形式读取特性时，*window* 参数为可选参数，默认指向主窗口（*window=0*）。

设置图表特性的函数（*ChartSetInteger*、*ChartSetDouble*、*ChartSetString*）是异步的，用于向图表发送更改命令。如果这些函数成功执行，该命令会被添加到图表事件的公共队列中，并返回 *true*。当发生错误时，函数将返回 *false*。在这种情况下，你应检查 *_LastError* 变量中的错误代码。

图表特性会在后续处理该图表的事件队列时发生更改，通常会有一定延迟，因此在应用新设置后不应期望图表立即更新。要强制更新图表的外观和特性，请使用 *ChartRedraw* 函数。如果要一次性更改多个图表特性，需在一个代码块中调用相应函数，然后仅调用一次 *ChartRedraw*。

通常，终端会自动更新图表以响应各类事件，例如：新报价到达、图表窗口大小更改、缩放操作、滚动操作、添加指标等。

获取图表特性的函数（*ChartGetInteger*、*ChartGetDouble*、*ChartGetString*）是同步的，这意味着调用代码会等待这些函数的执行结果。

5.7.6 图表描述性特性

ChartSetString/ChartGetString 函数可读取和设置图表的以下字符串特性：

标识符	说明
CHART_COMMENT	图表注释文本
CHART_EXPERT_NAME	图表上运行的 EA 交易名称（只读）
CHART_SCRIPT_NAME	图表上运行的脚本名称（只读）

在 在图表窗口中显示消息 一章中，我们曾了解过 *Comment* 函数，该函数用于在图表左上角显示文本消息。*CHART_COMMENT* 特性可以读取当前图表注释：*ChartGetString(0, CHART_COMMENT)*。还可以通过向该函数传递其他图表的标识符，访问这些图表的注释。通过使用 *ChartSetString*，如果知道图表的 *ID*，则可以修改当前图表及其他图表的注释：*ChartSetString(ID, CHART_COMMENT, "text")*。

如果有 EA 交易或/和脚本正在任意图表上运行，我们可以通过以下调用获取它们的名称：*ChartGetString(ID, CHART_EXPERT_NAME)*和 *ChartGetString(ID, CHART_SCRIPT_NAME)*。

脚本 *ChartList3.mq5*与 *ChartList2.mq5* 类似，可在图表列表中补充有关 EA 交易和脚本的信息。后续我们将为其添加有关指标的信息。

```

void ChartList()
{
    const long me = ChartID();
    long id = ChartFirst();
    int count = 0, used = 0, temp, experts = 0, scripts = 0;

    Print("Chart List\nN, ID, Symbol, TF, *active");
    // keep iterating over charts until there are none left
    while(id != -1)
    {
        temp = 0; // sign of MQL programs on this chart
        const string header = StringFormat("%d %lld %s %s %s",
            count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
            (id == me ? " *" : ""));
        // fields: N, id, symbol, timeframe, label of the current chart
        Print(header);
        string expert = ChartGetString(id, CHART_EXPERT_NAME);
        string script = ChartGetString(id, CHART_SCRIPT_NAME);
        if(StringLen(expert) > 0) expert = "[E] " + expert;
        if(StringLen(script) > 0) script = "[S] " + script;
        if(expert != NULL || script != NULL)
        {
            Print(expert, " ", script);
            if(expert != NULL) experts++;
            if(script != NULL) scripts++;
            temp++;
        }
        count++;
        if(temp > 0)
        {
            used++;
        }
        id = ChartNext(id);
    }
    Print("Total chart number: ", count, ", with MQL-programs: ", used);
    Print("Experts: ", experts, ", Scripts: ", scripts);
}

```

以下是该脚本的输出示例。

```

Chart List
N, ID, Symbol, TF, *active
0 132358585987782873 EURUSD M15
1 132360375330772909 EURUSD H1  *
[S] ChartList3
2 132544239145024745 XAUUSD H1
3 132544239145024732 USDRUB D1
4 132544239145024744 EURUSD H1
Total chart number: 5, with MQL-programs: 1
Experts: 0, Scripts: 1

```

如图所示，当前仅有一个脚本正在执行。

5.7.7 检查主窗口的状态

*ChartSetInteger/ChartGetInteger*函数对可用于查询图表的某些状态特征，还可用于修改其中部分特征。

标识符	说明	值类型
CHART_BRING_TO_TOP	图表活动 (输入焦点)置于所有其他窗口之上	bool
CHART_IS_MAXIMIZED	图表已最大化	bool
CHART_IS_MINIMIZED	图表已最小化	bool
CHART_WINDOW_HANDLE	图表窗口的 Windows 句柄 (只读)	int
CHART_IS_OBJECT	表示图表是图表对象 (OBJ_CHART)的标志； <i>true</i> 表示图形对象， <i>false</i> 表示普通图表 (只读)	bool

正如预期的一样，图表对象的窗口句柄和特性均为只读。其他特性可编辑：例如，通过调用 *ChartSetInteger(ID, CHART_BRING_TO_TOP, true)*，你可以激活指定 ID 的图表。

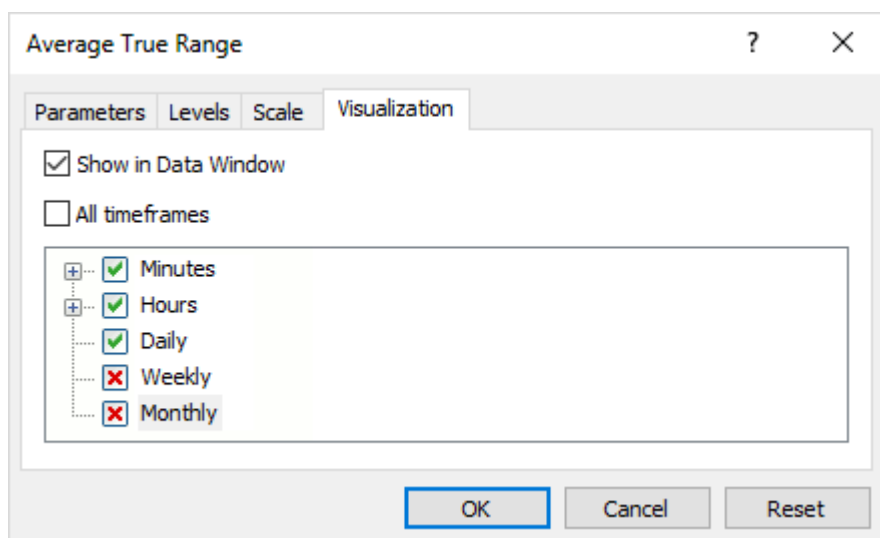
下一节的 *ChartList4.mq5*脚本提供应用这些特性的示例。

5.7.8 获取窗口/子窗口的数量和可见性

使用 *ChartGetInteger*函数，MQL 程序可以查找图表上的窗口数量 (包括子窗口)及其可见性状态。

标识符	说明	值类型
CHART_WINDOWS_TOTAL	图表窗口的总数，包括指标子窗口 (只读)。	int
CHART_WINDOW_IS_VISIBLE	子窗口可见性，'window' 参数为子窗口编号 (只读)。	bool

如果某些指标在当前时间范围的“特性”对话框的“可视化”选项卡中被禁用，则对应的子窗口可能会被隐藏。无法重置所有标志：由于 [tpl模板](#)的存储机制，这种状态会被解释为启用所有时间范围。因此，如果用户希望在某些时间段隐藏子窗口，必须在最不常用的时间范围上至少保留一个启用标志。



在不同时间范围上设置指标可见性

需要注意的是，MQL5 中未提供通过编程方式确定特定标志的状态和切换的标准工具。模拟这种控制的最简单方法是保存 tpl 模板并对其进行分析，后续可能进行编辑和加载（请参阅 [处理 tpl 模板](#) 章节）。

在新版脚本 *ChartList4.mq5* 中，我们输出以下内容：子窗口数量（始终存在一个主窗口）、图表活动状态标志、图表对象标志、Windows 句柄。

```
const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
const string header = StringFormat("%d %lld %s %s %s %s %s %s %lld",
    count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
    (win > 1 ? "#" + (string)(win - 1) : ""), (id == me ? "*" : ""),
    (ChartGetInteger(id, CHART_BRING_TO_TOP, 0) ? "active" : ""),
    (ChartGetInteger(id, CHART_IS_OBJECT) ? "object" : ""),
    ChartGetInteger(id, CHART_WINDOW_HANDLE));
...
for(int i = 0; i < win; i++)
{
    const bool visible = ChartGetInteger(id, CHART_WINDOW_IS_VISIBLE, i);
    if(!visible)
    {
        Print(" ", i, "/Hidden");
    }
}
```

以下是可能的输出结果。


```
Chart List
N, ID, Symbol, TF, #subwindows, *active, Windows handle
0 132358585987782873 EURUSD M15 #1 68030
1 132360375330772909 EURUSD H1 * active 68048
[S] ChartList4
2 132544239145024745 XAUUSD H1 395756
3 132544239145024732 USDRUB D1 395768
4 132544239145024744 EURUSD H1 #2 461286
2/Hidden
Total chart number: 5, with MQL-programs: 1
Experts: 0, Scripts: 1
```

在第一张图表 (索引 0) 上有一个子窗口 (#1)。在最后一张图表上有两个子窗口 (#2)，并且第二个子窗口当前处于隐藏状态。稍后，在 [管理图表上的指标](#) 章节中，我们将展示完整版的 *ChartList.mq5* 脚本，其中报告内容将包含位于子窗口和主窗口中的指标信息。

注意！对于 [图表对象](#) 中的图表，其 `CHART_WINDOW_IS_VISIBLE` 特性始终为 *true*，即使该对象的可视化在当前时间范围或全部时间范围中已被禁用。

5.7.9 图表显示模式

`ENUM_CHART_PROPERTY_INTEGER` 枚举中的四个特性描述了图表显示模式。所有这些特性均可通过 *ChartGetInteger* 读取，并通过 *ChartSetInteger* 记录，从而允许你更改图表的外观。

标识符	说明	值类型
CHART_MODE	图表类型 (蜡烛图、柱线或线图)	ENUM_CHART_MODE
CHART_FOREGROUND	价格图表置于前景	bool
CHART_SHIFT	价格图表距右侧边距缩进模式	bool
CHART_AUTOSCROLL	自动滚动至图表右侧边缘	bool

在 MQL5 中，`CHART_MODE` 模式有一个特殊枚举 `ENUM_CHART_MODE`。其元素如下表所示。

标识符	说明	值
CHART_BARS	以柱线显示	0
CHART_CANDLES	以日式蜡烛图显示	1
CHART_LINE	以收盘价连线显示	2

我们来实现一个名为 *ChartMode.mq5* 的脚本，该脚本将监控各种模式的状态，并在检测到更改时将消息输出到日志中。由于特性处理算法具有通用性，我们将它们放在一个单独的头文件 *ChartModeMonitor.mqh* 中，然后将其连接到不同的测试中。

让我们在抽象类 *ChartModeMonitorInterface* 中奠定基础：它为所有类型提供重载的 `get` 和 `set` 方法。派生类将需要通过重写虚方法 *snapshot* 直接检查特性，以达到所需的检查程度。

```

class ChartModeMonitorInterface
{
public:
    long get(const ENUM_CHART_PROPERTY_INTEGER property, const int window = 0)
    {
        return ChartGetInteger(0, property, window);
    }
    double get(const ENUM_CHART_PROPERTY_DOUBLE property, const int window = 0)
    {
        return ChartGetDouble(0, property, window);
    }
    string get(const ENUM_CHART_PROPERTY_STRING property)
    {
        return ChartGetString(0, property);
    }
    bool set(const ENUM_CHART_PROPERTY_INTEGER property, const long value, const int w
    {
        return ChartSetInteger(0, property, window, value);
    }
    bool set(const ENUM_CHART_PROPERTY_DOUBLE property, const double value)
    {
        return ChartSetDouble(0, property, value);
    }
    bool set(const ENUM_CHART_PROPERTY_STRING property, const string value)
    {
        return ChartSetString(0, property, value);
    }

    virtual void snapshot() = 0;
    virtual void print() { };
    virtual void backup() { }
    virtual void restore() { }
};

```

此类还包含预留方法：*print*（例如用于输出到日志）、*backup*（用于保存当前状态）以及*restore*（用于恢复状态）。这些方法被声明为非抽象方法，但实现为空，因为它们是可选的。

将不同类型的特性定义为特定类是合理的，这些类可以作为从 *ChartModeMonitorInterface* 继承的单一模板，并接受参数化值（T）和枚举（E）类型。例如，对于整数类型的特性，你需要设置 *T=long* 和 *E=ENUM_CHART_PROPERTY_INTEGER*。

该对象包含 *data* 数组，用于存储所有请求特性的 [键, 值] 对。它具有通用类型 *MapArray<K,V>*，我们在 [多货币和多时间范围指标](#) 一章中针对指标 *IndUnityPercent* 介绍过该类型。其独特之处在于，除了通常按数字访问数组元素外，还可以使用按键寻址。

为了填充该数组，需要将一个整数数组传递给构造函数，同时将先使用 *detect* 方法检查这些整数是否符合给定枚举类型 E 的标识符规范。所有正确的特性将通过 *get* 调用立即读取，得到的值会与其标识符一起存储在映射中。

```

#include <MQL5Book/MapArray.mqh>

template<typename T,typename E>
class ChartModeMonitorBase: public ChartModeMonitorInterface
{
protected:
    MapArray<E,T> data; // array-map of pairs [property, value]

    // the method checks if the passed constant is an enumeration element,
    // and if it is, then add it to the map array
    bool detect(const int v)
    {
        ResetLastError();
        EnumToString((E)v); // resulting string is not used
        if(_LastError == 0) // it only matters if there is an error or not
        {
            data.put((E)v, get((E)v));
            return true;
        }
        return false;
    }

public:
    ChartModeMonitorBase(int &flags[])
    {
        for(int i = 0; i < ArraySize(flags); ++i)
        {
            detect(flags[i]);
        }
    }

    virtual void snapshot() override
    {
        MapArray<E,T> temp;
        // collect the current state of all properties
        for(int i = 0; i < data.GetSize(); ++i)
        {
            temp.put(data.getKey(i), get(data.getKey(i)));
        }

        // compare with previous state, display differences
        for(int i = 0; i < data.GetSize(); ++i)
        {
            if(data[i] != temp[i])
            {
                Print(EnumToString(data.getKey(i)), " ", data[i], " -> ", temp[i]);
            }
        }

        // save for next comparison
        data = temp;
    }
};

```

```

    }
    ...
};

```

*snapshot*方法会遍历数组的所有元素，并为每个特性请求值。由于我们希望检测变更，新数据会先存储在一个临时映射数组 *temp* 中。然后，将数组 *data* 和 *temp* 进行逐元素对比，针对每个差异项，显示一条包含特性名称、旧值和新值的消息。此简化示例仅使用日志。然而，如果有必要，程序可以调用一些使行为适应环境的应用函数。

print、*backup* 和 *restore* 方法以尽可能简单的方式实现。

```

template<typename T,typename E>
class ChartModeMonitorBase: public ChartModeMonitorInterface
{
protected:
    ...
    MapArray<E,T> store; // backup
public:
    ...
    virtual void print() override
    {
        data.print();
    }
    virtual void backup() override
    {
        store = data;
    }

    virtual void restore() override
    {
        data = store;
        // restore chart properties
        for(int i = 0; i < data.getSize(); ++i)
        {
            set(data.getKey(i), data[i]);
        }
    }
}

```

通过结合使用 *backup/restore* 方法，可以在开始对图表进行实验之前保存其状态，并在测试脚本完成后将一切恢复为初始状态。

最后，*ChartModeMonitor.mqh* 文件中的最后一个类是 *ChartModeMonitor*。它组合了三个 *ChartModeMonitorBase* 实例，这些实例是为特性类型的可用组合创建的。它们有一个指向基础接口 *ChartModeMonitorInterface* 的 *m* 指针数组。该类本身也从该基础接口继承而来。

```

#include <MQL5Book/AutoPtr.mqh>

#define CALL_ALL(A,M) for(int i = 0, size = ArraySize(A); i < size; ++i) A[i][].M

class ChartModeMonitor: public ChartModeMonitorInterface
{
    AutoPtr<ChartModeMonitorInterface> m[3];

public:
    ChartModeMonitor(int &flags[])
    {
        m[0] = new ChartModeMonitorBase<long,ENUM_CHART_PROPERTY_INTEGER>(flags);
        m[1] = new ChartModeMonitorBase<double,ENUM_CHART_PROPERTY_DOUBLE>(flags);
        m[2] = new ChartModeMonitorBase<string,ENUM_CHART_PROPERTY_STRING>(flags);
    }

    virtual void snapshot() override
    {
        CALL_ALL(m, snapshot());
    }

    virtual void print() override
    {
        CALL_ALL(m, print());
    }

    virtual void backup() override
    {
        CALL_ALL(m, backup());
    }

    virtual void restore() override
    {
        CALL_ALL(m, restore());
    }
};

```

为了简化代码，此处使用 `CALL_ALL` 宏，该宏会为数组中的所有对象调用指定方法，并且在调用时会考虑 `AutoPtr`类中重载的 `[]` 运算符（该运算符用于解引用智能指针并获取指向“受保护”对象的直接指针）。

析构函数通常负责释放对象，但在这种情况下，决定使用 `AutoPtr`数组（之前在 [对象类型模板](#)一节中讨论过这个类）。这确保了在正常释放 `m`数组时，动态对象会被自动删除。

在文件 `ChartModeMonitorFull.mqh`中提供了一个支持子窗口编号的更完整版本的监控器。

基于 `ChartModeMonitor`类，可以轻松实现预期的脚本 `ChartMode.mq5`。其任务是每半秒检查一次给定特性集合的状态。目前，我们在此处使用无限循环和 `Sleep`，但很快我们将学习如何以不同的方式响应图表上的事件：通过终端发送的通知。

```

#include <MQL5Book/ChartModeMonitor.mqh>

void OnStart()
{
    int flags[] =
    {
        CHART_MODE, CHART_FOREGROUND, CHART_SHIFT, CHART_AUTOSCROLL
    };
    ChartModeMonitor m(flags);
    Print("Initial state:");
    m.print();
    m.backup();

    while(!IsStopped())
    {
        m.snapshot();
        Sleep(500);
    }
    m.restore();
}

```

在任意图表上运行该脚本，并尝试使用工具按钮切换模式。这样，除了 CHART_FOREGROUND 外，你可以访问所有元素，该特性可从“特性”对话框 *Common* 选项卡，*Chart on top* 标志)中切换。



图表模式切换工具栏按钮

例如，以下日志是在进行以下操作之后生成的：将显示从蜡烛图切换为柱线，再从柱线切换为折线图，然后切换回蜡烛图，随后启用缩进和自动滚动到起始位置。

Initial state:

	[key]	[value]
[0]	0	1
[1]	1	0
[2]	2	0
[3]	4	0

```

CHART_MODE 1 -> 0
CHART_MODE 0 -> 2
CHART_MODE 2 -> 1
CHART_SHIFT 0 -> 1
CHART_AUTOSCROLL 0 -> 1

```

使用 CHART_MODE 特性的一个更实用的示例是改进版 *IndSubChart.mq5* 指标（我们在 [多货币和多时间范围指标](#) 一节中讨论过简化版 *IndSubChartSimple.mq5*）。此指标用于在子窗口中显示第三方交易品种的报价，在早期版本中，我们必须通过输入参数向用户请求图表显示方式（蜡烛图、柱线或折线图）。现在不再需要该参数，因为我们可以自动将指标切换为主窗口中所使用的模式。

当前模式存储在全局变量 *mode* 中，并在初始化期间完成首次赋值。

```

ENUM_CHART_MODE mode = 0;

int OnInit()
{
    ...
    mode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE);
    ...
}

```

新模式的检测最好在专门设计的事件处理程序 *OnChartEvent* 中完成，我们将在单独的 [章节](#) 中学习这一内容。在这个阶段，重要的是要知道：如果代码中描述了具有预定义原型（名称和参数列表）的函数，则每当图表发生任何变化时，MQL 程序都可以从终端接收通知。特别是，它的第一个参数包含一个描述其含义的事件标识符。我们仍然关注图表本身，因此我们会检查 *eventId* 是否等于 CHARTEVENT_CHART_CHANGE。这是必要的，因为该处理程序还能够追踪图形对象、键盘、鼠标和任意用户消息。

```
void OnChartEvent(const int eventId,
                  // parameters not used here
                  const long &, const double &, const string &)
{
    if(eventId == CHARTEVENT_CHART_CHANGE)
    {
        const ENUM_CHART_MODE newmode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE)
        if(mode != newmode)
        {
            const ENUM_CHART_MODE oldmode = mode;
            mode = newmode;
            // change buffer bindings and rendering type on the go
            InitPlot(0, InitBuffers(mode), Mode2Style(mode));
            // TODO: we will auto-adjust colors later
            // SetPlotColors(0, mode);
            if(oldmode == CHART_LINE || newmode == CHART_LINE)
            {
                // switching to or from CHART_LINE mode requires updating the entire char
                // because the number of buffers changes
                Print("Refresh");
                ChartSetSymbolPeriod(0, _Symbol, _Period);
            }
            else
            {
                // when switching between candles and bars, it is enough
                // just redraw the chart in a new manner,
                // because data doesn't change (previous 4 buffers with values)
                Print("Redraw");
                ChartRedraw();
            }
        }
    }
}
```

你可以通过在图表上运行新指标并切换绘图方法来自行测试该指标。

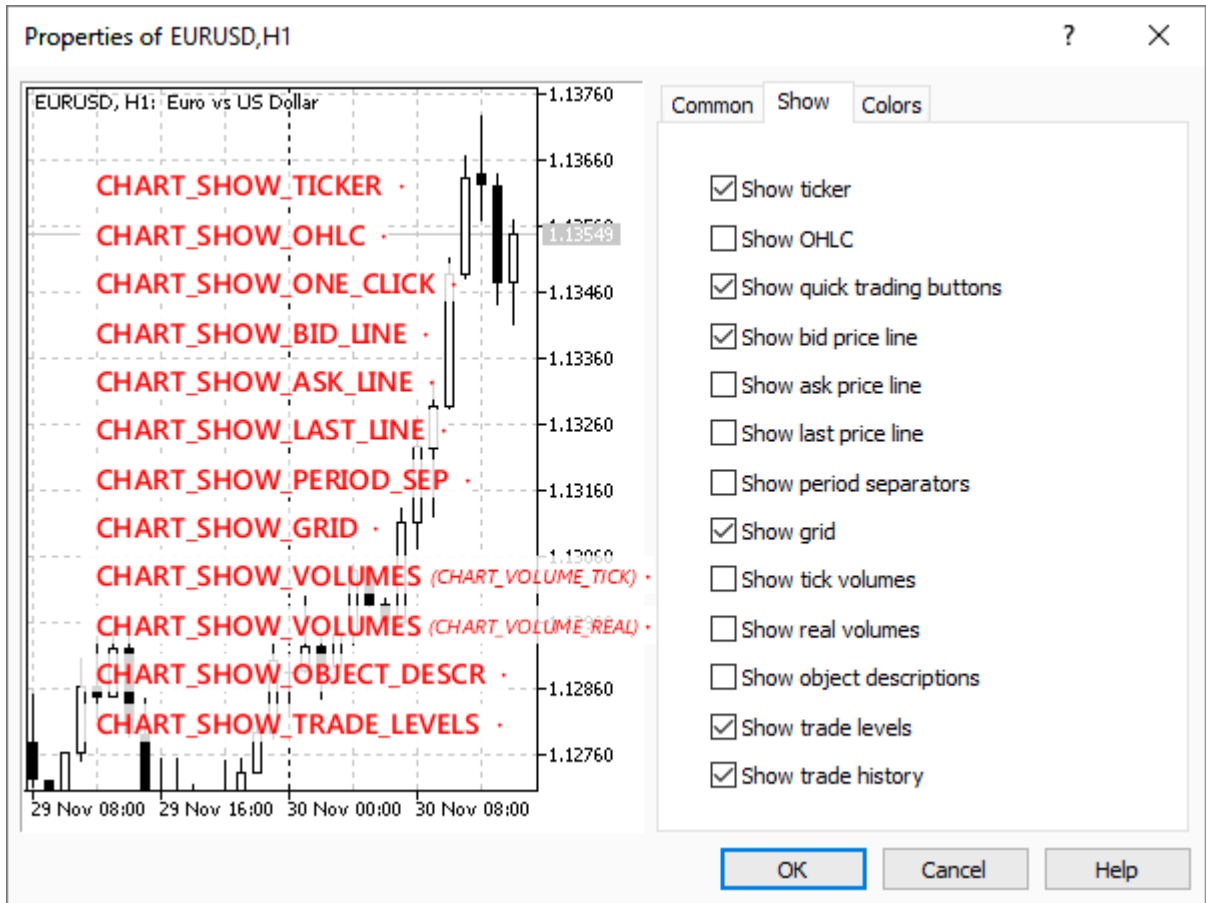
这些并非是 *IndSubChart.mq5* 中所做的全部改进。稍后，在 [图表颜色](#) 一节中，我们将展示如何使图形自动适配图表配色方案。

5.7.10 管理图表元素的可见性

枚举类型 `ENUM_CHART_PROPERTY_INTEGER` 中的大量特性用于控制图表元素的可见性。几乎所有这些特性均为布尔类型：*true*对应显示元素，*false* 对应隐藏元素。例外情况是 `CHART_SHOW_VOLUMES`，它使用 `ENUM_CHART_VOLUME_MODE` 枚举（见下文）。

标识符	说明	值类型
CHART_SHOW	常规价格图表显示。如果设置为 false，则将禁用所有价格图表特性的渲染，并移除图表边缘的所有填	bool

标识符	说明	值类型
	充内容：时间和价格刻度、快速导航栏、日历事件标记、交易图标、指标和柱线工具提示、指标子窗口、成交量直方图等。	
CHART_SHOW_TICKER	在左上角显示交易品种代码。禁用交易品种代码会自动禁用 OHLC (CHART_SHOW_OHLC)	bool
CHART_SHOW_OHLC	在左上角显示 OHLC 值。启用 OHLC 会自动启用交易品种代码 (CHART_SHOW_TICKER)	bool
CHART_SHOW_BID_LINE	将买价显示为一条水平线	bool
CHART_SHOW_ASK_LINE	将卖价显示为一条水平线	bool
CHART_SHOW_LAST_LINE	将最新价显示为一条水平线	bool
CHART_SHOW_PERIOD_SEP	显示相邻周期之间的垂直分隔线	bool
CHART_SHOW_GRID	在图表上显示网格线	bool
CHART_SHOW_VOLUMES	在图表上显示成交量	ENUM_CHART_VOLUME_MODE
CHART_SHOW_OBJECT_DESCR	显示对象的文本描述 (并非所有类型的对象都会显示描述)	bool
CHART_SHOW_TRADE_LEVELS	在图表上显示交易价位 (开仓价位、止损位、止盈位和挂单价位)	bool
CHART_SHOW_DATE_SCALE	在图表上显示日期刻度	bool
CHART_SHOW_PRICE_SCALE	在图表上显示价格刻度	bool
CHART_SHOW_ONE_CLICK	在图表上显示快速交易面板 (“一键交易”选项)	bool



在设置对话框中部分 ENUM_CHART_PROPERTY_INTEGER 特性的标志

对于其中部分特性，用户可通过图表上下文菜单使用，而一些特性只能通过设置对话框使用。还有一些设置只能在 MQL5 中更改，特别是垂直 (CHART_SHOW_DATE_SCALE) 刻度和水平 (CHART_SHOW_DATE_SCALE) 刻度的显示，以及整个图表的可见性 (CHART_SHOW)。最后一种情况尤其值得注意，因为关闭渲染功能是通过图形对象 图形资源 和 图形对象创建自定义程序界面的理想解决方案，无论 CHART_SHOW 的值如何，这些对象始终会被渲染。

这本书附带脚本 *ChartBlackout.mq5*，该脚本每次运行时会将 CHART_SHOW 模式从当前状态切换为相反状态。

```
void OnStart()
{
    ChartSetInteger(0, CHART_SHOW, !ChartGetInteger(0, CHART_SHOW));
}
```

因此，你可以在常规图表上应用该脚本以完全清空窗口，然后再次应用它以恢复先前的外观。

上述 ENUM_CHART_VOLUME_MODE 枚举包含以下成员：

标识符	说明	值
CHART_VOLUME_HIDE	成交量已隐藏	0
CHART_VOLUME_TICK	分时报价成交量	1
CHART_VOLUME_REAL	交易成交量 (如有)	2

与脚本 *ChartMode.mq5* 类似，我们在脚本 *ChartElements.mq5* 中实现了图表元素的可见性监控功能。主要区别在于所控制的标志集合不同。

```
void OnStart()
{
    int flags[] =
    {
        CHART_SHOW,
        CHART_SHOW_TICKER, CHART_SHOW_OHLC,
        CHART_SHOW_BID_LINE, CHART_SHOW_ASK_LINE, CHART_SHOW_LAST_LINE,
        CHART_SHOW_PERIOD_SEP, CHART_SHOW_GRID,
        CHART_SHOW_VOLUMES,
        CHART_SHOW_OBJECT_DESCR,
        CHART_SHOW_TRADE_LEVELS,
        CHART_SHOW_DATE_SCALE, CHART_SHOW_PRICE_SCALE,
        CHART_SHOW_ONE_CLICK
    };
    ...
}
```

此外，在创建设置备份后，我们会通过编程方式主动禁用时间刻度和价格刻度（当脚本结束时，会从备份中恢复这些设置）。

```
...
m.backup();

ChartSetInteger(0, CHART_SHOW_DATE_SCALE, false);
ChartSetInteger(0, CHART_SHOW_PRICE_SCALE, false);
...
}
```

以下是日志片段，包含了关于所采取操作的注释。前两条记录的出现，正是在创建初始备份后，MQL 代码中禁用了刻度显示。

```
CHART_SHOW_DATE_SCALE 1 -> 0 // disabled the time scale in the MQL5 code
CHART_SHOW_PRICE_SCALE 1 -> 0 // disabled the price scale in the MQL5 code
CHART_SHOW_ONE_CLICK 0 -> 1 // disabled "One click trading"
CHART_SHOW_GRID 1 -> 0 // disable "Grid"
CHART_SHOW_VOLUMES 0 -> 2 // showed real "Volumes"
CHART_SHOW_VOLUMES 2 -> 1 // showed "Tick volumes"
CHART_SHOW_TRADE_LEVELS 1 -> 0 // disabled "Trade levels"
```

5.7.11 水平偏移

图表显示的另一个细节在于左右边缘的水平缩进。它们的运行方式略有差异，但在同一个枚举 `ENUM_CHART_PROPERTY_DOUBLE` 中定义，并使用 *double* 类型。

标识符	说明
CHART_SHIFT_SIZE	零柱线距右侧边缘的缩进，以百分比表示 (范围为 10% 至 50%)。仅在启用 CHART_SHIFT 模式时有效。在图表上，该偏移通过窗口右侧顶部边框的灰色小倒三角形表示。
CHART_FIXED_POSITION	图表固定位置距左边缘的距离，以百分比表示 (范围为 0 至 100%)。图表固定位置通过水平时间轴的灰色小三角形表示，并且仅在禁用“新分时报价到达时自动向右滚动”(CHART_AUTOCROLL)时才显示。当你进行缩放操作时，处于固定位置的柱线会保持位置不变。默认情况下，该三角形位于图表的 (左下角)最角落位置。



水平边距特性的可视化呈现

我们提供了 *ChartShifts.mq5*脚本来检查对这些特性的可访问性，其工作原理与 *ChartMode.mq5*类似，仅控制的控特性集有所不同。

```
void OnStart()
{
    int flags[] =
    {
        CHART_SHIFT_SIZE, CHART_FIXED_POSITION
    };
    ChartModeMonitor m(flags);
    ...
}
```

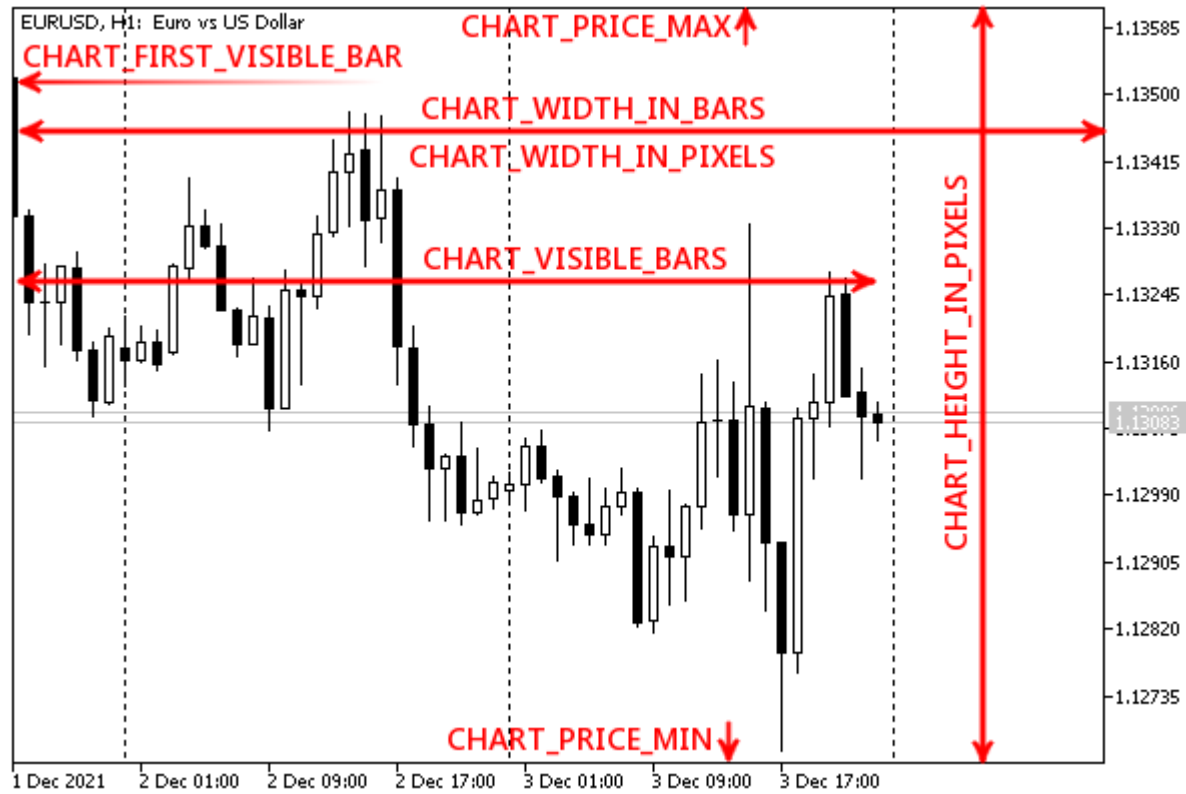
通过鼠标拖动固定定位标签 (左下角)将生成如下日志输出。

```
Initial state:
    [key]  [value]
[0]      3 21.78771
[1]     41 17.87709
CHART_FIXED_POSITION 17.87709497206704 -> 26.53631284916201
CHART_FIXED_POSITION 26.53631284916201 -> 27.93296089385475
CHART_FIXED_POSITION 27.93296089385475 -> 28.77094972067039
CHART_FIXED_POSITION 28.77094972067039 -> 50.0
```

5.7.12 水平刻度 (按时间)

要确定水平轴上的缩放比例和柱线数量,可使用 `ENUM_CHART_PROPERTY_INTEGER` 枚举中的整数特性组。在这些特性中,只有 `CHART_SCALE` 可编辑。

标识符	说明
CHART_SCALE	缩放比例 (0 到 5)
CHART_VISIBLE_BARS	图表上当前可见的柱线数量 (由于 <code>CHART_SHIFT_SIZE</code> 缩进,可能小于 <code>CHART_WIDTH_IN_BARS</code>) (只读)
CHART_FIRST_VISIBLE_BAR	图表上第一个可见柱线的编号编号方式与时间序列一致,从右向左递增 (只读)
CHART_WIDTH_IN_BARS	图表宽度,用柱线数量表示 (潜在容量,左右两端的柱线可能仅部分可见) (只读)
CHART_WIDTH_IN_PIXELS	图表宽度,用像素表示 (只读)



图表上的 ENUM_CHART_PROPERTY_INTEGER 特性

我们已准备好实现下一个测试脚本 *ChartScaleTime.mq5*, 该脚本可用于分析这些特性的变化。

```
void OnStart()
{
    int flags[] =
    {
        CHART_SCALE,
        CHART_VISIBLE_BARS,
        CHART_FIRST_VISIBLE_BAR,
        CHART_WIDTH_IN_BARS,
        CHART_WIDTH_IN_PIXELS
    };
    ChartModeMonitor m(flags);
    ...
}
```

以下是部分日志内容, 包含了关于所采取操作的注释。

```
Initial state:
    [key] [value]
[0]     5     4
[1]   100    35
[2]   104    34
[3]   105    45
[4]   106   715

                                // 1) changed the scale to a smaller one:
CHART_SCALE 4 -> 3                // - the value of the "scale" property has changed
CHART_VISIBLE_BARS 35 -> 69        // - increased the number of visible bars
CHART_FIRST_VISIBLE_BAR 34 -> 68 // - the number of the first visible bar has increas
CHART_WIDTH_IN_BARS 45 -> 90 // - increased the potential number of bars
                                // 2) disabled padding at the right edge
CHART_VISIBLE_BARS 69 -> 89 // - the number of visible bars has increased
CHART_FIRST_VISIBLE_BAR 68 -> 88 // - the number of the first visible bar has increas
                                // 3) reduced the window size
CHART_VISIBLE_BARS 89 -> 86 // - number of visible bars decreased
CHART_WIDTH_IN_BARS 90 -> 86 // - the potential number of bars has decreased
CHART_WIDTH_IN_PIXELS 715 -> 680 // - decreased width in pixels
                                // 4) clicked the "End" button to move to the curren
CHART_VISIBLE_BARS 86 -> 85 // - number of visible bars decreased
CHART_FIRST_VISIBLE_BAR 88 -> 84 // - the number of the first visible bar has decreas
```

5.7.13 垂直刻度 (按价格和指标读数)

与垂直刻度相关的特性通过两个枚举的元素进行设置和解析: ENUM_CHART_PROPERTY_INTEGER 和 ENUM_CHART_PROPERTY_DOUBLE. 下表列出了这些特性及其值类型。

某些特性不仅允许访问主窗口, 还允许访问子窗口。对于这些特性, *ChartSet* 和 *ChartGet* 函数应使用 *window* 参数 0 表示主窗口, 且表示简版 *ChartGet* 函数的默认值)。

标识符	说明	值类型
CHART_SCALEFIX	固定刻度模式	bool
CHART_FIXED_MAX	<i>window</i> 子窗口的固定最大值或主窗口的初始最大值	double
CHART_FIXED_MIN	<i>window</i> 子窗口的固定最小值或主窗口的初始最小值	double
CHART_SCALEFIX_11	1:1 刻度模式	bool
CHART_SCALE_PT_PER_BAR	刻度显示模式，以“点/柱”为单位	bool
CHART_POINTS_PER_BAR	每根柱线的刻度值，以点数为单位	double
CHART_PRICE_MIN	<i>window</i> 或子窗口中的最小值（只读）	double
CHART_PRICE_MAX	<i>window</i> 或子窗口中的最大值（只读）	double
CHART_HEIGHT_IN_PIXELS	窗口或子窗口的固定高度，以像素为单位（需要 <i>window</i> 参数）	int
CHART_WINDOW_YDISTANCE	<i>window</i> 子窗口顶部边框与主图表窗口上边框沿垂直 Y 轴的像素距离。（只读）	int

默认情况下，图表支持自适应刻度，以便报价线或指标线在可见时间段内完全垂直适配。对于某些应用场景，需要固定刻度，为此终端提供了多种模式。在这些模式中，图表不仅可以通过鼠标或按键（Shift + 箭头）进行左右滚动，还可以进行上下滚动，并且右侧刻度会出现一个滑动条，使用该滑动条可以通过鼠标快速滚动图表。

要设置固定模式，可启用 CHART_SCALEFIX 标志并在 CHART_FIXED_MAX 和 CHART_FIXED_MIN 字段中指定所需的最大值和最小值（在主窗口中，用户可以上下移动图表，这将导致 CHART_FIXED_MAX 和 CHART_FIXED_MIN 的值同步变化，但垂直刻度将保持不变）。用户还可以通过在价格刻度区域按住鼠标左键不放，上下移动来更改垂直刻度。子窗口不提供垂直刻度的交互式编辑。在这一方面，我们稍后将介绍一个名为 *SubScaler.mq5* 的指标（请参阅 [键盘事件章节](#)），该指标允许用户通过键盘控制子窗口中的值范围，而不是通过 *Scale* 选项卡上的字段从设置对话框进行控制。

CHART_SCALEFIX_11 模式可以使屏幕上正方形各边在视觉上近似相等：X 根柱线的（水平）像素长度等于 X 个点的（垂直）像素高度。这种相等是近似的，因为通常像素的垂直尺寸和水平尺寸并不相同。

最后，还有一种用于固定每根柱线点数比例的模式，该模式通过 CHART_SCALE_PT_PER_BAR 选项启用，而所需的比例本身则通过 CHART_POINTS_PER_BAR property 特性进行设置。与 CHART_SCALEFIX 模式不同，用户将无法在图表上通过鼠标交互式地更改刻度。在这种模式下，在屏幕上，一根柱线的水平距离与指定垂直点数的比例，将与图表的宽高比保持一致，以像素为单位。如果两张图表的时间范围和尺寸相等，根据它们的 CHART_POINTS_PER_BAR 值的比例，其中一张图表的价格显示会比另一张显得更紧凑。显然，时间范围越小，柱线的范围就越小，因此在相同刻度下，较小的时间范围看起来更“平缓”。

通过编程设置 CHART_HEIGHT_IN_PIXELS 特性会导致用户无法编辑窗口/子窗口的大小。这通常用于承载交易面板的窗口，该面板包含预定义的控制组件（如按钮、输入字段等）。如果要取消尺寸固定，需将该特性的值设置为 -1。

为正确处理图形对象，需要使用 `CHART_WINDOW_YDISTANCE` 值将主图表的绝对坐标转换为子窗口的局部坐标。问题的关键在于，当发生 [鼠标事件](#) 时，光标坐标是相对于主图表窗口传输的，而指标子窗口中图形对象的坐标则是相对于子窗口的左上角设置的。

我们来准备用于分析垂直刻度和尺寸变化的 *ChartScalePrice.mq5* 脚本。

```
void OnStart()
{
    int flags[] =
    {
        CHART_SCALEFIX, CHART_SCALEFIX_11,
        CHART_SCALE_PT_PER_BAR, CHART_POINTS_PER_BAR,
        CHART_FIXED_MAX, CHART_FIXED_MIN,
        CHART_PRICE_MIN, CHART_PRICE_MAX,
        CHART_HEIGHT_IN_PIXELS, CHART_WINDOW_YDISTANCE
    };
    ChartModeMonitor m(flags);
    ...
}
```

它通过以下方式对图表操作做出反应：


```
Initial state:
    [key] [value]    // ENUM_CHART_PROPERTY_INTEGER
[0]      6         0
[1]      7         0
[2]     10         0
[3]    107        357
[4]    110         0
    [key] [value]    // ENUM_CHART_PROPERTY_DOUBLE
[0]     11 10.00000
[1]      8  1.13880
[2]      9  1.12330
[3]    108  1.12330
[4]    109  1.13880
// reduced the vertical size of the window
CHART_HEIGHT_IN_PIXELS 357 -> 370
CHART_HEIGHT_IN_PIXELS 370 -> 408
CHART_FIXED_MAX 1.1389 -> 1.1388
CHART_FIXED_MIN 1.1232 -> 1.1233
CHART_PRICE_MIN 1.1232 -> 1.1233
CHART_PRICE_MAX 1.1389 -> 1.1388
// reduced the horizontal scale, which increased the price range
CHART_FIXED_MAX 1.1388 -> 1.139
CHART_FIXED_MIN 1.1233 -> 1.1183
CHART_PRICE_MIN 1.1233 -> 1.1183
CHART_PRICE_MAX 1.1388 -> 1.139
CHART_FIXED_MAX 1.139 -> 1.1406
CHART_FIXED_MIN 1.1183 -> 1.1167
CHART_PRICE_MIN 1.1183 -> 1.1167
CHART_PRICE_MAX 1.139 -> 1.1406
// expand the price range using the mouse (quotes "shrink" vertically)
CHART_FIXED_MAX 1.1406 -> 1.1454
CHART_FIXED_MIN 1.1167 -> 1.1119
CHART_PRICE_MIN 1.1167 -> 1.1119
CHART_PRICE_MAX 1.1406 -> 1.1454
```

5.7.14 颜色

MQL 程序可以识别和更改颜色，以显示所有图表元素。对应的特性是 ENUM_CHART_PROPERTY_INTEGER 枚举的一部分。

标识符	说明
CHART_COLOR_BACKGROUND	图表背景色
CHART_COLOR_FOREGROUND	坐标轴、刻度和 OHLC 线的颜色
CHART_COLOR_GRID	网格颜色
CHART_COLOR_VOLUME	成交量与建仓水平的颜色
CHART_COLOR_CHART_UP	上涨柱线、影线和实体边框的颜色

标识符	说明
CHART_COLOR_CHART_DOWN	下跌柱线、影线和实体边框的颜色
CHART_COLOR_CHART_LINE	图表线条和日式蜡烛图轮廓的颜色
CHART_COLOR_CANDLE_BULL	上涨蜡烛图实体颜色
CHART_COLOR_CANDLE_BEAR	下跌蜡烛图实体颜色
CHART_COLOR_BID	买价线颜色
CHART_COLOR_ASK	卖价线颜色
CHART_COLOR_LAST	最后交易价格线 (Last) 的颜色
CHART_COLOR_STOP_LEVEL	止损和止盈订单水平的颜色 (止损位和止盈位颜色)

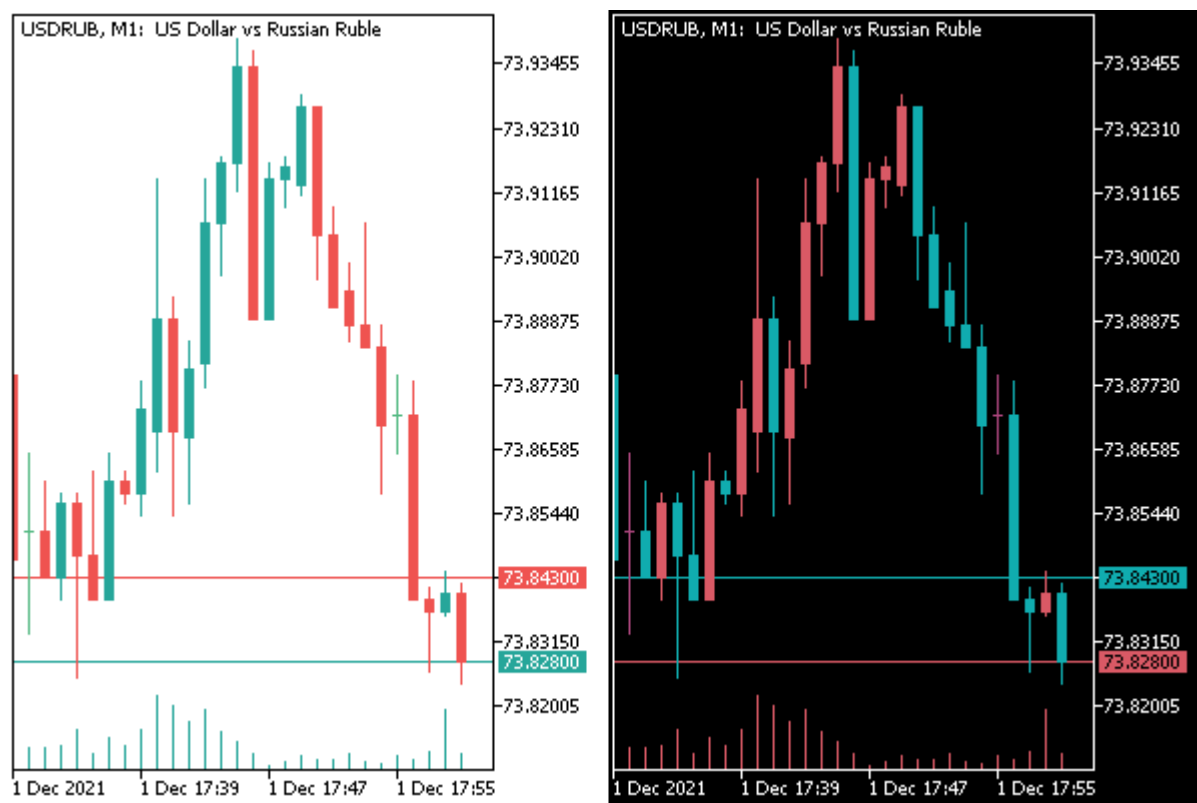
作为使用这些特性的示例，我们将创建一个脚本 *ChartColorInverse.mq5*。它会将图形的所有颜色转换为反色，即对颜色的二进制表示采用以下格式进行处理：**RGB XOR (^,XOR)**。因此，在同一图表上重新启动该脚本后，其设置将恢复。

```
#define RGB_INVERSE(C) ((color)C ^ 0xFFFFFF)

void OnStart()
{
    ENUM_CHART_PROPERTY_INTEGER colors[] =
    {
        CHART_COLOR_BACKGROUND,
        CHART_COLOR_FOREGROUND,
        CHART_COLOR_GRID,
        CHART_COLOR_VOLUME,
        CHART_COLOR_CHART_UP,
        CHART_COLOR_CHART_DOWN,
        CHART_COLOR_CHART_LINE,
        CHART_COLOR_CANDLE_BULL,
        CHART_COLOR_CANDLE_BEAR,
        CHART_COLOR_BID,
        CHART_COLOR_ASK,
        CHART_COLOR_LAST,
        CHART_COLOR_STOP_LEVEL
    };

    for(int i = 0; i < ArraySize(colors); ++i)
    {
        ChartSetInteger(0, colors[i], RGB_INVERSE(ChartGetInteger(0, colors[i])));
    }
}
```

以下图片将应用脚本前后的图表图像进行了合并。



通过 MQL 程序反转图表颜色

现在让我们完成对 *IndSubChart.mq5* 的编辑。我们需要读取主图表的颜色，并将其应用到的指标图表中。有一个函数可以执行这些操作：*SetPlotColors*，其调用已在 *OnChartEvent* 处理程序中被注释（请参阅 图表显示模式 章节的最后一个示例）。

```
void SetPlotColors(const int index, const ENUM_CHART_MODE m)
{
    if(m == CHART_CANDLES)
    {
        PlotIndexSetInteger(index, PLOT_COLOR_INDEXES, 3);
        PlotIndexSetInteger(index, PLOT_LINE_COLOR, 0, (int)ChartGetInteger(0, CHART_CC
        PlotIndexSetInteger(index, PLOT_LINE_COLOR, 1, (int)ChartGetInteger(0, CHART_CC
        PlotIndexSetInteger(index, PLOT_LINE_COLOR, 2, (int)ChartGetInteger(0, CHART_CC
    }
    else
    {
        PlotIndexSetInteger(index, PLOT_COLOR_INDEXES, 1);
        PlotIndexSetInteger(index, PLOT_LINE_COLOR, (int)ChartGetInteger(0, CHART_COLOR
    }
}
```

在这个新函数中，我们将根据图表的绘制模式，获取上涨和下跌蜡烛图的轮廓及实体颜色，或者线条颜色，并将这些颜色应用到图表中。当然，不要忘记在初始化期间调用此函数。

```
int OnInit()
{
    ...
    mode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE);
    InitPlot(0, InitBuffers(mode), Mode2Style(mode));
    SetPlotColors(0, mode);
    ...
}
```

指标已就绪。尝试在窗口中运行它，并在图表特性对话框中更改颜色。图表会根据新设置自动调整。

5.7.15 鼠标和键盘控制

在本节中，我们将了解一组特性，这些特性会影响图表如何捕获特定鼠标和键盘操作，这些操作在默认情况下被视为控制操作。特别是，MetaTrader 5 用户都很清楚，图表可以通过鼠标滚动，还可以调用上下文菜单来执行最常用的命令。MQL5 允许完全或部分禁用图表的这种行。需要注意的是，这只能通过编程方式实现：终端用户界面中没有类似选项。

需要注意的是，唯一的例外是 CHART_DRAG_TRADE_LEVELS 选项（见下表）：终端设置的 *Charts* 选项卡中提供了一个下拉列表，用于控制是否允许通过鼠标拖动交易水平线。

该组的所有特性均为布尔类型（*true*表示允许，*false*表示禁用），且它们包含在 ENUM_CHART_PROPERTY_INTEGER 枚举中。

标识符	说明
CHART_CONTEXT_MENU	通过按下鼠标右键启用/禁用对上下文菜单的访问。 <i>false</i> 值仅会禁用图表的上下文菜单，而图表上对象的上下文菜单仍可使用。默认值为 <i>true</i> 。
CHART_CROSSHAIR_TOOL	通过按下鼠标中键启用/禁用对十字准线工具的访问。默认值为 <i>true</i> 。
CHART_MOUSE_SCROLL	通过鼠标左键或滚轮滚动图表。当启用滚动时，这不仅适用于水平滚动，还适用于垂直滚动，但垂直滚动仅在设置固定比例时可用：即应用 CHART_SCALEFIX、CHART_SCALEFIX_11 或 CHART_SCALE_PT_PER_BAR 特性之一的情况下。默认值为 <i>true</i> 。
CHART_KEYBOARD_CONTROL	使用键盘管理图表的能力（通过 <i>Home</i> 、 <i>End</i> 、 <i>PageUp</i> / <i>PageDown</i> 、 <i>+/-</i> 、上下箭头等按键）。设置为 <i>false</i> 可禁用图表的滚动和缩放功能，但与此同时，仍可在 <i>OnChartEvent</i> 中接收这些按键的击键事件。默认值为 <i>true</i> 。
CHART_QUICK_NAVIGATION	在图表中启用快速导航栏，当你双击鼠标或按下 <i>Space</i> 或 <i>Input</i> 键时，该导航栏会自动出现在时间轴左侧区域。使用该导航栏，你可以快速更改交易品种、时间范围或第一个可见柱线的日期。默认情况下，该特性设置为 <i>true</i> ，即启用快速导航功能。

标识符	说明
CHART_DRAG_TRADE_LEVELS	允许通过鼠标在图表上拖动交易水平线的权限。默认情况下拖动模式处于启用状态 (<i>true</i>)。

在测试脚本 *ChartInputControl.mq5* 中，我们将为上述所有特性设置监控功能，并额外提供输入变量，供用户设置任意值。我们的脚本会在启动时保存设置的备份副本，因此当脚本结束时，所有已更改的特性都将恢复原状。

```

#property script_show_inputs

#include <MQL5Book/ChartModeMonitor.mqh>

input bool ContextMenu = true; // CHART_CONTEXT_MENU
input bool CrossHairTool = true; // CHART_CROSSHAIR_TOOL
input bool MouseScroll = true; // CHART_MOUSE_SCROLL
input bool KeyboardControl = true; // CHART_KEYBOARD_CONTROL
input bool QuickNavigation = true; // CHART_QUICK_NAVIGATION
input bool DragTradeLevels = true; // CHART_DRAG_TRADE_LEVELS

void OnStart()
{
    const bool Inputs[] =
    {
        ContextMenu, CrossHairTool, MouseScroll,
        KeyboardControl, QuickNavigation, DragTradeLevels
    };
    const int flags[] =
    {
        CHART_CONTEXT_MENU, CHART_CROSSHAIR_TOOL, CHART_MOUSE_SCROLL,
        CHART_KEYBOARD_CONTROL, CHART_QUICK_NAVIGATION, CHART_DRAG_TRADE_LEVELS
    };
    ChartModeMonitor m(flags);
    Print("Initial state:");
    m.print();
    m.backup();

    for(int i = 0; i < ArraySize(flags); ++i)
    {
        ChartSetInteger(0, (ENUM_CHART_PROPERTY_INTEGER)flags[i], Inputs[i]);
    }

    while(!IsStopped())
    {
        m.snapshot();
        Sleep(500);
    }
    m.restore();
}

```

例如，当我们运行脚本时，可以将上下文菜单、十字准线工具、鼠标和键盘控制的权限重置为 *false*。结果将如下日志所示。

```
Initial state:
  [key] [value]
[0]    50      1
[1]    49      1
[2]    42      1
[3]    47      1
[4]    45      1
[5]    43      1
CHART_CONTEXT_MENU 1 -> 0
CHART_CROSSHAIR_TOOL 1 -> 0
CHART_MOUSE_SCROLL 1 -> 0
CHART_KEYBOARD_CONTROL 1 -> 0
```

在这种情况下，你将无法通过鼠标或键盘移动图表，甚至无法调用上下文菜单。因此，为了恢复图表的性能，你需要将在图表上运行相同或另一个脚本（请记住，单个图表只能运行一个脚本，当应用新脚本时，之前的脚本将被卸载）。只需拖放脚本的新实例，但无需运行（在用于提供输入变量的对话框中点击 *Cancel*即可）。

5.7.16 分离图表窗口

终端中的图表窗口可以从主窗口中分离，分离后可移动到桌面任何位置，包括其他显示器。MQL5 允许你查询和更改此设置：相应特性包含在 ENUM_CHART_PROPERTY_INTEGER 枚举中。

标识符	说明	值 类型
CHART_IS_DOCKED	图表窗口已分离（默认值为 true）。如果设置为 false，则图表可拖至终端窗口之外	bool
CHART_FLOAT_LEFT	已分离图表相对于虚拟屏幕的左侧坐标	int
CHART_FLOAT_TOP	已分离图表相对于虚拟屏幕的顶部坐标	int
CHART_FLOAT_RIGHT	已分离图表相对于虚拟屏幕的右侧坐标	int
CHART_FLOAT_BOTTOM	已分离图表相对于虚拟屏幕的底部坐标	int

让我们在 *ChartDock.mq5*脚本中设置对这些特性的追踪。

```

void OnStart()
{
    const int flags[] =
    {
        CHART_IS_DOCKED,
        CHART_FLOAT_LEFT, CHART_FLOAT_TOP, CHART_FLOAT_RIGHT, CHART_FLOAT_BOTTOM
    };
    ChartModeMonitor m(flags);
    ...
}

```

如果你现在运行该脚本，使用上下文菜单分离图表（取消按下 **Docked** 开关命令），并移动或调整图表大小，相应的日志将添加到日志记录中。

```

Initial state:
    [key] [value]
[0]    51      1
[1]    52      0
[2]    53      0
[3]    54      0
[4]    55      0

                                // undocked

CHART_IS_DOCKED 1 -> 0
CHART_FLOAT_LEFT 0 -> 299
CHART_FLOAT_TOP 0 -> 75
CHART_FLOAT_RIGHT 0 -> 1263
CHART_FLOAT_BOTTOM 0 -> 472

                                // changed the vertical size

CHART_FLOAT_BOTTOM 472 -> 500
CHART_FLOAT_BOTTOM 500 -> 539

                                // changed the horizontal size

CHART_FLOAT_RIGHT 1263 -> 1024
CHART_FLOAT_RIGHT 1024 -> 1023

                                // docked back

CHART_IS_DOCKED 0 -> 1

```

学习本节后，对于通过 **ChartGet** 和 **ChartSet** 函数管理的特性的介绍便已完毕，因此我们将使用通用脚本 **ChartFullSet.mq5** 对内容进行总结。它会追踪所有类型的所有特性的状态。标志数组的初始化通过循环填充连续索引来完成。最大值会预留一定余量以应对新的特性，而额外的无效编号会被 **ChartModeMonitorBase** 类内置的检查机制（参阅 **detect** 方法）自动丢弃。

激活脚本后，尝试更改任何设置，同时观察日志中的程序消息。

5.7.17 获取 MQL 程序在图表上的放置坐标

用户经常使用鼠标将 MQL 程序拖放到图表上。除操作便捷之外，这还可以为算法提供一定的上下文。例如，指标可以应用到不同的子窗口，或者脚本可以在用户在图表上指定的价格点挂单。下一组函数用于获取程序被拖放至的点的坐标。

int ChartWindowOnDropped()

该函数返回当前 EA 交易、脚本或指标通过鼠标拖放操作所放置的图表子窗口编号。我们知道，主窗口编号为 0，子窗口编号则从 1 开始依次递增。子窗口编号不取决于其上方是否存在

隐藏子窗口，即使子窗口被隐藏，其原始索引仍会保留。换言之，当存在 `隐藏子窗口` 时，可见子窗口编号与其实际索引可能不一致。

```
double ChartPriceOnDropped()
datetime ChartTimeOnDropped()
```

此函数对返回程序放置点的坐标，以价格和时间为单位。请注意，子窗口中显示的可以是任意数据，而不仅限于价格，尽管函数名称 *ChartPriceOnDropped* 包含 'Price'。

注意！目标点的时间不会按图表时间范围的大小进行取整，这意味着，即使在 H1 或 D1 图表上，仍可获取到精确到分钟甚至秒的值。

```
int ChartXOnDropped()
int ChartYOnDropped()
```

这两个函数返回某点的 X 和 Y 屏幕坐标（以像素为单位）。坐标原点位于图表主窗口的左上角。我们在 `屏幕规格` 章节讨论了坐标轴的方向。

Y 坐标始终从主图表的左上角开始计算，即使放置点属于某个子窗口。要将此值转换为相对于子窗口的坐标 *y*，请使用特性 `CHART_WINDOW_YDISTANCE`（请参阅示例）。

在 *ChartDrop.mq5* 脚本中，我们将所有上述函数的值输出到日志中。

```
void OnStart()
{
    const int w = PRTF(ChartWindowOnDropped());
    PRTF(ChartTimeOnDropped());
    PRTF(ChartPriceOnDropped());
    PRTF(ChartXOnDropped());
    PRTF(ChartYOnDropped());

    // for the subwindow, recalculate the y coordinate to the local one
    if(w > 0)
    {
        const int y = (int)PRTF(ChartGetInteger(0, CHART_WINDOW_YDISTANCE, w));
        PRTF(ChartYOnDropped() - y);
    }
}
```

例如，如果我们将此脚本拖放到运行 W PR 指标的第一个子窗口中，将得到以下结果。

```
ChartWindowOnDropped()=1 / ok
ChartTimeOnDropped()=2021.11.30 03:52:30 / ok
ChartPriceOnDropped()=-50.0 / ok
ChartXOnDropped()=217 / ok
ChartYOnDropped()=312 / ok
ChartGetInteger(0,CHART_WINDOW_YDISTANCE,w)=282 / ok
ChartYOnDropped()-y=30 / ok
```

尽管脚本被拖放到 EURUSD, H1 图表上，但我们仍获取了精确到分钟和秒的时间戳。

请注意，“price”值为 -50，因为 W PR 值范围是 [0,-100]。

此外，点 312 的垂直坐标（相对于整个图表窗口）被转换为子窗口的本地坐标：由于主图表起点到子窗口的垂直距离为 282，因此子窗口内的 *y* 值为 30。

5.7.18 屏幕坐标与时间/价格的相互转换

由于图表工作区域的测量原理不同，因此需要在不同的测量单位之间进行重新计算。为此提供了两个函数。

`bool ChartTimePriceToXY(long chartId, int window, datetime time, double price, int &x, int &y)`

`bool ChartXYToTimePrice(long chartId, int x, int y, int &window, datetime &time, double &price)`

`ChartTimePriceToXY`函数将图表坐标从时间/价格表示 (*time/price*) 转换为以像素为单位的 *X* 和 *Y* 坐标 (*x/y*)。 `ChartXYToTimePrice`函数执行相反的操作：将 *X* 和 *Y* 坐标转换为时间和价格值。

这两个函数都需要在第一个参数 *chartId* 中指定图表 ID。此外，在 `ChartTimePriceToXY` 中还需传入 *window* 子窗口编号 (该编号应在窗口数量范围内)。如果存在多个子窗口，每个子窗口都有各自的时间序列和沿垂直轴的刻度 (根据参数条件，*price* 参数称为“price”)。

在 `ChartXYToTimePrice` 函数中，*window* 参数是输出参数。该函数会将此参数与 *time*、*price* 一同填充。这是因为像素坐标是整个屏幕通用的，其原点 *x/y* 可能位于任意子窗口中。



时间、价格与屏幕坐标

如果成功完成，函数返回 *true*。

请注意，在两种坐标系中，对应报价或屏幕坐标的可见矩形区域均有范围限制。因此，在特定初始数据下，可能会出现所获取的时间、价格或像素超出可见区域的情况。特别是，也可能获得负值。我们将在 [图表事件](#) 一章中了解交互式重新计算示例。

在上一节中，我们了解了如何确定 MQL 程序的启动位置。尽管在物理上只有一个最终放置点，但其在报价坐标和屏幕坐标中的表示通常存在计算误差。两个新函数用于像素和价格/时间的双向转换，它们将帮助我们验证这一点。

修改后的脚本名为 *ChartXY.mq5*。它大致可以分为 3 个阶段。在第一阶段，我们将像之前一样导出放置点的坐标。

```
void OnStart()
{
    const int w1 = PRTF(ChartWindowOnDropped());
    const datetime t1 = PRTF(ChartTimeOnDropped());
    const double p1 = PRTF(ChartPriceOnDropped());
    const int x1 = PRTF(ChartXOnDropped());
    const int y1 = PRTF(ChartYOnDropped());
    ...
}
```

在第二阶段，我们尝试在时间 (*t2*) 和价格 (*p2*) 期间转换屏幕坐标 *x1* 和 *y1*，并将其与从上述 *OnDropped* 函数中获得的坐标进行比较。

```
int w2;
datetime t2;
double p2;
PRTF(ChartXYToTimePrice(0, x1, y1, w2, t2, p2));
Print(w2, " ", p2, " ", t2);
PRTF(w1 == w2 && t1 == t2 && p1 == p2);
...
```

然后我们进行逆变换：使用获得的报价坐标 *t1* 和 *p1* 来计算屏幕坐标 *x2* 和 *y2*，并将其与原始值 *x1* 和 *y1* 进行比较。

```
int x2, y2;
PRTF(ChartTimePriceToXY(0, w1, t1, p1, x2, y2));
Print(x2, " ", y2);
PRTF(x1 == x2 && y1 == y2);
...
```

正如我们将在后续示例日志中看到的，上述所有检查都将失败（值之间会存在细微差异）。所以，我们需要第三步。

我们重新计算变量名中带有后缀 2 的屏幕坐标和报价坐标，并将它们保存到带有新后缀 3 的变量中。然后，我们将第一阶段和第三阶段的所有值相互进行比较。

```
int w3;
datetime t3;
double p3;
PRTF(ChartXYToTimePrice(0, x2, y2, w3, t3, p3));
Print(w3, " ", p3, " ", t3);
PRTF(w1 == w3 && t1 == t3 && p1 == p3);

int x3, y3;
PRTF(ChartTimePriceToXY(0, w2, t2, p2, x3, y3));
Print(x3, " ", y3);
PRTF(x1 == x3 && y1 == y3);
}
```

让我们在 XAUUSD, H1 图表上运行该脚本。以下是原始点数据。

```

ChartWindowOnDropped()=0 / ok
ChartTimeOnDropped()=2021.11.22 18:00:00 / ok
ChartPriceOnDropped()=1797.7 / ok
ChartXOnDropped()=234 / ok
ChartYOnDropped()=280 / ok

```

将像素转换为报价将得出以下结果。

```

ChartXYToTimePrice(0,x1,y1,w2,t2,p2)=true / ok
0 1797.16 2021.11.22 18:30:00
w1==w2&&t1==t2&&p1==p2=false / ok

```

在时间和价格上均存在差异。反向计算在精度方面也并非完美。

```

ChartTimePriceToXY(0,w1,t1,p1,x2,y2)=true / ok
232 278
x1==x2&&y1==y2=false / ok

```

精度损失是由于坐标轴上的值按照测量单位 (尤其是像素和点) 进行量化所导致的。

最后, 最后一步证明上述误差并非函数本身的问题, 因为循环重新计算会得到原始结果。

```

ChartXYToTimePrice(0,x2,y2,w3,t3,p3)=true / ok
0 1797.7 2021.11.22 18:00:00
w1==w3&&t1==t3&&p1==p3=true / ok
ChartTimePriceToXY(0,w2,t2,p2,x3,y3)=true / ok
234 280
x1==x3&&y1==y3=true / ok

```

在伪代码中, 这可以通过以下等式表示:

```

ChartTimePriceToXY(ChartXYToTimePrice(XY)) = XY
ChartXYToTimePrice(ChartTimePriceToXY(TP)) = TP

```

将 *ChartTimePriceToXY* 函数应用于 *ChartXYToTimePrice* 的计算结果时, 将得到原始坐标。反向转换同样如此: 将 *ChartXYToTimePrice* 函数应用于 *ChartTimePriceToXY* 的计算结果时, 将完全匹配。

因此, 如果对使用重新计算函数的算法有更高的精度要求, 应谨慎考量这些算法的实现方式。

另一个使用 *ChartWindowOnDropped* 的示例将在脚本 *ChartIndicatorMove.mq5* 中提供 (见 [管理图表上的指标](#) 章节)。

5.7.19 沿时间轴滚动图表

MetaTrader 5 用户都很熟悉快速图表导航面板, 可通过双击时间轴左上方或按下 *Space* 或 *Input* 键来打开该面板。通过 *ChartNavigate* 函数同样能以编程实现类似功能。

bool ChartNavigate(long chartId, ENUM_CHART_POSITION position, int shift = 0)

该函数将 *chartId* 图表相对于 *position* 参数指定的预定义图表位置, 按指定的柱线数量进行移动。它属于 *ENUM_CHART_POSITION* 枚举类型, 包含以下元素:

标识符	说明
CHART_BEGIN	图表起始位置 (最早的价格)
CHART_CURRENT_POS	当前位置
CHART_END	图表结束位置 (最新价格)

*shift*参数用于设置图表应移动的柱线数量。正数会将图表向右移动 (向结束位置), 负数会将图表向左移动 (向起始位置)。

如果成功, 该函数返回 *true*, 如果发生错误, 则返回 *false*。

为了测试该函数, 我们来创建一个简单脚本 *ChartNavigate.mq5*。通过输入变量, 用户可以选择起始点和偏移量 (以柱线为单位)。

```
#property script_show_inputs

input ENUM_CHART_POSITION Position = CHART_CURRENT_POS;
input int Shift = 0;

void OnStart()
{
    ChartSetInteger(0, CHART_AUTOSCROLL, false);
    const int start = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);
    ChartNavigate(0, Position, Shift);
    const int stop = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);
    Print("Moved by: ", stop - start, ", from ", start, " to ", stop);
}
```

日志会显示移动前后第一个可见柱线的编号。

一个更实用的示例是脚本 *ChartSynchro.mq5*, 该脚本允许在手动滚动其中一个图表时, 同步滚动所有运行该脚本的图表。因此, 你可以同步同一金融工具不同时间范围的窗口, 或分析不同金融工具的平行价格走势。

```

void OnStart()
{
    datetime bar = 0; // current position (time of the first visible bar)

    conststring namePosition = __FILE__; // global variable name

    ChartSetInteger(0, CHART_AUTOSCROLL, false); // disable autoscroll

    while(!IsStopped())
    {
        const bool active = ChartGetInteger(0, CHART_BRING_TO_TOP);
        const int move = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);

        // the active chart is the leader, and the rest are slaves
        if(active)
        {
            const datetime first = iTime(_Symbol, _Period, move);
            if(first != bar)
            {
                // if the position has changed, save it in a global variable
                bar = first;
                GlobalVariableSet(namePosition, bar);
                Comment("Chart ", ChartID(), " scrolled to ", bar);
            }
        }
        else
        {
            const datetime b = (datetime)GlobalVariableGet(namePosition);

            if(b != bar)
            {
                // if the value of the global variable has changed, adjust the position
                bar = b;
                const int difference = move - iBarShift(_Symbol, _Period, bar);
                ChartNavigate(0, CHART_CURRENT_POS, difference);
                Comment("Chart ", ChartID(), " forced to ", bar);
            }
        }

        Sleep(250);
    }
    Comment("");
}

```

通过第一个可见柱线的日期和时间 (CHART_FIRST_VISIBLE_BAR) 实现对齐。该脚本通过循环检测此值, 如果在活动图表上运行, 则将其写入一个全局变量。其他图表上的脚本读取此变量, 并通过 *ChartNavigate* 相应地调整其位置。参数指定图表的相对移动 (CHART_CURRENT_POS), 而移动的柱线数量则定义为当前第一个可见柱线的编号与从全局变量中读取的编号之间的差值。

下图展示了 EURUSD 的 H1 和 M15 图表的同步结果。



用于同步图表位置的脚本示例

在熟悉系统 [图表事件](#) 后，我们将会将此脚本转换为指标，并移除无限循环。

5.7.20 图表重绘请求

在多数情况下，图表会自动响应数据和终端设置的变化，并相应地更新窗口显示内容（如价格图表、指标图表等）。但 MQL 程序的功能非常丰富，可执行任意操作，因此难以判定确定是否需要重绘。此外，分析该账户上每个 MQL 程序的操作可能消耗大量资源，导致终端整体性能下降。因此，MQL5 API 提供了 **ChartRedraw** 函数，借助该函数，MQL 程序可以在必要时自行请求重绘图表。

void ChartRedraw(long chartId = 0)

该函数会强制重绘具有指定标识符的图表（默认值 0 表示当前图表）。通常，该函数在程序修改图表特性或其上放置的 [对象](#) 后调用。

在 [图表显示模式](#) 一节的 *IndSubChart.mq5* 指标中，我们已经见过 **ChartRedraw** 的应用示例。在 [打开和关闭图表](#) 一节中还将提供另一个示例。

该函数仅影响图表的重绘，不会重新计算含报价和指标的时间序列。图表更新（实际上为重建）的最后一种方法更为强硬，是通过 **ChartSetSymbolPeriod** 函数执行（见下一节）。

5.7.21 切换交易品种和时间范围

有时，MQL 程序需要切换图表的当前交易品种或时间范围。尤其是对于许多币种、多时间范围的交易面板或交易历史分析工具，这是一项常见功能。为此，MQL5 API 提供了 **ChartSetSymbolPeriod** 函数。

你还可以使用此函数启动整个图表的重新计算，包括图标上的指标。只需将当前交易品种和时间范围指定为参数即可。该技术适用于以下指标：在首次调用 *OnCalculate* 时未能完全计算、需要加载等待第三方数据（其他交易品种、分时报价或指标）的指标。此外，更改交易品种/时间范围会导致图标上加载 EA 交易重新初始化。在执行此操作期间，脚本（如果在循环中周期性执行）将从图表中完全消失（该脚本会从旧的交易品种/时间范围组合中卸载，但不会自动加载到新的组合中）。

bool ChartSetSymbolPeriod(long chartId, string symbol, ENUM_TIMEFRAMES timeframe)

该函数将标识符为 *chartId* 的指定图表的交易品种和时间范围更改为相应参数的值：*symbol* 和 *timeframe*。在 *chartId* 参数中，0 表示当前图表，在 *symbol* 参数中，NULL 表示当前交易品种，在 *timeframe* 中，0 表示当前时间周期。

该更改异步生效，即该函数仅向终端发送命令，而不会等待其执行完成。该命令将被添加到图表的消息队列中，且仅在所有前置命令处理完成后才会执行。

如果命令成功加入图表队列，函数将返回 *true*，如果出现问题，则返回 *false*。可通过 *_LastError* 中获取具体错误信息。

我们已经看到了使用该函数更新多个指标的示例，特别是：

- *IndDeltaVolume.mq5*（请参阅 [等待数据和管理可见性](#)）
- *IndUnityPercent.mq5*（请参阅 [多货币和多时间范围指标](#)）
- *UseWPRMTF.mq5*（请参阅 [支持多个交易品种和时间范围](#)）
- *UseM1MA.mq5*（请参阅 [使用内置指标](#)）
- *UseDemoAllLoop.mq5*（请参阅 [删除指标实例](#)）
- *IndSubChart.mq5*（请参阅 [图表显示模式](#)）

5.7.22 管理图表上的指标

正如我们已经了解到的，图表是指标的执行和可视化环境。它们之间的紧密联系在一组内置函数中得到了进一步验证，这些函数为图表上的指标控制提供了支持。在前一章中，我们已经完成了 [对这些功能的概述](#)。在熟悉了图表之后，我们已经准备好详细探讨这些功能了。

所有函数的统一性体现在前两个参数是统一的：即图表标识符（*chartId*）和窗口编号（*window*）。参数的零值分别表示当前图表和主窗口。

int ChartIndicatorsTotal(long chartId, int window)

该函数返回附加到指定图表窗口的所有指标的数量。它可用于枚举（逐一列出）附加到给定图表的所有指标。可以使用 *ChartGetInteger* 函数并通过特性 *CHART_WINDOWS_TOTAL* 来获取图表的所有窗口总数。

string ChartIndicatorName(long chartId, int window, int index)

该函数通过指定图表窗口中指标列表的 *index*，返回对应指标的短名称。短名称是通过 *IndicatorSetString* 函数在 *INDICATOR_SHORTNAME* 特性中指定的名称（如果未设置，则默认等于指标文件的名称）。

int ChartIndicatorGet(long chartId, int window, const string shortname)

它会返回特定图表窗口中具有指定短名称的指标的句柄。我们可以说，在 *ChartIndicatorGet* 函数中，指标的识别正是通过短名称进行的，因此建议将其构造为包含所有输入参数值的形式。如果由于某种原因而无法通过短名称来唯一标识指标实例，还有另一种方法可以通过指标的参

数列表来识别指标实例，该参数列表可以通过给定的描述符使用 *IndicatorParameters* 函数获取。

从 *ChartIndicatorGet* 函数获取句柄会增加该指标使用的内部计数器。终端执行系统会将所有计数器值大于零的指标保持加载状态。因此，不再需要的指标必须通过调用 *IndicatorRelease* 显式释放。否则，该指标将保持闲置状态并消耗资源。

bool ChartIndicatorAdd(long chartId, int window, int handle)

该函数会将最后一个参数中传递的描述符所指定的指标添加到指定的图表窗口中。指标和图表必须具有相同的交易品种和时间周期组合。否则，将出现错误 `ERR_CHART_INDICATOR_CANNOT_ADD (4114)`。

要将指标添加到新窗口，*window* 参数必须比最后一个现有窗口的索引大 1，即等于通过 *ChartGetInteger* 调用获取的 `CHART_WINDOWS_TOTAL` 特性值。如果参数值超过 *ChartGetInteger(ID, CHART_WINDOWS_TOTAL)* 的返回值，将不会创建新窗口和指标。

如果将一个本应绘制在独立子窗口中的指标（例如内置的 `IMACD` 或通过 *#property indicator_separate_window* 指定特性的自定义指标）添加到主图表窗口，则该指标可能看似不可见，尽管它会存在于指标列表中。这通常意味着该指标的值未落入价格图表的显示范围内。这种“不可见”指标的数值可以在“*Data window*”中观察到，并可通过其他 MQL 程序中的函数读取。

将指标添加到图表中时，由于其与图表的绑定关系，会增加其使用的内部计数器。如果 MQL 程序持有指标的描述符且不再需要它，那么值得通过调用 *IndicatorRelease* 来删除它。这实际上会减少计数器，但指标仍会保留在图表上。

bool ChartIndicatorDelete(long chartId, int window, const string shortname)

这个函数会从标识符为 *chartId* 的图表中，移除编号为 *window* 的窗口中具有指定短名称的指标。如果在指定的图表子窗口中有多个具有相同短名称的指标，将按顺序删除第一个指标。

如果同一图表上的其他指标是使用已移除指标的值计算得出的，那么这些其他指标也会被移除。

从图表中删除一个指标并不意味着其计算部分也会从终端内存中删除——前提是该指标的描述符仍保留在 MQL 程序中。若要释放指标句柄，可使用 *IndicatorRelease* 函数。

ChartWindowFind 函数用于返回指标所在子窗口的编号。有两种形式可用于搜索指标：一种是在当前指标所在的图表上搜索该指标，另一种是在标识符为 *chartId* 的任意图表上搜索具有给定短名称的指标。

int ChartWindowFind()

int ChartWindowFind(long chartId, string shortname)

第二种形式可用于脚本和 EA 交易。

作为展示这些函数的第一个示例，让我们来看一下完整版本的脚本 *ChartList.mq5*。在前面的章节中，我们逐步创建并完善了它，直至 *获取窗口/子窗口的数量和可见性* 这一章节。与之前展示的 *ChartList4.mq5* 相比，我们将添加输入变量，以便能够仅列出运行 MQL 程序的图表，并禁止显示隐藏窗口。

```
input bool IncludeEmptyCharts = true;
input bool IncludeHiddenWindows = true;
```

*IncludeEmptyCharts*参数的默认值为 *true*，表示在列表中包含所有图表，包括空图表。
*IncludeHiddenWindows*参数用于设置默认是否显示隐藏窗口。这些设置与之前的脚本逻辑 *ChartListN*相对应。

为了计算指标总数和子窗口中的指标数量，我们定义了 *indicators*和 *subs* 变量。

```
void ChartList()
{
    ...
    int indicators = 0, subs = 0;
    ...
}
```

在当前图表的窗口循环处理部分，代码发生了重大变化。

```
void ChartList()
{
    ...
    for(int i = 0; i < win; i++)
    {
        const bool visible = ChartGetInteger(id, CHART_WINDOW_IS_VISIBLE, i);
        if(!visible && !IncludeHiddenWindows) continue;
        if(!visible)
        {
            Print(" ", i, "/Hidden");
        }
        const int n = ChartIndicatorsTotal(id, i);
        for(int k = 0; k < n; k++)
        {
            if(temp == 0)
            {
                Print(header);
            }
            Print(" ", i, "/", k, " [I] ", ChartIndicatorName(id, i, k));
            indicators++;
            if(i > 0) subs++;
            temp++;
        }
    }
    ...
}
```

其中新增了 *ChartIndicatorsTotal*和 *ChartIndicatorName* 调用。现在，列表将提及所有类型的 MQL 程序：[E] -EA 交易、[S] -脚本、[I] -指标

以下是脚本在默认设置下生成的日志条目示例。

```

Chart List
N, ID, Symbol, TF, #subwindows, *active, Windows handle
0 132358585987782873 EURUSD M15 #1 133538
  1/0 [I] ATR(11)
1 132360375330772909 EURUSD D1 133514
2 132544239145024745 EURUSD M15 * 395646
  [S] ChartList
3 132544239145024732 USDRUB D1 395688
4 132544239145024744 EURUSD H1 #2 active 2361730
  1/0 [I] %R(14)
  2/Hidden
  2/0 [I] Momentum(15)
5 132544239145024746 EURUSD H1 133584
Total chart number: 6, with MQL-programs: 3
Experts: 0, Scripts: 1, Indicators: 3 (main: 0 / sub: 3)

```

如果将两个输入参数都设置为 *false*, 我们将得到一个精简的列表。

```

Chart List
N, ID, Symbol, TF, #subwindows, *active, Windows handle
0 132358585987782873 EURUSD M15 #1 133538
  1/0 [I] ATR(11)
2 132544239145024745 EURUSD M15 * active 395646
  [S] ChartList
4 132544239145024744 EURUSD H1 #2 2361730
  1/0 [I] %R(14)
Total chart number: 6, with MQL-programs: 3
Experts: 0, Scripts: 1, Indicators: 2 (main: 0 / sub: 2)

```

作为第二个示例, 我们来看一个有趣的脚本 *ChartIndicatorMove.mq5*。

当在图表上运行多个指标时, 我们可能经常需要更改指标的顺序。MetaTrader 5 没有为此提供内置工具, 这迫使你删除某些指标并重新添加它们, 而在此过程中保存和恢复设置就显得尤为重要。*ChartIndicatorMove.mq5*脚本提供了一种自动化此过程的选项。需要注意的是, 该脚本只是转移指标: 如果要更改子窗口的顺序 (包括其中的图形对象), 则应使用 [tpl](#)模板。

*ChartIndicatorMove.mq5*的工作原理如下。当脚本应用于图表时, 它会确定该脚本被添加到了哪个窗口/子窗口, 然后开始向用户列出在那里找到的指标, 并请求用户确认转移操作。用户可以同意, 或者继续列举。

移动方向 (向上或向下) 通过 *MoveDirection* 输入变量设置。DIRECTION 枚举将对其进行描述。

```

#property script_show_inputs

enum DIRECTION
{
    Up = -1,
    Down = +1,
};

input DIRECTION MoveDirection = Up;

```

我们还引入了 *jumpover* 输入变量, 可以将指标转移到下一个子窗口 (而非相邻子窗口), 即实际交换子窗口中指标的位置 (该需求较为常见)。

```
input bool JumpOver = true;
```

通过 *ChartWindowOnDropped* 获取的目标窗口的指标，循环在 *OnStart* 中开始执行。

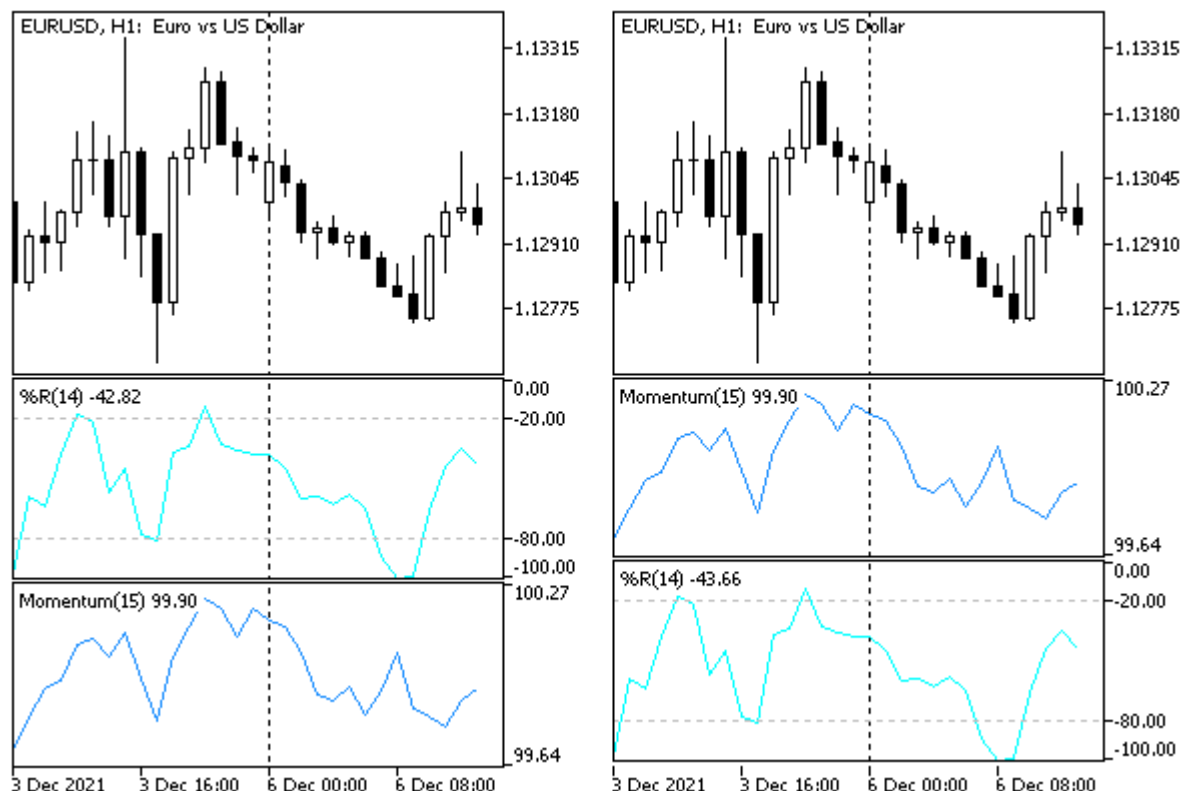
```
void OnStart()
{
    const int w = ChartWindowOnDropped();
    if(w == 0 && MoveDirection == Up)
    {
        Alert("Can't move up from window at index 0");
        return;
    }
    const int n = ChartIndicatorsTotal(0, w);
    for(int i = 0; i < n; ++i)
    {
        ...
    }
}
```

在循环内部，我们会执行以下操作：定义下一个指标的名称，向用户显示消息，并通过以下一系列操作将指标从一个窗口移动到另一个窗口：

- 通过调用以下函数获取句柄：*ChartIndicatorGet*。
- 根据所选方向，通过 *ChartIndicatorAdd* 将其添加至当前窗口的上方或下方窗口；当向下移动时，可自动创建新的子窗口。
- 通过以下函数从先前的窗口中移除指标：*ChartIndicatorDelete*。
- 释放描述符，因为程序中不再需要它。

```
...
const string name = ChartIndicatorName(0, w, i);
const string caption = EnumToString(MoveDirection);
const int button = MessageBox("Move '" + name + "' " + caption + "?",
    caption, MB_YESNOCANCEL);
if(button == IDCANCEL) break;
if(button == IDYES)
{
    const int h = ChartIndicatorGet(0, w, name);
    ChartIndicatorAdd(0, w + MoveDirection, h);
    ChartIndicatorDelete(0, w, name);
    IndicatorRelease(h);
    break;
}
...
```

下图展示了交换包含指标 *WPR* 和 *Momentum* 的子窗口后的结果。该脚本通过将其拖放到包含 *WPR* 指标的顶部子窗口进行启动，移动方向选择为向下 (*Down*)，且默认启用了跳转 (*JumpOver*) 功能。



在子窗口中交换指标

请注意，如果将指标从子窗口移至主窗口，由于其数值超出显示的价格范围，指标图表很可能不显示。如果误操作导致发生这种情况，你可以使用该脚本将指标移回子窗口。

5.7.23 打开和关闭图表

MQL 程序不仅可以分析图表列表，还可以对其进行修改：打开新图表或关闭现有图表。针对这些目的，分配了两个函数：*ChartOpen*和 *ChartClose*。

long ChartOpen(const string symbol, ENUM_TIMEFRAMES timeframe)

该函数会打开一个含指定交易品种和时间范围的新图表，并返回新图表的 ID。如果执行期间发生错误，结果为 0，并且可以在内置变量 *_LastError*中读取错误代码。

如果 *symbol*参数为 NULL，则表示当前图表（正在执行 MQL 程序的图表）的交易品种。*timeframe*参数中的 0 值对应 PERIOD_CURRENT。

终端中同时打开的图表最大数量不能超过 CHARTS_MAX (100 个)。

在下一节中，在学完用于处理 tp1模板的函数后，我们将看到一个使用 *ChartOpen*函数的示例。

请注意，终端不仅允许创建带有图表的完整窗口，还允许创建 **图表对象**。它们和其他图形对象（如趋势线、通道、价格标签等）一样，放置在普通图表内部。图表对象允许在一个标准图表中同时显示多个不同交易品种和时间范围的价格序列小片段。

bool ChartClose(long chartId = 0)

该函数用于关闭指定 ID 的图表（默认值 0 表示当前图表）。该函数返回一个成功状态指示值。

作为示例，我们将实现 *ChartCloseIdle.mq5*脚本，该脚本将关闭那些包含重复交易品种和时间周期组合但未包含 MQL 程序和图形对象的重复图表。

首先，我们需要创建一个列表来统计每个交易品种/时间范围对的图表数量。该任务由 *ChartIdleList* 函数实现，它与我们在 *ChartList.mq5* 脚本中看到的非常相似。该列表本身是在 *MapArray<string,int> chartCounts* 映射数组中形成的。

```
#include <MQL5Book/Periods.mqh>
#include <MQL5Book/MapArray.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
    MapArray<string,int> chartCounts;
    ulong duplicateChartIDs[];
    // collect duplicate empty charts
    if(ChartIdleList(chartCounts, duplicateChartIDs))
    {
        ...
    }
    else
    {
        Print("No idle charts.");
    }
}
```

与此同时，*ChartIdleList* 函数会将符合关闭条件的空闲图表标识符填充到 *duplicateChartIDs* 数组中。

```

int ChartIdleList(MapArray<string,int> &map, ulong &duplicateChartIDs[])
{
    // list charts until their list ends
    for(long id = ChartFirst(); id != -1; id = ChartNext(id))
    {
        // skip objects
        if(ChartGetInteger(id, CHART_IS_OBJECT)) continue;

        // getting the main properties of the chart
        const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
        const string expert = ChartGetString(id, CHART_EXPERT_NAME);
        const string script = ChartGetString(id, CHART_SCRIPT_NAME);
        const int objectCount = ObjectsTotal(id);

        // count the number of indicators
        int indicators = 0;
        for(int i = 0; i < win; ++i)
        {
            indicators += ChartIndicatorsTotal(id, i);
        }

        const string key = ChartSymbol(id) + "/" + PeriodToString(ChartPeriod(id));

        if(map[key] == 0 // the first time we always read a new symbol/TF combinati
            // otherwise, only empty charts are counted:
            || (indicators == 0 // no indicators
                && StringLen(expert) == 0 // no Expert Advisor
                && StringLen(script) == 0 // no script
                && objectCount == 0)) // no objects
        {
            const int i = map.inc(key);
            if(map[i] > 1) // duplicate
            {
                PUSH(duplicateChartIDs, id);
            }
        }
    }
    return map.getSize();
}

```

当待删除列表形成后，我们在 *OnStart* 中循环调用 *ChartClose* 函数以遍历列表。

```

void OnStart()
{
    ...
    if(CharIdleList(chartCounts, duplicateChartIDs))
    {
        for(int i = 0; i < ArraySize(duplicateChartIDs); ++i)
        {
            const ulong id = duplicateChartIDs[i];
            // request to bring the chart to the front
            ChartSetInteger(id, CHART_BRING_TO_TOP, true);
            // update the state of the windows, pumping the queue with the request
            ChartRedraw(id);
            // ask user for confirmation
            const int button = MessageBox(
                "Remove idle chart: "
                + ChartSymbol(id) + "/" + PeriodToString(ChartPeriod(id)) + "?",
                __FILE__, MB_YESNOCANCEL);
            if(button == IDCANCEL) break;
            if(button == IDYES)
            {
                ChartClose(id);
            }
        }
    }
    ...
}

```

对于每个图表，将先调用 `ChartSetInteger(id, CHART_BRING_TO_TOP, true)` 函数，向用户显示应关闭的窗口。由于该函数是异步的（仅将激活窗口命令加入事件队列），因此需要额外调用 `ChartRedraw` 以处理所有累积的消息。随后会提示用户确认该操作。只有在用户点击 **Yes** 后，图表才会关闭。若用户选择 **No** 时，将跳过当前图表（保持其开启状态），并继续处理后续图表。若点击 **Cancel**，可以提前中断整个关闭循环。

5.7.24 处理 tpl 图表模板

MQL5 API 提供了两个用于处理模板的函数。模板是扩展名为 `tpl` 的文件，用于保存图表的所有内容，即图表的全部设置，以及已绘制的图形、指标和 EA（如有）。

bool ChartSaveTemplate(long chartId, const string filename)

该函数将当前图表的设置保存为指定名称的 `.tpl` 模板文件。

图表由 `chartId` 参数指定，其中 0 表示当前活动图表。

保存模板的文件名 (`filename`) 可以不指定“`tpl`”扩展名：系统会自动添加该扩展名。默认情况下，模板会被保存到 `terminal_dir/Profiles/Templates/` 文件夹中，随后可以在终端中手动应用这些模板。不过，除了指定文件名外，还可以指定相对于 MQL5 目录的路径，尤其是以“`Files/`”开头的路径。因此，我们可以使用 `文件` 操作函数打开保存的模板，对其进行分析，并在必要时对它们进行编辑（请参阅后续的 `ChartTemplate.mq5` 示例）。

如果指定路径中已存在同名文件，其内容将被覆盖。

稍后，我们将看到一个保存和应用模板的综合示例。

bool ChartApplyTemplate(long chartId, const string filename)

该函数将指定文件中的模板应用到 **chartId** 对应的图表上。

将按照以下规则搜索模板文件：

- ① 如果 **filename** 包含路径（以反斜杠 “\” 或正斜杠 “/” 开头），则模板将相对于目录进行匹配 **terminal_data_directory/MQL5**。
- ① 如果文件名中不包含路径，则将在调用该函数的 EX5 文件可执行文件所在的同一位置搜索模板。
- ① 如果在前两个位置未找到模板，则会在标准模板文件夹 **terminal_dir/Profiles/Templates/** 中搜索模板。

请注意，**terminal_data_directory** 指的是存储修改后文件的文件夹，其位置可能因操作系统类型、用户名和计算机安全设置而异。通常，它与 **terminal_dir** 文件夹不同，尽管在某些情况下（例如使用管理员组账户操作时），它们可能会相同。可以使用 **TerminalInfoString** 函数查找 **terminal_data_directory** 和 **terminal_directory** 文件夹的位置（请分别参阅 **TERMINAL_DATA_PATH** 和 **TERMINAL_PATH** 常量）。

ChartApplyTemplate 调用将创建一个命令，该命令会被添加到图表的消息队列中，且仅在所有先前命令处理完成后才会执行。

加载模板会停止图表上运行的所有 MQL 程序，包括启动加载操作的程序。如果模板中包含指标和 EA 交易，将启动它们的新实例。

出于安全考虑，将包含 EA 交易的模板应用于图表时，可以限制 **交易权限**。如果调用 **ChartApplyTemplate** 函数的 MQL 程序没有交易权限，则通过模板加载的 EA 交易也将无交易权限，此限制不受模板设置影响。如果调用 **ChartApplyTemplate** 的 MQL 程序被允许交易，但模板设置中禁止交易，则加载的 EA 交易仍无法进行交易。

脚本 **ChartDuplicate.mq5** 的一个示例允许创建当前图表的副本。

```
void OnStart()
{
    const string temp = "/Files/ChartTemp";
    if(ChartSaveTemplate(0, temp))
    {
        const long id = ChartOpen(NULL, 0);
        if(!ChartApplyTemplate(id, temp))
        {
            Print("Apply Error: ", _LastError);
        }
    }
    else
    {
        Print("Save Error: ", _LastError);
    }
}
```

首先，使用 **ChartSaveTemplate** 创建一个临时 **tpl** 模板文件，然后打开新图表（调用 **ChartOpen**），最后通过 **ChartApplyTemplate** 函数将该模板应用于新图表。

但在许多情况下，程序员面临更具挑战性的任务：不仅仅是应用模板，还要预先编辑模板内容。

通过使用模板，你可以更改许多无法通过其他 MQL5 API 函数修改的图表特性。例如，在不同时间范围下指标的可见性、指标子窗口的排列顺序以及应用于这些子窗口的对象。

tpl 文件格式与终端在会话之间存储图表时使用的 .chr 文件格式完全相同（位于 *terminal_directory/Profiles/Charts/profile_name* 文件夹中）。

tpl 文件是具有特定语法的文本文件。其中的特性可以是单行书写的 `key=value` 键值对，也可以是包含多个 `key=value` 键值对特性的某种分组结构。这类分组在下文中将统称为“容器”，因为除了包含独立特性外，它们还可以包含其他嵌套的容器。

容器以类似“`<tag>`”的行开头，其中 **tag** 是预定义容器类型之一，并以配套的类似“`</tag>`”行结束（标签名称必须匹配）。换言之，这种格式在某种程度上类似于 XML（但没有头文件），其中所有词法单元必须单独成行，且标签特性并非通过其特性（如 XML 中在开始部分 `<tag attribute1=value1...>`）来表示，而是通过标签内部文本来表示。

支持的标签列表：

- ① `chart` - 根容器，包含主图表特性及所有子容器；
- ① `expert` - 包含 EA 交易通用特性的容器，例如交易权限（位于 `chart` 内）；
- ① `window` - 包含窗口/子窗口特性的容器及其从属容器（位于 `chart` 内）；
- ① `object` - 包含图形对象特性的容器（位于 `window` 内）；
- ① `indicator` - 包含指标特性的容器（位于 `window` 内）；
- ① `graph` - 包含指标图表特性的容器（位于 `indicator` 内）；
- ① `level` - 包含指标水平位特性的容器（位于 `indicator` 内）；
- ① `period` - 包含对象或指标在特定时间范围下的可见性特性的容器（位于 `object` 或 `indicator` 内）；
- ① `inputs` - 包含自定义指标和 EA 交易的设置的容器（input 变量）。

`key=value` 键值对形式的特性列表可能非常广泛，且没有官方文档记录。如果有必要，你可以自行处理平台的这些特性。

以下是一个 tpl 文件的片段（格式中的缩进用于直观展示容器的嵌套结构）：

```

<chart>
  id=0
  symbol=EURUSD
  description=Euro vs US Dollar
  period_type=1
  period_size=1
  digits=5
  ...
<window>
  height=117.133747
  objects=0
  <indicator>
    name=Main
    path=
    apply=1
    show_data=1
    ...
    fixed_height=-1
  </indicator>
</window>
<window>
  <indicator>
    name=Momentum
    path=
    apply=6
    show_data=1
    ...
    fixed_height=-1
    period=14
    <graph>
      name=
      draw=1
      style=0
      width=1
      color=16748574
    </graph>
  </indicator>
  ...
</window>
</chart>

```

我们有用于处理 tp1 文件的 *TplFile.mqh* 头文件，通过该文件，你可以分析和修改模板。它包含两个类：

- ⌚ **Container** - 用于读取并存储文件元素（考虑层级结构/嵌套关系），以及在可能的修改后写入文件；
- ⌚ **Selector** - 用于按顺序遍历层级结构中的元素（容器对象），以查找与特定查询匹配的元素，该查询以类似于 XPath 选择器的字符串形式编写（例如 `"/path/element[attribute=value]"`）。

Container 类的对象通过构造函数创建，该构造函数将用于读取的 tp1 文件描述符作为第一个参数，将标签名称作为第二个参数。默认情况下，标签名称为 `NULL`，这表示根容器（整个文件）。因此，容器本身会在读取文件的过程中自动填充内容（请参阅 **read** 方法）。

当前元素的特性 (即直接位于此容器内的 `"key=value"` 键值对)应被添加到 `MapArray<string,string> properties`映射中。嵌套容器会被添加到 `Container *children[]`数组中。

```

#include <MQL5Book/MapArray.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

class Container
{
    MapArray<string,string> properties;
    Container *children[];
    const string tag;
    const int handle;
public:
    Container(const int h, const string t = NULL): handle(h), tag(t) { }
    ~Container()
    {
        for(int i = 0; i < ArraySize(children); ++i)
        {
            if(CheckPointer(children[i]) == POINTER_DYNAMIC) delete children[i];
        }
    }

    bool read(const bool verbose = false)
    {
        while(!FileIsEnding(handle))
        {
            string text = FileReadString(handle);
            const int len = StringLen(text);
            if(len > 0)
            {
                if(text[0] == '<' && text[len - 1] == '>')
                {
                    const string subtag = StringSubstr(text, 1, len - 2);
                    if(subtag[0] == '/' && StringFind(subtag, tag) == 1)
                    {
                        if(verbose)
                        {
                            print();
                        }
                        return true;          // the element is ready
                    }

                    PUSH(children, new Container(handle, subtag)).read(verbose);
                }
                else
                {
                    string pair[];
                    if(StringSplit(text, '=', pair) == 2)
                    {
                        properties.put(pair[0], pair[1]);
                    }
                }
            }
        }
    }
}

```

```

    }
    return false;
}
...
};

```

在`read`方法中，我们将读取文件并逐行解析。如果遇到类似 `<tag>` 的开始标签，我们将创建一个新的容器对象，并在其中继续读取内容。如果遇到类似 `</tag>` 且名称相同的结束标签，我们将返回一个成功标志 (`true`)，表示该容器已生成完毕。在剩余行中，我们将读取 “键 = 值” 键值对并将它们添加到 `properties` 数组中。

我们已经准备好 `Selector` 用于在模板中搜索元素。一个包含被搜索标签层级结构的字符串将被传递给其构造函数。例如，字符串 `"/chart/window/indicator"` 对应一个图表，该图表包含一个窗口/子窗口，而该窗口又包含任意指标。搜索结果将是第一个匹配项。此查询通常会找到报价图表，因为它在模板中作为名为 “Main” 的指标存储，且位于文件开头、其他子窗口之前。

更实用的查询是指定特定特性的名称和值。具体来说，修改后的字符串 `"/chart/window/indicator[name=Momentum]"` 将仅查找 **Momentum** 指标。这种搜索方式与调用 `ChartWindowFind` 不同，因为在此处指定名称不带参数，而 `ChartWindowFind` 使用指标简称，该简称通常包含参数值，但参数值可能会变化。

对于内置指标，`name` 特性包含指标自身名称；而对于自定义指标，则显示为 “Custom Indicator”。自定义指标的链接在 `path` 特性中以可执行文件路径的形式提供，例如：`"Indicators\QL5Book\IndTripleEMA.ex5"`。

我们来看看 `Selector` 类的内部结构。

```

class Selector
{
    const string selector;
    string path[];
    int cursor;
public:
    Selector(const string s): selector(s), cursor(0)
    {
        StringSplit(selector, '/', path);
    }
    ...
}

```

在构造函数中，我们将 `selector` 查询分解为独立的组件，并将它们保存到 `path` 数组中。在模式中，当前正在匹配的路径组件由 `cursor` 变量提供。搜索开始时，我们位于根容器（即考虑整个 `tpl` 文件），且 `cursor` 为 0。当找到匹配项时，`cursor` 应递增（请参阅下文 `accept` 方法）。

在该类中，运算符 `[]` 被重载，通过该运算符可以获取路径的第 `i` 个片段。它还考虑了在片段中，方括号内可以指定 “[key=value]” 对。

```

string operator[](int i) const
{
    if(i < 0 || i >= ArraySize(path)) return NULL;
    const int param = StringFind(path[i], "[");
    if(param > 0)
    {
        return StringSubstr(path[i], 0, param);
    }
    return path[i];
}
...

```

*accept*方法用于检查元素的名称 (*tag*)及其特性 (*properties*)是否与选择器路径中当前光标位置指定的数据相匹配。*this[cursor]*记录使用了上述重载的 `[]` 运算符。

```

bool accept(const string tag, MapArray<string,string> &properties)
{
    const string name = this[cursor];
    if(!(name == "" && tag == NULL) && (name != tag))
    {
        return false;
    }

    // if the request has a parameter, check it among the properties
    // NB! so far only one attribute is supported, but many "tag[a1=v1][a2=v2]..."
    const int start = StringLen(path[cursor]) > 0 ? StringFind(path[cursor], "[") :
    if(start > 0)
    {
        const int stop = StringFind(path[cursor], "]");
        const string prop = StringSubstr(path[cursor], start + 1, stop - start - 1);

        // NB! only '=' is supported, but it should be '>', '<', etc.
        string kv[]; // key and value
        if(StringSplit(prop, '=', kv) == 2)
        {
            const string value = properties[kv[0]];
            if(kv[1] != value)
            {
                return false;
            }
        }
    }

    cursor++;
    return true;
}
...

```

如果标签名称与路径的当前片段不匹配,或者片段中包含某个参数的值且该值不相等或不存在于数组 *properties*中,该方法将返回*false*。在其他情况下,我们将获得条件匹配,其结果是光标将向前移动 (*cursor++*)且该方法将返回 *true*。

当游标到达请求中的最后一个片段时，搜索过程将成功完成，因此我们需要一个方法来判断这一时刻，即 *isComplete* 方法。

```
bool isComplete() const
{
    return cursor == ArraySize(path);
}

int level() const
{
    return cursor;
}
```

此外，在模板分析过程中可能会出现以下情况：我们遍历了路径的部分容器层级（即找到多个匹配项），但后续的请求片段不再匹配。在这种情况下，需要“返回”到请求的前一级别，为此实现了 *unwind* 方法。

```
bool unwind()
{
    if(cursor > 0)
    {
        cursor--;
        return true;
    }
    return false;
}
};
```

现在，我们已准备好使用 *Selector* 对象在容器层级结构（读取 `tp1` 文件后得到的结构）中执行搜索。所有必要操作都将由 *Container* 类中的 *find* 方法执行。它将 *Selector* 对象作为输入参数，并在根据 *Selector::accept* 方法找到匹配项时递归调用自身。到达请求的末尾意味着成功，此时 *find* 方法会将当前容器返回给调用代码。


```

Container *find(Selector *selector)
{
    const string element = StringFormat("%*s", 2 * selector.level(), " ")
        + "<" + tag + "> " + (string)ArraySize(children);
    if(selector.accept(tag, properties))
    {
        Print(element + " accepted");

        if(selector.isComplete())
        {
            return &this;
        }

        for(int i = 0; i < ArraySize(children); ++i)
        {
            Container *c = children[i].find(selector);
            if(c) return c;
        }
        selector.unwind();
    }
    else
    {
        Print(element);
    }

    return NULL;
}
...

```

请注意，当我们沿着对象树移动时，*find*方法会记录当前对象的标签名称和嵌套对象的数量，并且记录时会添加与对象嵌套层级成比例的缩进。如果该项与请求匹配，则会在日志条目中追加“accepted”一词。

还需要注意的是，此实现会返回第一个匹配的元素，并不再继续搜索其他候选项，从理论上讲，这对模板很有用，因为模板中同一容器内通常会有多个相同类型的标签。例如，一个窗口可能包含许多对象，而 MQL 程序可能需要解析对象的完整列表。这一方面建议作为可选研究内容。

为简化搜索调用，已添加一个同名方法，该方法接受字符串参数并在本地创建 *Selector* 对象。

```

Container *find(const string selector)
{
    Selector s(selector);
    return find(&s);
}

```

由于我们要编辑模板，因此应提供修改容器的方法，特别是用于添加键值对和具有指定标签的新嵌套容器的方法。

```

void assign(const string key, const string value)
{
    properties.put(key, value);
}

Container *add(const string subtag)
{
    return PUSH(children, new Container(handle, subtag));
}

void remove(const string key)
{
    properties.remove(key);
}

```

编辑完成后, 需要将容器的内容写回文件 (相同或不同文件)。辅助方法 **save** 将按上述 **tpl** 格式保存对象: 以开始标签 "**<tag>**" 开头, 接着导出所有 **key=value** 键值对特性, 然后对嵌套对象调用 **save** 方法, 最后以结束标签 "**</tag>**" 结尾。文件描述符作为保存操作的参数传入。

```

bool save(const int h)
{
    if(tag != NULL)
    {
        if(FileWriteString(h, "<" + tag + ">\n") <= 0)
            return false;
    }
    for(int i = 0; i < properties.getSize(); ++i)
    {
        if(FileWriteString(h, properties.getKey(i) + "=" + properties[i] + "\n") <= 0)
            return false;
    }
    for(int i = 0; i < ArraySize(children); ++i)
    {
        children[i].save(h);
    }
    if(tag != NULL)
    {
        if(FileWriteString(h, "</" + tag + ">\n") <= 0)
            return false;
    }
    return true;
}

```

将整个模板写入文件的高级方法称为 **write**。其输入参数 (文件描述符) 可以等于 0, 这意味着写入的文件就是之前读取的文件。但是, 文件必须以可写权限打开。

需要注意的是, 当覆盖 Unicode 文本文件时, MQL5 不会写入初始 UTF 标记 (即所谓的 BOM, 字节顺序标记), 因此我们必须自行处理这一步骤。否则, 没有该标记的话, 终端将无法读取和应用我们的模板。

如果调用代码向 **h** 参数传入另一个专门以 Unicode 格式打开用于写入的文件, MQL5 将自动写入 BOM。

```

bool write(int h = 0)
{
    bool rewriting = false;
    if(h == 0)
    {
        h = handle;
        rewriting = true;
    }
    if(!FileGetInteger(h, FILE_IS_WRITABLE))
    {
        Print("File is not writable");
        return false;
    }

    if(rewriting)
    {
        // NB! We write the BOM manually because MQL5 does not do this when overwrit
        ushort u[1] = {0xFEFF};
        FileSeek(h, SEEK_SET, 0);
        FileWriteString(h, ShortArrayToString(u));
    }

    bool result = save(h);

    if(rewriting)
    {
        // NB! MQL5 does not allow to reduce file size,
        // so we fill in the extra ending with spaces
        while(FileTell(h) < FileSize(h) && !IsStopped())
        {
            FileWriteString(h, " ");
        }
    }
    return result;
}

```

为了演示新类的功能，我们以隐藏特定指标窗口为例进行说明。如你所知，用户可以通过在指标特性对话框 **Display** 选项卡) 中重置时间范围的可见性标志来实现这一点。从编程角度来看，这无法直接实现。这正是模板编辑能力发挥作用的地方。

在模板中，指标对时间范围的可见性是在 `<indicator>` 容器中指定的，其中为每个可见的时间范围单独编写了一个 `<period>` 容器。例如，M15 时间范围的可见性如下：

```

<period>
period_type=0
period_size=15
</period>

```

在 `<period>` 容器内部，会使用 `period_type` 和 `period_size` 特性。`period_type` 是一个度量单位，可选值如下：

- ⊙0 表示分钟
- ⊙1 表示小时

⌚2 表示周

⌚3 表示月

`period_size` 是时间范围中的度量单位数量。需要注意的是，日线时间范围被指定为 24 小时。

当 `<indicator>` 容器中没有嵌套的 `<period>` 容器时，该指标将在所有时间范围中显示。

本书附带了脚本 *ChartTemplate.mq5*，该脚本可向图表添加 Momentum 指标（如果还不存在），并将其设置为仅在月线时间范围下可见。

```
void OnStart()
{
    // if Momentum(14) is not on the chart yet, add it
    const int w = ChartWindowFind(0, "Momentum(14)");
    if(w == -1)
    {
        const int momentum = iMomentum(NULL, 0, 14, PRICE_TYPICAL);
        ChartIndicatorAdd(0, (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL), momentum);
        // not necessarily here because the script will exit soon,
        // however explicitly declares that the handle will no longer be needed in the
        IndicatorRelease(momentum);
    }
    ...
}
```

接下来，我们将当前图表模板保存到一个文件中，然后打开该文件进行写入和读取操作。可以分配一个单独的文件用于写入。

```
const string filename = _Symbol + "-" + PeriodToString(_Period) + "-momentum-rw";
if(PRTF(ChartSaveTemplate(0, "/Files/" + filename)))
{
    int handle = PRTF(FileOpen(filename + ".tpl",
        FILE_READ | FILE_WRITE | FILE_TXT | FILE_SHARE_READ | FILE_SHARE_WRITE));
    // alternative - another file open for writing only
    // int writer = PRTF(FileOpen(filename + "w.tpl",
    //     FILE_WRITE | FILE_TXT | FILE_SHARE_READ | FILE_SHARE_WRITE));
}
```

在获得文件描述符后，我们将创建一个根容器 *main*，并将整个文件读取到该容器中（嵌套容器及其所有特性将自动读取）。

```
Container main(handle);
main.read();
```

然后，我们将定义一个选择器来搜索 *Momentum* 指标。理论上，更严谨的方法还需要检查指定周期（14），但我们的类不支持同时查询多个特性（此可能性留作自主研究）。

使用选择器搜索目标对象，输出找到的结果（仅作参考），并为其添加嵌套的 `<period>` 容器以设置月线时间范围的显示。

```

Container *found = main.find("/chart/window/indicator[name=Momentum]");
if(found)
{
    found.print();
    Container *period = found.add("period");
    period.assign("period_type", "3");
    period.assign("period_size", "1");
}

```

最后，我们将修改后的模板写入同一个文件，关闭该文件并将其应用到图表上。

```

main.write(); // or main.write(writer);
FileClose(handle);

PRTF(ChartApplyTemplate(0, "/Files/" + filename));
}
}

```

在空白图表上运行该脚本时，我们将会在日志中看到如下条目。

```

ChartSaveTemplate(0,/Files/+filename)=true / ok
FileOpen(filename+.tpl,FILE_READ|FILE_WRITE|FILE_TXT| »
» FILE_SHARE_READ|FILE_SHARE_WRITE|FILE_UNICODE)=1 / ok
<> 1 accepted
<chart> 2 accepted
  <window> 1 accepted
    <indicator> 0
  <window> 1 accepted
    <indicator> 1 accepted
Tag: indicator
      [key]      [value]
[ 0] "name"      "Momentum"
[ 1] "path"      ""
[ 2] "apply"     "6"
[ 3] "show_data" "1"
[ 4] "scale_inherit" "0"
[ 5] "scale_line" "0"
[ 6] "scale_line_percent" "50"
[ 7] "scale_line_value" "0.000000"
[ 8] "scale_fix_min" "0"
[ 9] "scale_fix_min_val" "0.000000"
[10] "scale_fix_max" "0"
[11] "scale_fix_max_val" "0.000000"
[12] "expertmode" "0"
[13] "fixed_height" "-1"
[14] "period"     "14"
ChartApplyTemplate(0,/Files/+filename)=true / ok

```

由此可见，在找到所需指标（标记为“accepted”）之前，算法已先在主窗口中找到该指标，但该指标不符合要求，因为其名称所需的“Momentum”不一致。

现在，如果打开图表上的指标列表，将看到 *momentum*，并且在其特性对话框的 *Display* 选项卡上，唯一启用的时间范围是 *Month*。

本书附有 *TplFileFull.mqh* 文件的扩展版本，该版本支持在标签选择条件中使用不同的比较运算，以及将多个匹配标签选择到数组中。在脚本 *ChartUnfix.mq5* 中可以找到使用该功能的示例，该脚本可解除所有图表子窗口的固定大小。

5.7.25 保存图表图像

在 MQL 程序中，往往需要记录程序自身及交易环境的当前状态。通常，我们会将各类分析指标或金融指标输出至日志中实现这一目的，但有些情况用图表图像显示会更清晰，例如在交易发生时。MQL5 API 包含一个可将图表图像保存为文件的函数。

```
bool ChartScreenShot(long chartId, string filename, int width, int height,
ENUM_ALIGN_MODE alignment = ALIGN_RIGHT)
```

会根据文件名称 *filename* (最长 63 个字符) 中扩展名的不同，将指定图表的截图保存为 GIF、PNG 或 BMP 格式。截图将自动保存到 *MQL5/Files* 目录中。

参数 *width* 和 *height* 用于设置图像的宽度和高度 (以像素为单位)。

参数 *alignment* 用于控制图形的哪一部分会被包含到文件中。值 *ALIGN_RIGHT* (默认) 表示截取最新价格区域 (这相当于终端在截图前自动执行了按下 *End* 键的翻页操作)。值 *ALIGN_LEFT* 确保截图中包含从当前左侧可见的第一根柱线开始的图表内容。因此，如果需要截取图表中特定位置的截图，必须先通过手动操作或调用 *ChartNavigate* 函数定位图表位置。

如果成功，函数 *ChartScreenShot* 返回 *true*。

让我们在脚本 *ChartPanorama.mq5* 中测试此函数。其任务是保存从当前左侧可见柱线到当前时间的图表副本。先将图形的起始点向后移动到所需的历史深度，你便可以获得一个相当长的全景图。在这种情况下，你无需考虑选择何种图像宽度。不过请注意，过长的历史数据会需要生成超大尺寸的图像，可能超出图形格式或软件的处理能力。

图像的高度将自动确定为当前图表的高度。

```

void OnStart()
{
    // the exact width of the price scale is not known, we take it empirically
    const int scale = 60;

    // calculating the total height, including gaps between windows
    const int w = (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL);
    int height = 0;
    int gutter = 0;
    for(int i = 0; i < w; ++i)
    {
        if(i == 1)
        {
            gutter = (int)ChartGetInteger(0, CHART_WINDOW_YDISTANCE, i) - height;
        }
        height += (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, i);
    }

    Print("Gutter=", gutter, ", total=", gutter * (w - 1));
    height += gutter * (w - 1);
    Print("Height=", height);

    // calculate the total width based on the number of pixels in one bar,
    // and also including chart offset from the right edge and scale width
    const int shift = (int)(ChartGetInteger(0, CHART_SHIFT) ?
        ChartGetDouble(0, CHART_SHIFT_SIZE) * ChartGetInteger(0, CHART_WIDTH_IN_PIXELS)
    Print("Shift=", shift);
    const int pixelPerBar = (int)MathRound(1.0 * ChartGetInteger(0, CHART_WIDTH_IN_PIX
        / ChartGetInteger(0, CHART_WIDTH_IN_BARS));
    const int width = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR) * pixelPerBar +
    Print("Width=", width);

    // write a file with a picture in PNG format
    const string filename = _Symbol + "-" + PeriodToString() + "-panorama.png";
    if(ChartScreenShot(0, filename, width, height, ALIGN_LEFT))
    {
        Print("File saved: ", filename);
    }
}

```

我们也可以使用 `ALIGN_RIGHT` 模式,但需强制禁用右侧边距偏移,因为该偏移会根据图像尺寸重新计算,导致最终效果与屏幕显示存在显著差异(右侧缩进会变得过大,因为它按宽度百分比指定)。

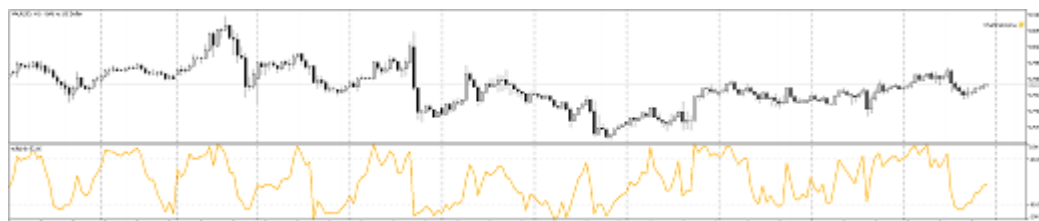
以下是在 XAUUSD,H1 图表上运行该脚本后的日志示例。

```

Gutter=2, total=2
Height=440
Shift=74
Width=2086
File saved: XAUUSD-H1-panorama.png

```

考虑到导航到的历史数据并非非常久远，我们获得了以下截图（以缩小到四分之一的副本形式呈现）。



图表全景

5.8 图形对象

MetaTrader 5 用户都很熟悉图形对象的概念：趋势线、价格标签、通道、Fibonacci 水平线、几何形状以及许多其他用于分析图表标记的可视化元素。MQL5 语言允许通过编程方式创建、编辑和删除图形对象。该功能可能非常实用，例如，当需要在指标子窗口和主窗口中同时显示特定数据时。由于平台仅支持在一个窗口中输出指标缓冲区，我们可以在另一个窗口中生成对象。借助图形对象创建的标记，使用 [EA 交易](#) 可轻松实现半自动化交易。此外，对象常被用于为 MQL 程序构建自定义图形界面，例如按钮、输入字段和标志。这些程序无需打开特性对话框即可进行控制，且 MQL 创建的面板比标准输入变量具有更高的灵活性。

每个对象都存在于特定图表的上下文中。因此，本章将讨论的函数有一个共同特征：第一个参数用于指定 [图表 ID](#)。此外，每个图形对象都有一个在单个图表（包括所有子窗口）中唯一的名称。更改图形对象名称将导致旧名称对象被删除，并创建同名新对象。不能创建两个同名对象。

用于定义图形对象特性的函数，以及在图表上创建 ([ObjectCreate](#)) 和移动 ([ObjectMove](#)) 对象的操作，本质上都是向图表发送异步命令。如果这些函数执行成功，命令将进入图表的共享事件队列。图形对象特性的可视化修改发生在处理该特定图表的事件队列期间。因此，在函数调用后，图表的外观可能会延迟反映对象状态的变化。

通常，图表上图形对象的更新由终端自动完成，以响应接收新报价、调整窗口大小等与图表相关的事件。要强制更新图形对象，可使用请求重绘图表的函数 ([ChartRedraw](#))。这在批量创建或修改对象后尤为重要。

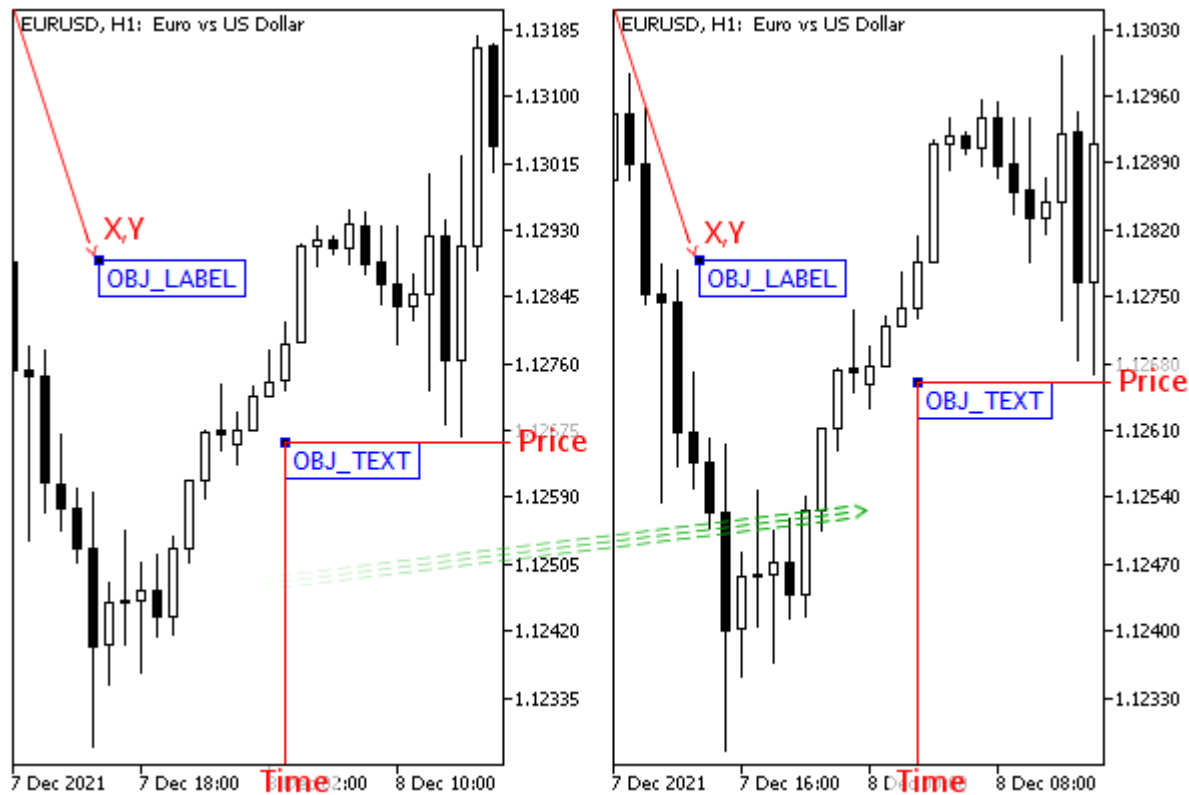
对象可作为编程事件的来源，例如创建、删除、修改其特性以及鼠标点击等事件。所有有关事件触发和处理的细节将在单独的 [章节](#) 中，与常规窗口上下文中的事件一起讨论。

我们将从理论基础开始，逐步过渡到实践应用。

5.8.1 对象类型及其坐标指定的特点

我们在 [图表](#) 章节中已经知道，窗口中存在两种坐标系：屏幕（像素）坐标和报价（时间和价格）坐标。在这方面，支持的对象类型总体上分为两大类：与屏幕关联的对象和与价格图表关联的对象。第一类对象始终相对于窗口的某个角保持固定位置（该角落为参考点，由用户或程序员在对象特性中确定）。第二类会随着窗口的工作区域一起滚动。

下图展示了两个带文本标签的对象用于对比：一个附加在屏幕上（OBJ_LABEL），另一个附加在价格图表上（OBJ_TEXT）。它们的类型（在括号中给出）以及用于设置坐标的特性，我们将在本章的相关章节中学习。需要注意的是，当滚动价格图表时，OBJ_TEXT 文本会与其同步移动，而 OBJ_LABEL 文本则保持在原位。



对象的两种不同坐标系

此外，这些对象在锚点数量上也有所不同。例如，单个价格标签（“箭头”）需要一个时间/价格点，而趋势线则需要两个这样的点。还有一些对象类型具有更多锚点，例如等距通道、三角形或 Elliott 波浪。

当对象被选中时（例如在 *Object List* 对话框中通过双击或单击图表选中，具体取决于 *Charts* 选项卡 / *Select objects with a single mouse click* 选项），其锚点会以对比色小方块标示。正是这些锚点被用于拖动对象以及调整尺寸和改变方向。

所有受支持的对象类型均在 `ENUM_OBJECT` 枚举中进行了描述。你可以在 MQL5 文档中查看其完整内容。我们将逐步、分部分讨论其元素。

5.8.2 时间与价格约束对象

下表提供了对象、对象的时间和价格坐标、对象在 `ENUM_OBJECT` 枚举中的标识符，以及锚点数量。

标识符	名称	锚点
单一直线		
OBJ_VLINE	垂直线（仅时间坐标）	1

标识符	名称	键
OBJ_HLINE	水平线 (仅价格坐标)	1
OBJ_TREND	趋势线	2
OBJ_ARROWED_LINE	末端带箭头	2
周期性重复垂直线		
OBJ_CYCLES	循环线	2
通道		
OBJ_CHANNEL	等距通道	3
OBJ_STDDEVCHANNEL	标准差通道	2
OBJ_REGRESSION	线性回归线	2
OBJ_PITCHFORK	Andrew s 分叉线	3
Fibonacci 工具		
OBJ_FIBO	关卡	2
OBJ_FIBOTIMES	时区	2
OBJ_FIBOFAN	扇形线	2
OBJ_FIBOARC	弧线	2
OBJ_FIBOCHANNEL	通道	3
OBJ_EXPANSION	扩展	3
Gann 工具		
OBJ_GANNLIN	线	2
OBJ_GANNFAN	扇形线	2
OBJ_GANNGRID	网格	2
Elliot 波浪		
OBJ_ELLIOTWAVE5	推动浪	5
OBJ_ELLIOTWAVE3	调整浪	3
形状		
OBJ_RECTANGLE	矩形	2
OBJ_TRIANGLE	三角形	3
OBJ_ELLIPSE	椭圆	3

标识符	名称	锚点
单个标记和标签		
OBJ_ARROW_THUMB_UP	向上拇指	1*
OBJ_ARROW_THUMB_DOWN	向下拇指	1*
OBJ_ARROW_UP	向上箭头	1*
OBJ_ARROW_DOWN	向下箭头	1*
OBJ_ARROW_STOP	停止标记	1*
OBJ_ARROW_CHECK	选中标记	1*
OBJ_ARROW_LEFT_PRICE	左侧价格标签	1
OBJ_ARROW_RIGHT_PRICE	右侧价格标签	1
OBJ_ARROW_BUY	买入信号 (蓝色向上箭头)	1
OBJ_ARROW_SELL	卖出信号 (红色向下箭头)	1
OBJ_ARROW	任意 W ingdings 字符	1*
文本和图形		
OBJ_TEXT	文本	1*
OBJ_BITMAP	图片	1*
事件		
OBJ_EVENT	主窗口底部时间戳 (仅时间坐标)	1

星号标记的对象允许在其自身上选择锚点 (例如, 在对象的某个角落或某条边的中点)。不同对象类型的选择方法可能不同, 具体细节将在 [定义对象锚点](#) 章节中详细说明。锚点是必需的, 因为对象具有一定尺寸, 没有锚点会导致位置不明确。

5.8.3 绑定到屏幕坐标的对象

下表列出了基于屏幕坐标定位的对象的名称和 `ENUM_OBJECT` 标识符。除图表对象外, 几乎所有这些对象均用于为程序创建用户界面。特别是, 存在包括按钮、输入字段等基本控件, 以及用于对象可视化分组的标签和面板。基于这些基本控件, 可以创建更复杂的控件 (例如下拉列表或复选框)。与终端一起提供的还有一个包含预定义控件的类库, 以头文件集的形式提供 (请参阅 `MQ5/Include/Controls` 目录)。

标识符	名称	设置锚点
OBJ_LABEL	文本标签	是
OBJ_RECTANGLE_LABEL	矩形面板	
OBJ_BITMAP_LABEL	带图像的面板	是
OBJ_BUTTON	按钮	
OBJ_EDIT	输入字段	
OBJ_CHART	图表对象	

所有这些对象都需要在图表窗口中 **确定锚点定位角**。默认情况下，它们的坐标相对于窗口的左上角进行定位。

此列表中的类型还会在对象上使用一个锚点，且仅使用一个锚点。在某些对象中，锚点可编辑，而在另一些对象中则为硬编码。例如，矩形面板、按钮、输入字段和图表对象始终锚定在其左上角。而对于标签或带图片的面板，有许多锚点选项可供选择。该选项基于 **定义对象锚点** 章节中所述的 `ENUM_ANCHOR_POINT` 作出。

文本标签（`OBJ_LABEL`）用于提供文本输出，但不支持编辑功能。如需进行编辑，请使用输入字段（`OBJ_EDIT`）。

5.8.4 创建对象

要创建对象，需要一组所有类型通用的最小特性集。每种类型特有的附加特性可在对象创建后进行设置或修改。必需特性包括要创建对象的图表标识符、对象名称、窗口/子窗口编号、第一个锚点的两个坐标，即时间和价格。

虽然存在基于屏幕坐标定位的对象组，但创建时仍需传入两个值（通常为零，因其实际未被使用）。

通常，`ObjectCreate`函数原型如下：

```
bool ObjectCreate(long chartId, const string name, ENUM_OBJECT type, int window,
datetime time1, double price1, datetime time2 = 0, double price2 = 0, ...)
```

`chartId`值为 0 表示当前图表。`name`参数在整个图表（包括子窗口）中必须唯一，且长度不超过 63 个字符。

此前章节已介绍 `type`参数对应的对象类型，即 `ENUM_OBJECT` 枚举中的元素。

如我们所知，`window`参数的窗口/子窗口编号从 0 开始（0 表示主图表窗口）。如果为子窗口指定较大的索引值，则该子窗口必须存在，否则函数将以错误终止并返回 `false`。

需要提醒的是，返回的成功标志（`true`）仅表示创建对象的命令已成功加入队列。其执行结果并非立即可知。这是采用异步调用提升性能所带来的固有特性。

要检查执行结果，可使用 `ObjectFind` 函数或任何 `ObjectGet` 函数查询对象特性。但需注意，此类函数会等待整个图表命令队列执行完毕，才会返回实际结果（对象状态）该过程可能需要一些

时间,期间 MQL 程序代码会暂停执行。换言之,与创建和修改对象的函数不同,检查对象状态的函数是同步执行的。

从第二个锚点开始,其他附加锚点为可选参数。允许的锚点数量最多 30 个(供未来扩展使用),当前对象类型中使用的锚点不超过 5 个。

需特别注意,调用 *ObjectCreate* 函数时,如果使用已存在的对象名称,该函数仅会修改锚点坐标(如果自上次调用后坐标发生了变更)。这一特性便于编写统一代码,无需根据对象是否存在而编写条件分支。换言之,只要不关心对象是否已存在,无条件调用 *ObjectCreate* 即可确保对象存在。但需注意一个细节。调用 *ObjectCreate* 时,如果对象类型或子窗口索引与现有对象不同,相关数据将保持不变,且不会产生错误。

调用 *ObjectCreate* 时,可以将所有锚点保留为默认值 (null),前提是在该指令后调用具有相应 OBJPROP_TIME 和 OBJPROP_PRICE 特性的 *ObjectSet* 函数。

对于某些对象类型,锚点的指定顺序可能很重要。例如,对于 OBJ_REGRESSION (线性回归通道)和 OBJ_STDDEVCHANNEL (标准差通道)等通道,必须满足 $time1 < time2$ 的条件。否则,尽管会成功创建对象,但将无法构建通道。

以 *ObjectSimpleShowcase.mq5* 脚本为例,该脚本在图表最新柱线位置创建多个不同类型的对象,且仅需一个锚点。

所有对象操作示例均使用 *ObjectPrefix.mqh* 头文件,该文件包含对象名称的通用前缀字符串。因而,在需要时,我们可以更便捷地从图表中清除“自有”对象。

```
const string ObjNamePrefix = "ObjShow-";
```

在 *OnStart* 函数中,定义一个包含对象类型的数组。

```
void OnStart()
{
    ENUM_OBJECT types[] =
    {
        // straight lines
        OBJ_VLINE, OBJ_HLINE,
        // labels (arrows and other signs)
        OBJ_ARROW_THUMB_UP, OBJ_ARROW_THUMB_DOWN,
        OBJ_ARROW_UP, OBJ_ARROW_DOWN,
        OBJ_ARROW_STOP, OBJ_ARROW_CHECK,
        OBJ_ARROW_LEFT_PRICE, OBJ_ARROW_RIGHT_PRICE,
        OBJ_ARROW_BUY, OBJ_ARROW_SELL,
        // OBJ_ARROW, // see the ObjectWingdings.mq5 example

        // text
        OBJ_TEXT,
        // event flag (like in a calendar) at the bottom of the window
        OBJ_EVENT,
    };
}
```

接下来,通过循环遍历数组元素,在主窗口创建对象,并传入第 *i* 根柱线的时间和收盘价。

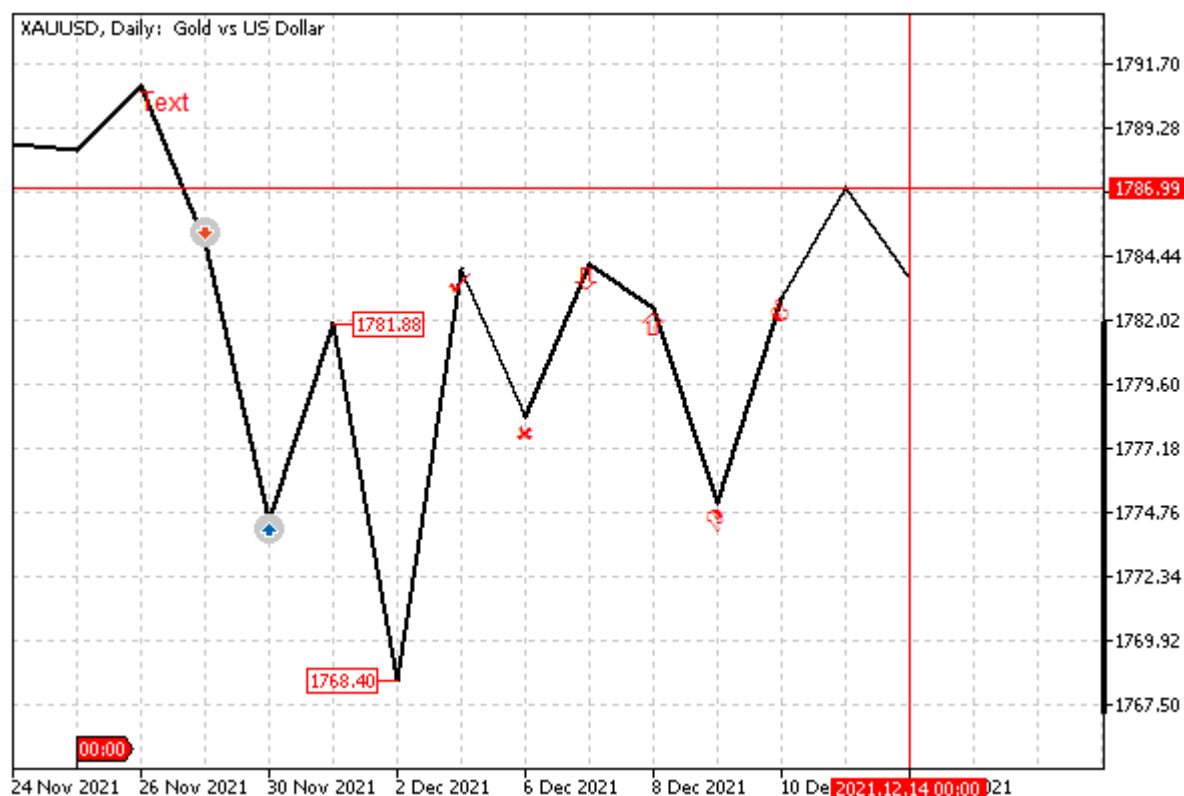
```

const int n = ArraySize(types);
for(int i = 0; i < n; ++i)
{
    ObjectCreate(0, ObjNamePrefix + (string)iTime(_Symbol, _Period, i), types[i],
        0, iTime(_Symbol, _Period, i), iClose(_Symbol, _Period, i));
}

PrintFormat("%d objects of various types created", n);
}

```

脚本执行后可能的结果如下：



最新柱线收盘价位置显示的简单类型对象

本示例中启用了 **Close** 价绘制线条和网格显示。我们将在后续章节学习如何调整对象的大小、颜色及其他特性。特别说明，大数图标对象的锚点默认位于顶边中点，因此视觉上会显示在价格线下方。但卖出图标显示在价格线上方，因为其锚点始终位于底边中点。

请注意，通过编程创建的对象默认不会显示在同名对话框的对象列表中。要查看这些对象，需点击 **All** 按钮。

5.8.5 删除对象

MQL5 API 提供了两个用于删除对象的函数。要批量删除符合名称前缀、类型或子窗口编号条件的对象，可使用 **ObjectsDeleteAll** 函数。如果需要通过其他条件（如过时的日期时间坐标）选择要删除的对象，或仅删除单个对象，则使用 **ObjectDelete** 函数。

ObjectsDeleteAll 函数有两种形式：带名称前缀参数的形式和不带名称前缀参数的形式

```
int ObjectsDeleteAll(long chartId, int window = -1, int type = -1)
int ObjectsDeleteAll(long chartId, const string prefix, int window = -1, int type = -1)
```

该函数将删除指定 *chartId* 图表上的所有对象，并考虑子窗口、类型和名称中的起始子字符串。

chartId 参数值为 0 时，照例表示当前图表。

window 和 *type* 参数的默认值 (-1) 分别定义所有子窗口和所有对象类型。

如果前缀为空，则将删除任意名称的对象。

该函数为同步执行，即将阻塞调用的 MQL 程序直至完成，并返回已删除对象的数量。由于该函数会等待调用前图表队列中所有命令执行完毕，因此操作可能需要一些时间。

```
bool ObjectDelete(long chartId, const string name)
```

该函数删除指定 *chartId* 图表上具有指定名称的对象。

与 *ObjectsDeleteAll* 不同，*ObjectDelete* 以异步方式执行，即它向图形发送删除对象命令，随后立即将控制权返回给 MQL 程序。返回值为 *true* 表示命令已成功加入队列。要检查执行结果，可使用 *ObjectFind* 函数或任何 *ObjectGet* 函数查询对象特性。

以 *ObjectCleanup1.mq5* 脚本为例。它的任务是移除带有 “our” 前缀的对象，这些对象由上一节的 *ObjectSimpleShowcase.mq5* 脚本生成。

在最简单的情况下，我们可以这样写：

```
#include "ObjectPrefix.mqh"

void OnStart()
{
    const int n = ObjectsDeleteAll(0, ObjNamePrefix);
    PrintFormat("%d objects deleted", n);
}
```

但为了增加多样性，我们还可以提供通过多次调用 *ObjectDelete* 函数来删除对象的选项。当然，当 *ObjectsDeleteAll* 满足所有要求时，采用此方法并无实际意义。但情况并非总是如此：当需要根据特殊条件（即不仅限于前缀和类型）选择对象时，*ObjectsDeleteAll* 将不再适用。

后续当我们掌握读取对象特性的函数后，将完善此示例。当前，我们仅引入一个用于切换到“高级”删除模式的输入变量 (*UseCustomDeleteAll*)。

```
#property script_show_inputs
input bool UseCustomDeleteAll = false;
```

在 *OnStart* 函数中，根据所选模式，我们将调用标准的 *ObjectsDeleteAll*，或我们自己实现的 *CustomDeleteAllObjects*。

```

void OnStart()
{
    const int n = UseCustomDeleteAll ?
        CustomDeleteAllObjects(0, ObjNamePrefix) :
        ObjectsDeleteAll(0, ObjNamePrefix);

    PrintFormat("%d objects deleted", n);
}

```

我们将先构建该函数的框架，再逐步完善其实现。

```

int CustomDeleteAllObjects(const long chart, const string prefix,
    const int window = -1, const int type = -1)
{
    int count = 0;
    const int n = ObjectsTotal(chart, window, type);

    // NB: cycle through objects in reverse order of the internal chart list
    // to keep numbering as we move away from the tail
    for(int i = n - 1; i >= 0; --i)
    {
        const string name = ObjectName(chart, i, window, type);
        if(StringLen(prefix) == 0 || StringFind(name, prefix) == 0)
            // additional checks that ObjectsDeleteAll does not provide,
            // for example, by coordinates, color or anchor point
            ...
        {
            // send a command to delete a specific object
            count += ObjectDelete(chart, name);
        }
    }
    return count;
}

```

此处涉及多项新功能，将在 [下一节](#) 中介绍 (*ObjectsTotal*、*ObjectName*)。总体而言，这些函数的功能应当明确：第一个函数返回图表上对象的索引，第二个函数返回指定索引下的对象名称。

同样需要注意的是，对象遍历采用降序索引循环方式：如果按常规顺序处理，删除列表起始位置的对象会导致索引编号错乱。严格来说，当前的循环也不能保证完全删除，假设有另一个 MQL 程序在我们删除的同时并行添加对象。确实，新的“外部”对象可能被添加到列表开头（列表按对象名称字母顺序生成），这会导致剩余索引值增加，从而将原本待删除的“our”对象推到当前索引 *i* 之后。新对象添加到开头的数量越多，就越有可能漏删自己的对象。

因此，为提高可靠性，可以在循环结束后检查剩余对象数量是否等于初始数量减去已删除对象数量。不过这也无法提供 100% 的保证，因为其他程序同样可能并行删除对象。我们将这些具体细节留待自行研究。

在当前实现中，无论是否切换 *UseCustomDeleteAll* 模式，我们的脚本都应删除所有带“our”前缀的对象。日志应显示如下内容：


```
ObjectSimpleShowcase (XAUUSD,H1) 14 objects of various types created
ObjectCleanup1 (XAUUSD,H1) 14 objects deleted
```

让我们先了解刚刚使用的 *ObjectsTotal*和 *ObjectName* 函数，然后再回到 *ObjectCleanup2.mq5* 版本的脚本。

5.8.6 查找对象

有三个函数可用于在图表上搜索对象。前两个函数 *ObjectsTotal*和 *ObjectName* 允许按名称遍历对象，然后在需要时使用每个对象的名称分析其他特性（下一节详述具体实现方法）。第三个函数 *ObjectFind*允许通过已知名称验证对象是否存在。同样，也可以通过 *ObjectGet*函数直接请求某个特性来实现这一点：如果不存在与传入名称匹配的对象，*_LastError* 中将返回错误，但这种方式不如调用 *ObjectFind* 便捷。此外，该函数会立即返回对象所在窗口编号。

int ObjectsTotal(long chartId, int window = -1, int type = -1)

该函数返回指定 *chartId*标识符（0 表示当前图表）的图表上的对象数量。计算时仅考虑指定 *window*编号的子窗口中的对象（0 表示主窗口，-1 表示主窗口及所有子窗口）。请注意，仅考虑 *type*参数中指定的特定类型的对象（默认值 -1 表示所有类型）。*type*的值可以是 *ENUM_OBJECT* 枚举中的元素。

该函数采用同步执行方式，即会阻塞调用的 MQL 程序，直到获取结果。

string ObjectName(long chartId, int index, int window = -1, int type = -1)

该函数返回标识符为 *chartId*的图表上指定索引号 *index* 的对象名称。在编译用于搜索对象的内部列表时，会考虑指定的子窗口编号（*window*）和对象类型（*type*）。此列表按对象名称的字典序排序，即具体按字母顺序排列且区分大小写。

与 *ObjectsTotal*类似，*ObjectName* 在执行期间会等待获取整个图表命令队列，然后从更新后的对象列表中返回对象名称。

如果发生错误，将返回空字符串，且将 *OBJECT_NOT_FOUND*（4202）错误代码存储在 *_LastError* 中。

为了测试这两个函数的功能，我们将创建一个名为 *ObjectFinder.mq5*的脚本，用于记录所有图表上的所有对象。该脚本使用 图表迭代 函数（*ChartFirst*和 *ChartNext*），以及获取 图表特性 的函数（*ChartSymbol*、*ChartPeriod* 和 *ChartGetInteger*）。

```

#include <MQL5Book/Periods.mqh>

void OnStart()
{
    int count = 0;
    long id = ChartFirst();
    // loop through charts
    while(id != -1)
    {
        PrintFormat("%s %s (%lld)", ChartSymbol(id), PeriodToString(ChartPeriod(id)), id);
        const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
        // loop through windows
        for(int k = 0; k < win; ++k)
        {
            PrintFormat("  Window %d", k);
            const int n = ObjectsTotal(id, k);
            // loop through objects
            for(int i = 0; i < n; ++i)
            {
                const string name = ObjectName(id, i, k);
                const ENUM_OBJECT type = (ENUM_OBJECT)ObjectGetInteger(id, name, OBJPROP_
                PrintFormat("    %s %s", EnumToString(type), name);
                ++count;
            }
        }
        id = ChartNext(id);
    }

    PrintFormat("%d objects found", count);
}

```

对于每个图表，我们确定子窗口的数量 (*ChartGetInteger(id, CHART_WINDOWS_TOTAL)*)，为每个子窗口调用 *ObjectsTotal* 函数，并在内层循环中调用 *ObjectName*。接下来，我们通过名称查找对象类型，并将它们一并显示在日志中。

下面是该脚本可能的运行结果 (采用缩写形式)。

```

EURUSD H1 (132358585987782873)
  Window 0
    OBJ_FIBO H1 Fibo 58513
    OBJ_TEXT H1 Text 40688
    OBJ_TREND H1 Trendline 3291
    OBJ_VLINE H1 Vertical Line 28732
    OBJ_VLINE H1 Vertical Line 33752
    OBJ_VLINE H1 Vertical Line 35549
  Window 1
  Window 2
EURUSD D1 (132360375330772909)
  Window 0
EURUSD M15 (132544239145024745)
  Window 0
    OBJ_VLINE H1 Vertical Line 27032
...
XAUUSD D1 (132544239145024746)
  Window 0
    OBJ_EVENT ObjShow-2021.11.25 00:00:00
    OBJ_TEXT ObjShow-2021.11.26 00:00:00
    OBJ_ARROW_SELL ObjShow-2021.11.29 00:00:00
    OBJ_ARROW_BUY ObjShow-2021.11.30 00:00:00
    OBJ_ARROW_RIGHT_PRICE ObjShow-2021.12.01 00:00:00
    OBJ_ARROW_LEFT_PRICE ObjShow-2021.12.02 00:00:00
    OBJ_ARROW_CHECK ObjShow-2021.12.03 00:00:00
    OBJ_ARROW_STOP ObjShow-2021.12.06 00:00:00
    OBJ_ARROW_DOWN ObjShow-2021.12.07 00:00:00
    OBJ_ARROW_UP ObjShow-2021.12.08 00:00:00
    OBJ_ARROW_THUMB_DOWN ObjShow-2021.12.09 00:00:00
    OBJ_ARROW_THUMB_UP ObjShow-2021.12.10 00:00:00
    OBJ_HLINE ObjShow-2021.12.13 00:00:00
    OBJ_VLINE ObjShow-2021.12.14 00:00:00
...
35 objects found

```

特别是，在此处可以看到，XAUUSD， D1 图表上存在由 *ObjectSimpleShowcase.mq5* 脚本生成的对象。部分图表和子窗口中没有对象。

int ObjectFind(long chartId, const string name)

该函数在指定标识符的图表上按名称搜索对象，如果成功，则返回其所在窗口编号。

如果未找到对象，该函数将返回负值。与本节前述函数类似，*ObjectFind* 函数采用同步调用方式。

在下一节 *ObjectCopy.mq5* 脚本中，我们将看到该函数的使用示例。

5.8.7 对象特性访问函数概述

对象具有多种类型的特性，这些特性可通过 *ObjectGet* 和 *ObjectSet* 函数进行读取和设置。我们知道，这一原则已在图表中应用（请参阅 [用于处理完整图表特性集的函数概述](#) 章节）。

所有此类函数的前三个参数均为图表标识符、对象名称和特性标识符，其中特性标识符必须是 `ENUM_OBJECT_PROPERTY_INTEGER`、`ENUM_OBJECT_PROPERTY_DOUBLE` 或 `ENUM_OBJECT_PROPERTY_STRING` 枚举的成员。我们将在后续章节逐步研究具体特性。完整的特性对照表可在 MQL5 文档的 [对象特性](#) 页面上找到。

需要注意的是，所有三个枚举中的特性标识符互不重叠，这使得我们可以将它们的联合处理合并为单个统一代码。我们将在示例中运用这一点。

某些特性为只读，并将标记为 “r/o” (只读)。

与绘图 API 的情况类似，特性读取函数具有简写和完整两种形式：简写形式直接返回请求的值，完整形式返回布尔值 (*true* 表示成功，*false* 表示发生错误)，实际值则通过引用传递的最后一个参数返回。调用简写形式时，必须通过内置变量 `_LastError` 检查是否发生错误。

访问某些特性时，必须指定一个额外参数 (*modifier*)，用于在特性为多值时指示值的编号或级别。例如，如果一个对象有多个锚点，则修饰符可用于选择特定的锚点。

以下是用于读取和写入整数型特性的函数原型。需特别注意，这些函数的值类型为 *long*，因此不仅允许存储 *int* 或 *long* 类型的特性，还可以存储 *bool*、*color* 或 *datetime* 类型的特性以及各种枚举 (见下文)。

```
bool ObjectSetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER property, long value)
```

```
bool ObjectSetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER property, int modifier, long value)
```

```
long ObjectGetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER property, int modifier = 0)
```

```
bool ObjectGetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER property, int modifier, long &value)
```

实数型特性的函数说明类似。

```
bool ObjectSetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property, double value)
```

```
bool ObjectSetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property, int modifier, double value)
```

```
double ObjectGetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property, int modifier = 0)
```

```
bool ObjectGetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property, int modifier, double &value)
```

最后，字符串同样存在四个相同的函数。

```
bool ObjectSetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, const string value)
```

```
bool ObjectSetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, int modifier, const string value)
```

```
string ObjectGetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, int modifier = 0)
```

```
bool ObjectGetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, int modifier, string &value)
```

为提升性能，所有设置对象特性的函数 (*ObjectSetInteger*、*ObjectSetDouble* 和 *ObjectSetString*) 均为异步函数，本质上是向图表发送修改对象的命令。当这些函数成功执行时，命令将被存入图

表的共享事件队列，并返回结果 *true* 表示操作成功。如果发生错误，函数将返回 *false*，此时必须通过 *_LastError* 变量检查错误代码。

对象特性的更改存在一定延迟，在图表事件队列处理期间完成。要强制更新图表上对象的外观和特性（尤其是在一次性更改多个对象后），可使用 *ChartRedraw* 函数。

获取图表特性的函数 (*ObjectGetInteger*、*ObjectGetDouble* 和 *ObjectGetString*) 是同步的，这意味着调用代码会等待这些函数的执行结果。在这种情况下，图表队列中的所有命令均被执行，以获取最新特性值。

让我们回到 [删除对象脚本](#) 的示例，更准确地说，是其新版本 *ObjectCleanup2.mq5*。回顾在 *CustomDeleteAllObjects* 函数中，我们希望实现根据对象特性选择对象的功能。假设这些特性应该是颜色和锚点。要获取这些特性，可使用 *ObjectGetInteger* 函数以及 *ENUM_OBJECT_PROPERTY_INTEGER* 枚举中的一对元素：*OBJPROP_COLOR* 和 *OBJPROP_ANCHOR*。我们将在后续详细探讨这些内容。

基于此，代码需补充以下检查逻辑（为简化演示，此处颜色和锚点分别使用 *clrRed* 和 *ANCHOR_TOP* 常量表示，实际应用中我们会提供对应的输入变量）。

```
int CustomDeleteAllObjects(const long chart, const string prefix,
    const int window = -1, const int type = -1)
{
    int count = 0;

    for(int i = ObjectsTotal(chart, window, type) - 1; i >= 0; --i)
    {
        const string name = ObjectName(chart, i, window, type);
        // condition on the name and additional properties, such as color and anchor pc
        if((StringLen(prefix) == 0 || StringFind(name, prefix) == 0)
            && ObjectGetInteger(0, name, OBJPROP_COLOR) == clrRed
            && ObjectGetInteger(0, name, OBJPROP_ANCHOR) == ANCHOR_TOP)
        {
            count += ObjectDelete(chart, name);
        }
    }
    return count;
}
```

请注意包含 *ObjectGetInteger* 的代码行。

这些代码较长且存在重复，因为特定特性与已知类型的 *ObjectGet* 函数绑定。此外，随着条件数量的增加，重复填写图表 ID 和对象名称会显得冗余。

为简化代码结构，我们将采用在 [图表显示模式](#) 章节中 *ChartModeMonitor.mqh* 文件中测试过的技术。这样做的意义在于构建一个中间类，通过方法重载实现对所有类型特性的读写操作。我们将新的头文件命名为 *ObjectMonitor.mqh*。

ObjectProxy 类的结构与用于图表 *ChartModeMonitorInterface* 类高度相似。主要区别在于它包含用于设置和获取图表 ID 及对象名称的虚方法。

```

class ObjectProxy
{
public:
    long get(const ENUM_OBJECT_PROPERTY_INTEGER property, const int modifier = 0)
    {
        return ObjectGetInteger(chart(), name(), property, modifier);
    }
    double get(const ENUM_OBJECT_PROPERTY_DOUBLE property, const int modifier = 0)
    {
        return ObjectGetDouble(chart(), name(), property, modifier);
    }
    string get(const ENUM_OBJECT_PROPERTY_STRING property, const int modifier = 0)
    {
        return ObjectGetString(chart(), name(), property, modifier);
    }
    bool set(const ENUM_OBJECT_PROPERTY_INTEGER property, const long value,
        const int modifier = 0)
    {
        return ObjectSetInteger(chart(), name(), property, modifier, value);
    }
    bool set(const ENUM_OBJECT_PROPERTY_DOUBLE property, const double value,
        const int modifier = 0)
    {
        return ObjectSetDouble(chart(), name(), property, modifier, value);
    }
    bool set(const ENUM_OBJECT_PROPERTY_STRING property, const string value,
        const int modifier = 0)
    {
        return ObjectSetString(chart(), name(), property, modifier, value);
    }

    virtual string name() = 0;
    virtual void name(const string) { }
    virtual long chart() { return 0; }
    virtual void chart(const long) { }
};

```

我们将在派生类中实现这些方法（后续我们将参照图表特性监视器的设计，扩展对象特性监视器的类层次结构）。

```

class ObjectSelector: public ObjectProxy
{
protected:
    long host; // chart ID
    string id; // chart ID
public:
    ObjectSelector(const string _id, const long _chart = 0): id(_id), host(_chart) { }

    virtual string name()
    {
        return id;
    }
    virtual void name(const string _id)
    {
        id = _id;
    }
    virtual void chart(const long _chart) override
    {
        host = _chart;
    }
};

```

我们将抽象接口 *ObjectProxy* 与其在 *ObjectSelector* 中的最小实现分离，这是考虑到未来可能需要为同类型多对象实现代理数组。然后，只需在新建的 “multiselector” 类中存储名称数组或公共前缀，并通过调用重载的 `[]` 运算符确保从 *name* 方法返回其中一个名称：*multiSelector[i].get(OBJPROP_XYZ)*。

现在回到 *ObjectCleanup2.mq5* 脚本，定义两个输入变量用于指定颜色和锚点，作为选择待删除对象的附加条件。

```

// ObjectCleanup2.mq5
...
input color CustomColor = clrRed;
input ENUM_ARROW_ANCHOR CustomAnchor = ANCHOR_TOP;

```

将这些值传递给 *CustomDeleteAllObjects* 函数，借助中间类，对象循环中的新条件检查可以更精简地实现。

```

#include <MQL5Book/ObjectMonitor.mqh>

void OnStart()
{
    const int n = UseCustomDeleteAll ?
        CustomDeleteAllObjects(0, ObjNamePrefix, CustomColor, CustomAnchor) :
        ObjectsDeleteAll(0, ObjNamePrefix);
    PrintFormat("%d objects deleted", n);
}

int CustomDeleteAllObjects(const long chart, const string prefix,
    color clr, ENUM_ARROW_ANCHOR anchor,
    const int window = -1, const int type = -1)
{
    int count = 0;
    for(int i = ObjectsTotal(chart, window, type) - 1; i >= 0; --i)
    {
        const string name = ObjectName(chart, i, window, type);

        ObjectSelector s(name);
        ResetLastError();
        if((StringLen(prefix) == 0 || StringFind(s.get(OBJPROP_NAME), prefix) == 0)
            && s.get(OBJPROP_COLOR) == CustomColor
            && s.get(OBJPROP_ANCHOR) == CustomAnchor
            && _LastError != 4203) // OBJECT_WRONG_PROPERTY
        {
            count += ObjectDelete(chart, name);
        }
    }
    return count;
}

```

需要重点说明的是，我们仅在创建 *ObjectSelector* 对象时指定一次对象名称（以及当前图表的隐式标识符 0）。后续所有特性均通过 *get* 方法请求，该方法仅需一个描述所需特性的参数，编译器将自动选择对应的 *ObjectGet* 函数。

通过额外检查错误代码 4203 (OBJECT_WRONG_PROPERTY)，可以过滤掉不具备所请求特性（例如 OBJPROP_ANCHOR）的对象。通过这种方式，尤其可以实现这样的筛选：所有类型的箭头均会被包含在内（无需分别请求不同类型的 OBJ_ARROW_XYZ），而线条和事件则会被排除在处理之外。

只需先在图表上运行 *ObjectSimpleShowcase.mq5* 脚本（创建 14 种不同类型的对象），然后运行 *ObjectCleanup2.mq5* 即可轻松验证这一点。如果启用 *UseCustomDeleteAll* 模式，图表上将保留 5 个未被删除的对象：OBJ_VLINE、OBJ_HLINE、OBJ_ARROW_BUY、OBJ_ARROW_SELL 和 OBJ_EVENT。前两个和最后一个对象不具备 OBJPROP_ANCHOR 特性，而买卖箭头对象则因颜色不符未被选中（默认其他所有创建的对象颜色均为红色）。

但 *ObjectSelector* 的设计初衷并不仅限于上述简单应用。它是创建单个对象特性监视器的基础，类似于为图表实现的监视器。因此，*ObjectMonitor.mqh* 头文件中包含更值得关注的内容。


```

class ObjectMonitorInterface: public ObjectSelector
{
public:
    ObjectMonitorInterface(const string _id, const long _chart = 0):
        ObjectSelector(_id, _chart) { }
    virtual int snapshot() = 0;
    virtual void print() { };
    virtual int backup() { return 0; }
    virtual void restore() { }
    virtual void applyChanges(ObjectMonitorInterface *reference) { }
};

```

这组方法应该会让人联想到 *ChartModeMonitor.mqh* 中的 *ChartModeMonitorInterface*。唯一的创新是 *applyChanges* 方法，该方法实现了将一个对象的特性复制到另一个对象的功能。

基于 *ObjectMonitorInterface*，以下是针对一对模板类型的特性监视器基本实现的描述：其中包含特性值类型 (*long*、*double* 或 *string* 三者之一) 与枚举类型 `ENUM_OBJECT_PROPERTY_` 风格的枚举之一) 的组合。

```

template<typename T, typename E>
class ObjectMonitorBase: public ObjectMonitorInterface
{
protected:
    MapArray<E,T> data; // array of pairs [property, value], current state
    MapArray<E,T> store; // backup (filled on demand)
    MapArray<E,T> change; // committed changes between two states
    ...

```

ObjectMonitorBase 构造函数包含两个参数：对象名称和标志数组，该数组包含需要在指定对象中观察的特性标识符。这段代码的大部分内容与 *ChartModeMonitor* 几乎完全相同。具体而言，与之前一样，标志数组会被传递给辅助方法 *detect*，该方法的主要目的是识别属于 *E* 枚举元素的整数常量，并剔除所有其他内容。需要说明的唯一新增内容是通过 *ObjectGetInteger(0, id, OBJPROP_LEVELS)* 获取对象中级别数量的特性。这是为了支持因存在级别 (例如 Fibonacci) 而具有多个值的特性迭代所必需的。对于没有级别的对象，我们将获得数量 0，而此类特性将是普通的标量形式。

```

public:
    ObjectMonitorBase(const string _id, const int &flags[]): ObjectMonitorInterface(_id)
    {
        const int levels = (int)ObjectGetInteger(0, id, OBJPROP_LEVELS);
        for(int i = 0; i < ArraySize(flags); ++i)
        {
            detect(flags[i], levels);
        }
    }
    ...

```

当然，*detect* 方法与我们在 *ChartModeMonitor* 中看到的有些不同。首先需要说明的是，它包含一个检查 *v* 常量是否属于 *E* 枚举的代码片段，这是通过调用 *EnumToString* 函数实现的：如果枚举中不存在该元素，将会引发错误代码。如果该元素存在，我们会将对应特性的值添加到 *data* 数组中。

```
// ChartModeMonitor.mqh
bool detect(const int v)
{
    ResetLastError();
    const strings = EnumToString((E)v); // resulting string is not important
    if(_LastError == 0)                // analyze the error code
    {
        data.put((E)v, get((E)v));
        return true;
    }
    return false;
}
```

在对象监视器中，我们不得不将该方案复杂化，因为 *ObjectGet*和 *ObjectSet* 函数中的 *modifier* 参数导致某些特性具有多值性。

因此我们引入了一个静态数组 *modifiabls*，其中列出了修饰符支持的特性（每个特性将在后续详细讨论）。关键在于，对于这类多值特性，需要多次读取并存储到 *data*数组中，而非仅一次。

```
// ObjectMonitor.mqh
bool detect(const int v, const int levels)
{
    // the following properties support multiple values
    static const int modifiabls[] =
    {
        OBJPROP_TIME,           // anchor point by time
        OBJPROP_PRICE,          // anchor point by price
        OBJPROP_LEVELVALUE,     // level value
        OBJPROP_LEVELTEXT,      // inscription on the level line
        // NB: the following properties do not generate errors when exceeded
        // actual number of levels or files
        OBJPROP_LEVELCOLOR,     // level line color
        OBJPROP_LEVELSTYLE,     // level line style
        OBJPROP_LEVELWIDTH,     // width of the level line
        OBJPROP_BMPFILE,        // image files
    };
    ...
}
```

在这里，我们同样使用 *EnumToString*的技巧来检查是否存在具有 *v* 标识符的特性。如果检查成功，我们会确认该特性是否在 *modifiabls*列表中，并将相应的 *modifiable* 标志设为 *true* 或 *false*。

```

    bool result = false;
    ResetLastError();
    conststrings =EnumToString((E)v); // resulting string is not important
    if(_LastError ==0)// analyze the error code
    {
        bool modifiable = false;
        for(int i = 0; i < ArraySize(modifiabls); ++i)
        {
            if(v == modifiabls[i])
            {
                modifiable = true;
                break;
            }
        }
        ...
    }

```

默认情况下，任何特性都被视为无歧义，因此通过 *ObjectGet*函数读取或通过 *ObjectSet* 函数写入的所需次数为 1（即下方的 *k* 变量）。

```

    int k = 1;
    // for properties with modifiers, set the correct amount
    if(modifiable)
    {
        if(levels > 0) k = levels;
        else if(v == OBJPROP_TIME || v == OBJPROP_PRICE) k = MOD_MAX;
        else if(v == OBJPROP_BMPFILE) k = 2;
    }

```

如果对象支持级别，我们会使用 *levels*参数限制潜在的读取/写入次数（如前所述，该参数在调用代码中通过 OBJPROP_LEVELS 特性获取）。

对于 OBJPROP_BMPFILE 特性（我们稍后会详细讨论），它只允许两种状态：开启（按钮按下，标志置位）或关闭（按钮释放，标志清零），因此 *k* = 2。

最后，对象坐标特性 OBJPROP_TIME 和 OBJPROP_PRICE 很便捷，因为在尝试读取/写入不存在的锚点时会产生错误。因此，我们为 *k*分配一个明显较大的值 MOD_MAX，这样就能在 *_LastError* 非零值时中断读取点的循环。

```

    // read property value - one or many
    for(int i = 0; i < k; ++i)
    {
        ResetLastError();
        T temp = get((E)v, i);
        // if there is no i-th modifier, we will get an error and break the loop
        if(_LastError != 0) break;
        data.put((E)MOD_COMBINE(v, i), temp);
        result = true;
    }
}
return result;
}

```

由于一个特性可能有多个值（通过循环读取直至 *k*次），我们不能再简单地编写 *data.put((E)v, get((E)v))*。我们需要以某种方式组合特性标识符 *v*及其修改编号 *i*。所幸的是，特性数量也受限

于整数常量 (type `int`), 最多占用两个低位字节。因此我们可以使用位操作符将 `i` 存入高字节。为此专门开发了 `MOD_COMBINE` 宏。

```
#define MOD_COMBINE(V,I) (V | (I << 24))
```

当然, 也提供了反向宏来提取特性 ID 和修订编号。

```
#define MOD_GET_NAME(V) (V & 0xFFFFF)
#define MOD_GET_INDEX(V) (V >> 24)
```

例如, 在 `snapshot` 方法中可以看到它们的使用方式。

```
virtual int snapshot() override
{
    MapArray<E,T> temp;
    change.reset();

    // collect all required properties in temp
    for(int i = 0; i < data.getSize(); ++i)
    {
        const E e = (E)MOD_GET_NAME(data.getKey(i));
        const int m = MOD_GET_INDEX(data.getKey(i));
        temp.put((E)data.getKey(i), get(e, m));
    }

    int changes = 0;
    // compare previous and new state
    for(int i = 0; i < data.getSize(); ++i)
    {
        if(data[i] != temp[i])
        {
            // save the differences in the change array
            if(changes == 0) Print(id);
            const E e = (E)MOD_GET_NAME(data.getKey(i));
            const int m = MOD_GET_INDEX(data.getKey(i));
            Print(EnumToString(e), (m > 0 ? (string)m : ""), " ", data[i], " -> ", te
            change.put(data.getKey(i), temp[i]);
            changes++;
        }
    }

    // save the new state as current
    data = temp;
    return changes;
}
```

该方法复用了 `ChartModeMonitor.mqh` 中同名方法的全部逻辑, 但在任何位置读取特性时, 必须首先通过 `MOD_GET_NAME` 从存储的键中提取特性名称, 并通过 `MOD_GET_INDEX` 提取编号。

`restore` 方法也不得不进行类似的复杂操作。

```

virtual void restore() override
{
    data = store;
    for(int i = 0; i < data.getSize(); ++i)
    {
        const E e = (E)MOD_GET_NAME(data.getKey(i));
        const int m = MOD_GET_INDEX(data.getKey(i));
        set(e, data[i], m);
    }
}

```

*ObjectMonitorBase*最具创新性的特性在于其如何处理变更。

```

MapArray<E,T> * const getChanges()
{
    return &change;
}

virtual void applyChanges(ObjectMonitorInterface *intf) override
{
    ObjectMonitorBase *reference = dynamic_cast<ObjectMonitorBase<T,E> *>(intf);
    if(reference)
    {
        MapArray<E,T> *event = reference.getChanges();
        if(event.getSize() > 0)
        {
            Print("Modifying ", id, " by ", event.getSize(), " changes");
            for(int i = 0; i < event.getSize(); ++i)
            {
                data.put(event.getKey(i), event[i]);
                const E e = (E)MOD_GET_NAME(event.getKey(i));
                const int m = MOD_GET_INDEX(event.getKey(i));
                Print(EnumToString(e), " ", m, " ", event[i]);
                set(e, event[i], m);
            }
        }
    }
}

```

通过向 *applyChanges*方法传递另一个对象的监控器状态，我们可以采用该对象的所有最新变更。

为支持所有三种基本类型 (*long*、*double*、*string*)的特性，我们需要实现 *ObjectMonitor* 类 (相当于 *ChartModeMonitor.mqh* 中的 *ChartModeMonitor*)。

```

class ObjectMonitor: public ObjectMonitorInterface
{
protected:
    AutoPtr<ObjectMonitorInterface> m[3];

    ObjectMonitorInterface *getBase(const int i)
    {
        return m[i][];
    }

public:
    ObjectMonitor(const string objid, const int &flags[]): ObjectMonitorInterface(objid)
    {
        m[0] = new ObjectMonitorBase<long,ENUM_OBJECT_PROPERTY_INTEGER>(objid, flags);
        m[1] = new ObjectMonitorBase<double,ENUM_OBJECT_PROPERTY_DOUBLE>(objid, flags);
        m[2] = new ObjectMonitorBase<string,ENUM_OBJECT_PROPERTY_STRING>(objid, flags);
    }
    ...

```

此处保留了之前的代码结构，仅新增了支持变更和名称的方法（我们应该还记得，图表本身不具备名称）。

```

...
virtual string name() override
{
    return m[0][].name();
}

virtual void name(const string objid) override
{
    m[0][].name(objid);
    m[1][].name(objid);
    m[2][].name(objid);
}

virtual void applyChanges(ObjectMonitorInterface *intf) override
{
    ObjectMonitor *monitor = dynamic_cast<ObjectMonitor *>(intf);
    if(monitor)
    {
        m[0][].applyChanges(monitor.getBase(0));
        m[1][].applyChanges(monitor.getBase(1));
        m[2][].applyChanges(monitor.getBase(2));
    }
}

```

基于已创建的对象监视器，可以轻松实现终端中不支持的多项实用功能。特别是对象副本创建和对象批量编辑功能。

ObjectCopy 脚本

*ObjectCopy.mq5*脚本演示了如何复制选中的对象。在其 *OnStart*函数起始阶段，我们用连续整数填充 *flags* 数组，这些整数是各类 `ENUM_OBJECT_PROPERTY_` 枚举元素的候选值。枚举元素的编

号按功能用途明显分组，且组间存在较大间隔（显然是为未来元素预留空间），因此形成的数组相当大：2048 个元素。

```
#include <MQL5Book/ObjectMonitor.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
    int flags[2048];
    // filling the array with consecutive integers, which will be
    // checked against the elements of enumerations of object properties,
    // invalid values will be discarded in the monitor's detect method
    for(int i = 0; i < ArraySize(flags); ++i)
    {
        flags[i] = i;
    }
    ...
}
```

接下来，我们将图表当前选中的对象名称收集到数组中。为此，我们使用 OBJPROP_SELECTED 特性。

```
string selected[];
const int n = ObjectsTotal(0);
for(int i = 0; i < n; ++i)
{
    const string name = ObjectName(0, i);
    if(ObjectGetInteger(0, name, OBJPROP_SELECTED))
    {
        PUSH(selected, name);
    }
}
...
```

最终，在遍历选中元素的主循环中，我们依次读取每个对象的特性、生成其副本名称并创建具有相同特性集合的新对象。

```

for(int i = 0; i < ArraySize(selected); ++i)
{
    const string name = selected[i];

    // make a backup of the properties of the current object using the monitor
    ObjectMonitor object(name, flags);
    object.print();
    object.backup();
    // form a correct, appropriate name for the copy
    const string copy = GetFreeName(name);

    if(StringLen(copy) > 0)
    {
        Print("Copy name: ", copy);
        // create an object of the same type OBJPROP_TYPE
        ObjectCreate(0, copy,
            (ENUM_OBJECT)ObjectGetInteger(0, name, OBJPROP_TYPE),
            ObjectFind(0, name), 0, 0);
        // change the name of the object in the monitor to a new one
        object.name(copy);
        // restore all properties from the backup to a new object
        object.restore();
    }
    else
    {
        Print("Can't create copy name for: ", name);
    }
}
}

```

这里需要注意，OBJPROP_TYPE 特性是少数只读特性之一，因此必须首先创建所需类型的对象。

辅助函数 *GetFreeName* 尝试在对象名后追加 “/Copy #x” 字符串，其中 x 为副本编号。因此，通过多次运行脚本，可创建第 2 个、第 3 个等副本。


```

string GetFreeName(const string name)
{
    const string suffix = "/Copy №";
    // check if there is a copy in the suffix name
    const int pos = StringFind(name, suffix);
    string prefix;
    int n;

    if(pos <= 0)
    {
        // if suffix is not found, assume copy number 1
        const string candidate = name + suffix + "1";
        // checking if the copy name is free, and if so, return it
        if(ObjectFind(0, candidate) < 0)
        {
            return candidate;
        }
        // otherwise, prepare for a loop with iteration of copy numbers
        prefix = name;
        n = 0;
    }
    else
    {
        // if the suffix is found, select the name without it
        prefix = StringSubstr(name, 0, pos);
        // and find the copy number in the string
        n = (int)StringToInteger(StringSubstr(name, pos + StringLen(suffix)));
    }

    Print("Found: ", prefix, " ", n);
    // loop trying to find a free copy number above n, but no more than 1000
    for(int i = n + 1; i < 1000; ++i)
    {
        const string candidate = prefix + suffix + (string)i;
        // check for the existence of an object with a name ending "Copy #i"
        if(ObjectFind(0, candidate) < 0)
        {
            return candidate; // return vacant copy name
        }
    }
    return NULL; // too many copies
}

```

终端会记住特定类型对象的最后设置，如果连续创建多个同类对象，其效果等同于复制。然而在使用不同图表的过程中，这些设置通常会发生变化，如果一段时间后需要复制某些“旧”对象，通常需要完全重新配置其设置。这对于特性数量庞大的对象类型（例如 Fibonacci 工具）尤其耗时。在这种情况下，该脚本将发挥重要作用。

本章中某些包含相同类型对象的图片，正是使用该脚本创建的。

ObjectGroupEdit indicator

*ObjectMonitor*的第二个应用示例是 *ObjectGroupEdit.mq5* 指标，该指标支持一次性编辑一组选定对象的特性。

假设我们在图表上选中了多个对象（不一定是相同类型），需要统一修改其某个特性。接下来，打开任意选中对象的特性对话框进行配置，点击 *OK* 后，这些更改将应用于所有选中对象。这就是我们下一个 MQL 程序的运行方式。

我们需要将指标作为程序类型，因为它涉及图表事件。关于 MQL5 编程的这一方面，后续将有整个专门的 章节 详述，但我们现在就需要掌握部分基础知识。

由于指标没有图表，*#property* 指令设为零，且 *OnCalculate* 函数几乎为空。

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots 0

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &price[])
{
    return rates_total;
}
```

为了自动生成对象的完整特性集，我们将再次使用包含 2048 个连续整数值的数组。我们还将提供一个用于存储所选元素名称的数组，以及一个由 *ObjectMonitor* 类组成的监控对象数组。

```
int consts[2048];
string selected[];
ObjectMonitor *objects[];
```

在 *OnInit* 处理程序中，我们将初始化数字数组并启动计时器。

```
void OnInit()
{
    for(int i = 0; i < ArraySize(consts); ++i)
    {
        consts[i] = i;
    }

    EventSetTimer(1);
}
```

在计时器处理程序中，我们将所选对象的名称保存到数组中。如果选择列表发生变化，则需要重新配置监控对象，此时会调用辅助函数 *TrackSelectedObjects*。

```

void OnTimer()
{
    string updates[];
    const int n = ObjectsTotal(0);
    for(int i = 0; i < n; ++i)
    {
        const string name = ObjectName(0, i);
        if(ObjectGetInteger(0, name, OBJPROP_SELECTED))
        {
            PUSH(updates, name);
        }
    }

    if(ArraySize(selected) != ArraySize(updates))
    {
        ArraySwap(selected, updates);
        Comment("Selected objects: ", ArraySize(selected));
        TrackSelectedObjects();
    }
}

```

TrackSelectedObjects函数本身非常简单：删除旧的监控对象并创建新的监控对象。如果你需要，可以通过保留选择中未变动的部分，使其更加智能化。

```

void TrackSelectedObjects()
{
    for(int j = 0; j < ArraySize(objects); ++j)
    {
        delete objects[j];
    }

    ArrayResize(objects, 0);

    for(int i = 0; i < ArraySize(selected); ++i)
    {
        const string name = selected[i];
        PUSH(objects, new ObjectMonitor(name, consts));
    }
}

```

请注意，当创建监控对象时，它会立即捕获对应图形对象的所有特性。

现在我们终于要讲到事件处理的部分。正如在 [事件函数概述](#) 中已经提到的，该处理程序负责处理图表上的 **OnChartEvent** 事件。在本示例中，我们重点关注特定的 `CHARTEVENT_OBJECT_CHANGE` 事件：当用户在对象特性对话框中更改任何特性时，该事件会被触发。被修改对象的名称通过 *sparam* 参数传递。

如果该名称与某个被监控对象匹配，我们会要求监控器对其特性生成新的快照，即调用 `objects[i].snapshot()`。

```

void OnChartEvent(const int id,
    const long &lparam, const double &dparam, const string &sparam)
{
    if(id == CHARTEVENT_OBJECT_CHANGE)
    {
        Print("Object changed: ", sparam);
        for(int i = 0; i < ArraySize(selected); ++i)
        {
            if(sparam == selected[i])
            {
                const int changes = objects[i].snapshot();
                if(changes > 0)
                {
                    for(int j = 0; j < ArraySize(objects); ++j)
                    {
                        if(j != i)
                        {
                            objects[j].applyChanges(objects[i]);
                        }
                    }
                }
                ChartRedraw();
                break;
            }
        }
    }
}

```

如果确认存在变更（一般都会有变更），则 *changes* 变量中的数量将大于 0。然后，开始遍历所有选中的对象，并将检测到的变更应用到每个对象，但原始对象除外。

由于我们可能需要修改多个对象，因此需要通过 *ChartRedraw* 请求图表重绘。

在 *OnDeinit* 处理程序中，我们将移除所有监控器。

```

void OnDeinit(const int)
{
    for(int j = 0; j < ArraySize(objects); ++j)
    {
        delete objects[j];
    }
    Comment("");
}

```

至此，新工具已准备就绪。

该指标允许你针对 [定义对象锚点](#) 章节中多组标签对象的通用外观进行自定义。

值得一提的是，基于 *ObjectMonitor* 的类似原理，还可以创建终端中尚未提供的另一个常用工具：撤销对对象特性的编辑操作，因为 *restore* 方法现在可供使用。

5.8.8 主要对象特性

所有对象都有一些通用特性。主要特性如下表所示。我们将在后续内容中介绍其他通用的特殊用途特性 (请参阅 [对象状态管理](#)、[Z 轴顺序](#)和 [时间范围上下文中的对象可见性](#) 章节)。

标识符	说明	类型
OBJPROP_NAME	对象名称	string
OBJPROP_TYPE	对象类型 (只读)	ENUM_OBJECT
OBJPROP_CREATETIME	对象创建时间 (只读)	datetime
OBJPROP_TEXT	对象说明 (对象中包含的文本)	string
OBJPROP_TOOLTIP	鼠标悬停提示文本	string

OBJPROP_NAME 特性是对象标识符。编辑该特性相当于删除旧对象并创建新对象。

对于某些能够显示文本的对象类型 (例如标签或按钮), OBJPROP_TEXT 特性会直接显示在图表中的对象内部。对于其他对象 (例如线条), 该特性包含的描述信息仅在图表设置中启用 “显示对象描述选项”时, 才会显示在图表上对象的旁边。无论何种情况, OBJPROP_TEXT 都会显示在工具提示中。

OBJPROP_CREATETIME 特性仅在当前会话期间有效, 不会写入 chr 文件。

可以通过编程或手动方式 (在对象特性对话框中)更改对象名称, 而其创建时间将保持不变。预先说明, 编程重命名不会在图表上触发任何与对象相关的事件。正如我们将在 [下一章](#)中学到的, 手动重命名会触发三个事件:

- 删除旧名称的对象 (CHARTEVENT_OBJECT_DELETE)
- 创建新名称的对象 (CHARTEVENT_OBJECT_CREATE)
- 修改新对象 (CHARTEVENT_OBJECT_CHANGE)

如果未设置 OBJPROP_TOOLTIP 特性, 终端会自动为对象生成工具提示。要禁用工具提示, 将其值设置为 “\n” (换行符)。

我们将对 [查找对象](#) 章节中的 *ObjectFinder.mq5*脚本进行调整, 用于记录当前图表中所有对象的上述特性。我们将新脚本命名为 *ObjectListing.mq5*。

在 *OnStart*的起始阶段, 我们将创建或修改一条位于最后一根柱线 (即脚本启动时)的垂直直线。如果图表设置中启用了显示对象描述的选项, 则我们将在右侧垂直线旁看到 “当前最新柱线”文本。

```
void OnStart()
{
    const string vline = ObjNamePrefix + "current";
    ObjectCreate(0, vline, OBJ_VLINE, 0, iTime(NULL, 0, 0), 0);
    ObjectSetString(0, vline, OBJPROP_TEXT, "Latest Bar At The Moment");
    ...
}
```

接下来, 通过遍历子窗口的循环, 我们将查询最多 *ObjectsTotal*个对象及其主要特性。

```
int count = 0;
const long id = ChartID();
const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
// loop through subwindows
for(int k = 0; k < win; ++k)
{
    PrintFormat(" Window %d", k);
    const int n = ObjectsTotal(id, k);
    //loop through objects
    for(int i = 0; i < n; ++i)
    {
        const string name = ObjectName(id, i, k);
        const ENUM_OBJECT type =
            (ENUM_OBJECT)ObjectGetInteger(id, name, OBJPROP_TYPE);
        const datetime created =
            (datetime)ObjectGetInteger(id, name, OBJPROP_CREATETIME);
        const string description = ObjectGetString(id, name, OBJPROP_TEXT);
        const string hint = ObjectGetString(id, name, OBJPROP_TOOLTIP);
        PrintFormat("    %s %s %s %s %s", EnumToString(type), name,
            TimeToString(created), description, hint);
        ++count;
    }
}

PrintFormat("%d objects found", count);
}
```

日志中会生成以下条目。

```
Window 0
OBJ_VLINE ObjShow-current 2021.12.21 20:20 Latest Bar At The Moment
OBJ_VLINE abc 2021.12.21 19:25
OBJ_VLINE xyz 1970.01.01 00:00
3 objects found
```

OBJPROP_CREATETIME 值为零 (1970.01.01 00:00)表示该对象并非在当前会话期间创建，而是在更早时间创建的。

5.8.9 价格和时间坐标

对于在报价坐标系中存在的对象类型，MQL5 API 支持多个用于指定时间和价格绑定的特性。如果对象有多个锚点，则在调用 *ObjectSet*和 *ObjectGet* 函数时，相关特性需要指定一个包含锚点索引的修饰符参数。

标识符	说明	值类型
OBJPROP_TIME	时间坐标	datetime
OBJPROP_PRICE	价格坐标	double

这些特性当然适用于所有对象，但为使用 `屏幕坐标` 的对象设置或读取这些特性并无实际意义。

为演示如何使用坐标，我们来分析无缓冲指标 *ObjectHighLowChannel.mq5*。对于给定的柱线区间，该指标将绘制两条趋势线。它们在时间轴上的起点和终点与该区间的第一根和最后一根柱线重合，而在价格轴上，每条线的值计算方式不同：上线使用最 *High* 价的最高值和最低值，下线使用最 *Low* 价的最高值和最低值。随着图表更新，我们临时绘制的通道应随价格移动。

柱线范围通过两个输入变量设置：初始柱线编号 *BarOffset* 和柱线数量 *BarCount*。默认情况下，线条基于最新价格绘制，因为 *bar offset = 0*。

```
input int BarOffset = 0;
input int BarCount = 10;

const string Prefix = "HighLowChannel-";
```

对象具有共同的名称前缀 "HighLowChannel"。

在 *OnCalculate* 处理程序中，我们通过第 0 根柱线的 *iTime* 时间监控新柱线的出现。一旦柱线形成，将分析指定区间内的价格，提取两种价格类型 (*MODE_HIGH* 和 *MODE_LOW*) 各自的最大值和最小值，然后调用辅助函数 *DrawFigure*，该函数负责对象相关操作：创建对象和修改坐标。

```
int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &price[])
{
    static datetime now = 0;
    if(now != iTime(NULL, 0, 0))
    {
        const int hh = iHighest(NULL, 0, MODE_HIGH, BarCount, BarOffset);
        const int lh = iLowest(NULL, 0, MODE_HIGH, BarCount, BarOffset);
        const int ll = iLowest(NULL, 0, MODE_LOW, BarCount, BarOffset);
        const int hl = iHighest(NULL, 0, MODE_LOW, BarCount, BarOffset);

        datetime t[2] = {iTime(NULL, 0, BarOffset + BarCount), iTime(NULL, 0, BarOffset);
        double ph[2] = {iHigh(NULL, 0, fmax(hh, lh)), iHigh(NULL, 0, fmin(hh, lh))};
        double pl[2] = {iLow(NULL, 0, fmax(ll, hl)), iLow(NULL, 0, fmin(ll, hl))};

        DrawFigure(Prefix + "Highs", t, ph, clrBlue);
        DrawFigure(Prefix + "Lows", t, pl, clrRed);

        now = iTime(NULL, 0, 0);
    }
    return rates_total;
}
```

DrawFigure 函数如下。

```

bool DrawFigure(const string name, const datetime &t[], const double &p[],
    const color clr)
{
    if(ArraySize(t) != ArraySize(p)) return false;

    ObjectCreate(0, name, OBJ_TREND, 0, 0, 0);

    for(int i = 0; i < ArraySize(t); ++i)
    {
        ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
        ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
    }

    ObjectSetInteger(0, name, OBJPROP_COLOR, clr);
    return true;
}

```

在调用 *ObjectCreate* 函数确保对象存在后,将在所有锚点 (本例为两个)处调用适用于 OBJPROP_TIME 和 OBJPROP_PRICE 的 *ObjectSet* 函数。

下图展示了该指标的运行结果:



基于最高价和最低价的两条趋势线构成的通道
可以在可视化测试程序中运行该指标,查看线条坐标如何实时变化。

5.8.10 锚定窗口角点和屏幕坐标

对于采用图表（像素）点坐标系的对象，必须选择窗口四个角之一作为参考点，以此作为计算对象锚点在水平 X 轴与垂直 Y 轴方向的坐标值的基准。这些方面由下表中的特性控制。

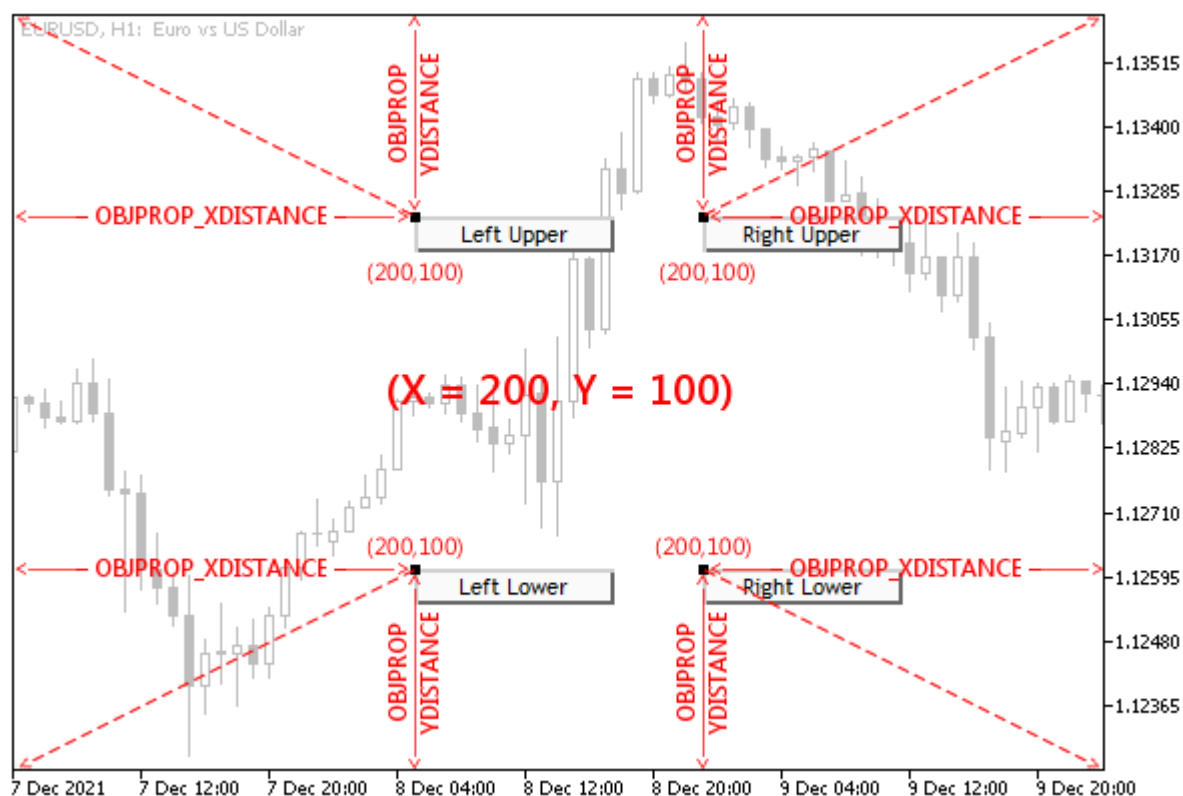
标识符	说明	类型
OBJPROP_CORNER	图形对象锚定的图表角点	ENUM_BASE_CORNER
OBJPROP_XDISTANCE	距锚定角点 X 轴方向的距离，以像素为单位	int
OBJPROP_YDISTANCE	距锚定角点 Y 轴方向的距离，以像素为单位	int

OBJPROP_CORNER 的有效选项汇总在 ENUM_BASE_CORNER 枚举中。

标识符	坐标中心位置
CORNER_LEFT_UPPER	窗口左上角
CORNER_LEFT_LOWER	窗口左下角
CORNER_RIGHT_LOWER	窗口右下角
CORNER_RIGHT_UPPER	窗口右上角

默认值为左上角。

下图显示了四个按钮对象，在窗口中，它们具有相同的大小，且各自距锚定角点的距离相同。这些对象仅在绑定的角点本身有所不同。需注意，按钮有一个锚点，其始终位于按钮的左上角。

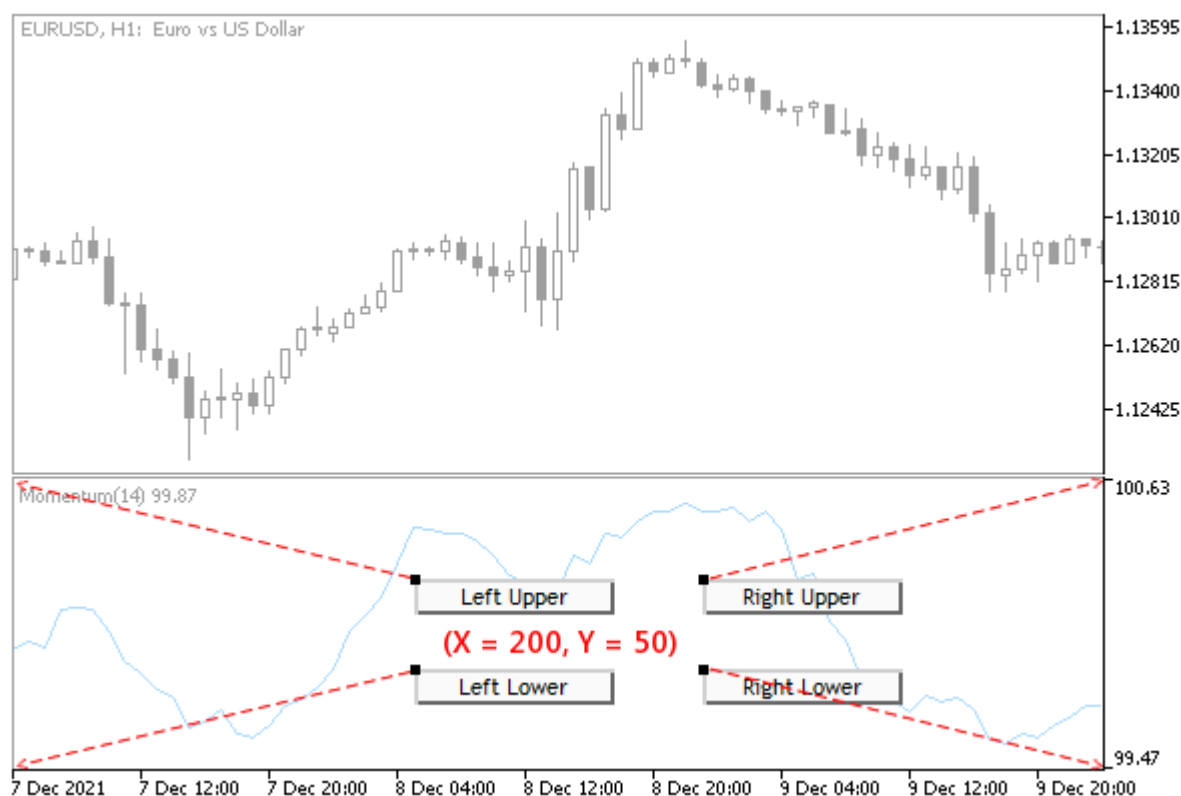


绑定到主窗口不同角点的对象布局

图表中所有四个对象当前均被选中，因此其锚点以对比色高亮显示。

当我们提及窗口角点时，指的是对象所在的特定窗口或子窗口，而非整个图表。换言之，对于子窗口中的对象，Y坐标是从该子窗口的上边框或下边框开始测量的。

下图展示了子窗口中绑定到子窗口各角点的类似对象。



绑定到主窗口不同角点的对象位置

用户可使用 `ObjectCornerLabel.mq5` 脚本测试文本标注的移动，其窗口锚定角通过输入参数 `Corner` 指定。

```
#property script_show_inputs
```

```
input ENUM_BASE_CORNER Corner = CORNER_LEFT_UPPER;
```

坐标会周期性变化，并显示在标注文本中。因此，标注会在窗口中移动，当移至边框时将产生反弹效果。对象将在鼠标放置脚本的窗口或子窗口内创建。

```
void OnStart()
{
    const int t = ChartWindowOnDropped();
    const string legend = EnumToString(Corner);

    const string name = "ObjCornerLabel-" + legend;
    int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t);
    int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
    int x = w / 2;
    int y = h / 2;
    ...
}
```

为正确定位，我们获取窗口的尺寸（然后检查其是否已更改），并找到对象初始放置的中间位置：使用坐标变量 `x` 和 `y`。

接下来，我们将创建并设置一个文本标注，暂不设置坐标。需要注意的是，我们启用了选择对象选择功能（`OBJPROP_SELECTABLE`）并将其选中（`OBJPROP_SELECTED`），因为这能让我们看到对象本身的锚点，从而测量其与窗口角点（坐标远点）的距离。这两个特性在 [对象状态管理](#) 章节有更详细的说明。

```

ObjectCreate(0, name, OBJ_LABEL, t, 0, 0);
ObjectSetInteger(0, name, OBJPROP_SELECTABLE, true);
ObjectSetInteger(0, name, OBJPROP_SELECTED, true);
ObjectSetInteger(0, name, OBJPROP_CORNER, Corner);
...

```

在变量 `px`和`py`, 中, 我们将记录用于模拟运动的坐标增量。坐标修改本身将在无限循环中执行, 直至被用户中断。迭代计数器允许每 50 次迭代时, 随机改变移动方向。

```

int px = 0, py = 0;
int pass = 0;

for( ;!IsStopped(); ++pass)
{
    if(pass % 50 == 0)
    {
        h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t);
        w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
        px = rand() * (w / 20) / 32768 - (w / 40);
        py = rand() * (h / 20) / 32768 - (h / 40);
    }

    // bounce off window borders so object doesn't hide
    if(x + px > w || x + px < 0) px = -px;
    if(y + py > h || y + py < 0) py = -py;
    // recalculate label positions
    x += px;
    y += py;

    // update the coordinates of the object and add them to the text
    ObjectSetString(0, name, OBJPROP_TEXT, legend
        + "[" + (string)x + "," + (string)y + "]");
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);

    ChartRedraw();
    Sleep(100);
}

ObjectDelete(0, name);
}

```

尝试多次运行该脚本, 并指定不同的锚定角点。

在下一节中, 我们将通过控制对象上的锚点来增强该脚本。

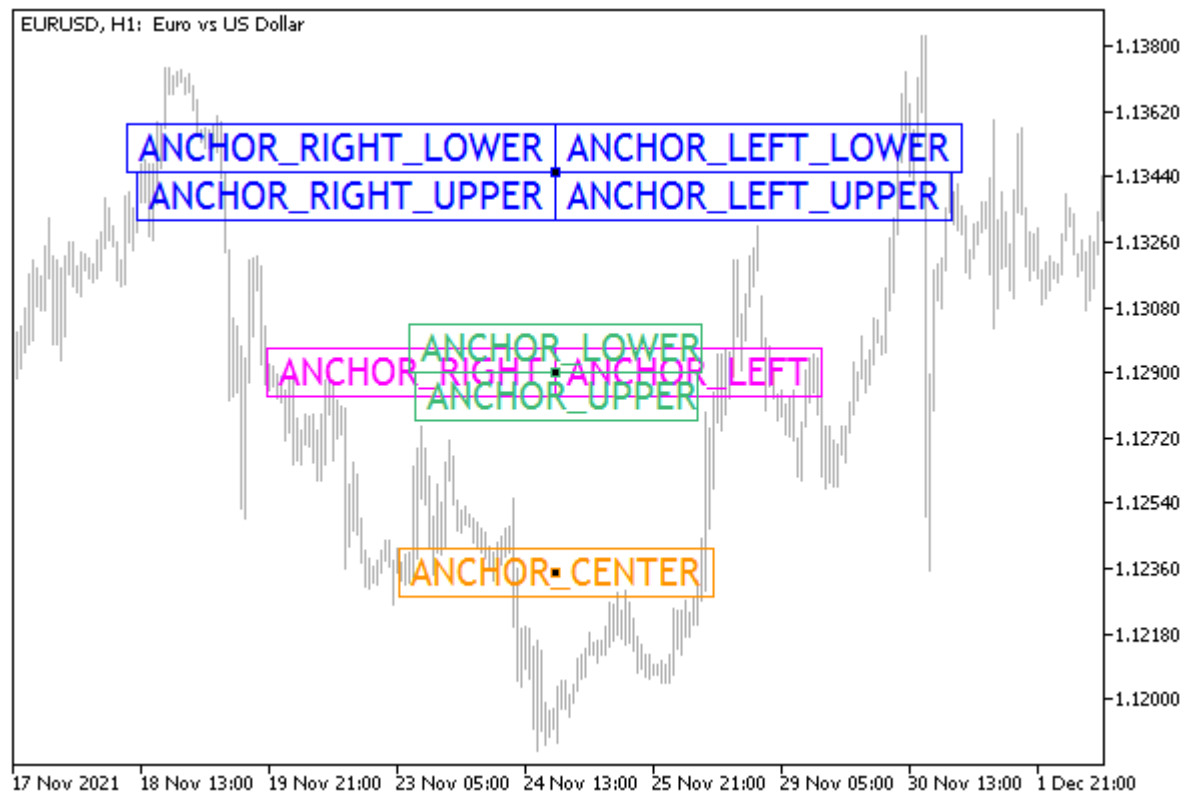
5.8.11 在对象上定义锚点

某些类型的对象允许选择锚点。属于此范畴的对象类型包括与报价关联的文本标签 (`OBJ_TEXT`) 和位图图像 (`OBJ_BITMAP`), 以及按屏幕坐标定位的标题 (`OBJ_LABEL`) 和带图像的面板 (`OBJ_BITMAP_LABEL`)。

要读取和设置锚点，可使用带有 OBJPROP_ANCHOR 特性的 *ObjectGetInteger*和 *ObjectSetInteger*。所有锚点选择选项均包含在 ENUM_ANCHOR_POINT 枚举中。

标识符	锚点位置
ANCHOR_LEFT_UPPER	左上角
ANCHOR_LEFT	左侧居中
ANCHOR_LEFT_LOWER	左下角
ANCHOR_LOWER	底部居中
ANCHOR_RIGHT_LOWER	右下角
ANCHOR_RIGHT	右侧居中
ANCHOR_RIGHT_UPPER	右上角
ANCHOR_UPPER	顶部居中
ANCHOR_CENTER	对象正中心

这些锚点在下图中清晰可见，图中的图表应用了多个标签对象。



具有不同锚点的 OBJ_LABEL 文本对象

上方的四个标签组具有相同的坐标对 (X,Y)，但由于锚定到对象的不同角落，它们位于该点的不同侧面。第二组四个文本标签的情况类似，但它们锚定到对象不同侧面的中点。最后，底部单独显示的标题锚定在其中心位置，因此该点位于对象内部。

按钮 (OBJ_BUTTON)、矩形面板 (OBJ_RECTANGLE_LABEL)、输入字段 (OBJ_EDIT) 和图表对象 (OBJ_CHART) 的锚点固定在左上角 (ANCHOR_LEFT_UPPER)。

单价格标记组中的某些图形对象 OBJ_ARROW、OBJ_ARROW_THUMB_UP、OBJ_ARROW_THUMB_DOWN、OBJ_ARROW_UP、OBJ_ARROW_DOWN、OBJ_ARROW_STOP 和 OBJ_ARROW_CHECK) 具有两种坐标锚定方式，这些方式由另一个枚举类型 ENUM_ARROW_ANCHOR 的标识符指定。

标识符	锚点位置
ANCHOR_TOP	顶部居中
ANCHOR_BOTTOM	底部居中

该组其余对象具有预定义锚点：买入箭头 (OBJ_ARROW_BUY) 和卖出箭头 (OBJ_ARROW_SELL) 分别位于上边缘和下边缘的中间位置，而价格标签 OBJ_ARROW_RIGHT_PRICE 和 OBJ_ARROW_LEFT_PRICE) 则分别位于左侧和右侧。

与上一节的 *ObjectCornerLabel.mq5* 脚本类似，我们将创建 *ObjectAnchorLabel.mq5* 脚本。在新版本中，除移动标注外，我们还将随机更改其中的锚点。

与之前一样，窗口锚定角点仍将由用户在脚本启动时选择。

```
input ENUM_BASE_CORNER Corner = CORNER_LEFT_UPPER;
```

我们将在图表上以注释形式显示该角点的名称。

```
void OnStart()
{
    Comment(EnumToString(Corner));
    ...
}
```

在无限循环中，将在选定时间生成 9 种可能的锚点值之一。

```
ENUM_ANCHOR_POINT anchor = 0;
for( ;!IsStopped(); ++pass)
{
    if(pass % 50 == 0)
    {
        ...
        anchor = (ENUM_ANCHOR_POINT)(rand() * 9 / 32768);
        ObjectSetInteger(0, name, OBJPROP_ANCHOR, anchor);
    }
    ...
}
```

锚点名称将与当前坐标一并成为标签的文本内容。

```
ObjectSetString(0, name, OBJPROP_TEXT, EnumToString(anchor)
    + "[" + (string)x + "," + (string)y + "]");
```

其余代码片段基本保持不变。

编译并运行脚本后，请注意标注如何根据所选锚点相对于当前坐标 (x, y) 改变位置。

目前，我们已能控制锚点，并可防止其本身超出窗口范围。但对象具有一定尺寸，因此可能出现标注大部分内容被被截断的情况。在未来学习相关特性后，我们将解决此问题（请参阅 [确定对象宽度和高度](#) 章节中的 *ObjectSizeLabel.mq5* 示例）。

5.8.12 管理对象状态

在对象的通用特性中，有多个特性用于控制对象的状态。所有这类特性均为布尔类型，这意味着它们可以被启用 (*true*) 或禁用 (*false*)，因此需要使用 *ObjectGetInteger* 和 *ObjectSetInteger* 函数。

标识符	说明
OBJPROP_HIDDEN	禁用在相关对话框的对象列表中显示图形对象的名称（可通过图表上下文菜单调用或按 Ctrl+B 触发该对话框）。
OBJPROP_SELECTED	对象选择
OBJPROP_SELECTABLE	对象可选性

如果 OBJPROP_HIDDEN 的值为 *true*，则允许从用户列表中隐藏不必要的对象。默认情况下，对于显示日历事件、交易历史的对象以及通过 MQL 程序创建的对象，会设置为 *true*。如果要查看此类图形对象并访问其特性，请在 *Object List* 对话框中按下 *All* 按钮。

列表中隐藏的对象在图表上仍保持可见。要在不删除对象的情况下将对象从图表中隐藏，可使用 [对象在时间范围下文中的可见性](#) 设置。

如果对象的 OBJPROP_SELECTABLE 的值为 *false*，用户将无法选择和修改这些对象的特性。通过编程方式创建的对象默认禁止被选中。正如我们在前面章节的 *ObjectCornerLabel.mq5* 和 *ObjectAnchorLabel.mq5* 脚本中所见，必须显式将 OBJPROP_SELECTABLE 设置为 *true*，才能解锁包含 OBJPROP_SELECTED 特性的能力。我们正是通过这种方式来高亮显示对象上的锚点。

通常，仅当 MQL 程序的对象用作控件时，这些程序才允许选择这些对象。例如，具有预定义名称的趋势线（用户可随意移动），可能意味着当价格突破该趋势线时发送交易订单的条件。

5.8.13 对象优先级（Z 轴顺序）

图表上的对象不仅提供信息展示，还能通过事件与用户及 MQL 程序进行交互，我们将在 [下一章](#) 对此进行详细探讨。其中一个事件源是鼠标指针。具体来说，图表能够追踪鼠标移动和按键操作。

如果某个对象位于鼠标下方，则可以针对该对象执行特定的事件处理。然而，对象可能会相互重叠（当它们的坐标在考虑 [尺寸](#) 的情况下重叠时）。在这种情况下，整数特性 OBJPROP_ZORDER 开始发挥作用。该特性用于设置图形对象接收鼠标事件的优先级。当多个对象重叠时，只有优先级最高的对象才能接收到事件。

默认情况下，创建对象时，其 Z 轴顺序为零，但可根据需要增大该值。

需特别注意，Z 轴顺序仅影响鼠标事件的处理顺序，而不影响对象的绘制顺序。对象始终按照其添加到图表的先后顺序进行绘制。这一点可能会导致误解。例如，当一个对象在视觉上位于另一个对象之上时，可能因被重叠对象的 Z 轴优先级更高（见示例）而无法显示其工具提示。

在 *ObjectZorder.mqh* 脚本中，我们将创建 12 个 OBJ_RECTANGLE_LABEL 类型的对象，并按类似钟面将其排列成圆形。对象的添加顺序对应小时刻度：从 1 到 12。为清晰起见，所有矩形将设置随机颜色（关于 OBJPROP_BGCOLOR 特性，请参阅 下一节）和随机优先级。当用户将鼠标悬停在对象上时，可通过工具提示确定当前所属对象。

为方便设置对象特性，我们定义了从 *ObjectSelector* 派生的特殊类 *ObjectBuilder*。

```
#include "ObjectPrefix.mqh"
#include <MQL5Book/ObjectMonitor.mqh>

class ObjectBuilder: public ObjectSelector
{
protected:
    const ENUM_OBJECT type;
    const int window;
public:
    ObjectBuilder(const string _id, const ENUM_OBJECT _type,
        const long _chart = 0, const int _win = 0):
        ObjectSelector(_id, _chart), type(_type), window(_win)
    {
        ObjectCreate(host, id, type, window, 0, 0);
    }

    // changing the name and chart is prohibited
    virtual void name(const string _id) override = delete;
    virtual void chart(const long _chart) override = delete;
};
```

对象标识符 (*id*) 和图表标识符 (*host*) 的字段已存在于 *ObjectSelector* 类中。在派生类中，我们添加了对对象类型 (*ENUM_OBJECT type*) 和窗口编号 (*int window*)。构造函数会调用 *ObjectCreate*。

特性的设置和读取完全继承自 *ObjectSelector* 中的一组 *get* 和 *set* 方法。

与之前的测试脚本一样，我们将确定脚本放置的窗口、窗口尺寸和中心点坐标。

```
void OnStart()
{
    const int t = ChartWindowOnDropped();
    int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t);
    int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
    int x = w / 2;
    int y = h / 2;
    ...
}
```

由于 OBJ_RECTANGLE_LABEL 对象类型支持显式像素尺寸，因此我们将每个矩形的宽度 *dx* 和高度 *dy* 计算为窗口的四分之一。我们使用这些值来设置 OBJPROP_XSIZE 和 OBJPROP_YSIZE 特性（相关说明请参阅 确定对象宽度和高度 章节）。


```
const int dx = w / 4;
const int dy = h / 4;
...
```

接下来，在循环中创建 12 个对象。变量 *px* 和 *py* 包含“表盘”上下一个“标记”相对于中心点 (x, y) 的偏移量。*z* 轴优先级采用随机选择。对象名称及其工具提示 (OBJPROP_TOOLTIP) 包含类似 “XX - YYY” 的字符串，其中 XX 表示“小时”编号 (表盘上的位置 1 至 12)，YYY 表示优先级值。

```
for(int i = 0; i < 12; ++i)
{
    const int px = (int)(MathSin((i + 1) * 30 * M_PI / 180) * dx) - dx / 2;
    const int py = -(int)(MathCos((i + 1) * 30 * M_PI / 180) * dy) - dy / 2;

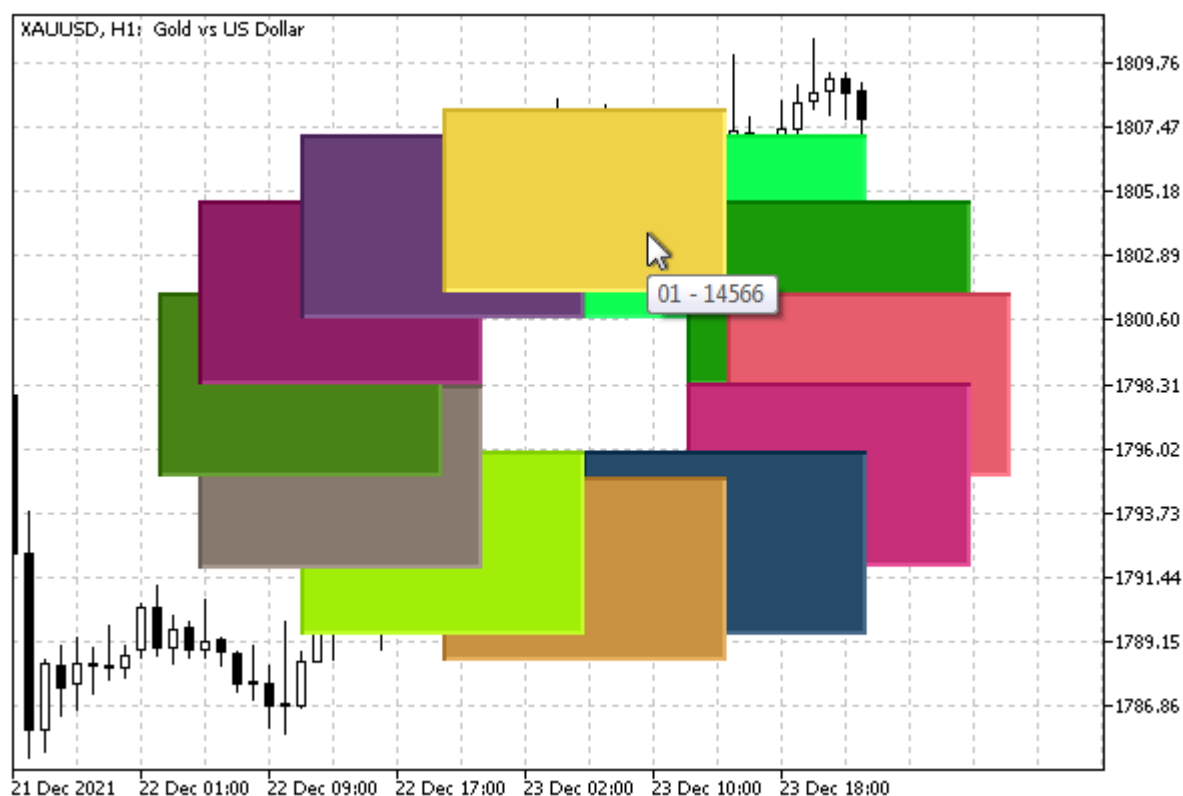
    const int z = rand();
    const string text = StringFormat("%02d - %d", i + 1, z);

    ObjectBuilder *builder =
        new ObjectBuilder(ObjNamePrefix + text, OBJ_RECTANGLE_LABEL);
    builder.set(OBJPROP_XDISTANCE, x + px).set(OBJPROP_YDISTANCE, y + py)
        .set(OBJPROP_XSIZE, dx).set(OBJPROP_YSIZE, dy)
        .set(OBJPROP_TOOLTIP, text)
        .set(OBJPROP_ZORDER, z)
        .set(OBJPROP_BGCOLOR, (rand() << 8) | rand());
    delete builder;
}
```

在调用 *ObjectBuilder* 构造函数后，针对新的 *builder* 对象，可以通过链式调用重载的 *set* 方法设置不同特性 (*set* 方法返回对象自身指针)。

由于图形对象创建和配置完成后不再需要 MQL 对象，因此我们会立即删除 *builder*。

脚本执行后，图表上会出现如下对象：



对象叠加和 Z 轴顺序优先级工具提示

每次运行时颜色和优先级都会不同，但矩形的视觉叠加顺序始终一致，按照创建顺序从最底层的 1 到最顶层的 12 排列（此处指的是对象的叠加层级，而非 12 一定位于表盘顶部）。

在图中，鼠标光标位于存在两个对象的位置，即 01（荧光酸橙绿）和 12（沙色）。此时，01 号对象的工具提示可见，尽管视觉上 12 号对象显示在 01 号之上。这是因为 01 号被随机生成了比 12 号更高的优先级。

由于一次只能显示一个工具提示，因此可以将鼠标移至没有对象重叠的其他区域，通过工具提示中的信息来查看优先级关系，此时工具提示中的信息属于光标下方的单个对象。

在下一章学习鼠标事件处理时，我们可以改进此示例，测试 Z 轴顺序对鼠标点击对象的影响。

要删除创建的对象，可使用 *ObjectCleanup1.mq5* 脚本。

5.8.14 对象显示设置：颜色、样式和边框

可通过各种特性更改对象的外观，本节我们将从颜色、样式、线宽和边框开始探讨这些特性。字体、倾斜度、文本对齐等其他格式特性将在后续章节详细说明。

下表所有特性类型均与整数兼容，因此可通过 *ObjectGetInteger* 和 *ObjectSetInteger* 函数进行管理。

标识符	说明	特性类型
OBJPROP_COLOR	对象线条和主要元素的颜色（例如字体或填充色）	color
OBJPROP_STYLE	线条样式	ENUM_LINE_STYLE
OBJPROP_WIDTH	线条粗细（像素）	int
OBJPROP_FILL	用颜色填充对象（适用于 OBJ_RECTANGLE、OBJ_TRIANGLE、OBJ_ELLIPSE、OBJ_CHANNEL、OBJ_STDDEVCHANNEL 和 OBJ_REGRESSION）	bool
OBJPROP_BACK	对象置于背景层	bool
OBJPROP_BGCOLOR	OBJ_EDIT、OBJ_BUTTON、OBJ_RECTANGLE_LABEL 的背景颜色	color
OBJPROP_BORDER_TYPE	矩形面板 OBJ_RECTANGLE_LABEL 的边框类型	ENUM_BORDER_TYPE
OBJPROP_BORDER_COLOR	输入字段 OBJ_EDIT 和按钮 OBJ_BUTTON 的边框颜色	color

与其他大多数带线条对象（如独立垂直线/水平线、趋势线、周期线、通道等）不同，这些对象的 OBJPROP_COLOR 特性定义线条颜色，而对于 OBJ_BITMAP_LABEL 和 OBJ_BITMAP 图像对象，该特性定义边框颜色，而 OBJPROP_STYLE 定义边框绘制类型。

我们在指标一章的 [绘图设置](#) 章节中已经介绍过用于 OBJPROP_STYLE 的 ENUM_LINE_STYLE 枚举类型。

务必需区分由前景色 OBJPROP_COLOR 执行的填充和背景色 OBJPROP_BGCOLOR 执行的填充。两者分别由不同的对象类型组支持，这些类型已在表中列出。

OBJPROP_BACK 特性需单独说明。需要明确的是，所有对象和指标都默认显示在价格图表上层。用户可以通过以下方式更改整个图表的此行为：进入图表的 **Setting** 对话框，然后进入 **Shared** 书签中的 **Chart on top** 选项。该标志在软件中有对应特性，即 CHART_FOREGROUND 特性（请参阅 [图表显示模式](#) 章节）。但有时可能需要仅将特定对象移至背景层，而非全部对象。此时，可将这些对象的 OBJPROP_BACK 设为 **true**。在此情况下，如果图表启用了网格和周期分隔线，该对象甚至会被这些分隔线覆盖。

当启用了 OBJPROP_FILL 填充模式时，形状内部的柱线颜色取决于 OBJPROP_BACK 特性。默认情况下，当 OBJPROP_BACK 为 **false** 时，与对象重叠的柱线会以 OBJPROP_COLOR 的反色绘制（反色通过将颜色值的所有二进制位取反得到，例如 0xFF0080 的反色为 0x00FF7F）。当 OBJPROP_BACK

为 `true`时，柱线将保持常规绘制，因为此时对象显示在背景层的图表“下层”（参见下方示例图）。

ENUM_BORDER_TYPE 枚举包含以下元素：

标识符	外观
BORDER_FLAT	平面
BORDER_RAISED	凸起
BORDER_SUNKEN	凹陷

当边框为平面（BORDER_FLAT）时，将根据 OBJPROP_COLOR、OBJPROP_STYLE 和 OBJPROP_WIDTH 特性，绘制为对应的颜色、样式和宽度。凸起和凹陷样式会通过 OBJPROP_BGCOLOR 的渐变色模拟边缘的立体倒角效果。

若未设置边框颜色（OBJPROP_BORDER_COLOR 默认值为 `clrNone`），输入字段将以主色 OBJPROP_COLOR 的线条勾勒边框，而按钮周围则会绘制带有 OBJPROP_BGCOLOR 渐变色调倒角的三维边框。

为测试新特性，可考虑使用 *ObjectStyle.mq5* 脚本。在该脚本中，我们将创建 5 个 OBJ_RECTANGLE 类型的矩形，这些对象均基于时间和价格。这些矩形将在窗口整个宽度范围均匀分布，用于高亮显示五个时间段中每个时间段内的最高价 *High* 和最低价 *Low* 之间的区间。对于所有对象，我们将调整并定期更改以下特性：线条颜色、线条样式、线条粗细以及图表底层的填充和显示选项。

我们将再次使用从 *Object Selector* 派生而来的辅助类 *ObjectBuilder*。和上一节不同的是，我们在 *ObjectBuilder* 中添加了析构函数，其中将调用 *ObjectDelete*。

```
#include <MQL5Book/ObjectMonitor.mqh>
#include <MQL5Book/AutoPtr.mqh>

class ObjectBuilder: public ObjectSelector
{
...
public:
    ~ObjectBuilder()
    {
        ObjectDelete(host, id);
    }
    ...
};
```

通过这种设计，该辅助类不仅可以配置对象，还可以在脚本运行结束时自动移除这些对象。

在 *OnStart* 函数中，我们将确定可见柱线数量和第一根柱线的索引，同时计算单个矩形所占的柱体宽度。

```

#define OBJECT_NUMBER 5

void OnStart()
{
    const string name = "ObjStyle-";
    const int bars = (int)ChartGetInteger(0, CHART_VISIBLE_BARS);
    const int first = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);
    const int rectsize = bars / OBJECT_NUMBER;
    ...
}

```

我们为对象预留一个智能指针数组，以确保调用 *ObjectBuilder*析构函数。

```
AutoPtr<ObjectBuilder> objects[OBJECT_NUMBER];
```

定义颜色调色板并创建 5 个矩形对象。

```

color colors[OBJECT_NUMBER] = {clrRed, clrGreen, clrBlue, clrMagenta, clrOrange};

for(int i = 0; i < OBJECT_NUMBER; ++i)
{
    // find the indexes of the bars that determine the range of prices in the i-th
    const int h = iHighest(NULL, 0, MODE_HIGH, rectsize, i * rectsize);
    const int l = iLowest(NULL, 0, MODE_LOW, rectsize, i * rectsize);
    // create and set up an object in the i-th subrange
    ObjectBuilder *object = new ObjectBuilder(name + (string)(i + 1), OBJ_RECTANGLE
    object.set(OBJPROP_TIME, iTime(NULL, 0, i * rectsize), 0);
    object.set(OBJPROP_TIME, iTime(NULL, 0, (i + 1) * rectsize), 1);
    object.set(OBJPROP_PRICE, iHigh(NULL, 0, h), 0);
    object.set(OBJPROP_PRICE, iLow(NULL, 0, l), 1);
    object.set(OBJPROP_COLOR, colors[i]);
    object.set(OBJPROP_WIDTH, i + 1);
    object.set(OBJPROP_STYLE, (ENUM_LINE_STYLE)i);
    // save to array
    objects[i] = object;
}
...

```

此处，为每个对象计算两个锚点的坐标；设置初始颜色、样式和线宽。

接下来，在无限循环中更改对象的特性。当 *ScrollLock*开启时，动画可以暂停。

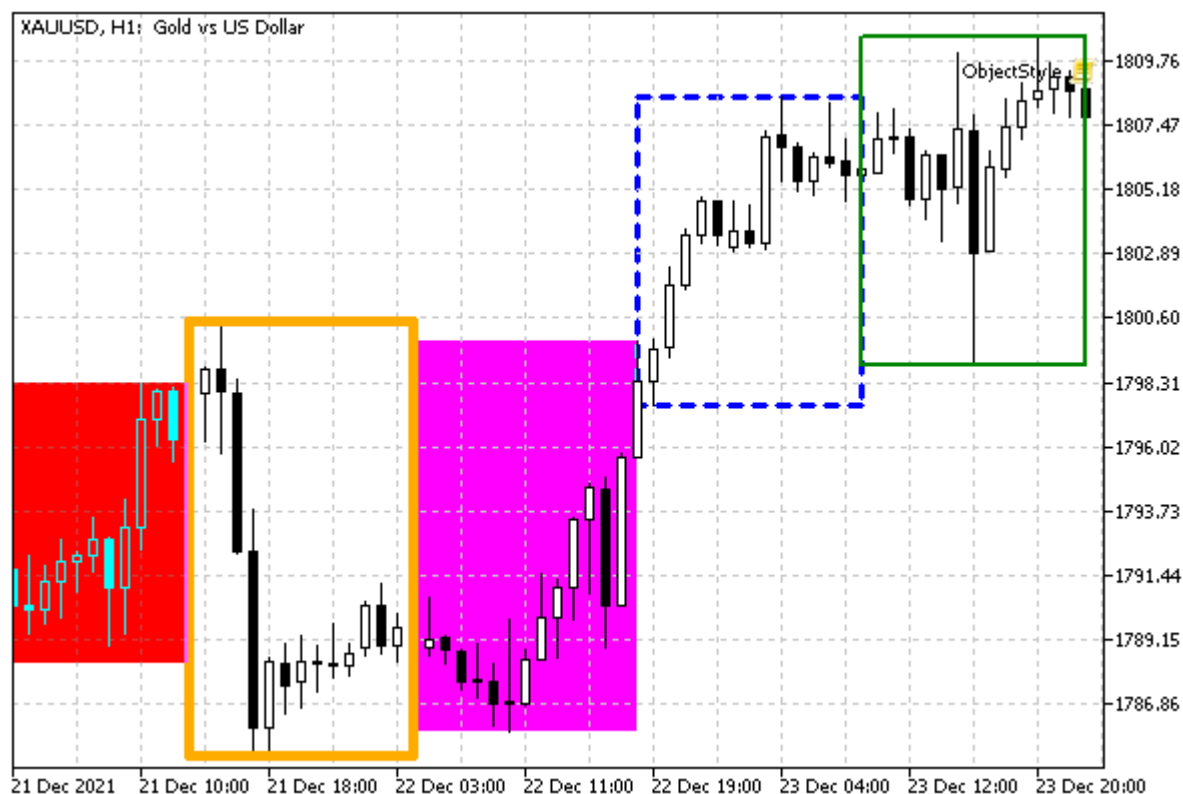
```

const int key = TerminalInfoInteger(TERMINAL_KEYSTATE_SCRLOCK);
int pass = 0;
int offset = 0;

for( ;!IsStopped(); ++pass)
{
    Sleep(200);
    if(TerminalInfoInteger(TERMINAL_KEYSTATE_SCRLOCK) != key) continue;
    // change color/style/width/fill/background from time to time
    if(pass % 5 == 0)
    {
        ++offset;
        for(int i = 0; i < OBJECT_NUMBER; ++i)
        {
            objects[i][].set(OBJPROP_COLOR, colors[(i + offset) % OBJECT_NUMBER]);
            objects[i][].set(OBJPROP_WIDTH, (i + offset) % OBJECT_NUMBER + 1);
            objects[i][].set(OBJPROP_FILL, rand() > 32768 / 2);
            objects[i][].set(OBJPROP_BACK, rand() > 32768 / 2);
        }
    }
    ChartRedraw();
}

```

图表实际效果如下。



具有不同显示设置的 OBJ_RECTANGLE 矩形

最左侧红色矩形启用了填充模式并位于前景层。因此，其内部柱线显示为对比鲜明的亮蓝色 (*clrAqua*，通常也称为 *cyan*，即 *clrRed* 的反色)。紫色矩形同样有填充，但设置了背景选项，内部柱线以标准方式显示。

请注意，橙色矩形因其较宽的边框线宽且设置为图表顶层显示，会完全覆盖其子范围起点和终点的柱线。

当启用填充时，不会考虑线宽。当边框宽度大于 1 时，部分虚线样式不会被应用。

ObjectShapesDraw

在本节的第二个示例中，回顾我们在第三章学习 OOP 时构思的虚拟绘图程序。当时我们的开发进度停留在：在虚拟绘图方法（名为 `draw`）中，只能向日志中输出“正在绘制特定形状”的消息。现在通过学习图形对象，我们已具备实际绘图能力。

让我们以 [Shapes5stats.mq5](#) 脚本为起点。更新后的版本将命名为 *ObjectShapesDraw.mq5*。

需要说明的是，除了基类 *Shape* 外，我们还描述了多个形状类：*Rectangle*、*Ellipse*、*Triangle*、*Square* 和 *Circle*。所有这些类都成功创建对应类型的图形对象：`OBJ_RECTANGLE`、`OBJ_ELLIPSE`、`OBJ_TRIANGLE`。但存在一些细微差别。

所有指定对象均与时间和价格坐标绑定，而我们的绘图程序采用统一的 X 轴和 Y 轴进行点定位。为此，我们需要以特殊方式配置图表以进行绘制，并调用 [ChartXYToTimePrice](#) 函数将屏幕坐标点重新换算为时间和价格值。

此外，`OBJ_ELLIPSE` 和 `OBJ_TRIANGLE` 对象允许任意旋转（特别是椭圆的长短轴可旋转），而 `OBJ_RECTANGLE` 的边始终保持水平和垂直方向。为简化示例，我们限制所有形状仅使用标准位置。

从理论上讲，新实现应视为图形对象的演示，而非完整的绘图程序。对于成熟的绘图功能，更合适的做法是使用 [图形资源](#)，这样可以避免图形对象带来的限制（因为图形对象一般用于其他用途，例如图表标记）。因此，我们将在后续关于资源章节中重新审视绘图程序的设计。

在新的 *Shape* 类中，我们将移除包含对象坐标的嵌套结构 *Pair*：该结构原本用于演示 OOP 的多项原则，但现在更简单的做法是将原始的字段描述 `int x, y` 直接还原到 *Shape* 类中。我们还将添加一个包含对象名的字段。

```

class Shape
{
    ...
protected:
    int x, y;
    color backgroundColor;
    const string type;
    string name;

    Shape(int px, int py, color back, string t) :
        x(px), y(py),
        backgroundColor(back),
        type(t)
    {
    }

public:
    ~Shape()
    {
        ObjectDelete(0, name);
    }
    ...
}

```

name 字段是必需的，可用于设置图形对象的特性，也可用于从图表中移除对象，这一操作在析构函数中执行是合乎逻辑的。

由于不同类型的形状需要不同数量的点或特征尺寸，除 *draw* 虚方法外，我们还会在 *Shape* 接口中添加 *setup* 方法：

```
virtual void setup(const int &parameters[]) = 0;
```

需要说明的是，我们在脚本中已实现了一个嵌套类 *Shape::Registrator*，它原本负责按类型统计形状数量。现在需要赋予它更重要的职责，作为形状工厂来运作。“工厂”类或方法的优势在于能够以统一的方式创建不同类的对象。

为此，我们向 *Registrator* 中添加一个创建形状的方法，其参数包括：第一个点的坐标（必需）、颜色以及一个额外参数数组（每个形状都能够按照自己的规则解析该数组，且未来可从中读取或写入文件）。

```
virtual Shape *create(const int px, const int py, const color back,
    const int &parameters[]) = 0;
```

该方法是抽象虚方法，因为特定类型的形状只能通过 *Shape* 派生类中描述的派生注册类来创建。为简化派生日志记录类的编写工作，我们引入了一个模板类 *MyRegistrator*，其包含适用于所有场景的 *create* 方法实现。


```

template<typename T>
class MyRegistrator : public Shape::Registrator
{
public:
    MyRegistrator() : Registrator(typename(T))
    {
    }

    virtual Shape *create(const int px, const int py, const color back,
        const int &parameters[]) override
    {
        T *temp = new T(px, py, back);
        temp.setup(parameters);
        return temp;
    }
};

```

在此方法中，我们调用某个先前未知形状 `T` 的构造函数，通过调用 `setup` 对其进行调整，并将实例返回给调用代码。

以下是在 `Rectangle` 类中的使用方式，该类有两个用于宽度和高度的额外参数。

```

class Rectangle : public Shape
{
    static MyRegistrator<Rectangle> r;

protected:
    int dx, dy; // dimensions (width, height)

    Rectangle(int px, int py, color back, string t) :
        Shape(px, py, back, t), dx(1), dy(1)
    {
    }

public:
    Rectangle(int px, int py, color back) :
        Shape(px, py, back, typename(this)), dx(1), dy(1)
    {
        name = typename(this) + (string)r.increment();
    }

    virtual void setup(const int &parameters[]) override
    {
        if(ArraySize(parameters) < 2)
        {
            Print("Insufficient parameters for Rectangle");
            return;
        }
        dx = parameters[0];
        dy = parameters[1];
    }
    ...
};

static MyRegistrator<Rectangle> Rectangle::r;

```

创建形状时，其名称不仅包含类名 (*typename*)，还包含通过 *r.increment()* 调用计算得出的实例序号。

其他形状类的描述方式类似。

现在我们需要研究 *Rectangle* 类的 *draw* 方法。在该方法中，我们使用 *ChartXYToTimePrice* 将一对点 (x,y) 和 (x + dx, y + dy) 转换为时间/价格坐标，并创建一个 OBJ_RECTANGLE 对象。

```

void draw() override
{
    // Print("Drawing rectangle");
    int subw;
    datetime t;
    double p;
    ChartXYToTimePrice(0, x, y, subw, t, p);
    ObjectCreate(0, name, OBJ_RECTANGLE, 0, t, p);
    ChartXYToTimePrice(0, x + dx, y + dy, subw, t, p);
    ObjectSetInteger(0, name, OBJPROP_TIME, 1, t);
    ObjectSetDouble(0, name, OBJPROP_PRICE, 1, p);

    ObjectSetInteger(0, name, OBJPROP_COLOR, backgroundColor);
    ObjectSetInteger(0, name, OBJPROP_FILL, true);
}

```

当然，不要忘记将颜色设置为 OBJPROP_COLOR，将填充设置为 OBJPROP_FILL。

对于 *Square* 类，实际上不需要做任何更改：只需将 *dx* 和 *dy* 设置为相等即可。

对于 *Ellipse* 类，两个额外选项 *dx* 和 *dy* 决定了相对于中心 (x,y) 绘制的短半径和长半径。相应地，在 *draw* 方法中，我们将计算 3 个锚点并创建一个 OBJ_ELLIPSE 对象。

```

class Ellipse : public Shape
{
    static MyRegistrator<Ellipse> r;
protected:
    int dx, dy; // large and small radii
    ...
public:
    void draw() override
    {
        // Print("Drawing ellipse");
        int subw;
        datetime t;
        double p;

        // (x, y) center
        // p0: x + dx, y
        // p1: x - dx, y
        // p2: x, y + dy

        ChartXYToTimePrice(0, x + dx, y, subw, t, p);
        ObjectCreate(0, name, OBJ_ELLIPSE, 0, t, p);
        ChartXYToTimePrice(0, x - dx, y, subw, t, p);
        ObjectSetInteger(0, name, OBJPROP_TIME, 1, t);
        ObjectSetDouble(0, name, OBJPROP_PRICE, 1, p);
        ChartXYToTimePrice(0, x, y + dy, subw, t, p);
        ObjectSetInteger(0, name, OBJPROP_TIME, 2, t);
        ObjectSetDouble(0, name, OBJPROP_PRICE, 2, p);

        ObjectSetInteger(0, name, OBJPROP_COLOR, backgroundColor);
        ObjectSetInteger(0, name, OBJPROP_FILL, true);
    }
};

static MyRegistrator<Ellipse> Ellipse::r;

```

Circle 是长半径和短半径相等的特殊椭圆。

最后一点，现阶段仅支持等边三角形：其边长包含在额外字段 *dx* 中。我们建议你自行查阅源代码中的 *draw* 方法。

新脚本将延续之前的功能，随机生成指定数量的形状。它们由 *addRandomShape* 函数创建。

```

Shape *addRandomShape()
{
    const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
    const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);

    const int n = random(Shape::Registrar::getTypeCount());

    int cx = 1 + w / 4 + random(w / 2), cy = 1 + h / 4 + random(h / 2);
    int clr = ((random(256) << 16) | (random(256) << 8) | random(256));
    int custom[] = {1 + random(w / 4), 1 + random(h / 4)};
    return Shape::Registrar::get(n).create(cx, cy, clr, custom);
}

```

在该函数中，我们可以看到工厂方法 *create* 的用法，该方法通过随机选择的寄存器对象，带有数字 *n*。如果决定后续添加其他形状类，也无需更改生成逻辑中的任何内容。

所有形状都均位于窗口中央，且尺寸不超过窗口的四分之一。

接下来需要直接研究 *addRandomShape* 函数的调用，以及前文提及的特殊时间表设置。

为实现屏幕上点的正方形呈现，可设置 *CHART_SCALEFIX_11* 模式。此外，我们将沿时间轴选择最密集（压缩）的刻度 *CHART_SCALE* (0)，因为其中一个柱线占用 1 个水平像素（最高精度）。最后，通过将 *CHART_SHOW* 设置为 *false* 来禁用图表本身的显示。

```

void OnStart()
{
    const int scale = (int)ChartGetInteger(0, CHART_SCALE);
    ChartSetInteger(0, CHART_SCALEFIX_11, true);
    ChartSetInteger(0, CHART_SCALE, 0);
    ChartSetInteger(0, CHART_SHOW, false);
    ChartRedraw();
    ...
}

```

为存储形状，我们将预留一个智能指针数组，并用随机形状填充该数组。

```

#define FIGURES 21
...
void OnStart()
{
    ...
    AutoPtr<Shape> shapes[FIGURES];

    for(int i = 0; i < FIGURES; ++i)
    {
        Shape *shape = shapes[i] = addRandomShape();
        shape.draw();
    }

    ChartRedraw();
    ...
}

```

然后，运行一个无限循环，直到用户停止脚本，在循环中通过 *move* 方法微调所有形状的位置。

```

while(!IsStopped())
{
    Sleep(250);
    for(int i = 0; i < FIGURES; ++i)
    {
        shapes[i][].move(random(20) - 10, random(20) - 10);
        shapes[i][].draw();
    }
    ChartRedraw();
}
...

```

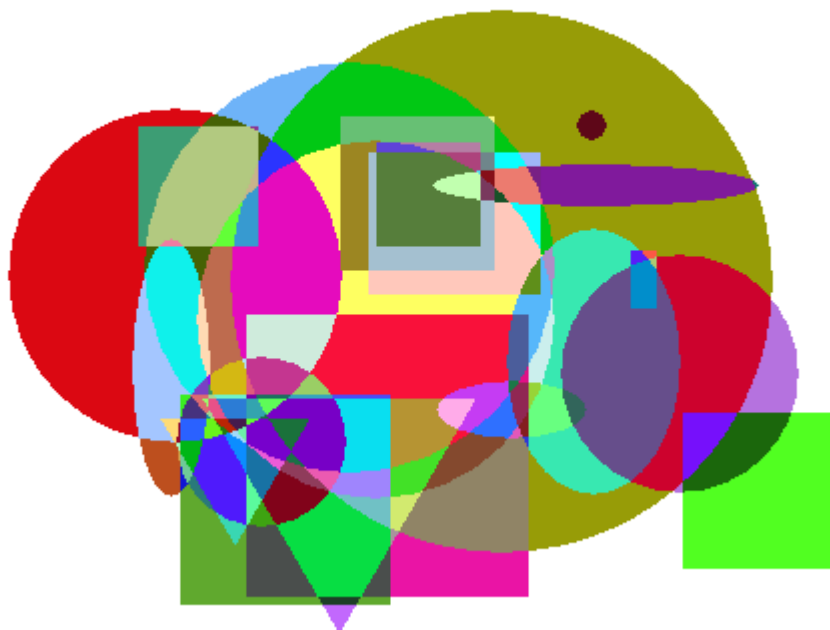
最后，我们将恢复图表设置。

```

// it's not enough to disable CHART_SCALEFIX_11, you need CHART_SCALEFIX
ChartSetInteger(0, CHART_SCALEFIX, false);
ChartSetInteger(0, CHART_SCALE, scale);
ChartSetInteger(0, CHART_SHOW, true);
}

```

以下截图展示了绘制形状后的图表可能呈现的效果：



图表形状对象

对象绘制的特点是在重叠区域产生颜色“叠加”效果。

由于 Y 轴采用自上而下的方向，所有三角形均呈现倒置状态，但这个问题不是很严重，因为我们最终将基于资源重构绘图程序。

5.8.15 字体设置

所有类型的对象都支持为其设置特定文本 (OBJPROP_TEXT)。其中许多对象会将指定的文本直接显示在图表上，而其余对象的文本则会成为工具提示的信息部分。

当文本显示在对象内部时 (例如 OBJ_TEXT、OBJ_LABEL、OBJ_BUTTON 和 OBJ_EDIT 类型)，可以选择字体名称和大小。对于其他类型的对象，字体设置不适用：它们的描述始终以图表的标准字体显示。

标识符	说明	类型
OBJPROP_FONTSIZE	字体大小 (以像素为单位)	int
OBJPROP_FONT	字体	string

在此处无法以 打印点 为单位设置字体大小。

测试脚本 *ObjectFont.mq5*用于创建带文本的对象，并更改其名称和字体大小。我们将使用上一个脚本中的 *ObjectBuilder*类。

在 *OnStart*起始阶段，脚本将计算窗口在屏幕坐标和时间/价格轴上的中点。这是必要的，因为参与测试的不同类型对象使用不同的坐标系。

```
void OnStart()
{
    const string name = "ObjFont-";

    const int bars = (int)ChartGetInteger(0, CHART_WIDTH_IN_BARS);
    const int first = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);

    const datetime centerTime = iTime(NULL, 0, first - bars / 2);
    const double centerPrice =
        (ChartGetDouble(0, CHART_PRICE_MIN)
         + ChartGetDouble(0, CHART_PRICE_MAX)) / 2;

    const int centerX = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) / 2;
    const int centerY = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS) / 2;
    ...
}
```

测试对象类型列表在 *types*数组中指定。对于某些对象 (特别是 OBJ_HLINE 和 OBJ_VLINE)，字体设置将不会生效，尽管其描述文本仍会显示在屏幕上 (为确保显示效果，我们启用了 CHART_SHOW_OBJECT_DESCR 模式)。

```

ChartSetInteger(0, CHART_SHOW_OBJECT_DESCR, true);

ENUM_OBJECT types[] =
{
    OBJ_HLINE,
    OBJ_VLINE,
    OBJ_TEXT,
    OBJ_LABEL,
    OBJ_BUTTON,
    OBJ_EDIT,
};
int t = 0; // cursor
...

```

*t*变量将用于在不同类型之间按顺序切换。

*fonts*数组包含了最常用的 Windows 标准字体。

```

string fonts[] =
{
    "Comic Sans MS",
    "Consolas",
    "Courier New",
    "Lucida Console",
    "Microsoft Sans Serif",
    "Segoe UI",
    "Tahoma",
    "Times New Roman",
    "Trebuchet MS",
    "Verdana"
};

int f = 0; // cursor
...

```

我们将使用 *f* 变量遍历这些字体。

在演示循环内部，我们指示 *ObjectBuilder* 在窗口中间创建当前类型 *types[t]* 的对象（为实现统一处理，坐标会同时在两种坐标系中指定，这样代码就无需根据对象类型来区分处理：对象不支持的坐标系将不会产生任何效果）。


```

while(!IsStopped())
{

    const string str = EnumToString(types[t]);
    ObjectBuilder *object = new ObjectBuilder(name + str, types[t]);
    object.set(OBJPROP_TIME, centerTime);
    object.set(OBJPROP_PRICE, centerPrice);
    object.set(OBJPROP_XDISTANCE, centerX);
    object.set(OBJPROP_YDISTANCE, centerY);
    object.set(OBJPROP_XSIZE, centerX / 3 * 2);
    object.set(OBJPROP_YSIZE, centerY / 3 * 2);
    ...
}

```

接下来，我们设置文本和字体（字体大小随机选择）。

```

const int size = rand() * 15 / 32767 + 8;
Comment(str + " " + fonts[f] + " " + (string)size);
object.set(OBJPROP_TEXT, fonts[f] + " " + (string)size);
object.set(OBJPROP_FONT, fonts[f]);
object.set(OBJPROP_FONTSIZE, size);
...

```

为进行下一轮遍历，我们会在对象类型数组和字体名称数组中移动光标。

```

t = ++t % ArraySize(types);
f = ++f % ArraySize(fonts);
...

```

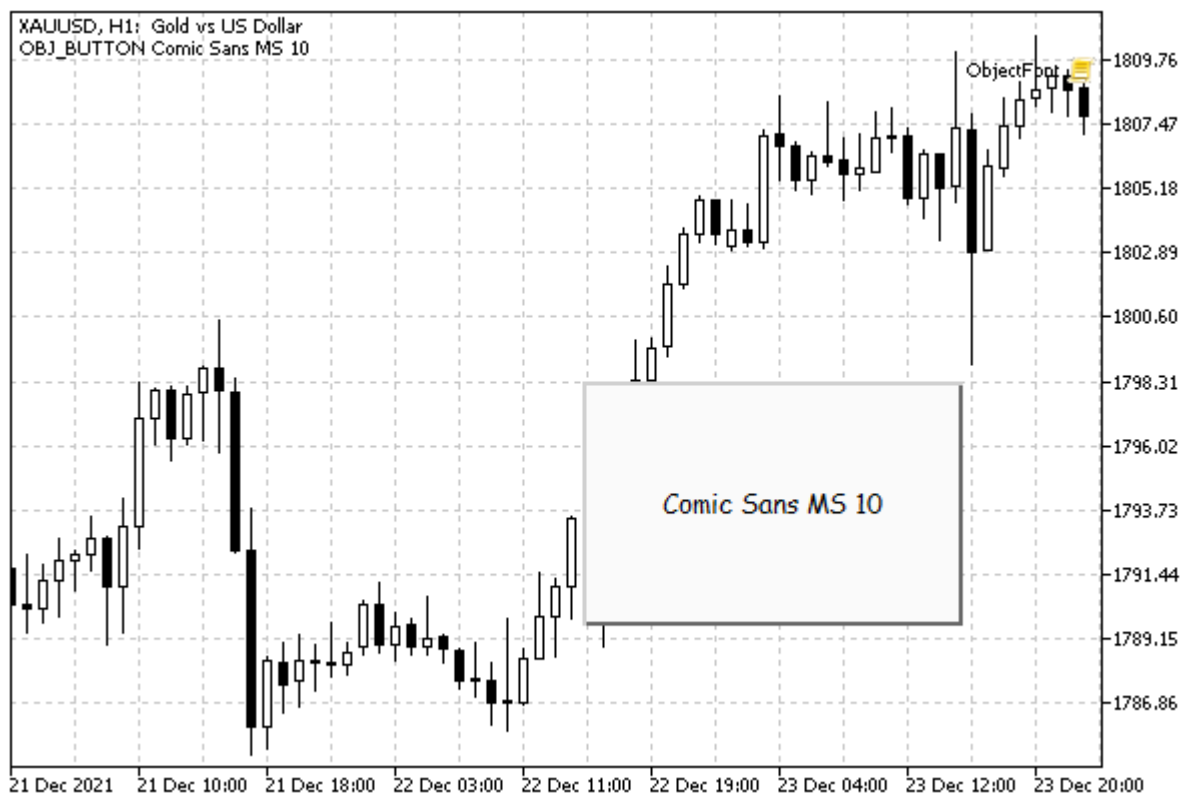
最后，我们更新图表，等待 1 秒，然后删除对象以创建另一个对象。

```

    ChartRedraw();
    Sleep(1000);
    delete object;
}
}

```

下图显示脚本运行时的画面状态。



自定义字体设置按钮

5.8.16 以任意角度旋转文本

文本类型的对象 - 标签 OBJ_TEXT (采用报价坐标) 和面板 OBJ_LABEL (采用屏幕坐标) - 允许你对文本标签进行任意角度旋转。为此，提供了类型为 *double* 的 OBJPROP_ANGLE 特性。该特性包含相对于对象正常位置的角度，以度为单位。正值将对象逆时针旋转，负值将对象顺时针旋转。

不过需要注意的是，相差 360 度倍数的角度实际效果相同。例如，+315 度和 -45 度即为等效角度。旋转围绕对象的锚点进行（默认情况下为左上角）。



将 OBJ_LABEL 和 OBJ_TEXT 对象旋转 45 度的整数倍

可以使用 *ObjectAngle.mq5* 脚本来检查 OBJPROP_ANGLE 特性对某一对象的影响。该脚本将在窗口中央创建一个文本标签 OBJ_LABEL，随后开始以 45 度为单位进行周期性旋转，直到用户停止该进程。

```
void OnStart()
{
    const string name = "ObjAngle";
    ObjectCreate(0, name, OBJ_LABEL, 0, 0, 0);
    const int centerX = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) / 2;
    const int centerY = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS) / 2;
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, centerX);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, centerY);
    ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_CENTER);

    int angle = 0;
    while(!IsStopped())
    {
        ObjectSetString(0, name, OBJPROP_TEXT, StringFormat("Angle: %d°", angle));
        ObjectSetDouble(0, name, OBJPROP_ANGLE, angle);
        angle += 45;

        ChartRedraw();
        Sleep(1000);
    }
    ObjectDelete(0, name);
}
```

文本将显示当前的角度值。

5.8.17 确定对象宽度和高度

某些对象类型支持以像素为单位设置尺寸。这些对象包括 `OBJ_BUTTON`、`OBJ_CHART`、`OBJ_BITMAP`、`OBJ_BITMAP_LABEL`、`OBJ_EDIT` 和 `OBJ_RECTANGLE_LABEL`。此外，`OBJ_LABEL` 对象仅支持读取（不支持设置）尺寸，因为标签会根据所含文本自动调整大小。如果尝试访问其他类型对象的特性，将导致 `OBJECT_WRONG_PROPERTY (4203)` 错误。

标识符	说明
<code>OBJPROP_XSIZE</code>	对象在 X 轴方向的宽度，以像素为单位
<code>OBJPROP_YSIZE</code>	对象在 Y 轴方向的高度，以像素为单位

这两个尺寸均为整数，需通过 `ObjectGetInteger/ObjectSetInteger` 函数处理。

对于 `OBJ_BITMAP` 和 `OBJ_BITMAP_LABEL` 对象，执行特殊的尺寸处理。

未指定图像时，这些对象允许设置任意尺寸。此时，对象以透明方式绘制（如果未通过设置颜色为 `clrNone` 隐藏边框，则仅边框可见），但仍会接收所有事件，尤其是鼠标移动事件（如果有文本描述则显示在工具提示中）和按钮点击对象事件。

分配图像后，图像默认采用对象的高度和宽度。但 MQL 程序可设置更小尺寸并选择显示图像片段，详情请参阅 [帧设置](#) 章节。如果尝试将高度或宽度设置为大于图像尺寸，图像将停止显示，且对象尺寸保持不变。

作为示例，我们将开发一个 [在对象上定义锚点](#) 章节中 `ObjectAnchorLabel.mq5` 脚本的改进版本。在该章节中，我们实现了文本标签在窗口内的移动，并在其接触窗口边框时反转方向，但当时我们仅考虑了锚点位置。因此，根据锚点在对象上的位置不同，可能出现标签几乎完全移出窗口的情况。例如，如果锚点位于对象右侧，向左移动时在锚点接触边缘前，几乎全部文本可能就已超出窗口左边框。

在新脚本 `ObjectSizeLabel.mq5` 中，我们将考虑对象尺寸，当对象任一边缘接触窗口边框时立即改变移动方向。

为确保该模式正确运行，需考虑以下因素，在将窗口的任意角点用作对象锚点的坐标参考中心时，会决定 X 轴和 Y 轴的方向特性。例如，如果用户在 `ENUM_BASE_CORNER` `Corner` 输入变量中选择左上角，则 X 轴方向为从左向右递增，Y 轴方向为从上向下递增。如果将参考中心设为右下角，则 X 轴方向为从右向左递增，Y 轴方向为从下向上递增。

窗口锚点角点与对象锚点位置的不同组合，需要对对象边缘与窗口边框的距离进行不同调整。具体而言，当选择窗口右侧某个角点和对象右侧的某个锚点时，则无需调整窗口右侧边框的距离，而在相反的左侧，则必须考虑对象的宽度（以确保其尺寸不会超出窗口左侧）。

关于对象尺寸调整的这一规则可归纳为以下节点：

- ① 在与锚点角点相邻的窗口边框处，当锚点位于对象远端（相对于该角点），则需进行调整；
- ② 在与锚点角点相邻的窗口边框处，当锚点位于对象近端（相对于该角点），则需进行调整。

换言之，如果角点名称（来自 `ENUM_BASE_CORNER` 元素）与锚点名称（来自 `ENUM_ANCHOR_POINT` 元素）包含相同词（例如 `RIGHT`），则需在窗口远端（即远离所选角点的一侧）进行调整。如果 `ENUM_BASE_CORNER` 与 `ENUM_ANCHOR_POINT` 组合中的方向相反（例如 `LEFT` 与 `RIGHT`），则需在窗口近端进行调整。这些规则同样适用于水平轴和垂直轴。

此外还需注意，锚点可能位于对象任意一条边的中点位置。此时在垂直方向上，需要设置与窗口边框的缩进，缩进值应为对象对应尺寸的一半。

特殊情况是锚点位于对象中心的情况。在任意方向上均需预留等于对象一半尺寸的距离余量。

上述逻辑通过名为 *GetMargins* 的特殊函数实现。该函数的输入参数包括所选角点、锚点位置以及对象尺寸 (*dx* 和 *dy*)。该函数返回一个包含 4 个字段的结构，这些字段分别表示从锚点向窗口近端和远端边框方向应预留的额外缩进量，以确保对象不会超出可视区域。缩进量根据对象自身的尺寸和相对位置进行预留。

```
struct Margins
{
    int nearX; // X increment between the object point and the window border adjacent
    int nearY; // Y increment between the object point and the window border adjacent
    int farX;  // X increment between the object's point and the opposite corner of th
    int farY;  // Y increment between the object's point and the opposite corner of th
};

Margins GetMargins(const ENUM_BASE_CORNER corner, const ENUM_ANCHOR_POINT anchor,
    int dx, int dy)
{
    Margins margins = {}; // zero corrections by default
    ...
    return margins;
}
```

为统一算法实现，引入以下方向（侧边）的宏定义：

```
#define LEFT 0x1
#define LOWER 0x2
#define RIGHT 0x4
#define UPPER 0x8
#define CENTER 0x16
```

通过这些宏，可定义位掩码（组合），用于描述 `ENUM_BASE_CORNER` 和 `ENUM_ANCHOR_POINT` 枚举中的元素。

```

const int corner_flags[] = // flags for ENUM_BASE_CORNER elements
{
    LEFT | UPPER,
    LEFT | LOWER,
    RIGHT | LOWER,
    RIGHT | UPPER
};

const int anchor_flags[] = // flags for ENUM_ANCHOR_POINT elements
{
    LEFT | UPPER,
    LEFT,
    LEFT | LOWER,
    LOWER,
    RIGHT | LOWER,
    RIGHT,
    RIGHT | UPPER,
    UPPER,
    CENTER
};

```

数组 *corner_flags* 和 *anchor_flags* 各自的元素数量与对应枚举中的元素数量完全一致。

接下来是主函数代码。首先，处理最简单的情况：中心锚点。

```

if(anchor == ANCHOR_CENTER)
{
    margins.nearX = margins.farX = dx / 2;
    margins.nearY = margins.farY = dy / 2;
}
else
{
    ...
}

```

对于其他情况，我们将通过接收到的 *corner* 和 *anchor* 值直接访问上述数组中的位掩码进行分析。

```

const int mask = corner_flags[corner] & anchor_flags[anchor];
...

```

如果角点与锚点位于同一水平侧边，则以下条件适用，并且将在窗口远端调整对象宽度。

```

if((mask & (LEFT | RIGHT)) != 0)
{
    margins.farX = dx;
}
...

```

如果两者不在同侧，则可能处于相反侧，或锚点位于水平边（上/下）的中间位置。通过表达式 *(anchor_flags[anchor] & (LEFT | RIGHT)) == 0* 可检查锚点是否位于中点，如果是，则调整量为对象宽度的一半。

```

else
{
    if((anchor_flags[anchor] & (LEFT | RIGHT)) == 0)
    {
        margins.nearX = dx / 2;
        margins.farX = dx / 2;
    }
    else
    {
        margins.nearX = dx;
    }
}
...

```

否则，如果角点与锚点方向相反，则需在窗口近端按对象宽度进行调整。

对 Y 轴执行类似的检查。

```

if((mask & (UPPER | LOWER)) != 0)
{
    margins.farY = dy;
}
else
{
    if((anchor_flags[anchor] & (UPPER | LOWER)) == 0)
    {
        margins.farY = dy / 2;
        margins.nearY = dy / 2;
    }
    else
    {
        margins.nearY = dy;
    }
}

```

现在，*GetMargins*函数已准备就绪，我们可以在 *OnStart* 函数中继续编写脚本主代码。与之前一样，我们先确定窗口尺寸、计算中心初始坐标、创建 OBJ_LABEL 对象并选中它。

```

void OnStart()
{
    const int t = ChartWindowOnDropped();
    Comment(EnumToString(Corner));

    const string name = "ObjSizeLabel";
    int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t) - 1;
    int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) - 1;
    int x = w / 2;
    int y = h / 2;

    ObjectCreate(0, name, OBJ_LABEL, t, 0, 0);
    ObjectSetInteger(0, name, OBJPROP_SELECTABLE, true);
    ObjectSetInteger(0, name, OBJPROP_SELECTED, true);
    ObjectSetInteger(0, name, OBJPROP_CORNER, Corner);
    ...
}

```

为实现动画效果，通过无限循环提供 *pass* (迭代计数器)和 *anchor* (锚点周期性随机选择)变量。

```

int pass = 0;
ENUM_ANCHOR_POINT anchor = 0;
...

```

但与 *ObjectAnchorLabel.mq5*相比存在一些变化。

我们将不再生成对象的随机移动。而是设置 5 像素的恒定对角线移动速度。

```
int px = 5, py = 5;
```

为存储文本标签的尺寸，我们将预留两个新变量。

```
int dx = 0, dy = 0;
```

额外缩进的统计结果将存储在 *Margins*类型的变量 *m* 中。

```
Margins m = {};
```

接下来直接进入移动和修改对象的循环。在循环中，每 75 次迭代 (每次迭代 100 毫秒，详见后文)会随机选择一个新锚点，根据该锚点生成新文本 (对象内容)，然后等待将更改应用于对象 (调用 *ChartRedraw*)。此步骤必不可少，因为标签尺寸大小会随内容自动调整，而新尺寸对于 *GetMargins*调用中正确计算缩进至关重要。

我们通过调用 *ObjectGetInteger*并使用性 *OBJPROP_XSIZE* 和 *OBJPROP_YSIZE* 来获取尺寸。


```

for( ;!IsStopped(); ++pass)
{
    if(pass % 75 == 0)
    {
        // ENUM_ANCHOR_POINT consists of 9 elements: randomly choose one
        const int r = rand() * 8 / 32768 + 1;
        anchor = (ENUM_ANCHOR_POINT)((anchor + r) % 9);
        ObjectSetInteger(0, name, OBJPROP_ANCHOR, anchor);
        ObjectSetString(0, name, OBJPROP_TEXT, " " + EnumToString(anchor)
            + StringFormat("[%3d,%3d] ", x, y));
        ChartRedraw();
        Sleep(1);

        dx = (int)ObjectGetInteger(0, name, OBJPROP_XSIZE);
        dy = (int)ObjectGetInteger(0, name, OBJPROP_YSIZE);

        m = GetMargins(Corner, anchor, dx, dy);
    }
    ...
}

```

确定锚点位置及所有距离后，即可移动对象。如果对象“碰撞”到边界，移动方向改为相反方向（根据碰撞侧边将 *px* 转为 *-px* 或将 *py* 转为 *-py*）。

```

// bounce off window borders, object fully visible
if(x + px >= w - m.farX)
{
    x = w - m.farX + px - 1;
    px = -px;
}
else if(x + px < m.nearX)
{
    x = m.nearX + px;
    px = -px;
}

if(y + py >= h - m.farY)
{
    y = h - m.farY + py - 1;
    py = -py;
}
else if(y + py < m.nearY)
{
    y = m.nearY + py;
    py = -py;
}

// calculate the new label position
x += px;
y += py;
...

```

最后需要更新对象状态:在文本标签中显示当前坐标,并将其赋值给 `OBJPROP_XDISTANCE` 和 `OBJPROP_YDISTANCE` 特性。

```

ObjectSetString(0, name, OBJPROP_TEXT, " " + EnumToString(anchor)
    + StringFormat("[%3d,%3d] ", x, y));
ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
...

```

修改对象后,我们将调用 *ChartRedraw*并等待 100 毫秒,以确保动画流畅运行。

```

ChartRedraw();
Sleep(100);
...

```

在循环结束时,我们将再次检查窗口尺寸 (因为脚本运行时用户可能改变窗口大小),并重新获取对象尺寸。

```
h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t) - 1;
w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) - 1;

dx = (int)ObjectGetInteger(0, name, OBJPROP_XSIZE);
dy = (int)ObjectGetInteger(0, name, OBJPROP_YSIZE);
m = GetMargins(Corner, anchor, dx, dy);
}
```

为保持说明简洁，我们省略了 *ObjectSizeLabel.mq5* 脚本的其他一些改进功能。感兴趣的读者可以参考代码。具体来说，标注使用了差异化颜色：每个锚点都对应特定的颜色，从而使锚点切换更加直观可见。此外，可在脚本运行过程中点击 *Delete*，以将所选对象从图表中移除，且脚本将自动终止。

5.8.18 对象在时间范围内的可见性

MetaTrader 5 用户了解对象特性对话框中的“显示”选项卡，在该选项卡中可以使用开关选择对象将在哪些时间范围显示或隐藏。特别是，可以临时在所有时间范围上完全隐藏对象。

MT5 具有类似的程序特性 OBJPROP_TMEFRAMES，该特性可控制对象在某个时间范围上的可见性。该特性值可以是特殊整型标志的任意组合：每个标志（常量）包含对应一个时间范围的位（见下表）。要设置/获取 OBJPROP_TMEFRAMES 特性，请使用 *ObjectSetInteger/ObjectGetInteger* 函数。

常量	值	在时间范围中的可见性
OBJ_NO_PERIODS	0	对象在所有时间范围中隐藏
OBJ_PERIOD_M1	0x00000001	M1
OBJ_PERIOD_M2	0x00000002	M2
OBJ_PERIOD_M3	0x00000004	M3
OBJ_PERIOD_M4	0x00000008	M4
OBJ_PERIOD_M5	0x00000010	M5
OBJ_PERIOD_M6	0x00000020	M6
OBJ_PERIOD_M10	0x00000040	M10
OBJ_PERIOD_M12	0x00000080	M12
OBJ_PERIOD_M15	0x00000100	M15
OBJ_PERIOD_M20	0x00000200	M20
OBJ_PERIOD_M30	0x00000400	M30
OBJ_PERIOD_H1	0x00000800	H1
OBJ_PERIOD_H2	0x00001000	H2
OBJ_PERIOD_H3	0x00002000	H3

常量	值	在时间范围中的可见性
OBJ_PERIOD_H4	0x00004000	H4
OBJ_PERIOD_H6	0x00008000	H6
OBJ_PERIOD_H8	0x00010000	H8
OBJ_PERIOD_H12	0x00020000	H12
OBJ_PERIOD_D1	0x00040000	D1
OBJ_PERIOD_W1	0x00080000	W1
OBJ_PERIOD_MN1	0x00100000	MN1
OBJ_ALL_PERIODS	0x001fffff	所有时间范围

这些标志可使用按位或运算符 (“|”) 进行组合，例如，标志 OBJ_PERIOD_M15 | OBJ_PERIOD_H4 的叠加表示对象将在 15 分钟和 4 小时时间范围上可见。

请注意，每个标志可通过如下方式来获得：左移 1，左移的位数等于常量在表中的编号。这样，当算法在多个时间范围而非特定某个时间范围下运行时，可以更轻松地动态生成标志。

我们将在测试脚本 *ObjectTimeframes.mq5* 中使用这一功能。该脚本的任务是在图表上创建大量带有时间范围名称的大型文本标签，且每个标题仅在对应的时间范围中显示。例如，大型标签 “D1” 将仅在日线图上可见，而切换到 H4 时，我们会看到 “H4”。

还实现了一个简单的辅助函数，用于获取不带 “PERIOD_” 前缀的时间范围简称。

```
string GetPeriodName(const int tf)
{
    const static int PERIOD_ = StringLen("PERIOD_");
    return StringSubstr(EnumToString((ENUM_TIMEFRAMES)tf), PERIOD_);
}
```

要从 ENUM_TIMEFRAMES 枚举中获取所有时间范围的列表，我们将使用 [枚举转换](#) 章节介绍的 EnumToArray 函数。

```
#include "ObjectPrefix.mqh"
#include <MQL5Book/EnumToArray.mqh>

void OnStart()
{
    ENUM_TIMEFRAMES tf = 0;
    int values[];
    const int n = EnumToArray(tf, values, 0, USHORT_MAX);
    ...
}
```

所有标签将在脚本启动时显示在窗口中央。脚本执行完毕后调整窗口大小，会导致已创建的标题不再居中。这是因为 MQL5 仅支持将对象锚定至窗口角点，不支持锚定到中心位置。如果希望自动保持对象的位置，应当在指标中实现类似算法，并对 [窗口大小调整事件](#) 做出响应。或者，我们可以将标签显示在某个角落，例如右下角。

```

const int centerX = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) / 2;
const int centerY = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS) / 2;
...

```

在遍历时间范围的循环中，我们将每个时间范围创建一个 OBJ_LABEL 对象，并将其放置在窗口中央并以对象中心为锚点。

```

for(int i = 1; i < n; ++i)
{
    // create and setup text label for each timeframe
    const string name = ObjNamePrefix + (string)i;
    ObjectCreate(0, name, OBJ_LABEL, 0, 0, 0);
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, centerX);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, centerY);
    ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_CENTER);
    ...
}

```

接下来，设置文本（时间范围名称）、大号字体大小、灰色以及背景显示特性。

```

ObjectSetString(0, name, OBJPROP_TEXT, GetPeriodName(values[i]));
ObjectSetInteger(0, name, OBJPROP_FONTSIZE, fmin(centerY, centerX));
ObjectSetInteger(0, name, OBJPROP_COLOR, clrLightGray);
ObjectSetInteger(0, name, OBJPROP_BACK, true);
...

```

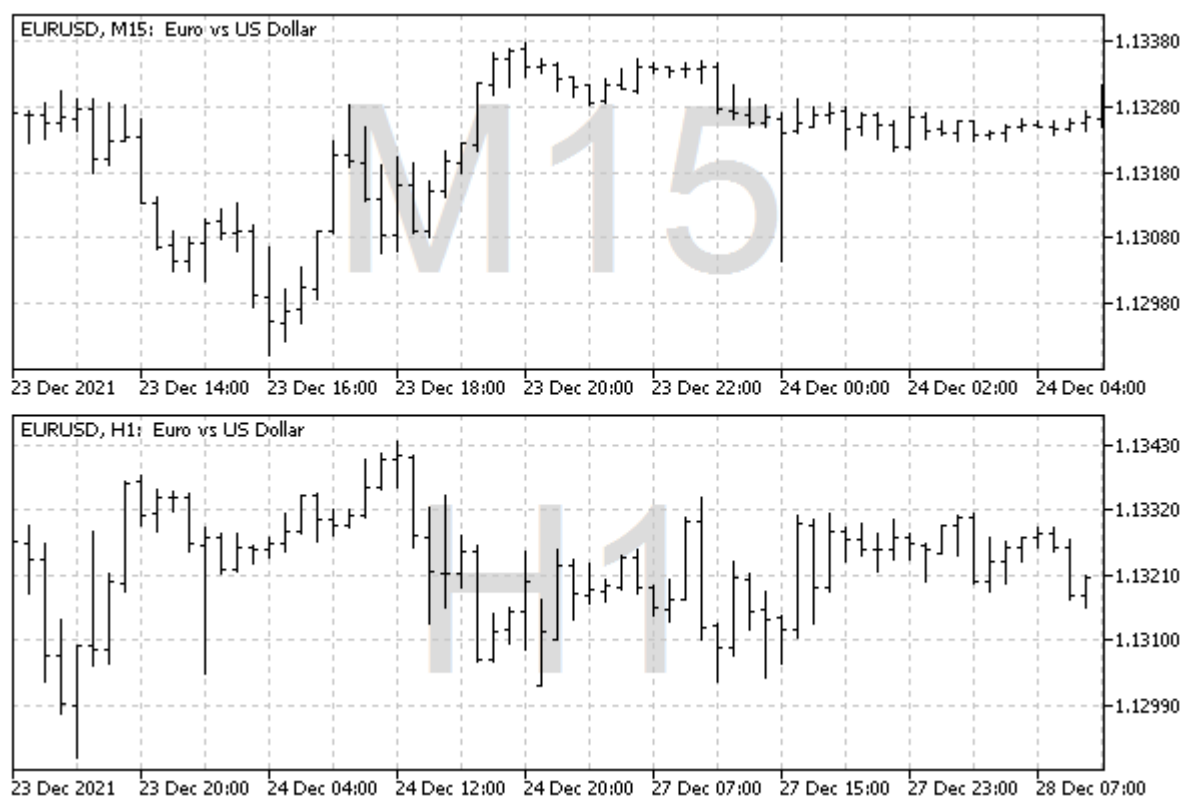
最后，为第 i 个时间范围生成正确的可见性标志，并将其写入 OBJPROP_TIMEFRAMES 特性。

```

const int flag = 1 << (i - 1);
ObjectSetInteger(0, name, OBJPROP_TIMEFRAMES, flag);
}

```

查看切换时间范围时图表上的变化。



带时间范围名称的标签

如果打开 **Object List** 对话框并在列表中启用 **All** 对象，可以轻松确认已为所有时间范围生成标签，并检查其可见性标志。

要移除对象，可运行 **ObjectCleanup1.mq5** 脚本。

5.8.19 为标签分配字符代码

正如 [与时间和价格关联的对象](#) 章节所述，**OBJ_ARROW** 标签允许在图表上显示任意的 **Wingdings** 字体符号（可用符号完整列表在 [MQL5 文档](#)）。对象本身的字符代码由整数特性 **OBJPROP_ARROW CODE** 决定。

脚本可用于展示 **ObjectWingdings.mq5** *font* 的所有字符。在该脚本中，我们通过循环创建具有不同字符的标签，并将它们逐一放置在柱线上。

```

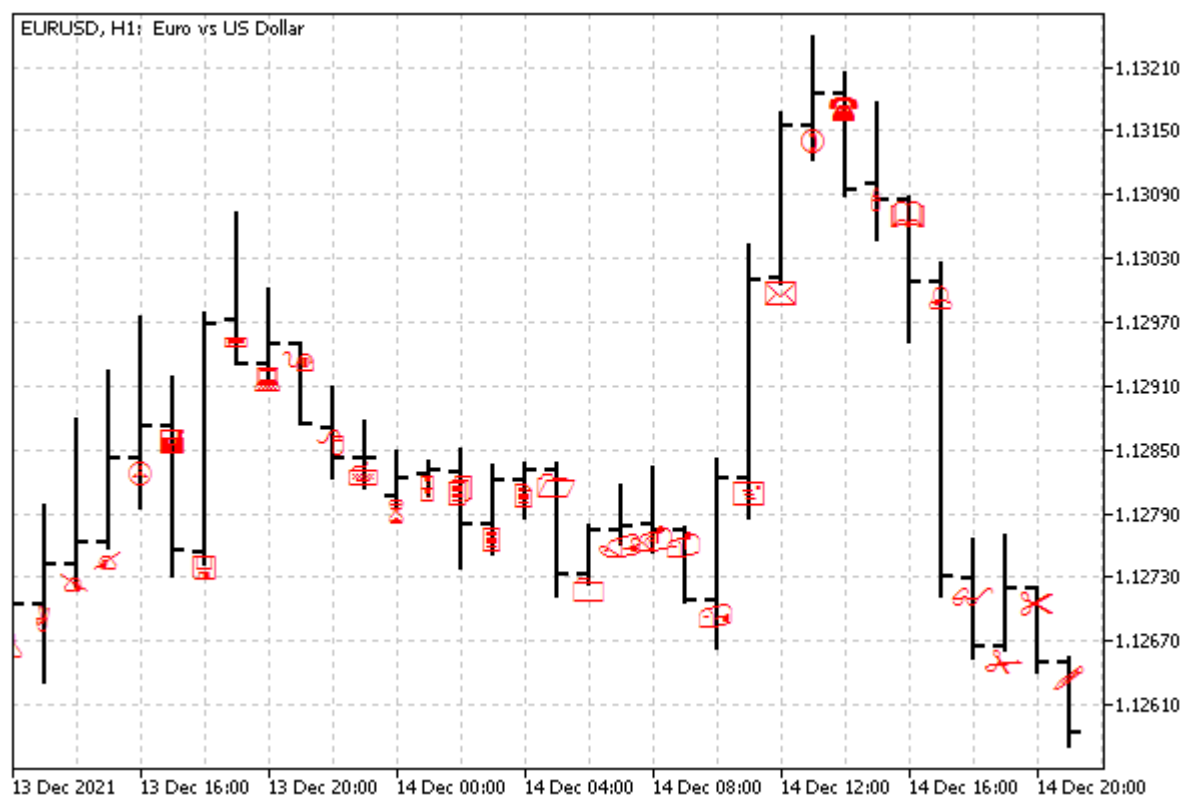
#include "ObjectPrefix.mqh"

void OnStart()
{
    for(int i = 33; i < 256; ++i) // character codes
    {
        const int b = i - 33; // bar number
        const string name = ObjNamePrefix + "Wingdings-"
            + (string)iTime(_Symbol, _Period, b);
        ObjectCreate(0, name, OBJ_ARROW,
            0, iTime(_Symbol, _Period, b), iOpen(_Symbol, _Period, b));
        ObjectSetInteger(0, name, OBJPROP_ARROWCODE, i);
    }

    PrintFormat("%d objects with arrows created", 256 - 33);
}

```

在图表上的显示效果如下截图所示。



OBJ_ARROW 标签中的 Wingdings 字符

5.8.20 直线对象体的射线特性

在图形对象中，有多种类型的对象，其锚点之间的线条可显示为线段（即严格限定在两点之间），或显示为向某个方向无限延伸的直线，贯穿整个窗口可见区域。这类对象有：

- 趋势线
- 按角度绘制的趋势线
- 所有类型的通道（等距通道、标准差通道、回归通道、Andrews 分叉线）

- Gann 线
- Fibonacci 线
- Fibonacci 通道
- Fibonacci 扩展

对于这些对象，你可以分别使用 OBJPROP_RAY_LEFT 和 OBJPROP_RAY_RIGHT 特性来单独启用向左或向右的线条延伸。此外，对于垂直线，你可以指定其是绘制在所有图表子窗口中还是仅绘制在当前子窗口中（锚点所在位置）：这一功能由 OBJPROP_RAY 特性控制。所有特性均为布尔值，这意味着它们可以被启用 (*true*) 或禁用 (*false*)。

标识符	说明
OBJPROP_RAY_LEFT	射线向左延伸
OBJPROP_RAY_RIGHT	射线向右延伸
OBJPROP_RAY	垂直线延伸到所有图表窗口

你可以使用 *ObjectRays.mq5* 脚本来检查射线的运行效果。该脚本会创建 3 个具有不同射线设置的标准差通道。

特定对象通过辅助函数 *SetupChannel* 创建并配置。通过其参数，可以设置通道长度（以柱线为单位）、通道宽度（偏差），以及左右射线的显示选项和颜色。


```

#include "ObjectPrefix.mqh"

void SetupChannel(const int length, const double deviation = 1.0,
    const bool right = false, const bool left = false,
    const color clr = clrRed)
{
    const string name = ObjNamePrefix + "Channel"
        + (right ? "R" : "") + (left ? "L" : "");
    // NB: Anchor point 0 must have an earlier time than anchor point 1,
    // otherwise the channel will degenerate
    ObjectCreate(0, name, OBJ_STDDEVCHANNEL, 0, iTime(NULL, 0, length), 0);
    ObjectSetInteger(0, name, OBJPROP_TIME, 1, iTime(NULL, 0, 0));
    // deviation
    ObjectSetDouble(0, name, OBJPROP_DEVIATION, deviation);
    // color and description
    ObjectSetInteger(0, name, OBJPROP_COLOR, clr);
    ObjectSetString(0, name, OBJPROP_TEXT, StringFormat("%2.1", deviation)
        + ((!right && !left) ? " NO RAYS" : "")
        + (right ? " RIGHT RAY" : "") + (left ? " LEFT RAY" : ""));
    // properties of rays
    ObjectSetInteger(0, name, OBJPROP_RAY_RIGHT, right);
    ObjectSetInteger(0, name, OBJPROP_RAY_LEFT, left);
    // lighting up objects by highlighting
    // (besides, it's easier for the user to remove them)
    ObjectSetInteger(0, name, OBJPROP_SELECTABLE, true);
    ObjectSetInteger(0, name, OBJPROP_SELECTED, true);
}

```

在 *OnStart* 函数中，我们针对 3 个不同的通道调用 *SetupChannel* 函数。

```

void OnStart()
{
    SetupChannel(24, 1.0, true);
    SetupChannel(48, 2.0, false, true, clrBlue);
    SetupChannel(36, 3.0, false, false, clrGreen);
}

```

结果，我们得到了以下形式的图表。



具有不同 OBJPROP_RAY_LEFT 和 OBJPROP_RAY_RIGHT 特性设置的通道

当启用射线功能后，可以通过我们将在 [获取线条上给定点的时间或价格](#) 章节中说明的函数，请求对象外推计算时间和价格值。

5.8.21 管理对象的按下状态

对于按钮 (OBJ_BUTTON) 和带图像的面板 (OBJ_BITMAP_LABEL) 等对象，终端支持一种特殊特性，可直观地将对象在正常 (释放) 状态与按下状态之间进行双向切换。预留常量 OBJPROP_STATE 便是用于这种情况。该特性为布尔类型：当值为 **true** 时，对象被视为处于按下状态；当值为 **false** 时则被视为处于释放状态 (默认)。

对于 OBJ_BUTTON，终端将自行绘制三维边框效果；而对于 OBJ_BITMAP_LABEL，程序员必须指定两张图像 (作为文件或 [资源](#))，以提供合适的外观展示。由于该特性在技术上只是一个切换开关，因此除了用于 “按下” 和 “释放” 效果外，也很容易将其用于其他用途。例如，借助合适的图像，你可以实现一个标志 (选项) 功能。

将在下一节中讨论如何在对象中使用图像。

在交互式 MQL 程序中，对象状态通常会随用户操作 (特别是鼠标点击) 而改变。我们将在 [事件](#) 一章中讨论这种可能性。

现在让我们在静态模式下，通过简单按钮来测试该特性。*ObjectButtons.mq5* 脚本会在图表上创建两个按钮：一个处于按下状态，另一个处于释放状态。

单个按钮的设置工作由 *SetupButton* 函数完成，该函数通过参数指定按钮的名称、文本、坐标、大小和状态。

```

#include "ObjectPrefix.mqh"

void SetupButton(const string button,
    const int x, const int y,
    const int dx, const int dy,
    const bool state = false)
{
    const string name = ObjNamePrefix + button;
    ObjectCreate(0, name, OBJ_BUTTON, 0, 0, 0);
    // position and size
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
    ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
    ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
    // label on the button
    ObjectSetString(0, name, OBJPROP_TEXT, button);

    // pressed (true) / released (false)
    ObjectSetInteger(0, name, OBJPROP_STATE, state);
}

```

然后在 *OnStart* 函数中调用该函数两次。

```

void OnStart()
{
    SetupButton("Pressed", 100, 100, 100, 20, true);
    SetupButton("Normal", 100, 150, 100, 20);
}

```

以下是脚本运行后，两个按钮在图表上的预期显示效果示意图：



按下和释放状态的 OBJ_BUTTON 按钮

有趣的是，你可以用鼠标点击任意一个按钮，按钮将改变其状态。不过，我们尚未探讨如何拦截相关状态变更通知。

需特别注意，只有当对象特性中选中 *Disable selection* 选项时，才会执行这种自动状态切换，而通过编程创建的所有对象默认均满足此条件。需注意，必要时可通过显式设置 OBJPROP_SELECTABLE 特性为 *true* 启用此选择功能。我们在先前的一些示例中已使用过该特性。

要移除不再需要的按钮，请使用 *ObjectCleanup1.mq5* 脚本。

5.8.22 调整位图对象中的图像

OBJ_BITMAP_LABEL 类型的对象（基于屏幕坐标定位的图片面板）支持显示位图图像。位图图像指的是 BMP 图形格式：尽管理论上存在许多其他栅格格式（例如 PNG 或 GIF），但目前 MQL5 不支持这些格式，也不支持矢量格式。

字符串特性 OBJPROP_BMPFILE 允许为对象指定图像。该特性必须包含 BMP 文件或 [资源](#) 的名称。

由于此对象支持双位状态切换（请参阅 [OBJPROP_STATE](#)），因此应针对它使用修饰符参数：索引 0 下设置“开启/按下”状态的图片，索引 1 下设置“关闭/释放”状态的图片。如果仅指定一张图片（无修饰符，等同于索引 0），该图片将用于两种状态。对象的默认状态为“关闭/释放”状态。

对象的大小会与图像大小相同，但可通过在 OBJPROP_XSIZE 和 OBJPROP_YSIZE 特性中指定较小值来更改对象大小：这种情况下，仅显示图像的一部分（详情见下一章节 [帧设置](#)）。

OBJPROP_BMPFILE 字符串的长度不得超过 63 个字符。该字符串不仅可以包含文件名，还可以包含文件路径。如果字符串以路径分隔符（正斜杠 ‘/’ 或双反斜杠 ‘\\’）开头，则会相对于

`terminal_data_directory/MQL5/`目录搜索文件。否则，将相对于 `MQL` 程序所在的文件夹搜索文件。

例如，字符串 `"\\Images\\euro.bmp"` 或 `"Images/euro.bmp"` 指向的是 `MQL5/Images/euro.bmp` 目录下的文件。标准终端安装包在 `MQL5` 目录中包含 `Images` 文件夹，并提供了 `euro.bmp` 和 `dollar.bmp` 两个测试文件，因此该路径有效。如果指定字符串 `"Images\\euro.bmp"` 或 `"Images/euro.bmp"`，则对于从 `MQL5/Scripts/MQL5Book/` 目录启动的脚本，意味着包含 `euro.bmp` 文件的 `Images` 文件夹应直接位于该目录下，即完整路径为 `MQL5/Scripts/MQL5Book/Images/euro.bmp`。本书没有此类文件，这会导致图像加载错误。但将图形文件与程序放在一起有其优势：更易于控制程序组装过程，还能避免不同程序的图像文件混淆。

`ObjectBitmap.mqh` 脚本会在图表上创建带图像的面板，并为其分配两张图片：`"\\Images\\dollar.bmp"` 和 `"\\Images\\euro.bmp"`。

```
#include "ObjectPrefix.mqh"

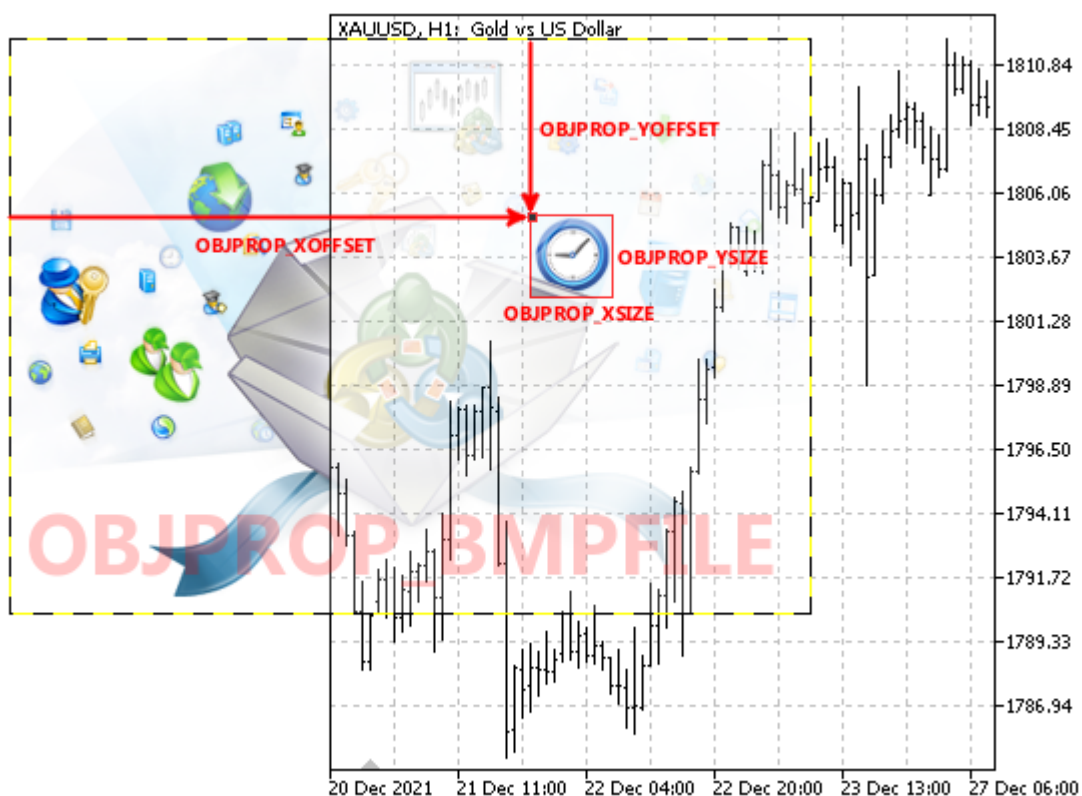
void SetupBitmap(const string button, const int x, const int y,
    const string imageOn, const string imageOff = NULL)
{
    // creating a panel
    const string name = ObjNamePrefix + "Bitmap";
    ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
    // set position
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
    // include images
    ObjectSetString(0, name, OBJPROP_BMPFILE, 0, imageOn);
    if(imageOff != NULL) ObjectSetString(0, name, OBJPROP_BMPFILE, 1, imageOff);
}

void OnStart()
{
    SetupBitmap("image", 100, 100,
        "\\Images\\dollar.bmp", "\\Images\\euro.bmp");
}
```

与上一节中脚本的结果相同，此处也可以点击图片对象，查看其在美元和欧元图像之间切换。

5.8.23 图像的裁剪（输出部分）

对于带有图片的图形对象 `OBJ_BITMAP_LABEL` 和 `OBJ_BITMAP`，`MQL5` 允许通过 `OBJPROP_BMPFILE` 特性指定要显示的图像部分。要实现这一点，需要将对象的大小 (`OBJPROP_XSIZE` 和 `OBJPROP_YSIZE`) 设置为小于图像尺寸，并使用整数特性 `OBJPROP_XOFFSET` 和 `OBJPROP_YOFFSET` 设置可见矩形区域的左上角坐标。这两个特性分别设置从原始图像左边界和上边界开始的 X 轴和 Y 轴像素偏移量。



将图像的一部分输出到一个对象

通常，工具栏图标（按钮组、菜单等）会使用大型图像的局部区域来实现类似效果：相比用多个小文件储存单独图标，用一个文件储存所有图标更能高效利用资源。

测试脚本 *ObjectBitmapOffset.mq5* 创建了多个带图片的面板（OBJ_BITMAP_LABEL），并且为所有这些面板在 OBJPROP_BMPFILE 特性中指定了相同的图形文件。然而，通过设置 OBJPROP_XOFFSET 和 OBJPROP_YOFFSET 特性，所有对象将显示图像的不同部分。

```

void SetupBitmap(const int i, const int x, const int y, const int size,
    const string imageOn, const string imageOff = NULL)
{
    // create an object
    const string name = ObjNamePrefix + "Tool-" + (string)i;
    ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
    ObjectSetInteger(0, name, OBJPROP_CORNER, CORNER_RIGHT_UPPER);
    ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_RIGHT_UPPER);
    // position and size
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
    ObjectSetInteger(0, name, OBJPROP_XSIZE, size);
    ObjectSetInteger(0, name, OBJPROP_YSIZE, size);
    // offset in the original image, according to which the i-th fragment is read
    ObjectSetInteger(0, name, OBJPROP_XOFFSET, i * size);
    ObjectSetInteger(0, name, OBJPROP_YOFFSET, 0);
    // generic image (file)
    ObjectSetString(0, name, OBJPROP_BMPFILE, imageOn);
}

void OnStart()
{
    const int icon = 46; // size of one icon
    for(int i = 0; i < 7; ++i) // loop through the icons in the file
    {
        SetupBitmap(i, 10, 10 + i * icon, icon,
            "\\Files\\MQL5Book\\icons-322-46.bmp");
    }
}

```

原始图像包含多个小图标，每个图标的尺寸为 46×46 像素。该脚本会将这些图标逐一“裁剪”出来，并垂直排列在窗口的右侧边缘。

以下展示了一个通用文件 (*Files/MQL5Book/icons-322-46.bmp*)，以及在图表上的呈现效果。



含图标的 BMP 文件



图表上带图标的按钮对象

5.8.24 输入字段特性：文本对齐和只读

对于类型为 OBJ_EDIT (输入字段)的对象，MQL 程序可以使用 *ObjectSetInteger/ObjectGetInteger* 函数设置两个特定特性。

标识符	说明	值类型
OBJPROP_ALIGN	水平文本对齐	ENUM_ALIGN_MODE
OBJPROP_READONLY	文本编辑能力	bool

ENUM_ALIGN_MODE 枚举包含以下成员。

标识符	说明
ALIGN_LEFT	左对齐
ALIGN_CENTER	居中对齐
ALIGN_RIGHT	右对齐

请注意，与 OBJ_TEXT 和 OBJ_LABEL 对象不同，输入字段不会自动调整自身大小以适应输入的文本，因此，对于长字符串，您可能需要显式设置 OBJPROP_XSIZE 特性。

在编辑模式下，输入字段内支持文本的水平滚动。

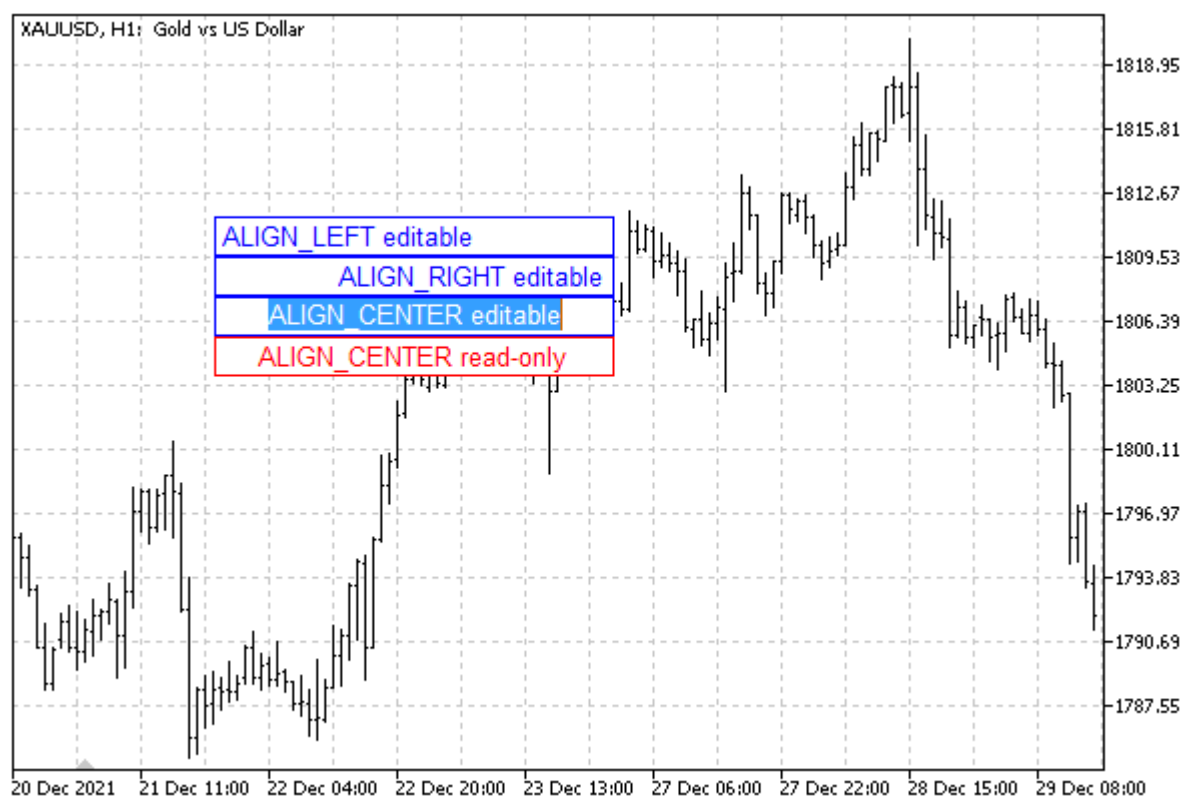
*ObjectEdit.mq5*脚本会创建四个 OBJ_EDIT 对象：其中三个可编辑且采用不同的文本对齐方式，第四个处于只读模式。

```
#include "ObjectPrefix.mqh"

void SetupEdit(const int x, const int y, const int dx, const int dy,
    const ENUM_ALIGN_MODE alignment = ALIGN_LEFT, const bool readonly = false)
{
    // create an object with a description of the properties
    const string props = EnumToString(alignment)
        + (readonly ? " read-only" : " editable");
    const string name = ObjNamePrefix + "Edit" + props;
    ObjectCreate(0, name, OBJ_EDIT, 0, 0, 0);
    // position and size
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
    ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
    ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
    // specific properties of input fields
    ObjectSetInteger(0, name, OBJPROP_ALIGN, alignment);
    ObjectSetInteger(0, name, OBJPROP_READONLY, readonly);
    // colors (different depending on editability)
    ObjectSetInteger(0, name, OBJPROP_BGCOLOR, clrWhite);
    ObjectSetInteger(0, name, OBJPROP_COLOR, readonly ? clrRed : clrBlue);
    // content
    ObjectSetString(0, name, OBJPROP_TEXT, props);
    // tooltip for editable
    ObjectSetString(0, name, OBJPROP_TOOLTIP,
        (readonly ? "\n" : "Click me to edit"));
}

void OnStart()
{
    SetupEdit(100, 100, 200, 20);
    SetupEdit(100, 120, 200, 20, ALIGN_RIGHT);
    SetupEdit(100, 140, 200, 20, ALIGN_CENTER);
    SetupEdit(100, 160, 200, 20, ALIGN_CENTER, true);
}
```

脚本的运行结果如下图所示。



不同模式下的输入字段

可以点击任何可编辑字段并更改其内容。

5.8.25 标准偏差通道宽度

标准差通道 `OBJ_STDDEVCHANNEL` 有一个特殊特性，用于将通道宽度定义为标准 (均方根) 偏差的倍数。该特性称为 `OBJPROP_DEVIATION`，可接受正实数值 (*double*)。默认情况下，该值等于 1.0。

我们已经在 [直线对象的射线特性](#) 章节的 `Object Rays.mq5` 脚本中看到该特性的使用示例。

5.8.26 在水平对象中设置水平位

一些图形对象是使用多个水平位 (重复线条) 构建的。这些包括：

- Andrews 分叉线 `OBJ_PITCHFORK`
- Fibonacci 工具：
 - `OBJ_FBO` 水平位
 - 时区 `OBJ_FBOTIMES`
 - 扇形线 `OBJ_FBOFAN`
 - 弧线 `OBJ_FBOARC`
 - 通道 `OBJ_FBOCHANNEL`
 - 扩展线 `OBJ_EXPANSION`

MQL5 允许你为这些对象设置水平位特性。这些特性包括它们的数量、颜色、数值和标签。

标识符	说明	类型
OBJPROP_LEVELS	水平线数量	int
OBJPROP_LEVELCOLOR	水平线颜色	color
OBJPROP_LEVELSTYLE	水平线样式	ENUM_LINE_STYLE
OBJPROP_LEVELWIDTH	水平线宽度	int
OBJPROP_LEVELTEXT	水平线描述	string
OBJPROP_LEVELVALUE	水平线值	double

在调用 *ObjectGet* 和 *ObjectSet* 函数处理除 OBJPROP_LEVELS 外的所有特性时，需要提供一个额外的修饰符参数，用于指定具体级别编号。

作为示例，我们来看看指标 *ObjectHighLowFibo.mq5*。对于给定的柱形范围（由最后一根柱线的编号 (*baroffset*) 及其左侧的柱数 (*BarCount*) 定义），该指标会先查找 *High* 和 *Low* 价格，然后为这些价格点创建 OBJ_FIBO 对象。随着新柱线的形成，Fibonacci 水平位将向右移动，以贴合更近期的价格。

```

#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots 0

#include <MQL5Book/ColorMix.mqh>

input int BarOffset = 0;
input int BarCount = 24;

const string Prefix = "HighLowFibo-";

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &price[])
{
    static datetime now = 0;
    if(now != iTime(NULL, 0, 0))
    {
        const int hh = iHighest(NULL, 0, MODE_HIGH, BarCount, BarOffset);
        const int ll = iLowest(NULL, 0, MODE_LOW, BarCount, BarOffset);

        datetime t[2] = {iTime(NULL, 0, hh), iTime(NULL, 0, ll)};
        double p[2] = {iHigh(NULL, 0, hh), iLow(NULL, 0, ll)};

        DrawFibo(Prefix + "Fibo", t, p, clrGray);

        now = iTime(NULL, 0, 0);
    }
    return rates_total;
}

```

对象的直接设置是在辅助函数 *DrawFibo* 中完成的。在该函数中，水平位特别采用彩虹色进行绘制，而其样式和粗细则基于相应数值是否为“整数”（无小数部分）来决定。

```

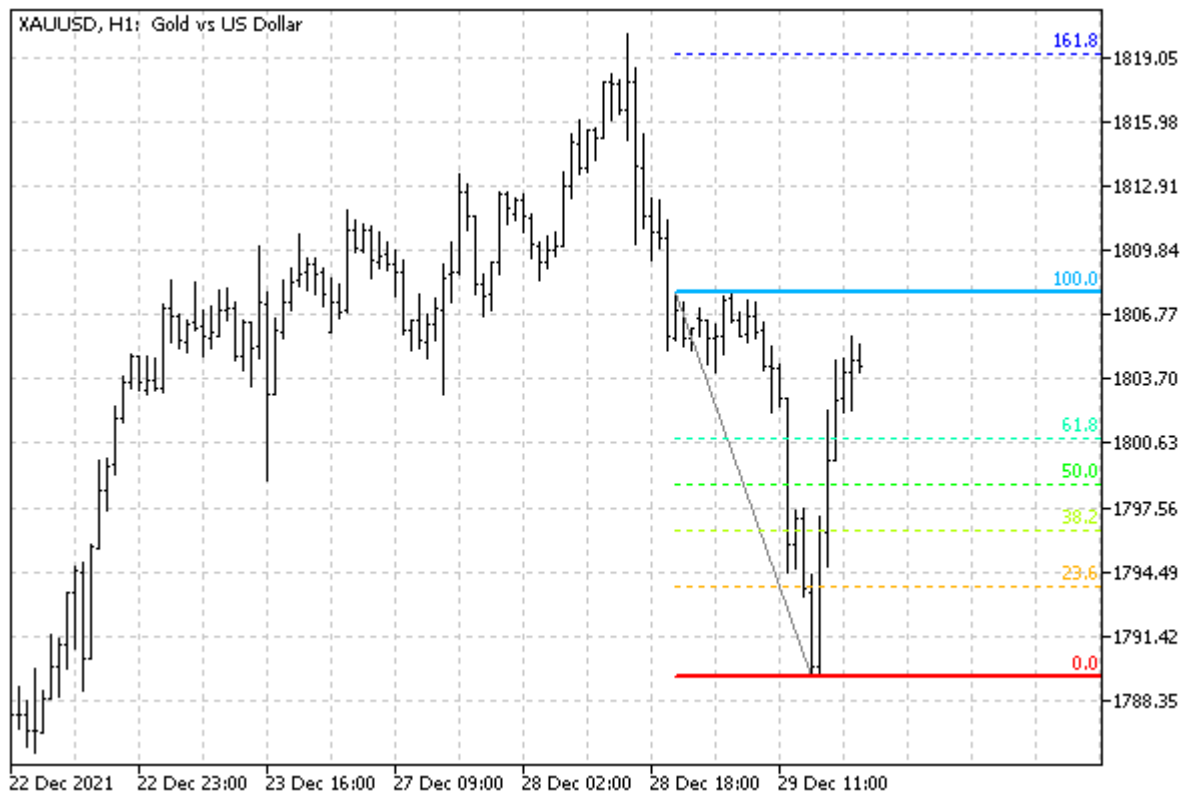
bool DrawFibo(const string name, const datetime &t[], const double &p[],
             const color clr)
{
    if(ArraySize(t) != ArraySize(p)) return false;

    ObjectCreate(0, name, OBJ_FIBO, 0, 0, 0);
    // anchor points
    for(int i = 0; i < ArraySize(t); ++i)
    {
        ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
        ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
    }
    // general settings
    ObjectSetInteger(0, name, OBJPROP_COLOR, clr);
    ObjectSetInteger(0, name, OBJPROP_RAY_RIGHT, true);
    // level settings
    const int n = (int)ObjectGetInteger(0, name, OBJPROP_LEVELS);
    for(int i = 0; i < n; ++i)
    {
        const color gradient = ColorMix::RotateColors(ColorMix::HSVtoRGB(0),
            ColorMix::HSVtoRGB(359), n, i);
        ObjectSetInteger(0, name, OBJPROP_LEVELCOLOR, i, gradient);
        const double level = ObjectGetDouble(0, name, OBJPROP_LEVELVALUE, i);
        if(level - (int)level > DBL_EPSILON * level)
        {
            ObjectSetInteger(0, name, OBJPROP_LEVELSTYLE, i, STYLE_DOT);
            ObjectSetInteger(0, name, OBJPROP_LEVELWIDTH, i, 1);
        }
        else
        {
            ObjectSetInteger(0, name, OBJPROP_LEVELSTYLE, i, STYLE_SOLID);
            ObjectSetInteger(0, name, OBJPROP_LEVELWIDTH, i, 2);
        }
    }

    return true;
}

```

以下是该对象在图表中的一种可能呈现样式。



带水平位设置的 Fibonacci 对象

5.8.27 Gann、Fibonacci 和 Elliot 对象的附加特性

Gann、Fibonacci 和 Elliot 对象具有特定且唯一的特性。根据特性类型，需使用 *ObjectGetInteger/ ObjectSetInteger* 或 *ObjectGetDouble/ ObjectSetDouble* 函数。

标识符	说明	类型
OBJPROP_DIRECTION	Gann 对象趋势 (Fan OBJ_GANNFAN 和 Grid OBJ_GANNGRID)	ENUM_GANN_DIRECTION
OBJPROP_DEGREE	Elliot 波浪等级	ENUM_ELLIOT_WAVE_DEGREE
OBJPROP_DRAWLINES	显示 Elliot 波浪等级的线条	bool
OBJPROP_SCALE	每根柱线的点数比例 (Gann 对象和 FibonacciArcs 对象的特性)	double
OBJPROP_ELLIPSE	FibonacciArcs 弧线对象的全椭圆显示 (OBJ_FBOARC)	bool

ENUM_GANN_DIRECTION 枚举具有以下成员：

常量	趋势方向
GANN_UP_TREND	上升线
GANN_DOWN_TREND	下降线

ENUM_ELLIOT_WAVE_DEGREE 枚举用于设置 Elliott 波浪的大小 (标注方法)。

常量	说明
ELLIOTT_GRAND_SUPERCYCLE	超大循环浪
ELLIOTT_SUPERCYCLE	大循环浪
ELLIOTT_CYCLE	循环浪
ELLIOTT_PRIMARY	基本循环浪
ELLIOTT_INTERMEDIATE	中型浪
ELLIOTT_MINOR	小型循环浪
ELLIOTT_MINUTE	细浪
ELLIOTT_MINUETTE	微浪
ELLIOTT_SUBMINUETTE	次微浪

5.8.28 图表对象

图表对象 OBJ_CHART 允许在图表内部为其他金融工具和时间范围创建缩略图。图表对象包含在常规 图表列表 中，我们可以使用 *ChartFirst*和*ChartNext* 函数以编程方式获得。正如 检查主窗口状态 章节所述，特殊图表特性 CHART_IS_OBJECT 可通过标识符判断该图表是完整窗口还是图表对象。在后一种情况下，调用 *ChartGetInteger(id, CHART_IS_OBJECT)*将返回 *true*。

图表对象具有一组专属特性集。

标识符	说明	类型
OBJPROP_CHART_ID	图表 ID (只读)	long
OBJPROP_PERIOD	图表周期	ENUM_TIMEFRAMES
OBJPROP_DATE_SCALE	显示时间轴	bool
OBJPROP_PRICE_SCALE	显示价格轴	bool
OBJPROP_CHART_SCALE	缩放比例 (值的范围是 0-5)	int
OBJPROP_SYMBOL	交易品种	string

通过 OBJPROP_CHART_ID 特性获取的标识符，允许使用 [处理图表](#) 章节中的函数像管理常规图表一样管理该对象。但存在一些限制：

- 无法使用以下函数关闭该对象：*ChartClose*
- 无法使用 *ChartSetSymbolPeriod* 函数更改对象中的交易品种/周期
- 以下特性无法在对象中修改：*CHART_SCALE*、*CHART_BRING_TO_TOP*、*CHART_SHOW_DATE_SCALE* 和 *CHART_SHOW_PRICE_SCALE*。

默认情况下，所有特性（除 OBJPROP_CHART_ID 外）均与当前窗口的对应特性一致。

图表对象的演示通过无缓冲指标 *ObjectChart.mq5* 实现。该指标会创建一个子窗口，其中包含两个图表对象，这两个对象显示与当前图表相同交易品种，但时间周期分别为当前时间范围的上一个范围和下一个范围。

对象会吸附到子窗口的右上角，并具有相同的预设尺寸：

```
#define SUBCHART_HEIGHT 150
#define SUBCHART_WIDTH 200
```

当然，子窗口的高度必须与对象高度匹配，直到能够自适应地响应尺寸调整事件。

```
#property indicator_separate_window
#property indicator_height SUBCHART_HEIGHT
#property indicator_buffers 0
#property indicator_plots 0
```

在 *SetupSubChart* 函数中配置了一个迷你图表，该函数将对象编号、对象尺寸和所需时间范围作为输入。*SetupSubChart* 函数的执行结果是图表对象的标识符，我们仅将其输出到日志中以供参考。

```
void OnInit()
{
    Print(SetupSubChart(0, SUBCHART_WIDTH, SUBCHART_HEIGHT, PeriodUp(_Period)));
    Print(SetupSubChart(1, SUBCHART_WIDTH, SUBCHART_HEIGHT, PeriodDown(_Period)));
}
```

宏 *PeriodUp* 和 *PeriodDown* 使用辅助函数 *PeriodRelative*。


```

#define PeriodUp(P) PeriodRelative(P, +1)
#define PeriodDown(P) PeriodRelative(P, -1)

ENUM_TIMEFRAMES PeriodRelative(const ENUM_TIMEFRAMES tf, const int step)
{
    static const ENUM_TIMEFRAMES stdtfs[] =
    {
        PERIOD_M1, // =1 (1)
        PERIOD_M2, // =2 (2)
        ...
        PERIOD_W1, // =32769 (8001)
        PERIOD_MN1, // =49153 (C001)
    };
    const int x = ArrayBsearch(stdtfs, tf == PERIOD_CURRENT ? _Period : tf);
    const int needle = x + step;
    if(needle >= 0 && needle < ArraySize(stdtfs))
    {
        return stdtfs[needle];
    }
    return tf;
}

```

下面是主要工作函数 *SetupSubChart*。

```

long SetupSubChart(const int n, const int dx, const int dy,
    ENUM_TIMEFRAMES tf = PERIOD_CURRENT, const string symbol = NULL)
{
    // create an object
    const string name = Prefix + "Chart-"
        + (symbol == NULL ? _Symbol : symbol) + PeriodToString(tf);
    ObjectCreate(0, name, OBJ_CHART, ChartWindowFind(), 0, 0);

    // anchor to the top right corner of the subwindow
    ObjectSetInteger(0, name, OBJPROP_CORNER, CORNER_RIGHT_UPPER);
    // position and size
    ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
    ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, (n + 1) * dx);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, 0);

    // specific chart settings
    if(symbol != NULL)
    {
        ObjectSetString(0, name, OBJPROP_SYMBOL, symbol);
    }

    if(tf != PERIOD_CURRENT)
    {
        ObjectSetInteger(0, name, OBJPROP_PERIOD, tf);
    }
    // disable the display of lines
    ObjectSetInteger(0, name, OBJPROP_DATE_SCALE, false);
    ObjectSetInteger(0, name, OBJPROP_PRICE_SCALE, false);
    // add the MA indicator to the object by its id just for demo
    const long id = ObjectGetInteger(0, name, OBJPROP_CHART_ID);
    ChartIndicatorAdd(id, 0, iMA(NULL, tf, 10, 0, MODE_EMA, PRICE_CLOSE));
    return id;
}

```

对于图表对象，锚点始终固定在对象的左上角，因此当锚定到窗口的右角时，需要添加对象的宽度（这通过在 OBJPROP_XDISTANCE 的表达式 $(n+1)*dx$ 中执行 +1 操作来实现）。

下图展示了该指标在 XAUUSD,H1 图表上的运行结果。



指标子窗口中的两个图表对象

可以看到，迷你图表显示 M30 和 H2 时间范围。

需要特别注意，可以向图表对象添加指标并应用 `tpl` 模板，包括带 EA 交易的模板。但无法在图表对象内部创建对象。

当图表对象因当前时间范围或全部时间范围禁用可视化而被隐藏时，其内部图表的 `CHART_WINDOW_IS_VISIBLE` 特性仍会返回 `true`。

5.8.29 移动对象

要在时间/价格坐标中移动对象，不仅可以使使用更改特性的 `ObjectSet` 函数，还可以使用特殊函数 `ObjectMove`，该函数可更改对象指定锚点的坐标。

`bool ObjectMove(long chartId, const string name, int index, datetime time, double price)`

`chartId` 参数用于设置图表 ID (0 表示当前图表)。`name` 参数中传入对象的名称。锚点索引和坐标分别通过 `index`、`time` 和 `price` 参数指定。

该函数使用异步调用，即它只是将命令发送到图表的事件队列，但不会等待对象实际完成移动。

该函数返回命令是否成功加入队列的标识 (在这种情况下，结果为 `true` 表示成功)。应通过调用 `ObjectGet` 函数获取对象的实际位置。

在指标 `ObjectHighLowFibo.mq5` 中，我们对 `DrawFibo` 函数进行修改以启用 `ObjectMove`。在遍历锚点的循环中，我们不再使用两次对 `ObjectSet` 函数的调用，而是改为使用一次 `ObjectMove` 调用：

```

bool DrawFibo(const string name, const datetime &t[], const double &p[],
             const color clr)
{
    ...
    for(int i = 0; i < ArraySize(t); ++i)
    {
        // was:
        // ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
        // ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
        // became:
        ObjectMove(0, name, i, t[i], p[i]);
    }
    ...
}

```

当锚点的两个坐标都需要更改时，使用 *ObjectMove* 函数是合理的。在某些情况下，只有一个坐标会产生影响（例如，在标准差通道和线性回归通道的锚点处，只有起始和结束日期/时间至关重要，通道会自动计算这些点的价格值）。在这种情况下，调用一次 *ObjectSet* 函数比使用 *ObjectMove* 更合适。

5.8.30 获取指定线条点的时间或价格坐标

许多图形对象包含一条或多条直线。MQL5 允许对这些直线上的点进行插值和外推，并实现坐标间的相互转换，例如通过时间获取价格，或通过价格获取时间。

插值始终可用：其作用范围在对象“内部”，即在锚点之间。仅当对象在相应方向上的射线特性被启用时，才可以在对象外部进行外推（请参阅 [直线对象的射线特性](#)）。

ObjectGetValueByTime 函数返回指定时间的价格值。*ObjectGetTimeByValue* 函数返回指定价格的时间值。

```

double ObjectGetValueByTime(long chartId, const string name, datetime time, int line)
datetime ObjectGetTimeByValue(long chartId, const string name, double value, int line)

```

计算是针对指定标识符为 *chartId* 的图表上名为 *name* 的对象进行的。*time* 和 *value* 参数用于指定已知坐标，基于这个已知坐标来计算未知坐标。由于一个对象可能包含多条线条，因此一个坐标可能对应多个值，因此需要在 *line* 参数中指定线条编号。

该函数返回指定初始坐标相对于线条的投影点所对应的价格或时间值。

如果发生错误，将返回 0，并将错误代码写入 *_LastError*。例如，尝试对未启用射线特性的直线进行外推时，将生成 OBJECT_GETVALUE_FAILED (4205) 错误。

这些函数适用于以下对象：

- 趋势线 (OBJ_TREND)
- 角度趋势线 (OBJ_TRENDBYANGLE)
- Gann 线 (OBJ_GANNLN)
- 等距通道 (OBJ_CHANNEL), 2 条线
- 线性回归通道 (OBJ_REGRESSION), 3 条线
- 标准差通道 (OBJ_STDDEVCHANNEL), 3 条线
- 带箭头直线 (OBJ_ARROWED_LINE)

我们将通过无缓冲指标 *ObjectChannels.mq5* 验证函数的运行情况。该指标会创建两个对象，分别为标准差通道和线性回归通道，随后请求获取未来柱线上通道上轨和下轨的价格，并将这些价格显示在图表注释中。对于标准差通道，已启用 OBJPROP_RAY_RIGHT 特性，但线性回归通道未启用该特性（有意为之）。因此，无法从第二个通道获取值，其在屏幕上始终显示为零。

随着新柱线的形成，通道将自动向右移动。通道长度通过输入参数 *WorkPeriod* 设置（默认 10 根柱线）。

```
input int WorkPeriod = 10;

const string Prefix = "ObjChnl-";
const string ObjStdDev = Prefix + "StdDev";
const string ObjRegr = Prefix + "Regr";

void OnInit()
{
    CreateObjects();
    UpdateObjects();
}
```

CreateObjects 函数创建 2 个通道并进行初始设置。

```
void CreateObjects()
{
    ObjectCreate(0, ObjStdDev, OBJ_STDDEVCHANNEL, 0, 0, 0);
    ObjectCreate(0, ObjRegr, OBJ_REGRESSION, 0, 0, 0);
    ObjectSetInteger(0, ObjStdDev, OBJPROP_COLOR, clrBlue);
    ObjectSetInteger(0, ObjStdDev, OBJPROP_RAY_RIGHT, true);
    ObjectSetInteger(0, ObjRegr, OBJPROP_COLOR, clrRed);
    // NB: ray is not enabled for the regression channel (intentionally)
}
```

UpdateObjects 函数将通道移动至最后 *WorkPeriod* 根柱线处。

```
void UpdateObjects()
{
    const datetime t0 = iTime(NULL, 0, WorkPeriod);
    const datetime t1 = iTime(NULL, 0, 0);

    // we don't use ObjectMove because channels work
    // only with time coordinate (price is calculated automatically)
    ObjectSetInteger(0, ObjStdDev, OBJPROP_TIME, 0, t0);
    ObjectSetInteger(0, ObjStdDev, OBJPROP_TIME, 1, t1);
    ObjectSetInteger(0, ObjRegr, OBJPROP_TIME, 0, t0);
    ObjectSetInteger(0, ObjRegr, OBJPROP_TIME, 1, t1);
}
```

在 *OnCalculate* 处理程序中，我们会在新柱线上更新通道位置，每次收到分时报价时，我们便调用 *DisplayObjectData* 获取外推并将其显示为注释。

```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &price[])
{
    static datetime now = 0;
    if(now != iTime(NULL, 0, 0))
    {
        UpdateObjects();
        now = iTime(NULL, 0, 0);
    }

    DisplayObjectData();

    return rates_total;
}

```

在 *DisplayObjectData* 函数中，我们将在中线上的锚点上找到价格 (OBJPROP PRICE)。此外，我们将使用 *ObjectGetValueByTime* 请求未来 *WorkPeriod* 根柱线处通道上下轨的价格值。

```

void DisplayObjectData()
{
    const double p0 = ObjectGetDouble(0, ObjStdDev, OBJPROP_PRICE, 0);
    const double p1 = ObjectGetDouble(0, ObjStdDev, OBJPROP_PRICE, 1);

    // the following equalities are always true due to the channel calculation algorithm
    // - the middle lines of both channels are the same,
    // - anchor points always lie on the middle line,
    // ObjectGetValueByTime(0, ObjStdDev, iTime(NULL, 0, 0), 0) == p1
    // ObjectGetValueByTime(0, ObjRegr, iTime(NULL, 0, 0), 0) == p1

    // trying to extrapolate future prices from the upper and lower lines
    const double d1 = ObjectGetValueByTime(0, ObjStdDev, iTime(NULL, 0, 0)
        + WorkPeriod * PeriodSeconds(), 1);
    const double d2 = ObjectGetValueByTime(0, ObjStdDev, iTime(NULL, 0, 0)
        + WorkPeriod * PeriodSeconds(), 2);

    const double r1 = ObjectGetValueByTime(0, ObjRegr, iTime(NULL, 0, 0)
        + WorkPeriod * PeriodSeconds(), 1);
    const double r2 = ObjectGetValueByTime(0, ObjRegr, iTime(NULL, 0, 0)
        + WorkPeriod * PeriodSeconds(), 2);

    // display all received prices in a comment
    Comment(StringFormat("%.2f %.2f\ndev: up=%.2f dn=%.2f\nreg: up=%.2f dn=%.2f",
        _Digits, p0, _Digits, p1,
        _Digits, d1, _Digits, d2,
        _Digits, r1, _Digits, r2));
}

```

需要注意的是，由于线性回归通道未启用射线特性，其在未来时段始终返回零值（不过如果请求的是通道时间周期内的价格，则会得到正确值）。



通道及其线条上各点的价格值

在此，对于长度为 10 根柱线的通道，外推计算也会向前进行 10 根柱线，这使得“dev:”行中显示的未来值大致对应窗口的右边框

5.9 图表上的交互事件

MetaTrader 5 的图表不仅能直观展示数据，还是 MQL 程序的运行环境，同时还支持交互式事件机制，该机制让程序可以对用户操作及其他程序做出响应。这通过一种特殊事件类型 **OnChartEvent** 实现，我们已经在 [事件处理函数概述](#) 章节中讨论过该类型。

任何指标或 EA 交易均可接收此类事件，前提是在代码中定义了具有预定义签名的同名事件处理函数。在之前介绍的部分指标示例中，我们已经利用过这一功能。本章将详细探讨事件系统。

当用户执行以下图表操作时，客户端终端将生成 **OnChartEvent** 事件：

- 更改图表尺寸或设置
- 图表窗口处于焦点状态时的按键操作
- 鼠标光标移动
- 鼠标点击图表
- 鼠标点击图形对象
- 创建图形对象
- 删除图形对象
- 用鼠标移动图形对象
- 完成对 OBJ_EDIT 对象输入字段中测试内容的编辑

MQL 程序仅接收其运行所在图表触发的上述事件。与其他事件类型一样，这些事件会被添加到队列中。所有事件按到达顺序逐个处理。如果 MQL 程序的事件队列中已存在特定类型的 *OnChartEvent* 事件，或该事件正在处理中，则同类型的新事件将不会加入队列（将被丢弃）。

部分事件类型始终处于激活状态，而另一部分事件类型默认处于禁用状态，必须通过调用 *ChartSetInteger* 设置相应的图表特性来显式启用。这些禁用事件尤其包括鼠标移动和鼠标滚轮滚动。它们的特点是可能产生大量事件流，为节省资源，建议仅在必要时启用。

除标准事件外，还有“自定义事件”的概念。这类事件的参数意义和参数内容由 MQL 程序自身定义和解释（如果涉及多个程序交互，则由一组程序共同定义）。MQL 程序可通过 *EventChartCustom* 函数向某一图表（包括其他图表）发送“用户事件”这类事件同样由 *OnChartEvent* 函数处理。

如果图表上存在多个包含 *OnChartEvent* 处理程序的 MQL 程序，这些程序将接收同一事件流。

所有 MQL 程序均在应用程序主线程之外的线程中运行。终端主线程负责处理所有 Windows 系统消息，而处理这些消息的结果，又会为其自身应用程序生成 Windows 消息。例如，用鼠标拖动图表时会生成多个 WM_MOUSE_MOVE 系统消息（从 Windows API 角度），用于应用程序窗口的后续绘制操作，同时也会向在该图表上运行的 EA 交易和指标发送内部消息。在这种情况下，可能会出现以下情况：应用程序主线程尚未完成 WM_PAINT 窗口重绘系统消息的处理（因此尚未改变图表的外观），而 EA 交易或指标已接收到鼠标光标移动事件。此时，图表特性 *CHART_FIRST_VISIBLE_BAR* 仅在图表绘制完成后发生更改。

由于在两类交互式 MQL 程序中，我们目前仅学习了指标，因此本章所有示例均基于指标构建。另一类程序 EA 交易将在本书下一章详细介绍。但其事件处理原理与本章所述完全一致。

5.9.1 事件处理函数 OnChartEvent

如果代码中包含具有以下原型的 *OnChartEvent* 函数，指标或 EA 交易可以从终端接收交互事件。

void OnChartEvent(const int event, const long &lparam, const double &dparam, const string &sparam)

当用户执行操作时，或在通过 *EventChartCustom* 生成“用户事件”的情况下，终端会调用该函数。

在 *event* 参数中，事件标识符（即事件类型）将以 *ENUM_CHART_EVENT* 枚举值的形式传递（见下表）。

标识符	说明
CHARTEVENT_KEYDOWN	键盘操作
CHARTEVENT_MOUSE_MOVE	移动鼠标和点击鼠标按钮（如果为图表设置了 <i>CHART_EVENT_MOUSE_MOVE</i> 特性）
CHARTEVENT_MOUSE_WHEEL	点击或滚动鼠标滚轮（如果为图表设置了 <i>CHART_EVENT_MOUSE_WHEEL</i> 特性）
CHARTEVENT_CLICK	鼠标点击图表
CHARTEVENT_OBJECT_CREATE	创建图形对象（如果为图表设置了 <i>CHART_EVENT_OBJECT_CREATE</i> 特性）
CHARTEVENT_OBJECT_CHANGE	通过特性对话框修改图形对象

标识符	说明
CHARTEVENT_OBJECT_DELETE	删除图形对象（如果为图表设置了 CHART_EVENT_OBJECT_DELETE 特性）
CHARTEVENT_OBJECT_CLICK	鼠标点击图形对象
CHARTEVENT_OBJECT_DRAG	拖动图形对象
CHARTEVENT_OBJECT_ENDEDIT	完成“输入字段”图形对象中的文本编辑
CHARTEVENT_CHART_CHANGE	更改图表尺寸或特性（通过特性对话框、工具栏或上下文菜单）
CHARTEVENT_CUSTOM	自定义事件范围的起始编号
CHARTEVENT_CUSTOM_LAST	自定义事件范围的结束编号

lparam、*dparam* 和 *sparam* 参数的用途根据事件类型的不同而不同。通常，这些参数包含处理特定事件所需的其他数据。后续章节将针对每种类型提供详细说明。

注意！*OnChartEvent*函数仅对直接加载在图表上的指标和 EA 交易生效。如果使用 *iCustom* 或 *IndicatorCreate* 以编程方式创建任何指标，则 *OnChartEvent*事件不会传递给该指标。

此外，*OnChartEvent*处理程序在 测试程序中不会被调用，即使在可视化模式下。

首次演示 *OnChartEvent*处理程序时，我们将使用一个无缓冲区指标 *EventAll.mq5*，它可以拦截并记录所有事件。

```
void OnChartEvent(const int id,
    const long &lparam, const double &dparam, const string &sparam)
{
    ENUM_CHART_EVENT evt = (ENUM_CHART_EVENT)id;
    PrintFormat("%s %lld %f '%s'", EnumToString(evt), lparam, dparam, sparam);
}
```

默认情况下，图表上可以生成所有类型的事件，但有四种批量事件除外，如上表所示，这些事件需要通过图表的特殊特性启用。在下一节中，我们将为该指标添加设置，以便根据需要包含特定类型的事件。

在包含现有对象的图表上运行该指标，或在该指标运行时创建新对象。

更改图表的尺寸或设置，进行鼠标点击操作，并编辑对象的特性。日志中将出现以下条目。

```
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_CLICK 149 144.000000 ''
CHARTEVENT_OBJECT_CLICK 112 105.000000 'Daily Rectangle 53404'
CHARTEVENT_CLICK 112 105.000000 ''
CHARTEVENT_KEYDOWN 46 1.000000 '339'
CHARTEVENT_CLICK 13 252.000000 ''
CHARTEVENT_OBJECT_DRAG 0 0.000000 'Daily Button 61349'
CHARTEVENT_OBJECT_CLICK 145 104.000000 'Daily Button 61349'
CHARTEVENT_CLICK 145 104.000000 ''
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_OBJECT_DRAG 0 0.000000 'Daily Vertical Line 22641'
CHARTEVENT_OBJECT_DRAG 0 0.000000 'Daily Vertical Line 22641'
CHARTEVENT_OBJECT_CLICK 177 206.000000 'Daily Vertical Line 22641'
CHARTEVENT_CLICK 177 206.000000 ''
CHARTEVENT_OBJECT_CHANGE 0 0.000000 'Daily Rectangle 37930'
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_CLICK 152 118.000000 ''
```

在此处我们将看到各种类型的事件，阅读后续章节后便可以知晓其参数的含义。

5.9.2 事件相关图表特性

有四种类型的事件能够生成大量消息，因此默认处于禁用状态。要稍后启用或禁用这些事件，使用 *ChartSetInteger* 函数设置相应的图表特性。所有特性均为 Boolean 类型：*true*表示启用，*false*表示禁用。

标识符	说明
CHART_EVENT_MOUSE_WHEEL	向图表发送鼠标滚轮事件的 CHARTEVENT_MOUSE_WHEEL 消息
CHART_EVENT_MOUSE_MOVE	向图表发送鼠标移动的 CHARTEVENT_MOUSE_MOVE 消息
CHART_EVENT_OBJECT_CREATE	向图表发送图形对象创建的 CHARTEVENT_OBJECT_CREATE 消息
CHART_EVENT_OBJECT_DELETE	向图表发送图形对象删除的 CHARTEVENT_OBJECT_DELETE 消息

如果任何 MQL 程序修改了这些特性中的一项，它将影响同一图表上运行的所有其他程序，并且即使原始程序终止后，该设置仍然保持有效。

默认情况下，所有特性的值均为 *false*。

让我们为上一节的 *EventAll.mq5*指标补充四个输入变量，以便可以启用这些类型的事件（除了其他无法禁用的事件之外）。此外，我们将描述四个辅助变量，以便能够在删除指标后恢复图表设置。

```

input bool ShowMouseMove = false;
input bool ShowMouseWheel = false;
input bool ShowObjectCreate = false;
input bool ShowObjectDelete = false;

bool mouseMove, mouseWheel, objectCreate, objectDelete;

```

在启动时，记住这些特性的当前值，然后应用用户选择的设置。

```

void OnInit()
{
    mouseMove = PRTF(CharGetInteger(0, CHART_EVENT_MOUSE_MOVE));
    mouseWheel = PRTF(CharGetInteger(0, CHART_EVENT_MOUSE_WHEEL));
    objectCreate = PRTF(CharGetInteger(0, CHART_EVENT_OBJECT_CREATE));
    objectDelete = PRTF(CharGetInteger(0, CHART_EVENT_OBJECT_DELETE));

    ChartSetInteger(0, CHART_EVENT_MOUSE_MOVE, ShowMouseMove);
    ChartSetInteger(0, CHART_EVENT_MOUSE_WHEEL, ShowMouseWheel);
    ChartSetInteger(0, CHART_EVENT_OBJECT_CREATE, ShowObjectCreate);
    ChartSetInteger(0, CHART_EVENT_OBJECT_DELETE, ShowObjectDelete);
}

```

特性在 *OnDeinit* 处理程序中恢复。

```

void OnDeinit(const int)
{
    ChartSetInteger(0, CHART_EVENT_MOUSE_MOVE, mouseMove);
    ChartSetInteger(0, CHART_EVENT_MOUSE_WHEEL, mouseWheel);
    ChartSetInteger(0, CHART_EVENT_OBJECT_CREATE, objectCreate);
    ChartSetInteger(0, CHART_EVENT_OBJECT_DELETE, objectDelete);
}

```

在启用新事件类型的情况下运行此指标。请做好接收大量鼠标移动消息的准备。以下是日志片段：

```

CHARTEVENT_MOUSE_WHEEL 5308557 -120.000000 ''
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_MOUSE_WHEEL 5308557 -120.000000 ''
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_MOUSE_MOVE 141 81.000000 '2'
CHARTEVENT_MOUSE_MOVE 141 81.000000 '0'
...
CHARTEVENT_OBJECT_CREATE 0 0.000000 'Daily Rectangle 37664'
CHARTEVENT_MOUSE_MOVE 323 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 322 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 321 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 320 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 318 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 316 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 314 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 314 145.000000 '0'
...
CHARTEVENT_OBJECT_DELETE 0 0.000000 'Daily Rectangle 37664'

```

```
CHARTEVENT_KEYDOWN 46 1.000000 '339
```

我们将在下文的相关章节中详细介绍每类事件的具体信息内容。

5.9.3 图表更改事件

当更改图表尺寸、价格显示模式、比例或其他参数时，终端将发送无参数的 CHARTEVENT_CHART_CHANGE 事件。MQL 程序必须通过调用 **ChartGet** 函数自行获取变更信息。

我们在 [图表显示模式](#) 章节中的 *ChartModeMonitor.mq5* 示例中已经使用该事件。现在让我们看看另一个示例。

如你所知，MetaTrader 5 允许将当前图表的截屏保存为指定尺寸的文件（通过上下文菜单中的 **Save as Picture** 命令）。但这种获取截屏的方法并非适用于所有情况。具体来说，当需要包含工具提示的图像，或当输入字段类型的对象处于激活状态时（例如字段内文本被选中且文本光标可见），标准命令将无法满足需求，因为它会重新生成图表图像，而不会考虑窗口当前状态的细节特征。

获取窗口精确副本的唯一替代方法是使用终端外部的工具（例如通过 Windows 剪贴板的 **PrtSc** 键），但该方法无法保证所需的窗口尺寸。为了避免通过反复尝试或借助其他程序来选择尺寸，我们将创建一个名为 *EventWindowSizer.mq5* 的指标，它会实时追踪用户设置的尺寸，并在注释中显示当前值。

所有操作均在 **OnChartEvent** 处理程序中完成，首先检查事件 ID 是否为 CHARTEVENT_CHART_CHANGE。窗口尺寸可通过 **CHART_WIDTH_IN_PIXELS** 和 **CHART_HEIGHT_IN_PIXELS** 特性获取，以像素为单位。然而，它们返回的尺寸并未考虑边框，而截屏时通常需要包含边框。因此，我们将在注释中不仅显示特性值（标记为 "Screen"），还会显示修正后的值（标记为 "Picture"）：宽度需增加 2 像素，垂直方向增加 1 像素（这是终端窗口渲染的特性）。

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &str)
{
    if(id == CHARTEVENT_CHART_CHANGE)
    {
        const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
        const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);
        // "Raw" sizes "as is" are displayed with the "Screen" mark,
        // correction for (-2,-1) is needed to include frames - it is displayed with th
        // correction for (-54,-22) is needed to include scales - it is displayed with
        Comment(StringFormat("Screen: %d x %d\nPicture: %d x %d\nIncluding scales: %d x %d",
            w, h, w + 2, h + 1, w + 2 + 54, h + 1 + 22));
    }
}
```

此外，所得到的数值也未考虑时间和价格刻度。如果截屏尺寸还需要包含这些元素，则还需对它们的尺寸进行调整。遗憾的是，MQL5 API 并未提供获取这些尺寸的方法，因此我们只能通过经验确定：在标准 Windows 字体设置下，价格轴宽度为 54 像素，时间轴高度为 22 像素。这些常量可能因 Windows 版本而异，你需要编辑它们，或通过输入参数进行设置。

在图表上运行该指标后，尝试调整窗口大小，观察注释中的数值如何变化。



窗口 s

窗口截屏 (注释中显示工具提示和当前大小)

5.9.4 键盘事件

MQL 程序可通过在 `OnChartEvent` 函数中处理 `CHARTEVENT_KEYDOWN` 事件来接收来自终端的按键消息。

需要注意的是, 仅当图表处于活动状态且获得输入焦点时, 才会生成此类事件。

在 Windows 中, 焦点指的是当前与用户交互的特定窗口在逻辑和视觉上的选中状态。通常, 可通过鼠标点击或特殊键盘快捷键 (例如 `Tab`、`Ctrl+Tab`) 切换焦点, 这会使被选中的窗口高亮显示。例如, 输入字段将会出现文本光标, 列表中的当前行会以另一种颜色标记等等。

在终端中也能观察到类似的视觉效果, 尤其是当 **Market Watch** 窗口、**Data Window** 窗口或专家日志获得焦点时。但图表窗口的情况略有不同。从外观上往往难以判断显示前台显示的图表是否已获得输入焦点。如前所述, 可以通过以下方式确保切换焦点: 点击所需图表 (图表区域, 而非窗口标题或边框) 或使用快捷键:

- `Alt+W`: 打开图表列表窗口, 从中选择图表。
- `Ctrl+F6`: 切换至下一个图表 (按窗口列表顺序, 通常对应选项卡顺序)。
- `Ctrl+Shift+F6`: 切换至上一个图表。

MetaTrader 5 的完整快捷键列表可在 [文档](#) 中查阅。请注意, 部分快捷键组合不符合 Microsoft 的常规建议 (例如 `F10` 用于打开报价窗口, 而非激活主菜单)。

`CHARTEVENT_KEYDOWN` 事件参数包含以下信息:

- `lparam` - 按下按键的代码
- `dparam` - 按住按键期间生成的击键次数

• *sparam* –描述键盘按键状态的位掩码 (已转换为字符串)

位	说明
0-7	按键扫描码 (取决于硬件和 OEM)
8	扩展键盘键特性
9-12	供 Windows 系统服务使用 (请勿使用)
13	Alt 键状态 (1-按下, 0-释放), 不可用 (见下文)
14	按键之前的状态 (1-按下, 0-释放)
15	按键状态变化 (1表示释放, 0表示按下)

Alt键状态不可用, 因为会被终端拦截, 因此, 此位始终为 0。由于该事件的触发机制, 第 15 位始终为 0: 仅按键按下事件会传递给 MQL 程序, 按键释放事件不会传递。

扩展键盘的特性 (第 8 位) 将进行设置, 例如数字小键盘按键 (笔记本电脑通常通过 **Fn** 键激活), 如 **NumLock**、**ScrollLock**、右侧 **Ctrl** (区别于左侧主 **Ctrl**)。更多相关内容, 请参阅 Windows 文档。

首次按下任何非系统按键时, 第 14 位为 0。如果持续按住按键, 在后续自动生成的重复事件中, 该位将为 1。

以下结构有助于确保位说明的准确性。

```
struct KeyState
{
    uchar scancode;
    bool extended;
    bool altPressed;
    bool previousState;
    bool transitionState;

    KeyState() { }
    KeyState(const ushort keymask)
    {
        this = keymask; // use operator overload=
    }
    void operator=(const ushort keymask)
    {
        scancode = (uchar)(0xFF & keymask);
        extended = 0x100 & keymask;
        altPressed = 0x2000 & keymask;
        previousState = 0x4000 & keymask;
        transitionState = 0x8000 & keymask;
    }
};
```

在 MQL 程序中, 可按如下方式使用。

```

void OnChartEvent(const int id,
                  const long &lparam,
                  const double &dparam,
                  const string &sparam)
{
    if(id == CHARTEVENT_KEYDOWN)
    {
        PrintFormat("%lld %lld %4lX", lparam, (ulong)dparam, (ushort)sparam);
        KeyState state[1];
        state[0] = (ushort)sparam;
        ArrayPrint(state);
    }
}

```

从实际应用角度来看,使用宏从按键掩码中提取位特性会更为便捷。

```

#define KEY_SCANCODE(SPAM) ((uchar)(((ushort)SPAM) & 0xFF))
#define KEY_EXTENDED(SPAM) ((bool)(((ushort)SPAM) & 0x100))
#define KEY_PREVIOUS(SPAM) ((bool)(((ushort)SPAM) & 0x4000))

```

可以在图表上运行 *EventAll.mq5* 指标 (该指标位于 [事件相关图表特性](#) 章节), 观察按下特定按键时日志中显示的参数值。

请注意, *lparam* 中的代码为虚拟键盘键码。其列表可在 [MetaTrader 5](#) 安装目录下的 *MQL5/Include/VirtualKeys.mqh* 文件中查看。例如, 部分代码如下:

```

#define VK_SPACE          0x20
#define VK_PRIOR          0x21
#define VK_NEXT          0x22
#define VK_END            0x23
#define VK_HOME          0x24
#define VK_LEFT          0x25
#define VK_UP            0x26
#define VK_RIGHT         0x27
#define VK_DOWN          0x28
...
#define VK_INSERT        0x2D
#define VK_DELETE        0x2E
...
// VK_0 - VK_9 ASCII codes of characters '0' - '9' (0x30 - 0x39)
// VK_A - VK_Z ASCII codes of characters 'A' - 'Z' (0x41 - 0x5A)

```

这些代码被称为虚拟键码, 原因在于, 相同功能的按键在不同键盘上的位置可能不同, 甚至可能通过组合辅助键实现 (如笔记本电脑的 *Fn* 键)。此外, 虚拟性还体现在另一方面, 同一按键可能生成不同符号或控制动作。例如, 同一按键在不同的语言布局中可能代表不同的字母。此外, 每个字母键根据 *CapsLock* 的模式和 *Shift* 键的状态, 可生成大写或小写字母。

为此, MQL5 API 提供了专门的 *TranslateKey* 函数, 用于将虚拟键码转换为相应字符。

short TranslateKey(int key)

该函数会根据当前输入语言和控制键的状态, 基于传入的虚拟键码返回对应的 Unicode 字符。

如果发生错误, 将返回值 *-1*。如果键码与正确字符不匹配, 例如尝试获取 *Shift* 键的字符时, 可能会发生错误。

需要注意，MQL 程序除了能获取按键码外，还可通过控制键和模式 [检查键盘状态](#)。顺便提一下，作为参数传递给 *TerminalInfoInteger* 函数的 `TERMINAL_KEYSTATE_XXX` 形式的常量，是基于 1000 + 虚拟键码的原则。例如，`TERMINAL_KEYSTATE_UP` 为 1038，因为 `VK_UP` 是 38 (0x26)。

在设计对按键做出反应的算法时，请记住，终端可能会拦截许多组合键，因为它们被保留用于执行特定操作（上文已提供文档链接）。特别要注意，按下空格键会打开一个用于沿时间轴快速导航的字段。MQL5 API 允许你对这种内置键盘处理进行部分控制，并且在必要时可将其禁用。请参阅 [鼠标和键盘控制](#) 章节。

简单的无缓冲指标 *EventTranslateKey.mq5* 可用于演示该函数的功能。在其针对 `CHARTEVENT_KEYDOWN` 事件的 *OnChartEvent* 处理程序中，将调用 *TranslateKey* 获取有效 *Unicode* 字符。如果成功，该符号将被添加到图表注释中显示的消息字符串中。按下 *Enter*，将在文本中插入换行符，按下 *Backspace*，则会删除文本末尾的最后一个字符。

```
#include <VirtualKeys.mqh>

string message = "";

void OnChartEvent(const int id,
    const long &lparam, const double &dparam, const string &sparam)
{
    if(id == CHARTEVENT_KEYDOWN)
    {
        if(lparam == VK_RETURN)
        {
            message += "\n";
        }
        else if(lparam == VK_BACK)
        {
            StringSetLength(message, StringLen(message) - 1);
        }
        else
        {
            ResetLastError();
            const ushort c = TranslateKey((int)lparam);
            if(_LastError == 0)
            {
                message += ShortToString(c);
            }
        }
        Comment(message);
    }
}
```

你可以尝试输入不同大小写形式和不同语言的字符。

注意事项：该函数返回有符号的 *short*，主要是为了能够返回错误码 `-1`。“宽字符”（双字节字符）的类型被视为无符号整数，即 *ushort*。如果将接收变量声明为 *ushort*，使用 `-1` 进行检查（例如 `c != -1`）将引发“符号不匹配”的编译器警告（需要显式类型转换），而另一种检查方式（`c >= 0`）通常是错误的，因为其结果始终等于 *true*。

为了能够在消息中插入单词间的空格，在 *OnInit* 处理程序中预先禁用了由空格键触发的快速导航功能。


```
void OnInit()
{
    ChartSetInteger(0, CHART_QUICK_NAVIGATION, false);
}
```

作为使用键盘事件的一个完整示例，考虑以下应用程序任务。终端用户知道，无需打开设置对话框，使用鼠标即可交互式地更改主图表窗口的刻度，只需在价格刻度上按下鼠标按钮，然后在不松开的情况下向上/向下移动。遗憾的是，此方法不适用于子窗口。

子窗口始终自动缩放以适应所有内容，要更改比例，必须打开对话框并手动输入数值。当子窗口中的指标显示“异常值”时，即过大的单个读数，会干扰对其他正常（中等）尺寸数据的分析，此时就需要进行这种操作。此外，有时为了处理更精细的细节，简单地放大图像即可。

为解决此问题并允许用户通过键盘操作调整子窗口比例，我们实现了 *SubScaler* 指标。该指标没有缓冲区且不显示任何内容。

SubScaler 必须作为子窗口中的首个指标，或更严格地说，必须在你需要控制刻度的工作指标添加到子窗口之前，先将其添加到该子窗口中。要使 *SubScaler* 成为首个指标，应将其放置于图表（主窗口）上，从而创建新的子窗口，之后可在该子窗口中添加从属指标。

在工作指标的设置对话框中，务必启用 *Inherit scale* 选项（位于 *Scale* 选项卡上）。

当两个指标同时在子窗口中运行时，你可以使用 *Up/Down* 箭头键进行放大/缩小操作。如果按住 *Shift* 键，垂直轴上的当前可见数值范围将向上或向下移动。

放大意味着聚焦细节（“镜头拉近”），因此部分数据可能超出窗口显示范围。缩小则表示整体画面变小（“镜头拉远”）。

输入参数设置如下：

- 初始最大值 (Initial maximum) - 初次放置在图表上时数据的上限，默认值为 +1000。
- 初始最小值 (Initial minimum) - 初次放置在图表上时数据的下限，默认值为 -1000。
- 缩放系数 (Scaling factor) - 按键时缩放比例的变化步长，取值范围为 [0.01 ... 0.5]，默认值为 0.1。

由于 *SubScaler* 无法预知后续添加到子窗口中的任意第三方指标的有效数值范围，因此必须由用户设置最小值和最大值。

当启动新的终端会话后恢复图表，或加载 tpl 模板时，*SubScaler* 会自动恢复上次保存的缩放状态。

现在让我们分析 *SubScaler* 的具体实现。

上述设置通过对应的输入变量进行设置：

```
input double FixedMaximum = 1000; // Initial Maximum
input double FixedMinimum = -1000; // Initial Minimum
input double _ScaleFactor = 0.1; // Scale Factor [0.01 ... 0.5]
input bool Disabled = false;
```

此外，*Disabled* 变量允许临时禁用特定指标实例的键盘响应，以便在不同子窗口中逐个设置不同的比例。

由于 MQL5 中的输入变量是只读的，我们不得不额外声明一个变量 *ScaleFactor*，用于将输入值校正到允许范围 [0.01 ... 0.5] 内。

```
double ScaleFactor;
```

当前子窗口编号 (*w*) 和其中的指标数量 (*n*) 存储在全局变量中, 这些变量均在 *OnInit* 处理程序中初始化。

```
int w = -1, n = -1;

void OnInit()
{
    ScaleFactor = _ScaleFactor;
    if(ScaleFactor < 0.01 || ScaleFactor > 0.5)
    {
        PrintFormat("ScaleFactor %f is adjusted to default value 0.1,"
            " valid range is [0.01, 0.5]", ScaleFactor);
        ScaleFactor = 0.1;
    }
    w = ChartWindowFind();
    n = ChartIndicatorsTotal(0, w);
}
```

在 *OnChartEvent* 函数中, 我们处理两种类型的事件: 图表变更事件和键盘事件。

CHARTEVENT_CHART_CHANGE 事件用于追踪子窗口中后续指标 (即需要缩放的工作指标) 的添加情况。同时, 我们会请求子窗口数值的当前范围 (CHART_PRICE_MIN 和 CHART_PRICE_MAX), 并判断该范围是否为无效范围, 即最大值和最小值都等于零的情况。如果出现这种情况, 则需应用输入参数中指定的初始限制 (*FixedMinimum* 和 *FixedMaximum*)。

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &strparam)
{
    switch(id)
    {
        case CHARTEVENT_CHART_CHANGE:
            if(ChartIndicatorsTotal(0, w) > n)
            {
                n = ChartIndicatorsTotal(0, w);
                const double min = ChartGetDouble(0, CHART_PRICE_MIN, w);
                const double max = ChartGetDouble(0, CHART_PRICE_MAX, w);
                PrintFormat("Change: %f %f %d", min, max, n);
                if(min == 0 && max == 0)
                {
                    IndicatorSetDouble(INDICATOR_MINIMUM, FixedMinimum);
                    IndicatorSetDouble(INDICATOR_MAXIMUM, FixedMaximum);
                }
            }
            break;
        ...
    }
}
```

当接收到键盘按下事件时, 将调用主 *Scale* 函数, 该函数不仅接收 *lparam*, 还会通过引用 *TerminalInfoInteger*(*TERMINAL_KEYSTATE_SHIFT*) 获取 *Shift* 键的状态。

```

void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &str)
{
    switch(id)
    {
        case CHARTEVENT_KEYDOWN:
            if(!Disabled)
                Scale(lparam, TerminalInfoInteger(TERMINAL_KEYSTATE_SHIFT));
            break;
        ...
    }
}

```

在 *Scale* 函数内部，首先将当前数值范围存入 *min* 和 *max* 变量中。

```

void Scale(const long cmd, const int shift)
{
    const double min = ChartGetDouble(0, CHART_PRICE_MIN, w);
    const double max = ChartGetDouble(0, CHART_PRICE_MAX, w);
    ...
}

```

然后，根据当前是否按下 *Shift* 键，执行缩放或平移操作，即向上或向下移动可见数值范围。在这两种情况下，修改操作均以给定步长（乘数）*ScaleFactor* 相对于 *min* 和 *max* 限值执行，并分别将结果赋值给指标特性 *INDICATOR_MINIMUM* 和 *INDICATOR_MAXIMUM*。由于从属指标设置了“继承比例（Inherit scale）”选项，因此该设置也会成为其工作设置。

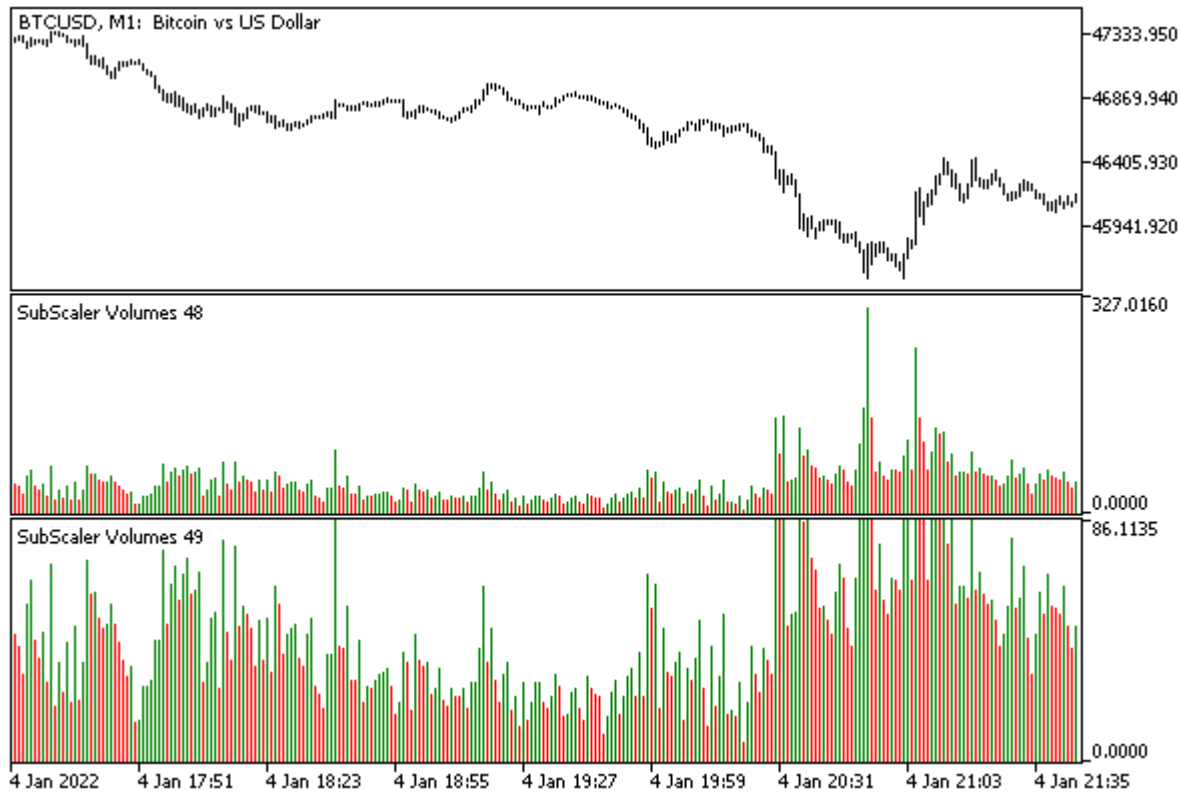
```

if((shift &0x10000000) ==0)// Shift is not pressed - scalechange
{
    if(cmd == VK_UP) // enlarge (zoom in)
    {
        IndicatorSetDouble(INDICATOR_MINIMUM, min / (1.0 + ScaleFactor));
        IndicatorSetDouble(INDICATOR_MAXIMUM, max / (1.0 + ScaleFactor));
        ChartRedraw();
    }
    else if(cmd == VK_DOWN) // shrink (zoom out)
    {
        IndicatorSetDouble(INDICATOR_MINIMUM, min * (1.0 + ScaleFactor));
        IndicatorSetDouble(INDICATOR_MAXIMUM, max * (1.0 + ScaleFactor));
        ChartRedraw();
    }
}
else // Shift pressed - pan/shift range
{
    if(cmd == VK_UP) // shifting charts up
    {
        const double d = (max - min) * ScaleFactor;
        IndicatorSetDouble(INDICATOR_MINIMUM, min - d);
        IndicatorSetDouble(INDICATOR_MAXIMUM, max - d);
        ChartRedraw();
    }
    else if(cmd == VK_DOWN) // shifting charts down
    {
        const double d = (max - min) * ScaleFactor;
        IndicatorSetDouble(INDICATOR_MINIMUM, min + d);
        IndicatorSetDouble(INDICATOR_MAXIMUM, max + d);
        ChartRedraw();
    }
}
}
}

```

每次修改后都会调用 *ChartRedraw* 以更新图表。

让我们看看 *SubScaler* 如何与标准成交量指标配合使用（其他任何指标，包括自定义指标，都以相同方式控制）。



SubScaler 指标在两个子窗口中设置的不同刻度

在此处两个子窗口中，两个 *SubScaler*实例分别为成交量指标应用了不同的垂直刻度。

5.9.5 鼠标事件

我们已经在 [事件相关图表特性](#) 章节中使用 *EventAll.mq5* 指标验证过鼠标事件的接收功能。当在窗口中点击鼠标按钮时，系统会向 MQL 程序发送 CHARTEVENT_CLICK，而 CHARTEVENT_MOUSE_MOVE 光标移动和 CHARTEVENT_MOUSE_WHEEL 滚轮滚动事件则需要事先在图表设置中激活，分别通过 CHART_EVENT_MOUSE_MOVE 和 CHART_EVENT_MOUSE_WHEEL 特性来实现（这两个特性默认均处于禁用状态）。

当鼠标点击位置存在图形对象时，当按下鼠标按钮时，不仅会生成 CHARTEVENT_CLICK 事件，还会生成 [CHARTEVENT_OBJECT_CLICK](#)事件。

对于 CHARTEVENT_CLICK 和 CHARTEVENT_MOUSE_MOVE 事件，*OnChartEvent*处理函数的参数包含以下信息：

- ① pparam -X 坐标
- ② dparam -Y 坐标

此外，对于 CHARTEVENT_MOUSE_MOVE 事件，*sparam*参数包含一个字符串形式的位掩码，用于描述鼠标按键和控制键 (*Ctrl*、*Shift*)的状态。将特定位设置为 1 表示对应的鼠标按键或控制键处于按下状态。

位	说明
0	鼠标左键状态

位	说明
1	鼠标右键状态
2	SHIFT 键状态
3	CTRL 键状态
4	鼠标中键状态
5	第一个附加鼠标按钮状态
6	第二个附加鼠标按钮状态

例如，如果设置了第 0 位，对应的数值为 1 (即 $1 \ll 0$)，如果设置了第 4 位，对应的数值为 16 (即 $1 \ll 4$)。当同时按下多个按钮或按键时，状态将通过位叠加来表示。

对于 CHARTEVENT_MOUSE_WHEEL 事件，X 和 Y 坐标以及鼠标按键和控制键的状态标志以特殊方式编码在 *lparam* 参数中，而 *dparam* 参数则报告滚轮滚动的方向 (正/负) 和滚动量 (±20 的倍数)。

8 字节整数 *lparam* 合并了多个上述信息字段。

字节	说明
0	X 坐标的 <i>short</i> 类型值
1	
2	Y 坐标的 <i>short</i> 类型值
3	
4	按钮和按键状态的位掩码
5	未使用
6	
7	

无论事件类型如何，鼠标坐标都是相对于整个窗口 (包括子窗口) 传输的，因此如有必要，应针对特定子窗口重新计算坐标。

为了更好地理解 CHARTEVENT_MOUSE_WHEEL，可使用指标 *EventMouseWheel.mq5*。该指标会接收并解码消息，然后将消息描述输出到日志中。

```

#define KEY_FLAG_NUMBER 7

const string keyNameByBit[KEY_FLAG_NUMBER] =
{
    "[Left Mouse] ",
    "[Right Mouse] ",
    "(Shift) ",
    "(Ctrl) ",
    "[Middle Mouse] ",
    "[Ext1 Mouse] ",
    "[Ext2 Mouse] ",
};

void OnChartEvent(const int id,
    const long &lparam, const double &dparam, const string &sparam)
{
    if(id == CHARTEVENT_MOUSE_WHEEL)
    {
        const int keymask = (int)(lparam >> 32);
        const short x = (short)lparam;
        const short y = (short)(lparam >> 16);
        const short delta = (short)dparam;
        string message = "";

        for(int i = 0; i < KEY_FLAG_NUMBER; ++i)
        {
            if(((1 << i) & keymask) != 0)
            {
                message += keyNameByBit[i];
            }
        }

        PrintFormat("X=%d Y=%d D=%d %s", x, y, delta, message);
    }
}

```

在图表上运行该指标，然后依次按住不同的鼠标按键和按键并滚动鼠标滚轮。以下是示例结果：

```

X=186 Y=303 D=-120
X=186 Y=312 D=120
X=230 Y=135 D=-120
X=230 Y=135 D=-120 (Ctrl)
X=230 Y=135 D=-120 (Shift) (Ctrl)
X=230 Y=135 D=-120 (Shift)
X=230 Y=135 D=120
X=230 Y=135 D=-120 [Middle Mouse]
X=230 Y=135 D=120 [Middle Mouse]
X=236 Y=210 D=-240
X=236 Y=210 D=-360

```

5.9.6 图形对象事件

对于图表上的 **图形对象**，终端会生成多种专门的事件。其中大部分事件适用于任何类型的对象。输入字段中的文本编辑结束事件 `CHARTEVENT_OBJECT_ENDEDIT` 仅针对 `OBJ_EDIT` 类型的对象生成。

对象点击事件 (`CHARTEVENT_OBJECT_CLICK`)、鼠标拖动事件 (`CHARTEVENT_OBJECT_DRAG`) 和对象特性更改事件 (`CHARTEVENT_OBJECT_CHANGE`) 始终处于激活状态，而对象创建事件 (`CHARTEVENT_OBJECT_CREATE`) 和对象删除事件 (`CHARTEVENT_OBJECT_DELETE`) 则需要通过设置图表的相关特性来显式启用：`CHART_EVENT_OBJECT_CREATE` 和 `CHART_EVENT_OBJECT_DELETE`。

当手动重命名对象时（通过特性对话框），终端会依次生成事件序列：`CHARTEVENT_OBJECT_DELETE`、`CHARTEVENT_OBJECT_CREATE`、`CHARTEVENT_OBJECT_CHANGE`。通过编程方式重命名对象时，不会生成这些事件。

所有对象事件都会在 `OnChartEvent` 函数的 `sparam` 参数中携带关联对象的名称。

此外，`CHARTEVENT_OBJECT_CLICK` 事件会传递点击坐标：`X` 存储在 `lparam` 参数中，`Y` 存储在 `dparam` 参数中。坐标在整个图表中通用（包括子窗口）。

点击不同类型的对象时，其工作机制会有所不同。对于部分对象（如椭圆），光标必须位于任意锚点上方。对于其他对象（如三角形、矩形、线条），光标可以位于对象周边区域（不仅限于点）。在所有这些情况下，当鼠标光标悬停在对象的交互式区域时，将显示包含对象名称的工具提示。

与屏幕坐标绑定的对象（可用于构建程序图形界面，例如按钮、输入字段和矩形面板），鼠标点击对象内部任意位置时将生成事件。

如果光标所在处有多个对象，则仅对 **Z 轴优先级** 最高的对象生成事件。如果多个对象优先级相同，则事件将被分配给创建时间较晚的对象（这与其视觉显示行为一致，即后创建的对象会覆盖更早创建的对象）。

指标新版本 `EventAllObjects.mq5` 可用于检测对象事件。我们将使用已熟悉的多对象 **Object Selector** 类创建并配置该指标，然后在 `OnChartEvent` 处理程序中捕捉其特征事件。

```
#include <MQL5Book/ObjectMonitor.mqh>

class ObjectBuilder: public ObjectSelector
{
protected:
    const ENUM_OBJECT type;
    const int window;
public:
    ObjectBuilder(const string _id, const ENUM_OBJECT _type,
        const long _chart = 0, const int _win = 0):
        ObjectSelector(_id, _chart), type(_type), window(_win)
    {
        ObjectCreate(host, id, type, window, 0, 0);
    }
};
```

首先，在 `OnInit` 中创建一个按钮对象和一条垂直线。针对该线条，我们将追踪其移动（拖拽）事件，当用户按下按钮时，我们将创建一个输入字段，并对输入的文本进行校验。


```

const string ObjNamePrefix = "EventShow-";
const string ButtonName = ObjNamePrefix + "Button";
const string EditBoxName = ObjNamePrefix + "EditBox";
const string VLineName = ObjNamePrefix + "VLine";

bool objectCreate, objectDelete;

void OnInit()
{
    // remember the original settings to restore in OnDeinit
    objectCreate = ChartGetInteger(0, CHART_EVENT_OBJECT_CREATE);
    objectDelete = ChartGetInteger(0, CHART_EVENT_OBJECT_DELETE);

    // set new properties
    ChartSetInteger(0, CHART_EVENT_OBJECT_CREATE, true);
    ChartSetInteger(0, CHART_EVENT_OBJECT_DELETE, true);

    ObjectBuilder button(ButtonName, OBJ_BUTTON);
    button.set(OBJPROP_XDISTANCE, 100).set(OBJPROP_YDISTANCE, 100)
        .set(OBJPROP_XSIZE, 200).set(OBJPROP_TEXT, "Click Me");

    ObjectBuilder line(VLineName, OBJ_VLINE);
    line.set(OBJPROP_TIME, iTime(NULL, 0, 0))
        .set(OBJPROP_SELECTABLE, true).set(OBJPROP_SELECTED, true)
        .set(OBJPROP_TEXT, "Drag Me").set(OBJPROP_TOOLTIP, "Drag Me");

    ChartRedraw();
}

```

在此过程中，不要忘记将图表特性 `CHART_EVENT_OBJECT_CREATE` 和 `CHART_EVENT_OBJECT_DELETE` 设置为 `true`，以便在对象集合发生更改时获得通知。

在 `OnChartEvent` 函数中，我们将对必要事件提供额外响应：拖拽操作完成后，我们将在日志中显示线条的新位置，并在输入字段的文本编辑完成后，显示其内容。

```

void OnChartEvent(const int id,
    const long &lparam, const double &dparam, const string &sparam)
{
    ENUM_CHART_EVENT evt = (ENUM_CHART_EVENT)id;
    PrintFormat("%s %lld %f '%s'", EnumToString(evt), lparam, dparam, sparam);
    if(id == CHARTEVENT_OBJECT_CLICK && sparam == ButtonName)
    {
        if(ObjectGetInteger(0, ButtonName, OBJPROP_STATE))
        {
            ObjectBuilder edit(EditBoxName, OBJ_EDIT);
            edit.set(OBJPROP_XDISTANCE, 100).set(OBJPROP_YDISTANCE, 150)
                .set(OBJPROP_BGCOLOR, clrWhite)
                .set(OBJPROP_XSIZE, 200).set(OBJPROP_TEXT, "Edit Me");
        }
        else
        {
            ObjectDelete(0, EditBoxName);
        }

        ChartRedraw();
    }
    else if(id == CHARTEVENT_OBJECT_ENDEDIT && sparam == EditBoxName)
    {
        Print(ObjectGetString(0, EditBoxName, OBJPROP_TEXT));
    }
    else if(id == CHARTEVENT_OBJECT_DRAG && sparam == VLineName)
    {
        Print(TimeToString((datetime)ObjectGetInteger(0, VLineName, OBJPROP_TIME)));
    }
}

```

请注意，当首次按下按钮时，其状态将从释放转变为按下，我们将针对这一状态变化创建一个输入字段。如果再次点击按钮，其状态会恢复原状，输入字段将从图表中移除。

以下是指标运行时的图表图像。



由 OnChartEvent 事件处理程序控制的对象

指标启动后，日志中会立即显示以下内容：

```
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-Button'
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-VLine'
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
```

如果我们用鼠标拖拽线条，将看到类似如下的信息：

```
CHARTEVENT_OBJECT_DRAG 0 0.000000 'EventShow-VLine'
2022.01.05 10:00
```

接下来，你可以点击按钮，并在新创建的输入字段中编辑文本（编辑完成后按 **Enter** 键或点击输入字段外部）。这将在日志中生成以下条目（坐标和消息文本可能不同，此处输入的文本为 "new message"）：

```
CHARTEVENT_OBJECT_CLICK 181 113.000000 'EventShow-Button'
CHARTEVENT_CLICK 181 113.000000 ''
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-EditBox'
CHARTEVENT_OBJECT_CLICK 152 160.000000 'EventShow-EditBox'
CHARTEVENT_CLICK 152 160.000000 ''
CHARTEVENT_OBJECT_ENDEDIT 0 0.000000 'EventShow-EditBox'
new message
```

如果随后松开按钮，输入字段将被删除。

```
CHARTEVENT_OBJECT_CLICK 162 109.000000 'EventShow-Button'
CHARTEVENT_CLICK 162 109.000000 ''
CHARTEVENT_OBJECT_DELETE 0 0.000000 'EventShow-EditBox'
```

值得注意的是，该按钮默认为双位开关，即每次鼠标点击会使其在按下和释放状态之间交替切换。对于普通按钮，这种行为是多余的：如果仅需追踪按钮的按下操作，应在事件处理过程中通过调用 `ObjectSetInteger(0, ButtonName, OBJPROP_STATE, false)` 将其恢复为释放状态。

5.9.7 生成自定义事件

除标准事件外，终端还支持通过程序生成自定义事件，其本质和内容由 MQL 程序定义。这类事件将被添加到图表事件的通用队列中，并可由所有相关程序在 `OnChartEvent` 函数中处理。

为自定义事件预留了 65536 个整数标识符的特殊范围：CHARTEVENT_CUSTOM 到 CHARTEVENT_CUSTOM_LAST (含首尾值)。换言之，自定义事件的 ID 必须为 CHARTEVENT_CUSTOM + n，其中 n 的取值范围为 0 到 65535。CHARTEVENT_CUSTOM_LAST 正好等于 CHARTEVENT_CUSTOM + 65535。

通过 `EventChartCustom` 函数向图表发送自定义事件。

```
bool EventChartCustom(long chartId, ushort customEventId,
long lparam, double dparam, string sparam)
```

`chartId` 是事件接收图表的标识符，0 表示当前图表，`customEventId` 是事件 ID (由 MQL 程序开发者选择)。该标识符会自动添加到 CHARTEVENT_CUSTOM 值中，并转换为整数类型。该值将作为第一个参数传递给 `OnChartEvent` 处理程序。`EventChartCustom` 的其他参数与 `OnChartEvent` 中的标准事件参数对应，类型为 `long`、`double` 和 `string`，可包含任意信息。

如果用户事件成功加入队列，函数返回 `true`；如果发生错误则返回 `false` (错误代码可通过 `_LastError` 获取)。

在即将进入本书最复杂且关键的交易自动化部分时，我们将着手解决交易机器人开发中的实际应用问题。在展示自定义事件功能的场景下，我们将转向交易环境的多货币 (更广义的说，多交易品种) 分析。

在前面的指标章节中，我们讨论了 [多货币指标](#)，但忽略了一个重要细节：尽管这些指标处理了不同交易品种的报价，但其计算过程仅在 `OnCalculate` 处理程序中启动，而仅当一个交易品种 (即图表的工作交易品种) 的新分时报价到时才会触发这个处理程序。实际上，其他金融工具的分时报价基本被跳过了。例如，如果指标应用于交易品种 A，则每次 A 的分时报价到来时，我们将仅获取其他品种 (B、C、D) 的最新已知分时报价，但很可能在此期间其他交易品种的分时报价而未进行处理。

如果将多货币指标应用于流动性最强的金融工具 (即接收分时报价最频繁的金融工具)，这种情况的影响就不那么显著了。然而，不同金融工具在全天不同时段可能存在速度差异，如果分析或交易算法需要对投资组合中所有金融工具的最新报价做出最快响应，则当前解决方案并不适用。

遗憾的是，在 MQL5 中，新分时报价到达的标准事件仅适用于当前图表的交易品种。在指标中，此时会调用 `OnCalculate` 处理程序，而在 EA 交易中则会调用 `OnTick` 处理程序。

因此，有必要设计一种机制，使 MQL 程序能够接收所有相关金融工具的分时报价通知。这正是自定义事件发挥作用的场景。当然，对于仅分析单一金融工具的程序，这并非必要。

现在，我们将开发一个名为 *EventTickSpy.mq5* 的指标示例，该指标应用于特定交易品种 *X*，能够通过其 *OnCalculate* 函数使用 *EventChartCustom* 发送分时报价通知。因此，在专门为接收此类通知而设计的 *OnChartEvent* 处理程序中，将能够收集来自不同交易品种、不同指标实例的通知。

提供此示例仅用于演示目的。后续在研究多货币自动化交易时，我们将对该技术进行调整，以便在 EA 交易中更便捷地使用。

首先，我们将为该指标设计一个自定义事件编号。由于我们需要针对给定列表中的多个交易品种发送分时报价通知，因此可以在此选择不同的策略。例如，可以选择一个事件标识符，并分别在 *lparam* 和 *sparam* 参数中传递交易品种在列表中的编号和/或交易品种名称。或者，你可以选取某个常量（大于或等于 *CHARTEVENT_CUSTOM*），通过将交易品种编号与该常量叠加，获得事件编号（这样我们就可以自由使用所有参数，特别是 *lparam* 和 *dparam*，它们可用于传递 *Ask*、*Bid* 价格或其他数据）。

我们将重点讨论使用单一事件代码的方案。我们将在 *TICKSPY* 宏中声明这个事件编号。这将作为默认值，用户可在必要时更改，以避免与其他程序发生冲突（尽管这种可能性较低）。

```
#define TICKSPY 0xFEED // 65261
```

该值特意设置为与首个允许的 *CHARTEVENT_CUSTOM* 值有较大间隔。

在指标首次（交互式）启动时，用户必须指定指标应追踪其分时报价的金融工具列表。为此，我们将描述输入字符串变量 *SymbolList*，其中包含以逗号分隔的交易品种列表。

用户事件的标识符在 *message* 参数中设置。

最后，我们需要接收图表的标识符来传递该事件。为此，我们将提供 *Chart* 参数。用户不应编辑该参数：在手动启动指标的首个实例时，通过将指标附加到图表，即可隐式确定图表。在首个实例以编程方式启动的其他指标副本中，该参数将通过调用 *ChartID* 函数（见下文）填充算法。

```
input string SymbolList = "EURUSD,GBPUSD,XAUUSD,USDJPY"; // List of symbols separated
input ushort message = TICKSPY;                          // Custom message
input long chart = 0;                                       // Receiving chart (do not e
```

例如，在 *SymbolList* 参数中，会指定一个包含四个常用交易品种的列表。根据你的 *Market Watch* 需求对其进行编辑。

在 *OnInit* 处理程序中，我们将列表转换为交易品种的 *Symbols* 数组，然后在循环中，对数组中的所有交易品种（当前交易品种除外）运行相同的指标（通常存在这样的匹配情况，因为当前品种已由指标的初始副本处理）。

```

string Symbols[];

void OnInit()
{
    PrintFormat("Starting for chart %lld, msg=0x%X [%s]", Chart, Message, SymbolList);
    if(Chart == 0)
    {
        if(StringLen(SymbolList) > 0)
        {
            const int n = StringSplit(SymbolList, ',', Symbols);
            for(int i = 0; i < n; ++i)
            {
                if(Symbols[i] != _Symbol)
                {
                    ResetLastError();
                    // run the same indicator on another symbol with different settings,
                    // in particular, we pass our ChartID to receive notifications back
                    iCustom(Symbols[i], PERIOD_CURRENT, MQLInfoString(MQL_PROGRAM_NAME),
                        "", Message, ChartID());
                    if(_LastError != 0)
                    {
                        PrintFormat("The symbol '%s' seems incorrect", Symbols[i]);
                    }
                }
            }
        }
        else
        {
            Print("SymbolList is empty: tracking current symbol only!");
            Print("To monitor other symbols, fill in SymbolList, i.e."
                " 'EURUSD,GBPUSD,XAUUSD,USDJPY'");
        }
    }
}

```

在 *OnInit* 起始阶段，指标已启动实例的信息会显示在日志中，便于用户明确了解当前运行状况。

如果我们选择为每个交易品种使用独立事件代码的方案，则必须按如下方式调用 *iCustom*（在 *message* 中添加 *i*）：

```

iCustom(Symbols[i], PERIOD_CURRENT, MQLInfoString(MQL_PROGRAM_NAME), "",
    Message + i, ChartID());

```

请注意，*Chart* 参数的非零值表明该副本通过编程方式启动，且应监控单个交易品种，即图表的工作交易品种。因此，在运行从属副本时，我们无需传递交易品种列表。

在接收新分时报价时调用的 *OnCalculate* 函数中，我们通过调用 *EventChartCustom* 向 *Chart* 图表发送 *Message* 自定义事件。在这种情况下，*lparam* 参数未被使用（等于 0）。在 *dparam* 参数中，我们传递当前（最新）价格 *price[0]*（这是 *Bid* 或 *Last*，具体取决于图表基于何种价格类型：它也是图表处理的最后一个分时报价的价格），并在 *sparam* 参数中传递交易品种名称。

```

int OnCalculate(const int rates_total, const int prev_calculated,
    const int, const double &price[])
{
    if(prev_calculated)
    {
        ArraySetAsSeries(price, true);
        if(Chart > 0)
        {
            // send a tick notification to the parent chart
            EventChartCustom(Chart, Message, 0, price[0], _Symbol);
        }
        else
        {
            OnSymbolTick(_Symbol, price[0]);
        }
    }

    return rates_total;
}

```

在指标的原始实例中（即 *Chart* 参数为 0 的情况），我们直接调用一个特殊函数，即一种多资产分时报价处理程序 *OnSymbolTick*。在这种情况下，无需调用 *EventChartCustom*：尽管此类消息仍会到达图表和该指标副本，但传输过程会消耗数毫秒时间，而且会无谓地增加队列负担。

在该演示中，*OnSymbolTick* 的唯一目的是在日志中输出交易品种名称和最新价格。

```

void OnSymbolTick(const string &symbol, const double price)
{
    Print(symbol, " ", DoubleToString(price,
        (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS)));
}

```

当然，在指标的接收（源）副本中，如果接收到我们发送的消息，同样会从 *OnChartEvent* 处理函数调用该函数。需注意，终端仅在交互式指标副本（即应用于图表的副本）中调用 *OnChartEvent*，而通过 *iCustom* 创建的“不可见”副本中不会显示该事件。

```

void OnChartEvent(const int id,
    const long &lparam, const double &dparam, const string &sparam)
{
    if(id >= CHARTEVENT_CUSTOM + Message)
    {
        OnSymbolTick(sparam, dparam);
        // OR (if using custom event range):
        // OnSymbolTick(Symbols[id - CHARTEVENT_CUSTOM - Message], dparam);
    }
}

```

我们可以避免在事件中发送价格或交易品种名称，因为初始指标（启动该流程的指标）已知晓通用交易品种列表，因此可通过某种方式告知其列表中交易品种的编号。这可以通过在 *lparam* 参数中实现，或如前文所述，通过在用户事件的基准常量中添加一个数字来完成。随后，原始指标在接收事件时，可通过索引从数组中提取对应交易品种，并使用 *SymbolInfoTick* 获取最新分时报价的全部信息，包括不同类型的价格。

我们将在 EURUSD 图表上以默认设置运行该指标，包括 “EURUSD,GBPUSD,XAUUSD,USDJPY” 测试列表。以下是日志内容：

```
16:45:48.745 (EURUSD,H1) Starting for chart 0, msg=0xFEED [EURUSD,GBPUSD,XAUUSD,USDJPY]
16:45:48.761 (GBPUSD,H1) Starting for chart 132358585987782873, msg=0xFEED []
16:45:48.761 (USDJPY,H1) Starting for chart 132358585987782873, msg=0xFEED []
16:45:48.761 (XAUUSD,H1) Starting for chart 132358585987782873, msg=0xFEED []
16:45:48.777 (EURUSD,H1) XAUUSD 1791.00
16:45:49.120 (EURUSD,H1) EURUSD 1.13068 *
16:45:49.135 (EURUSD,H1) USDJPY 115.797
16:45:49.167 (EURUSD,H1) XAUUSD 1790.95
16:45:49.167 (EURUSD,H1) USDJPY 115.796
16:45:49.229 (EURUSD,H1) USDJPY 115.797
16:45:49.229 (EURUSD,H1) XAUUSD 1790.74
16:45:49.369 (EURUSD,H1) XAUUSD 1790.77
16:45:49.572 (EURUSD,H1) GBPUSD 1.35332
16:45:49.572 (EURUSD,H1) XAUUSD 1790.80
16:45:49.791 (EURUSD,H1) XAUUSD 1790.80
16:45:49.791 (EURUSD,H1) USDJPY 115.796
16:45:49.931 (EURUSD,H1) EURUSD 1.13069 *
16:45:49.931 (EURUSD,H1) XAUUSD 1790.86
16:45:49.931 (EURUSD,H1) USDJPY 115.795
16:45:50.056 (EURUSD,H1) USDJPY 115.793
16:45:50.181 (EURUSD,H1) XAUUSD 1790.88
16:45:50.321 (EURUSD,H1) XAUUSD 1790.90
16:45:50.399 (EURUSD,H1) EURUSD 1.13066 *
16:45:50.727 (EURUSD,H1) EURUSD 1.13067 *
16:45:50.773 (EURUSD,H1) GBPUSD 1.35334
```

请注意，在记录来源的（交易品种、时间范围）列中，我们首先看到的是四个请求交易品种上启动的指标实例。

启动后，第一个分时报价是 XAUUSD，而非 EURUSD。随后各交易品种的分时报价以大致相同的频率交替出现。EURUSD 的分时报价标有星号，由此可以直观了解到，如果没有通知，将会遗漏多少其他交易品种的分时报价。

时间戳保存在左侧列中以供参考。

当同一交易品种连续两个事件的报价相同时，通常表明 **Ask** 价发生了变化（此处未显示该价格）。

稍后，在深入研究 MQL5 交易 API 之后，我们将会 EA 交易中运用相同的原理来实现多货币分时报价响应机制。

第六章 交易自动化

在本部分中，我们研究 MQL5 API 中用于交易操作自动化的最复杂且最重要部分。

我们将从介绍实体开始，没有这些实体就无法写出合适的 EA 交易。其中包括 [金融交易工具](#) 和 [交易账户](#) 设置。

然后，我们将研究内置的 [交易函数](#) 和数据结构，以及机器人特有的 [事件](#) 和操作模式。特别是，EA 交易的关键特征 [与测试程序集成在一起](#)，允许用户评估财务表现和优化交易策略。我们将通过 API 考虑内部优化机制和优化管理。

策略测试程序是开发 MQL 程序的基本工具，因为它提供了在各种模式下调试程序的能力，包括柱线和分时报价，基于模拟的或真实的分时报价，有无价格流的可视化。

我们已经尝试了以可视化模式 [测试指标](#)。但是，这些指标的测试参数集有限。在开发 EA 交易时，我们会使用测试程序的全部功能。

此外，我们还会介绍一种新的市场信息：[市场深度](#) 及其软件界面。

 [面向交易员的 MQL5 编程指南](#) - 本书中的源代码。第六章

 书中的示例也可以在 [公共项目](#) \MQL5\Shared Projects\MQL5Book 中找到。

6.1 金融工具和市场报价

MetaTrader 5 允许用户分析和交易金融工具 (又称为交易品种或报价符号)，这是几乎所有终端子系统的基础。图表、指标和报价的历史都与交易品种有关。终端的主要功能建立在金融工具上，如交易订单、交易、保证金要求控制和交易账户历史。

通过终端，经纪商可向交易者提供一个指定的交易品种列表，每个用户从中选择自己喜欢的交易品种，从而形成市场报价。市场报价窗口可确定终端请求在线报价的交易品种，并允许你打开图表和查看历史。

MQL5 API 提供了类似的软件工具，允许你查看和分析所有交易品种特征，将它们添加到市场报价中，或将其从中删除。

除了经纪商提供信息的标准交易品种之外，MetaTrader 5 还可以创建自定义交易品种：它们的特性和价格历史可以从任意数据源加载，并使用公式或 MQL 程序进行计算。自定义交易品种也可参与市场报价，并可用于 [测试策略](#) 和技术分析。但也存在一个天然的局限性：自定义交易品种无法使用常规 MQL5 API 工具进行在线交易，因为自定义交易品种无法存储在服务器上。[自定义交易品种](#) 将在本书的最后第七章的单列一章讨论。

不久前，在相关章节中，我们已经接触过包含个别交易品种价格数据的 [时间序列](#)，这些时间序列中包括历史分页，并以 [指标](#) 为例介绍了历史分页。所有这些功能实际上均假设相应的交易品种已经在市场报价中启用。对于多币种指标和 EA 交易而言尤其如此，它们不仅指图表的工作交易品种，还指其他交易品种。在本章中，我们将学习如何通过 MQL 程序管理市场报价列表。

关于图表的章节已经介绍了某些交易品种特性，这些交易品种特性可通过当前图表的 [基本特性获取函数 \(Point, Digits\)](#) 来获得，因为图表没有关联的交易品种便无法运行。现在我们将研究交易品种的大部分特性，包括其规范。交易品种特性的完整集合可在 [网站上的 MQL5 文档](#) 中找到。

6.1.1 获取可用的交易品种和市场报价列表

MQL5 API 有几个函数可用于进行交易品种操作。使用这些函数，可以找到可用交易品种的总数、*Market Watch* 中选择的交易品种数及其名称。我们知道，终端中可用交易品种的一般列表以分层结构的形式显示在 *Symbols* 对话框中，用户可以使用命令 *View -> Symbols* 或从 *Market Watch* 上下文菜单打开该对话框。该列表包括经纪商提供的交易品种和本地创建的 *自定义交易品种*。可以使用 *SymbolsTotal* 函数来计算交易品种的总数。

int SymbolsTotal(bool selected)

selected 参数指定是仅请求 *Market Watch* 中的交易品种 (*true*)，还是请求所有可用的交易品种 (*false*)。

SymbolName 函数通常与 *SymbolsTotal* 一起使用。该函数通过交易品种的索引返回交易品种的名称 (这里不考虑将交易品种存储分组到逻辑文件夹中，请参见特性 *SYMBOL_PATH*)。

string SymbolName(int index, bool selected)

index 参数用于指定所请求交易品种的索引。*Index* 索引值必须介于 0 到交易品种数之间，取决于第二个参数 *selected* 指定的请求上下文：*true* 将枚举限制为在 *Market Watch* 中选择的交易品种，而 *false* 则绝对匹配所有交易品种 (类似于 *SymbolsTotal*)。因此，在调用 *SymbolName* 时，请将参数 *selected* 设置为与之前用于定义索引范围的 *SymbolsTotal* 调用相同的值。

如果出现错误，特别是如果请求的索引超出列表范围，该函数将返回一个空字符串，错误代码将被写入变量 *_LastError*。

值得注意的是，当启用 *selected* 选项时，函数对 *SymbolsTotal* 和 *SymbolName* 将返回由终端实际更新的交易品种列表信息，即，与服务器执行持续同步的交易品种以及 MQL 程序可获得报价历史的交易品种。该列表可能比 *Market Watch* 中显示的列表大，其中，元素是由用户或通过 MQL 程序显式添加的 (要了解如何添加，请参阅 *Market Watch* 中的 *编辑列表* 一节)。这些交易品种在窗口中是不可见的，当需要它们来计算交叉汇率时，终端会自动连接它们。在交易品种特性中，有两个特性可用于区分显式选择 (*SYMBOL_VISIBLE*) 和隐式选择 (*SYMBOL_SELECT*) 的选项；我们将在 *交易品种状态检查* 一节中讨论。严格地讲，对于 *SymbolsTotal* 和 *SymbolName* 函数，将 *selected* 设置为 *true* 对应启用了 *SYMBOL_SELECT* 的扩展交易品种集，而不仅仅是 *SYMBOL_VISIBLE* 为 *true* 的扩展交易品种集。

Market Watch 交易品种返回的顺序与终端窗口中的顺序一致 (需考虑用户可能进行重新排列，但不考虑按任何列排序 (即使已启用))。不能以编程方式更改 *Market Watch* 中的交易品种顺序。

Symbols 的一般列表中的顺序由终端自己设定 (*Market Watch* 的内容和排序方式不影响该顺序)。

例如，我们看一下简单脚本 *SymbolList.mq5*，用于将可用的交易品种打印到日志中。通过输入参数 *MarketWatchOnly*，用户可将列表限定为 *Market Watch* 交易品种 (如果参数为 *true*)，也可获取完整列表 (*false*)。

```

#property script_show_inputs

#include <MQL5Book/PRTF.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
    const int n = SymbolsTotal(MarketWatchOnly);
    Print("Total symbol count: ", n);
    // write a list of symbols in the Market Watch or all available
    for(int i = 0; i < n; ++i)
    {
        PrintFormat("%4d %s", i, SymbolName(i, MarketWatchOnly));
    }
    // intentionally asking for out-of-range to show an error
    PRTF(SymbolName(n, MarketWatchOnly)); // MARKET_UNKNOWN_SYMBOL(4301)
}

```

以下是一个日志示例。

```

Total symbol count: 10
0 EURUSD
1 XAUUSD
2 BTCUSD
3 GBPUSD
4 USDJPY
5 USDCHF
6 AUDUSD
7 USDCAD
8 NZDUSD
9 USDRUB
SymbolName(n,MarketWatchOnly)= / MARKET_UNKNOWN_SYMBOL(4301)

```

6.1.2 编辑市场报价列表

使用 *SymbolSelect* 函数，MQL 程序开发人员可以在 *Market Watch* 中添加或删除特定交易品种。

boolSymbolSelect(常量字符串名称, boolselect)

参数 *name* 包含受此运算影响的交易品种名称。根据参数 *select* 的值，可以向 *Market Watch* 中添加 (*true*) 或移除一个交易品种。交易品种名称区分大小写：例如，*EURUSD.m* 不等于 *EURUSD.M*。

该函数返回成功 (*true*) 或错误 (*false*) 的指示。错误代码可在 *_LastError* 中找到。

如果某个交易品种有打开的图表或为平仓的头寸，则不能删除该交易品种。此外，不能删除添加到 *Market Watch* 的合成 (自定义) 金融工具的计算公式明确使用的交易品种。

应注意，即使一个交易品种没有打开的图表和未平仓的头寸，也可以被 MQL 程序间接使用：例如，它们可以读取其报价或分时报价的历史。删除此类交易品种可能会导致这些程序出现问题。

以下脚本 *SymbolRemoveUnused.mq5* 能够隐藏所有没有显式使用的交易品种，所以建议在一个演示账户上对其进行检查，或者首先通过上下文菜单保存当前的交易品种集。

```
#include <MQL5Book/MqlError.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
    // request user confirmation for deletion
    if(IDOK == MessageBox("This script will remove all unused symbols"
        " from the Market Watch. Proceed?", "Please, confirm", MB_OKCANCEL))
    {
        const int n = SymbolsTotal(true);
        ResetLastError();
        string removed[];
        // go through the symbols of the Market Watch in reverse order
        for(int i = n - 1; i >= 0; --i)
        {
            const string s = SymbolName(i, true);
            if(SymbolSelect(s, false))
            {
                // remember what was deleted
                PUSH(removed, s);
            }
            else
            {
                // in case of an error, display the reason
                PrintFormat("Can't remove '%s': %s (%d)", s, E2S(_LastError), _LastError)
            }
        }
        const int r = ArraySize(removed);
        PrintFormat("%d out of %d symbols removed", r, n);
        ArrayPrint(removed);
        ...
    }
}
```

在用户确认对交易品种列表的分析后，程序会试图通过调用 *SymbolSelect(s, false)* 来依次隐藏每个交易品种。这种情况仅适用于没有明确使用的金融工具。交易品种的枚举以相反的顺序执行，以避免违反索引。所有成功移除的交易品种都收集在 *removed* 数组中。日志可显示统计数据 and 数组本身。

如果 *Market Watch* 被改变，则用户可以通过在循环调用 *SymbolSelect(removed[i], true)* 来恢复所有被删除的交易品种。

```

if(r > 0)
{
    // it is possible to return the deleted symbols back to the Market Watch
    // (at this point, the window displays a reduced list)
    if(IDOK == MessageBox("Do you want to restore removed symbols"
        " in the Market Watch?", "Please, confirm", MB_OKCANCEL))
    {
        int restored = 0;
        for(int i = r - 1; i >= 0; --i)
        {
            restored += SymbolSelect(removed[i], true);
        }
        PrintFormat("%d symbols restored", restored);
    }
}
}
}

```

下面是日志输出情况。

```

Can't remove 'EURUSD': MARKET_SELECT_ERROR (4305)
Can't remove 'XAUUSD': MARKET_SELECT_ERROR (4305)
Can't remove 'BTCUSD': MARKET_SELECT_ERROR (4305)
Can't remove 'GBPUSD': MARKET_SELECT_ERROR (4305)
...
Can't remove 'USDRUB': MARKET_SELECT_ERROR (4305)
2 out of 10 symbols removed
"NZDUSD" "USDCAD"
2 symbols restored

```

请注意，虽然交易品种是按照它们在 *Market Watch* 中的原始相对顺序恢复的，但是添加的交易品种会出现在列表的末尾，在剩余交易品种之后。因此，所有“被占用”交易品种将位于列表的开头，所有恢复的交易品种将位于其后。*SymbolSelect* 的具体操作如下：交易品种始终添加在列表的末尾，也就是说，不能在特定位置插入交易品种。因此，列表元素的重新排列只能通过手动编辑来进行。

6.1.3 检查交易品种是否存在

MQL 程序可以通过名称来检查特定交易品种的存在，而不用查看整个交易品种列表。为此，提供了一个 *SymbolExist* 函数。

bool SymbolExist (const string name, bool &isCustom)

在 *name* 参数中，应传递相应交易品种的名称。通过引用传递的 *isCustom* 参数将由函数来设置，具体取决于指定的交易品种是标准 (*false*) 还是自定义 (*true*)。

如果在标准或自定义交易品种中均找不到该交易品种，则该函数返回 *false*。

与该函数有一些相似的是 *SYMBOL_EXIST* 特性查询。

我们来分析简单脚本 *SymbolExists.mq5* 来测试该特性。在其参数中，用户可以指定名称，然后将该名称传递给 *SymbolExist*，并记录结果。如果输入空字符串，将检查当前图表的工作交易品种。默认情况下，该参数设置为“XYZ”，这可能与任何可用的交易品种都不匹配。

```
#property script_show_inputs

input string SymbolToCheck = "XYZ";

void OnStart()
{
    const string _SymbolToCheck = SymbolToCheck == "" ? _Symbol : SymbolToCheck;
    bool custom = false;
    PrintFormat("Symbol '%s' is %s", _SymbolToCheck,
        (SymbolExist(_SymbolToCheck, custom) ? (custom ? "custom" : "standard") : "miss
}
```

当该脚本运行两次时（第一次使用默认值，第二次在 EURUSD 图表上使用空行），我们将在日志中获得以下条目。

```
Symbol 'XYZ' is missing
Symbol 'EURUSD' is standard
```

如果已经有自定义交易品种或使用简单的计算公式创建新交易品种，可以确保自定义变量已填充。例如，如果在终端中打开 **Symbols** 窗口并按下 **Create symbol** 按钮，则可以在 **Synthetic tool formula** 字段中输入“SP500/FTSE100”（指数名称可能因你的经纪商而异）并在带有 **Symbol** 名称的字段中输入“GBPUSD.INDEX”。单击 **OK** 将创建一个自定义工具，你可以为其打开一个图表，我们的脚本应在其上显示以下内容：

```
Symbol 'GBPUSD.INDEX' is custom
```

设置你自己的交易品种时，不要忘记不仅要设置公式，还要为点大小和价格变化步长（分时报价）设置足够“小”的值。否则，合成报价序列可能会变成“阶梯状”，甚至退化为一 条直线。

6.1.4 检查交易品种数据相关性

由于采用了分布式客户端/服务器架构，客户端和服务器数据可能偶尔会有所不同。例如，在终端会话开始后，当连接丢失时或者当计算机资源负载过重时，可能会立即数据不同的情况。此外，该交易品种在添加到市场报价后，很可能会在一段时间内保持不同步。MQL5 API 允许你使用函数 **SymbolsSynchronized** 来检查特定交易品种报价数据的相关性。

bool SymbolsSynchronized (const string name)

如果名为 *name* 的交易品种的本地数据与交易服务器上的数据同步，则该函数返回 **true**。

在 **获取价格数组的特征** 一节中，除了其他特性外，还介绍了 **SERIES_SYNCHRONIZED** 的特性，该特性返回一个含义较窄的同步特性：它适用于交易品种和时间范围的特定组合。与此特性相反，函数 **SymbolsSynchronized** 可返回交易品种一般历史的同步属性。

所有时间范围的构建仅在历史下载完成后开始。由于终端中的多线程架构和并行计算，可能会发生 **SymbolsSynchronized** 返回 **true**，并且对于同一交易品种的时间范围，**SERIES_SYNCHRONIZED** 特性将暂时为 **false**。

让我们来看看 **SymbolListSync.mq5** 指标中的新函数是如何工作的。它旨在定期检查 **Market Watch** 的所有交易品种是否同步。检查周期由用户在参数 **SyncCheckupPeriod** 中进行设置，以秒为单位。它可导致 **OnInit** 中的计时器启动。

```

#property indicator_chart_window
#property indicator_plots 0

input int SyncCheckupPeriod = 1; // SyncCheckupPeriod (seconds)

void OnInit()
{
    EventSetTimer(SyncCheckupPeriod);
}

```

在 *OnTimer* 处理程序中，在一个循环内，我们调用 *SymbolsSynchronized* 并将所有未同步交易品种收集到一个公共字符串内，之后它们会显示在注释和日志中。

```

void OnTimer()
{
    string unsynced;
    const int n = SymbolsTotal(true);
    // check all symbols in the Market Watch
    for(int i = 0; i < n; ++i)
    {
        const string s = SymbolName(i, true);
        if(!SymbolIsSynchronized(s))
        {
            unsynced += s + "\n";
        }
    }

    if(StringLen(unsynced) > 0)
    {
        Comment("Unsynced symbols:\n" + unsynced);
        Print("Unsynced symbols:\n" + unsynced);
    }
    else
    {
        Comment("All Market Watch is in sync");
    }
}

```

例如，如果我们将一些先前缺失的交易品种 (Brent) 添加到 *Market Watch* 中，我们会得到如下条目：

```

Unsynced symbols:
Brent

```

正常情况下，大多数时候 (当指标运行时)，日志中应没有此类消息。但是，在通信出现问题时，可能会生成海量警报。

6.1.5 获取交易品种的最后一条分时报价

在时间序列一章的 [使用真实分时报价数组](#) 一节中，我们介绍了内置结构体 *MqlTick*，其包含针对特定交易品种的价格与成交量值字段，这些值在报价每次变动时即可获取。在在线模式下，MQL 程序可以使用采用相同结构体的函数 *SymbolInfoTick* 来查询最后收到的价格和交易量。

bool SymbolInfoTick (const string symbol, MqlTick &tick)

对于具有给定名称 *symbol* 的交易品种，该函数可填充通过引用传递的 *tick* 结构体。如果成功，其返回 *true*。

我们知道，指标和 EA 交易会在新的分时报价到达时由终端自动调用，前提是其包含相应的处理程序 *OnCalculate* 和 *OnTick* 的说明。但是，有关价格变化的意义、最新交易量和分时报价生成时间的信息不会直接传递给处理程序。可以使用 *SymbolInfoTick* 函数获得更详细的信息。

仅针对图表交易品种生成分时报价事件，因此我们已经考虑了为基于 [自定义事件](#) 的分时报价获取多个分时报价事件的选项。在这种情况下，*SymbolInfoTick* 可以读取关于接收通知的第三方交易品种的分时报价信息。

我们获取 *EventTickSpy.mq5* 指标并将其转换为 *SymbolTickSpy.mq5*，后者将请求 *MqlTick* 结构体以获得每个“多货币”分时报价的相应交易品种，然后计算并在图表上显示所有点差。

我们添加一个新的输入参数 *Index*。下面这种新的通知发送方式需要该参数：仅发送用户事件中被改变的交易品种索引（见下文）。

```
#define TICKSPY 0xFEED // 65261

input string SymbolList =
    "EURUSD,GBPUSD,XAUUSD,USDJPY,USDCHF"; // List of symbols, comma separated (example
input ushort Message = TICKSPY;           // Custom message id
input long Chart = 0;                       // Receiving chart id (do not edit)
input int Index = 0;                        // Index in symbol list (do not edit)
```

此外，我们可以添加 *Spreads* 数组来按交易品种存储点差，并添加 *SelfIndex* 变量来记录当前图表交易品种在列表中的位置（如果该交易品种包含在列表中，通常就会记录）。后者用于从指标的原始副本中的 *OnCalculate* 调用我们新的分时报价处理函数。显式获取 *_Symbol* 的现成索引更为简单且准确，无需通过事件将其发回给自身。

```
int Spreads[];
int SelfIndex = -1;
```

引入的数据结构体在 *OnInit* 中初始化。除此之外，*OnInit* 保持不变，包括在第三方交易品种上启动指标的从属实例（此处省略了这些行）。


```

void OnInit()
{
    ...
    const int n = StringSplit(SymbolList, ',', Symbols);
    ArrayResize(Spreads, n);
    for(int i = 0; i < n; ++i)
    {
        if(Symbols[i] != _Symbol)
        {
            ...
        }
        else
        {
            SelfIndex = i;
        }
        Spreads[i] = 0;
    }
    ...
}

```

在 *OnCalculate* 处理程序中，如果指标的副本处理另一个交易品种（同时，需发送通知的 *ChartID* 不等于 0），我们即可在每条分时报价上生成一个自定义事件。请注意，事件中填充的唯一参数为 *lparam*，其等同于 *Index*（*dparam* 为 0，*sparam* 为 NULL）。如果 *Chart* 等于 0，意味着我们在指标的主副本中处理图表交易品种 *_Symbol*，如果在输入交易品种列表中找到该交易品种，我们可使用相应的 *SelfIndex* 索引直接调用 *OnSymbolTick*。

```

int OnCalculate(const int rates_total, const int prev_calculated, const int, const dc
{
    if(prev_calculated)
    {
        if(Chart > 0)
        {
            EventChartCustom(Chart, Message, Index, 0, NULL);
        }
        else if(SelfIndex > -1)
        {
            OnSymbolTick(SelfIndex);
        }
    }

    return rates_total;
}

```

在 *OnChartEvent* 中事件算法的接收部分，我们也调用 *OnSymbolTick*，但是这次我们从 *lparam* 中的列表中获取交易品种编号（从指标的另一个副本中作为参数 *Index* 发送的内容）。

```

void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &text)
{
    if(id == CHARTEVENT_CUSTOM + Message)
    {
        OnSymbolTick((int)lparam);
    }
}

```

函数 *OnSymbolTick* 使用 *SymbolInfoTick* 请求完整的分时报价信息，并按照 *Ask* 和 *Bid* 的价格之差除以点数大小来计算点差（*SYMBOL_POINT* 的特性将在后面讨论）。

```

void OnSymbolTick(const int index)
{
    const string symbol = Symbols[index];

    MqlTick tick;
    if(SymbolInfoTick(symbol, tick))
    {
        Spreads[index] = (int)MathRound((tick.ask - tick.bid)
            / SymbolInfoDouble(symbol, SYMBOL_POINT));
        string message = "";
        for(int i = 0; i < ArraySize(Spreads); ++i)
        {
            message += Symbols[i] + "=" + (string)Spreads[i] + "\n";
        }

        Comment(message);
    }
}

```

新的点差会更新 *Spreads* 数组中相应的单元，之后整个数组会显示在图表的注释中。如下所示。



所请求交易品种列表的当前点差

你可以实时比较注释和 *Market Watch* 窗口中信息是否一致。

6.1.6 交易和报价时段时间表

稍后，在接下来的章节中，我们将讨论 MQL5 API 函数，这些函数将实现交易操作自动化。但首先要研究平台的技术特性，这些特性决定了能否成功调用这些 API。特别是，金融工具的规范施加了一些限制。本章中，我们将逐步全面考虑这些金融工具的编程分析，我们将从交易时段开始。

在交易金融工具时，应考虑到许多国际市场（如证券交易所）都有预先确定的开放时间，只有在这些时间内才能获得信息和进行交易。尽管终端一直连接在经纪商的服务器上，但是在工作时间表之外进行交易的尝试将会失败。在这方面，对于每个交易品种，终端都会存储交易时段时间表，即一天中可以执行某些操作的时间段。

我们知道，有两种主要的交易时段类型：报价和交易。在报价交易时段内，终端会（可能会）接收当前报价。在交易时段内，允许终端发送交易指令和进行交易。在一天中，每种类型可能有几次交易时段，并有休息时间（例如，早上和晚上）。显然，报价交易时段的持续时间大于等于交易时段的持续时间。

在任何情况下，交易时段时间（即开盘和收盘时间）可由终端从交易所的本地时区转换为经纪商所处的时区（服务器时间）。

MQL5 API 允许你使用 *SymbolInfoSessionQuote* 和 *SymbolInfoSessionTrade* 函数来确定每种工具的报价和交易交易时段。特别是，这种重要信息允许程序在向服务器发送交易请求之前，检查市场当前是否开放。因此，我们可以防止必然的错误结果，并避免给服务器增加不必要的负载。请记住，如果由于 MQL 程序的实现方式不当而向服务器发出大量错误请求，服务器可能会开始“忽略”你的终端，并在一段时间内拒绝执行后续命令（即使命令正确）。

```
bool SymbolInfoSessionQuote (const string symbol, ENUM_DAY_OF_WEEK dayOfWeek, uint
sessionIndex, datetime &from, datetime &to)
```

```
bool SymbolInfoSessionTrade (const string symbol, ENUM_DAY_OF_WEEK dayOfWeek, uint
sessionIndex, datetime &from, datetime &to)
```

这些函数可以相同的方式工作。对于给定的 *symbol* 和星期几 *dayOfWeek*，它们会将交易时段为 *sessionIndex* 的开盘和收盘时间，分别填充通过引用传递的参数 *from* 和 *to*。交易时段索引从 0 开始。ENUM_DAY_OF_WEEK 结构体如 [枚举](#)一节中所述。

未提供用于查询交易时段数的单独函数：要查询的话，我们可使用递增的索引 *sessionIndex* 来调用 *SymbolInfoSessionQuote* 和 *SymbolInfoSessionTrade*，直到函数返回一个错误标志 (*false*)。若存在具有指定数量的交易时段，并且输出自变量 *from* 和 *to* 接收到正确的值，这些函数会返回一个成功指示符 (*true*)。

根据 MQL5 文档，在收到的 *datetime* 类型的 *from* 和 *to* 的值中，应忽略日期，仅考虑时间。这是因为该信息为当天的时间表。但是，这个规则有一个重要的例外。

由于市场可能一天 24 小时开放（如在外汇交易）或交易所在地球另一端，其白天的营业时间与你的经纪商“时区”的日期变化重叠，所以交易时段结束时间可能大于等于 24 小时。例如，如果外汇交易时段的开始时间为 00:00，则结束时间为 24:00。但是，从 *datetime* 类型的角度来看，24 小时为第二天的 00 小时 00 分钟。

如果对于某些交易所，其时间表相对于你的经纪商所在时区有几个小时的偏移，导致交易时段会在头一天开始，而在第二天结束，那么情况会更加复杂。因此，*to* 变量不仅记录时间，还记录不可忽略的“额外天数”，否则日间时间 *from* 将大于日间时间 *to* 例如，一个交易时段可能会从今天 21:00 持续到明天 8:00，即 $21 > 8$ 。在这种情况下，检查当前时间是否在交易时段内（时间 x 大于开始时间且小于结束时间）会得出错误结论（例如，若 $x = 23$ ， $x \geq 21 \ \&\& \ x < 8$ 条件不成立，但实际上该时段仍处于有效的交易状态）。

因此，我们得出的结论是，不能忽略参数 *from/to* 中的日期，算法中必须考虑这一点（参见示例）。

为了演示这些函数的功能，让我们回到脚本 *EnvPermissions.mq5* 中的示例，该示例出现在 [权限](#)一节中。其中一种权限（也可以叫限制）特指交易的可用性。之前，该脚本考虑了终端设置 (TERMINAL_TRADE_ALLOWED) 和特定 MQL 程序的设置 (MQL_TRADE_ALLOWED)。现在，我们可以添加交易时段检查来确定在给定时刻对特定交易品种有效的交易许可。

该脚本的新版本名为 *SymbolPermissions.mq5*。这一版本也不是最终版本：在后面的章节中，我们将研究 [交易账户](#) 设置施加的限制。

注意，该脚本实现了 *Permissions* 类，提供了适用于 MQL 程序的所有类型权限/限制的集中说明。除此之外，该类还具有检查交易可用性的方法：*isTradeEnabled* 和 *isTradeOnSymbolEnabled*。前者与全局权限相关，并且几乎保持不变：

```

class Permissions
{
public:
    static bool isTradeEnabled(const string symbol = NULL, const datetime now = 0)
    {
        return TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)
            && MQLInfoInteger(MQL_TRADE_ALLOWED)
            && isTradeOnSymbolEnabled(symbol == NULL ? _Symbol : symbol, now);
    }
    ...
}

```

在检查了终端和 MQL 程序的特性之后, 该脚本会继续执行 *isTradeOnSymbolEnabled*, 对交易品种规范进行分析。以前这种方法几乎是空白。

除了在 *symbol* 参数中传递的工作交易品种之外, *isTradeOnSymbolEnabled* 函数还接收当前时间 (*now*) 和所需的交易模式 (*mode*)。我们将在后面的章节中更详细地讨论后者 (参见 [交易权限](#))。现在, 我们只需要注意 *SYMBOL_TRADE_MODE_FULL* 的默认值给出了最大的自由度 (允许所有交易操作)。

```

static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
    const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
    // checking sessions
    bool found = now == 0;
    if(!found)
    {
        const static ulong day = 60 * 60 * 24;
        const ulong time = (ulong)now % day;
        datetime from, to;
        int i = 0;

        ENUM_DAY_OF_WEEK d = TimeDayOfWeek(now);

        while(!found && SymbolInfoSessionTrade(symbol, d, i++, from, to))
        {
            found = time >= (ulong)from && time < (ulong)to;
        }
    }
    // checking the trading mode for the symbol
    return found && (SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode);
}

```

如果没有指定 *now* 时间 (默认等于 0), 我们认为不必考虑交易时段。这意味着带有已经找到合适交易时段 (即, 包含给定时间的交易时段) 指示的 *found* 变量被立即设置为 *true*。但是, 如果指定了 *now* 参数, 该函数将进入交易时段分析块。

为了从 *datetime* 类型的值中提取日期 (不考虑时间), 我们将 *day* 常量解释为等于一天中的秒数。类似 *now % day* 的表达式将返回完整日期和时间除以一天的持续时间所得的余数, 这将只给出时间 (*datetime* 中最重要的数字将为空)。

TimeDayOfWeek 函数返回给定 *datetime* 值对应的星期几。该函数位于我们之前已经使用过的 *MQL5Book/DateTime.mqh* 头文件中 (参见 [日期和时间](#))。

此外，在 `while` 循环中，我们还会调用 `SymbolInfoSessionTrade` 函数，同时不断递增交易时段索引 `i`，直到找到合适的交易时段或函数返回 `false`（不再有交易时段）。因此，该程序可以获得一周内每一天的完整交易时段列表，类似于在终端的交易品种 `Specifications` 窗口中显示的内容。

显然，合适的交易时段是包含交易时段开始时间 `from` 和结束时间 `to` 之间的指定 `time` 值的交易时段。此处，我们还考虑了与可能的全天候交易相关的问题：在不丢弃日期 (`from % day` 或 `to % day`) 的情况下，将 `from` 和 `to` 与 `time` 按原样比较。

一旦 `found` 等于 `true`，就会退出循环。否则，当超过允许的交易时段数时，该循环将结束（函数 `SymbolInfoSessionTrade` 将返回 `false`），并且不再寻找合适的交易时段。

如果根据交易时段时间表，现在允许进行交易，则会额外检查该交易品种的交易模式 (`SYMBOL_TRADE_MODE`)。例如，交易品种交易可以完全禁止（“指示性”）或处于“仅平仓”模式。

与文件 `SymbolPermissions.mq5` 中的最终版本相比，上面的代码有一些简化。此外，还实现了一种机制可用于标记导致交易被禁用的限制来源。所有这些来源都在 `TRADE_RESTRICTIONS` 枚举中进行了汇总。

```
enum TRADE_RESTRICTIONS
{
    TERMINAL_RESTRICTION = 1,
    PROGRAM_RESTRICTION = 2,
    SYMBOL_RESTRICTION = 4,
    SESSION_RESTRICTION = 8,
};
```

此时，限制可能来自 4 个实例：终端、程序、交易品种和交易时段时间表。我们稍后将添加更多选项。

为了记录在 `Permissions` 类中发现约束这一事实，我们使用了 `lastFailReasonBitMask` 变量，该变量允许使用辅助方法 `pass` 从枚举的元素中收集位掩码（当检查条件 `value` 为 `false` 且该位等于 `false` 时，设置该位）。

```
static uint lastFailReasonBitMask;
static bool pass(const bool value, const uint bitflag)
{
    if(!value) lastFailReasonBitMask |= bitflag;
    return value;
}
```

在适当的验证步骤中执行调用带有特定标志的 `pass` 方法。例如，完整的 `isTradeEnabled` 方法如下所示：

```
static bool isTradeEnabled(const string symbol = NULL, const datetime now = 0)
{
    lastFailReasonBitMask = 0;
    return pass(TerminalInfoInteger(TERMINAL_TRADE_ALLOWED), TERMINAL_RESTRICTION)
        && pass(MQLInfoInteger(MQL_TRADE_ALLOWED), PROGRAM_RESTRICTION)
        && isTradeOnSymbolEnabled(symbol == NULL ? _Symbol : symbol, now);
}
```

因此，当调用 `TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)` 或 `MQLInfoInteger(MQL_TRADE_ALLOWED)` 的结果为否时，将分别设置 `TERMINAL_RESTRICTION` 或 `PROGRAM_RESTRICTION` 标志。

当检测到问题时，*isTradeOnSymbolEnabled* 方法还可以设置自己的标志，包括交易时段标志。

```
static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
    const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
    ...
    return pass(found, SESSION_RESTRICTION)
        && pass(SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode, SYMBOL_RESTRICTION);
}
```

因此，使用 *Permissions::isTradeEnabled* 查询的 MQL 程序在收到限制后，可以使用 *getFailReasonBitMask* 和 *explainBitMask* 方法来阐明其含义：前者“按原样”返回设置禁止标志的掩码，后者形成有关限制的文本说明（用户易于理解）。

```
static uint getFailReasonBitMask()
{
    return lastFailReasonBitMask;
}

static string explainBitMask()
{
    string result = "";
    for(int i = 0; i < 4; ++i)
    {
        if(((1 << i) & lastFailReasonBitMask) != 0)
        {
            result += EnumToString((TRADE_RESTRICTIONS)(1 << i));
        }
    }
    return result;
}
```

使用 *OnStart* 处理程序中的上述 *Permissions* 类，可检查 *Market Watch*（当前为 *TimeCurrent*）中所有交易品种的交易可用性。

```

void OnStart()
{
    string disabled = "";

    const int n = SymbolsTotal(true);
    for(int i = 0; i < n; ++i)
    {
        const string s = SymbolName(i, true);
        if(!Permissions::isTradeEnabled(s, TimeCurrent()))
        {
            disabled += s + "=" + Permissions::explainBitMask() + "\n";
        }
    }
    if(disabled != "")
    {
        Print("Trade is disabled for the following symbols and origins:");
        Print(disabled);
    }
}

```

如果某个交易品种被禁止交易，我们会在日志中看到解释。

```

Trade is disabled for following symbols and origins:
USDRUB=SESSION_RESTRICTION
SP500m=SYMBOL_RESTRICTION

```

在这种情况下，市场会对“USDRUB”交易品种关闭，对“SP500m”交易品种禁用交易（更严格地说，后者不对应于 SYMBOL_TRADE_MODE_FULL 模式）。

假设在运行脚本时，在终端中全局启用了算法交易。否则，我们还会在日志中看到 TERMINAL_RESTRICTION 和 PROGRAM_RESTRICTION 被禁止。

6.1.7 交易品种保证金比率

MT5 API 提供的交易品种规范的特性中，有几个与保证金要求相关的特性，这些特性在开立和维持交易头寸时适用，我们将在后面的章节中详细讨论。由于终端提供不同市场和不同类型金融工具的交易，这些要求可能会有很大差异。概括地说，其表现在保证金修正率的应用上，保证金修正率是针对各种交易品种和不同类型的交易运算单独设定的。对于用户，该比率显示在终端的 *Specifications* 窗口中。

正如我们将在下面看到的，乘数（如果应用的话）应与交易品种特性中的保证金值相乘。可以使用 *SymbolInfoMarginRate* 函数以编程方式获得保证金比率。

```

bool SymbolInfoMarginRate (const string symbol, ENUM_ORDER_TYPE orderType, double &initial,
double &maintenance)

```

对于指定的交易品种和订单类型（ENUM_ORDER_TYPE），该函数分别用初始和维护保证金比率填充通过引用传递的 *initial* 和 *maintenance* 参数。由此获得的比率应乘以相应类型的保证金值（在保证金要求一节中介绍了如何请求保证金值），以获取在下订单（如 *orderType*）时应在帐户中保留的金额。

如果执行成功，则该函数返回 *true*。

让我们使用一个简单脚本 *SymbolMarginRate.mq5* 作为示例，该示例根据 *MarketWatchOnly* 参数输出 *Market Watch* 或所有可用交易品种的保证金比率。可以在 *OrderType* 参数中指定操作类型。

```
#include <MQL5Book/MqlError.mqh>

input bool MarketWatchOnly = true;
input ENUM_ORDER_TYPE OrderType = ORDER_TYPE_BUY;

void OnStart()
{
    const int n = SymbolsTotal(MarketWatchOnly);
    PrintFormat("Margin rates per symbol for %s:", EnumToString(OrderType));
    for(int i = 0; i < n; ++i)
    {
        const string s = SymbolName(i, MarketWatchOnly);
        double initial = 1.0, maintenance = 1.0;
        if(!SymbolInfoMarginRate(s, OrderType, initial, maintenance))
        {
            PrintFormat("Error: %s(%d)", E2S(_LastError), _LastError);
        }
        PrintFormat("%4d %s = %f %f", i, s, initial, maintenance);
    }
}
```

以下是其日志。

```
Margin rates per symbol for ORDER_TYPE_BUY:
0 EURUSD = 1.000000 0.000000
1 XAUUSD = 1.000000 0.000000
2 BTCUSD = 0.330000 0.330000
3 USDCHF = 1.000000 0.000000
4 USDJPY = 1.000000 0.000000
5 AUDUSD = 1.000000 0.000000
6 USDRUB = 1.000000 1.000000
```

你可以将收到的值与终端中的交易品种规范进行比较。

6.1.8 用于获取交易品种特性的函数概述

每个交易品种的完整规范可以通过查询其特性来获得：为此，MQL5 API 提供了三个函数，即 *SymbolInfoInteger*、*SymbolInfoDouble* 和 *SymbolInfoString*，每个函数负责一个特定类型的特性。这些特性分别被描述为三种枚举：ENUM_SYMBOL_INFO_INTEGER、ENUM_SYMBOL_INFO_DOUBLE 和 ENUM_SYMBOL_INFO_STRING。我们已知的图表和对象 API中也使用了类似的技术。

交易品种的名称和所请求特性的标识符可传递给任何一个函数。

每个函数都有两种形式：缩写型和完整型。缩写版本直接返回请求的特性，而完整版本则将其写入通过引用传递的输出参数。例如，对于与整数类型兼容的特性，该函数的原型如下：

```
long SymbolInfoInteger (const string symbol, ENUM_SYMBOL_INFO_INTEGER property)
```

```
bool SymbolInfoInteger (const string symbol, ENUM_SYMBOL_INFO_INTEGER property, long &value)
```

第二种形式返回成功 (*true*) 或错误 (*false*) 的布尔指示符。该函数返回 *false* 的最大原因可能是交易品种名称无效 (MARKET_UNKNOWN_SYMBOL, 4301), 或所请求特性的标识符无效 (MARKET_WRONG_PROPERTY, 4303)。详细信息在 *_LastError* 中提供。

如上所述, ENUM_SYMBOL_INFO_INTEGER 枚举中的特性具有各种整数兼容类型: bool、int、long、color、datetime 和特殊枚举 (所有这些内容将在单独的章节中讨论)。bool、int、long、color、datetime 和特殊枚举 (所有这些内容将在单独的章节中讨论)。

对于实数类型的特性, 定义了以下两种形式的 *SymbolInfoDouble* 函数。

```
double SymbolInfoDouble (const string symbol, ENUM_SYMBOL_INFO_DOUBLE property)
```

```
bool SymbolInfoDouble (const string symbol, ENUM_SYMBOL_INFO_DOUBLE property, double &value)
```

最后, 对于字符串特性, 类似的函数如下所示:

```
string SymbolInfoString (const string symbol, ENUM_SYMBOL_INFO_STRING property)
```

```
bool SymbolInfoString (const string symbol, ENUM_SYMBOL_INFO_STRING property, string &value)
```

开发 EA 交易时经常使用的各种类型的特性在本章以下章节的说明中进行了逻辑分组。

基于上述函数, 我们将创建一个通用类 *SymbolMonitor* (文件 *SymbolMonitor.mqh*) 来获取任何交易品种特性。其基于三个枚举的一组重载 *get* 方法。

```
class SymbolMonitor
{
public:
    const string name;
    SymbolMonitor(): name(_Symbol) { }
    SymbolMonitor(const string s): name(s) { }

    long get(const ENUM_SYMBOL_INFO_INTEGER property) const
    {
        return SymbolInfoInteger(name, property);
    }

    double get(const ENUM_SYMBOL_INFO_DOUBLE property) const
    {
        return SymbolInfoDouble(name, property);
    }

    string get(const ENUM_SYMBOL_INFO_STRING property) const
    {
        return SymbolInfoString(name, property);
    }
    ...
}
```

其他三种类似的方法可以消除第一个参数中的枚举类型, 并由编译器根据第二个伪参数选择必要的重载 (此处, 其类型始终匹配结果类型)。我们将在未来的模板类中使用这一技巧。

```

long get(const int property, const long) const
{
    return SymbolInfoInteger(name, (ENUM_SYMBOL_INFO_INTEGER)property);
}

double get(const int property, const double) const
{
    return SymbolInfoDouble(name, (ENUM_SYMBOL_INFO_DOUBLE)property);
}

string get(const int property, const string) const
{
    return SymbolInfoString(name, (ENUM_SYMBOL_INFO_STRING)property);
}
...

```

因此，通过创建具有所需交易品种名称的对象，可以统一查询其任何类型的特性。为了查询和记录相同类型的所有特性，我们可以实现类似这样的特性。

```

// project (draft)
template<typename E,typename R>
void list2log()
{
    E e = (E)0;
    int array[];
    const int n = EnumToArray(e, array, 0, USHORT_MAX);
    for(int i = 0; i < n; ++i)
    {
        e = (E)array[i];
        R r = get(e);
        PrintFormat("% 3d %s=%s", i, EnumToString(e), (string)r);
    }
}

```

但是，由于在 `long` 类型的特性中，其他类型的值实际上是“隐藏的”，应以特定的方式显示（例如，通过调用 `EnumToString` 进行枚举，通过调用 `imeToString` 查询日期和时间等），因此有必要定义另外三个重载方法以返回特性的字符串表示形式。我们称之为 **stringify**。那么在上面的 `list2log` 草案中，可以使用 **stringify** 而不是将值强制转换为 `(string)`，该方法本身会消除一个模板参数。

```

template<typename E>
void list2log()
{
    E e = (E)0;
    int array[];
    const int n = EnumToArray(e, array, 0, USHORT_MAX);
    for(int i = 0; i < n; ++i)
    {
        e = (E)array[i];
        PrintFormat("% 3d %s=%s", i, EnumToString(e), stringify(e));
    }
}

```

对于实数和字符串类型，*stringify* 的实现看起来非常简单。

```

string stringify(const ENUM_SYMBOL_INFO_DOUBLE property, const string format = NUL
{
    if(format == NULL) return (string)SymbolInfoDouble(name, property);
    return StringFormat(format, SymbolInfoDouble(name, property));
}

string stringify(const ENUM_SYMBOL_INFO_STRING property) const
{
    return SymbolInfoString(name, property);
}

```

但对于 `ENUM_SYMBOL_INFO_INTEGER` 来说，就要稍微复杂一些了。当然，若特性是 *long* 或 *int* 类型，将其强制转换为 (*string*) 就足够了。所有其他情况都需要在 *switch* 运算符中单独分析和转换。

```

string stringify(const ENUM_SYMBOL_INFO_INTEGER property) const
{
    const long v = SymbolInfoInteger(name, property);
    switch(property)
    {
        ...
    }

    return (string)v;
}

```

例如，如果一个特性具有布尔类型，那么用字符串 “true” 或 “false” 来表示它是很方便的（因此其在看上去不同于简单的数字 1 和 0）。下一步，为了举例，假设特性中有 `SYMBOL_EXIST`，相当于 *SymbolExist* 函数，即返回一个布尔值，以表明字符是否存在。对于其处理和其他逻辑特性，实现一种辅助方法 *boolean* 是有意义的。

```

static string boolean(const long v)
{
    return v ? "true" : "false";
}

string stringify(const ENUM_SYMBOL_INFO_INTEGER property) const
{
    const long v = SymbolInfoInteger(name, property);
    switch(property)
    {
        case SYMBOL_EXIST:
            return boolean(v);
        ...
    }

    return (string)v;
}

```

对于枚举特性，最合适的解决方案是使用 *EnumToString* 函数的模板方法。

```

template<typename E>
static string enumstr(const long v)
{
    return EnumToString((E)v);
}

```

例如，`SYMBOL_SWAP_ROLLOVER3DAYS` 特性确定在一周中的哪一天对某个交易品种的未结头寸收取三倍掉期费用，该特性的类型为 `ENUM_DAY_OF_WEEK`。因此，为了对其进行处理，我们可以在 `switch` 中编写以下语句：

```

case SYMBOL_SWAP_ROLLOVER3DAYS:
    return enumstr<ENUM_DAY_OF_WEEK>(v);

```

其值是位标志组合的特性呈现了一种特殊情况。特别地，对于每个交易品种，经纪商可为特定类型的订单设置权限，例如市价、限价、止损、止盈等等（我们将分别介绍这些 [权限](#)）。每种类型的订单都由启用了一位的常量表示，因此它们的叠加（由按位 OR 运算符 `|` 组合）存储在 `SYMBOL_ORDER_MODE` 特性中，并且在没有限制的情况下，所有位同时启用。对于这样的特性，我们将在头文件中定义自己的枚举，例如：

```

enum SYMBOL_ORDER
{
    _SYMBOL_ORDER_MARKET = 1,
    _SYMBOL_ORDER_LIMIT = 2,
    _SYMBOL_ORDER_STOP = 4,
    _SYMBOL_ORDER_STOP_LIMIT = 8,
    _SYMBOL_ORDER_SL = 16,
    _SYMBOL_ORDER_TP = 32,
    _SYMBOL_ORDER_CLOSEBY = 64,
};

```

此处，对于每个内置常量，如 `SYMBOL_ORDER_MARKET`，都声明了一个相应的元素，其标识符与常量相同，但前面有一个下划线，以避免命名冲突。

为了以字符串的形式表示这种枚举的标志组合，我们实现了另一种模板方法 *maskstr*。

```

template<typename E>
static string maskstr(const long v)
{
    string text = "";
    for(int i = 0; ; ++i)
    {
        ResetLastError();
        const string s = EnumToString((E)(1 << i));
        if(_LastError != 0)
        {
            break;
        }
        if((v & (1 << i)) != 0)
        {
            text += s + " ";
        }
    }
    return text;
}

```

其含义类似于 *enumstr*，但对特性值中的每个启用位调用函数 *EnumToString*，之后结果字符串被“粘合在一起”。

现在可以用类似的方式在语句 *switch* 中处理 *SYMBOL_ORDER_MODE*：

```

case SYMBOL_ORDER_MODE:
    return maskstr<SYMBOL_ORDER>(v);

```

下面是 *ENUM_SYMBOL_INFO_INTEGER* 的 *stringify* 方法的完整代码。所有的特性和枚举，我们将在后面的章节中逐步了解。

```

string stringify(const ENUM_SYMBOL_INFO_INTEGER property) const
{
    const long v = SymbolInfoInteger(name, property);
    switch(property)
    {
        case SYMBOL_SELECT:
        case SYMBOL_SPREAD_FLOAT:
        case SYMBOL_VISIBLE:
        case SYMBOL_CUSTOM:
        case SYMBOL_MARGIN_HEDGED_USE_LEG:
        case SYMBOL_EXIST:
            return boolean(v);
        case SYMBOL_TIME:
            return TimeToString(v, TIME_DATE|TIME_SECONDS);
        case SYMBOL_TRADE_CALC_MODE:
            return enumstr<ENUM_SYMBOL_CALC_MODE>(v);
        case SYMBOL_TRADE_MODE:
            return enumstr<ENUM_SYMBOL_TRADE_MODE>(v);
        case SYMBOL_TRADE_EXEMODE:
            return enumstr<ENUM_SYMBOL_TRADE_EXECUTION>(v);
        case SYMBOL_SWAP_MODE:
            return enumstr<ENUM_SYMBOL_SWAP_MODE>(v);
        case SYMBOL_SWAP_ROLLOVER3DAYS:
            return enumstr<ENUM_DAY_OF_WEEK>(v);
        case SYMBOL_EXPIRATION_MODE:
            return maskstr<SYMBOL_EXPIRATION>(v);
        case SYMBOL_FILLING_MODE:
            return maskstr<SYMBOL_FILLING>(v);
        case SYMBOL_START_TIME:
        case SYMBOL_EXPIRATION_TIME:
            return TimeToString(v);
        case SYMBOL_ORDER_MODE:
            return maskstr<SYMBOL_ORDER>(v);
        case SYMBOL_OPTION_RIGHT:
            return enumstr<ENUM_SYMBOL_OPTION_RIGHT>(v);
        case SYMBOL_OPTION_MODE:
            return enumstr<ENUM_SYMBOL_OPTION_MODE>(v);
        case SYMBOL_CHART_MODE:
            return enumstr<ENUM_SYMBOL_CHART_MODE>(v);
        case SYMBOL_ORDER_GTC_MODE:
            return enumstr<ENUM_SYMBOL_ORDER_GTC_MODE>(v);
        case SYMBOL_SECTOR:
            return enumstr<ENUM_SYMBOL_SECTOR>(v);
        case SYMBOL_INDUSTRY:
            return enumstr<ENUM_SYMBOL_INDUSTRY>(v);
        case SYMBOL_BACKGROUND_COLOR: // Bytes: Transparency Blue Green Red
            return StringFormat("TBGR(0x%08X)", v);
    }

    return (string)v;
}

```

为了测试 *SymbolMonitor* 类, 我们创建了一个简单脚本 *SymbolMonitor.mq5*。该脚本记录了工作图表交易品种的所有特性。

```
#include <MQL5Book/SymbolMonitor.mqh>

void OnStart()
{
    SymbolMonitor m;
    m.list2log<ENUM_SYMBOL_INFO_INTEGER>();
    m.list2log<ENUM_SYMBOL_INFO_DOUBLE>();
    m.list2log<ENUM_SYMBOL_INFO_STRING>();
}
```

例如, 如果我们在 EURUSD 图表上运行该脚本, 我们可以得到以下记录 (以缩写形式给出)。


```

ENUM_SYMBOL_INFO_INTEGER Count=36
    0 SYMBOL_SELECT=true
    ...
    4 SYMBOL_TIME=2022.01.12 10:52:22
    5 SYMBOL_DIGITS=5
    6 SYMBOL_SPREAD=0
    7 SYMBOL_TICKS_BOOKDEPTH=10
    8 SYMBOL_TRADE_CALC_MODE=SYMBOL_CALC_MODE_FOREX
    9 SYMBOL_TRADE_MODE=SYMBOL_TRADE_MODE_FULL
    10 SYMBOL_TRADE_STOPS_LEVEL=0
    11 SYMBOL_TRADE_FREEZE_LEVEL=0
    12 SYMBOL_TRADE_EXEMODE=SYMBOL_TRADE_EXECUTION_INSTANT
    13 SYMBOL_SWAP_MODE=SYMBOL_SWAP_MODE_POINTS
    14 SYMBOL_SWAP_ROLLOVER3DAYS=WEDNESDAY
    15 SYMBOL_SPREAD_FLOAT=true
    16 SYMBOL_EXPIRATION_MODE=_SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY »
        _SYMBOL_EXPIRATION_SPECIFIED _SYMBOL_EXPIRATION_SPECIFIED_DAY
    17 SYMBOL_FILLING_MODE=_SYMBOL_FILLING_FOK
    ...
    23 SYMBOL_ORDER_MODE=_SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP »
        _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP _SYMBOL_ORDER_CLOSEBY
    ...
    26 SYMBOL_VISIBLE=true
    27 SYMBOL_CUSTOM=false
    28 SYMBOL_BACKGROUND_COLOR=TBGR(0xFF000000)
    29 SYMBOL_CHART_MODE=SYMBOL_CHART_MODE_BID
    30 SYMBOL_ORDER_GTC_MODE=SYMBOL_ORDERS_GTC
    31 SYMBOL_MARGIN_HEDGED_USE_LEG=false
    32 SYMBOL_EXIST=true
    33 SYMBOL_TIME_MSC=1641984742149
    34 SYMBOL_SECTOR=SECTOR_CURRENCY
    35 SYMBOL_INDUSTRY=INDUSTRY_UNDEFINED
ENUM_SYMBOL_INFO_DOUBLE Count=57
    0 SYMBOL_BID=1.13681
    1 SYMBOL_BIDHIGH=1.13781
    2 SYMBOL_BIDLOW=1.13552
    3 SYMBOL_ASK=1.13681
    4 SYMBOL_ASKHIGH=1.13781
    5 SYMBOL_ASKLOW=1.13552
    ...
    12 SYMBOL_POINT=1e-05
    13 SYMBOL_TRADE_TICK_VALUE=1.0
    14 SYMBOL_TRADE_TICK_SIZE=1e-05
    15 SYMBOL_TRADE_CONTRACT_SIZE=100000.0
    16 SYMBOL_VOLUME_MIN=0.01
    17 SYMBOL_VOLUME_MAX=500.0
    18 SYMBOL_VOLUME_STEP=0.01
    19 SYMBOL_SWAP_LONG=-0.7
    20 SYMBOL_SWAP_SHORT=-1.0
    21 SYMBOL_MARGIN_INITIAL=0.0
    22 SYMBOL_MARGIN_MAINTENANCE=0.0

```

```
...
28 SYMBOL_TRADE_TICK_VALUE_PROFIT=1.0
29 SYMBOL_TRADE_TICK_VALUE_LOSS=1.0
...
43 SYMBOL_MARGIN_HEDGED=100000.0
...
47 SYMBOL_PRICE_CHANGE=0.0132
ENUM_SYMBOL_INFO_STRING Count=15
0 SYMBOL_BANK=
1 SYMBOL_DESCRIPTION=Euro vs US Dollar
2 SYMBOL_PATH=Forex\EURUSD
3 SYMBOL_CURRENCY_BASE=EUR
4 SYMBOL_CURRENCY_PROFIT=USD
5 SYMBOL_CURRENCY_MARGIN=EUR
...
13 SYMBOL_SECTOR_NAME=Currency
```

特别地，可以看到，交易品种价格是用 5 位数字播报的 (SYMBOL_DIGITS)，该交易品种确实存在 (SYMBOL_EXIST)，合约规模为 100000.0 (SYMBOL_TRADE_CONTRACT_SIZE) 等。所有信息都符合规范。

6.1.9 检查交易品种状态

前面我们看了几个与交易品种状态相关的函数。注意，*SymbolExist* 用于检查某个交易品种是否存在，*SymbolSelect* 用于检查是否包含在 *Market Watch* 列表中。在交易品种的特性中，有几个用途相似的标志，与上述函数相比，它们的使用既有优点也有缺点。

特别是，SYMBOL_SELECT 特性可用于确定在 *Market Watch* 中是否选择了指定的交易品种，而 *SymbolSelect* 函数可更改该特性。

与类似的 SYMBOL_EXIST 特性不同，*SymbolExist* 函数还可以在输出变量中填充一个指示，用于表明交易品种是否为用户定义交易品种。查询特性时，必须分别分析这两个特性，因为自定义交易品种的特性存储在另一个特性 *SYMBOL_CUSTOM* 中。但是，在某些情况下，程序可能只需要一个特性，这样就可以进行单独查询。

所有标志都是通过 *SymbolInfoInteger* 函数获得的布尔值。

标识符	说明
SYMBOL_EXIST	表示具有给定名称的交易品种存在
SYMBOL_SELECT	表示该交易品种已被选中显示在 <i>Market Watch</i>
SYMBOL_VISIBLE	表示指定的交易品种已被选中显示在 <i>Market Watch</i>

特别值得关注的是 SYMBOL_VISIBLE。事实上，一些交易品种 (通常，这些交易品种为计算保证金要求和存款货币利润所必需的交叉汇率)可在 *Market Watch* 中自动选择，并且不会显示在用户可见的列表中。这种交易品种必须进行明确地选择 (由用户选择或以编程方式选择)来显示。因此，正是 SYMBOL_VISIBLE 特性允许你确定某个交易品种在窗口中是否可见：它可以等于 *false* (对于该列表中的某些元素)，这是使用一对函数 *SymbolsTotal* 和 *SymbolName* 在 *selected* 参数等于 *true* 的情况下获得的。

考虑一个简单脚本 (*SymbolInvisible.mq5*), 该脚本在终端中搜索隐式选择的交易品种, 即那些没有在市场报价中显示的交易品种 (SYMBOL_VISIBLE 被重置), 同时其 SYMBOL_SELECT 为 *true*。

```
#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
    const int n = SymbolsTotal(false);
    int selected = 0;
    string invisible[];
    // loop through all available symbols
    for(int i = 0; i < n; ++i)
    {
        const string s = SymbolName(i, false);
        if(SymbolInfoInteger(s, SYMBOL_SELECT))
        {
            selected++;
            if(!SymbolInfoInteger(s, SYMBOL_VISIBLE))
            {
                // collect selected but invisible symbols into an array
                PUSH(invisible, s);
            }
        }
    }
    PrintFormat("Symbols: total=%d, selected=%d, implicit=%d",
        n, selected, ArraySize(invisible));
    if(ArraySize(invisible))
    {
        ArrayPrint(invisible);
    }
}
```

尝试以不同账户上编译和运行脚本。并不总会遇到隐式选择交易品种的情况。例如, 如果在 *Market Watch* 中选择了以卢布报价的俄罗斯蓝筹股的分时报价, 并且交易账户使用不同的货币 (例如, 美元或欧元, 但不是卢布), 那么将会自动选择 USDRUB 交易品种。当然, 这是假设其之前没有被明确地添加到 *Market Watch* 中。然后, 我们可以在日志中得到以下结果:

```
Symbols: total=50681, selected=49, implicit=1
"USDRUB"
```

6.1.10 用于构建交易品种图表的价格类型

MetaTrader 5 价格图表上的柱线可以根据 *Bid* 或 *Last* 价格绘制, 绘制类型在每个金融工具的规范中指明。MQL 程序可以通过对 SYMBOL_CHART_MODE 特性调用 *SymbolInfoInteger* 函数来找到这一特征。返回值为 ENUM_SYMBOL_CHART_MODE 枚举之一。

标识符	说明
SYMBOL_CHART_MODE_BID	柱线按买入价格构建
SYMBOL_CHART_MODE_LAST	柱线按最新价格构建

具有 **Last** 价格的模式用作在交易所交易的交易品种 (与分散的外汇市场相反), 而 **市场深度** 可用于这种交易品种。可依据 **SYMBOL_TICKS_BOOKDEPTH** 特性可以找到市场深度。

SYMBOL_CHART_MODE 的特性对于调整所构建的指标或策略的信号很有用, 例如, 在图表的 **Last** 价格; 而订单将“以市价”执行, 即, 根据方向以 **Ask** 或 **Bid** 价格执行。

此外, 在计算 **自定义金融工具** 的柱线时, 价格类型也是必需的: 如果价格类型取决于标准交易品种, 则按价格类型考虑其设置是有意义的。若用户将 **合成金融工具** 的公式输入 **Custom Symbol** 窗口 (通过选择 **Symbols** 对话框中的 **Create Symbol** 打开), 可以根据所使用的各个标准交易品种的规范来选择价格类型。但是, 当计算算法在 MQL 程序中形成时, 精确地说, 其负责正确选择价格类型。

首先, 让我们收集使用 **Bid** 和 **Last** 价格的统计数据, 以构建特定账户的图表。这就是脚本 **SymbolStatsByPriceType.mq5** 的功能。

```
const bool MarketWatchOnly = false;

void OnStart()
{
    const int n = SymbolsTotal(MarketWatchOnly);
    int k = 0;
    // loop through all available characters
    for(int i = 0; i < n; ++i)
    {
        if(SymbolInfoInteger(SymbolName(i, MarketWatchOnly), SYMBOL_CHART_MODE)
            == SYMBOL_CHART_MODE_LAST)
        {
            k++;
        }
    }
    PrintFormat("Symbols in total: %d", n);
    PrintFormat("Symbols using price types: Bid=%d, Last=%d", n - k, k);
}
```

用不同账户上尝试运行该脚本 (有些账户可能没有股票交易品种)。结果如下所示:

```
Symbols in total: 52304
Symbols using price types: Bid=229, Last=52075
```

一个更实际的示例是指标 **SymbolBidAskChart.mq5**, 其设计用于根据指定类型的价格绘制柱线形式。这将允许你将使用 **SYMBOL_CHART_MODE** 特性中价格构建的蜡烛图与其他价格类型的柱线进行比较。例如, 你可以在金融工具图表的 **Last** 价格处看到 **Bid** 价格的柱线或者得到 **Ask** 价格的柱线, 而标准终端图表不支持这些功能。

作为新指标的依据, 我们将采用一个现成指标 **IndDeltaVolume.mq5**, 如 **等待数据和管理可见性** 一节所述。在这个指标中, 我们下载了一定数量柱线 **BarCount** 的分时报价历史, 并计算了交易量的增量, 即分别买入和卖出的交易量。在新指标中, 我们只需要将计算算法替换为基于每根柱线内分时报价搜索 **Open**、**High**、**Low** 和 **Close** 价格。

指标设置包括显示在主窗口中的四个缓冲区和一根柱线图 (DRAW_BARS)。

```

#property indicator_chart_window
#property indicator_buffers 4
#property indicator_plots 1

#property indicator_type1 DRAW_BARS
#property indicator_color1 clrDodgerBlue
#property indicator_width1 2
#property indicator_label1 "Open;High;Low;Close;"

```

选择显示为柱线是为了在主蜡烛图上运行时更容易阅读，同时显示每根柱线的两个版本。

新增的 *ChartMode* 输入参数允许用户从三种价格类型中选择一种（注意，与 `ENUM_SYMBOL_CHART_MODE` 中的标准元素集相比，*Ask* 是我们添加的元素）。

```

enum ENUM_SYMBOL_CHART_MODE_EXTENDED
{
    _SYMBOL_CHART_MODE_BID, // SYMBOL_CHART_MODE_BID
    _SYMBOL_CHART_MODE_LAST, // SYMBOL_CHART_MODE_LAST
    _SYMBOL_CHART_MODE_ASK, // SYMBOL_CHART_MODE_ASK*
};

input int BarCount = 100;
input COPY_TICKS TickType = INFO_TICKS;
input ENUM_SYMBOL_CHART_MODE_EXTENDED ChartMode = _SYMBOL_CHART_MODE_BID;

```

以前的 *CalcDeltaVolume* 类将其名称改为 *CalcCustomBars*，但几乎保持不变。不同之处包括一组新的四个缓冲区和 *chartMode* 字段，该字段在构造函数中由输入变量 *ChartMode* 初始化。

```

class CalcCustomBars
{
    const int limit;
    const COPY_TICKS tickType;
    const ENUM_SYMBOL_CHART_MODE_EXTENDED chartMode;

    double open[];
    double high[];
    double low[];
    double close[];
    ...
public:
    CalcCustomBars(
        const int bars,
        const COPY_TICKS type,
        const ENUM_SYMBOL_CHART_MODE_EXTENDED mode)
        : limit(bars), tickType(type), chartMode(mode) ...
    {
        // register arrays as indicator buffers
        SetIndexBuffer(0, open);
        SetIndexBuffer(1, high);
        SetIndexBuffer(2, low);
        SetIndexBuffer(3, close);
        const static string defTitle[] = {"Open;High;Low;Close;"};
        const static string types[] = {"Bid", "Last", "Ask"};
        string name = defTitle[0];
        StringReplace(name, ";", types[chartMode] + ";");
        PlotIndexSetString(0, PLOT_LABEL, name);
        IndicatorSetInteger(INDICATOR_DIGITS, _Digits);
    }
    ...
}

```

根据 *chartMode* 的模式，辅助方法 *price* 可从每条分时报价返回特定的价格类型。

```

protected:
    double price(const MqlTick &t) const
    {
        switch(chartMode)
        {
            case _SYMBOL_CHART_MODE_BID:
                return t.bid;
            case _SYMBOL_CHART_MODE_LAST:
                return t.last;
            case _SYMBOL_CHART_MODE_ASK:
                return t.ask;
        }
        return 0; // error
    }
    ...
}

```

使用 *price* 方法，我们可以很容易地实现对主计算方法 *calc* 的修改，该方法用于根据该柱线的 *ticks* 数组来填充第 *i* 根柱线的缓冲区。

```

void calc(const int i, const MqlTick &ticks[], const int skip = 0)
{
    const int n = ArraySize(ticks);
    for(int j = skip; j < n; ++j)
    {
        const double p = price(ticks[j]);
        if(open[i] == EMPTY_VALUE)
        {
            open[i] = p;
        }

        if(p > high[i] || high[i] == EMPTY_VALUE)
        {
            high[i] = p;
        }

        if(p < low[i])
        {
            low[i] = p;
        }

        close[i] = p;
    }
}

```

其余的源代码片段及其工作原理对应于 *IndDeltaVolume.mq5* 的说明。

在 *OnInit* 处理程序中，我们还显示了图表的当前价格类型，如果用户决定在缺少 *Last* 的情况下基于该金融工具的 *Last* 价格类型构建一个指标，系统将会返回警告。

```

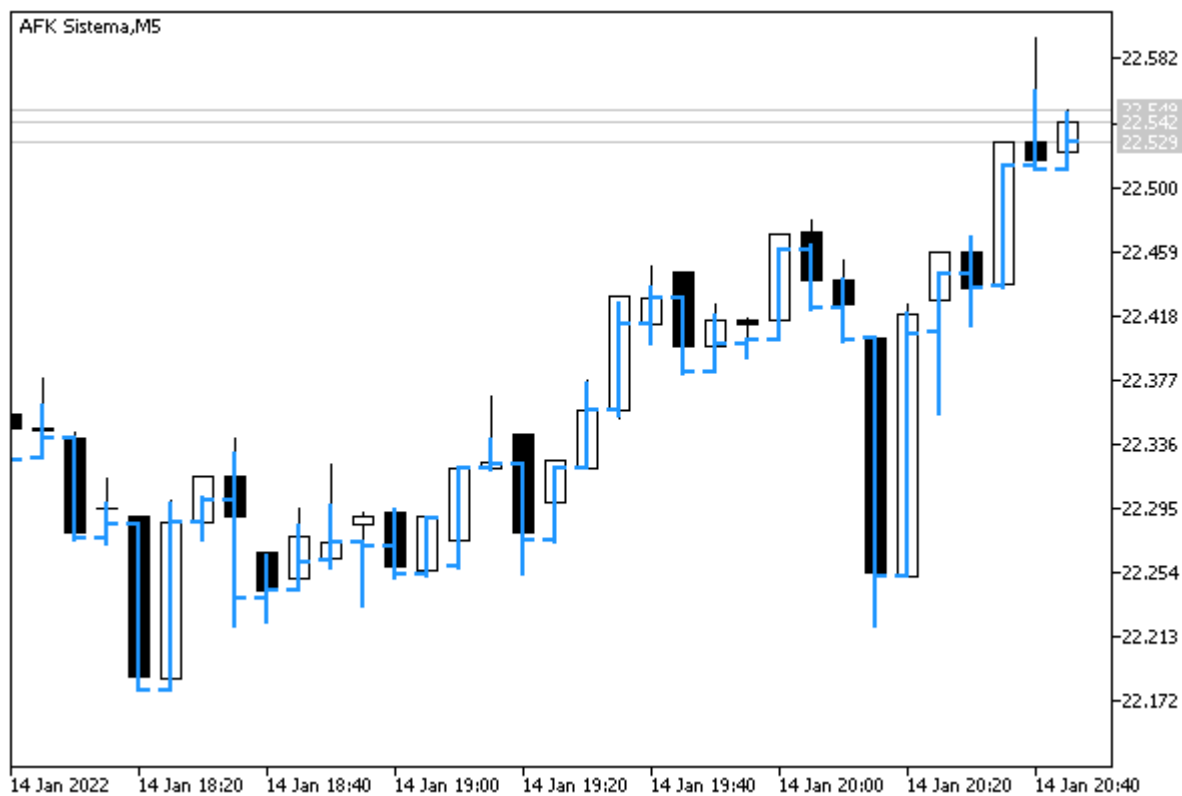
int OnInit()
{
    ...
    ENUM_SYMBOL_CHART_MODE mode =
        (ENUM_SYMBOL_CHART_MODE)SymbolInfoInteger(_Symbol, SYMBOL_CHART_MODE);
    Print("Chart mode: ", EnumToString(mode));

    if(mode == SYMBOL_CHART_MODE_BID
        && ChartMode == _SYMBOL_CHART_MODE_LAST)
    {
        Alert("Last price is not available for ", _Symbol);
    }

    return INIT_SUCCEEDED;
}

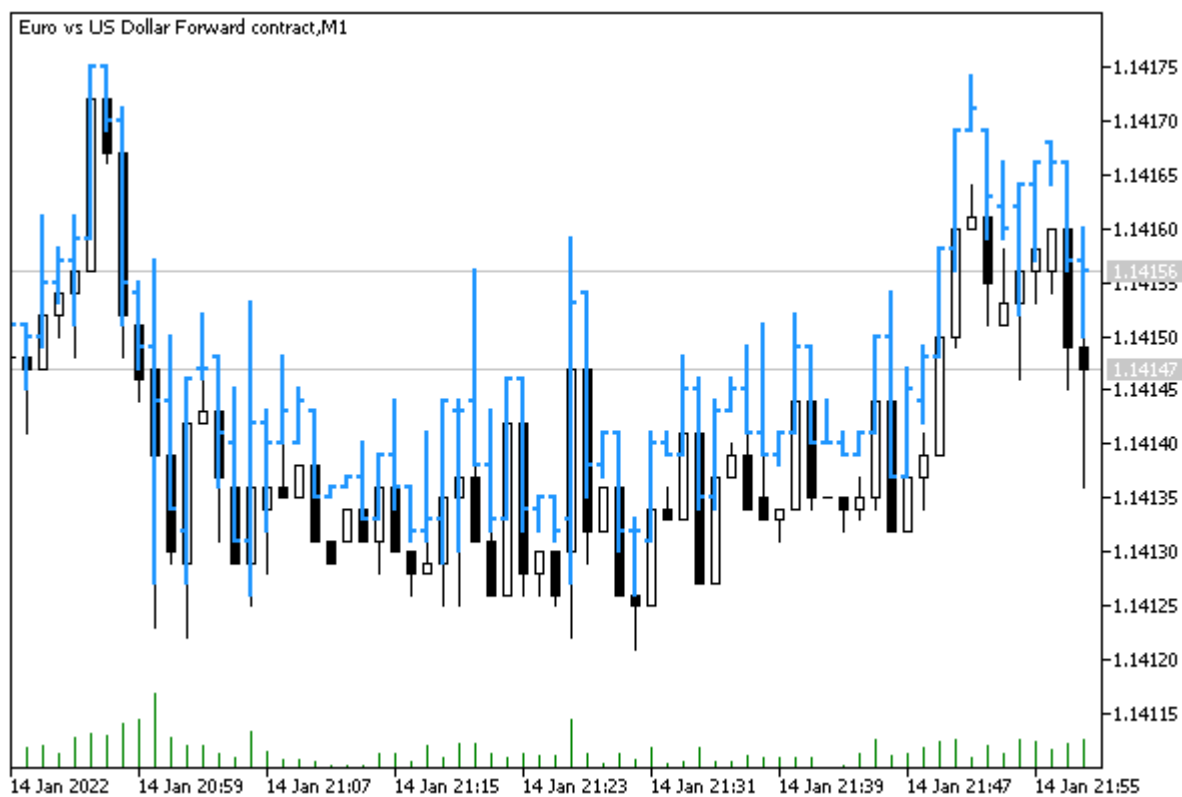
```

下面是基于 *Last* 价格图表绘图模式的工具屏幕截图；价格类型 *Bid* 的指标位于图表上方。



最新价格图表上带有买入价柱线的指标

观察普通 *Bid* 价格图表上叠加的 *Ask* 价格柱线也很有趣。



在买入价格图表上叠加卖出价格柱线的指标

在低流动性时段，当点差扩大时，你可以看到 *Bid* 和 *Ask* 价格图表之间的巨大差异。

6.1.11 金融工具的基础货币、报价货币和保证金货币

每种金融工具最重要的特性之一是其工作货币：

- 表示购买或出售资产的基础货币（对于外汇金融工具）
- 利润计算（报价）货币
- 保证金计算货币

MQL 程序可以使用 *SymbolInfoString* 函数和下表中的三个特性来获取这些货币的名称。

标识符	说明
SYMBOL_CURRENCY_BASE	基础货币
SYMBOL_CURRENCY_PROFIT	盈利货币
SYMBOL_CURRENCY_MARGIN	保证金货币

这些特性有助于分析外汇金融工具，许多经纪商会在这些金融工具的名称中添加各种前缀和后缀，以及兑换金融工具。特别是，其算法将能够找到一个交易品种，以获得两种给定货币的交叉汇率或选择具有给定共同报价货币的指数组合。

有一个很常见的任务是根据特定需求来搜索工具，因此我们创建一个类 *SymbolFilter* (*SymbolFilter.mqh*) 来构建一个合适交易品种及其选定特性的列表。将来，我们不仅要用这个类来分析货币，还要分析其他特性。

首先，我们将考虑一个简化版本，然后用方便的功能予以补充。

开发中，我们将使用现成的辅助工具：一个关联映射数组 (*MapArray.mqh*) 来存储所选类型的键值对和一个交易品种特性监视器 (*SymbolMonitor.mqh*)。

```
#include <MQL5Book/MapArray.mqh>
#include <MQL5Book/SymbolMonitor.mqh>
```

为了简化在数组中累加工作结果的语句，我们使用了 `PUSH` 宏的改进版本，我们已经在前面的示例中看到过该版本及其多维数组的扩展版本（在这种情况下，不可能进行简单的赋值）。

```
#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1, ArraySize(A) * 2) - 1] = V)
#define EXPAND(A) (ArrayResize(A, ArrayRange(A, 0) + 1, ArrayRange(A, 0) * 2) - 1)
```

SymbolFilter 类的对象必须存储用于筛选交易品种的特性值。因此，我们将在类中描述三个 *MapArray* 数组，分别用于整数、实数和字符串特性。

```
class SymbolFilter
{
    MapArray<ENUM_SYMBOL_INFO_INTEGER,long> longs;
    MapArray<ENUM_SYMBOL_INFO_DOUBLE,double> doubles;
    MapArray<ENUM_SYMBOL_INFO_STRING,string> strings;
    ...
}
```

使用重载的 *let* 方法来设置所需的筛选特性。

```

public:
    SymbolFilter *let(const ENUM_SYMBOL_INFO_INTEGER property, const long value)
    {
        longs.put(property, value);
        return &this;
    }

    SymbolFilter *let(const ENUM_SYMBOL_INFO_DOUBLE property, const double value)
    {
        doubles.put(property, value);
        return &this;
    }

    SymbolFilter *let(const ENUM_SYMBOL_INFO_STRING property, const string value)
    {
        strings.put(property, value);
        return &this;
    }
    ...

```

请注意，这些方法会返回一个指向过滤器的指针，允许你将条件写成一个链：例如，如果在代码的前面说明了 *SymbolFilter* 类的一个对象 *f*，那么你可以对价格类型和盈利货币的名称施加两个条件，如下所示：

```
f.let(SYMBOL_CHART_MODE, SYMBOL_CHART_MODE_LAST).let(SYMBOL_CURRENCY_PROFIT, "USD");
```

满足条件的交易品种数组排列是由筛选对象在 *select* 方法的几种变体中执行的，下面给出了其中最简单的一种（其他选项将在后面讨论）。

watch 参数定义了交易品种的搜索上下文：搜索 *Market Watch* 中搜索对应 *true*，搜索所有可用交易品种则对应 *false*。将用匹配交易品种的名称来填充输出数组 *symbols*。我们已经知道了该方法内部的代码结构：其中包含一个循环，该循环会遍历交易品种，并为每个交易品种创建一个监视对象 *m*。

```

void select(const bool watch, string &symbols[]) const
{
    const int n = SymbolsTotal(watch);
    for(int i = 0; i < n; ++i)
    {
        const string s = SymbolName(i, watch);
        SymbolMonitor m(s);
        if(match<ENUM_SYMBOL_INFO_INTEGER,long>(m, longs)
            && match<ENUM_SYMBOL_INFO_DOUBLE,double>(m, doubles)
            && match<ENUM_SYMBOL_INFO_STRING,string>(m, strings))
        {
            PUSH(symbols, s);
        }
    }
}

```

借助监视器，我们可以以统一的方式获得任何特性的值。检查与 *longs*、*doubles* 和 *strings* 数组中的条件集匹配的当前交易品种的特性是否是通过辅助方法 *match* 实现的。只有所有请求的特性匹配，交易品种名称才会保存在 *symbols* 的输出数组中。

在最简单的情况下，*match* 方法的实现如下（随后其将被改变）。

```
protected:
    template<typename K,typename V>
    bool match(const SymbolMonitor &m, const MapArray<K,V> &data) const
    {
        for(int i = 0; i < data.getSize(); ++i)
        {
            const K key = data.getKey(i);
            if(!equal(m.get(key), data.getValue(i)))
            {
                return false;
            }
        }
        return true;
    }
}
```

如果 *data* 数组中至少有一个值与相应的字符特性不匹配，则该方法返回 *false*。如果所有特性都匹配（或者这种类型的特性没有条件），则该方法返回 *true*。

使用 *equal* 比较两个值。鉴于在这些特性中可能存在 *double* 类型的特性，其实现并不像人们想象的那么简单。

```
template<typename V>
static bool equal(const V v1, const V v2)
{
    return v1 == v2 || eps(v1, v2);
}
```

对于 *double* 类型，表达式 *v1 == v2* 可能不适用于接近的数字，因此应考虑实数 *DBL_EPSILON* 类型的精度。这是在一个单独的方法 *eps* 中完成的，由于模板的原因，该方法可为 *double* 类型和其他所有类型分别重载。

```
static bool eps(const double v1, const double v2)
{
    return fabs(v1 - v2) < DBL_EPSILON * fmax(v1, v2);
}

template<typename V>
static bool eps(const V v1, const V v2)
{
    return false;
}
```

如果除 *double* 之外的任何类型的值都相等，则不会调用模板方法 *eps*；在所有其他情况下（包括值不同时），其会按要求返回 *false*（因此，*v1 == v2* 为唯一条件）。

上述筛选器选项仅允许你检查特性是否相等。但在实践中，经常需要分析不相等的条件，以及大于/小于的条件。出于这个原因，*SymbolFilter* 类具有带基本比较运算的 *IS* 枚举（如果需要，可以进行补充）。

```

class SymbolFilter
{
    ...
    enum IS
    {
        EQUAL,
        GREATER,
        NOT_EQUAL,
        LESS
    };
    ...

```

对于 `ENUM_SYMBOL_INFO_INTEGER`、`ENUM_SYMBOL_INFO_DOUBLE` 和 `ENUM_SYMBOL_INFO_STRING` 枚举中的每个特性，不仅需要保存所需的特性值（注意关联数组 *longs*、*doubles* 和 *strings*），还需要保存新的 *IS* 枚举中的比较方法。

由于标准枚举的元素具有不重叠的值（有一种与 *交易量* 相关的例外，但不是很重要），因此为比较方法保留一个公共映射数组 *conditions* 是有意义的。这就提出了一个问题：为映射键选择哪种类型才能从技术上“组合”不同的枚举。为了做到这一点，我们必须说明伪枚举 `ENUM_ANY`，其仅表示某种类型的泛型枚举。注意，所有枚举都有一个等价于整数 *int* 的内部表示，因此可以相互简化。

```

enum ENUM_ANY
{
};

MapArray<ENUM_ANY,IS> conditions;
MapArray<ENUM_ANY,long> longs;
MapArray<ENUM_ANY,double> doubles;
MapArray<ENUM_ANY,string> strings;
...

```

现在，我们可以通过添加指定比较方法的 *cmp* 输入参数来补全设置所需特性值的所有 *let* 方法。默认情况下，该参数用于设置相等检查（`EQUAL`）。

```

SymbolFilter *let(const ENUM_SYMBOL_INFO_INTEGER property, const long value,
    const IS cmp = EQUAL)
{
    longs.put((ENUM_ANY)property, value);
    conditions.put((ENUM_ANY)property, cmp);
    return &this;
}

```

这是整数特性的一个变体。其他两个重载以同样的方式改变。

考虑到关于比较和同时消除映射数组中不同类型键的不同方法的新信息，我们修改了 *match* 方法。其中，对于每个指定的特性，我们基于 *data* 映射数组中的键从 *conditions* 数组中检索一个条件，并使用 *switch* 运算符执行适当的检查。

```

template<typename V>
bool match(const SymbolMonitor &m, const MapArray<ENUM_ANY,V> &data) const
{
    // dummy variable to select m.get method overload below
    static const V type = (V)NULL;
    // cycle by conditions imposed on the properties of the symbol
    for(int i = 0; i < data.getSize(); ++i)
    {
        const ENUM_ANY key = data.getKey(i);
        // choice of comparison method in the condition
        switch(conditions[key])
        {
            case EQUAL:
                if(!equal(m.get(key, type), data.getValue(i))) return false;
                break;
            case NOT_EQUAL:
                if(equal(m.get(key, type), data.getValue(i))) return false;
                break;
            case GREATER:
                if(!greater(m.get(key, type), data.getValue(i))) return false;
                break;
            case LESS:
                if(greater(m.get(key, type), data.getValue(i))) return false;
                break;
        }
    }
    return true;
}

```

新模板 *greater* 方法实现起来很简单。

```

template<typename V>
static bool greater(const V v1, const V v2)
{
    return v1 > v2;
}

```

现在, *match* 方法调用可以以更短的形式编写, 因为模板 *V* 的唯一剩余类型是由传递的 *data* 自变量自动确定的 (*longs*、*doubles* 或 *strings* 数组中的一个)。

```

void select(const bool watch, string &symbols[]) const
{
    const int n = SymbolsTotal(watch);
    for(int i = 0; i < n; ++i)
    {
        const string s = SymbolName(i, watch);
        SymbolMonitor m(s);
        if(match(m, longs)
            && match(m, doubles)
            && match(m, strings))
        {
            PUSH(symbols, s);
        }
    }
}

```

这还不是 *SymbolFilter* 类的最终版本，但是我们已经可以测试其实际运行效果了。

我们来创建一个脚本 *SymbolFilterCurrency.mq5*，该脚本可根据基础货币和盈利货币的特性筛选交易品种；此处为美元。默认情况下，*MarketWatchOnly* 参数仅在 *Market Watch* 中进行搜索。

```

#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
    SymbolFilter f; // filter object
    string symbols[]; // array for results
    ...

```

比方说，我们希望找到有直接报价的外汇金融工具，即“美元”出现在其名称的开头。为了不依赖于特定经纪商的名称构成细节，我们将使用 *SYMBOL_CURRENCY_BASE* 特性，其包含第一种货币。

让我们写下交易品种的基础货币等于美元的条件，并应用过滤器。

```

f.let(SYMBOL_CURRENCY_BASE, "USD")
.select(MarketWatchOnly, symbols);
Print("==== Base is USD =====");
ArrayPrint(symbols);
...

```

生成的数组会输出到日志中。

```

===== Base is USD =====
"USDCHF" "USDJPY" "USDCNH" "USDRUB" "USDCAD" "USDSEK" "SP500m" "Brent"

```

如你所见，该数组不仅包括交易代码以美元开头的外汇交易品种，还包括 S&P500 指数和商品（原油）。后两个交易品种以美元报价，但它们也有相同的基础货币。同时，外汇交易品种的报价货币（也是盈利货币）是第二位的，且不同于美元。这允许你补充筛选条件，让非外汇交易品种不再匹配。

我们清除数组，添加一个条件，即盈利货币不等于“美元”，并再次请求合适的交易品种（前面的条件保存在 *f* 对象中）。

```
...
ArrayResize(symbols, 0);

f.let(SYMBOL_CURRENCY_PROFIT, "USD", SymbolFilter::IS::NOT_EQUAL)
.select(MarketWatchOnly, symbols);
Print("==== Base is USD and Profit is not USD =====");
ArrayPrint(symbols);
}
```

这次，日志中实际上只显示了你要查找的交易品种。

```
==== Base is USD and Profit is not USD =====
"USDCHF" "USDJPY" "USDCNH" "USDRUB" "USDCAD" "USDSEK"
```

6.1.12 价格表示的准确性和变化步长

前面，我们已经遇到了图表工作交易品种的两个相互关联的特性：最小价格变化步长（点）和以小数位数表示的价格表示准确性（位数）。二者在 预定义变量 中也是可用的。要获取任意交易品种的相似特性，应分别查询 SYMBOL_POINT 和 SYMBOL_DIGITS 特性。SYMBOL_POINT 特性与最小价格变化（在 MQL 程序中称为 SYMBOL_TRADE_TICK_SIZE 特性）及其值（SYMBOL_TRADE_TICK_VALUE）密切相关，通常以交易账户的货币表示（但有些交易品种可以配置为使用基础货币；如有必要，你可以联系你的经纪商了解详情）。下表显示了所有这些特性。

标识符	说明
SYMBOL_DIGITS	小数位数
SYMBOL_POINT	报价货币中一个点的值
SYMBOL_TRADE_TICK_VALUE	SYMBOL_TRADE_TICK_VALUE_PROFIT 的值
SYMBOL_TRADE_TICK_VALUE_PROFIT	盈利头寸的当前分时报价值
SYMBOL_TRADE_TICK_VALUE_LOSS	亏损头寸的当前分时报价值
SYMBOL_TRADE_TICK_SIZE	以报价货币表示的最小价格变化

除 SYMBOL_DIGITS 之外的所有特性都是实数，并使用 *SymbolInfoDouble* 函数请求。SYMBOL_DIGITS 特性可通过 *SymbolInfoInteger* 获得。为了测试使用这些特性的工作，我们将使用现成的类 *SymbolFilter* 和 *SymbolMonitor*，它们将用于自动为任何特性调用所需的函数。

我们还可以通过添加 *select* 方法的新重载来改进 *SymbolFilter* 类，该方法不仅能够用合适交易品种的名称填充一个数组，还能够用其特定特性值填充另一个数组。

在更普遍的情况下，我们可能同时对每个交易品种的几个特性感兴趣，因此建议不要对输出数组使用内置数据类型，而是使用具有不同字段的特殊复合类型。

在编程中，这种类型被称为元组，在某种程度上相当于 MQL5 结构体。

```

template<typename T1,typename T2,typename T3> // we can describe up to 64 fields
struct Tuple3                                // MQL5 allows 64 template parameters
{
    T1 _1;
    T2 _2;
    T3 _3;
};

```

但是，此类结构体需要所有字段的初步说明，而我们事先不知道所请求交易品种特性的数量和列表。因此，为了简化代码，我们将元组表示为接收查询结果的动态数组的第二维向量。

```
T array[][S];
```

作为数据类型 *T*，我们可以使用任何用于特性的内置类型和枚举。大小 *S* 必须与请求的特性数量相匹配。

实际上，这种简化将我们限制在一个查询中只能查询相同类型的值，也就是说，只能查询整数、实数或字符串。但是，筛选条件可以包括任何特性。稍后，我们将使用其他交易实体的筛选示例来实现元组方法，包括订单、交易和 [头寸](#)。

因此，新版本的 *SymbolFilter::select* 方法将对 *property* 数组的引用作为输入，该数组带有要从筛选后的交易品种中读取的特性标识符。交易品种本身的名称和这些特性的值将被写入 *symbols* 和 *data* 输出数组。


```

template<typename E,typename V>
bool select(const bool watch, const E &property[], string &symbols[],
            V &data[][], const bool sort = false) const
{
    // the size of the array of requested properties must match the output tuple
    const int q = ArrayRange(data, 1);
    if(ArraySize(property) != q) return false;

    const int n = SymbolsTotal(watch);
    // iterate over characters
    for(int i = 0; i < n; ++i)
    {
        const string s = SymbolName(i, watch);
        // access to the symbol properties is provided by the monitor
        SymbolMonitor m(s);
        // check all filter conditions
        if(match(m, longs)
            && match(m, doubles)
            && match(m, strings))
        {
            // properties of a suitable symbol are written to arrays
            const int k = EXPAND(data);
            for(int j = 0; j < q; ++j)
            {
                data[k][j] = m.get(property[j]);
            }
            PUSH(symbols, s);
        }
    }

    if(sort)
    {
        ...
    }
    return true;
}

```

此外，新方法可以按照第一维度（请求的第一个特性）对输出数组进行排序：该功能留给读者使用源代码独立研究。要启用排序，请将 `sort` 参数设置为 `true`。带有交易品种名和数据的数组排序一致。

当只需要从筛选的字符中请求一个特性时，为了避免调用代码中的元组，在 `SymbolFilter` 中实现了下面的 `select` 选项：在其内部，我们在第二维度中定义了大小为 1 的特性（*properties*）和值（*tuples*）的中间数组，用于调用上面的完整版本的 `select`。

```

template<typename E,typename V>
bool select(const bool watch, const E property, string &symbols[], V &data[],
            const bool sort = false) const
{
    E properties[1] = {property};
    V tuples[][1];

    const bool result = select(watch, properties, symbols, tuples, sort);
    ArrayCopy(data, tuples);
    return result;
}

```

通过使用高级筛选器，我们尝试构建一个按分时报价值 `SYMBOL_TRADE_TICK_VALUE` 排序的交易品种列表（参见文档 *SymbolFilterTickValue.mq5*）。假设存款货币为美元，我们应该为以美元报价的外汇金融工具（XXXUSD 类型）获得等于 1.0 的值。对于其他资产，我们将看到非平凡值。

```

#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
    SymbolFilter f;          // filter object
    string symbols[];        // array with symbol names
    double tickValues[];     // array for results

    // apply the filter without conditions, fill and sort the array
    f.select(MarketWatchOnly, SYMBOL_TRADE_TICK_VALUE, symbols, tickValues, true);

    PrintFormat("==== Tick values of the symbols (%d) =====",
                ArraySize(tickValues));
    ArrayPrint(symbols);
    ArrayPrint(tickValues, 5);
}

```

以下是运行脚本的结果。

```

==== Tick values of the symbols (13) =====
"BTCUSD" "USDRUB" "XAUUSD" "USDSEK" "USDCNH" "USDCAD" "USDJPY" "NZDUSD" "AUDUSD" "EUR
0.00100 0.01309 0.10000 0.10955 0.15744 0.80163 0.87319 1.00000 1.00000 1.0

```

6.1.13 允许的交易操作量

在后面的章节中，我们将学习如何进行 EA 交易编程，我们需要控制许多决定成功发送交易指令的交易品种特征。特别是，这适用于交易品种规范中指定允许操作范围的部分。MQL5 中也提供了相应的特性。它们都是 *double* 类型，并且通过 *SymbolInfoDouble* 函数请求。

标识符	说明
SYMBOL_VOLUME_MIN	最小交易量手数
SYMBOL_VOLUME_MAX	最大交易量手数
SYMBOL_VOLUME_STEP	改变交易量的最小步长手数
SYMBOL_VOLUME_LIMIT	某一方向 (买入或卖出) 上未平仓头寸和挂单的最大允许总交易量
SYMBOL_TRADE_CONTRACT_SIZE	交易合约大小 = 1 手

尝试买卖交易量小于最小值、大于最大值或不是一个步长倍数的金融工具将会导致错误。在与交易 API 相关的章节中，我们将实现一个代码来统一必要的检查，并在调用 MQL5 API 交易函数之前使交易量标准化。

此外，MQL 程序还应检查 SYMBOL_VOLUME_LIMIT。例如，限制为 5 手，你可以有一个交易量为 5 手的未平仓买入头寸，并下达一个交易量为 5 手的 *Sell Limit* 挂单。但是，你不能下达挂单 *Buy Limit* (因为一个方向的累计成交量将超过该限制) 或设置超过 5 手的 *Sell Limit*。

作为一个说明性示例，考虑脚本 *SymbolFilterVolumes.mq5*，该脚本记录了所选交易品种的上述特性值。我们将 *MinimalContractSize* 变量添加到输入参数中，以便能够通过 SYMBOL_TRADE_CONTRACT_SIZE 特性筛选交易品种：我们只显示那些合约大小大于指定值的交易品种 (默认情况下为 0，即所有交易品种均满足条件)。

```
#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;
input double MinimalContractSize = 0;

void OnStart()
{
    SymbolFilter f;                // filter object
    string symbols[];              // receiving array with names
    double volumeLimits[][4];      // receiving array with data vectors

    // requested symbol properties
    ENUM_SYMBOL_INFO_DOUBLE volumeIds[] =
    {
        SYMBOL_VOLUME_MIN,
        SYMBOL_VOLUME_STEP,
        SYMBOL_VOLUME_MAX,
        SYMBOL_VOLUME_LIMIT
    };
    ...
}
```

接下来，我们根据合约大小 (应大于指定的大小) 应用筛选器，并获取与匹配交易品种的交易量相关联的规范字段。

```

f.let(SYMBOL_TRADE_CONTRACT_SIZE, MinimalContractSize, SymbolFilter::IS::GREATER)
.select(MarketWatchOnly, volumeIds, symbols, volumeLimits);

const int n = ArraySize(volumeLimits);
PrintFormat("==== Volume limits of the symbols (%d) =====", n);
string title = "";
for(int i = 0; i < ArraySize(volumeIds); ++i)
{
    title += "\t" + EnumToString(volumeIds[i]);
}
Print(title);
for(int i = 0; i < n; ++i)
{
    Print(symbols[i]);
    ArrayPrint(volumeLimits, 3, NULL, i, 1, 0);
}
}

```

对于默认设置，该脚本可能会显示如下结果（带缩写）。

```

==== Volume limits of the symbols (13) =====
SYMBOL_VOLUME_MIN SYMBOL_VOLUME_STEP SYMBOL_VOLUME_MAX SYMBOL_VOLUME_LIMIT
EURUSD
    0.010    0.010 500.000    0.000
GBPUSD
    0.010    0.010 500.000    0.000
USDCHF
    0.010    0.010 500.000    0.000
USDJPY
    0.010    0.010 500.000    0.000
USDCNH
    0.010    0.010 1000.000    0.000
USDRUB
    0.010    0.010 1000.000    0.000
...
XAUUSD
    0.010    0.010 100.000    0.000
BTCUSD
    0.010    0.010 1000.000    0.000
SP500m
    0.100    0.100   5.000 15.000

```

某些交易品种可能不受 SYMBOL_VOLUME_LIMIT 的限制（值为 0）。你可以将结果与交易品种规范进行比较：它们必须匹配。

6.1.14 交易许可

在 [上一节](#)中，我们开始了与正确准备交易订单的相关主题，现在我们将继续学习开发 EA 交易中起非常重要作用的一对特性。

标识符	说明
SYMBOL_TRADE_MODE	交易品种不同交易模式的许可 (参见 ENUM_SYMBOL_TRADE_MODE)
SYMBOL_ORDER_MODE	允许订单类型的标志, 位掩码 (详见下文)

这两个特性均为整数类型, 可通过 *SymbolInfoInteger* 函数获得。

我们已经在脚本 *SymbolPermissions.mq5*中使用了 `SYMBOL_TRADE_MODE` 特性。其值为 `ENUM_SYMBOL_TRADE_MODE` 枚举的元素之一。

标识符	值	说明
SYMBOL_TRADE_MODE_DISABLED	0	该交易品种的交易被禁用
SYMBOL_TRADE_MODE_LONGONLY	1	仅允许买入交易
SYMBOL_TRADE_MODE_SHORTONLY	2	仅允许卖出交易
SYMBOL_TRADE_MODE_CLOSEONLY	3	仅允许平仓操作
SYMBOL_TRADE_MODE_FULL	4	不限制交易操作

注意, *Permissions* 类包含 *isTradeOnSymbolEnabled* 方法, 该方法用于检查影响交易品种交易可用性的几个方面, 其中之一为 `SYMBOL_TRADE_MODE` 特性。默认情况下, 我们认为我们关注的是交易的完全访问权限, 即买入和卖出: `SYMBOL_TRADE_MODE_FULL`。根据交易策略, MQL 程序可能认为只允许买入、只允许卖出或只允许平仓的权限就足够了。

```
static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
    const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
    // checking sessions
    bool found = now == 0;
    ...
    // checking the trading mode for the symbol
    return found && (SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode);
}
```

除了交易模式之外, 我们将来还需要分析不同类型订单的权限: 它们由 `SYMBOL_ORDER_MODE` 特性中单独的位表示, 并且可以与逻辑 OR (|) 任意组合。例如, 值 127 (0x7F) 对应于所有位的设置, 即所有订单类型的可用性。

标识符	值	说明
<code>SYMBOL_ORDER_MARKET</code>	1	允许市场订单 (买入和卖出)
<code>SYMBOL_ORDER_LIMIT</code>	2	允许限制订单 (买入限制和卖出限制)
<code>SYMBOL_ORDER_STOP</code>	4	允许止损订单 (买入止损和卖出止损)
<code>SYMBOL_ORDER_STOP_LIMIT</code>	8	允许止损限制订单 (买入止损限制和卖出止损限制)
<code>SYMBOL_ORDER_SL</code>	16	允许设置止损水平
<code>SYMBOL_ORDER_TP</code>	32	允许设置止盈水平
<code>SYMBOL_ORDER_CLOSEBY</code>	64	允许对同一交易品种通过相反的头寸进行平仓, 即“对冲平仓”操作

`SYMBOL_ORDER_CLOSEBY` 特性仅针对具有对冲会计的账户设置 `ACCOUNT_MARGIN_MODE_RETAIL_HEDGING`, 参见 [账户类型](#))。

在测试脚本 *SymbolFilterTradeMode.mq5* 中, 我们将为 *Market Watch* 中可见的交易品种请求几个说明的特性。位及其组合作为数字的输出信息不多, 所以我们将利用这样一个事实, 即在 *SymbolMonitor* 类中, 我们有一个方便的方法 *stringify* 来打印所有特性的枚举元素和位掩码。

```

void OnStart()
{
    SymbolFilter f;                // filter object
    string symbols[];              // array for names
    long permissions[][2];         // array for data (property values)

    // list of requested symbol properties
    ENUM_SYMBOL_INFO_INTEGER modes[] =
    {
        SYMBOL_TRADE_MODE,
        SYMBOL_ORDER_MODE
    };

    // apply the filter, get arrays with results
    f.let(SYMBOL_VISIBLE, true).select(true, modes, symbols, permissions);

    const int n = ArraySize(symbols);
    PrintFormat("==== Trade permissions for the symbols (%d) =====", n);
    for(int i = 0; i < n; ++i)
    {
        Print(symbols[i] + ":");
        for(int j = 0; j < ArraySize(modes); ++j)
        {
            // display bit and number descriptions "as is"
            PrintFormat("  %s (%d)",
                SymbolMonitor::stringify(permissions[i][j], modes[j]),
                permissions[i][j]);
        }
    }
}

```

以下是运行该脚本产生的部分日志。

```

===== Trade permissions for the symbols (13) =====
EURUSD:
    SYMBOL_TRADE_MODE_FULL (4)
    [ _SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP
      _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP
      _SYMBOL_ORDER_CLOSEBY ] (127)
GBPUSD:
    SYMBOL_TRADE_MODE_FULL (4)
    [ _SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP
      _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP
      _SYMBOL_ORDER_CLOSEBY ] (127)
...
SP500m:
    SYMBOL_TRADE_MODE_DISABLED (0)
    [ _SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP
      _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP ] (63)

```

请注意，最后一个交易品种 SP500m 的交易被完全禁用（其报价仅作为“指示性”提供）。同时，其订单类型标志集不为 0，但没有任何区别。

根据市场中的事件，经纪商可以自行决定是否改变交易品种特性，例如，只留下一段时间的平仓机会，因此正确的交易机器人必须在每次操作之前控制这些特性。

6.1.15 交易品种交易条件及订单执行模式

在本节中，我们将深入探讨依赖于金融工具设置的交易自动化的各个方面。现在，我们将只研究这些特性，而其实际应用将在后面的章节中介绍。假设读者已经熟悉市场和挂单、交易和头寸等基本术语。

当发送交易请求供执行时，应考虑到在金融市场中，无法保证在特定时刻，该金融工具能够以期望的价格获得全部请求的交易量。因此，实时交易是由价格和交易量执行模式共同调控。模式，或者称为执行策略，定义了当价格已经改变或者当前不能完全执行所请求的交易量时的规则。

在 MQL5 API 中，这些模式可用于每个交易品种，如以下特性，这些特性可通过 `SymbolInfoInteger` 函数获得。

标识符	说明
SYMBOL_TRADE_EXEMODE	与价格相关的交易执行模式
SYMBOL_FILLING_MODE	与交易量相关的允许订单填充模式标志（位掩码，详见下文）

SYMBOL_TRADE_EXEMODE 特性的值为 ENUM_SYMBOL_TRADE_EXECUTION 枚举的元素。

标识符	说明
SYMBOL_TRADE_EXECUTION_REQUEST	以请求价格进行交易
SYMBOL_TRADE_EXECUTION_INSTANT	即时执行（以流式价格进行交易）
SYMBOL_TRADE_EXECUTION_MARKET	市场执行
SYMBOL_TRADE_EXECUTION_EXCHANGE	交易所执行

终端用户应从 **New order** 对话框 (F9) 的下拉菜单 **Type** 中看到所有或大部分这些模式。让我们简单回忆一下它们的含义。有关详细信息，请参考终端文档。

- 按请求执行 (SYMBOL_TRADE_EXECUTION_REQUEST) - 以先前从经纪商处收到的价格执行市价订单。在发送市价订单之前，交易者向经纪商请求当前价格。以此价格进一步执行订单可以被确认或拒绝。
- 即时执行 (SYMBOL_TRADE_EXECUTION_INSTANT) - 以当前价格执行市价订单。当发送交易请求执行时，终端自动将当前价格插入订单。如果经纪商接受该价格，订单就被执行。如果经纪商不接受请求的价格，经纪商返回该订单可以执行的价格，称为重新报价。
- 市场执行 (SYMBOL_TRADE_EXECUTION_MARKET) - 经纪商将执行价格插入订单，无需交易者的额外确认。在这种模式下发送市价订单意味着对执行价格的早期约定。
- 交易所执行 (SYMBOL_TRADE_EXECUTION_EXCHANGE) - 交易操作以当前市场报价执行。

至于 SYMBOL_FILLING_MODE 中可以与逻辑运算符 OR (|) 组合的位，其存在或不存在表示以下操作。

标识符	值	成交策略
SYMBOL_FILLING_FOK	1	填充或删除 (FOK); 订单必须只在指定的交易量内执行或取消
SYMBOL_FILLING_IOC	2	立即执行或取消 (IOC); 在订单规定的限额内交易市场上的最大交易量或取消
(标识符缺失)	(任意, 包括 0)	返回; 在部分执行的情况下, 剩余交易量的市价订单或限价订单不会被取消, 而是保持有效

使用 FOK 和 IOC 模式的可能性由交易服务器决定。

如果启用了 SYMBOL_FILLING_FOK 模式, 那么在使用 *OrderSend* 函数发送订单时, MQL 程序将能够使用相关订单填充类型, 其位于 *MqlTradeRequest* 结构体中: ORDER_FILLING_FOK。同时, 如果市场上没有足够的金融工具交易量, 订单将不会被执行。应考虑到, 所需的交易量可以由市场上现有的几个报价组成, 从而导致几笔交易。

如果启用了 SYMBOL_FILLING_IOC 模式, MQL 程序将可以访问同名的 ORDER_FILLING_IOC 订单填充方法 (在将订单发送给 *OrderSend* 函数之前, 也会在 *MqlTradeRequest* 结构中的特殊 “填充” 字段 (*type_filling*) 中指定)。使用此模式时, 如果无法完全执行, 订单将在可用交易量上执行, 其剩余订单交易量将被取消。

没有标识符的最新策略为默认模式, 并且无论其他模式如何均可用 (这就是其能够匹配 0 或任何其他值的原因)。换言之, 对于 SYMBOL_FILLING_MODE 特性, 即使我们得到值 1 (SYMBOL_FILLING_FOK)、2 (SYMBOL_FILLING_IOC) 或 3 (SYMBOL_FILLING_FOK | SYMBOL_FILLING_IOC), 均会隐含返回模式。要使用此策略, 在形成订单 (填充 *MqlTradeRequest* 结构) 时, 我们应指定填充类型 ORDER_FILLING_RETURN。

在所有的 SYMBOL_TRADE_EXEMODE 模式中, 有一个关于市场执行 (SYMBOL_TRADE_EXECUTION_MARKET) 的特殊性: 在市场执行模式中, 始终禁止返回订单。

因为 ORDER_FILLING_FOK 对应于常量 0, 所以在交易请求中缺少对填充类型的明确指示即意味着这种特定模式。

在开发 EA 交易时, 我们会在实践中考虑所有这些细微差别, 但是现在, 让我们在一个简单脚本 *SymbolFilterExecMode.mq5* 中检查特性的读取情况。

```

#include <MQL5Book/SymbolFilter.mqh>

void OnStart()
{
    SymbolFilter f;                                // filter object
    string symbols[];                              // array of symbol names
    long permissions[][2];                         // array with property value vectors

    // properties to read
    ENUM_SYMBOL_INFO_INTEGER modes[] =
    {
        SYMBOL_TRADE_EXEMODE,
        SYMBOL_FILLING_MODE
    };
    // apply filter - fill arrays
    f.select(true, modes, symbols, permissions);

    const int n = ArraySize(symbols);
    PrintFormat("==== Trade execution and filling modes for the symbols (%d) =====",
        for(int i = 0; i < n; ++i)
        {
            Print(symbols[i] + ":");
            for(int j = 0; j < ArraySize(modes); ++j)
            {
                // output properties as descriptions and numbers
                PrintFormat("  %s (%d)",
                    SymbolMonitor::stringify(permissions[i][j], modes[j]),
                    permissions[i][j]);
            }
        }
    }
}

```

以下是包含脚本结果的日志片段。除了最后一个 SP500m (SYMBOL_TRADE_EXECUTION_MARKET)，此处几乎所有交易品种都有按价格 (SYMBOL_TRADE_EXECUTION_INSTANT) 立即执行模式。此处，我们可以找到各种交易量填充模式，包括单独的 SYMBOL_FILLING_FOK、SYMBOL_FILLING_IOC 及其组合。只有 BTCUSD 指定了 SYMBOL_FILLING_RETURN，即收到值 0 (没有 FOK 和 IOC 位)。

```

===== Trade execution and filling modes for the symbols (13) =====
EURUSD:
    SYMBOL_TRADE_EXECUTION_INSTANT (1)
    [ _SYMBOL_FILLING_FOK ] (1)
GBPUSD:
    SYMBOL_TRADE_EXECUTION_INSTANT (1)
    [ _SYMBOL_FILLING_FOK ] (1)
...
USDCNH:
    SYMBOL_TRADE_EXECUTION_INSTANT (1)
    [ _SYMBOL_FILLING_FOK _SYMBOL_FILLING_IOC ] (3)
USDRUB:
    SYMBOL_TRADE_EXECUTION_INSTANT (1)
    [ _SYMBOL_FILLING_IOC ] (2)
AUDUSD:
    SYMBOL_TRADE_EXECUTION_INSTANT (1)
    [ _SYMBOL_FILLING_FOK ] (1)
NZDUSD:
    SYMBOL_TRADE_EXECUTION_INSTANT (1)
    [ _SYMBOL_FILLING_FOK _SYMBOL_FILLING_IOC ] (3)
...
XAUUSD:
    SYMBOL_TRADE_EXECUTION_INSTANT (1)
    [ _SYMBOL_FILLING_FOK _SYMBOL_FILLING_IOC ] (3)
BTCUSD:
    SYMBOL_TRADE_EXECUTION_INSTANT (1)
    [ (_SYMBOL_FILLING_RETURN) ] (0)
SP500m:
    SYMBOL_TRADE_EXECUTION_MARKET (2)
    [ _SYMBOL_FILLING_FOK ] (1)

```

注意，填充模式标识符中出现下划线是因为我们必须用具有常量值的元素来定义自己的枚举 **SYMBOL_FILLING** (*SymbolMonitor.mqh*)。执行此操作是因为 MQL5 没有此类内置枚举，但同时，我们不能将枚举的元素命名为内置常量，因为会导致名称冲突。

6.1.16 保证金要求

对于交易者而言，关于金融工具最重要的信息之一是开仓所需的资金量。如果不知道买入或卖出给定数量的手数需要多少钱，就不可能在 EA 交易中实现资金管理系统并控制账户余额。

由于 MetaTrader 5 用于交易各种工具（货币、商品、股票、债券、期权和期货），因此保证金计算原则有很大不同。本文档提供了详细信息，特别是 [外汇和期货](#) 以及 [交易所](#)。

MQL5 API 的几个特性允许你定义市场类型和特定金融工具的保证金计算方法。

未来，假设对于给定的参数组合，如交易运算类型、工具、交易量和价格，MQL5 允许你使用 [OrderCalcMargin](#) 函数计算保证金。这是最简单的方法，但它有一个明显的局限性：该函数不考虑当前的持仓和挂单。特别是，当账户上允许相反头寸时，会忽略对重叠交易量的可能调整。

因此，为了获得当前用作持仓和订单保证金的账户资金明细，MQL 程序可能需要使用公式分析以下特性和计算。此外，禁止在指标中使用 [OrderCalcMargin](#) 函数。你可以使用 [OrderCheck](#) 预先估算拟定交易完成后可用保证金。

标识符	说明
SYMBOL_TRADE_CALC_MODE	计算保证金和利润的方法 (参见 <code>ENUM_SYMBOL_CALC_MODE</code>)
SYMBOL_MARGIN_HEDGED_USE_LEG	启用 (<code>true</code>) 或禁用 (<code>false</code>) 最大重叠头寸 (买入和卖出) 的对冲保证金计算模式的布尔标志
SYMBOL_MARGIN_INITIAL	交易所工具的初始保证金
SYMBOL_MARGIN_MAINTENANCE	交易所金融工具的维持保证金
SYMBOL_MARGIN_HEDGED	一手已覆盖的合约规模或保证金 (一个交易品种的相反头寸) 头寸

前两个特性包含在 `ENUM_SYMBOL_INFO_INTEGER` 枚举中, 后三个特性包含在 `ENUM_SYMBOL_INFO_DOUBLE` 中, 它们可以分别通过函数 `SymbolInfoInteger` 和 `SymbolInfoDouble` 读取。

具体的保证金计算公式取决于 `SYMBOL_TRADE_CALC_MODE` 特性, 如下表所示。更完整的信息可以在 [MQL5 文档](#) 中找到。

请注意, 初始保证金和维持保证金不用于外汇金融工具, 这些特性对它们而言始终为 0。

初始保证金表示必需的 [保证金货币](#) 存款金额, 用于开立 [一手](#) 交易量。初始保证金用于在入市前检查客户资金是否充足。要根据订单类型和方向获得收取的最终保证金金额, 可使用 `SymbolInfoMarginRate` 函数计算保证金比率。因此, 经纪商可以为每种金融工具设置单独的杠杆或折扣。

维持保证金表示金融工具保证金货币中维持一手持仓的最低资金值。用于在账户状态 (交易条件) 发生变化时检查客户资金是否充足。如果资金水平低于所有持仓的维持保证金金额, 经纪商将开始强制平仓。

如果维持保证金特性为 0, 则使用初始保证金。与初始保证金的情况一样, 要获得根据类型和方向收取的最终保证金金额, 应使用 `SymbolInfoMarginRate` 函数检查保证金费率。

对冲头寸, 即同一交易品种的多向头寸, 只能存在于 [对冲](#) 交易账户。显然, 对冲保证金的计算以及 `SYMBOL_MARGIN_HEDGED_USE_LEG` 和 `SYMBOL_MARGIN_HEDGED` 的特性只有在此类账户上才有意义。对冲保证金适用于覆盖的交易量。

经纪商可以为每种工具选择两种现有方法中的一种来计算覆盖头寸的保证金:

- ① 当禁用最长边计算模式时, 即 `SYMBOL_MARGIN_HEDGED_USE_LEG` 特性等于 `false` 时, 应用基础计算。在这种情况下, 保证金由三个部分组成: 现有头寸未覆盖量的保证金, 覆盖量的保证金 (如果有相反头寸并且 `SYMBOL_MARGIN_HEDGED` 特性不为零) 以及挂单的保证金。如果为金融工具设置了初始保证金 (`SYMBOL_MARGIN_INITIAL` 特性不为零), 则对冲保证金被指定为绝对值 (以货币为单位)。如果初始保证金未设置 (等于 0), 则 `SYMBOL_MARGIN_HEDGED` 指定根据交易金融工具类型 (`SYMBOL_TRADE_CALC_MODE`) 对应的公式计算保证金时应使用的合约规模。
- ② 当 `SYMBOL_MARGIN_HEDGED_USE_LEG` 特性等于 `true` 时, 应用最高头寸计算。在这种情况下, `SYMBOL_MARGIN_HEDGED` 的值被忽略。取而代之的是, 计算金融工具上所有短仓和长仓的交易量, 并计算每一方的加权平均开盘价。此外, 使用对应于金融工具类型

(SYMBOL_TRADE_CALC_MODE) 的公式，计算空头方和多头方的保证金。将最大值用作最终值。

下表列出了 ENUM_SYMBOL_CALC_MODE 元素及其各自的保证金计算方法。同样的特性 (SYMBOL_TRADE_CALC_MODE) 也负责计算头寸的盈亏，但是我们将在后面的 MQL5 交易函数一章中考虑这个方面。

标识符	公式
SYMBOL_CALC_MODE_FOREX <i>Forex</i>	$\text{Lots} * \text{ContractSize} * \text{MarginRate} / \text{Leverage}$
SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE <i>Forex without leverage</i>	$\text{Lots} * \text{ContractSize} * \text{MarginRate}$
SYMBOL_CALC_MODE_CFD <i>CFD</i>	$\text{Lots} * \text{ContractSize} * \text{MarketPrice} * \text{MarginRate}$
SYMBOL_CALC_MODE_CFDLEVERAGE <i>CFD with leverage</i>	$\text{Lots} * \text{ContractSize} * \text{MarketPrice} * \text{MarginRate} / \text{Leverage}$
SYMBOL_CALC_MODE_CFDINDEX <i>CFDs on indices</i>	$\text{Lots} * \text{ContractSize} * \text{MarketPrice} * \text{TickPrice} / \text{TickSize} * \text{MarginRate}$
SYMBOL_CALC_MODE_EXCH_STOCKS <i>Securities on the stock exchange</i>	$\text{Lots} * \text{ContractSize} * \text{LastPrice} * \text{MarginRate}$
SYMBOL_CALC_MODE_EXCH_STOCKS_MOEX <i>Securities on MOEX</i>	$\text{Lots} * \text{ContractSize} * \text{LastPrice} * \text{MarginRate}$
SYMBOL_CALC_MODE_FUTURES <i>Futures</i>	$\text{Lots} * \text{InitialMargin} * \text{MarginRate}$
SYMBOL_CALC_MODE_EXCH_FUTURES <i>Futures on the stock exchange</i>	$\text{Lots} * \text{InitialMargin} * \text{MarginRate}$ or $\text{Lots} * \text{MaintenanceMargin} * \text{MarginRate}$
SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS <i>Futures on FORTS</i>	$\text{Lots} * \text{InitialMargin} * \text{MarginRate}$ or $\text{Lots} * \text{MaintenanceMargin} * \text{MarginRate}$
SYMBOL_CALC_MODE_EXCH_BONDS <i>Bonds on the stock exchange</i>	$\text{Lots} * \text{ContractSize} * \text{FaceValue} * \text{OpenPrice} / 100$
SYMBOL_CALC_MODE_EXCH_BONDS_MOEX <i>Bonds on MOEX</i>	$\text{Lots} * \text{ContractSize} * \text{FaceValue} * \text{OpenPrice} / 100$
SYMBOL_CALC_MODE_SERV_COLLATERAL	非交易资产 (保证金不适用)

上述公式中使用了以下符号：

- ⌚ Lots - 手数中的头寸或订单量 (合约份额)
- ⌚ ContractSize - 合约规模 (一手, [SYMBOL_TRADE_CONTRACT_SIZE](#))
- ⌚ Leverage - 交易账户杠杆 ([ACCOUNT_LEVERAGE](#))
- ⌚ InitialMargin - 初始保证金 (SYMBOL_MARGIN_INITIAL)

- ① `MaintenanceMargin` - 维持保证金 (`SYMBOL_MARGIN_MAINTENANCE`)
- ① `TickPrice` - 分时报价价格 (`SYMBOL_TRADE_TICK_VALUE`)
- ① `TickSize` - 分时报价规模 (`SYMBOL_TRADE_TICK_SIZE`)
- ① `MarketPrice` - 最新已知的 *买入/卖出* 价格, 取决于交易类型
- ① `LastPrice` - 最后已知的 *最新* 价格
- ① `OpenPrice` - 头寸或订单开盘价的加权平均价格
- ① `FaceValue` - 债券的面值
- ① `MarginRate` - 根据 *SymbolInfoMarginRate* 函数的保证金比率, 也可以有 2 个不同的值: 初始保证金和维持保证金

MarginProfitMeter.mqh 文档给出了公式计算大多数类型交易品种的另一实现方式 (参见 *估算交易运算的利润率*)。该实现方式也可以用在指标中。

让我们对一些模式做一些注释。

上表中, 只有三种期货公式使用了初始保证金 (`SYMBOL_MARGIN_INITIAL`)。但是, 如果该特性在任何其他交易品种的规范中具有非零值, 则其可以确定保证金。

一些交易所可能会对保证金调整施加自己的具体要求, 例如 `FORSTS` (`SYMBOL_CALC_MODE_EXCH_FUTURES_FORSTS`) 的折扣系统。有关详细信息, 请参见 *ML5* 文档并咨询你的经纪商。

在 `SYMBOL_CALC_MODE_SERV_COLLATERAL` 模式中, 金融工具的价值被计入资产, 并被纳入净值中。因此, 此类金融工具的持仓增加了可用保证金金额, 并作为交易金融工具持仓的额外抵押。根据交易量、合约规模、当前市场价格和流动性比率计算持仓的市场价值: *Lots * ContractSize * MarketPrice * LiquidityRate* (后者的值可以作为 `SYMBOL_TRADE_LIQUIDITY_RATE` 特性获得)

作为使用保证金相关特性的示例, 请考虑脚本 *SymbolFilterMarginStats.mq5*。其作用是计算所选交易品种列表中保证金计算方法的统计数据, 并选择性地记录每个交易品种的这些特性。我们将使用已知的筛选类 *SymbolFilter* 以及通过输入变量为其提供的条件来选择用于分析的交易品种。

```
#include <ML5Book/SymbolFilter.mqh>

input bool UseMarketWatch = false;
input bool ShowPerSymbolDetails = false;
input bool ExcludeZeroInitMargin = false;
input bool ExcludeZeroMainMargin = false;
input bool ExcludeZeroHedgeMargin = false;
```

默认情况下, 会请求所有可用交易品种的信息。为了将上下文仅限于市场概要, 我们应将 *UseMarketWatch* 设置为 *true*。

参数 *ShowPerSymbolDetails* 允许你输出每个交易品种的详细信息 (默认情况下, 该参数为 *false*, 且仅显示统计数据)。

最后三个参数用于根据零保证金值的条件 (分别为初始、维持和对冲) 筛选交易品种。

为了在日志中收集并方便地显示每个交易品种的完整特性集 (当 *ShowPerSymbolDetails* 打开时), 在代码中定义了结构体 *MarginSettings*。

```

struct MarginSettings
{
    string name;
    ENUM_SYMBOL_CALC_MODE calcMode;
    bool hedgeLeg;
    double initial;
    double maintenance;
    double hedged;
};

```

由于一些特性为整数 (SYMBOL_TRADE_CALC_MODE 和 SYMBOL_MARGIN_HEDGED_USE_LEG), 而一些特性为实数 (SYMBOL_MARGIN_INITIAL、SYMBOL_MARGIN_MAINTENANCE 和 SYMBOL_MARGIN_HEDGED), 这些特性必须通过筛选对象单独请求。

现在让我们直接进入 *OnStart* 中的工作代码。此处, 像往常一样, 我们定义筛选对象 (*f*)、字符名称的输出数组 (*symbols*) 以及请求特性的值 (*flags, values*)。除此之外, 我们还添加了一个结构体数组 *MarginSettings*。

```

void OnStart()
{
    SymbolFilter f;           // filter object
    string symbols[];         // array for names
    long flags[][2];          // array for integer vectors
    double values[][3];        // array for real vectors
    MarginSettings margins[];  // composite output array
    ...
}

```

引入了 *stats* 数组映射来计算统计数据, 使用 ENUM_SYMBOL_CALC_MODE 这样的关键字和 *int* 整数值来计算遇到每种方法的次数。此外, 零保证金的所有情况和长边启用的计算模式应记录在相应的计数器变量中。

```

MapArray<ENUM_SYMBOL_CALC_MODE,int> stats; // counters for each method/mode
int hedgeLeg = 0;                          // and other counters
int zeroInit = 0;                          // ...
int zeroMaintenance = 0;
int zeroHedged = 0;
...

```

接下来, 我们指定与保证金相关的重要特性, 这些特性将从交易品种设置中读取。首先是 *ints* 数组中的整数, 然后是 *doubles* 数组中的实数。

```

ENUM_SYMBOL_INFO_INTEGER ints[] =
{
    SYMBOL_TRADE_CALC_MODE,
    SYMBOL_MARGIN_HEDGED_USE_LEG
};

ENUM_SYMBOL_INFO_DOUBLE doubles[] =
{
    SYMBOL_MARGIN_INITIAL,
    SYMBOL_MARGIN_MAINTENANCE,
    SYMBOL_MARGIN_HEDGED
};
...

```

根据输入参数，我们可设置筛选条件。

```

if(ExcludeZeroInitMargin) f.let(SYMBOL_MARGIN_INITIAL, 0, SymbolFilter::IS::GREATE
if(ExcludeZeroMainMargin) f.let(SYMBOL_MARGIN_MAINTENANCE, 0, SymbolFilter::IS::GR
if(ExcludeZeroHedgeMargin) f.let(SYMBOL_MARGIN_HEDGED, 0, SymbolFilter::IS::GREATE
...

```

现在一切都准备好了，可以按条件选择交易品种，并将其特性放入数组。我们执行此操作两次，分别针对整数和实数特性。

```

f.select(UseMarketWatch, ints, symbols, flags);
const int n = ArraySize(symbols);
ArrayResize(symbols, 0, n);
f.select(UseMarketWatch, doubles, symbols, values);
...

```

在第一次应用筛选器之后，必须将带有交易品种的数组清零，这样名称才不会重复。尽管有两次独立的查询，但所有输出数组 (*ints* 和 *doubles*) 中元素的顺序是相同的，因为筛选条件没有改变。

如果用户启用了详细日志，我们会为结构体的 *margins* 数组分配内存。

```

if(ShowPerSymbolDetails) ArrayResize(margins, n);

```

最后，我们通过迭代结果数组的所有元素来计算统计数据，并可选地填充结构数组。


```

for(int i = 0; i < n; ++i)
{
    stats.inc((ENUM_SYMBOL_CALC_MODE)flags[i].value[0]);
    hedgeLeg += (int)flags[i].value[1];
    if(values[i].value[0] == 0) zeroInit++;
    if(values[i].value[1] == 0) zeroMaintenance++;
    if(values[i].value[2] == 0) zeroHedged++;

    if(ShowPerSymbolDetails)
    {
        margins[i].name = symbols[i];
        margins[i].calcMode = (ENUM_SYMBOL_CALC_MODE)flags[i][0];
        margins[i].hedgeLeg = (bool)flags[i][1];
        margins[i].initial = values[i][0];
        margins[i].maintenance = values[i][1];
        margins[i].hedged = values[i][2];
    }
}
...

```

现在，我们可以在日志中显示统计数据。

```

PrintFormat("==== Margin calculation modes for %s symbols %s====",
    (UseMarketWatch ? "Market Watch" : "all available"),
    (ExcludeZeroInitMargin || ExcludeZeroMainMargin || ExcludeZeroHedgeMargin
        ? "(with conditions) " : ""));
PrintFormat("Total symbols: %d", n);
PrintFormat("Hedge leg used in: %d", hedgeLeg);
PrintFormat("Zero margin counts: initial=%d, maintenance=%d, hedged=%d",
    zeroInit, zeroMaintenance, zeroHedged);

Print("Stats per calculation mode:");
stats.print();
...

```

由于 `ENUM_SYMBOL_CALC_MODE` 枚举的元素显示为整数（信息不多），我们还显示了一个文本，其中每个值都有一个名称（来自 `EnumToString`）。

```

Print("Legend: key=calculation mode, value=count");
for(int i = 0; i < stats.getSize(); ++i)
{
    PrintFormat("%d -> %s", stats.getKey(i), EnumToString(stats.getKey(i)));
}
...

```

如果需要所选字符的详细信息，我们可以输出结构体数组 `margins`。

```

if(ShowPerSymbolDetails)
{
    Print("Settings per symbol:");
    ArrayPrint(margins);
}
}

```

我们用不同的设置运行这个脚本几次。从默认设置开始。

```

===== Margin calculation modes for all available symbols =====
Total symbols: 131
Hedge leg used in: 14
Zero margin counts: initial=123, maintenance=130, hedged=32
Stats per calculation mode:
    [key] [value]
[0]      0      101
[1]      4       16
[2]      1        1
[3]      2       11
[4]      5        2
Legend: key=calculation mode, value=count
0 -> SYMBOL_CALC_MODE_FOREX
4 -> SYMBOL_CALC_MODE_CFDLEVERAGE
1 -> SYMBOL_CALC_MODE_FUTURES
2 -> SYMBOL_CALC_MODE_CFD
5 -> SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE

```

第二次运行时，我们将 *ShowPerSymbolDetails* 和 *ExcludeZeroInitMargin* 设置为 *true*。这需要关于初始保证金值不为零的所有交易品种的详细信息。

```

===== Margin calculation modes for all available symbols (with conditions) =====
Total symbols: 8
Hedge leg used in: 0
Zero margin counts: initial=0, maintenance=7, hedged=0
Stats per calculation mode:
    [key] [value]
[0]      0        5
[1]      1         1
[2]      5         2
Legend: key=calculation mode, value=count
0 -> SYMBOL_CALC_MODE_FOREX
1 -> SYMBOL_CALC_MODE_FUTURES
5 -> SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE
Settings per symbol:
    [name] [calcMode] [hedgeLeg]    [initial] [maintenance]    [hedged]
[0] "XAUEUR"          0      false    100.00000    0.00000    50.00000
[1] "XAUAUD"          0      false    100.00000    0.00000   100.00000
[2] "XAGEUR"          0      false   1000.00000    0.00000  1000.00000
[3] "USDGEL"          0      false 100000.00000 100000.00000 50000.00000
[4] "SP500m"          1      false    6600.00000    0.00000   6600.00000
[5] "XBRUSD"          5      false    100.00000    0.00000    50.00000
[6] "XNGUSD"          0      false   10000.00000    0.00000 10000.00000
[7] "XTIUSD"          5      false    100.00000    0.00000    50.00000

```

6.1.17 挂单到期规则

当处理挂单时 (包括 *Stop Loss* 和 *Take Profit* 水平), MQL 程序应检查几个定义过期规则的特性。这两个特性都可以作为 `ENUM_SYMBOL_INFO_INTEGER` 枚举的元素, 用于调用 *SymbolInfoInteger* 函数。

标识符	说明
<code>SYMBOL_EXPIRATION_MODE</code>	允许订单到期模式的标志 (位掩码)
<code>SYMBOL_ORDER_GTC_MODE</code>	有效期由 <code>ENUM_SYMBOL_ORDER_GTC_MODE</code> 枚举的一个元素定义

仅当 `SYMBOL_EXPIRATION_MODE` 包含 `SYMBOL_EXPIRATION_GTC` 时, 才考虑 `SYMBOL_ORDER_GTC_MODE` 特性 (详见下文)。GTC为 Good TillCanceled (撤销前有效)的缩写。

对于每种金融工具, `SYMBOL_EXPIRATION_MODE` 特性都可以指定挂单的几种有效 (到期)模式。每种模式都有一个与之相关的标志 (位)。

标识符 (值)	说明
<code>SYMBOL_EXPIRATION_GTC (1)</code>	根据 <code>ENUM_SYMBOL_ORDER_GTC_MODE</code> 特性, 订单有效
<code>SYMBOL_EXPIRATION_DAY (2)</code>	订单在当天结束前有效
<code>SYMBOL_EXPIRATION_SPECIFIED (4)</code>	到期日期和时间在订单中指定
<code>SYMBOL_EXPIRATION_SPECIFIED_DAY (8)</code>	到期日期在订单中指定

这些标志可以与逻辑 `OR` (`|`) 运算结合使用, 例如, `SYMBOL_EXPIRATION_GTC | SYMBOL_EXPIRATION_SPECIFIED`, 相当于 `1 | 4`, 即数字 `5`。要检查工具是否启用了特定模式, 请对函数结果和所需的模式位执行逻辑 `AND` (`&`) 运算: 非零值表示该模式可用。

在 `SYMBOL_EXPIRATION_SPECIFIED_DAY` 的情况下, 订单到期日期至指定日期的 `23:59:59`。如果该时间不在交易时段, 则到期时间将发生在最近的下一个交易时间。

`ENUM_SYMBOL_ORDER_GTC_MODE` 枚举包含以下元素。

标识符	说明
<code>SYMBOL_ORDERS_GTC</code>	挂单和止损/止盈水平无限期有效, 除非被明确取消
<code>SYMBOL_ORDERS_DAILY</code>	订单仅在一个交易日内有效: 交易完成后, 所有挂单以及止损和止盈均将被删除
<code>SYMBOL_ORDERS_DAILY_EXCLUDING_STOPS</code>	当更改交易日时, 仅挂单被删除, 但止损和止盈水平均被保存

根据 `SYMBOL_EXPIRATION_MODE` 特性中的设置位，当准备发送订单时，MQL 程序可以选择与这些位对应的模式之一。从技术上讲，这是通过填充特殊结构体 `MqlTradeRequest` 中的 `type_time` 字段来实现，该操作在调用 `OrderSend` 函数之前进行。字段值必须是 `ENUM_ORDER_TYPE_TIME` 枚举的一个元素（见 [挂单到期日期](#)）：后面我们会看到，其与上面的一组标志有共同之处，就是每个标志都按照顺序设置对应的模式：`ORDER_TIME_GTC`、`ORDER_TIME_DAY`、`ORDER_TIME_SPECIFIED` 和 `ORDER_TIME_SPECIFIED_DAY`。到期时间或日期本身必须在同一结构体的另一个字段中指定。

脚本 `SymbolFilterExpiration.mqh` 允许你找出可用交易品种中每个标志使用情况的统计数据（在市场概览或一般情况下，取决于输入参数 `UseMarketWatch`）。`ShowPerSymbolDetails` 中的第二个参数被设置为 `true`，将导致记录每个字符的所有标志，所以必须小心：如果模式 `UseMarketWatch` 也等于 `false`，将会生成大量的日志条目。

```
#property script_show_inputs

#include <MQL5Book/SymbolFilter.mqh>

input bool UseMarketWatch = false;
input bool ShowPerSymbolDetails = false;
```

在 `OnStart` 函数中，除了筛选对象和接收交易品种名称和特性值的数组之外，我们还介绍了 `MapArray`，用于分别计算 `SYMBOL_EXPIRATION_MODE` 和 `SYMBOL_ORDER_GTC_MODE` 特性的统计信息。

```
void OnStart()
{
    SymbolFilter f;                // filter object
    string symbols[];             // receiving array for symbol names
    long flags[][2];              // receiving array for property values

    MapArray<SYMBOL_EXPIRATION,int> stats;        // mode counters
    MapArray<ENUM_SYMBOL_ORDER_GTC_MODE,int> gtc; // GTC counters

    ENUM_SYMBOL_INFO_INTEGER ints[] =
    {
        SYMBOL_EXPIRATION_MODE,
        SYMBOL_ORDER_GTC_MODE
    };
    ...
}
```

接下来，应用筛选器并计算统计数据。

```

f.select(UseMarketWatch, ints, symbols, flags);
const int n = ArraySize(symbols);

for(int i = 0; i < n; ++i)
{
    if(ShowPerSymbolDetails)
    {
        Print(symbols[i] + ":");
        for(int j = 0; j < ArraySize(ints); ++j)
        {
            // properties in the form of descriptions and numbers
            PrintFormat("  %s (%d)",
                SymbolMonitor::stringify(flags[i][j], ints[j]),
                flags[i][j]);
        }
    }

    const SYMBOL_EXPIRATION mode = (SYMBOL_EXPIRATION)flags[i][0];
    for(int j = 0; j < 4; ++j)
    {
        const SYMBOL_EXPIRATION bit = (SYMBOL_EXPIRATION)(1 << j);
        if((mode & bit) != 0)
        {
            stats.inc(bit);

            if(bit == SYMBOL_EXPIRATION_GTC)
            {
                gtc.inc((ENUM_SYMBOL_ORDER_GTC_MODE)flags[i][1]);
            }
        }
    }
}
...

```

最后，我们将收到的数字输出到日志中。

```

PrintFormat("==== Expiration modes for %s symbols ====",
    (UseMarketWatch ? "Market Watch" : "all available"));
PrintFormat("Total symbols: %d", n);

Print("Stats per expiration mode:");
stats.print();
Print("Legend: key=expiration mode, value=count");
for(int i = 0; i < stats.getSize(); ++i)
{
    PrintFormat("%d -> %s", stats.getKey(i), EnumToString(stats.getKey(i)));
}
Print("Stats per GTC mode:");
gtc.print();
Print("Legend: key=GTC mode, value=count");
for(int i = 0; i < gtc.getSize(); ++i)
{
    PrintFormat("%d -> %s", gtc.getKey(i), EnumToString(gtc.getKey(i)));
}
}

```

让我们运行脚本两次。第一次使用默认设置，我们可以得到类似下图的结果。

```

==== Expiration modes for all available symbols ====
Total symbols: 52357
Stats per expiration mode:
    [key] [value]
[0]     1   52357
[1]     2   52357
[2]     4   52357
[3]     8   52303
Legend: key=expiration mode, value=count
1 -> _SYMBOL_EXPIRATION_GTC
2 -> _SYMBOL_EXPIRATION_DAY
4 -> _SYMBOL_EXPIRATION_SPECIFIED
8 -> _SYMBOL_EXPIRATION_SPECIFIED_DAY
Stats per GTC mode:
    [key] [value]
[0]     0   52357
Legend: key=GTC mode, value=count
0 -> SYMBOL_ORDERS_GTC

```

此处，你可以看到，几乎所有的标志都允许大多数交易品种，对于 `SYMBOL_EXPIRATION_GTC` 模式，仅使用了变体 `SYMBOL_ORDERS_GTC`。

将 `UseMarketWatch` 和 `ShowPerSymbolDetails` 设置为 `true` (假设在 *Market Watch* 中选择了有限数量的交易品种)，然后再次运行该脚本。

```

GBPUSD:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED ] (7)
    SYMBOL_ORDERS_GTC (0)
USDCHF:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED ] (7)
    SYMBOL_ORDERS_GTC (0)
USDJPY:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED ] (7)
    SYMBOL_ORDERS_GTC (0)
...
XAUUSD:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED
      _SYMBOL_EXPIRATION_SPECIFIED_DAY ] (15)
    SYMBOL_ORDERS_GTC (0)
SP500m:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED
      _SYMBOL_EXPIRATION_SPECIFIED_DAY ] (15)
    SYMBOL_ORDERS_GTC (0)
UK100:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED
      _SYMBOL_EXPIRATION_SPECIFIED_DAY ] (15)
    SYMBOL_ORDERS_GTC (0)
===== Expiration modes for Market Watch symbols =====
Total symbols: 15
Stats per expiration mode:
    [key] [value]
[0]      1      15
[1]      2      15
[2]      4      15
[3]      8       6
Legend: key=expiration mode, value=count
1 -> _SYMBOL_EXPIRATION_GTC
2 -> _SYMBOL_EXPIRATION_DAY
4 -> _SYMBOL_EXPIRATION_SPECIFIED
8 -> _SYMBOL_EXPIRATION_SPECIFIED_DAY
Stats per GTC mode:
    [key] [value]
[0]      0      15
Legend: key=GTC mode, value=count
0 -> SYMBOL_ORDERS_GTC

```

在 15 个选定的交易品种中，只有 6 个设置了 `SYMBOL_EXPIRATION_SPECIFIED_DAY` 标志。关于每个交易品种的标志的细节可以在上面找到。

6.1.18 点差以及订单价格与当前价格的距离

对于许多交易策略，尤其是那些基于短期交易的策略，关于当前价格的点差和偏离当前价格距离的信息，允许设置或修改订单是很重要的。所有这些特性均为 `ENUM_SYMBOL_INFO_INTEGER` 枚举的一部分，可通过 [*SymbolInfoInteger*](#) 函数获得。

标识符	说明
SYMBOL_SPREAD	点差大小 (以点为单位)
SYMBOL_SPREAD_FLOAT	浮动点差的布尔符号
SYMBOL_TRADE_STOPS_LEVEL	设置止损、止盈和挂单时与当前价格的最小允许距离 (以点为单位)
SYMBOL_TRADE_FREEZE_LEVEL	从当前价格 (以点为单位)到冻结订单和头寸的距离

上表中, 当前价格指的是 *Ask* 或 *Bid* 价格, 具体取决于所执行操作的特性。

保护性 *Stop Loss* 和 *Take Profit* 水平表明应执行平仓的时机。平仓通过与开仓相反的操作来执行。因此, 对于 *Ask* 价格的买入订单, 保护水平表示 *Bid*, 对于 *Bid* 价格的卖出订单, 保护水平表示 *Ask*。下达挂单时, 以标准方式选择开盘价类型: 买入订单 (*Buy Stop*, *Buy Limit*, *Buy Stop Limit*) 基于 *Ask*, 卖出订单 (*Sell Stop*, *Sell Limit*, *Sell Stop Limit*) 基于 *Bid*。考虑到上述交易操作环境中的价格类型, 应计算 `SYMBOL_TRADE_STOPS_LEVEL` 和 `SYMBOL_TRADE_FREEZE_LEVEL` 特性的距离 (以点数为单位)。

如果 `SYMBOL_TRADE_STOPS_LEVEL` 特性不为零, 则在新水平比指定距离更接近当前价格时, 禁止修改未结头寸的 *Stop Loss* 和 *Take Profit* 水平。同样, 挂单的开盘价格与当前价格的差距也不可能小于 `SYMBOL_TRADE_STOPS_LEVEL` 的点数。

如果 `SYMBOL_TRADE_FREEZE_LEVEL` 特性不为零, 则限制挂单或未结头寸在当前价格指定距离内的任何交易操作。对于挂单, 若指定的开盘价格与当前价格之间的距离小于 `SYMBOL_TRADE_FREEZE_LEVEL` 的点数, 就会触发冻结 (同样, 当前价格的类型为 *Ask* 或 *Bid*, 取决于其是买入还是卖出)。对于头寸, 若 *Stop Loss* 和 *Take Profit* 水平接近当前价格就会触发冻结, 因此对其测量是针对“相反的”价格类型进行的。

如果 `SYMBOL_SPREAD_FLOAT` 特性为 *true*, 则 `SYMBOL_SPREAD` 特性不是交易品种规范的一部分, 而是包含实际点差, 其根据市场条件随每次调用动态变化。也可以在 `MqlTick` 结构中找到其作为 *Ask* 和 *Bid* 之间的差额, 这是通过调用 `SymbolInfoTick`来完成的。

脚本 `SymbolFilterSpread.mq5` 将允许你分析指定的特性。该脚本定义了一个自定义枚举 `ENUM_SYMBOL_INFO_INTEGER_PART`, 该枚举仅包含 `ENUM_SYMBOL_INFO_INTEGER` 中我们感兴趣的特性。

```
enum ENUM_SYMBOL_INFO_INTEGER_PART
{
    SPREAD_FIXED = SYMBOL_SPREAD,
    SPREAD_FLOAT = SYMBOL_SPREAD_FLOAT,
    STOPS_LEVEL = SYMBOL_TRADE_STOPS_LEVEL,
    FREEZE_LEVEL = SYMBOL_TRADE_FREEZE_LEVEL
};
```

新的枚举定义了 `Property` 输入参数, 该参数指定将会分析四个特性中的哪一个。参数 `UseMarketWatch` 和 `ShowPerSymbolDetails` 以已知的方式控制该流程, 就像前面的测试脚本一样。


```
input bool UseMarketWatch = true;
input ENUM_SYMBOL_INFO_INTEGER_PART Property = SPREAD_FIXED;
input bool ShowPerSymbolDetails = true;
```

为了使用 *ArrayPrint* 函数方便地显示每个交易品种的信息 (每行中的特性名和值), 定义了一个辅助结构体 *SymbolDistance* (仅在 *ShowPerSymbolDetails* 等于 *true* 时使用)。

```
struct SymbolDistance
{
    string name;
    int value;
};
```

在 *OnStart* 处理程序中, 我们介绍了必要的对象和数组。

```
void OnStart()
{
    SymbolFilter f;                // filter object
    string symbols[];              // receiving array for names
    long values[];                 // receiving array for values
    SymbolDistance distances[];    // array to print
    MapArray<long,int> stats;       // counters of specific values of the selected prop
    ...
```

然后我们应用筛选器并用指定 *Property* 值填充接收数组, 同时也进行排序。

```
f.select(UseMarketWatch, (ENUM_SYMBOL_INFO_INTEGER)Property, symbols, values, true
const int n = ArraySize(symbols);
if(ShowPerSymbolDetails) ArrayResize(distances, n);
...
```

在一个循环中, 如果需要的话, 我们可以计算统计数据并填充 *SymbolDistance* 结构体。

```
for(int i = 0; i < n; ++i)
{
    stats.inc(values[i]);
    if(ShowPerSymbolDetails)
    {
        distances[i].name = symbols[i];
        distances[i].value = (int)values[i];
    }
}
...
```

最后, 我们可以将结果输出到日志中。

```

PrintFormat("==== Distances for %s symbols ====",
    (UseMarketWatch ? "Market Watch" : "all available"));
PrintFormat("Total symbols: %d", n);

PrintFormat("Stats per %s:", EnumToString((ENUM_SYMBOL_INFO_INTEGER)Property));
stats.print();

if(ShowPerSymbolDetails)
{
    Print("Details per symbol:");
    ArrayPrint(distances);
}
}

```

以下是使用默认设置运行脚本时得到的结果，这与点差分析一致。

```

==== Distances for Market Watch symbols ====
Total symbols: 13
Stats per SYMBOL_SPREAD:
    [key] [value]
[0]      0      2
[1]      2      3
[2]      3      1
[3]      6      1
[4]      7      1
[5]      9      1
[6]    151      1
[7]    319      1
[8]   3356      1
[9]   3400      1
Details per symbol:
    [name] [value]
[ 0] "USDJPY"      0
[ 1] "EURUSD"      0
[ 2] "USDCHF"      2
[ 3] "USDCAD"      2
[ 4] "GBPUSD"      2
[ 5] "AUDUSD"      3
[ 6] "XAUUSD"      6
[ 7] "SP500m"      7
[ 8] "NZDUSD"      9
[ 9] "USDCNH"    151
[10] "USDSEK"    319
[11] "BTCUSD"   3356
[12] "USDRUB"   3400

```

为了理解点差是浮动的 (动态变化的) 还是固定的，让我们使用不同的设置运行脚本：Property = SPREAD_FLOAT, ShowPerSymbolDetails = false。

```
===== Distances for Market Watch symbols =====
Total symbols: 13
Stats per SYMBOL_SPREAD_FLOAT:
    [key] [value]
[0]      1      13
```

根据该数据，市场报价中的所有交易品种均具有浮动点差（在 `SYMBOL_SPREAD_FLOAT` 中，*key* 键中的值 1 表示 *true*）。因此，如果我们用默认设置多次运行脚本，我们将会收到新的值（在市场开盘的情况下）。

6.1.19 获取掉期规模

对于中期和长期战略的实施，掉期规模很重要，因为其可能对财务结果产生重大影响，而且通常是负面的影响。但是，一些读者可能是“利差交易”策略的粉丝，这种策略最初建立在从积极的掉期交易中止盈的基础上。MQL5 有几个交易品种特性，可提供对与掉期相关的规范字符串的访问。

标识符	说明
<code>SYMBOL_SWAP_MODE</code>	掉期计算模型 <code>ENUM_SYMBOL_SWAP_MODE</code>
<code>SYMBOL_SWAP_ROLLOVER3DAYS</code>	三倍掉期信用对应的星期几 <code>ENUM_DAY_OF_WEEK</code>
<code>SYMBOL_SWAP_LONG</code>	多头仓位的掉期规模
<code>SYMBOL_SWAP_SHORT</code>	空头仓位的掉期规模

`ENUM_SYMBOL_SWAP_MODE` 枚举包含指定度量单位选项和掉期计算原则的元素。和 `SYMBOL_SWAP_ROLLOVER3DAYS` 一样，其引用了 `ENUM_SYMBOL_INFO_INTEGER` 的整数特性。

掉期规模直接在 `SYMBOL_SWAP_LONG` 和 `SYMBOL_SWAP_SHORT` 特性中指定，作为 `ENUM_SYMBOL_INFO_DOUBLE` 的一部分，即 *double* 类型。

以下是 `ENUM_SYMBOL_SWAP_MODE` 的元素。

标识符	说明
<code>SYMBOL_SWAP_MODE_DISABLED</code>	无掉期
<code>SYMBOL_SWAP_MODE_POINTS</code>	点数
<code>SYMBOL_SWAP_MODE_CURRENCY_SYMBOL</code>	交易品种的基础货币
<code>SYMBOL_SWAP_MODE_CURRENCY_MARGIN</code>	交易品种的保证金货币
<code>SYMBOL_SWAP_MODE_CURRENCY_DEPOSIT</code>	存款货币
<code>SYMBOL_SWAP_MODE_INTEREST_CURRENT</code>	计算掉期时金融工具价格的年度百分比
<code>SYMBOL_SWAP_MODE_INTEREST_OPEN</code>	每年占交易品种头寸开盘价的百分比
<code>SYMBOL_SWAP_MODE_REOPEN_CURRENT</code>	点数 (以收盘价重新开仓)
<code>SYMBOL_SWAP_MODE_REOPEN_BID</code>	点数 (以新一天的买入价格重新开仓) (在 <code>SYMBOL_SWAP_LONG</code> 和 <code>SYMBOL_SWAP_SHORT</code> 参数中)

对于 `SYMBOL_SWAP_MODE_INTEREST_CURRENT` 和 `SYMBOL_SWAP_MODE_INTEREST_OPEN` 期权, 假设一年中有 360 个银行工作日。

对于 `SYMBOL_SWAP_MODE_REOPEN_CURRENT` 和 `SYMBOL_SWAP_MODE_REOPEN_BID` 期权, 在交易日结束时强行平仓, 但之后两者的行为存在差异。

使用 `SYMBOL_SWAP_MODE_REOPEN_CURRENT` 时, 持仓次日会以前一天的收盘价+/-指定的点数重新开仓。使用 `SYMBOL_SWAP_MODE_REOPEN_BID` 时, 持仓次日会以当前买入价+/-指定点数重新开仓。在这两种情况下, 点数都在 `SYMBOL_SWAP_LONG` 和 `SYMBOL_SWAP_SHORT` 参数中。

我们使用脚本 *SymbolFilterSwap.mq5* 来检查特性的操作。在输入参数中, 我们提供了分析上下文的选择: *Market Watch* 或所有依赖于 *UseMarketWatch* 的交易品种。当 *ShowPerSymbolDetails* 参数为 *false* 时, 我们将计算统计数据, 即 `ENUM_SYMBOL_SWAP_MODE` 中的一个或另一个模式在交易品种中使用了多少次。当 *ShowPerSymbolDetails* 参数为 *true* 时, 我们将使用 *mode* 中指定的模式输出所有交易品种的数组, 并按照 `SYMBOL_SWAP_LONG` 和 `SYMBOL_SWAP_SHORT` 字段中值的降序对数组进行排序。

```
input bool UseMarketWatch = true;
input bool ShowPerSymbolDetails = false;
input ENUM_SYMBOL_SWAP_MODE Mode = SYMBOL_SWAP_MODE_POINTS;
```

对于掉期组合数组的元素, 我们用交易品种名称和掉期值来说明 *SymbolSwap* 结构。掉期的方向用名称字段中的前缀表示: “+”表示多头掉期; “-”表示空头掉期。

```
struct SymbolSwap
{
    string name;
    double value;
};
```

按照惯例, 我们在 *OnStart* 的开头描述筛选对象。但是, 根据 *ShowPerSymbolDetails* 变量的值, 下面的代码会有很大的不同。

```

void OnStart()
{
    SymbolFilter f;                // filter object
    PrintFormat("==== Swap modes for %s symbols =====",
        (UseMarketWatch ? "Market Watch" : "all available"));

    if>ShowPerSymbolDetails)
    {
        // summary table of swaps of the selected Mode
        ...
    }
    else
    {
        // calculation of mode statistics
        ...
    }
}

```

先介绍第二个分支。此处，我们使用取自 `SYMBOL_SWAP_MODE` 特性的筛选器 (*symbols*) 和掉期模式 (*values*)，用交易品种名称填充数组。结果值累积在数组映射 `MapArray<ENUM_SYMBOL_SWAP_MODE,int> stats` 中。

```

// calculation of mode statistics
string symbols[];
long values[];
MapArray<ENUM_SYMBOL_SWAP_MODE,int> stats; // counters for each mode
// apply filter and collect mode values
f.select(UseMarketWatch, SYMBOL_SWAP_MODE, symbols, values);
const int n = ArraySize(symbols);
for(int i = 0; i < n; ++i)
{
    stats.inc((ENUM_SYMBOL_SWAP_MODE)values[i]);
}
...

```

接下来，我们显示收集的统计数据。

```

PrintFormat("Total symbols: %d", n);
Print("Stats per swap mode:");
stats.print();
Print("Legend: key=swap mode, value=count");
for(int i = 0; i < stats.GetSize(); ++i)
{
    PrintFormat("%d -> %s", stats.getKey(i), EnumToString(stats.getKey(i)));
}

```

对于用掉期值构造表的情况，算法如下。多头和空头的掉期是分别请求的，因此我们为名称和价值定义了成对数组。它们将一起被放入 *swaps* 结构体数组中。

```
// summary table of swaps of the selected Mode
string buyers[], sellers[]; // arrays for names
double longs[], shorts[]; // arrays for swap values
SymbolSwap swaps[]; // total array to print
```

在筛选器中设置所选掉期模式的条件。这是比较和排序数组元素所必需的。

```
f.let(SYMBOL_SWAP_MODE, Mode);
```

然后我们对不同的特性 (SYMBOL_SWAP_LONG 和 SYMBOL_SWAP_SHORT) 应用筛选器两次, 并用其值 (*longs*, *shorts*) 填充不同的数组。在每次调用中, 数组按升序排序。

```
f.select(UseMarketWatch, SYMBOL_SWAP_LONG, buyers, longs, true);
f.select(UseMarketWatch, SYMBOL_SWAP_SHORT, sellers, shorts, true);
```

理论上, 数组的大小应相同, 因为筛选条件是相同的, 但是为了清楚起见, 让我们为每个大小分配一个变量。由于每个交易品种将在结果表中出现两次, 对于多头和空头, 我们为 *swaps* 数组提供了双倍规模。

```
const int l = ArraySize(longs);
const int s = ArraySize(shorts);
const int n = ArrayResize(swaps, l + s); // should be l == s
PrintFormat("Total symbols with %s: %d", EnumToString(Mode), l);
```

接下来, 我们连接两个数组 *longs* 和 *shorts*, 以相反的顺序对其进行处理, 因为我们需要从正值到负值进行排序。

```

if(n > 0)
{
    int i = l - 1, j = s - 1, k = 0;
    while(k < n)
    {
        const double swapLong = i >= 0 ? longs[i] : -DBL_MAX;
        const double swapShort = j >= 0 ? shorts[j] : -DBL_MAX;

        if(swapLong >= swapShort)
        {
            swaps[k].name = "+" + buyers[i];
            swaps[k].value = longs[i];
            --i;
            ++k;
        }
        else
        {
            swaps[k].name = "-" + sellers[j];
            swaps[k].value = shorts[j];
            --j;
            ++k;
        }
    }
    Print("Swaps per symbols (ordered):");
    ArrayPrint(swaps);
}

```

有趣的是用不同的设置多次运行这个脚本。例如，默认情况下，我们可以得到以下结果。

```

===== Swap modes for Market Watch symbols =====
Total symbols: 13
Stats per swap mode:
    [key] [value]
[0]      1      10
[1]      0       2
[2]      2       1
Legend: key=swap mode, value=count
1 -> SYMBOL_SWAP_MODE_POINTS
0 -> SYMBOL_SWAP_MODE_DISABLED
2 -> SYMBOL_SWAP_MODE_CURRENCY_SYMBOL

```

统计数据显示，10 个交易品种具有掉期模式，SYMBOL_SWAP_MODE_POINTS；两个交易品种禁用掉期，SYMBOL_SWAP_MODE_DISABLED；一个交易品种处于基础货币，SYMBOL_SWAP_MODE_CURRENCY_SYMBOL。

我们来确定什么样的交易品种有 SYMBOL_SWAP_MODE_POINTS，并找出其掉期。为此，我们将 *ShowPerSymbolDetails* 设置为 *true*（参数 *mode* 已设置为 SYMBOL_SWAP_MODE_POINTS）。

```

===== Swap modes for Market Watch symbols =====
Total symbols with SYMBOL_SWAP_MODE_POINTS: 10
Swaps per symbols (ordered):
      [name]      [value]
[ 0] "+AUDUSD"    6.30000
[ 1] "+NZDUSD"    2.80000
[ 2] "+USDCHF"    0.10000
[ 3] "+USDRUB"    0.00000
[ 4] "-USDRUB"    0.00000
[ 5] "+USDJPY"   -0.10000
[ 6] "+GBPUSD"   -0.20000
[ 7] "-USDCAD"   -0.40000
[ 8] "-USDJPY"   -0.60000
[ 9] "+EURUSD"   -0.70000
[10] "+USDCAD"   -0.80000
[11] "-EURUSD"   -1.00000
[12] "-USDCHF"   -1.00000
[13] "-GBPUSD"   -2.20000
[14] "+USDSEK"   -4.50000
[15] "-XAUUSD"   -4.60000
[16] "-USDSEK"   -4.90000
[17] "-NZDUSD"   -6.70000
[18] "+XAUUSD"  -12.60000
[19] "-AUDUSD"  -14.80000

```

你可以将这些值与交易品种规范进行比较。

最后，我们将模式改为 `SYMBOL_SWAP_Mode_CURRENCY_SYMBOL`。在我们的示例中，我们应得到一个交易品种，但是分成两行：名称中有一个加号和一个减号。

```

===== Swap modes for Market Watch symbols =====
Total symbols with SYMBOL_SWAP_MODE_CURRENCY_SYMBOL: 1
Swaps per symbols (ordered):
      [name]      [value]
[0] "-SP500m"  -35.00000
[1] "+SP500m"  -41.41000

```

从表中可以看出，两种掉期均为负。

6.1.20 当前市场信息（分时报价）

在 [获取交易品种的最新分时报价](#) 一节中，我们已经遇到过 `SymbolInfoTick` 函数，其以 `MqlTick` 结构体的形式提供了关于最后一条分时报价（价格变化事件）的完整信息。如有必要，MQL 程序可以分别请求与该结构体的字段相对应的价格和交易量的值。二者表示为不同类型的特性，这些特性为 `ENUM_SYMBOL_INFO_INTEGER` 和 `ENUM_SYMBOL_INFO_DOUBLE` 枚举的一部分。

标识符	说明	特性类型
SYMBOL_TIME	最新报价时间	datetime
SYMBOL_BID	买入价；最佳卖出价格	double
SYMBOL_ASK	卖出价；最佳买入价格	double
SYMBOL_LAST	最后一笔；最后一笔交易的价格	double
SYMBOL_VOLUME	最后一笔交易的成交量	long
SYMBOL_TIME_MSC	自 1970 年 01 月 01 日以来最后一次报价的时间（以毫秒为单位）	long
SYMBOL_VOLUME_REAL	最后一笔交易的成交量（精度更高）	double

注意，两个与成交量相关的特性 SYMBOL_VOLUME 和 SYMBOL_VOLUME_REAL 的代码在两个枚举中是相同的。这是不同枚举的元素 ID 发生重叠的唯一情况。事实上，二者返回本质上相同的分时报价特性，只是表示精度不同而已。

与结构体不同的是，特性未提供类似 *uint flags* 的字段，该字段表示市场上哪种变化导致了分时报价的生成。该字段仅在结构体中有意义。

我们来尝试单独请求分时报价特性，并将其与 *SymbolInfoTick* 调用的结果进行比较。在快速变化的市场上，结果可能会有所不同。函数调用之间可能会出现一条新分时报价（甚至几条分时报价）。

```
void OnStart()
{
    PRTF(TimeToString(SymbolInfoInteger(_Symbol, SYMBOL_TIME), TIME_DATE | TIME_SECONDS));
    PRTF(SymbolInfoDouble(_Symbol, SYMBOL_BID));
    PRTF(SymbolInfoDouble(_Symbol, SYMBOL_ASK));
    PRTF(SymbolInfoDouble(_Symbol, SYMBOL_LAST));
    PRTF(SymbolInfoInteger(_Symbol, SYMBOL_VOLUME));
    PRTF(SymbolInfoInteger(_Symbol, SYMBOL_TIME_MSC));
    PRTF(SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_REAL));

    MqlTick tick[1];
    SymbolInfoTick(_Symbol, tick[0]);
    ArrayPrint(tick);
}
```

在特定情况下，很容易验证信息是否一致。

```
TimeToString(SymbolInfoInteger(_Symbol,SYMBOL_TIME),TIME_DATE|TIME_SECONDS)
    =2022.01.25 13:52:51 / ok
SymbolInfoDouble(_Symbol,SYMBOL_BID)=1838.44 / ok
SymbolInfoDouble(_Symbol,SYMBOL_ASK)=1838.49 / ok
SymbolInfoDouble(_Symbol,SYMBOL_LAST)=0.0 / ok
SymbolInfoInteger(_Symbol,SYMBOL_VOLUME)=0 / ok
SymbolInfoInteger(_Symbol,SYMBOL_TIME_MSC)=1643118771166 / ok
SymbolInfoDouble(_Symbol,SYMBOL_VOLUME_REAL)=0.0 / ok
                [time]   [bid]   [ask] [last] [volume]   [time_msc] [flags] [volume
[0] 2022.01.25 13:52:51 1838.44 1838.49   0.00           0 1643118771166      6
```

6.1.21 交易品种的说明性特性

该平台为 MQL 程序提供了一组说明重要定性特征的文本特性。例如，在基于一篮子金融工具开发指标或交易策略时，可能需要根据来源国、经济部门或基础资产名称（如果工具为衍生工具）选择交易品种。

标识符	说明
SYMBOL_BASIS	衍生品基础资产的名称
SYMBOL_CATEGORY	金融工具所属类别的名称
SYMBOL_COUNTRY	金融工具所归属的国家
SYMBOL_SECTOR_NAME	金融工具所属的经济部门
SYMBOL_INDUSTRY_NAME	金融工具所属的经济分支或行业类型
SYMBOL_BANK	当前报价来源
SYMBOL_DESCRIPTION	交易品种的字符串说明
SYMBOL_EXCHANGE	交易品种进行交易的交易所或市场名称
SYMBOL_ISIN	国际证券识别代码系统中唯一的 12 位字母数字代码 - ISN（国际证券识别号）
SYMBOL_PAGE	带有交易品种信息的互联网网页地址
SYMBOL_PATH	交易品种树中的路径

程序可以应用这些特性分析的另一种情况发生在寻找一种货币到另一种货币的兑换率时。我们已经知道如何找到一个 **基础货币和报价货币** 正确组合的交易品种，但问题是可能有多个这样的交易品种。在这种情况下，读取像 SYMBOL_SECTOR_NAME（你需要寻找“货币”或者一个同义词；查看经纪商的规范）或 SYMBOL_PATH 之类的特性可能有所帮助。

SYMBOL_PATH 包含交易品种目录中包含特定交易品种的文件夹的整个层次结构：文件夹名称用双反斜杠 (\\) 分隔，与文件系统相同。路径的最后一个元素为交易品种本身的名称。

一些字符串特性有整数对应项。特别是，你可以使用 `SYMBOL_SECTOR` 特性代替 `SYMBOL_SECTOR_NAME`，该特性可返回一个枚举元素 `ENUM_SYMBOL_SECTOR` 以及所有支持的行

业板块。类似地，对于 `SYMBOL_INDUSTRY_NAME`，也有一个类似的特性 `SYMBOL_INDUSTRY`，它具有 `ENUM_SYMBOL_INDUSTRY` 枚举类型。

如果需要，MQL 程序甚至可以通过读取 `SYMBOL_BACKGROUND_COLOR` 特性来找到在市场报价中显示交易品种时使用的背景颜色。这将允许那些程序使用 **图形对象**（对话框、列表等）在图表上创建自己的界面，使其与本机终端控件保持统一。

考虑示例脚本 *SymbolFilterDescription.mqh*，该脚本可为 *Market Watch* 交易品种输出四个预定义的文本特性。第一个是 `SYMBOL_DESCRIPTION`（不要与交易品种本身的名称混淆），结果列表将通过其进行排序。其他三个仅供参考：`SYMBOL_SECTOR_NAME`、`SYMBOL_COUNTRY` 和 `SYMBOL_PATH`。对于每个经纪商，所有值都以特定的方式填写（对于同一个报价机，可能会有差异）。

虽然我们没有提及，但是我们的 *SymbolFilter* 类实现了 *equal* 方法的特殊重载来比较字符串。其支持使用通配符 `*` 代表 0 或更多任意字符的模式来搜索子字符串的出现。例如，`*ian*` 可查找包含子字符串 `ian`（任意位置）的所有字符，`*Index` 仅查找以 `Index` 结尾的字符串。

此功能类似于用户可用的 *Symbols* 对话框中的子字符串搜索。但是，不需要指定通配符，因为总是会搜索子字符串。在源代码（*SymbolFilter.mqh*）提供的算法中，我们保留了搜索完全匹配（没有 `*` 通配符）或子串（至少有一个星号）的功能。

比较区分大小写。如果需要，很容易修改代码进行比较，而不用区分大小写字母。

如果给定新特性，我们可在交易品种说明中为搜索字符串定义一个输入变量。如果该变量为空，将显示 *Market Watch* 窗口中的所有交易品种。

```
input string SearchPattern = "";
```

其它方面，一切如常。

```

void OnStart()
{
    SymbolFilter f;                // filter object
    string symbols[];              // array of names
    string text[][4];              // array of vectors with data

    // properties to read
    ENUM_SYMBOL_INFO_STRING fields[] =
    {
        SYMBOL_DESCRIPTION,
        SYMBOL_SECTOR_NAME,
        SYMBOL_COUNTRY,
        SYMBOL_PATH
    };

    if(SearchPattern != "")
    {
        f.let(SYMBOL_DESCRIPTION, SearchPattern);
    }

    // apply the filter and get arrays sorted by description
    f.select(true, fields, symbols, text, true);

    const int n = ArraySize(symbols);
    PrintFormat("==== Text fields for symbols (%d) =====", n);
    for(int i = 0; i < n; ++i)
    {
        Print(symbols[i] + ":");
        ArrayPrint(text, 0, NULL, i, 1, 0);
    }
}

```

以下是该列表的一个可能版本（带缩写）。

```

===== Text fields for symbols (16) =====
AUDUSD:
"Australian Dollar vs US Dollar" "Currency" "" "Forex\AUDUSD"
EURUSD:
"Euro vs US Dollar" "Currency" "" "Forex\EURUSD"
UK100:
"FTSE 100 Index" "Undefined" "" "Indexes\UK100"
XAUUSD:
"Gold vs US Dollar" "Commodities" "" "Metals\XAUUSD"
JAGG:
"JPMorgan U.S. Aggregate Bond ETF" "Financial"
"USA" "ETF\United States\NYSE\JPMorgan\JAGG"
NZDUSD:
"New Zealand Dollar vs US Dollar" "Currency" "" "Forex\NZDUSD"
GBPUSD:
"Pound Sterling vs US Dollar" "Currency" "" "Forex\GBPUSD"
SP500m:
"Standard & Poor's 500" "Undefined" "" "Indexes\SP500m"
FIHD:
"UBS AG FI Enhanced Global High Yield ETN" "Financial"
"USA" "ETF\United States\NYSE\UBS\FIHD"
...

```

如果我们将搜索字符串 `*ian*` 输入到输入变量 `SearchPattern` 中, 可得到以下结果。

```

===== Text fields for symbols (3) =====
AUDUSD:
"Australian Dollar vs US Dollar" "Currency" "" "Forex\AUDUSD"
USDCAD:
"US Dollar vs Canadian Dollar" "Currency" "" "Forex\USDCAD"
USDRUB:
"US Dollar vs Russian Ruble" "Currency" "" "Forex\USDRUB"

```

6.1.22 市场深度

当涉及到交易所金融工具时, MetaTrader 5 不仅可以让你获得以 [分时报价](#) 方式打包的价格和交易量信息, 还可以获得市场深度 (订单簿, 二级价格), 即围绕当前价格的几个最接近水平下已下达买入和卖出订单的交易量分布。交易品种 `SYMBOL_TICKS_BOOKDEPTH` 的其中一个整数特性包含市场深度中显示的最大层级数。每一方均允许该交易量, 即订单簿的总规模可以扩大两倍 (并且不考虑未播报的零交易量价位)。

根据市场情况, 传输的订单簿的实际规模可能会小于该特性中显示的规模。对于非交易所金融工具, 此特性通常等于 0, 尽管一些经纪商可以播报仅受其客户订单限制的外汇交易品种的订单簿。

订单簿本身及其更新通知必须通过感兴趣的 MQL 程序使用一个特殊的 API 进行请求, 我们将在 [下一章](#) 讨论。

需要注意的是, 由于平台的架构特点, 该特性与订单簿的转换没有直接关系, 也就是说, 它只是由经纪商填充的规范字段。换言之, 该特性的非零值并不意味着订单簿一定会在开放市场中到达终端。这取决于其他服务器设置以及其是否与数据提供程序有活动连接。

让我们尝试使用脚本 *SymbolFilterBookDepth.mq5* 来获得所有或选定交易品种的市场深度统计数据。

```
input bool UseMarketWatch = false;
input int ShowSymbolsWithDepth = -1;
```

参数 *ShowSymbolsWithDepth* 默认等于 -1 指示收集所有交易品种中不同市场深度设置的统计数据。如果将该参数设置为不同的值，程序将尝试查找具有指定订单簿深度的所有交易品种。

```
void OnStart()
{
    SymbolFilter f;           // filter object
    string symbols[];         // array for symbol names
    long depths[];           // array of property values
    MapArray<long,int> stats;  // counters of occurrences of each depth

    if(ShowSymbolsWithDepth > -1)
    {
        f.let(SYMBOL_TICKS_BOOKDEPTH, ShowSymbolsWithDepth);
    }

    // apply filter and fill arrays
    f.select(UseMarketWatch, SYMBOL_TICKS_BOOKDEPTH, symbols, depths, true);
    const int n = ArraySize(symbols);

    PrintFormat("==== Book depths for %s symbols %s====",
        (UseMarketWatch ? "Market Watch" : "all available"),
        (ShowSymbolsWithDepth > -1 ? "(filtered by depth="
        + (string)ShowSymbolsWithDepth + ") " : ""));
    PrintFormat("Total symbols: %d", n);
    ...
}
```

如果给定了特定深度，我们只需输出一个交易品种数组（均满足筛选条件），然后退出。

```
if(ShowSymbolsWithDepth > -1)
{
    ArrayPrint(symbols);
    return;
}
...
```

否则，我们将计算并显示统计数据。

```
for(int i = 0; i < n; ++i)
{
    stats.inc(depths[i]);
}

Print("Stats per depth:");
stats.print();
Print("Legend: key=depth, value=count");
}
```

使用默认设置时，我们可以得到下图。

```

===== Book depths for all available symbols =====
Total symbols: 52357
Stats per depth:
    [key] [value]
[0]      0   52244
[1]      5      3
[2]     10     67
[3]     16      5
[4]     20     13
[5]     32     25
Legend: key=depth, value=count

```

如果将 *ShowSymbolsWithDepth* 设置为检测到的值之一（例如 32），我们可获得一个具有此订单簿深度的交易品种列表。

```

===== Book depths for all available symbols (filtered by depth=32) =====
Total symbols: 25
[ 0] "USDCNH" "USDZAR" "USDHUF" "USDPLN" "EURHUF" "EURNOK" "EURPLN" "EURSEK" "EURZAR"
[13] "NZDCAD" "NZDCHF" "USDMXN" "EURMXN" "GBPMXN" "CADMXN" "CHFMXN" "MXNJPY" "NZDMXN"

```

6.1.23 自定义交易品种特性

在本章的前言中，我们提到了 [自定义交易品种](#)。这些交易品种的报价是根据用户的命令或以编程方式在终端中直接创建的。

例如，自定义交易品种可用于根据包含其他市场报价交易品种的公式创建合成工具。用户可以直接在 [终端界面](#) 中执行该操作。

MQL 程序可以在 MQL5 中实现更复杂的场景，例如合并不同时间段的不同金融工具、根据给定的随机分布生成序列，或者从外部来源接收数据（报价、柱线或分时报价）。

为了能够在算法中区分标准交易品种和自定义交易品种，MQL5 提供了 `SYMBOL_CUSTOM` 特性，这是一个逻辑符号，用于判定某个交易品种是否为自定义。

如果该交易品种有公式，则可通过 `SYMBOL_FORMULA` 字符串特性获得。我们知道，在公式中，你可以使用其他交易品种的名称，以及数学函数和运算符。以下是几个例子：

- 合成交易品种：“@ESU19”/EURCAD
- 日历点差：“Si9.13”-“Si6.13”
- 欧元指数： $34.38805726 * \text{pow}(\text{EURUSD}, 0.3155) * \text{pow}(\text{EURGBP}, 0.3056) * \text{pow}(\text{EURJPY}, 0.1891) * \text{pow}(\text{EURCHF}, 0.1113) * \text{pow}(\text{EURSEK}, 0.0785)$

指定公式对用户而言很方便，但通常不在 MQL 程序中使用，因为 MQL 程序可以直接在代码中计算公式，借助非标准函数并拥有更强的控制能力（特别是对于每条分时报价，而非依赖每 100 毫秒 1 次的计时器）。

我们用脚本 *SymbolFilterCustom.mq5* 中的特性来检查运行情况：该脚本记录了所有自定义交易品种及其公式（如果有的话）。

```

input bool UseMarketWatch = false;

void OnStart()
{
    SymbolFilter f;           // filter object
    string symbols[];         // array for symbol names
    string formulae[];        // array for formulas

    // apply filter and fill arrays
    f.let(SYMBOL_CUSTOM, true)
    .select(UseMarketWatch, SYMBOL_FORMULA, symbols, formulae);
    const int n = ArraySize(symbols);

    PrintFormat("==== %s custom symbols ====",
        (UseMarketWatch ? "Market Watch" : "All available"));
    PrintFormat("Total symbols: %d", n);

    for(int i = 0; i < n; ++i)
    {
        Print(symbols[i], " ", formulae[i]);
    }
}

```

以下是找到的唯一自定义字符的结果。

```

==== All available custom symbols ====
Total symbols: 1
synthEURUSD SP500m/UK100

```

6.1.24 特定特性 (证券交易所、期货市场、债券)

在本章的最后一节，我们将简要回顾其他交易品种特性，这些特性不在本书的讨论范围内，但可能对实现高级交易策略有用。关于这些特性的详细信息可以在 [MQL5 文档](#) 中找到。

我们知道，MetaTrader 5 允许你交易衍生市场金融工具，包括期权、期货和债券。这一功能也反映在软件界面上。MQL5 API 提供了许多与提到的金融工具类别相关的特定交易品种特性。

具体而言，期权交易品种特性包括流通期 (交易的开始日期 `SYMBOL_START_TIME` 和结束日期 `SYMBOL_EXPIRATION_TIME`)、执行价格 (`SYMBOL_OPTION_STRIKE`)、买入或卖出权利 (`SYMBOL_OPTION_RIGHT`、看涨/看跌期权)、欧洲或美洲类型 (`SYMBOL_OPTION_MODE`)，具体取决于提前行使的可能性、每天的收盘价格变化 (`SYMBOL_PRICE_CHANGE`) 和波动率 (`SYMBOL_PRICE_VOLATILITY`) 以及表征价格行为动态的估计系数 (希腊字母系数)。

特别相关的债券交易品种特性包括累计票息收益 (`SYMBOL_TRADE_ACCRUED_INTEREST`)、面值 (`SYMBOL_TRADE_FACE_VALUE`)、流动性比率 (`SYMBOL_TRADE_LIQUIDITY_RATE`)。

期货交易品种特性包括持仓量 (`SYMBOL_SESSION_INTEREST`)、买入委托总量 (`SYMBOL_SESSION_BUY_ORDERS_VOLUME`) 和卖出委托总量 (`SYMBOL_SESSION_SELL_ORDERS_VOLUME`) 以及交易时段收盘时的结算价 (`SYMBOL_SESSION_PRICE_SETTLEMENT`)。

除了组成一条分时报价的 [当前市场数据](#) 之外，MQL5 还允许你了解这些数据的每日范围：每个分时报价字段的最大值和最小值。例如，`SYMBOL_BIDHIGH` 为每日的最高 *Bid*，`SYMBOL_BIDLOW`

为最低买价。请注意，`SYMBOL_VOLUMEHIGH` 和 `SYMBOL_VOLUMELOW` (*long* 类型)特性实际上复制了 `SYMBOL_VOLUMEHIGH_REAL` 和 `SYMBOL_VOLUMELOW_REAL` (*double*) 中的交易量，但精度较低。

通常，有关 *Last* 价格和交易量的信息仅适用于交易所的交易品种。

请记住，特性的填写取决于经纪商实现的服务器设置。

6.2 市场深度

除了几种类型的最新市场价格数据 (*Ask/Bid/Last*) 和终端中以 *分时报价* 形式接收的最近交易量之外，MetaTrader 5 还支持市场深度 (订单簿)，这是一组关于围绕当前市场价格的买入和卖出订单交易量记录。根据交易品种规范，交易量在当前价格上下的几个水平上汇总，具有最小的价格变动增量。正如我们所看到的，最大订单簿规模 (价格水平的数量) 在交易品种特性 `SYMBOL_TICKS_BOOKDEPTH` 中设置。

终端用户在界面中了解市场深度特性及其运算原理。如需更多详情，请参阅 [文档](#)。

订单簿包含扩展的市场信息，通常称为“市场深度”。了解市场深度可让你创建更复杂的交易系统。

事实上，关于分时报价的信息只是订单簿的一小部分。在某种简化意义上，分时报价是一个 2 级订单簿，具有一个最接近的 *Ask* 价格 (可用报价) 和一个最接近的 *Bid* 价格 (可用需求)。此外，分时报价不提供这些价格的订单交易量。

市场深度的变化可能比分时报价更频繁，因为它们不仅影响对已完成交易的反应，还影响市场深度中挂单 (限价订单) 的成交量变化。

通常，订单簿和报价 (分时报价和交易) 的数据提供者是不同的实例，分时报价事件 (`EA` 交易中的 *OnTick* 或指标中的 *OnCalculate*) 与市场深度事件并不匹配。这两个线程以异步方式并行到达，但最终都会进入 MQL 程序的 *事件队列* 中。

需要注意的是，订单簿通常适用于交易所交易品种，但也存在正反两方面的例外情况：

- 由于这样或那样的原因，交易金融工具可能缺乏市场深度
- 经纪商可以根据其收集的客户端订单信息，为 OTC 金融工具提供市场深度

在 MQL5 中，市场数据深度适用于 `EA` 交易和指标。通过使用特殊函数 (*MarketBookAdd* 和 *MarketBookRelease*)，程序可以启用或禁用其订阅，以接收关于平台中市场深度变化的通知。为了接收通知，程序必须在其代码中定义 *OnBookEvent* 事件处理函数。收到通知后，可以使用 *MarketBookGet* 函数读取订单簿数据。

终端可维护报价和分时报价的历史，但不会维护市场深度数据。特别是，用户或 MQL 程序可以下载所要求时间范围的历史数据 (如果经纪商有)，并在其上测试 `EA` 交易和指标。

相比之下，市场深度仅在线播报，并不提供给测试程序。经纪商在服务器上没有市场深度数据的存档。为了在测试程序中模拟订单簿的行为，你应在线收集市场深度的历史，然后从测试程序中运行的 MQL 程序内读取。你可以在 MQL5 市场上找到现成的产品。

6.2.1 管理市场深度事件的订阅

终端在订阅的基础上接收市场深度信息：MQL 程序必须表达其接收市场深度 (订单簿) 事件的意图，或者相反，通过调用适当的函数 *MarketBookAdd* 和 *MarketBookRelease* 来终止其订阅。

MarketBookAdd 函数用于订阅接收有关指定金融工具订单簿变更的通知。因此，你可以订阅许多金融工具的订单簿，而不仅限于当前图表的工作金融工具。

bool MarketBookAdd (const string symbol)

通常，这个函数从 **OnInit** 或一个长期对象的类构造函数中调用。有关订单簿更改的通知会以 **OnBookEvent** 事件的形式发送给程序，因此，为了进行处理，程序必须有一个同名的处理程序函数。

如果在调用该函数之前没有在市场报价中选择指定的交易品种，该交易品种将被自动添加到窗口中。

MarketBookRelease 函数用于取消订阅有关指定订单簿变更的通知。

bool MarketBookRelease (const string symbol)

通常，这个函数应从 **OnDeinit** 或者一个长期对象的析构函数中调用。

如果调用成功，两个函数均返回值 **true**，否则返回值 **false**。

对于在同一图表上运行的所有应用程序，单独的订阅计数器通过交易品种进行维护。换言之，图表上的不同交易品种可以有几个订阅，每个订阅都有自己的计数器。

若通过单次调用任何函数而订阅或取消订阅，则仅改变当前运行程序的特定图表上特定交易品种的订阅计数器。这意味着两个图表可以订阅相同交易品种的 **OnBookEvent** 事件，但订阅计数器的值不同。

订阅计数器的初始值为零。每次调用 **MarketBookAdd** 时，给定图表上指定交易品种的订阅计数器都会加 1 (图表交易品种和 **MarketBookAdd** 中的交易品种不必匹配)。当调用 **MarketBookRelease** 时，图表中指定交易品种的订阅计数器减 1。

OnBookEvent 事件 (针对图表中的任何交易符号)生成。但前提是该交易品种的订阅计数器大于零。因此必须要确保每个包含 **MarketBookAdd** 调用的 MQL 程序在完成其工作后，均可使用 **MarketBookRelease** 正确地取消订阅每个交易品种的接收事件。为此，应确保 **MarketBookAdd** 调用和 **MarketBookRelease** 调用的数量相匹配。MQL5 不支持查询计数器的值。

第一个示例是一个简单的无缓冲指标 **MarketBookAddRelease.mq5**，该指标在启动时可启用对订单簿的订阅，在卸载时禁用订阅。在 **WorkSymbol** 输入参数中，可以指定要订阅的交易品种。如果留空 (默认值)，则为当前图表的工作交易品种启动订阅。

```

input string WorkSymbol = ""; // WorkSymbol (empty means current chart symbol)

const string _WorkSymbol = StringLen(WorkSymbol) == 0 ? _Symbol : WorkSymbol;
string symbols[];

void OnInit()
{
    const int n = StringSplit(_WorkSymbol, ',', symbols);
    for(int i = 0; i < n; ++i)
    {
        if(!PRTF(MarketBookAdd(symbols[i])))
            PrintFormat("MarketBookAdd(%s) failed", symbols[i]);
    }
}

int OnCalculate(const int rates_total, const int prev_calculated, const int, const dc
{
    return rates_total;
}

void OnDeinit(const int)
{
    for(int i = 0; i < ArraySize(symbols); ++i)
    {
        if(!PRTF(MarketBookRelease(symbols[i])))
            PrintFormat("MarketBookRelease(%s) failed", symbols[i]);
    }
}

```

作为一项附加功能，允许指定多个金融工具（以逗号分隔）。在这种情况下，可请求订阅所有交易品种。

当指标启动时，日志中会显示订阅成功的标志或错误代码。然后，指标会尝试取消订阅 *OnDeinit* 处理程序中的事件。

使用默认设置时，在可以使用订单簿的交易品种图表上，日志中将生成以下条目。

```

MarketBookAdd(symbols[i])=true / ok
MarketBookRelease(symbols[i])=true / ok

```

如果将指标放在有交易品种而没有订单簿的图表上，会看到错误代码。

```

MarketBookAdd(symbols[i])=false / BOOKS_CANNOT_ADD(4901)
MarketBookAdd(XPDUSD) failed
MarketBookRelease(symbols[i])=false / BOOKS_CANNOT_DELETE(4902)
MarketBookRelease(XPDUSD) failed

```

你可以通过在输入参数 *WorkSymbol* 中指定现有的或缺少的字符来进行体验。在下一节中，我们将考虑订阅几个交易品种订单簿的情况。

6.2.2 接收市场深度变化的事件

当订单簿状态变化时，终端会生成 **OnBookEvent** 事件。该事件由源代码中定义的 **OnBookEvent** 函数处理。为了让终端开始向 MQL 程序发送特定交易品种的 **OnBookEvent** 通知，必须首先使用 **MarketBookAdd** 函数订阅接收通知。

若要取消订阅某个交易品种的 **OnBookEvent** 事件，可调用 **MarketBookRelease** 函数。

OnBookEvent 事件是播报的，这意味着图表上有一个 MQL 程序订阅 **OnBookEvent** 事件就足够了，同一图表上的所有其他程序也将开始接收事件，前提是其他程序的代码中有 **OnBookEvent** 处理程序。因此，有必要分析交易品种的名称，其作为参数传递给处理程序。

OnBookEvent 处理程序的原型如下。

```
void OnBookEvent(const string &symbol)
```

OnBookEvent 事件始终会加入队列，即使前一个 **OnBookEvent** 事件的处理尚未完成。

重要的是，**OnBookEvent** 事件仅为通知，并不会提供订单簿的状态。要获得市场深度数据，必须调用 **MarketBookGet** 函数。

但应注意的是，**MarketBookGet** 调用即使是直接从 **OnBookEvent** 处理程序进行，也会在调用 **MarketBookGet** 时接收订单簿的当前状态，该状态不一定与触发 **OnBookEvent** 事件发送的订单簿状态相匹配。若一连串订单簿变更以极快速度到达终端时，可能会发生这种情况。

在这方面，为了获得市场深度变化的完整时间线，需要编写 **OnBookEvent** 的实现，并根据执行速度来确定优化顺序。

同时，在 MQL5 中，无法保证可获得所有唯一的市場深度状态。

如果程序已经开始成功接收通知，但随后当市场开放时通知消失不见（并且分时报价继续出现），可能表明订阅存在问题。特别是，另一个设计不当的 MQL 程序可能会执行超出必要次数的取消订阅操作。在这种情况下，建议在预定义的超时（例如，几十秒或一分钟）后，使用新的 **MarketBookAdd** 调用重新订阅。

一个无缓冲指标 **MarketBookEvent.mq5** 的示例允许你跟踪 **OnBookEvent** 事件的到达，并在注释中打印交易品种名称和当前时间（毫秒系统计数器）。为了清楚起见，我们使用 **Comments.mqh** 文件中的多行注释函数，该文件选自 [在图表窗口中显示消息](#) 一节。

有趣的是，如果你将输入参数 **WorkSymbol** 留空（默认值），该指标本身将不会启动订单簿的订阅，但能够拦截同一图表上其他 MQL 程序请求的消息。我们来检验一下。

```

#include <MQL5Book/Comments.mqh>

input string WorkSymbol = ""; // WorkSymbol (if empty, intercept events initiated by

void OnInit()
{
    if(StringLen(WorkSymbol))
    {
        PRTF(MarketBookAdd(WorkSymbol));
    }
    else
    {
        Print("Start listening to OnBookEvent initiated by other programs");
    }
}

void OnBookEvent(const string &symbol)
{
    ChronoComment(symbol + " " + (string)GetTickCount());
}

void OnDeinit(const int)
{
    Comment("");
    if(StringLen(WorkSymbol))
    {
        PRTF(MarketBookRelease(WorkSymbol));
    }
}

```

让我们使用默认设置 (没有自己的订阅) 运行 **MarketBookEvent**, 然后添加上一节中的 **MarketBookAddRelease** 指标, 并为其指定一个包含可用订单簿的若干交易品种列表 (在以下示例中, 为 "XAUUSD,BTCUSD,USDCNH")。在哪个图表上运行指标并不重要: 可以是一个完全不同的交易品种, 比如欧元兑美元 (EURUSD)。

在启动 **MarketBookEvent** 之后, 该图表将立即变为空白 (没有注释), 因为还没有订阅。一旦 **MarketBookAddRelease** 启动 (日志中应出现三行, 成功订阅的状态等于 **true**), 交易品种名称将随着其订单簿更新开始交替出现在注释中 (我们还没有学习如何读取订单簿; 这方面的内容将在下一节讨论)。

在屏幕上的显示效果如下。



关于“第三方”交易品种订单簿变更的通知

如果我们现在移除 `MarketBookAddRelease` 指标，即可取消订阅，注释将停止更新。随后移除 `MarketBookEvent` 将会清除该注释。

请注意，从请求退订到市场深度事件实际停止更新注释之间会间隔一段时间（一两秒钟）。

你可以在图表上单独运行 `MarketBookEvent` 指标，在其 `WorkSymbol` 参数中指定一些交易品种，以确保通知在同一个应用程序中运行。`MarketBookAddRelease` 以前仅用于演示通知的播报特性。换言之，在一个程序中启用订单簿变更订阅会影响另一个程序中通知的接收。

6.2.3 读取当前的市场深度数据

成功执行 `MarketBookAdd` 函数后，MQL 程序可以在 `OnBookEvent` 事件到达后使用 `MarketBookGet` 函数查询订单簿状态。`MarketBookGet` 函数使用指定交易品种的市场深度值填充通过引用传递的 `MqlBookInfo` 结构体数组。

`bool MarketBookGet(string symbol, MqlBookInfo &book[])`

对于接收数组，可以预先为足够数量的记录分配内存。如果动态数组的大小为零或不足，终端本身将为其分配内存。

该函数返回成功 (`true`) 或错误 (`false`) 的指示。

`MarketBookGet` 通常直接在 `OnBookEvent` 处理程序代码中使用，也可以该函数所调用的函数中使用。

关于市场深度价格水平的单独记录存储在 `MqlBookInfo` 结构体中。

```
struct MqlBookInfo
{
    ENUM_BOOK_TYPE type;           // request type
    double price;                  // price
    long volume;                   // volume
    double volume_real;            // volume with increased accuracy
};
```

枚举 ENUM_BOOK_TYPE 包含以下元素。

标识符	说明
BOOK_TYPE_SELL	限价卖出请求
BOOK_TYPE_BUY	限价买入请求
BOOK_TYPE_SELL_MARKET	市价卖出请求
BOOK_TYPE_BUY_MARKET	市价买入请求

在订单簿中，卖出订单位于其上半部分，买入订单位于下半部分。通常，这样可形成从高价到低价的元素排列顺序。换言之，在索引 0 处是最高价格，最后一个条目为最低价格，中间的价格逐渐降低。在这种情况下，各个价格水平之间的最小价格步长为 `SYMBOL_TRADE_TICK_SIZE`，但是，交易量为零的层级不会被转换，也就是说，相邻元素之间的间隔可能会很大。

在终端用户界面中，订单簿窗口提供了启用/禁用 *Advanced Mode* 的选项，在该模式下，零交易量的层级也会显示出来。但是默认在标准模式下，这些层级是隐藏的（在表中被跳过）。

实践中，订单簿的内容有时会与公布的规则相矛盾。特别是，一些限价买入或卖出请求可能落入订单簿的另一半部分（可能有人以不利的高价买入或以不利的低价卖出，但提供商也可能会出现数据汇总错误）。由于遵守了优先级“所有卖出订单在上方，所有买入订单在下方”，因此会违反订单簿中的价格顺序（参见以下示例）。此外，在订单簿的上半部分和下半部分均可以发现价格（层级）的重复值。

理论上，买入卖出价格在订单簿中间重合是正常情况。这表示零点差。但遗憾的是，在订单簿深度较大时会出现重复的价格层级。

当我们说订单簿的“半区”时，不应从字面上理解。根据流动性，供给和需求层级的数量可能不匹配。总之，订单簿并不是对称的。

MQL 程序必须检查订单簿（尤其是价格排序订单）是否正确，并准备好处理潜在的偏差。

有一些异常情况虽然不算严重，但在算法中仍应该予以考虑，具体包括：

- ⌚连续相同的订单簿（无变化）
- ⌚空白订单簿
- ⌚包含一个层级的订单簿

以下是从一个经纪商获得的真实市场深度片段。字母 ‘S’ 和 ‘B’ 分别代表卖出和买入请求的价格。

请注意，买入和卖出层级实际上是有重叠的：尽管乍一看并不明显，因为订单簿中的所有 ‘S’ 记录都是专门置于上半部分（接收数组的开头），而 ‘B’ 记录置于下半部分（数组的结尾）。但如果

仔细看就会发现：元素 20 和 21 中的买入价格分别为 143.23 和 138.86，高于所有卖出报价。并且，与此同时，元素 18 和 19 中的销售价格为 134.62 和 133.55，低于所有买入报价。

```
...
10 S 138.48 652
11 S 138.47 754
12 S 138.45 2256
13 S 138.43 300
14 S 138.42 14
15 S 138.40 1761
16 S 138.39 670    // Duplicate
17 S 138.11 200
18 S 134.62 420    // Low
19 S 133.55 10627  // Low

20 B 143.23 9564    // High
21 B 138.86 533     // High
22 B 138.39 739     // Duplicate
23 B 138.38 106
24 B 138.31 100
25 B 138.25 29
26 B 138.24 6072
27 B 138.23 571
28 B 138.21 17
29 B 138.20 201
30 B 138.19 1
...
```

此外，138.39 价格同时出现在上半部分（索引号为 16）和下半部分（索引号为 22）。

订单簿中的错误最有可能出现在极端条件下：剧烈波动或缺乏流动性。

我们使用 *MarketBookDisplay.mq5* 指标检查订单簿的接收情况。其用于订阅参数 *WorkSymbol* 中指定交易品种的市场深度事件（如果你在此处留空，则假定为当前图表的工作交易品种）。


```

input string WorkSymbol = ""; // WorkSymbol (if empty, use current chart symbol)

const string _WorkSymbol = StringLen(WorkSymbol) == 0 ? _Symbol : WorkSymbol;
int digits;

void OnInit()
{
    PRTF(MarketBookAdd(_WorkSymbol));
    digits = (int)SymbolInfoInteger(_WorkSymbol, SYMBOL_DIGITS);
    ...
}

void OnDeinit(const int)
{
    Comment("");
    PRTF(MarketBookRelease(_WorkSymbol));
}

```

OnBookEvent 处理程序是在用于处理事件的代码中定义的，其中调用了 *MarketBookGet*，生成的 *MqlBookInfo* 数组的所有元素均作为多行注释输出。

```

void OnBookEvent(const string &symbol)
{
    if(symbol == _WorkSymbol) // take only order books of the requested symbol
    {
        MqlBookInfo mbi[];
        if(MarketBookGet(symbol, mbi)) // getting the current order book
        {
            ...
            int half = ArraySize(mbi) / 2; // estimate of the middle of the order book
            bool correct = true;
            // collect information about levels and volumes in one line (with hyphens)
            string s = "";
            for(int i = 0; i < ArraySize(mbi); ++i)
            {
                s += StringFormat("%02d %s %s %d %g\n", i,
                    (mbi[i].type == BOOK_TYPE_BUY ? "B" :
                     (mbi[i].type == BOOK_TYPE_SELL ? "S" : "?")),
                    DoubleToString(mbi[i].price, digits),
                    mbi[i].volume, mbi[i].volume_real);

                if(i > 0) // look for the middle of the order book as a change in request
                {
                    if(mbi[i - 1].type == BOOK_TYPE_SELL
                       && mbi[i].type == BOOK_TYPE_BUY)
                    {
                        half = i; // this is the middle, because there has been a type char
                    }

                    if(mbi[i - 1].price <= mbi[i].price)
                    {
                        correct = false; // reverse order = data problem
                    }
                }
            }
            Comment(s + (!correct ? "\nINCORRECT BOOK" : ""));
            ...
        }
    }
}

```

由于订单簿变化相当快，追踪注释不是很方便。因此，我们将向该指标添加几个缓冲区，其中我们将把订单簿的两个部分内容显示为直方图：分别为卖出和买入。零柱线对应于形成点差的中心层级。随着条柱线数量的增加，“市场深度”也在增加，也就是说，此处会显示越来越远的价格水平：在上面的直方图中，意味着买入订单的价格较低，而在下面的直方图中，卖出订单的价格较高。

```

#property indicator_separate_window
#property indicator_plots 2
#property indicator_buffers 2

#property indicator_type1    DRAW_HISTOGRAM
#property indicator_color1   clrDodgerBlue
#property indicator_width1   2
#property indicator_label1   "Buys"

#property indicator_type2    DRAW_HISTOGRAM
#property indicator_color2   clrOrangeRed
#property indicator_width2   2
#property indicator_label2   "Sells"

double buys[], sells[];

```

我们能以标准和扩展模式（即，跳过或显示零交易量的价格水平）可视化订单簿，以及以手数或单位的分数显示交易量本身。这两种选择与内置的市场深度窗口功能类似。

```

input bool AdvancedMode = false;
input bool ShowVolumeInLots = false;

```

我们在 *OnInit* 中设置缓冲区并获取一些交易品种特性（稍后我们会用到）。

```

int depth, digits;
double tick, contract;

void OnInit()
{
    ...
    // setting indicator buffers
    SetIndexBuffer(0, buys);
    SetIndexBuffer(1, sells);
    ArraySetAsSeries(buys, true);
    ArraySetAsSeries(sells, true);
    // getting the necessary symbol properties
    depth = (int)PRTF(SymbolInfoInteger(_WorkSymbol, SYMBOL_TICKS_BOOKDEPTH));
    tick = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_TICK_SIZE);
    contract = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_CONTRACT_SIZE);
}

```

我们将缓冲区填充添加到处理程序 *OnBookEvent* 中。

```

#define VOL(V) (ShowVolumeInLots ? V / contract : V)

void OnBookEvent(const string &symbol)
{
    if(symbol == _WorkSymbol) // take only order books of the requested symbol
    {
        MqlBookInfo mbi[];
        if(MarketBookGet(symbol, mbi)) // getting the current order book
        {
            // clear the buffers to the depth with 10 times the margin of the maximum de
            // because extended mode can have a lot of empty elements
            for(int i = 0; i <= depth * 10; ++i)
            {
                buys[i] = EMPTY_VALUE;
                sells[i] = EMPTY_VALUE;
            }
            ...// further along we form and display the comment as before
            if(!correct) return;

            // filling buffers with data
            if(AdvancedMode) // show skips enabled
            {
                for(int i = 0; i < ArraySize(mbi); ++i)
                {
                    if(i < half)
                    {
                        int x = (int)MathRound((mbi[i].price - mbi[half - 1].price) / tick);
                        sells[x] = -VOL(mbi[i].volume_real);
                    }
                    else
                    {
                        int x = (int)MathRound((mbi[half].price - mbi[i].price) / tick);
                        buys[x] = VOL(mbi[i].volume_real);
                    }
                }
            }
            else // standard mode: show only significant elements
            {
                for(int i = 0; i < ArraySize(mbi); ++i)
                {
                    if(i < half)
                    {
                        sells[half - i - 1] = -VOL(mbi[i].volume_real);
                    }
                    else
                    {
                        buys[i - half] = VOL(mbi[i].volume_real);
                    }
                }
            }
        }
    }
}

```

```
}  
}
```

下图演示了在设置 `AdvancedMode=true` 和 `ShowVolumeInLots=true` 的情况下指标的工作方式。



USDCNH 图表上 MarketBookDisplay.mq5 指标中订单簿的内容

买入显示为正值 (顶部的蓝色柱线), 卖出显示为负值 (底部的红色柱线)。为了清楚起见, 右边有一个标准的市场深度窗口, 具有相同的设置 (高级模式下, 交易量以手数显示), 因此你可以确保这些值匹配。

需要注意的是, 该指标的重新绘制速度可能不够快, 无法与内置订单簿保持同步。这并不意味着 MQL 程序没有及时接收到事件, 而只是异步图表呈现的副作用。工作算法通常包括分析处理和订单簿下单, 而不包括可视化。

在这种情况下, 更新图表是在调用 `Comment` 函数时隐式请求的。

6.2.4 在应用算法中使用市场深度数据

市场深度被认为是开发高级交易系统的一项非常实用的技术。特别是, 在贴近市场的层级分析市场交易量的深度分布, 可以让你提前发现特定交易量的平均订单执行价格: 只需反向累加确保成交所需的各价位成交量即可。在交易量不足的清谈市场中, 该算法可能会禁止开仓交易, 以避免价格大幅下滑。

基于市场深度数据, 也可以构建其他策略。例如, 了解大宗交易量所处的价格层级很重要。

MarketBookVolumeAlert.mq5

在下一个测试指标 `MarketBookVolumeAlert.mq5` 中, 我们实现了一个简单的算法来跟踪超过给定值的交易量或其变化情况。

```

#property indicator_chart_window
#property indicator_plots 0

input string WorkSymbol = ""; // WorkSymbol (if empty, use current chart symbol)
input bool CountVolumeInLots = false;
input double VolumeLimit = 0;

const string _WorkSymbol = StringLen(WorkSymbol) == 0 ? _Symbol : WorkSymbol;

```

该指标中没有图形。受控交易品种在 *WorkSymbol* 参数中输入 (如果留空,则表示图表的工作交易品种)。被跟踪对象的最小阈值,即算法的敏感度,在 *VolumeLimit* 参数中指定。根据 *CountVolumeInLots* 参数,对交易量进行分析并以手数 (*true*) 或单位 (*false*) 的形式显示给用户。这也会影响 *VolumeLimit* 值的输入方式。从单位到手数分数的转换是由 VOL 宏提供的: *contract* 中使用的合约规模是在 *OnInit* 中初始化的 (见下文)。

```
#define VOL(V) (CountVolumeInLots ? V / contract : V)
```

如果发现大宗交易量超过该阈值,程序将在注释中显示相应层级的消息。为了保存最新的警告历史,我们使用已知的多行注释类 (*Comments.mqh*)。

```

#define N_LINES 25 // number of lines in the comment buffer
#include <MQL5Book/Comments.mqh>

```

在 *OnInit* 处理程序中,让我们准备必要的设置并订阅 DOM 事件。

```

double contract;
int digits;

void OnInit()
{
    MarketBookAdd(_WorkSymbol);
    contract = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_CONTRACT_SIZE);
    digits = (int)MathRound(MathLog10(contract));
    Print(SymbolInfoDouble(_WorkSymbol, SYMBOL_SESSION_BUY_ORDERS_VOLUME));
    Print(SymbolInfoDouble(_WorkSymbol, SYMBOL_SESSION_SELL_ORDERS_VOLUME));
}

```

如果 *SYMBOL_SESSION_BUY_ORDERS_VOLUME* 和 *SYMBOL_SESSION_SELL_ORDERS_VOLUME* 特性是由你的经纪商为所选交易品种填写的,则这些特性可帮助你确定选择哪个阈值是有意义的。默认情况下, *VolumeLimit* 为 0,这就是为什么订单簿中的任何更改都一定会生成警告的原因。为了筛选掉不显著的波动,建议将 *VolumeLimit* 设置为超过所有层次的交易量的平均规模值 (提前查看内置订单簿或 *MarketBookDisplay.mq5* 指标)。

按常规方式完成最终实现。

```

void OnDeinit(const int)
{
    MarketBookRelease(_WorkSymbol);
    Comment("");
}

```

主要工作由 *OnBookEvent* 处理器完成。该处理器描述了一个静态数组 *MqlBookInfo mbp*, 用于存储订单簿的前一个版本 (自上次函数调用以来)。

```

void OnBookEvent(const string &symbol)
{
    if(symbol != _WorkSymbol) return; // process only the requested symbol

    static MqlBookInfo mbp[];          // previous table/book
    MqlBookInfo mbi[];
    if(MarketBookGet(symbol, mbi)) // read the current book
    {
        if(ArraySize(mbp) == 0) // first time we just save, because nothing to compare
        {
            ArrayCopy(mbp, mbi);
            return;
        }
        ...
    }
}

```

如果有一个旧的和一个新的订单簿，我们在 *i* 和 *j* 的嵌套循环中比较其各个层级上的交易量。请注意，该指数增加意味着价格降低。

```

int j = 0;
for(int i = 0; i < ArraySize(mbi); ++i)
{
    bool found = false;
    for( ; j < ArraySize(mbp); ++j)
    {
        if(MathAbs(mbp[j].price - mbi[i].price) < DBL_EPSILON * mbi[i].price)
        {
            // mbp[j].price == mbi[i].price
            if(VOL(mbi[i].volume_real - mbp[j].volume_real) >= VolumeLimit)
            {
                NotifyVolumeChange("Enlarged", mbp[j].price,
                                    VOL(mbp[j].volume_real), VOL(mbi[i].volume_real));
            }
            else
            if(VOL(mbp[j].volume_real - mbi[i].volume_real) >= VolumeLimit)
            {
                NotifyVolumeChange("Reduced", mbp[j].price,
                                    VOL(mbp[j].volume_real), VOL(mbi[i].volume_real));
            }
            found = true;
            ++j;
            break;
        }
        else if(mbp[j].price > mbi[i].price)
        {
            if(VOL(mbp[j].volume_real) >= VolumeLimit)
            {
                NotifyVolumeChange("Removed", mbp[j].price,
                                    VOL(mbp[j].volume_real), 0.0);
            }
            // continue the loop increasing ++j to lower prices
        }
        else // mbp[j].price < mbi[i].price
        {
            break;
        }
    }
    if(!found) // unique (new) price
    {
        if(VOL(mbi[i].volume_real) >= VolumeLimit)
        {
            NotifyVolumeChange("Added", mbi[i].price, 0.0, VOL(mbi[i].volume_real))
        }
    }
}
...

```

此处，重点不在于水平的类型，而仅在于交易量值。但是，如果你愿意，你可以根据发生重要变更的层级的 *type* 字段，轻松地将指定的买入或卖出添加到通知中。

最后，我们在静态数组 *mbp* 中保存 *mbi* 的一个新副本，以便在下一次函数调用时与其进行比较。


```

    if(ArrayCopy(mbp, mbi) <= 0)
    {
        Print("ArrayCopy failed:", _LastError);
    }
    if(ArrayResize(mbp, ArraySize(mbi)) <= 0) // shrink if needed
    {
        Print("ArrayResize failed:", _LastError);
    }
}
}

```

`ArrayCopy` 不会自动缩小动态目标数组 (即使其大小大于源数组), 因此我们需使用 `ArrayResize` 来设置确切大小。

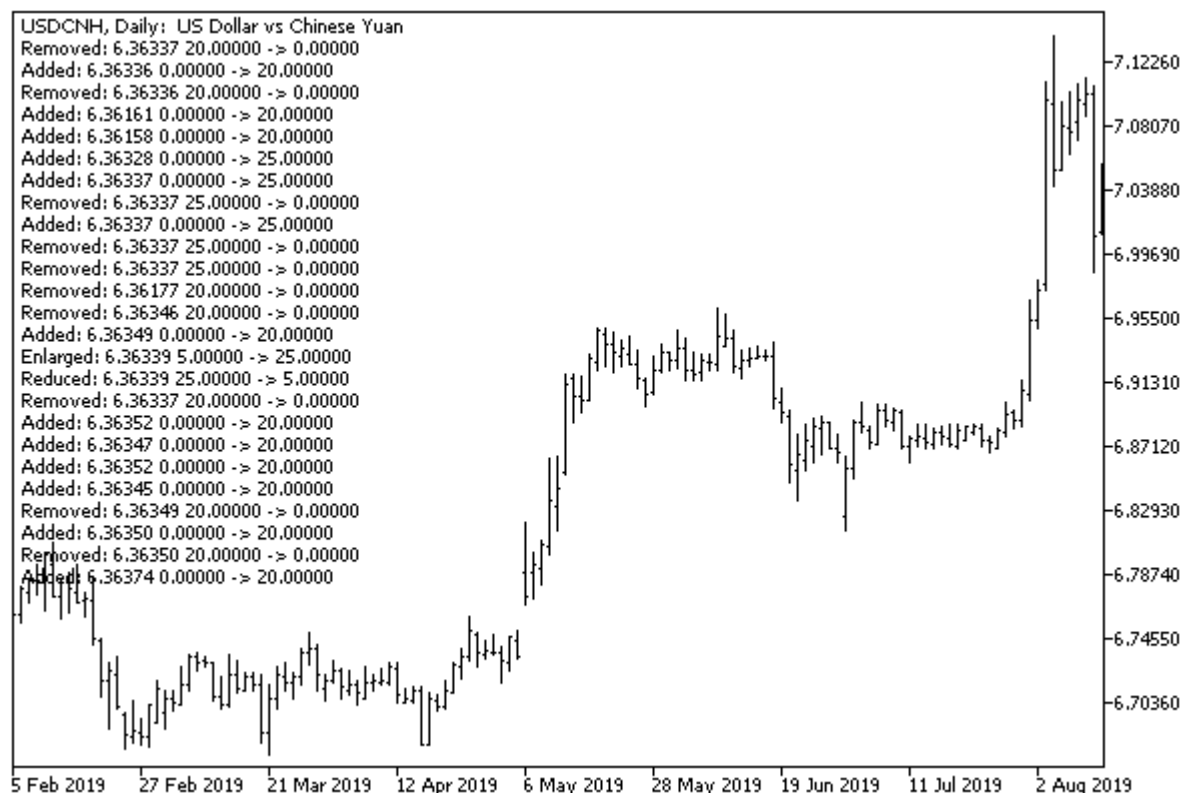
辅助函数 `NotifyVolumeChange` 仅用于将关于所发现的变化信息添加到注释中。

```

void NotifyVolumeChange(const string action, const double price,
    const double previous, const double volume)
{
    const string message = StringFormat("%s: %s %s -> %s",
        action,
        DoubleToString(price, (int)SymbolInfoInteger(_WorkSymbol, SYMBOL_DIGITS)),
        DoubleToString(previous, digits),
        DoubleToString(volume, digits));
    ChronoComment(message);
}

```

下图显示了设置 `CountVolumeInLots=false` 和 `VolumeLimit=20` 时该指标的结果。



关于订单簿中交易量变化的通知

MarketBookQuasiTicks.mq5

作为订单簿可能用途的第二个示例，让我们来看获取多货币分时报价的问题。我们已经在 [自定义事件的生成](#) 一节中提到了该问题，并且在其中看到了一种可能的解决方案和指标 *EventTickSpy.mq5*。现在，在熟悉了市场深度 API 之后，我们可以实现一个替代方案。

我们来创建一个指标 *MarketBookQuasiTicks.mq5*，其将订阅给定金融工具列表的订单簿，并在其中找到最佳报价和需求价格，即点差周围的价格对，这些价格仅为 *Ask* 和 *Bid* 价格。

当然，这些信息并不完全等同于标准分时报价（注意，交易/分时报价和订单簿流量可能来自完全不同的提供商），但提供了对市场充分和及时的了解。

各个交易品种的新价格值将显示在多行注释中。

工作交易品种列表在 *SymbolList* 输入参数中指定为逗号分隔列表。启用和禁用对市场深度事件的订阅是在 *OnInit* 和 *OnDeinit* 处理程序中完成的。

```
#define N_LINES 25 // number of lines in the comment buffer
#include <MQL5Book/Comments.mqh>

input string SymbolList = "EURUSD,GBPUSD,XAUUSD,USDJPY"; // SymbolList (comma,separat

const string WorkSymbols = StringLen(SymbolList) == 0 ? _Symbol : SymbolList;
string symbols[];

void OnInit()
{
    const int n = StringSplit(WorkSymbols, ',', symbols);
    for(int i = 0; i < n; ++i)
    {
        if(!MarketBookAdd(symbols[i]))
        {
            PrintFormat("MarketBookAdd(%s) failed with code %d", symbols[i], _LastError)
        }
    }
}

void OnDeinit(const int)
{
    for(int i = 0; i < ArraySize(symbols); ++i)
    {
        if(!MarketBookRelease(symbols[i]))
        {
            PrintFormat("MarketBookRelease(%s) failed with code %d", symbols[i], _LastEr
        }
    }
    Comment("");
}
```

每个新订单簿的分析都在 *OnBookEvent* 中进行。

```

void OnBookEvent(const string &symbol)
{
    MqlBookInfo mbi[];
    if(MarketBookGet(symbol, mbi)) // getting the current order book
    {
        int half = ArraySize(mbi) / 2; // estimate the middle of the order book
        bool correct = true;
        for(int i = 0; i < ArraySize(mbi); ++i)
        {
            if(i > 0)
            {
                if(mbi[i - 1].type == BOOK_TYPE_SELL
                    && mbi[i].type == BOOK_TYPE_BUY)
                {
                    half = i; // specify the middle of the order book
                }

                if(mbi[i - 1].price <= mbi[i].price)
                {
                    correct = false;
                }
            }
        }

        if(correct) // retrieve the best Bid/Ask prices from the correct order book
        {
            // mbi[half - 1].price // Ask
            // mbi[half].price      // Bid
            OnSymbolTick(symbol, mbi[half].price);
        }
    }
}

```

找到的市场 *Ask/Bid* 价格传递给辅助函数 *OnSymbolTick*, 以在注释中显示。

```

void OnSymbolTick(const string &symbol, const double price)
{
    const string message = StringFormat("%s %s",
        symbol, DoubleToString(price, (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS)));
    ChronoComment(message);
}

```

如果需要, 你可以验证我们的合成分时报价与标准分时报价没有太大差异。

以下为关于收到的模拟分时报价信息在图表上的呈现效果。



基于订单簿事件的多交易品种模拟分时报价

同时，需要再次注意的是，订单簿事件仅在平台上在线可用，在 [测试程序](#) 中不可用。如果交易系统完全建立在订单簿的模拟分时报价上，其测试将需要使用第三方解决方案，以确保在测试程序中收集并回放订单簿。

6.3 交易账户信息

在这一章中，我们将研究 MQL 程序交易环境的最后一个重要方面，特别是 EA 交易，我们将在接下来的几章中详细阐述。先介绍交易账户。

拥有一个有效的账户和一个有效的连接是运行大多数 MQL 程序的必要条件。迄今为止，我们还没有关注这一点，但是获得报价、分时报价以及能够打开可用图表即意味着已成功连接到交易账户。

在 EA 交易的情况下，账户还会反映客户的财务状况，积累交易历史，并确定允许交易的具体模式。

MQL5 API 允许你获取账户的特性，从账号到当前利润都可以查询。这些特性在终端中都是只读的，由经纪商在服务器上设置。

终端一次只能连接到一个账户。所有 MQL 程序都使用这个账户。正如我们在 [各类程序启动和停止的特征](#) 一节中所提到的，切换账户会重新加载图表所附带的指标和 EA 交易。但在 `OnDeinit` 处理程序中，该程序可以找到取消初始化的原因，当切换账户时，该原因将等于 `REASON_ACCOUNT`。

6.3.1 获取账户特性的函数概述

根据类型，完整的账户特性在逻辑上可分为三组。字符串特性在 `ENUM_ACCOUNT_INFO_STRING` 枚举中汇总，可通过 `AccountInfoString` 函数查询。实型特性组合在 `ENUM_ACCOUNT_INFO_DOUBLE` 枚举中，`AccountInfoDouble` 函数可以处理这些特性。`AccountInfoInteger` 函数中使用的 `ENUM_ACCOUNT_INFO_INTEGER` 枚举包含整数和布尔特性的标识符（标志），以及几个应用的 `ENUM_ACCOUNT_INFO` 枚举。

`double AccountInfoDouble (ENUM_ACCOUNT_INFO_DOUBLE property)`

`long AccountInfoInteger (ENUM_ACCOUNT_INFO_INTEGER property)`

`string AccountInfoString (ENUM_ACCOUNT_INFO_STRING property)`

我们创建了 `AccountMonitor` 类 (`AccountMonitor.mqh`) 来简化特性的读取。通过重载 `get` 方法，该类可根据参数中传递的特定枚举元素来提供所需 API 函数的自动调用。

```
class AccountMonitor
{
public:
    long get(const ENUM_ACCOUNT_INFO_INTEGER property) const
    {
        return AccountInfoInteger(property);
    }

    double get(const ENUM_ACCOUNT_INFO_DOUBLE property) const
    {
        return AccountInfoDouble(property);
    }

    string get(const ENUM_ACCOUNT_INFO_STRING property) const
    {
        return AccountInfoString(property);
    }

    long get(const int property, const long) const
    {
        return AccountInfoInteger((ENUM_ACCOUNT_INFO_INTEGER)property);
    }

    double get(const int property, const double) const
    {
        return AccountInfoDouble((ENUM_ACCOUNT_INFO_DOUBLE)property);
    }

    string get(const int property, const string) const
    {
        return AccountInfoString((ENUM_ACCOUNT_INFO_STRING)property);
    }
    ...
}
```

此外，还具有 `stringify` 方法的几个重载，这些重载形成了特性值的用户友好的字符串表示形式（特别是，它对于应用的枚举非常有用，否则这些枚举将显示为无信息的数字）。每个特性的特性将在后面的章节中讨论。

```

static string boolean(const long v)
{
    return v ? "true" : "false";
}

template<typename E>
static string enumstr(const long v)
{
    return EnumToString((E)v);
}

// "decode" properties according to subtype inside integer values
static string stringify(const long v, const ENUM_ACCOUNT_INFO_INTEGER property)
{
    switch(property)
    {
        case ACCOUNT_TRADE_ALLOWED:
        case ACCOUNT_TRADE_EXPERT:
        case ACCOUNT_FIFO_CLOSE:
            return boolean(v);
        case ACCOUNT_TRADE_MODE:
            return enumstr<ENUM_ACCOUNT_TRADE_MODE>(v);
        case ACCOUNT_MARGIN_MODE:
            return enumstr<ENUM_ACCOUNT_MARGIN_MODE>(v);
        case ACCOUNT_MARGIN_SO_MODE:
            return enumstr<ENUM_ACCOUNT_STOPOUT_MODE>(v);
    }

    return (string)v;
}

string stringify(const ENUM_ACCOUNT_INFO_INTEGER property) const
{
    return stringify(AccountInfoInteger(property), property);
}

string stringify(const ENUM_ACCOUNT_INFO_DOUBLE property, const string format = NU
{
    if(format == NULL) return DoubleToString(AccountInfoDouble(property),
        (int)get(ACCOUNT_CURRENCY_DIGITS));
    return StringFormat(format, AccountInfoDouble(property));
}

string stringify(const ENUM_ACCOUNT_INFO_STRING property) const
{
    return AccountInfoString(property);
}

...

```

最后，还有一个模板方法 *list2log*，该方法允许获得关于账户的全面信息。

```
// list of names and values of all properties of enum type E
template<typename E>
void list2log()
{
    E e = (E)0; // suppress warning 'possible use of uninitialized variable'
    int array[];
    const int n = EnumToArray(e, array, 0, USHORT_MAX);
    Print(typename(E), " Count=", n);
    for(int i = 0; i < n; ++i)
    {
        e = (E)array[i];
        PrintFormat("% 3d %s=%s", i, EnumToString(e), stringify(e));
    }
}
};
```

我们将在下一节中测试新类的实际运行效果。

6.3.2 识别账户、客户、服务器和经纪商

账户最重要的特性是其账号和识别数据：服务器和经纪商公司的名称，以及客户的名称。除了账号，所有这些特性均为字符串特性。

标识符	说明
ACCOUNT_LOGIN	账号（长整型）
ACCOUNT_NAME	客户名称
ACCOUNT_SERVER	交易服务器名称
ACCOUNT_COMPANY	为账户提供服务的公司名称

我们使用上一节中的 *AccountMonitor* 类来记录这些特性以及稍后将讨论的许多其他特性。我们创建相应的对象，并在脚本 *AccountInfo.mq5* 中调用其特性。

```
#include <MQL5Book/AccountMonitor.mqh>

void OnStart()
{
    AccountMonitor m;
    m.list2log<ENUM_ACCOUNT_INFO_INTEGER>();
    m.list2log<ENUM_ACCOUNT_INFO_DOUBLE>();
    m.list2log<ENUM_ACCOUNT_INFO_STRING>();
}
```

以下是该脚本可能结果的一个示例。

```

ENUM_ACCOUNT_INFO_INTEGER Count=10
0 ACCOUNT_LOGIN=30000003
1 ACCOUNT_TRADE_MODE=ACCOUNT_TRADE_MODE_DEMO
2 ACCOUNT_TRADE_ALLOWED=true
3 ACCOUNT_TRADE_EXPERT=true
4 ACCOUNT_LEVERAGE=100
5 ACCOUNT_MARGIN_SO_MODE=ACCOUNT_STOPOUT_MODE_PERCENT
6 ACCOUNT_LIMIT_ORDERS=200
7 ACCOUNT_MARGIN_MODE=ACCOUNT_MARGIN_MODE_RETAIL_HEDGING
8 ACCOUNT_CURRENCY_DIGITS=2
9 ACCOUNT_FIFO_CLOSE=false
ENUM_ACCOUNT_INFO_DOUBLE Count=14
0 ACCOUNT_BALANCE=10000.00
1 ACCOUNT_CREDIT=0.00
2 ACCOUNT_PROFIT=-78.76
3 ACCOUNT_EQUITY=9921.24
4 ACCOUNT_MARGIN=1000.00
5 ACCOUNT_MARGIN_FREE=8921.24
6 ACCOUNT_MARGIN_LEVEL=992.12
7 ACCOUNT_MARGIN_SO_CALL=50.00
8 ACCOUNT_MARGIN_SO_SO=30.00
9 ACCOUNT_MARGIN_INITIAL=0.00
10 ACCOUNT_MARGIN_MAINTENANCE=0.00
11 ACCOUNT_ASSETS=0.00
12 ACCOUNT_LIABILITIES=0.00
13 ACCOUNT_COMMISSION_BLOCKED=0.00
ENUM_ACCOUNT_INFO_STRING Count=4
0 ACCOUNT_NAME=Vincent Silver
1 ACCOUNT_COMPANY=MetaQuotes Software Corp.
2 ACCOUNT_SERVER=MetaQuotes-Demo
3 ACCOUNT_CURRENCY=USD

```

注意本节中的特性 ACCOUNT_LOGIN、ACCOUNT_NAME、ACCOUNT_COMPANY 和 ACCOUNT_SERVER)。在本例中，脚本是在演示服务器账户“MetaQuotes-Demo”上执行的。显然，这应是一个演示账户，不仅是服务器的名称表明了这一点，另一个特性 ACCOUNT_TRADE_MODE 也表明了这一点，将在下一节中讨论。

账户标识符通常用于将 MQL 程序与特定的交易环境关联起来。这种算法的一个示例在 服务一节中给出。

6.3.3 账户类型：真实、模拟或竞赛

MetaTrader 5 支持多种可以为客户开立的账户类型。ACCOUNT_TRADE_MODE 特性是 ENUM_ACCOUNT_INFO_INTEGER 的一部分，可用于确定当前的账户类型。ENUM_ACCOUNT_TRADE_MODE 枚举中描述了此特性的可能值。

标识符	说明
ACCOUNT_TRADE_MODE_DEMO	模拟交易账户
ACCOUNT_TRADE_MODE_CONTEST	竞赛交易账户
ACCOUNT_TRADE_MODE_REAL	真实交易账户

该特性便于构建 MQL 程序的模拟 (免费) 版本。全功能的付费版本可能需要链接到一个账号, 并且必须是真实账户。

正如我们在上一节运行 *AccountInfo.mq5* 脚本的示例中看到的那样, “MetaQuotes-Demo” 服务器上的账户属于 ACCOUNT_TRADE_MODE_DEMO 类型。

6.3.4 账户货币

结余、盈利、保证金、佣金和其他 **财务指标** 最终总会转换为账户货币, 即使某些交易规范要求以其他货币结算, 例如, 以外汇对的保证金货币结算。

MQL5 API 提供了两个描述账户货币的特性: 其名称和表示精度, 即最小度量单位的大小 (比如美分)。

标识符	说明
ACCOUNT_CURRENCY	存款货币 (字符串)
ACCOUNT_CURRENCY_DIGITS	准确显示交易结果所需账户货币的小数位数 (整数)

例如, 对于用于测试 *AccountInfo* 脚本的模拟账户 (如 [账户识别](#) 一节中所述), ACCOUNT_CURRENCY 特性为 “USD”, ACCOUNT_CURRENCY_DIGITS 的精度为两位小数。我们已经在 *stringify* 方法中的 *AccountMonitor* 类中为 *double* 类型的值使用了 ACCOUNT_CURRENCY_DIGITS 特性 (在账户特性中, 它们均与货币相关)。

6.3.5 账户类型: 净额结算或对冲

MetaTrader 5 支持几种类型的账户, 特别是 **净额结算**和**对冲**。对于净额结算, 每个交易品种只允许有一个 **头寸**。对于对冲, 你可以为一个交易品种开立多个头寸, 包括多向头寸。订单、交易和头寸将在后面的章节中详细讨论。

MQL 程序通过使用 *AccountInfoInteger* 函数查询 ACCOUNT_MARGIN_MODE 特性来确定账户类型。从特性名称可以看出, 其不仅描述了账户类型, 还描述了保证金计算模式。其可能值在 ENUM_ACCOUNT_MARGIN_MODE 枚举中指定。

标识符	说明
ACCOUNT_MARGIN_MODE_RETAIL_NETTING	OTC 市场，考虑净额结算模式下的头寸。保证金计算基于 SYMBOL_TRADE_CALC_MODE 特性。
ACCOUNT_MARGIN_MODE_EXCHANGE	外汇市场，考虑净额结算模式下的头寸。保证金是根据交易所的规则计算的，但经纪商可以在工具设置中指定折扣。
ACCOUNT_MARGIN_MODE_RETAIL_HEDGING	OTC 市场，在对冲模式下独立考虑头寸。保证金计算基于 SYMBOL_TRADE_CALC_MODE 交易品种特性，同时考虑对冲保证金 SYMBOL_MARGIN_HEDGED 。

例如，运行 `AccountInfo` 脚本 (在 [账户识别](#) 一节中)，显示账户类型为 `ACCOUNT_MARGIN_MODE_RETAIL_HEDGING`。

6.3.6 账户操作的限制和权限

在账户的特性中，有一些是对交易操作的限制，包括完全禁用交易。所有这些特性都属于 `ENUM_ACCOUNT_INFO_INTEGER` 枚举，并且为布尔标志，`ACCOUNT_LIMIT_ORDERS` 除外。

标识符	说明
ACCOUNT_TRADE_ALLOWED	允许在当前账户上进行交易
ACCOUNT_TRADE_EXPERT	允许使用 EA 交易和脚本进行算法交易
ACCOUNT_LIMIT_ORDERS	有效挂单的最大允许数量
ACCOUNT_FIFO_CLOSE	仅根据 FIFO 规则进行平仓的要求

由于本书为关于 `MQL5` 编程的指南，其中包括算法交易，所以需要注意的是，当 `ACCOUNT_TRADE_ALLOWED` 等于 `false` 时，将全面禁止交易，而禁用 `ACCOUNT_TRADE_EXPERT` 权限的重要性与之相当。经纪商可禁止使用 EA 交易和脚本进行交易，但允许手动交易。

如果使用投资密码连接到账户，`ACCOUNT_TRADE_ALLOWED` 特性通常等于 `false`。

如果 `ACCOUNT_FIFO_CLOSE` 特性值为 `true`，则每个交易品种的头寸只能按其开仓时的相同顺序平仓，即首先平仓最旧的订单，然后平仓较新的订单，依此类推，直到最后一个订单。如果试图以不同顺序平仓，则会收到错误。对于没有头寸对冲的账户，即，如果 `ACCOUNT_MARGIN_MODE` 特性不等于 `ACCOUNT_MARGIN_MODE_RETAIL_HEDGING`，则 `ACCOUNT_FIFO_CLOSE` 特性始终为 `false`。

在 [权限](#) 和 [交易和报价时段时间表](#) 章节中，我们已经开始开发一个类来检测 MQL 程序可用的交易操作。现在我们可以用账户权限检查作为其补充，并将其完善为最终版本 (`Permissions.mqh`) 中。

`TRADE_RESTRICTIONS` 枚举中提供了限制级别，在添加了两个与账户特性相关的新元素后，该枚举采用以下形式。

```

class Permissions
{
    enum TRADE_RESTRICTIONS
    {
        NO_RESTRICTIONS = 0,
        TERMINAL_RESTRICTION = 1, // user's restriction for all programs
        PROGRAM_RESTRICTION = 2,  // user's restriction for a specific program
        SYMBOL_RESTRICTION = 4,   // the symbol is not traded according to the specific
        SESSION_RESTRICTION = 8,  // the market is closed according to the session sche
        ACCOUNT_RESTRICTION = 16, // investor password or broker restriction
        EXPERTS_RESTRICTION = 32, // broker restricted algorithmic trading
    };
    ...
}

```

在检查过程中, 由于各种原因, MQL 程序可能会检测到若干限制, 因此这些元素由单独的位进行编码。最终结果可以代表它们的叠加状态。

最后两个限制刚好与新特性相对应, 并在 *getTradeRestrictionsOnAccount* 方法中设置。检测到的限制的通用位掩码 (如果有的话) 在 *lastRestrictionBitMask* 变量中形成。

```

private:
    static uint lastRestrictionBitMask;
    static bool pass(const uint bitflag)
    {
        lastRestrictionBitMask |= bitflag;
        return lastRestrictionBitMask == 0;
    }

public:
    static uint getTradeRestrictionsOnAccount()
    {
        return (AccountInfoInteger(ACCOUNT_TRADE_ALLOWED) ? 0 : ACCOUNT_RESTRICTION)
            | (AccountInfoInteger(ACCOUNT_TRADE_EXPERT) ? 0 : EXPERTS_RESTRICTION);
    }

    static bool isTradeOnAccountEnabled()
    {
        lastRestrictionBitMask = 0;
        return pass(getTradeRestrictionsOnAccount());
    }
    ...
}

```

如果调用代码与限制的原因无关, 而只需要确定执行交易操作的可能性, 那么使用 *isTradeOnAccountEnabled* 方法会更方便, 该方法可返回一个布尔符号 (*true/false*)。

交易品种和终端特性的检查已根据类似的原则进行了重新组织。例如, *getTradeRestrictionsOnSymbol* 方法包含以前版本的类中已经熟悉的源代码 (检查交易品种的交易会话和交易模式), 但返回一个标志掩码。如果至少有一位被设置, 则该方法描述限制的来源。

```

static uint getTradeRestrictionsOnSymbol(const string symbol, datetime now = 0,
    const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
    if(now == 0) now = TimeTradeServer();
    bool found = false;
    // checking the symbol trading sessions and setting 'found' to 'true',
    // if the 'now' time is inside one of the sessions
    ...

    // in addition to sessions, check the trading mode
    const ENUM_SYMBOL_TRADE_MODE m = (ENUM_SYMBOL_TRADE_MODE)SymbolInfoInteger(symbol,
    return (found ? 0 : SESSION_RESTRICTION)
        | (((m & mode) != 0) || (m == SYMBOL_TRADE_MODE_FULL) ? 0 : SYMBOL_RESTRICTI
}

static bool isTradeOnSymbolEnabled(const string symbol, const datetime now = 0,
    const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
    lastRestrictionBitMask = 0;
    return pass(getTradeRestrictionsOnSymbol(symbol, now, mode));
}
...

```

最后，在 *getTradeRestrictions* 和 *isTradeEnabled* 方法中执行对所有潜在“实例”的一般检查，包括（除了前面的水平之外）终端和程序的设置。

```

static uint getTradeRestrictions(const string symbol = NULL, const datetime now =
    const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
    return (TerminalInfoInteger(TERMINAL_TRADE_ALLOWED) ? 0 : TERMINAL_RESTRICTION)
        | (MQLInfoInteger(MQL_TRADE_ALLOWED) ? 0 : PROGRAM_RESTRICTION)
        | getTradeRestrictionsOnSymbol(symbol == NULL ? _Symbol : symbol, now, mode)
        | getTradeRestrictionsOnAccount();
}

static bool isTradeEnabled(const string symbol = NULL, const datetime now = 0,
    const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
    lastRestrictionBitMask = 0;
    return pass(getTradeRestrictions(symbol, now, mode));
}

```

脚本 *AccountPermissions.mq5* 演示了使用新类对交易权限全面检查。

```
#include <MQL5Book/Permissions.mqh>

void OnStart()
{
    PrintFormat("Run on %s", _Symbol);
    if(!Permissions::isTradeEnabled()) // checking for current character, default
    {
        Print("Trade is disabled for the following reasons:");
        Print(Permissions::explainLastRestrictionBitMask());
    }
    else
    {
        Print("Trade is enabled");
    }
}
```

如果发现了限制，可以使用 `explainLastRestrictionBitMask` 方法以清晰的字符串表示形式显示其位掩码。

以下是一些脚本的结果。在前两种情况下，交易在终端的全局设置中被禁用（特性 `TERMINAL_TRADE_ALLOWED` 和 `MQL_TRADE_ALLOWED` 等于 *false*，对应于 `TERMINAL_RESTRICTION` 和 `PROGRAM_RESTRICTION` 位）。

当在市场关闭期间在 `USDRUB` 上运行时，我们还会收到 `SESSION_RESTRICTION`：

```
Trade is disabled for USDRUB following reasons:
TERMINAL_RESTRICTION PROGRAM_RESTRICTION SESSION_RESTRICTION
```

对于完全禁止交易的交易品种 `SP500m`，会出现 `SYMBOL_RESTRICTION` 标志。

```
Trade is disabled for SP500m following reasons:
TERMINAL_RESTRICTION PROGRAM_RESTRICTION SYMBOL_RESTRICTION SESSION_RESTRICTION
```

最后一点，如果已经允许在终端中交易，但已经使用投资者的密码登录账户，我们将在任何交易品种上看到 `ACCOUNT_RESTRICTION`。

```
Run on XAUUSD
Trade is disabled for following reasons:
ACCOUNT_RESTRICTION
```

MQL 程序中的早期权限检查有助于避免连续发送交易指令失败的情况。

6.3.7 账户保证金设置

对于交易机器人而言，控制被冻结的保证金金额和可用于确保新交易的金额非常重要。特别是，如果没有足够的可用资金，程序将无法执行交易。当维持未平仓的亏存头寸时，首先会收到一个追加保证金的通知，如果未及时追加保证金，经纪商会强行平仓（止损离场）。所有关联的账户特性都包含在 `ENUM_ACCOUNT_INFO_DOUBLE` 枚举中。

标识符	说明
ACCOUNT_MARGIN	账户中以存款货币表示的当前预留保证金
ACCOUNT_MARGIN_FREE	账户中以存款货币表示的当前可用保证金，可用于开立头寸
ACCOUNT_MARGIN_LEVEL	账户的保证金水平，以百分比表示（净值/保证金*100）
ACCOUNT_MARGIN_SO_CALL	要求补充账户的最低保证金水平（追加保证金通知）
ACCOUNT_MARGIN_SO_SO	亏损最多的头寸被强制平仓（止损离场）的最低保证金水平
ACCOUNT_MARGIN_INITIAL	账户上预留的资金，用于为所有挂单提供保证金
ACCOUNT_MARGIN_MAINTENANCE	账户上预留的资金，用于为所有未平仓头寸提供最低要求的保证金

ACCOUNT_MARGIN_SO_CALL 和 ACCOUNT_MARGIN_SO_SO 以百分比或存款货币表示，具体取决于 ACCOUNT_MARGIN_SO_MODE 设置（详见下文）。该特性包含在 ENUM_ACCOUNT_INFO_INTEGER 枚举中，可以衡量用于追加保证金或止损离场的保证金阈值。此外，总杠杆（用于计算某些类型金融工具的保证金）也在此处显示。

标识符	说明
ACCOUNT_LEVERAGE	杠杆倍数
ACCOUNT_MARGIN_SO_MODE	设置最低允许保证金水平的模式，取值来自 ENUM_ACCOUNT_STOPOUT_MODE 枚举

以下是 ENUM_ACCOUNT_STOPOUT_MODE 枚举的元素。

标识符	说明
ACCOUNT_STOPOUT_MODE_PERCENT	水平设置为百分比
ACCOUNT_STOPOUT_MODE_MONEY	水平以账户货币设置

例如，对于 ACCOUNT_STOPOUT_MODE_PERCENT 选项，应根据净值与 ACCOUNT_MARGIN 特性值的比率检查指定百分比（追加保证金通知或止损离场）：

```
AccountInfoDouble(ACCOUNT_EQUITY) / AccountInfoDouble(ACCOUNT_MARGIN) * 100
> AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)
```

在下一节中，你将找到关于 ACCOUNT_EQUITY 特性和该账户其他财务指标的更多详细信息。

但是，ACCOUNT_MARGIN_LEVEL 特性中已经提供了以百分比表示的当前保证金水平。使用 *AccountInfo.mq5* 脚本很容易检查这一点，该脚本记录了所有账户特性，包括上述特性。

我们已经在 [账户识别](#) 一节中运行了该脚本。当时，开立了一个头寸（1 手 USDRUB，等于 100,000 美元），财务状况如下：

```
0 ACCOUNT_BALANCE=10000.00
1 ACCOUNT_CREDIT=0.00
2 ACCOUNT_PROFIT=-78.76
3 ACCOUNT_EQUITY=9921.24
4 ACCOUNT_MARGIN=1000.00
5 ACCOUNT_MARGIN_FREE=8921.24
6 ACCOUNT_MARGIN_LEVEL=992.12
7 ACCOUNT_MARGIN_SO_CALL=50.00
8 ACCOUNT_MARGIN_SO_SO=30.00
```

保证金为 1000.00 美元，很容易检查到账户的杠杆 ACCOUNT_LEVERAGE 确实是 100（根据 [外汇保证金](#) 和 [保证金费率](#) 的计算公式，等于 1.0）。保证金金额不需要按当前汇率转换为账户货币，因为它与该金融工具的基础货币相同。

要在 ACCOUNT_MARGIN_LEVEL 中得到 992.12，只需将 9921.24 除以 1000.00，再乘以 100% 即可。

然后，另 1 手头寸被开立，报价朝着不利的方向发展，导致情况发生了变化：

```
0 ACCOUNT_BALANCE=10000.00
1 ACCOUNT_CREDIT=0.00
2 ACCOUNT_PROFIT=-1486.07
3 ACCOUNT_EQUITY=8513.93
4 ACCOUNT_MARGIN=2000.00
5 ACCOUNT_MARGIN_FREE=6513.93
6 ACCOUNT_MARGIN_LEVEL=425.70
```

我们可以看到 ACCOUNT_PROFIT 列出现了亏损，净值 ACCOUNT_EQUITY 相应减少。保证金 ACCOUNT_MARGIN 从 1000 按比例增加到 2000，可用保证金和保证金水平下降（但仍远未达到 50% 和 30% 的限制）。同样，水平 425.70 是作为计算表达式 $8513.93 / 2000.00 * 100$ 的结果而获得的。

在开仓前用这个公式计算期货保证金水平更实用。在这种情况下，有必要通过额外的保证金 X 来增加现有保证金金额。此外，如果市场进入交易涉及即时佣金扣除 C ，那么，严格地说，也应将其考虑在内（尽管通常佣金远远少于保证金，并且可以忽略不计，加上 API 没有提供在执行交易之前找出佣金的方法：只能通过交易历史中已经完成的交易佣金进行估算）。

$$(\text{AccountInfoDouble}(\text{ACCOUNT\_EQUITY}) - C) / (\text{AccountInfoDouble}(\text{ACCOUNT\_MARGIN}) + X) * 100 > \text{AccountInfoDouble}(\text{ACCOUNT\_MARGIN\_SO\_CALL})$$

稍后，我们将学习如何获得 X 值（使用 [OrderCalcMargin](#) 函数），但除此之外，可能还需要根据 [保证金要求](#) 一节公布的规则进行调整，特别是考虑到可能的 [头寸对冲](#)、[折扣](#) 和 [保证金调整](#)。

对于以货币设置保证金限额的选项（ACCOUNT_STOPOUT_MODE_MONEY），对资金是否充足的检查方式必须不同。

$$\text{AccountInfoDouble}(\text{ACCOUNT\_EQUITY}) > \text{AccountInfoDouble}(\text{ACCOUNT\_MARGIN\_SO\_CALL})$$

此处省略了佣金。请注意，当前准备开立的新头寸的保证金 X 不会以任何方式影响期货保证金的评估。

但无论如何，建议避免过度使用存款，以免不等式条件勉强满足。ACCOUNT_MARGIN_SO_CALL 和 ACCOUNT_MARGIN_SO_SO 的值相当接近，虽然 ACCOUNT_MARGIN_SO_CALL 水平的保证金只是

对交易者的一个警告，但是很容易触发强制平仓。这就是公式使用 `ACCOUNT_MARGIN_SO_CALL` 特性的原因。

6.3.8 账户的当前财务业绩

MQL5 API 允许通过其主要财务指标来控制多个账户特性。这些特性都包含在 `ENUM_ACCOUNT_INFO_DOUBLE` 枚举中。

标识符	说明
<code>ACCOUNT_BALANCE</code>	以存款货币表示的账户余额
<code>ACCOUNT_PROFIT</code>	以存款货币表示的账户当前盈利额
<code>ACCOUNT_EQUITY</code>	以存款货币表示的账户净值
<code>ACCOUNT_CREDIT</code>	经纪商以存款货币提供的信用额度
<code>ACCOUNT_ASSETS</code>	账户上的当前资产金额
<code>ACCOUNT_LIABILITIES</code>	账户上的当前负债额
<code>ACCOUNT_COMMISSION_BLOCKED</code>	账户上被冻结佣金的当前金额

在前面的章节中，我们看到了在不同条件下运行 `AccountInfo.mq5` 脚本时这些特性值的示例。尝试比较不同账户的这些特性。

在交易过程中，我们主要对前三个特性感兴趣：结余、盈利（如果值为负，则为亏损）和净值，它们共同涵盖了账户余额、信用、利润和管理费用（掉期和佣金）。

佣金可根据经纪商的设置以不同方式处理。如果佣金是在 **交易** 时立即从账户余额中扣除并反映在交易特性中，则账户特性 `ACCOUNT_COMMISSION_BLOCKED` 将等于 0。但是，如果佣金计算推迟到期末（例如，一天或一个月结束时），则该特性中将显示为佣金预留的金额。然后，当期末确定了最终佣金金额并从结余中扣除时，该特性将被重置。

通常，只为交易所交易填充 `ACCOUNT_ASSETS` 和 `ACCOUNT_LIABILITIES` 特性。它们反映了证券多头和空头头寸的当前价值。

6.4 创建 EA 交易

在本章中，我们开始学习用于实现 EA 交易的 MQL5 交易 API。就无错编码和涉及的技术数量和种类而言，这种类型的程序可能是最复杂且要求最高的。特别是，我们需要利用从前几章学到的许多技能，从面向对象编程（OOP）到使用图形对象、指标、交易品种和软件环境设置等的实际应用层面。

根据所选择的交易策略，EA 交易开发人员可能需要特别注意以下几点：

- ① 决策和订单发送速度（针对高频交易（HFT））
- ① 根据工具的相关性和波动性选择金融工具的最佳组合（针对集群交易）
- ① 动态计算订单之间的手数和距离（针对策略和网格策略）
- ① 分析新闻或外部数据源（将在本书的第七章讨论）

开发人员应将所有这些特性以最佳方式应用于 MQL5 API 提供的交易机制中（之前已经介绍过这些机制）。

接下来，我们将详细考虑用于管理交易活动的内置函数、EA 交易事件模型及特定的数据结构体，并回忆终端和服务器之间交互的基本原则，以及 MetaTrader 5 中算法交易的基本概念：订单、交易和头寸。

同时，由于材料的多样性，EA 交易开发的许多重要细微差别，如测试和优化，将在下一章中重点介绍。

我们之前已经探讨了 [各种类型的 MQL 程序的设计](#)，包括 EA 交易，以及 [程序启动和停止的功能入门知识](#)。尽管 EA 交易是在定义了工作交易品种的特定图表上启动，但可以成功地集中管理任意一组金融工具的交易。这种 EA 交易传统上被称为多货币，尽管实际上其投资组合可能包括差价合约 (CFD)、股票、大宗商品和其他市场的交易代码。

在 EA 交易和指标中，有许多 [关键事件 OnInit 和 OnDeinit](#)。这些事件不是强制性的，但通常出现在程序准备和常规完成的代码中：我们之前在示例中使用了这些事件，后面还会继续使用。在单独的一节中，我们提供了 [所有事件处理函数的概述](#)：迄今为止，我们已经详细研究了其中的一些函数（例如，[OnCalculate](#) 指标事件和 [OnTimer](#) 计时器）。本章中将介绍 EA 交易特定的事件 [OnTick](#)、[ontrade](#)、[OnTradeTransaction](#)）。

EA 交易可以使用最广泛的来源数据作为交易信号：[报价](#)、[分时报价](#)、[市场深度](#)、[交易账户历史](#)或指标读数。在后一种情况下，创建指标实例和从缓冲区读取值的原则与[使用 MQL 程序中的现成指标](#)中讨论的原则相同。在以下几节的 EA 交易示例中，我们将演示大多数技巧。

需要注意的是，交易函数不仅可以在 EA 交易中使用，也可以在脚本中使用。我们将看到两个选项的示例。

6.4.1 EA 交易的主要事件：OnTick

当新的分时报价出现时，其包含 EA 交易正在运行的当前图表的工作交易品种的价格，EA 交易的终端将生成 [OnTick](#) 事件。要处理此事件，必须在 EA 交易代码中定义 [OnTick](#) 函数。该函数有以下原型。

[void OnTick \(void\)](#)

可以看到，该函数没有参数。如有必要，应通过调用 [SymbolInfoTick](#) 来请求新价格和其他分时报价特性的值。

从对新分时报价事件的反应来看，该处理程序类似于指标中的 [OnCalculate](#)。但是，[OnCalculate](#) 只能在指标中定义，[OnTick](#) 只能在 EA 交易中定义（更准确地说，指标、脚本或服务代码中的 [OnTick](#) 函数将会被完全忽略）。

同时，EA 交易不必包含 [OnTick](#) 处理程序。除了这个事件之外，EA 交易还可以处理 [OnTimer](#)、[OnBookEvent](#) 和 [OnChartEvent](#) 事件，并从中执行所有必要的交易操作。

EA 交易中的所有事件按照到达的顺序逐个处理，因为 EA 交易和所有其他 MQL 程序一样，是单线程的。如果队列中已经有 [OnTick](#) 事件或正在处理此类事件，则新的 [OnTick](#) 事件不会排队。

无论自动交易是禁用还是启用（终端界面中的 [Algo trading](#) 按钮），都会生成 [OnTick](#) 事件。禁用自动交易仅意味着限制 EA 交易发送交易请求，但不阻止 EA 交易运行。

切记，只能为一个交易品种生成分时报价事件，也就是当前图表的交易品种。如果 EA 交易是多币种的，那么从其他交易品种获取分时报价应以某种替代方式进行组织，例如，使用间谍指

标 *EventTickSpy.mq5* 或订阅市场深度行情事件，如 *MarketBookQuasiTicks.mq5* 来请求新价格和其他分时报价特性的值。

举一个简单的示例，考虑 EA 交易 *ExpertEvents.mq5*。该脚本为通常用于启动交易算法的所有事件定义了处理程序。我们稍后将研究一些其他事件 (*OnTrade*、*OnTradeTransaction* 以及测试程序事件)。

所有处理程序均可调用 *display* 辅助函数，该函数在多行注释中输出当前时间 (毫秒系统计数器标签) 和处理程序名称。

```
#define N_LINES 25
#include <MQL5Book/Comments.mqh>

void Display(const string message)
{
    ChronoComment((string)GetTickCount() + ": " + message);
}
```

新的分时报价到达时，将自动调用 *OnTick* 事件。对于计时器和订单簿事件，你需要使用 *OnInit* 中的 *EventSetTimer* 和 *MarketBookAdd* 调用来激活相应的处理程序。

```
void OnInit()
{
    Print(__FUNCTION__);
    EventSetTimer(2);
    if(!MarketBookAdd(_Symbol))
    {
        Print("MarketBookAdd failed:", _LastError);
    }
}

void OnTick()
{
    Display(__FUNCTION__);
}

void OnTimer()
{
    Display(__FUNCTION__);
}

void OnBookEvent(const string &symbol)
{
    if(symbol == _Symbol) // react only to order book of "our" symbol
    {
        Display(__FUNCTION__);
    }
}
```

也可以提供图表变更事件：可以用于基于图形对象的标记交易，这些交易通过按按钮或热键以及在来自其他程序的自定义事件到达时实现，比如像 *EventTickSpy.mq5* 这样的指标。

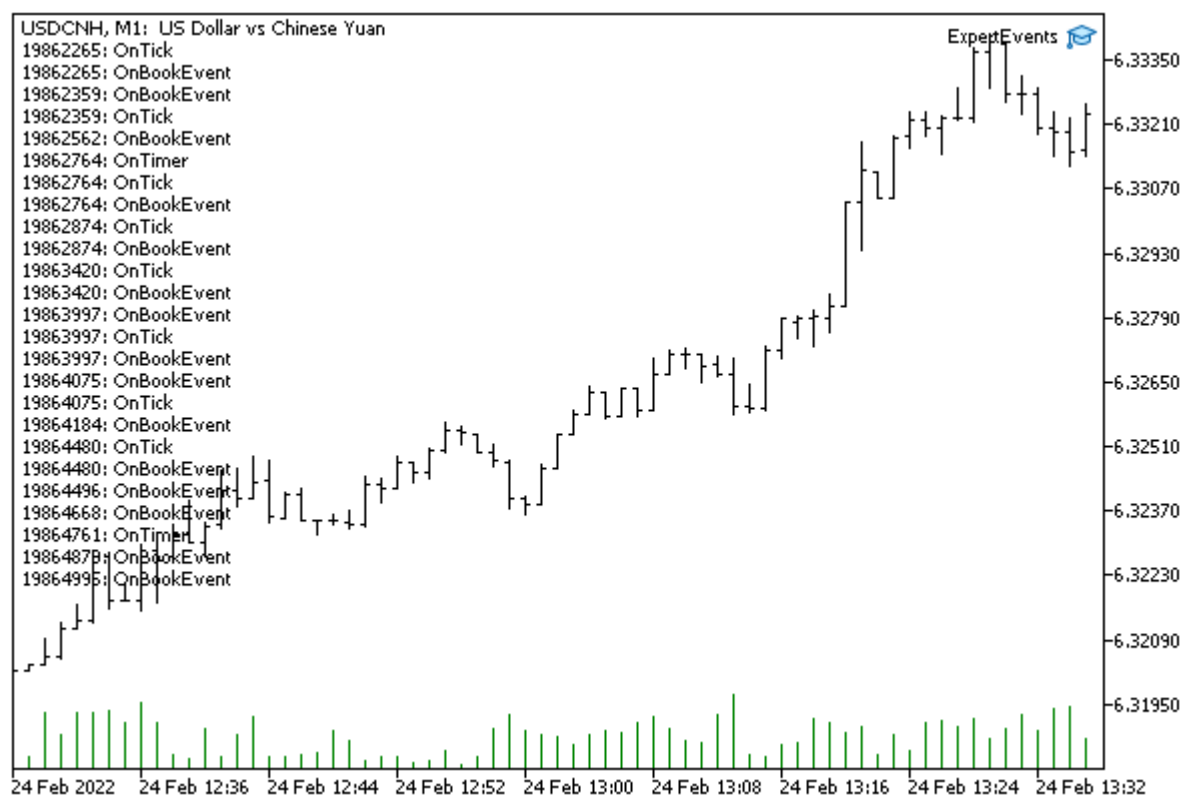
```

void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &str)
{
    Display(__FUNCTION__);
}

void OnDeinit(const int)
{
    Print(__FUNCTION__);
    MarketBookRelease(_Symbol);
    Comment("");
}

```

下面的屏幕截图显示了图表上 EA 交易操作的结果。



EA 交易中各种类型事件的注释

请注意，*OnBookEvent* 事件 (如果是为交易品种播报的) 到达频次要高于 *OnTick*。

6.4.2 基本原理和概念：订单、交易和仓位

在开始研究 MQL5 中的 EA 交易开发之前，让我们回忆一下该平台的一般架构和形式化交易活动的基本概念。

MetaTrader 5 是一个客户终端，连接到多层级的服务器，这些服务器遍布于经纪商、经销商或交易所的计算机。一旦用户填写了执行交易订单，该订单将经过几个转发和验证阶段，然后由交易商或交易所登记或拒绝。然后，根据流动性、价格变化率、交易品种交易暂停或技术问题等情况，在市场上登记的订单可能会执行，也可能不会执行。



处理交易请求的一般方案

此处，绿色箭头表示交易操作成功执行，从终端进入市场，而红色箭头表示可能被拒绝。

MQL 程序生成的订单也会经历类似的情况。如果出现不利的结果，MQL5 API 将允许我们通过错误代码了解失败的原因。

整个过程用三个基本术语来表达（并记录在报告中）：订单、交易和仓位。

订单是交易员向经纪商发出的买卖金融工具的指令。MetaTrader 5 支持多种类型的订单，但在一个简化形式中，可以有条件地分为市价单、挂单和特殊保护级别 **Take Profit** 和 **Stop Loss**。

订单成功执行的结果是在交易系统中执行交易。具体而言，如果是市价单，则可以按当前价格达成交易，如果是挂单，则在价格达到订单中指定的值时触发挂单。换言之，交易是买入或卖出特定金融工具的事实。

需要注意的是，在某些情况下，一个订单的执行可能会导致几笔交易。例如，如果订单簿未包含足够数量的交易品种流动性，那么买入订单可以通过多种反向订单执行，包括那些价格略有不同的订单。

根据交易买入或卖出的金融工具分别形成多头或空头仓位，这反映在交易账户的资产/负债中。由于仓位金融工具价格的后续变化，账户上会形成浮动盈亏，可以通过反向交易操作（订单和交易）平仓进行锁定。根据交易账户的类型（净额结算或对冲），同一金融工具的交易可修改单个净仓位或创建/删除独立仓位。

详细信息可参见 [终端用户手册](#)。

所有订单、交易和仓位都包含在账户的交易历史中。

接下来，我们来看软件 API，其包括以下功能：发送交易订单、获取账户中投资组合的当前状态、检查保证金占用情况和潜在利润/损失以及分析交易历史。

6.4.3 交易操作的类型

MQL5 中的交易是通过使用 [OrderSend](#) 函数发送订单来实现的。我们将在下面的一节中学习该含税，因为必须首先要熟悉几个概念。

第一个新概念是交易操作类型。每个交易请求都包含所请求交易类型的指示，并允许你执行不同操作，如开仓和平仓，以及下达、修改和删除挂单。ENUM_TRADE_REQUEST_ACTIONS 枚举中描述了所有类型的交易操作。

标识符	说明
TRADE_ACTION_DEAL	根据指定的参数下单进行即时交易（下达市场订单）
TRADE_ACTION_PENDING	下单在指定条件下执行交易（挂单）
TRADE_ACTION_SLTP	更改持仓的 <i>Stop Loss</i> 和 <i>Take Profit</i> 值
TRADE_ACTION_MODIFY	更改之前下单的参数
TRADE_ACTION_REMOVE	删除之前的挂单
TRADE_ACTION_CLOSE_BY	通过反向持仓进行平仓

当请求 `TRADE_ACTION_DEAL` 和 `TRADE_ACTION_PENDING` 时，程序需要指定特定的订单类型。这是另一个在 MQL5 API 中有其自身体现的重要概念，我们将在下一节中探讨。

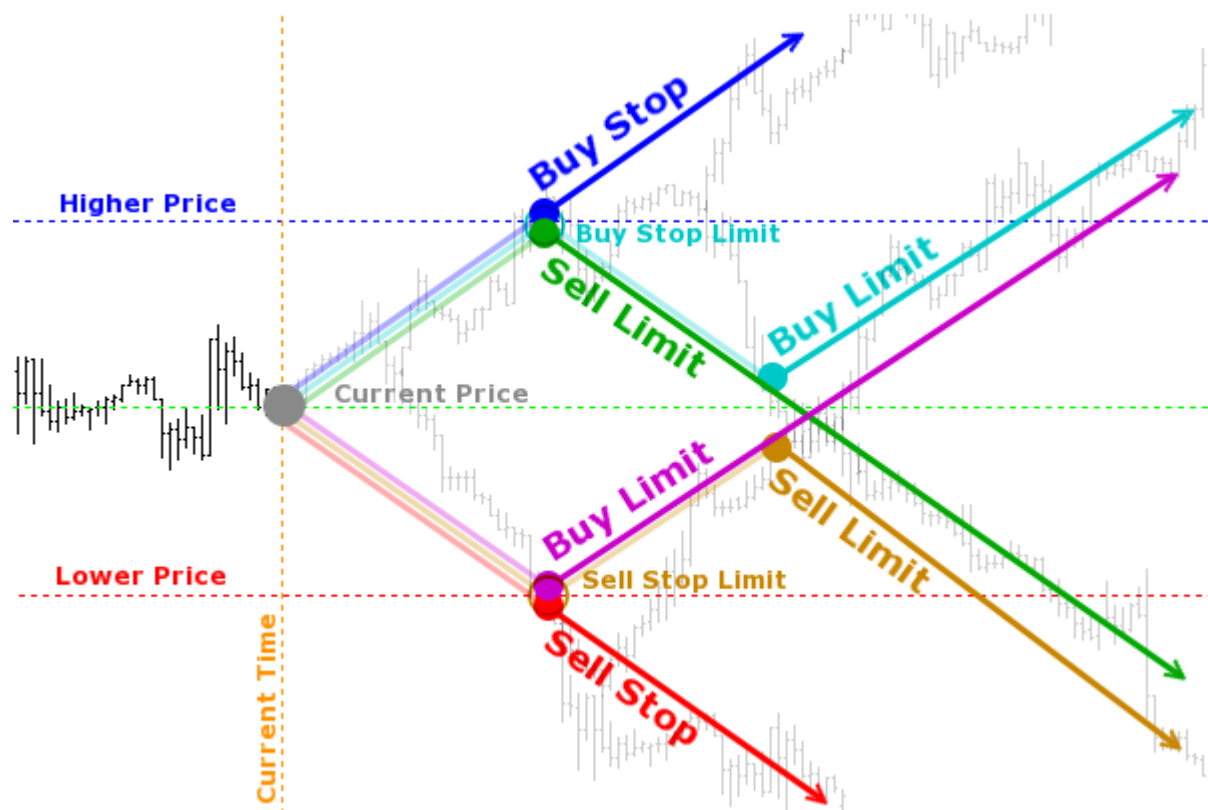
6.4.4 订单类型

我们知道，MetaTrader 5 支持多种 **订单类型**：两种以当前价格买入和卖出的市场订单，以及六种具有高于和低于市价的预定义激活水平的挂单。所有这些类型在 MQL5 API 中都是可用的，并且由 `ENUM_ORDER_TYPE` 枚举的元素进行描述。稍后我们将探讨如何在程序中创建特定类型的订单。现在，让我们熟悉一下枚举。

标识符	说明
ORDER_TYPE_BUY	市场买入订单
ORDER_TYPE_SELL	市场卖出订单
ORDER_TYPE_BUY_LIMIT	<i>Buy Limit</i> 挂单
ORDER_TYPE_SELL_LIMIT	<i>Sell Limit</i> 挂单
ORDER_TYPE_BUY_STOP	<i>Buy Stop</i> 挂单
ORDER_TYPE_SELL_STOP	<i>Sell Stop</i> 挂单
ORDER_TYPE_BUY_STOP_LIMIT	<i>Buy Limit</i> 当价格达到指定的较高水平时下达的挂单
ORDER_TYPE_SELL_STOP_LIMIT	<i>Sell Limit</i> 当价格达到指定的较低水平时下达的挂单
ORDER_TYPE_CLOSE_BY	以相反方向的仓位平仓的订单

最后一个要素对应于反向平仓的行为：这种可能性仅存在于 **对冲** 账户和金融工具的特性允许此类操作 (`SYMBOL_ORDER_CLOSEBY`)。

下图可帮助你理解挂单激活的一般原则。该图以灰色显示预期的期货价格变动情况。但目前还不知道哪个预测是正确的。



挂单激活方案

Buy Stop 和 **Sell Stop** 挂单遵循水平突破原则：对于 **Buy Stop**，该水平应位于当前价格之上，对于 **Sell Stop**，该水平应低于当前价格。换言之，在一个给定的水平上，我们希望执行买入或卖出运算，以期待行情沿趋势方向进一步发展。

Buy Limit 和 **Sell Limit** 执行从水平反弹的策略，在这种情况下，买入激活价格低于当前价格，卖出价格更高。这意味着趋势发生变化或者在区间内波动。在上图中，挂单的相同上限（较高价格）和下限（较低价格）激活水平用于说明突破和反弹。

挂单可以按当前价格下单，很可能会立即执行。此外，与市场订单不同的是，这种应用于限价订单的技术保证了交易价格不会低于要求的价格。

Buy Stop Limit 和 **Sell Stop Limit** 类型的订单不会因为激活而被发送到市场，而是在原始订单中指定的某个额外价位下预设挂单 **Buy Limit** 或 **Sell Limit**。

对于交易所金融工具，限价订单（**Buy Limit** 和 **Sell Limit**）通常直接显示在订单簿中，并且对其他市场参与者可见。

相比之下，**Stop** 和 **Stop Limit** 订单（**Buy Stop**、**Sell Stop**、**Buy Stop Limit** 和 **Sell Stop Limit**）不直接输出到外部交易系统。在达到止损价格之前，这些类型的订单将在 MetaTrader 5 平台中处理。当达到买入止损订单或卖出止损订单中指定的止损价格时，执行相应的市场操作。当达到 **Buy Stop Limit** 或 **Sell Stop Limit** 订单中指定的止损价格时，就会下达相应的限制订单。

在交易所执行模式下，不会检查下达限制订单时指定的价格。该价格可以被指定为高于当前 **Ask** 价格（对于买入订单）和低于 **Bid** 价格（对于卖出订单）。以这样的价格下单时，该订单几乎会立即被触发并变成市场订单。

请注意，并非所有类型的订单都允许用于特定的金融工具：**SYMBOL_ORDER_MODE** 特性描述了允许的订单类型标志。

6.4.5 按价格和交易量的订单执行模式

发送交易请求时，我们需要以特殊的方式在算法中指定买入/卖出价格和交易量。同时应注意的是，在金融市场中，无法保证在某个时刻，金融工具能以期望的价格获得全部请求交易量。因此，交易操作受价格和交易量执行模式 (或策略) 的制约。二者定义了 在发送请求过程中价格发生变化或者不能完全满足需求时的规则。

在交易品种一章的 [交易条件和订单执行模式](#) 一节中，我们已经讨论了按价格执行订单 (SYMBOL_TRADE_EXEMODE) 和按交易量执行订单 (SYMBOL_FILLING_MODE) 的设置，二者均由经纪商设置。根据可用的 SYMBOL_FILLING_MODE 模式，MQL 程序必须为特殊结构体 [MqlTradeRequest](#) 中新形成的订单选择填充模式 (很快我们将在实践中看到这一点)。

版本在 ENUM_ORDER_TYPE_FILLING 枚举中提供：其标识符与 SYMBOL_FILLING_MODE 的标识符相呼应。

执行策略 (值)	说明
ORDER_FILLING_FOK (0)	全部成交否则取消
ORDER_FILLING_IOC (1)	立即成交否则取消
ORDER_FILLING_RETURN (2)	部分成交

根据 ORDER_FILLING_FOK 策略，订单必须按指定的交易量成交。如果目前市场上没有足够交易量的金融工具，订单将不会被执行。所需的交易量可以由目前市场上可获得的几种报价组成。能否使用 FOK 订单取决于是否存在 SYMBOL_FILLING_FOK 权限。

采用 ORDER_FILLING_IOC 策略时，交易员同意在订单中指定的限制内，以市场上可用的最大交易量进行交易。如果无法全额成交，订单将以可用交易量执行，缺失的交易量将被取消。能否使用 IOC 订单取决于是否存在 SYMBOL_FILLING_IOC 权限。

使用 ORDER_FILLING_RETURN 时，在部分执行的情况下，剩余交易量的订单不会被取消，而是继续操作。这是默认模式，并且始终可用。但是有一个例外：市场执行模式下不允许部分成交订单 (SYMBOL_TRADE_EXEMODE 交易品种特性中的 SYMBOL_TRADE_EXECUTION_MARKET)。

因此，在发送市场 (非挂单) 订单之前，MQL 程序应根据相应金融工具的 SYMBOL_FILLING_MODE 特性正确设置 ORDER_TYPE_FILLING 策略之一：该特性包含允许模式的位标志组合。

对于挂单，无论 SYMBOL_TRADE_EXEMODE 执行模式如何，都必须使用 ORDER_FILLING_RETURN 策略，因为此类订单稍后将根据经纪商当时设定的规则成交。

不同于交易量成交策略，不能选择价格的订单执行模式，因为它是由经纪商为每个交易品种预先确定的。这会影响在提交交易请求之前应填写 [MqlTradeRequest](#) 结构体的哪些字段。

根据执行模式应用成交策略可以表示为一个表 (+ 表示允许，- 表示禁用，± 表示取决于交易品种设置)：

成交策略	ORDER_FILLING	ORDER_FILLING	ORDER_FILLING
执行模式	_FOK	_IOC	_RETURN
SYMBOL_TRADE_EXECUTION_INSTANT	+	+	+
SYMBOL_TRADE_EXECUTION_REQUEST	+	+	+
SYMBOL_TRADE_EXECUTION_MARKET	±	±	-
SYMBOL_TRADE_EXECUTION_EXCHANGE	±	±	+
挂单	-	-	+

在 `SYMBOL_TRADE_EXECUTION_INSTANT` 和 `SYMBOL_TRADE_EXECUTION_REQUEST` 执行模式下，允许使用所有交易量成交策略。

6.4.6 挂单到期日期

对于挂单，一个重要的特性是其到期模式。在 MQL5 API 中，可以在特殊 *MqlTradeRequest* 结构体的 `type_time` 字段中设置订单有效期（前提是交易请求是通过 *OrderSend* 函数发送的）。`ENUM_ORDER_TYPE_TIME` 枚举中描述了可接受的值。

标识符（值）	说明
ORDER_TIME_GTC (0)	订单在被取消之前一直在队列中
ORDER_TIME_DAY (1)	订单在当前交易日内有效
ORDER_TIME_SPECIFIED (2)	订单在到期日期之前有效
ORDER_TIME_SPECIFIED_DAY (3)	订单在指定日期的 23:59:59 之前有效（如果该时间不在交易时段内，则到期时间将发生在最近的下一个交易时间）

应当注意，每个金融工具具有两个特性 `SYMBOL_EXPIRATION_MODE` 和 `SYMBOL_ORDER_GTC_MODE`，它们确定该金融工具的 **挂单到期规则**。当形成订单时，MQL 程序可以选择允许的模式之一。在学习了 *OrderSend* 函数之后，我们将探讨一个示例。

6.4.7 期货订单的保证金计算：OrderCalcMargin

在向服务器发送交易请求之前，MQL 程序可以使用 *OrderCalcMargin* 函数计算计划交易所需的保证金。建议始终执行此操作，以避免过度使用存款。


```
bool OrderCalcMargin(ENUM_ORDER_TYPE action, const string symbol,
double volume, double price, double &margin)
```

该函数可计算指定 *action* 订单类型和带 *volume* 手数的 *symbol* 金融工具所需的保证金。这与当前账户的设置一致，但不考虑现有的挂单和持仓。ENUM_ORDER_TYPE 枚举在 [订单类型](#) 一节介绍。

保证金值（以账户货币表示）被写入通过引用传递的 *margin* 参数。

应强调的是，这是对单个新仓位或订单的保证金估算，而不是执行后抵押品的总价值。此外，执行估算时不考虑当前账户的其他挂单和持仓。现实中，保证金的值取决于许多因素，包括其他订单和仓位，并可能随着市场环境（如杠杆）的变化而变化。

该函数返回一个成功（*true*）或错误（*false*）的指示符。可以用通常的方法从变量 *_LastError* 中获得错误代码。

OrderCalcMargin 函数只能在 EA 交易和脚本中使用。要计算指标的保证金，需要实现一个替代方法，例如，在一个图表对象中启动一个辅助 EA 交易，向其传递参数。并通过事件机制得到结果，或者在 MQL5 中使用 [公式](#) 独立说明计算（根据工具的类型）。在 [下一节](#)中，我们将给出这样一个实现的示例以及对潜在利润/损失的估算。

我们可以编写一个简单脚本，为 *Market Watch* 中的交易品种调用 *OrderCalcMargin*，并比较它们的保证金值。不过我们反而要将这个任务稍微复杂化，考虑一下头文件

LotMarginExposure.mqh，该文件允许在打开一个具有预定风险水平的仓位后评估存款负载和保证金水平。稍后我们将讨论 *OrderCheck* 函数，其能够提供类似的信息。但是，我们的算法还能够解决根据给定的负载或风险水平选择每手规模的逆问题。

非交易性 EA 交易 *LotMarginExposureTable.mq5* 演示了新功能的使用。

理论上，MQL 程序作为 EA 交易执行的事实并不意味着交易操作必须在其中执行。很多时候，就像在我们的示例中一样，各种工具都是以 EA 交易的形式创建的。与脚本相比，其优势在于它们保留在图表上，并且可以响应某些事件而无限期地执行其函数。

在新的 EA 交易中，我们使用了用 [对象](#) 创建交互式图形接口的技巧。更简单地说，对于给定的交易品种列表，EA 交易将在图表上显示一个具有几列保证金指标的表格，并且该表格可以按每列排序。稍后我们将提供这些列的列表。

由于分析手数、保证金和存款负载是一项常见任务，我们将把实现分离到一个单独的头文件 *LotMarginExposure.mqh* 中。

所有文件函数都分组在一个名称空间中，以避免冲突，并且为了清晰起见（在调用一个内部函数之前指明上下文，告知该函数的来源和位置）。

```
namespace LEMLR
{
    ...
};
```

缩写 *LEMLR* 的意思是“手数、敞口、保证金水平、风险”。

主要计算在 *Estimate* 函数中执行。考虑到内置 *OrderCalcMargin* 函数的原型，在 *Estimate* 函数参数中，我们需要传递交易品种名称、订单类型、数量和价格。但这还不是我们需要的全部功能。

```
bool Estimate(const ENUM_ORDER_TYPE type, const string symbol, const double lot,
             const double price,...)
```

我们打算评估一个交易操作的几个指标，这些指标是相互联系的，可以在不同的方向上计算，具体取决于用户输入的初始数据及其想要计算的内容。例如，使用上述参数，很容易找到新的保证金水平和账户负载。其公式完全相反：

```
Ml = money / margin * 100
Ex = margin / money * 100
```

此处，*margin* 变量是指保证金金额，为此只需要调用 *OrderCalcMargin*。

但是，交易者通常喜欢从预定的负载或保证金水平开始计算交易量。此外，还有一种同样常用的基于风险的手数计算方法。风险被理解为在不利的价格变动情况下交易的潜在损失金额，其结果是上述公式中的另一个变量（即 *money*）的值将减少。

为了计算损失，重要的是要知道交易期间（策略的持续时间）金融工具的波动性或用户假设的止损距离。

因此，*Estimate* 函数的参数列表会扩展。

```
bool Estimate(const ENUM_ORDER_TYPE type, const string symbol, const double lot,
             const double price,
             const double exposure, const double riskLevel, const int riskPoints,
             const ENUM_TIMEFRAMES riskPeriod, double money,...)
```

在 *exposure* 参数中，我们将期望的存款负载指定为百分比，在 *riskLevel* 参数中，我们指出我们愿意承担风险的存款部分（也以百分比表示）。对于基于风险的计算，可以在 *riskPoints* 参数中以点数为单位传递止损规模。当其值等于 0 时，*riskPeriod* 参数开始发挥作用：指定算法自动计算以点数为单位的交易品种报价范围的时段。最后，在 *money* 参数中，可以为手数评估指定任意金额的自由保证金。一些交易员会有条件地在几个机器人之间分配存款。当 *money* 为 0 时，该函数会用 *AccountInfoDouble(ACCOUNT_MARGIN_FREE)* 特性填充该变量。

现在，我们需要决定如何返回该函数的结果。由于该函数能够评估许多交易指标和几个交易量选项，因此有必要定义 *SymbolLotExposureRisk* 结构体。

```
struct SymbolLotExposureRisk
{
    double lot; // requested volume (or minimum)
    int atrPointsNormalized; // price range normalized by tick size
    double atrValue; // range as the amount of profit/loss for 1 lot
    double lotFromExposureRaw; // not normalized ( can be less than the minimum)
    double lotFromExposure; // normalized lot from deposit loading
    double lotFromRiskOfStopLossRaw; // not normalized (can be less than the minimum)
    double lotFromRiskOfStopLoss; // normalized lot from risk
    double exposureFromLot; // loading based on the volume of 'lot'
    double marginLevelFromLot; // margin level from 'lot' volume
    int lotDigits; // number of digits in normalized lots
};
```

如果手数不等于 0，则结构体中的 *lot* 字段包含传递给 *Exposure* 函数的手数。如果传递的手数为 0，则代之以交易品种特性 *SYMBOL_VOLUME_MIN*。

基于存款负载和风险，为交易量的计算值分配了两个字段：带后缀 *Raw* (*lotFromExposureRaw*, *lotFromRiskOfStopLossRaw*) 和不带后缀 (*lotFromExposure*, *lotFromRiskOfStopLoss*)。Raw 原始字段

包含“纯算术”结果，可能与交易品种规范不匹配。在不带后缀的字段中，考虑最小值、最大值和步长来对手数进行归一化。这种复制非常有用，尤其是在计算得出的值小于最小手数的情况下（例如，*lotFromExposureRaw* 等于 0.023721，最小值为 0.1，因此 *lotFromExposure* 减少到 0）：然后，从 *Raw* 字段的内容中，你可以估算要增加多少资金或增加多少风险才能达到最小手数。

我们说明一下 *Estimate* 函数的最后一个输出参数，作为这个结构体的参考。我们将逐步填充该函数中的所有字段。首先，我们通过调用 *OrderCalcMargin* 获得一个手数的保证金，并将其保存到一个局部变量 *lot1margin* 中。

```
bool Estimate(const ENUM_ORDER_TYPE type, const string symbol, const double lot,
             const double price, const double exposure,
             const double riskLevel, const int riskPoints, const ENUM_TIMEFRAMES riskPeriod,
             double money, SymbolLotExposureRisk &r)
{
    double lot1margin;
    if(!OrderCalcMargin(type, symbol, 1.0,
                       price == 0 ? GetCurrentPrice(symbol, type) : price,
                       lot1margin))
    {
        Print("OrderCalcMargin ", symbol, " failed: ", _LastError);
        return false;
    }
    if(lot1margin == 0)
    {
        Print("Margin ", symbol, " is zero, ", _LastError);
        return false;
    }
    ...
}
```

如果没有指定输入价格，即 *price* 等于 0，辅助函数 *GetCurrentPrice* 将根据订单类型返回一个合适的价格：对于买入，将采用交易品种特性 *SYMBOL_ASK*，对于卖出，将采用 *SYMBOL_BID*。此处省略了这个和其他辅助函数，其内容可以在所附的源代码中找到。

如果保证金计算失败，或者收到零值，*Estimate* 函数将返回 *false*。

请记住，零保证金可能正常，但也可能是错误，具体取决于金融工具和订单类型。因此，对于交易所交易代码，挂单需要缴纳存款，但对于场外交易代码则不需要（即存款为 0 是正确的）。在调用代码中应考虑到这一点：代码应只为这样的交易品种组合和操作类型请求保证金，这对它们有意义并且应假定为非零。

有了一个手数的存款，我们可以计算出手数数量，以确保给定的存款负载。

```
double usedMargin = 0;
if(money == 0)
{
    money = AccountInfoDouble(ACCOUNT_MARGIN_FREE);
    usedMargin = AccountInfoDouble(ACCOUNT_MARGIN);
}

r.lotFromExposureRaw = money * exposure / 100.0 / lot1margin;
r.lotFromExposure = NormalizeLot(symbol, r.lotFromExposureRaw);
...
```

辅助函数 *NormalizeLot* 如下所示。

为了根据风险和波动性得到手数，需要进行更多计算。

```
const double tickValue = SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_VALUE);
const int pointsInTick = (int)(SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_SIZE)
    / SymbolInfoDouble(symbol, SYMBOL_POINT));
const double pointValue = tickValue / pointsInTick;
const int atrPoints = (riskPoints > 0) ? (int)riskPoints :
    (int)((MathMax(iHigh(symbol, riskPeriod, 1), iHigh(symbol, riskPeriod, 0))
        - MathMin(iLow(symbol, riskPeriod, 1), iLow(symbol, riskPeriod, 0)))
        / SymbolInfoDouble(symbol, SYMBOL_POINT));
// rounding by tick size
r.atrPointsNormalized = atrPoints / pointsInTick * pointsInTick;
r.atrValue = r.atrPointsNormalized * pointValue;

r.lotFromRiskOfStopLossRaw = money * riskLevel / 100.0
    / (pointValue * r.atrPointsNormalized);
r.lotFromRiskOfStopLoss = NormalizeLot(symbol, r.lotFromRiskOfStopLossRaw);
...
```

此处，我们找到了该工具的一个点的成本及其在指定期间的变化范围，之后我们已经计算了手数。

最后，我们得到给定手的账户负载和保证金水平。

```
r.lot = lot <= 0 ? SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : lot;
double margin = r.lot * lotMargin;

r.exposureFromLot = (margin + usedMargin) / money * 100.0;
r.marginLevelFromLot = margin > 0 ? money / (margin + usedMargin) * 100.0 : 0;
r.lotDigits = (int)MathLog10(1.0 / SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN))

return true;
}
```

如果计算成功，该函数将返回 *true*。

以下是 *NormalizeLot* 函数的简化视图 (为简单起见，省略了所有 0 检查)。有关相应特性的详细信息，请参见 *允许的交易量* 一节。

```
double NormalizeLot(const string symbol, const double lot)
{
    const double stepLot = SymbolInfoDouble(symbol, SYMBOL_VOLUME_STEP);
    const double newLotsRounded = MathFloor(lot / stepLot) * stepLot;
    const double minLot = SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN);
    if(newLotsRounded < minLot) return 0;
    const double maxLot = SymbolInfoDouble(symbol, SYMBOL_VOLUME_MAX);
    if(newLotsRounded > maxLot) return maxLot;
    return newLotsRounded;
}
```

上述 *Estimate* 实现未考虑重叠仓位的调整。通常，它们会导致存款减少，因此当前对账户负载和保证金水平的估算可能比实际情况更加悲观，但这提供了额外的保护。若感兴趣，可以添加一个代码来分析已经冻结的、按仓位和订单细分的账户资金组成 (其总额包含在

ACCOUNT_MARGIN 账户特性中):然后可以考虑新订单对保证金的潜在影响(例如,只考虑相反方向的最大仓位或应用降低的对冲保证金率,详见 [保证金要求](#) 一节)。

现在,在 *LotMarginExposureTable.mq5* 中练习保证金和手数估算。考虑到 *Raw* 字段仅在手数标准化导致其归零的情况下才会显示,结果指标表中的总列数为 8。

```
#include <MQL5Book/LotMarginExposure.mqh>
#define TBL_COLUMNS 8
```

在输入参数中,我们将提供指定订单类型、要分析的交易品种列表(用逗号分隔的列表)、可用资金以及手数、目标存款负载、保证金水平和风险的可能性。

```
input ENUM_ORDER_TYPE Action = ORDER_TYPE_BUY;
input string WorkList = ""; // Symbols (comma,separated,list)
input double Money = 0; // Money (0 = free margin)
input double Lot = 0; // Lot (0 = min lot)
input double Exposure = 5.0; // Exposure (%)
input double RiskLevel = 5.0; // RiskLevel (%)
input int RiskPoints = 0; // RiskPoints/SL (0 = auto-range of Ris
input ENUM_TIMEFRAMES RiskPeriod = PERIOD_W1;
```

对于挂单类型,必须选择股票交易品种,因为对于其他交易品种,将获得零保证金,这将导致 *Estimate* 函数中的错误。如果交易品种列表为空,EA 交易将只处理当前图表的交易品种。*Money* 和 *Lot* 参数中的零默认值分别表示账户上的当前自由资金量和每个交易品种的最小手数。

RiskPoints 参数中的 0 值表示在 *RiskPeriod* (默认值为一周)期间获得一个价格范围。

输入参数 *UpdateFrequency* 以秒为单位设置重新计算的频率。如果让该参数等于零,将对每个新的柱线进行重新计算。

```
input int UpdateFrequency = 0; // UpdateFrequency (sec, 0 - once per bar)
```

在全局上下文中需要说明的是:一个交易品种数组(稍后通过解析输入参数 *WorkList* 来填充)和上次成功计算的时间戳。

```
string symbols[];
datetime lastTime;
```

启动时,我们打开第二个计时器。

```
void OnInit()
{
    Comment("Starting...");
    lastTime = 0;
    EventSetTimer(1);
}
```

在计时器处理程序中,如果在分时报价到达时还没有调用 *OnTick*,我们可提供对 *OnTick* 中主计算的第一次调用。例如,这种情况可能发生在周末或市场平静时。此外,*OnTimer* 为以给定频率进行重新计算的入口点。

```

void OnTimer()
{
    if(lastTime == 0) // calculation for the first time (if OnTick did have time to tr
    {
        OnTick();
        Comment("Started");
    }
    else if(lastTime != -1)
    {
        if(UpdateFrequency <= 0) // if there is no frequency, we work on new bars in On
        {
            EventKillTimer(); // and the timer is no longer needed
        }
        else if(TimeCurrent() - lastTime >= UpdateFrequency)
        {
            lastTime = LONG_MAX; // prevent re-entering this 'if' branch
            OnTick();
            if(lastTime != -1) // completed without error
            {
lastTime = TimeCurrent();// update timestamp
            }
        }
        Comment("");
    }
}

```

在 *OnTick* 处理程序中，我们首先检查输入参数，并将交易品种列表转换为字符串数组。如果发现问题，错误交易品种将被写入 *lastTime*：值 -1，随后的分时报价处理从一开始就被中断。

```

void OnTick()
{
    if(lastTime == -1) return; // already had an error, exit

    if(UpdateFrequency <= 0)    // if the update rate is not set
    {
        if(lastTime == iTime(NULL, 0, 0)) return; // waiting for a new bar
    }
    else if(TimeCurrent() - lastTime < UpdateFrequency)
    {
        return;
    }

    const int ns = StringSplit((WorkList == "" ? _Symbol : WorkList), ',', symbols);
    if(ns <= 0)
    {
        Print("Empty symbols");
        lastTime = -1;
        return;
    }

    if(Exposure > 100 || Exposure <= 0)
    {
        Print("Percent of Exposure is incorrect: ", Exposure);
        lastTime = -1;
        return;
    }

    if(RiskLevel > 100 || RiskLevel <= 0)
    {
        Print("Percent of RiskLevel is incorrect: ", RiskLevel);
        lastTime = -1;
        return;
    }
    ...
}

```

具体来说，如果输入值 *Exposure* 和 *Risk Level* 超出了 0 到 100 的范围，则视为错误，因为该数值应为百分比。在正常输入数据的情况下，我们可以更新时间戳，说明结构体

LEMLR::SymbolLotExposureRisk，以从函数 *LEMLR::Estimate*（每个交易品种一个）接收计算的指标以及二维数组 *LME*（来自“Lot Margin Exposure”），以收集所有交易品种的指标。

```

lastTime = UpdateFrequency > 0 ? TimeCurrent() : iTime(NULL, 0, 0);

LEMLR::SymbolLotExposureRisk r = {};

double LME[][13];
ArrayResize(LME, ns);
ArrayInitialize(LME, 0);
...

```

在交易品种循环中，我们可以调用 *LEMLR::Estimate* 函数并填充 *LME* 数组。

```

for(int i = 0; i < ns; i++)
{
    if(!LEMLR::Estimate(Action, symbols[i], Lot, 0,
        Exposure, RiskLevel, RiskPoints, RiskPeriod, Money, r))
    {
        Print("Calc failed (will try on the next bar, or refresh manually)");
        return;
    }

    LME[i][eLot] = r.lot;
    LME[i][eAtrPointsNormalized] = r.atrPointsNormalized;
    LME[i][eAtrValue] = r.atrValue;
    LME[i][eLotFromExposureRaw] = r.lotFromExposureRaw;
    LME[i][eLotFromExposure] = r.lotFromExposure;
    LME[i][eLotFromRiskOfStopLossRaw] = r.lotFromRiskOfStopLossRaw;
    LME[i][eLotFromRiskOfStopLoss] = r.lotFromRiskOfStopLoss;
    LME[i][eExposureFromLot] = r.exposureFromLot;
    LME[i][eMarginLevelFromLot] = r.marginLevelFromLot;
    LME[i][eLotDig] = r.lotDigits;
    LME[i][eMinLot] = SymbolInfoDouble(symbols[i], SYMBOL_VOLUME_MIN);
    LME[i][eContract] = SymbolInfoDouble(symbols[i], SYMBOL_TRADE_CONTRACT_SIZE);
    LME[i][eSymbol] = pack2double(symbols[i]);
}
...

```

特殊枚举 LME 字段的元素用作数组索引，同时为结构体中的指标提供名称和编号。

```

enum LME_FIELDS // 10 fields + 3 additional symbol properties
{
    eLot,
    eAtrPointsNormalized,
    eAtrValue,
    eLotFromExposureRaw,
    eLotFromExposure,
    eLotFromRiskOfStopLossRaw,
    eLotFromRiskOfStopLoss,
    eExposureFromLot,
    eMarginLevelFromLot,
    eLotDig,
    eMinLot,
    eContract,
    eSymbol
};

```

增加了 `SYMBOL_VOLUME_MIN` 和 `SYMBOL_TRADE_CONTRACT_SIZE` 的特性，以供参考。使用 `pack2double` 函数将交易品种名称“打包”成 `double` 类型的近似值，以便随后通过任何字段（包括名称）实现统一排序。


```
double pack2double(const string s)
{
    double r = 0;
    for(int i = 0; i < StringLen(s); i++)
    {
        r = (r * 255) + (StringGetCharacter(s, i) % 255);
    }
    return r;
}
```

在这个阶段，我们已经可以运行了 EA 交易并在日志中打印了结果，就像这样。

```
ArrayPrint(LME);
```

但是一直查看日志并不方便。此外，来自不同列的值的统一格式，甚至是 *double* 中“打包”行的表示，都不能称之为用户友好。因此，开发了计分板类 (*Tableau.mqh*)，用于在图表上显示任意表格。除了在准备表格时，我们可以自己控制每个字段的格式（将来，用不同的颜色突出显示），这个类还允许你按任意列对表格进行交互式排序：第一次鼠标单击按一个方向排序，第二次单击按相反方向排序，第三次单击取消排序。

此处我们不会详细介绍这个类，但是你可以研究它的源代码。值得注意的是，该接口是基于 [图形对象](#)。事实上，表格单元格是由 OBJ_LABEL 类型的对象构成的，读者已经熟悉了它们的所有特性。但是，记分板源代码中使用的一些技术，特别是使用 [图形资源](#) 和测量 [显示文本](#)，将在第七部分中介绍。

tableau 类的构造函数有几个参数：

- prefix - 创建的图形对象名称的前缀
- rows - 行数
- cols - 列数
- height - 以像素为单位的行高（1 表示两倍的字体大小）
- width - 以像素为单位的单元格宽度
- c - 锚定对象的图表角度
- g - 单元格之间的像素间距
- f - 字体大小
- font - 常规单元格的字体名称
- bold - 标题的粗体名称
- bgc - 上下文色
- bgt - 上下文透明度

```
class Tableau
{
public:
    Tableau(const string prefix, const int rows, const int cols,
            const int height = 16, const int width = 100,
            const ENUM_BASE_CORNER c = CORNER_RIGHT_LOWER, const int g = 8,
            const int f = 8, const string font = "Consolas", const string bold = "Arial Bla
            const int mask = TBL_FLAG_COL_0_HEADER,
            const color bgc = 0x808080, const uchar bgt = 0xC0)
        ...
};
```

这些参数中的大多数都可以由用户在 *LotMarginExposureTable.mq5* EA 交易的输入变量中进行设置。

```
input ENUM_BASE_CORNER Corner = CORNER_RIGHT_LOWER;
input int Gap = 16;
input int FontSize = 8;
input string DefaultFontName = "Consolas";
input string TitleFontName = "Arial Black";
input string MotoTypeFontsHint = "Consolas/Courier/Courier New/Lucida Console/Lucida
input color BackgroundColor = 0x808080;
input uchar BackgroundTransparency = 0xC0; // BackgroundTransparency (255 - opaque, 0
```

表格中的列数是预先确定的，行数等于交易品种数加上带有标题的顶行。

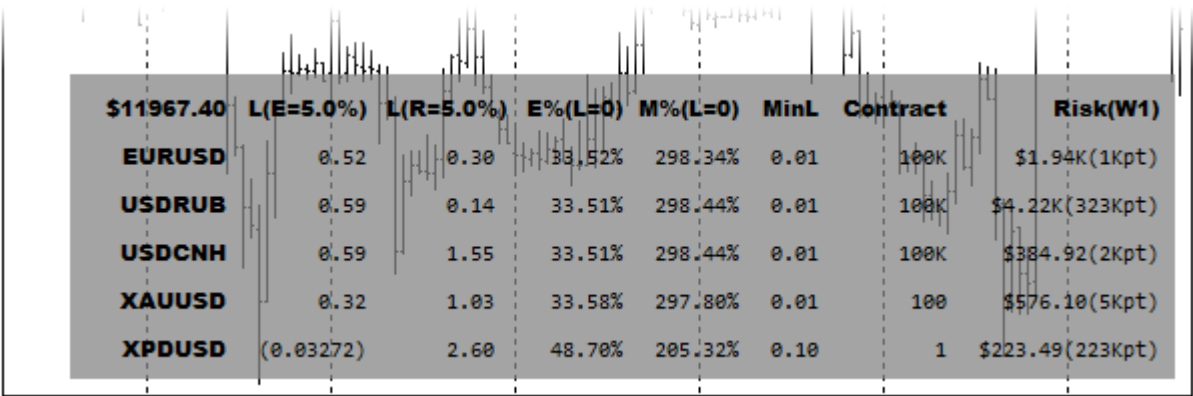
需要注意的是，表格字体的选择不应是成比例的，所以在 *MotoTypeFontsHint* 变量中，工具提示提供了一组标准的 Windows 等宽字体。

使用 *Tableau* 类的 *fill* 方法填充创建的图形对象。

```
bool fill(const string &data[], const string &hint[]) const;
```

我们的 EA 交易通过 *StringFormat* 的一系列转换以及带有标题工具提示的 *hint* 数组，传递从 *LME* 数组获得的 *data* 字符串数组。

下图显示了运行 EA 交易的图表的一部分，运行时采用默认设置，但指定了交易品种列表“EURUSD,USDRUB,USDCNH,XAUUSD,XPDUSD”。



每个交易品种的最小手数的存款和保证金负载水平

交易品种名称显示在左栏中。作为第一列的标题，显示了资金量（在这种情况下，当前账户上是自由的，因为在 *Money* 输入参数中保留为 0）。当你将鼠标放在在列名称上时，可以看到带有解释的工具提示。

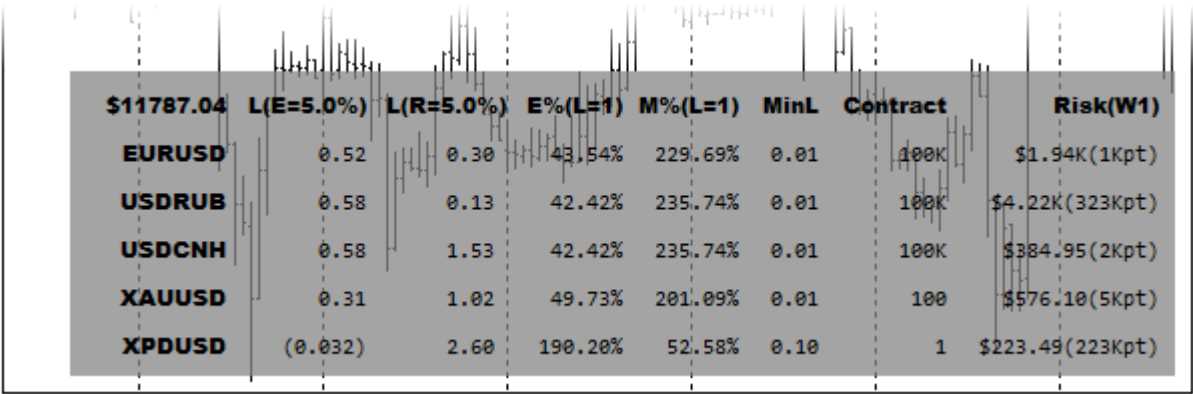
在以下各列中：

- **L(E)** -为交易后 5% 定金的负载水平 E 计算的手数
- **L(R)** -交易失败后，按风险 R 计算的5%保证金的手数（点数和风险金额的范围 - 在最后一列）
- **E%** -最小手数输入后的存款负载
- **M%** -最小手数输入后的保证金水平
- **MinL** -每个交易品种的最小手数
- **Contract** -每个交易品种的合约规模（1 手）
- **Risk** -当交易 1 手且点数范围相同时的止盈/亏损

在 E% 和 M% 列中，在这种情况下，使用最小手数，因为 **Lot** 输入参数为 0（默认值）。

当加载 5% 的存款时，除了“XPDUSD”之外，所有选中的交易品种均可进行交易。对于后者，交易量结果为 0.03272，小于最小手数 0.1，因此结果括在括号中。如果我们允许加载 20%（在 **Exposure** 参数中输入 20），可得到“XPD USD”的最小手数为 0.1。

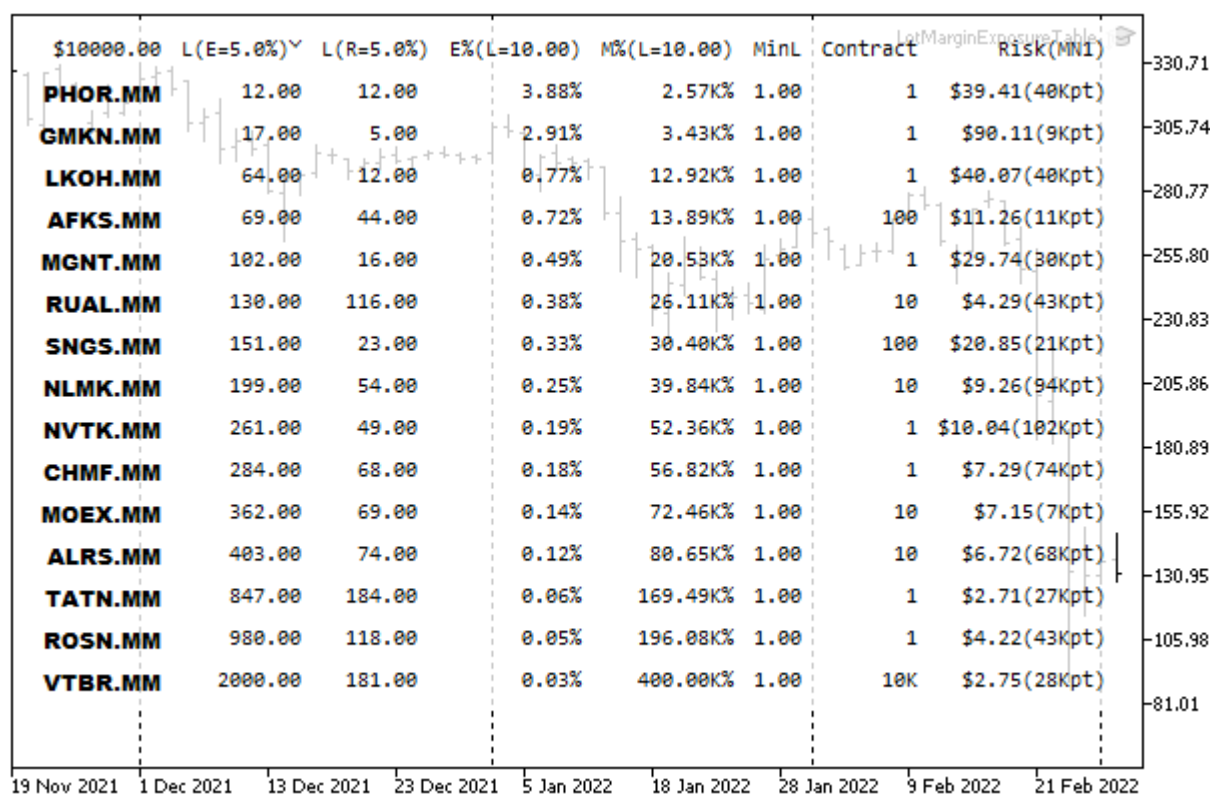
如果我们在 **Lot** 参数中输入值 1，将在表中的 E% 和 M% 列中看到更新的值（负载将增加，保证金水平将下降）。



	L(E=5.0%)	L(R=5.0%)	E%(L=1)	M%(L=1)	MinL	Contract	Risk(W1)
EURUSD	0.52	0.30	43.54%	229.69%	0.01	100K	\$1.94K(1Kpt)
USDRUB	0.58	0.13	42.42%	235.74%	0.01	100K	\$4.22K(323Kpt)
USDCNH	0.58	1.53	42.42%	235.74%	0.01	100K	\$384.95(2Kpt)
XAUUSD	0.31	1.02	49.73%	201.09%	0.01	100	\$576.10(5Kpt)
XPDUSD	(0.032)	2.60	190.20%	52.58%	0.10	1	\$223.49(223Kpt)

每个交易品种的单手存款和保证金负载水平

说明 EA 交易工作的最后一个屏幕截图显示了俄罗斯交易所 MOEX 的一大组蓝筹股，按 5% 存款量计算的交易量排序（第 2 列）。在非标准设置中，可以注意到 **Lot=10**，计算价格区间和风险的周期等于 MN1。背景是半透明的白色，锚定在图表的左上角。



MOEX 工具的手数、存款量和保证金水平

6.4.8 估算交易操作的利润: OrderCalcProfit

MySQL API 函数之一 **OrderCalcProfit** 允许你在满足预期条件的情况下预先评估交易的财务结果。例如,使用此函数,你可以找出达到 **Take Profit** 水平时的盈利金额,以及触发 **Stop Loss** 时的亏损金额。

```
bool OrderCalcProfit(ENUM_ORDER_TYPE action, const string symbol, double volume,
double openPrice, double closePrice, double &profit)
```

该函数根据传递的参数计算当前市场环境下账户货币的利润或损失。

订单类型在 **action** 参数中指定。仅允许使用 **ENUM_ORDER_TYPE** 枚举中的市场订单 **ORDER_TYPE_BUY** 或 **ORDER_TYPE_SELL**。金融工具的名称及其交易量在 **symbol** 和 **volume** 参数中传递。市场进入和退出价格分别由 **openPrice** 和 **closePrice** 参数设置。**profit** 变量作为最后一个参数被引用传递,利润值将被写入其中。

该函数返回一个成功 (**true**) 或错误 (**false**) 的指示符。

OrderCalcProfit 中使用的财务结果计算公式取决于交易品种类型。

标识符	公式
SYMBOL_CALC_MODE_FOREX	$(ClosePrice - OpenPrice) * ContractSize * Lots$
SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE	$(ClosePrice - OpenPrice) * ContractSize * Lots$
SYMBOL_CALC_MODE_CFD	$(ClosePrice - OpenPrice) * ContractSize * Lots$
SYMBOL_CALC_MODE_CFDINDEX	$(ClosePrice - OpenPrice) * ContractSize * Lots$
SYMBOL_CALC_MODE_CFDLEVERAGE	$(ClosePrice - OpenPrice) * ContractSize * Lots$
SYMBOL_CALC_MODE_EXCH_STOCKS	$(ClosePrice - OpenPrice) * ContractSize * Lots$
SYMBOL_CALC_MODE_EXCH_STOCKS_MOEX	$(ClosePrice - OpenPrice) * ContractSize * Lots$
SYMBOL_CALC_MODE_FUTURES	$(ClosePrice - OpenPrice) * Lots * TickPrice / TickSize$
SYMBOL_CALC_MODE_EXCH_FUTURES	$(ClosePrice - OpenPrice) * Lots * TickPrice / TickSize$
SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS	$(ClosePrice - OpenPrice) * Lots * TickPrice / TickSize$
SYMBOL_CALC_MODE_EXCH_BONDS	$Lots * ContractSize * (ClosePrice * FaceValue + AccruedInterest)$
SYMBOL_CALC_MODE_EXCH_BONDS_MOEX	$Lots * ContractSize * (ClosePrice * FaceValue + AccruedInterest)$
SYMBOL_CALC_MODE_SERV_COLLATERAL	$Lots * ContractSize * MarketPrice * LiquidityRate$

上述公式中使用了以下符号：

- Lots - 每手仓量 (合约份额)
- ContractSize - 合约规模 (一手, [SYMBOL_TRADE_CONTRACT_SIZE](#))
- TickPrice - 分时报价价格 ([SYMBOL_TRADE_TICK_VALUE](#))
- TickSize - 分时报价规模 ([SYMBOL_TRADE_TICK_SIZE](#))
- MarketPrice - 最后已知 [买入/卖出](#) 价格 (取决于交易类型)
- OpenPrice - 仓位开盘价
- ClosePrice - 仓位收盘价
- FaceValue - 债券的面值 ([SYMBOL_TRADE_FACE_VALUE](#))
- LiquidityRate - 流动性比率 ([SYMBOL_TRADE_LIQUIDITY_RATE](#))
- AccruedInterest - 累积息票收入 ([SYMBOL_TRADE_ACCRUED_INTEREST](#))

OrderCalcProfit 函数只能在 EA 交易和脚本中使用。要计算指标中的潜在利润/损失，需要实施替代方法，例如，使用公式的独立计算。

为了绕过在指标中使用 **OrderCalcProfit** 和 **OrderCalcMargin** 函数的限制，我们开发了一组函数，这些含税使用本节以及 [保证金要求](#) 一节的公式执行计算。些函数位于头文件 **MarginProfitMeter.mqh** 中，在公共名称空间 **MPM** 内 (MPM 是 Margin Profit Meter 的缩写)。

特别是，为了计算财务结果，必须有特定金融工具的一个点的值。在上述公式中，该值间接参与了开盘价和收盘价之间的差额 (**ClosePrice - OpenPrice**)。

该函数用于计算一个价格点的值 *PointValue*。

```
namespace MPM
{
    double PointValue(const string symbol, const bool ask = false,
        const datetime moment = 0)
    {
        const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
        const double contract = SymbolInfoDouble(symbol, SYMBOL_TRADE_CONTRACT_SIZE);
        const ENUM_SYMBOL_CALC_MODE m =
            (ENUM_SYMBOL_CALC_MODE)SymbolInfoInteger(symbol, SYMBOL_TRADE_CALC_MODE);
        ...
    }
}
```

在函数开始时，我们请求计算所需的所有交易品种特性。然后，根据交易品种类型，我们以该金融工具利润的货币形式获得利润/损失。请注意，此处没有债券，公式中也就不用考虑名义价格和票息收入。

```
double result = 0;
switch(m)
{
    case SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE:
    case SYMBOL_CALC_MODE_FOREX:
    case SYMBOL_CALC_MODE_CFD:
    case SYMBOL_CALC_MODE_CFDINDEX:
    case SYMBOL_CALC_MODE_CFDLEVERAGE:
    case SYMBOL_CALC_MODE_EXCH_STOCKS:
    case SYMBOL_CALC_MODE_EXCH_STOCKS_MOEX:
        result = point * contract;
        break;

    case SYMBOL_CALC_MODE_FUTURES:
    case SYMBOL_CALC_MODE_EXCH_FUTURES:
    case SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS:
        result = point * SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_VALUE)
            / SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_SIZE);
        break;
    default:
        PrintFormat("Unsupported symbol %s trade mode: %s", symbol, EnumToString(m))
}
...
```

最后一点，如果货币不同，我们将金额转换为账户货币。

```

string account = AccountInfoString(ACCOUNT_CURRENCY);
string current = SymbolInfoString(symbol, SYMBOL_CURRENCY_PROFIT);

if(current != account)
{
    if(!Convert(current, account, ask, result, moment)) return 0;
}

return result;
}
...
};

```

辅助函数 *Convert* 用于转换金额。反过来，该函数依赖于 *FindExchangeRate* 函数，后者会在所有可用的交易品种中搜索包含从 *current* 货币到 *account* 货币汇率的交易品种。

```

bool Convert(const string current, const string account,
const bool ask, double &margin, const datetime moment = 0)
{
    string rate;
    int dir = FindExchangeRate(current, account, rate);
    if(dir == +1)
    {
        margin *= moment == 0 ?
            SymbolInfoDouble(rate, ask ? SYMBOL_BID : SYMBOL_ASK) :
            GetHistoricPrice(rate, moment, ask);
    }
    else if(dir == -1)
    {
        margin /= moment == 0 ?
            SymbolInfoDouble(rate, ask ? SYMBOL_ASK : SYMBOL_BID) :
            GetHistoricPrice(rate, moment, ask);
    }
    else
    {
        static bool once = false;
        if(!once)
        {
            Print("Can't convert ", current, " -> ", account);
            once = true;
        }
    }
    return true;
}

```

FindExchangeRate 函数在 *Market Watch* 中查找字符，并在 *result* 参数中返回第一个匹配的外汇交易品种名称（如果有多个匹配结果）。如果报价对应货币的直接顺序 “*current/account*”，则该函数返回 +1，如果相反，则为 “*account/current*”，即 -1。

```

int FindExchangeRate(const string current, const string account, string &result)
{
    for(int i = 0; i < SymbolsTotal(true); i++)
    {
        const string symbol = SymbolName(i, true);
        const ENUM_SYMBOL_CALC_MODE m =
            (ENUM_SYMBOL_CALC_MODE)SymbolInfoInteger(symbol, SYMBOL_TRADE_CALC_MODE);
        if(m == SYMBOL_CALC_MODE_FOREX || m == SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE)
        {
            string base = SymbolInfoString(symbol, SYMBOL_CURRENCY_BASE);
            string profit = SymbolInfoString(symbol, SYMBOL_CURRENCY_PROFIT);
            if(base == current && profit == account)
            {
                result = symbol;
                return +1;
            }
            else
            if(base == account && profit == current)
            {
                result = symbol;
                return -1;
            }
        }
    }
    return 0;
}

```

这些函数的完整代码可以在随附的文件 *MarginProfitMeter.mqh* 中找到。

我们用一个测试脚本 *ProfitMeter.mq5* 来检查 *OrderCalcProfit* 函数和函数组 *MPM* 的性能：我们将为市场报价的所有交易品种计算虚拟交易的利润/损失估算，可使用两种方法来完成该计算：内置方法和我们自己编写的方法。

在该脚本的输入参数中，可以选择操作类型 *Action*（买入或卖出）、手数 *Lot* 和持仓时间（以柱线 *Duration* 表示）。财务结果是针对当前时间范围内最后一个 *Duration* 柱线的报价计算的。

```
#property script_show_inputs
```

```

input ENUM_ORDER_TYPE Action = ORDER_TYPE_BUY; // Action (only Buy/Sell allowed)
input float Lot = 1;
input int Duration = 20; // Duration (bar number in past)

```

在该脚本的正文中，我们可以连接头文件并显示带有参数的头文件。


```

#include <MQL5Book/MarginProfitMeter.mqh>
#include <MQL5Book/Periods.mqh>

void OnStart()
{
    // guarantee that the operation will only be a buy or a sell
    ENUM_ORDER_TYPE type = (ENUM_ORDER_TYPE)(Action % 2);
    const string text[] = {"buying", "selling"};
    PrintFormat("Profits/Losses for %s %s lots"
        " of %d symbols in Market Watch on last %d bars %s",
        text[type], (string)Lot, SymbolsTotal(true),
        Duration, PeriodToString(_Period));
    ...
}

```

然后，在对交易品种迭代中，以两种方式执行计算，并打印结果以供比较。

```

for(int i = 0; i < SymbolsTotal(true); i++)
{
    const string symbol = SymbolName(i, true);
    const double enter = iClose(symbol, _Period, Duration);
    const double exit = iClose(symbol, _Period, 0);

    double profit1, profit2; // 2 adopted variables

    // standard method
    if(!OrderCalcProfit(type, symbol, Lot, enter, exit, profit1))
    {
        PrintFormat("OrderCalcProfit(%s) failed: %d", symbol, _LastError);
        continue;
    }

    // our own method
    const int points = (int)MathRound((exit - enter)
        / SymbolInfoDouble(symbol, SYMBOL_POINT));
    profit2 = Lot * points * MPM::PointValue(symbol);
    profit2 = NormalizeDouble(profit2,
        (int)AccountInfoInteger(ACCOUNT_CURRENCY_DIGITS));
    if(type == ORDER_TYPE_SELL) profit2 *= -1;

    // output to the log for comparison
    PrintFormat("%s: %f %f", symbol, profit1, profit2);
}
}

```

尝试为不同的账户和金融工具集运行该脚本。

```

Profits/Losses for buying 1.0 lots of 13 symbols in Market Watch on last 20 bars H1
EURUSD: 390.000000 390.000000
GBPUSD: 214.000000 214.000000
USDCHF: -254.270000 -254.270000
USDJPY: -57.930000 -57.930000
USDCNH: -172.570000 -172.570000
USDRUB: 493.360000 493.360000
AUDUSD: 84.000000 84.000000
NZDUSD: 13.000000 13.000000
USDCAD: -97.480000 -97.480000
USDSEK: -682.910000 -682.910000
XAUUSD: -1706.000000 -1706.000000
SP500m: 5300.000000 5300.000000
XPDUSD: -84.030000 -84.030000

```

理想情况下，每行中的数字应匹配。

6.4.9 MqlTradeRequest 结构体

MQL5 API 交易函数（特别是 [OrderCheck](#) 和 [OrderSend](#)）在几个内置结构体上运行。因此，在讨论这些函数本身之前，我们必须考虑这些结构体。

我们从 [MqlTradeRequest](#) 结构体开始，该结构体包含执行交易所需的所有字段。

```

struct MqlTradeRequest
{
    ENUM_TRADE_REQUEST_ACTIONS action;    // Type of action to perform
    ulong magic;                          // Unique Expert Advisor number
    ulong order;                          // Order ticket
    string symbol;                        // Name of the trading instrument
    double volume;                        // Requested trade volume in lots
    double price;                         // Price
    double stoplimit;                    // StopLimit order level
    double sl;                           // Stop Loss order level
    double tp;                           // Take Profit order level
    ulong deviation;                     // Maximum deviation from the given price
    ENUM_ORDER_TYPE type;                // Order type
    ENUM_ORDER_TYPE_FILLING type_filling; // Order type by execution
    ENUM_ORDER_TYPE_TIME type_time;      // Order type by duration
    datetime expiration;                 // Order expiration date
    string comment;                      // Comment to the order
    ulong position;                      // Position ticket
    ulong position_by;                   // Opposite position ticket
};

```

你无须害怕大量的字段：该结构体的设计绝对服务于所有可能类型的交易请求，但在每种具体情况下，通常仅使用少数几个字段。

在填充字段之前，建议通过在其定义中显式初始化或调用 [ZeroMemory](#) 函数来使该结构体归零。

```
MqlTradeRequest request = {};  
...  
ZeroMemory(request);
```

这样可以避免在没有明确赋值的字段中向 API 函数传递随机值时可能出现的错误和负面影响。

下表提供了这些字段的简要说明。我们将在介绍交易操作时看到如何填充这些字段。

字 段	说 明
action	交易操作类型，来自 <code>ENUM_TRADE_REQUEST_ACTIONS</code>
magic	专家 ID (可选)
order	请求修改的挂单订单号
symbol	交易金融工具名称
volume	每手请求的交易量
price	执行订单时必须满足的价格
stoplimit	当 <code>ORDER_TYPE_BUY_STOP_LIMIT</code> 和 <code>ORDER_TYPE_SELL_STOP_LIMIT</code> 订单被激活时，下达限价订单的价格
sl	当价格朝着不利的方向移动时，触发 <i>Stop Loss</i> 的价格
tp	当价格朝着有利的方向移动时，触发 <i>Take Profit</i> 的价格
deviation	与要价的最大可接受偏差 (以点数为单位)
type	订单类型，来自 <code>ENUM_ORDER_TYPE</code>
type_filling	订单填充类型，来自 <code>ENUM_ORDER_TYPE_FILLING</code>
type_time	挂单到期类型，来自 <code>ENUM_ORDER_TYPE_TIME</code>
expiration	挂单到期日期
comment	订单的注释
position	仓位订单号
position_by	<code>TRADE_ACTION_CLOSE_BY</code> 操作的反向仓位订单号

要发送交易操作的订单，需要根据操作特性填写一组不同的字段。有些字段是必填的，有些是可选的 (填写时可以省略)。接下来，我们将在具体操作的上下文中更深入地了解字段需求。

该程序可以检查已形成的 `MqlTradeRequest` 结构体是否正确 (使用 `OrderCheck` 函数)，或将其发送到服务器 (使用 `OrderSend` 函数)。如果成功，将执行请求的操作。

只有 `action` 字段是所有交易活动的必填字段。

magic 字段中的唯一编号通常仅用于市场买入/卖出请求或在创建新挂单时使用。这导致随后用该编号标记已完成的交易和仓位，从而能够对交易行为的处理进行分析。修改仓位或挂单的价格水平以及将其删除时，此字段无效。

从 MetaTrader 5 界面手动执行交易操作时，无法设置 **magic** 标识符，因此其等于零。这提供了一种在分析历史时区分手动和自动交易的常用方法，但这种方法不完全可靠。事实上，EA 交易也可以使用零标识符。因此，要找出谁以及如何执行特定的交易操作，请使用各自的对应特性，订单对应 **ORDER_REASON**、交易对应 **DEAL_REASON**、仓位对应 **POSITION_REASON**。

每个 EA 交易可以设置自己唯一的 ID，甚至可以出于不同目的而使用几个 ID（按交易策略、信号等细分）。仓位的 **magic** 号对应于参与仓位形成的最后一笔交易的 **magic** 号

symbol 字段中的交易品种名称仅在开仓或加仓时重要，在下达挂单时也是如此。在修改和关闭订单和仓位的情况下将被忽略，但此处有一个小的例外。由于每个交易品种在净额结算账户上只能有一个仓位，因此 **symbol** 字段可用于在更改其保护价格水平（**Stop Loss** 和 **Take Profit**）的请求中识别仓位。

volume 字段的使用方式相同：在即时买入/卖出订单中或创建挂单时需要交易量字段。需要注意的是，操作中的实际交易量将取决于 **执行模式**，并且可能不同于所请求的交易量。

price 字段也有一些限制：为执行模式为 **SYMBOL_TRADE_EXECUTION_MARKET** 或 **SYMBOL_TRADE_EXECUTION_EXCHANGE** 的金融工具发送市场订单（**action** 中的 **TRADE_ACTION_DEAL**）时，将忽略该字段。

stoplimit 字段仅在设置止损订单时才有意义，即当 **type** 字段包含 **ORDER_TYPE_BUY_STOP_LIMIT** 或 **ORDER_TYPE_SELL_STOP_LIMIT** 时。该字段指定当价格达到 **price** 值时下达限价挂单的价格（这个事实可被 MetaTrader 5 服务器跟踪，直到此时挂单才会显示在交易系统中）。

下达挂单时，其到期规则在如下一对字段中设置：**type_time** 和 **expiration**。后者包含 **datetime** 类型的值，仅当 **type_time** 等于 **ORDER_TIME_SPECIFIED** 或 **ORDER_TIME_SPECIFIED_DAY** 时，才会考虑该值。

最后一点，后面几个字段与查询中的仓位标识相关。基于订单（手动或编程）创建的每个新仓位都会获得一个由系统分配的订单号，这是一个唯一编号。通常，其对应于订单的票据，因此会开仓，但可能根据服务器上的服务操作而变化，例如，通过重新开仓来累计掉期。

我们将在单独的章节中讨论如何获取仓位、交易和订单的特性。目前我们需要注意的是，当变更仓位或平仓时，应填写 **position** 字段，以便明确识别。理论上，在净额结算账户上，在 **symbol** 字段中指明仓位交易品种就足够了，但为了算法的统一，最好是保留 **position** 字段的功能。

position_by 字段用于反向平仓（**TRADE_ACTION_CLOSE_BY**）。其应表明一个为相同的交易品种开立的仓位，但方向与 **position** 相反（仅适用于 **对冲账户**）。

deviation 字段仅在“即时执行”和“请求执行”模式下影响市场订单的执行。

在相关章节中将给出填充每种交易操作结构体的示例。

6.4.10 MqlTradeCheckResult 结构体

在向交易服务器发送交易操作请求之前，建议检查该请求是否已完成且没有形式上的错误。检查由 **OrderCheck** 函数执行，将 **MqlTradeRequest** 结构体中的请求和 **MqlTradeCheckResult** 结构体类型的接收变量传递给该函数。

除了请求的正确性之外，该结构体还允许在执行交易操作之后评估账户状态，特别是结余、资金和保证金。

```
struct MqlTradeCheckResult
{
    uint   retcode;      // Response code
    double balance;      // Balance after the transaction
    double equity;       // Equity after the transaction
    double profit;       // Floating profit
    double margin;       // Margin requirements
    double margin_free;  // Free margin
    double margin_level; // Margin level
    string comment;      // Comment to the response code (description of the error)
};
```

下表对这些字段进行了说明。

字 段	说 明
retcode	假定的返回码
balance	执行交易操作后观察到的余额值
equity	执行交易操作后进行观察到的自有资金值
profit	执行交易操作后观察到的浮动盈利值
margin	交易后锁定的总保证金
margin_free	执行交易操作后剩余的自有可用资金量
margin_level	将在执行交易操作后设定的保证金水平
comment	响应代码的注释，错误说明

在通过调用 `OrderCheck` 填充的结构体中，`retcode` 字段将包含一个平台支持的结果代码，用于处理真实交易请求并将类似的 `retcode` 字段放入 `MqlTradeResult` 结构体中（在调用 `OrderSend` 和 `OrderSendAsync` 交易函数之后）。

返回码常量显示在 `SQL5` 文档中。为了在调试 EA 交易时更直观地输出到日志中，在 `TradeRetcode.mqh` 文件中定义了所应用的枚举 `TRADE_RETCODE`。其中的所有元素都有与内置常量相匹配的标识符，但没有公共前缀“`TRADE_RETCODE_`”。例如，

```

enum TRADE_RETCODE
{
    OK_0           = 0,          // no standard constant
    REQUOTE        = 10004,     // TRADE_RETCODE_REQUOTE
    REJECT         = 10006,     // TRADE_RETCODE_REJECT
    CANCEL         = 10007,     // TRADE_RETCODE_CANCEL
    PLACED         = 10008,     // TRADE_RETCODE_PLACED
    DONE           = 10009,     // TRADE_RETCODE_DONE
    DONE_PARTIAL   = 10010,     // TRADE_RETCODE_DONE_PARTIAL
    ERROR          = 10011,     // TRADE_RETCODE_ERROR
    TIMEOUT        = 10012,     // TRADE_RETCODE_TIMEOUT
    INVALID        = 10013,     // TRADE_RETCODE_INVALID
    INVALID_VOLUME = 10014,     // TRADE_RETCODE_INVALID_VOLUME
    INVALID_PRICE  = 10015,     // TRADE_RETCODE_INVALID_PRICE
    INVALID_STOPS  = 10016,     // TRADE_RETCODE_INVALID_STOPS
    TRADE_DISABLED = 10017,     // TRADE_RETCODE_TRADE_DISABLED
    MARKET_CLOSED  = 10018,     // TRADE_RETCODE_MARKET_CLOSED
    ...
};

#define TRCSTR(X) EnumToString((TRADE_RETCODE)(X))

```

因此，使用 `TRCSTR(r.retcode)` (其中 `r` 是一个结构) 可提供数字代码的最小说明。

我们将考虑一个应用宏和分析结构体的示例，在下一节关于 `OrderCheck` 函数的内容中进行介绍。

6.4.11 请求验证: OrderCheck

要执行任何交易操作，MQL 程序必须首先用必要的的数据填充 `MqlTradeRequest` 结构体。在使用交易函数将其发送到服务器之前，必须检查其形式上的正确性，并评估该请求的结果，特别是需要的保证金金额和剩余的可用资金。这项检查由 `OrderCheck` 函数执行。

```
bool OrderCheck(const MqlTradeRequest &request, MqlTradeCheckResult &result)
```

如果没有足够的资金或参数填写不正确，函数将返回 `false`。此外，当交易被禁用时，该函数还会对拒绝作出反应，无论是在整个终端中还是针对特定程序。有关错误代码，请检查 `result` 结构体的 `retcode` 字段。

`request` 结构体的成功检查和交易环境以 `true` 状态结束，但是，如果使用 `OrderSend` 或 `OrderSendAsync` 函数重复请求的操作，并不能保证请求的操作一定会成功。在不同调用之间，交易条件可能已发生变化，或者服务器上的经纪人可能具有应用于特定外部交易系统的设置，而 `OrderCheck` 执行的正式验证算法无法满足这些设置。

要获得预期财务结果的说明，应分析 `result` 结构体的字段。

函数 `OrderCalcMargin` 仅计算一个拟定仓位所需要的估算保证金，而 `OrderCheck` 则不同，是以简化模式考虑了交易账户的一般状态。所以，该函数用订单执行后形成的累积变量填充 `MqlTradeCheckResult` 结构体中的 `margin` 字段和其他相关字段 (`margin_free` 和 `margin_level`)。例如，如果在调用 `OrderCheck`，时任何金融工具已建仓，并且正在检查的请求增加仓位，则 `margin` 字段将反映存款的金额，包括以前的保证金负债。如果新订单包含相反方向的操作，保证金将不会增加 (实际上，保证金应减少，因为仓位应在净额结算账户上完全平仓，对冲保证金须应用于对冲账户上的相反仓位；但是，该函数不会执行如此精确的计算)。

首先, **OrderCheck** 对于程序员在熟悉交易 API 的初始阶段非常有用,可在不将请求发送到服务器的情况下进行实验。

我们使用一个简单的非交易性 EA 交易 **CustomOrderCheck.mq5** 来测试 **OrderCheck** 函数的性能。为了便于使用,我们将其作为 EA 交易而不是脚本:这样,在使用当前设置启动后 EA 交易还会留在图表上,可以通过更改单个输入参数轻松编辑。而对于脚本,我们必须每次从默认值开始设置这些字段。

要运行检查,让我们在 **OnInit** 中设置一个计时器。

```
void OnInit()
{
    // initiate pending execution
    EventSetTimer(1);
}
```

至于计时器处理程序,主要算法将在此处实现。一开始,我们取消了计时器,因为我们需要代码执行一次,然后等待用户更改参数。

```
void OnTimer()
{
    // execute the code once and wait for new user settings
    EventKillTimer();
    ...
}
```

EA 交易的输入参数完全重复交易请求结构体的字段集。

```
input ENUM_TRADE_REQUEST_ACTIONS Action = TRADE_ACTION_DEAL;
input ulong Magic;
input ulong Order;
input string Symbol;    // Symbol (empty = current _Symbol)
input double Volume;    // Volume (0 = minimal lot)
input double Price;     // Price (0 = current Ask)
input double StopLimit;
input double SL;
input double TP;
input ulong Deviation;
input ENUM_ORDER_TYPE Type;
input ENUM_ORDER_TYPE_FILLING Filling;
input ENUM_ORDER_TYPE_TIME ExpirationType;
input datetime ExpirationTime;
input string Comment;
input ulong Position;
input ulong PositionBy;
```

其中的许多字段不影响检查和财务计算,但保留了下来以便你验证。

默认情况下,变量的状态对应于以当前金融工具的最小手数建仓的请求。特别是没有显式初始化的 **Type** 参数会得到 0 值,等于 **ENUM_ORDER_TYPE** 结构体的 **ORDER_TYPE_BUY** 成员。在 **Action** 参数中,我们指定了显式初始化,因为 0 不对应于 **ENUM_TRADE_REQUEST_ACTIONS** 枚举的任何元素 (**TRADE_ACTION_DEAL** 的第一个元素为 1)。

```

void OnTimer()
{
    ...
    // initialize structures with zeros
    MqlTradeRequest request = {};
    MqlTradeCheckResult result = {};

    // default values
    const bool kindOfBuy = (Type & 1) == 0;
    const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
    const double volume = Volume == 0 ?
        SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : Volume;
    const double price = Price == 0 ?
        SymbolInfoDouble(symbol, kindOfBuy ? SYMBOL_ASK : SYMBOL_BID) : Price;
    ...
}

```

我们来填充该结构体。真正的机器人通常只需要为几个字段赋值，但是由于这个测试是通用的，我们必须确保用户输入的任何参数都能通过。

```

request.action = Action;
request.magic = Magic;
request.order = Order;
request.symbol = symbol;
request.volume = volume;
request.price = price;
request.stoptlimit = StopLimit;
request.sl = SL;
request.tp = TP;
request.deviation = Deviation;
request.type = Type;
request.type_filling = Filling;
request.type_time = ExpirationType;
request.expiration = ExpirationTime;
request.comment = Comment;
request.position = Position;
request.position_by = PositionBy;
...

```

请注意，此处我们还没有标准化价格和手数，尽管在真实程序中必须这样做。因此，这个测试可以输入“不均匀”值，并确保它们会导致错误。在下面的示例中，将启用规范化。

然后我们调用 *OrderCheck* 并记录 *request* 和 *result* 结构体。我们仅关注后者的 *retcode* 字段，所以使用宏 *TRCSTR (TradeRetcode.mqh)* 将其额外打印为文本形式的“解密”结果。你还可以分析 *comment* 字符串字段，但是其格式可能会改变，以便更适合向用户显示。


```

ResetLastError();
PRTF(OrderCheck(request, result));
StructPrint(request, ARRAYPRINT_HEADER);
Print(TRCSTR(result.retcode));
StructPrint(result, ARRAYPRINT_HEADER, 2);
...

```

结构体的输出由基于 *ArrayPrint* 的 *StructPrint* 辅助函数提供。因此，我们仍然会得到原始的数据显示。具体来说，枚举的元素以数字原样表示。稍后，我们将开发一个更透明（用户友好）的 *MqlTradeRequest* 结构体输出函数（参见 *TradeUtils.mqh*）。

为了便于分析结果，在 *OnTimer* 函数的开始，我们将显示账户的当前状态，最后，为了便于比较，我们将使用 *OrderCalcMargin* 函数计算给定交易操作的保证金。

```

void OnTimer()
{
    PRTF(AccountInfoDouble(ACCOUNT_EQUITY));
    PRTF(AccountInfoDouble(ACCOUNT_PROFIT));
    PRTF(AccountInfoDouble(ACCOUNT_MARGIN));
    PRTF(AccountInfoDouble(ACCOUNT_MARGIN_FREE));
    PRTF(AccountInfoDouble(ACCOUNT_MARGIN_LEVEL));
    ...
    // filling in the structure MqlTradeRequest
    // calling OrderCheck and printing results
    ...
    double margin = 0;
    ResetLastError();
    PRTF(OrderCalcMargin(Type, symbol, volume, price, margin));
    PRTF(margin);
}

```

以下是使用默认设置的 XAUUSD 的日志示例。

```

AccountInfoDouble(ACCOUNT_EQUITY)=15565.22 / ok
AccountInfoDouble(ACCOUNT_PROFIT)=0.0 / ok
AccountInfoDouble(ACCOUNT_MARGIN)=0.0 / ok
AccountInfoDouble(ACCOUNT_MARGIN_FREE)=15565.22 / ok
AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)=0.0 / ok
OrderCheck(request,result)=true / ok
[action] [magic] [order] [symbol] [volume] [price] [stoplimit] [sl] [tp] [deviation]
1 0 0 "XAUUSD" 0.01 1899.97 0.00 0.00 0.00 0 0 »
» [type_filling] [type_time] [expiration] [comment] [position] [position_by] [reserve]
» 0 0 1970.01.01 00:00:00 "" 0 0 0
OK_0
[retcode] [balance] [equity] [profit] [margin] [margin_free] [margin_level] [comment]
0 15565.22 15565.22 0.00 19.00 15546.22 81922.21 "Done" 0
OrderCalcMargin(Type,symbol,volume,price,margin)=true / ok
margin=19.0 / ok

```

下一个示例显示了对账户保证金预期增长的估算，其中已经有一个未结头寸，我们将对其翻倍。

```

AccountInfoDouble(ACCOUNT_EQUITY)=9999.5400000000001 / ok
AccountInfoDouble(ACCOUNT_PROFIT)=-0.83 / ok
AccountInfoDouble(ACCOUNT_MARGIN)=79.22 / ok
AccountInfoDouble(ACCOUNT_MARGIN_FREE)=9920.32 / ok
AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)=12622.49431961626 / ok
OrderCheck(request,result)=true / ok
[action] [magic] [order] [symbol] [volume] [price] [stoplimit] [sl] [tp] [deviation]
1 0 0 "PLZL.MM" 1.0 12642.0 0.0 0.0 0.0 0 0 »
» [type_filling] [type_time] [expiration] [comment] [position] [position_by] [reserve]
» 0 0 1970.01.01 00:00:00 "" 0 0 0
OK_0
[retcode] [balance] [equity] [profit] [margin] [margin_free] [margin_level] [comment]
0 10000.87 9999.54 -0.83 158.26 9841.28 6318.43 "Done" 0
OrderCalcMargin(Type,symbol,volume,price,margin)=true / ok
margin=79.040000000000001 / ok

```

尝试更改任何请求参数，看看请求是否成功。不正确的参数组合将导致 [标准列表](#) 中的错误代码，但是由于无效选项比保留选项多得多（最常见的错误），该函数通常会返回通用代码 `TRADE_RETCODE_INVALID`（10013）。在这方面，建议使用更高程度的诊断来实现你自己的结构体检查。

向服务器发送真实请求时，可在各种不可预见的情况下使用相同的 `TRADE_RETCODE_INVALID` 代码，例如，当试图重新编辑一个在外部交易系统中已经开始（但尚未完成）修改操作的订单时。

6.4.12 请求发送结果：`MqlTradeResult` 结构体

为了响应由 `OrderSend` 或 `OrderSendAsync` 函数（将在下一章节中介绍这两个函数）执行的交易请求，服务器应返回请求处理结果。为此，使用了一个特殊的预定义结构体 `MqlTradeResult`。

```

struct MqlTradeResult
{
    uint    retcode;           // Operation result code
    ulong   deal;             // Transaction ticket, if it is completed
    ulong   order;            // Order ticket, if it is placed
    double  volume;           // Trade volume confirmed by the broker
    double  price;            // Trade price confirmed by the broker
    double  bid;              // Current market bid price
    double  ask;              // Current market offer price
    string  comment;          // Broker's comment on the operation
    uint    request_id;       // Request identifier, set by the terminal when sending
    uint    retcode_external; // Response code of the external trading system
};

```

下表对其字段进行了说明。

字段	说明
retcode	交易服务器返回码
deal	执行时的交易订单号 (在 <code>TRADE_ACTION_DEAL</code> 交易操作期间)
order	下达的订单号 (在 <code>TRADE_ACTION_PENDING</code> 交易操作期间)
volume	经纪商确认的交易量 (取决于订单 执行模式)
price	由经纪商确认的交易价格 (取决于 <code>deviation</code> 字段, 该字段包含在 交易请求中、执行模式和交易操作中)
bid	当前市场买价
ask	当前市场卖价
comment	经纪商对交易的注释 (默认情况下, 用解密的交易服务器返回代码填充)
request_id	请求 ID, 由终端在发送给交易服务器时设置
retcode_external	外部交易系统返回的错误代码

正如我们将在下面看到的, 在进行交易操作时, `MqlTradeResult` 类型的变量作为 `OrderSend` 或 `OrderSendAsync` 函数中的第二个参数通过引用传递。该变量会返回结果。

向服务器发送交易请求时, 终端将 `request_id` 标识符设置为唯一值。在随后进行交易事件分析时需要这个标识符 (使用异步函数 `OrderSendAsync` 时必须进行分析)。此标识符允许你将发送的请求与其传递给 `OnTradeTransaction` 事件处理程序的处理结果相关联。

`retcode_external` 字段中错误的存在和类型取决于经纪商和转发交易操作的外部交易系统。

根据交易操作和发送方式的不同, 请求结果会以不同的方式进行分析。我们将在随后的具体操作部分处理这个问题: 市场买卖、下单和撤单、修改仓位和平仓。

6.4.13 发送交易请求: `OrderSend` 和 `OrderSendAsync`

为了执行交易操作, MQL5 API 提供了两个函数: `OrderSend` 和 `OrderSendAsync`。 `OrderSend` 和 `OrderSendAsync`。就像 `OrderCheck` 一样, 它们对以 `MqlTradeRequest` 结构体形式传递的请求参数执行形式检查, 如果成功, 就将请求发送到服务器。

这两个函数的区别如下。 `OrderSend` 期望订单在服务器上排队等待处理, 从中接收有意义的数
据, 然后将数据填入 `MqlTradeResult` 结构体的字段中, 该结构体作为第二个函数参数传递。无论服务器如何响应, `OrderSendAsync` 都会立即将控制权返回给调用代码。同时, 除 `retcode` 之外, `MqlTradeResult` 结构体的所有字段中的重要信息都只填充到 `request_id` 中。使用此请求标识符, MQL 程序可以在 `OnTradeTransaction` 事件中接收有关处理此请求进度的更多信息。另一种方法是定期分析订单、交易和仓位列表。这种方法也可以在一个循环中完成, 在出现通信问题时设置一些超时。

值得注意的是, 尽管第二个函数的名称中有 `Async` (异步) 后缀, 但没有这个后缀的第一个函数也不是完全同步的。事实是, 服务器处理订单的结果, 特别是一笔交易的执行 (也可能是基于一

笔订单的几笔交易)和开仓,通常是在外部交易系统中异步发生。因此, **OrderSend** 函数还需要延迟收集和分析请求执行的结果,如果有必要,必须由 MQL 程序自己实现。我们将查看一个真正同步发送请求并在稍后接收其所有结果的示例 (参见 *MqlTradeSync.mqh*)。

bool OrderSend(const MqlTradeRequest &request, MqlTradeResult &result)

如果在终端中对 **request** 结构体进行了成功的基本检查,并且在服务器上进行了一些附加检查,则该函数返回 **true**。但是,这仅表示服务器接受订单,并不保证交易操作的成功执行。

如果在服务器格式化对 **OrderSend** 调用的响应时,该数据已知,则交易服务器可以在返回的 **result** 结构体中填充 **deal** 或 **order** 字段值。但是,在一般情况下,执行交易或发出与订单相对应的限价订单事件可以在响应被发送到终端中的 MQL 程序之后进行。因此,对于任何类型的交易请求,当接收到执行 **OrderSend** 结果时,必须检查返回的 **result** 结构体中提供的交易服务器返回代码 **retcode** 和外部交易系统响应代码 **retcode_external** (如有必要)。你应根据这两个代码决定是等待服务器上的挂起操作,还是采用你自己的操作。

每个接受的订单都存储在交易服务器上等待处理,直到发生下列任何影响其生命周期的事件:

- 当计数器请求出现时执行
- 当执行价格到达时触发
- 到期日期
- 用户或 MQL 程序取消
- 经纪商删除 (例如,在清算或资金短缺的情况下 **Stop Out**)

OrderSendAsync 原型完全重复 **OrderSend** 的原型。

bool OrderSendAsync(const MqlTradeRequest &request, MqlTradeResult &result)

该函数用于高频交易,根据算法的条件,浪费时间等待服务器的响应是不可接受的。使用 **OrderSendAsync** 不会加快服务器处理请求或向外部交易系统发送请求的速度。

注意! 在测试程序中, **OrderSendAsync** 函数的工作方式类似于 **OrderSend**。这使得调试异步请求的挂起处理变得困难。

当请求成功发送到 MetaTrader 5 服务器时,该函数返回 **true**。但是,这并不意味着请求到达了服务器并被接受进行处理。同时,收到的 **result** 结构体中的响应代码包含 **TRADE_RETCODE_PLACED** (10008) 值,即“订单已下达”。

在处理接收到的请求时,服务器将向终端发送一条关于仓位、订单和交易当前状态变化的响应消息,这将导致 MQL 程序中生成 **OnTrade** 事件。此时,程序可以分析新的交易环境和账户历史。我们将在下面看到相关示例。

此外,可以使用 **OnTradeTransaction** 处理程序跟踪服务器上交易者请求执行的细节。同时要注意,执行一个交易请求后,会多次调用 **OnTradeTransaction** 处理程序。例如,当发送市场买入请求时,该请求会被服务器接受进行处理,为账户创建相应的“买入”订单,订单被执行并且交易被执行,结果该订单被从未结订单列表中移除并且被添加到订单历史中。然后,交易被添加到历史中,并创建一个新的仓位。对于所有这些事件,将调用 **OnTradeTransaction** 函数。

让我们从一个简单的 EA 交易示例 *CustomOrderSend.mq5* 开始。该示例允许你在输入参数中设置请求的所有字段,类似于 *CustomOrderCheck.mq5*,但进一步的区别在于其会向服务器发送请求,而不是在终端中进行简单的检查。在你的模拟账户上运行 EA 交易。完成实验后,不要忘记从图表中删除 EA 交易或关闭图表,这样就不会在每次终端启动时都发送测试请求。

这个新示例还有其他一些改进。首先,添加了 **Async** 输入参数。

```
input bool Async = false;
```

此选项允许选择用于将请求发送到服务器的函数。默认情况下，该参数等于 *false*，并且使用 *OrderSend* 函数。如果将其设置为 *true*，则会调用 *OrderSendAsync*。

此外，通过这个示例，我们将开始说明并完成 *TradeUtils.mqh* 头文件中的一组特殊函数，这些函数用于简化机器人的编码。所有函数都放在名称空间 TU 中（是“Trade Utilities”的缩写），首先，我们引入一些函数，以便于向结构体日志输出 *MqlTradeRequest* 和 *MqlTradeResult*。

```

namespace TU
{
    string StringOf(const MqlTradeRequest &r)
    {
        SymbolMetrics p(r.symbol);

        // main block: action, type, symbol
        string text = EnumToString(r.action);
        if(r.symbol != NULL) text += ", " + r.symbol;
        text += ", " + EnumToString(r.type);
        // volume block
        if(r.volume != 0) text += ", V=" + p.StringOf(r.volume, p.lotDigits);
        text += ", " + EnumToString(r.type_filling);
        // block of all prices
        if(r.price != 0) text += ", @ " + p.StringOf(r.price);
        if(r.stoplimit != 0) text += ", X=" + p.StringOf(r.stoplimit);
        if(r.sl != 0) text += ", SL=" + p.StringOf(r.sl);
        if(r.tp != 0) text += ", TP=" + p.StringOf(r.tp);
        if(r.deviation != 0) text += ", D=" + (string)r.deviation;
        // pending orders expiration block
        if(IsPendingType(r.type)) text += ", " + EnumToString(r.type_time);
        if(r.expiration != 0) text += ", " + TimeToString(r.expiration);
        // modification block
        if(r.order != 0) text += ", #=" + (string)r.order;
        if(r.position != 0) text += ", #P=" + (string)r.position;
        if(r.position_by != 0) text += ", #b=" + (string)r.position_by;
        // auxiliary data
        if(r.magic != 0) text += ", M=" + (string)r.magic;
        if(StringLen(r.comment)) text += ", " + r.comment;

        return text;
    }

    string StringOf(const MqlTradeResult &r)
    {
        string text = TRCSTR(r.retcode);
        if(r.deal != 0) text += ", D=" + (string)r.deal;
        if(r.order != 0) text += ", #=" + (string)r.order;
        if(r.volume != 0) text += ", V=" + (string)r.volume;
        if(r.price != 0) text += ", @ " + (string)r.price;
        if(r.bid != 0) text += ", Bid=" + (string)r.bid;
        if(r.ask != 0) text += ", Ask=" + (string)r.ask;
        if(StringLen(r.comment)) text += ", " + r.comment;
        if(r.request_id != 0) text += ", Req=" + (string)r.request_id;
        if(r.retcode_external != 0) text += ", Ext=" + (string)r.retcode_external;

        return text;
    }
    ...
};

```

这些函数的目的是以简洁且方便的形式提供所有重要的（非空）字段：它们显示在一行中，每个字段都有一个唯一的名称。

可以看到，该函数为 *MqlTradeRequest* 使用了 *SymbolMetrics* 类。这样有助于将同一金融工具的多个价格或交易量进行标准化。不要忘记，价格和交易量的标准化是准备正确交易请求的先决条件。

```
class SymbolMetrics
{
public:
    const string symbol;
    const int digits;
    const int lotDigits;

    SymbolMetrics(const string s): symbol(s),
        digits((int)SymbolInfoInteger(s, SYMBOL_DIGITS)),
        lotDigits((int)MathLog10(1.0 / SymbolInfoDouble(s, SYMBOL_VOLUME_STEP)))
    { }

    double price(const double p)
    {
        return TU::NormalizePrice(p, symbol);
    }

    double volume(const double v)
    {
        return TU::NormalizeLot(v, symbol);
    }

    string StringOf(const double v, const int d = INT_MAX)
    {
        return DoubleToString(v, d == INT_MAX ? digits : d);
    }
};
```

这些值的直接标准化委托给 *NormalizePrice* 和 *NormalizeLot* 辅助函数（后者的方案与我们在文件 *LotMarginExposure.mqh* 中看到的完全相同）。

```
double NormalizePrice(const double price, const string symbol = NULL)
{
    const double tick = SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_SIZE);
    return MathRound(price / tick) * tick;
}
```

如果我们连接 *TradeUtils.mqh*，示例 *CustomOrderSend.mq5* 具有以下形式（省略的代码片段 ‘...’ 与 *CustomOrderCheck.mq5* 相同）。

```

void OnTimer()
{
    ...
    MqlTradeRequest request = {};
    MqlTradeCheckResult result = {};

    TU::SymbolMetrics sm(symbol);

    // fill in the request structure
    request.action = Action;
    request.magic = Magic;
    request.order = Order;
    request.symbol = symbol;
    request.volume = sm.volume(volume);
    request.price = sm.price(price);
    request.stoplimit = sm.price(StopLimit);
    request.sl = sm.price(SL);
    request.tp = sm.price(TP);
    request.deviation = Deviation;
    request.type = Type;
    request.type_filling = Filling;
    request.type_time = ExpirationType;
    request.expiration = ExpirationTime;
    request.comment = Comment;
    request.position = Position;
    request.position_by = PositionBy;

    // send the request and display the result
    ResetLastError();
    if(Async)
    {
        PRTF(OrderSendAsync(request, result));
    }
    else
    {
        PRTF(OrderSend(request, result));
    }
    Print(TU::StringOf(request));
    Print(TU::StringOf(result));
}

```

由于价格和交易量现已标准化，你可以尝试在相应的输入参数中输入不规则的值。它们通常是在计算过程中在程序中获得的，我们的代码可根据交易品种规范对其进行转换。

在默认设置下，EA 交易会创建一个按市场买入当前金融工具的最小手数请求，并使用 *OrderSend* 函数进行请求。

```

OrderSend(request,result)=true / ok
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.12462
DONE, D=1250236209, #=1267684253, V=0.01, @ 1.12462, Bid=1.12456, Ask=1.12462, Reques

```

一般而言，在允许交易的情况下，该操作应成功完成（状态为“DONE”，注释为“Request executed”）。在 *result* 结构体中，我们会立即收到交易编号 *D*。

如果我们打开 EA 交易设置并将 `Async` 参数的值替换为 `true`，我们将发送一个类似的请求，但会使用 `OrderSendAsync` 函数。

```
OrderSendAsync(request,result)=true / ok
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.12449
PLACED, Order placed, Req=2
```

在这种情况下，状态为“PLACED”，函数返回时的交易编号未知。我们只知道唯一的请求 ID `Req=2`。要获得交易和仓位编号，你需要在 `OnTradeTransaction` 处理程序中截取具有相同请求 ID 的 `TRADE_TRANSACTION_REQUEST` 消息，其中填充的结构体将作为 `MqlTradeResult` 参数接收。

从用户的角度来看，这两个请求应一样快。

这两个函数的性能可以在 MQL 程序的代码中使用 EA 交易的另一个示例直接进行比较（参见 [同步和异步请求](#) 一节），我们将在研究交易事件模型后探讨这一点。

应注意的是，交易事件被发送到 `OnTradeTransaction` 处理程序（如果代码中有），不管哪个函数用于发送请求：`OrderSend` 或 `OrderSendAsync`。情况如下：若应用 `OrderSend`，有关订单执行的部分或全部信息在接收 `MqlTradeResult` 结构体中立即可用。但一般情况下，其结果是随时间和交易量分布的，例如，当一个订单被“填充”成几个交易时。那么可以从交易事件中或者通过分析交易和订单的历史来获得完整的信息。

如果你尝试故意发送不正确的请求，例如，将订单类型更改为挂单 `ORDER_TYPE_BUY_STOP`，你将会收到一条错误消息，因为对于此类订单，你应使用 `TRADE_ACTION_PENDING` 操作。此外，它们应位于与当前价格有一段距离的位置（我们默认使用市场价格）。在这个测试之前，重要的是不要忘记将查询模式改回同步（`Async=false`），这样在结束 `OrderSend` 调用之后便能立即看到 `MqlTradeResult` 结构体中的错误。否则，`OrderSendAsync` 虽然返回 `true`，但顺序仍然不会被设置，并且程序只能在 `OnTradeTransaction` 中接收到相关信息，而我们还没有这个信息。

```
OrderSend(request,result)=false / TRADE_SEND_FAILED(4756)
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, @ 1.12452,
REQUOTE, Bid=1.12449, Ask=1.12452, Requote, Req=5
```

在这种情况下，错误会报告无效的 Requote 价格。

使用函数执行特定交易操作的示例将在后面几个章节中介绍。

6.4.14 买入和卖出操作

本节中，我们最终开始研究 MQL5 函数在特定交易任务中的应用。这些函数的目的是以特殊的方式填充 `MqlTradeRequest` 结构体，并调用 `OrderSend` 或 `OrderSendAsync` 函数。

我们将学习的第一个操作是以当前市场价格买入或卖出金融工具。执行此操作的过程包括：

- ① 根据提交的订单创建市场订单
- ② 根据订单执行交易（一笔或多笔）
- ③ 结果应为一个未平仓仓位

正如我们在 [交易操作类型](#) 一节中看到的，即时买/卖对应于 `ENUM_TRADE_REQUEST_ACTIONS` 枚举中的 `TRADE_ACTION_DEAL` 元素。因此，在填充 `MqlTradeRequest` 结构体时，应在 `action` 字段中写入 `TRADE_ACTION_DEAL`。

使用 `type` 字段设置交易方向，该字段应包含以下 [订单类型](#) 之一：`ORDER_TYPE_BUY` 或 `ORDER_TYPE_SELL`。

当然，要想买入或卖出，需要在 `symbol` 字段中指定交易品种名称，并在 `volume` 字段中指定其所需的交易量。

`type_filling` 字段必须使用 `ENUM_ORDER_TYPE_FILLING` 枚举中的一个填充策略来填充，该填充策略是基于字符特性 `SYMBOL_FILLING_MODE` 和允许的策略选择的。

一个可选操作是，程序可以用保护价格水平 (`sl` 和 `tp`)、注释 (`comment`) 和 EA 交易 ID (`magic`) 填充字段。

其他字段的内容会有所不同，具体取决于所选交易品种的价格执行模式。在某些模式下，某些字段不起作用。例如，在请求执行和即时执行模式中，具有 `price` 的字段必须用合适的价格（最后已知的买入 `Ask` 和卖出 `Bid` 价格）填充，并且 `deviation` 字段可以包含价格与成功执行交易设定价格之间的最大允许偏差。在交易执行和市场执行中，这些字段可以忽略。为了简化源代码，可以在所有模式下统一填写价格和滑点，但在最后两个选项中，价格仍然会由交易服务器根据该模式的规则进行选择和替换。

此处没有提到的 `MqlTradeRequest` 结构体的其他字段不用于这些交易操作。

下表总结了不同执行模式的字段填充规则。必填字段标有星号，可选字段标有加号。

字段	请求	即时	交易	市场
action	*	*	*	*
symbol	*	*	*	*
volume	*	*	*	*
type	*	*	*	*
type_filling	*	*	*	*
price	*	*		
sl	+	+	+	+
tp	+	+	+	+
deviation	+	+		
magic	+	+	+	+
comment	+	+	+	+

根据服务器设置，可能禁止在开仓时填写具有保护性 `sl` 和 `tp` 水平的字段。这是交易执行或市场执行模式的常见情况，但是 MQL5 API 没有提供预先澄清这种情况的特性。在这种情况下，**Stop Loss** 和 **Take Profit** 应通过 **修改** 已经开仓的仓位来设置。顺便说一下，这种方法是所有执行模式的推荐方法，因为这是唯一一种允许你从实际开盘价精确推迟保护水平的方法。另一方面，在两个步骤中创建和设置仓位可能会导致这样的情况，即该仓位处于开仓状态，但第二个设置保护水平的请求由于这样或那样的原因而失败。

无论交易方向如何 (买入/卖出), **Stop Loss** 订单始终作为止损订单 (`ORDER_TYPE_BUY_STOP` 或 `ORDER_TYPE_SELL_STOP`) 下达, **Take Profit** 单作为限价订单 (`ORDER_TYPE_BUY_LIMIT` 或 `ORDER_TYPE_SELL_LIMIT`) 下达。此外, 止损订单始终由 MetaTrader 5 服务器控制, 只有当价格达到指定水平时, 才会发送到外部交易系统。相反, 限价订单可以直接输出到外部交易系统。具体而言, 这通常是交易所交易工具的情况。

为了简化交易操作的编码, 不仅是买入和卖出, 还有所有其他操作, 我们将从这一节开始, 通过开发类或结构体, 为交易请求提供自动和正确的字段填充以及真正同步的结果等待。后者尤其重要, 因为 **OrderSend** 和 **OrderSendAsync** 函数会在交易操作完全完成之前将控制权返回给调用代码。特别是, 对于市场买卖, 该算法通常不需要知道服务器上创建的订单号, 而是需要知道仓位开仓与否。例如, 根据这一点, 如果已经开仓, 可以通过设置 **Stop Loss** 和 **Take Profit** 来修改仓位, 如果订单被拒绝, 则重复尝试开仓。

稍后我们将了解 **OnTrade** 和 **OnTradeTransaction** 交易事件, 这些事件可通知程序账户状态的变化, 包括订单、交易和仓位的状态。但是, 将算法分成两个片段 - 根据某些信号或规则单独生成订单, 并在事件处理程序中单独分析情况 - 会使代码不太容易理解和维护。

理论上, 无论是在速度上还是在编码的容易程度上, 异步编程模式都不比同步编程模式差。但是, 其实现方式可以不同, 例如, 基于回调函数 (Java、JavaScript 和许多其他语言中的一种基本技术) 或事件 (如 MQL5 中) 的直接指针预先确定了一些功能, 将在 **OnTradeTransaction** 一节中讨论。由于对请求执行的延迟控制, 异步模式允许你加快请求的发送。但这种控制迟早仍需要在同一线程中完成, 因此回路的平均性能是相同的。

所有新的结构体都将放在 **MqlTradeSync.mqh** 文件中。为了不“另起炉灶”, 我们以内置的 MQL5 结构体为起点, 将我们的结构体描述为子结构体。例如, 要获得查询结果, 我们定义 **MqlTradeResultSync**, 该函数是从 **MqlTradeResult** 派生而来的。此处, 我们将添加有用的字段和方法, 特别是 **position** 字段, 用于存储作为市场买入或卖出操作结果的未结仓位订单号。

```
struct MqlTradeResultSync: public MqlTradeResult
{
    ulong position;
    ...
};
```

第二项重要改进将是一个重置所有字段的构造函数 (这使我们在说明结构体类型的变量时不必指定显式初始化)。

```
MqlTradeResultSync()
{
    ZeroMemory(this);
}
```

接下来, 我们将引入一种通用的同步机制, 即等待请求的结果 (每种类型的请求都有自己的检查准备情况规则)。

我们来定义 **condition** 回调函数的类型。这种类型的函数必须采用 **MqlTradeResultSync** 结构体参数, 如果成功, 则返回 **true**: 操作结果被接收。

```
typedef bool (*condition)(MqlTradeResultSync &ref);
```

像这样的函数应传递给 **wait** 方法, 该方法在预定义的超时时间 (以毫秒为单位) 内执行结果准备情况的循环检查。

```

bool wait(condition p, const ulong msc = 1000)
{
    const ulong start = GetTickCount64();
    bool success;
    while(!(success = p(this)) && GetTickCount64() - start < msc);
    return success;
}

```

我们马上澄清一下，超时为最大等待时间：即使设置为非常大的值，只要收到结果，该循环就会立即结束，这可能瞬间发生。当然，有意义的超时不应超过几秒钟。

我们看一个方法的示例，该方法将用于同步等待订单出现在服务器上（与什么状态无关：状态分析是调用代码的任务）。

```

static bool orderExist(MqlTradeResultSync &ref)
{
    return OrderSelect(ref.order) || HistoryOrderSelect(ref.order);
}

```

此处应用了两个内置的 MQL5 API 函数 *OrderSelect* 和 *HistoryOrderSelect*：二者在终端内部交易环境中根据订单号搜索和逻辑选择订单。首先，确认了订单的存在（如果其中一个函数返回 *true*），其次，允许你使用其他函数读取其特性，这对我们而言还不重要。我们将在单独的章节中介绍所有这些特性。这两个函数是一起编写的，因为一个市场订单可以很快执行，它的活动阶段（进入 *OrderSelect*）将立即流入历史（*HistoryOrderSelect*）。

请注意，该方法声明为静态。这是因为 MQL5 不支持指向对象方法的指针。如果是这种情况，我们可以在使用指向 *condition* 回调函数的指针原型时将方法声明为非静态，而不使用引用 *MqlTradeResultSync* 的参数（因为所有字段都存在于 *this* 对象中）。

等待机制可以按如下方式启动：

```

if(wait(orderExist))
{
    // there is an order
}
else
{
    // timeout
}

```

当然，这个片段必须在我们从服务器收到状态为 *TRADE_RETCODE_DONE* 或 *TRADE_RETCODE_DONE_PARTIAL* 的结果后执行，*MqlTradeResultSync* 结构体中的 *order* 字段保证包含一个订单号。请注意，由于系统的分布式特性，来自服务器的订单可能不会立即显示在终端环境中。这就是为什么需要等待时间的原因。

只要 *orderExist* 函数向 *wait* 方法返回 *false*，内部的等待循环就会运行，直到超时到期。一般情况下，我们几乎会瞬间在终端环境中找到一个订单，该循环会以一个成功的标志结束（*true*）。

positionExist 函数以类似但更复杂的方式检查是否存在未结仓位。由于之前的 *orderExist* 函数已经完成了对订单的检查，因此包含在结构体的 *ref.order* 字段中的订单号被确认为有效。

```

static bool positionExist(MqlTradeResultSync &ref)
{
    ulong posid, ticket;
    if(HistoryOrderGetInteger(ref.order, ORDER_POSITION_ID, posid))
    {
        // in most cases, the position ID is equal to the ticket,
        // but not always: the full code implements getting a ticket by ID,
        // for which there are no built-in MQL5 tools
        ticket = posid;

        if(HistorySelectByPosition(posid))
        {
            ref.position = ticket;
            ...
            return true;
        }
    }
    return false;
}

```

使用内置的 *HistoryOrderGetInteger* 和 *HistorySelectByPosition* 函数,我们可以根据订单获取仓位 ID 和订单号。

稍后我们将看到在验证买入/卖出请求时 *orderExist* 和 *positionExist* 的使用,但是现在我们来看另一个结构体: *MqlTradeRequestSync*。该结构体也是从内置结构体继承而来的,并包含额外的字段,特别是一个带有结果的结构体 (以免在调用代码中对其进行说明)和同步请求的超时。

```

struct MqlTradeRequestSync: public MqlTradeRequest
{
    MqlTradeResultSync result;
    ulong timeout;
    ...
}

```

因为新结构体的继承字段是公共的,所以 MQL 程序可以显式地为其赋值,就像对标准 *MqlTradeRequest* 结构体的操作那样。我们添加的用于执行交易操作的方法将考虑、检查并在必要时纠正这些有效值。

在构造函数中,如果省略了参数,我们应重置所有字段并将交易品种设置为默认值。

```

MqlTradeRequestSync(const string s = NULL, const ulong t = 1000): timeout(t)
{
    ZeroMemory(this);
    symbol = s == NULL ? _Symbol : s;
}

```

理论上,由于该结构体的所有字段都是公共的,所以从技术上讲,它们可以被直接赋值,但是对于那些需要验证并且实现了 setter 方法的字段,我们不建议这样做:它们将在执行交易操作之前调用。这些方法中的第一个是 *setSymbol*。

该方法用于填充 *symbol* 字段,以确保发送的交易代码存在,并启动交易量填充模式的后续设置。

```

bool setSymbol(const string s)
{
    if(s == NULL)
    {
        if(symbol == NULL)
        {
            Print("symbol is NULL, defaults to " + _Symbol);
            symbol = _Symbol;
            setFilling();
        }
        else
        {
            Print("new symbol is NULL, current used " + symbol);
        }
    }
    else
    {
        if(SymbolInfoDouble(s, SYMBOL_POINT) == 0)
        {
            Print("incorrect symbol " + s);
            return false;
        }
        if(symbol != s)
        {
            symbol = s;
            setFilling();
        }
    }
    return true;
}

```

因此，使用 *setSymbol* 更改交易品种将通过 *setFilling* 的嵌套调用自动选择正确的填充模式。

setFilling 方法可根据 SYMBOL_FILLING_MODE 和 SYMBOL_TRADE_EXEMODE 交易品种特性提供交易量填充方法的自动规范（请参阅 [交易条件和订单执行模式](#) 一节）。

```

private:
    void setFilling()
    {
        const int filling = (int)SymbolInfoInteger(symbol, SYMBOL_FILLING_MODE);
        const bool market = SymbolInfoInteger(symbol, SYMBOL_TRADE_EXEMODE)
            == SYMBOL_TRADE_EXECUTION_MARKET;

        // the field may already be filled
        // and bit match means a valid mode
        if(((type_filling + 1) & filling) != 0
            || (type_filling == ORDER_FILLING_RETURN && !market)) return;

        if((filling & SYMBOL_FILLING_FOK) != 0)
        {
            type_filling = ORDER_FILLING_FOK;
        }
        else if((filling & SYMBOL_FILLING_IOC) != 0)
        {
            type_filling = ORDER_FILLING_IOC;
        }
        else
        {
            type_filling = ORDER_FILLING_RETURN;
        }
    }
}

```

如果 EA 交易没有正确设置 *type_filling* 字段,该方法可隐式地 (没有错误和消息)纠正它。如果你的算法需要一个有保证的特定填充方法,没有它交易是不可能的,请进行适当的编辑来中断这个过程。

对于正在开发的结构体集,假设除了 *type_filling* 字段之外,可以直接仅设置可选字段,对其内容没有具体要求,比如 *magic* 或 *comment*。

在下文中,为了简单起见,许多方法以较短的形式提供。它们包含我们稍后将会看到的操作类型部分以及分支错误检查。

对于买卖操作,我们需要 *price* 和 *volume* 字段;这两个值均应规范化,并检查可接受的范围。这是通过 *setVolumePrices* 方法完成的。

```

bool setVolumePrices(const double v, const double p,
    const double stop, const double take)
{
    TU::SymbolMetrics sm(symbol);
    volume = sm.volume(v);

    if(p != 0) price = sm.price(p);
    else price = sm.price(TU::GetCurrentPrice(type, symbol));

    return setSLTP(stop, take);
}

```

如果未设置交易价格 (*p == 0*),该程序将根据从 *type* 字段读取的方向,自动获取正确类型的当前价格。

虽然 *Stop Loss* 和 *Take Profit* 水平不是必需的,但如果存在,它们也应规范化,这就是为什么将其添加到该方法的参数中的原因。

TU 这个缩写我们已经知道了。它表示 *TradeUtils.mqh* 文件中的名称空间,其有很多有用的函数,包括有关价格和交易量规范化的函数。

sl 和 *tp* 字段的处理由单独的 *setSLTP* 方法执行,因为这不仅在买卖操作中需要,而且在 修改现有仓位时也需要。

```
bool setSLTP(const double stop, const double take)
{
    TU::SymbolMetrics sm(symbol);
    TU::TradeDirection dir(type);

    if(stop != 0)
    {
        sl = sm.price(stop);
        if(!dir.worse(sl, price))
        {
            PrintFormat("wrong SL (%s) against price (%s)",
                TU::StringOf(sl), TU::StringOf(price));
            return false;
        }
    }
    else
    {
        sl = 0; // remove SL
    }

    if(take != 0)
    {
        tp = sm.price(take);
        if(!dir.better(tp, price))
        {
            PrintFormat("wrong TP (%s) against price (%s)",
                TU::StringOf(tp), TU::StringOf(price));
            return false;
        }
    }
    else
    {
        tp = 0; // remove TP
    }
    return true;
}
```

除了对 *sl* 和 *tp* 字段进行规范化和赋值之外,该方法还用于检查水平相对于 *price* 的相互正确位置。为此, *TradeDirection* 类在空间 TU 中进行描述。

它的构造函数允许你指定交易的分析方向:买入或卖出,在这种情况下,很容易识别两个价格的可盈利或无盈利安排。利用这个类,分析可以统一的方式执行,代码中的检查减少了一半,因为不需要单独处理买入和卖出操作。具体来说, *worse* 方法具有两个价格参数 *p1* 和 *p2*, 并且如果价格 *p1* 相对于价格 *p2* 位置更差,即无盈利,则返回 *true*。一个类似的方法 *better* 表示相反

的逻辑：如果价格 *p1* 比价格 *p2* 好，将返回 *true*。例如，对于卖出，最好的价格应位置更低，因为 *Take Profit* 低于当前价格。

```
TU::TradeDirection dir(ORDER_TYPE_SELL);
Print(dir.better(100, 200)); // true
```

现在，如果订单下达不当，*setSLTP* 函数会记录一条警告并中止验证过程，而不尝试更正其值，因为不同的程序可能会有不同的响应。例如，两个通过的 *stop* 和 *take* 水平中，只有一个可能是错误的，然后使用第二个（正确的）可能是有意义的。

例如，你可以通过跳过无效值的赋值（这样保护水平就不会改变）或向结构体中添加一个带有错误标志的字段（对于这样的结构体，应禁止发送请求的尝试，以免服务器加载明显不可能的请求）来改变行为。发送无效请求将会结束，且显示等于 `TRADE_RETCODE_INVALID_STOPS` 的 *retcode* 错误码。

setSLTP 方法还可以通过检查来确保保护水平与当前价格的距离不小于交易品种的 `SYMBOL_TRADE_STOPS_LEVEL` 特性中的点数（如果设置了该特性，即大于 0），并且当其位于 `SYMBOL_TRADE_FREEZE_LEVEL` 冻结区域内时（如果设置了该特性），不请求修改仓位。此处没有显示这些细微差别：它们可以在源代码中找到。

现在我们准备实现一组交易方法。例如，对于具有最完整字段集的买卖操作，我们定义了 *buy* 和 *sell* 方法。

```
public:
    ulong buy(const string name, const double lot, const double p = 0,
              const double stop = 0, const double take = 0)
    {
        type = ORDER_TYPE_BUY;
        return _market(name, lot, p, stop, take);
    }
    ulong sell(const string name, const double lot, const double p = 0,
              const double stop = 0, const double take = 0)
    {
        type = ORDER_TYPE_SELL;
        return _market(name, lot, p, stop, take);
    }
```

如前所述，要设置可选字段，如 *deviation*、*comment* 和 *magic*，应在调用 *buy/sell* 之前直接赋值。这样更加方便，因为 *deviation* 和 *magic* 在大多数情况下仅设置一次，并在随后的查询中使用。

这些方法会返回一个订单号，但下面我们将展示“同步”接收仓位订单号的机制，这将是一个创建或修改的仓位订单号（如果仓位增加或部分平仓）。

方法 *buy* 和 *sell* 仅在 *type* 字段值上有所不同，其他方面都是相同的。这就是为什么一般部分被框定为单独的方法 *_market*。这是我们在 `TRADE_ACTION_DEAL` 中设置 *action* 并调用 *setSymbol* 和 *setVolumePrices* 的位置。

```
private:
    ulong _market(const string name, const double lot, const double p = 0,
        const double stop = 0, const double take = 0)
    {
        action = TRADE_ACTION_DEAL;
        if(!setSymbol(name)) return 0;
        if(!setVolumePrices(lot, p, stop, take)) return 0;
        ...
    }
```

接下来,我们可以只调用 *OrderSend*, 但是考虑到重新报价的可能性 (订单发送期间服务器上的价格更新), 我们将这个调用放在一个循环中。因此, 该方法将能够重试几次, 但不会超过 MAX_REQUOTES 的预设次数 (代码中选择宏为 10)。

```
int count = 0;
do
{
    ZeroMemory(result);
    if(OrderSend(this, result)) return result.order;
    // automatic price selection means automatic processing of requotes
    if(result.retcodes == TRADE_RETCODE_REQUOTE)
    {
        Print("Requote N" + (string)++count);
        if(p == 0)
        {
            price = TU::GetCurrentPrice(type, symbol);
        }
    }
}
while(p == 0 && result.retcodes == TRADE_RETCODE_REQUOTE
    && ++count < MAX_REQUOTES);
return 0;
}
```

因为默认情况下, 金融工具是在结构体构造函数中设置的, 所以我们可以提供几个简化的 *buy/sell* 方法重载, 而无需使用 *symbol* 参数。

```
public:
    ulong buy(const double lot, const double p = 0,
        const double stop = 0, const double take = 0)
    {
        return buy(symbol, lot, p, stop, take);
    }

    ulong sell(const double lot, const double p = 0,
        const double stop = 0, const double take = 0)
    {
        return sell(symbol, lot, p, stop, take);
    }
```

因此, 在最小配置中, 程序调用 *request.buy(1.0)* 就足以进行一手买入操作了。

现在我们回到获取请求最终结果的问题上来，在 `TRADE_ACTION_DEAL` 操作的情况下，这意味着仓位订单号。在 `MqlTradeRequestSync` 结构体中，这个问题由 `completed` 方法解决：对于每种类型的操作，它必须要求嵌套的 `MqlTradeResultSync` 结构体按照操作类型等待其填充。

```
bool completed()
{
    if(action == TRADE_ACTION_DEAL)
    {
        const bool success = result.opened(timeout);
        if(success) position = result.position;
        return success;
    }
    ...
    return false;
}
```

开仓由 `opened` 方法控制。在内部，我们将发现对上述 `wait` 方法的两次调用：第一次针对 `orderExist` 调用，第二次针对 `positionExist` 调用。

```
bool opened(const ulong msc = 1000)
{
    if(retcode != TRADE_RETCODE_DONE
        && retcode != TRADE_RETCODE_DONE_PARTIAL)
    {
        return false;
    }

    if(!wait(orderExist, msc))
    {
        Print("Waiting for order: #" + (string)order);
    }

    if(deal != 0)
    {
        if(HistoryDealGetInteger(deal, DEAL_POSITION_ID, position))
        {
            return true;
        }
        Print("Waiting for position for deal D=" + (string)deal);
    }

    if(!wait(positionExist, msc))
    {
        Print("Timeout");
        return false;
    }
    position = result.position;

    return true;
}
```

当然，只有当 `retcode` 的状态指示成功时，等待订单和仓位出现才有意义。其他状态指操作的错误或取消或特定的中间代码 (`TRADE_RETCODE_PLACED` 和 `TRADE_RETCODE_TIMEOUT`)，这些代

码不附带其他字段中的有用信息。在这两种情况下，会阻止在这个“同步”框架内的进一步处理。

值得注意的是，我们使用的是 *OrderSync*，因此我们依赖于从服务器接收的结构体中订单号的强制性存在。

在某些情况下，系统不仅会发送订单号，还会同时发送交易订单号。然后从交易方面，可以更快地找到仓位。但是即使有关于交易的信息，终端的交易环境也可能暂时没有关于新仓位的信息。这就是为什么应用 *wait(positionExist)* 等待该消息的原因。

对于中间结果，我们来总结一下。创建的结构体允许编写以下代码来买入 1 手当前交易品种：

```
MqlTradeRequestSync request;
if(request.buy(1.0) && request.completed())
{
    Print("OK Position: P=", request.result.position);
}
```

我们只有在有保证开仓并且知道其订单号的情况下，才能进入条件操作块。如果我们仅使用 *buy/sell* 方法，他们可在输出端收到订单号，并且必须自己检查执行情况。如果出现错误，我们将无法进入 *if* 块，服务器代码将包含在 *request.result.retcode* 中。

当我们在以下章节中实现其他交易的方法时，它们可以以类似的“阻止”模式执行，例如，修改止损水平：

```
if(request.adjust(SL, TP) && request.completed())
{
    Print("OK Adjust")
}
```

当然，如果不想在阻止模式下检查操作结果，就不需要调用 *completed*。相反，你可以坚持异步模式，在 *交易事件* 处理程序中分析环境。但即使在这种情况下，*MqlTradeRequestAsync* 结构体对于检查和规范化操作参数也很有用。

我们编写一个测试 EA 交易 *MarketOrderSend.mq5*，将所有这些参数放在一起。输入参数将为交易请求的主要字段和一些可选字段提供输入值。

```
enum ENUM_ORDER_TYPE_MARKET
{
    MARKET_BUY = ORDER_TYPE_BUY, // ORDER_TYPE_BUY
    MARKET_SELL = ORDER_TYPE_SELL // ORDER_TYPE_SELL
};

input string Symbol;           // Symbol (empty = current _Symbol)
input double Volume;           // Volume (0 = minimal lot)
input double Price;            // Price (0 = current Ask)
input ENUM_ORDER_TYPE_MARKET Type;
input string Comment;
input ulong Magic;
input ulong Deviation;
```

ENUM_ORDER_TYPE_MARKET 枚举是标准 *ENUM_ORDER_TYPE* 的子集，引入这个枚举是为了将可用的操作类型限制为两种：市场买入和卖出。

该操作将在计时器上运行一次，方式与前面的示例相同。

```

void OnInit()
{
    // scheduling a delayed start
    EventSetTimer(1);
}

```

在计时器处理程序中，我们应禁用计时器，以便请求仅执行一次。对于下一次发布，你需要更改 EA 交易参数。

```

void OnTimer()
{
    EventKillTimer();
    ...
}

```

我们描述一个 *MqlTradeRequestSync* 类型的变量，并准备好主字段的值。

```

const bool wantToBuy = Type == MARKET_BUY;
const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
const double volume = Volume == 0 ?
    SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : Volume;

MqlTradeRequestSync request(symbol);
...

```

可选字段将直接填写。

```

request.magic = Magic;
request.deviation = Deviation;
request.comment = Comment;
...

```

在可选字段中，你可以选择填充模式 (*type_filling*)。默认情况下，*MqlTradeRequestSync* 会自动将第一个允许的模式 `ENUM_ORDER_TYPE_FILLING` 写入到该字段。注意，该结构体为此有一个特殊的方法 *setFilling*。

接下来，我们用这些参数调用 *buy* 或 *sell* 方法，如果它返回一个订单号，我们就等待一个未结仓位出现。

```

ResetLastError();
const ulong order = (wantToBuy ?
    request.buy(volume, Price) :
    request.sell(volume, Price));
if(order != 0)
{
    Print("OK Order: #=", order);
    if(request.completed()) // waiting for an open position
    {
        Print("OK Position: P=", request.result.position);
    }
}
Print(TU::StringOf(request));
Print(TU::StringOf(request.result));
}

```

在函数结束时，查询和结果结构体被记录下来，以供参考。

如果我们使用默认参数运行 EA 交易 (以最小手数买入当前交易品种), 我们可以得到 “XTIUSD” 的以下结果。

```
OK Order: #=218966930
Waiting for position for deal D=215494463
OK Position: P=218966930
TRADE_ACTION_DEAL, XTIUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 109.340, P=21
DONE, D=215494463, #=218966930, V=0.01, @ 109.35, Request executed, Req=8
```

请注意关于临时缺少仓位的警告: 由于请求的分布式处理, 该警告总会出现 (可以通过删除 EA 交易代码中的 `SHOW_WARNINGS` 宏来禁用警告本身, 但这种情况将保持不变)。但由于开发了新的结构体, 应用代码不会被这些内部复杂性转移, 而是以一系列简单步骤的形式编写, 其中每个下一个步都对前一个步的成功 “有信心”。

在净额结算账户上, 我们可以通过随后以双倍的最小手数 (本例中为 0.02) 卖出, 来实现有趣的仓位反转效果。

```
OK Order: #=218966932
Waiting for position for deal D=215494468
Position ticket <> id: 218966932, 218966930
OK Position: P=218966932
TRADE_ACTION_DEAL, XTIUSD, ORDER_TYPE_SELL, V=0.02, ORDER_FILLING_FOK, @ 109.390, P=2
DONE, D=215494468, #=218966932, V=0.02, @ 109.39, Request executed, Req=9
```

务必要注意的是, 在反转之后, 仓位订单号不再等于仓位标识符: 该标识符源自第一个订单, 而订单号源自第二个订单。为了简化表示, 我们有意绕过通过标识符查找仓位订单号的任务。在大多数情况下, 订单号和 ID 是相同的, 但是为了精确控制, 请使用 `TU::PositionSelectById` 函数。若有兴趣, 可以研究一下附带的源代码。

只要仓位存在 (直到交易量接近于零), 标识符就是恒定的, 并且对于分析账户历史很有用。订单号说明了处于开放状态的仓位 (历史上没有仓位订单号的概念), 用于某些类型的请求, 特别是修改保护水平或反向平仓。但是填充部分也有细微差别。我们将在 [单独的章节](#) 中更详细介绍仓位特性。

当进行买入或卖出操作时, 我们的 *buy/sell* 方法允许你立即设置 *Stop Loss* 和/或 *Take Profit* 水平。要做到这一点, 只需将它们作为附加参数进行传递 (这些参数获取自输入变量或使用一些公式计算得出)。例如,

```
input double SL;
input double TP;
...
void OnTimer()
{
    ...
    const ulong order = (wantToBuy ?
        request.buy(symbol, volume, Price, SL, TP) :
        request.sell(symbol, volume, Price, SL, TP));
    ...
}
```

新结构体的所有方法都提供了传递参数的自动规范化, 因此不需要使用 *NormalizeDouble* 或其他方法。

上面已经指出, 一些服务器设置可能会禁止在开仓时间设置保护水平。在这种情况下, 你应通过单独的请求设置 *sl* 和 *tp* 字段。当需要修改已经设定的水平时, 特别是为了实现跟踪止损或跟踪利润时, 也使用完全相同的请求。

在下一节中，我们将使用成功建仓后的第二个请求来完成 *sl* 和 *tp* 的延迟设置，从而完成当前示例。

6.4.15 修改仓位的止损和/或止盈水平

MQL 程序可以改变未平仓仓位的保护性 *Stop Loss* 和 *Take Profit* 价格水平。`TRADE_ACTION_SLTP` 元素 (在 `ENUM_TRADE_REQUEST_ACTIONS` 枚举中)就是用于这个目的，也就是在填充 *MqlTradeRequest* 结构体时，应在 *action* 字段写入 `TRADE_ACTION_SLTP`。

这是唯一的必填字段。是否需要填写其他字段是由账户操作模式 `ENUM_ACCOUNT_MARGIN_MODE` 决定的。在对冲账户上，应填写 *symbol* 字段，但可以省略仓位订单号。相反，在对冲账户上，必须注明 *position* 仓位订单号，但可以省略交易品种。这是由于不同类型账户的仓位标识方式的特殊性。在净额结算期间，每个交易品种只能有一个仓位。

为了统一代码，建议填写这两个字段 (如果有相关信息)。

保护价格水平在 *sl* 和 *tp* 字段中设置。可以只测试其中一个字段。要删除保护水平，请为其赋零值。

下表总结了根据计数模式填写字段的要求。必填字段标有星号，可选字段标有加号。

字 段	净 额 结 算	对 冲
action	*	*
symbol	*	+
position	+	*
sl	+	+
tp	+	+

为了执行修改保护水平的操作，我们在 *MqlTradeRequestSync* 结构体中引入了 *adjust* 方法的几个重载。

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
    ...
    bool adjust(const ulong pos, const double stop = 0, const double take = 0);
    bool adjust(const string name, const double stop = 0, const double take = 0);
    bool adjust(const double stop = 0, const double take = 0);
    ...
};
```

正如我们上面看到的，根据环境的不同，修改只能通过订单号或仓位交易品种来完成。在前两个原型中考虑了这些选项。

此外，由于该结构体可能已经用于先前的请求，它可能已经填充了 *position* 和 *symbols* 字段。然后可以用最后一个原型调用该方法。

我们还没有展示这三个方法的实现，因为很明显它必须有一个包含发送请求的公共主体。这一部分被设计成一个私有的辅助方法 `_adjust`，带有一整套选项。此处其代码用一些不影响工作逻辑的缩写给出。

```
private:
    bool _adjust(const ulong pos, const string name,
        const double stop = 0, const double take = 0)
    {
        action = TRADE_ACTION_SLTP;
        position = pos;
        type = (ENUM_ORDER_TYPE)PositionGetInteger(POSITION_TYPE);
        if(!setSymbol(name)) return false;
        if(!setSLTP(stop, take)) return false;
        ZeroMemory(result);
        return OrderSend(this, result);
    }
```

我们按照上面的规则填充该结构的所有字段，调用前面说明的 `setSymbol` 和 `setSLTP` 方法，然后向服务器发送请求。结果为成功 (`true`) 或错误 (`false`) 状态。

每个重载的 `adjust` 方法分别为请求准备源参数。在有仓位订单号的情况下是这样做的。

```
public:
    bool adjust(const ulong pos, const double stop = 0, const double take = 0)
    {
        if(!PositionSelectByTicket(pos))
        {
            Print("No position: P=" + (string)pos);
            return false;
        }
        return _adjust(pos, PositionGetString(POSITION_SYMBOL), stop, take);
    }
```

此处，使用内置的 `PositionSelectByTicket` 函数，我们可以检查在终端的交易环境中是否存在仓位及其选择，这是后续读取其特性（在本例中为交易品种 (`PositionGetString(POSITION_SYMBOL)`)）所必需的。那么通用的变体就叫 `adjust`。

按交易品种名称修改仓位时（仅适用于净额结算账户），可以使用另一个选项 `adjust`。

```
bool adjust(const string name, const double stop = 0, const double take = 0)
{
    if(!PositionSelect(name))
    {
        Print("No position: " + s);
        return false;
    }

    return _adjust(PositionGetInteger(POSITION_TICKET), name, stop, take);
}
```

此处，仓位选择是使用内置的 `PositionSelect` 函数完成的，订单号是从其特性 (`PositionGetInteger(POSITION_TICKET)`) 中获得的。

所有这些功能将在关于 使用仓位 和 仓位特性的章节中详细讨论。

具有最少参数集的 *adjust* 方法版本，即仅具有 *stop* 和 *take* 水平，如下所示。

```
bool adjust(const double stop = 0, const double take = 0)
{
    if(position != 0)
    {
        if(!PositionSelectByTicket(position))
        {
            Print("No position with ticket P=" + (string)position);
            return false;
        }
        const string s = PositionGetString(POSITION_SYMBOL);
        if(symbol != NULL && symbol != s)
        {
            Print("Position symbol is adjusted from " + symbol + " to " + s);
        }
        symbol = s;
    }
    else if(AccountInfoInteger(ACCOUNT_MARGIN_MODE)
        != ACCOUNT_MARGIN_MODE_RETAIL_HEDGING
        && StringLen(symbol) > 0)
    {
        if(!PositionSelect(symbol))
        {
            Print("Can't select position for " + symbol);
            return false;
        }
        position = PositionGetInteger(POSITION_TICKET);
    }
    else
    {
        Print("Neither position ticket nor symbol was provided");
        return false;
    }
    return _adjust(position, symbol, stop, take);
}
```

此代码确保在各种模式下正确填充 *position* 和 *symbols* 字段，或者出错时提前退出并在日志中记录错误消息。最后，调用私有版本的 *_adjust*，其通过 *OrderSend* 发送请求。

与 *buy/sell* 方法类似，*adjust* 方法集会“异步”工作：在它们完成时，只有请求发送状态是已知的，但是无法确认水平修改是否成功。我们知道，对于证券交易所而言，*Take Profit* 水平可以作为限价订单转发。因此，在 *MqlTradeResultSync* 结构体中，我们应提供“同步”等待，直到更改生效。

作为 *MqlTradeResultSync::wait* 方法形成的一般等待机制已经准备就绪，并已用于等待开仓。*wait* 方法可接收一个指针作为第一个参数，该指针指向另一个具有预定义原型 *condition* 的方法，以便在循环中进行轮询，直到满足所需的条件或发生超时。在这种情况下，这种 *condition* 兼容的方法应对该仓位的止损水平进行应用检查。

我们添加这样一个名为 *adjusted* 的新方法。

```

struct MqlTradeResultSync: public MqlTradeResult
{
    ...
    bool adjusted(const ulong msc = 1000)
    {
        if(retcode != TRADE_RETCODE_DONE || retcode != TRADE_RETCODE_PLACED)
        {
            return false;
        }

        if(!wait(checkSLTP, msc))
        {
            Print("SL/TP modification timeout: P=" + (string)position);
            return false;
        }

        return true;
    }
}

```

当然，首先我们应检查 *retcode* 字段中的状态。如果有标准状态，我们将继续检查水平本身，将辅助方法 *checkSLTP* 传递给 *wait*。

```

struct MqlTradeResultSync: public MqlTradeResult
{
    ...
    static bool checkSLTP(MqlTradeResultSync &ref)
    {
        if(PositionSelectByTicket(ref.position))
        {
            return TU::Equal(PositionGetDouble(POSITION_SL), /*.*.*/)
                && TU::Equal(PositionGetDouble(POSITION_TP), /*.*.*/);
        }
        else
        {
            Print("PositionSelectByTicket failed: P=" + (string)ref.position);
        }
        return false;
    }
}

```

此代码可确保在终端的交易环境中使用 *PositionSelectByTicket* 通过订单号选择仓位，并读取仓位特性 *POSITION_SL* 和 *POSITION_TP*，这些特性应与请求中的特性进行比较。问题是，我们还没有获得对请求对象的访问权，必须以某种方式为标记为“?”的仓位传递一些值。

基本上，因为我们正在设计 *MqlTradeResultSync* 结构体，所以我们可以向其中添加 *sl* 和 *tp* 字段，并在发送请求之前用 *MqlTradeRequestSync* 中的值填充它们（内核并不“知道”我们添加的字段，在 *OrderSend* 调用期间不会对它们进行修改）。但是为了简单起见，我们将使用已经可用的值。*MqlTradeResultSync* 结构体中的 *bid* 和 *ask* 字段仅用于报告重新报价价格（*TRADE_RETCODE_REQUOTE* 状态），这与 *TRADE_ACTION_SLTP* 请求无关，因此我们可以将来自己完成的 *MqlTradeRequestSync* 的 *sl* 和 *tp* 存储在其中。

在 *MqlTradeRequestSync* 结构体的 *completed* 方法中这样做是合乎逻辑的，该方法使用预定义的超时启动对交易操作结果的阻塞等待。目前为止，其代码只有一个 *TRADE_ACTION_DEAL* 操作的分支。为了继续，让我们为 *TRADE_ACTION_SLTP* 添加一个分支。

```

struct MqlTradeRequestSync: public MqlTradeRequest
{
    ...
    bool completed()
    {
        if(action == TRADE_ACTION_DEAL)
        {
            const bool success = result.opened(timeout);
            if(success) position = result.position;
            return success;
        }
        else if(action == TRADE_ACTION_SLTP)
        {
            // pass the original request data for comparison with the position properties
            // by default they are not in the result structure
            result.position = position;
            result.bid = sl; // bid field is free in this result type, use under StopLoss
            result.ask = tp; // ask field is free in this type of result, we use it under TakeProfit
            return result.adjusted(timeout);
        }
        return false;
    }
}

```

正如你所看到的，在根据请求设置仓位订单号和价格水平之后，我们可调用上面讨论的 *adjusted* 方法来检查 *wait(checkSLTP)*。现在，我们可以返回到 *MqlTradeResultSync* 结构中的辅助方法 *checkSLTP*，并将其完善为最终形式。

```

struct MqlTradeResultSync: public MqlTradeResult
{
    ...
    static bool checkSLTP(MqlTradeResultSync &ref)
    {
        if(PositionSelectByTicket(ref.position))
        {
            return TU::Equal(PositionGetDouble(POSITION_SL), ref.bid) // sl from request
                && TU::Equal(PositionGetDouble(POSITION_TP), ref.ask); // tp from request
        }
        else
        {
            Print("PositionSelectByTicket failed: P=" + (string)ref.position);
        }
        return false;
    }
}

```

这样就完成了 *MqlTradeRequestSync* 和 *MqlTradeResultSync* 结构体用于 *Stop Loss* 和 *Take Profit* 修改操作的功能扩展。

记住这一点，让我们继续 EA 交易的示例 *MarketOrderSend.mq5*，我们在上一节中便已经开始。我们为其添加一个输入参数 *Distance2SLTP*，该参数允许你以点数为单位指定 *Stop Loss* 和 *Take Profit* 水平之间的距离。

```
input int Distance2SLTP = 0; // Distance to SL/TP in points (0 = no)
```

当该参数为零时，将不设置任何保护水平。

在工作代码中，收到开仓确认后，我们可以计算 SL 和 TP 变量中的水平值，并执行同步修改：
`request.adjust(SL, TP) && request.completed()`。

```
...
const ulong order = (wantToBuy ?
    request.buy(symbol, volume, Price) :
    request.sell(symbol, volume, Price));
if(order != 0)
{
    Print("OK Order: #=", order);
    if(request.completed()) // waiting for position opening
    {
        Print("OK Position: P=", request.result.position);
        if(Distance2SLTP != 0)
        {
            // position "selected" in the trading environment of the terminal inside
            // so it is not required to do this explicitly on the ticket
            // PositionSelectByTicket(request.result.position);

            // with the selected position, you can find out its properties, but we ne
            // to step back from it by a given number of points
            const double price = PositionGetDouble(POSITION_PRICE_OPEN);
            const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
            // we count the levels using the auxiliary class TradeDirection
            TU::TradeDirection dir((ENUM_ORDER_TYPE)Type);
            // SL is always "worse" and TP is always "better" of the price: the code
            const double SL = dir.negative(price, Distance2SLTP * point);
            const double TP = dir.positive(price, Distance2SLTP * point);
            if(request.adjust(SL, TP) && request.completed())
            {
                Print("OK Adjust");
            }
        }
    }
}
Print(TU::StringOf(request));
Print(TU::StringOf(request.result));
}
```

在成功的买入或卖出操作后第一次调用 `completed` 时，仓位订单号保存在请求结构体的 `position` 字段中。因此，要修改止损点，只有价格水平就足够了，并且仓位的交易品种和订单号已经存在于 `request` 中。

我们来尝试使用默认设置的 EA 交易执行买入操作，但 `Distance2SLTP` 设置为 500 点。

```

OK Order: #=1273913958
Waiting for position for deal D=1256506526
OK Position: P=1273913958
OK Adjust
TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10889, »
» SL=1.10389, TP=1.11389, P=1273913958
DONE, Bid=1.10389, Ask=1.11389, Request executed, Req=26

```

最后两行对应于 *request* 和 *request.result* 结构体 (在函数结束时启动) 内容日志的调试输出。在这些行中, 有趣的是这些字段存储了来自两个查询的共生值: 首先是进行了开仓, 然后对其进行了修改。具体来说, 请求中具有交易量 (0.01) 和价格 (1.10889) 的字段保留在 *TRADE_ACTION_DEAL* 之后, 但是不会阻止 *TRADE_ACTION_SLTP* 的执行。理论上, 通过重置两个请求之间的结构体很容易解决这个问题, 但是, 我们更喜欢让它们保持原样, 因为在填充的字段中也有有用的字段: *position* 字段接收了我们请求修改所需的订单号。如果我们重置这个结构体, 那么就需要引入一个变量作为该订单号的中间存储。

当然, 在一般情况下, 坚持严格的数据初始化策略是可取的, 但是知道如何在特定的场景中使用它们 (比如两个或多个预定义类型的相关请求) 可以让你优化代码。

此外, 在包含结果的结构体中, 我们能在 *Bid* 和 *Ask* 价格字段中看到请求水平 *sl* 和 *tp*, 这一点也不奇怪: 它们是由 *MqlTradeRequestSync::completed* 方法写入的, 目的是与实际的仓位变化进行比较。当执行请求时, 系统内核只在 *result* 结构体中填充 *retcode* (已完成)、*comment* (“请求已执行”) 和 *request_id* (26)。

接下来, 我们将考虑另一个实施 [跟踪止损](#) 的水平修改示例。

6.4.16 跟踪止损

使用改变保护价格水平能力的最常见的任务之一是随着有利趋势的继续, 在更好的价格上按顺序转移 *Stop Loss*。这就是跟踪止损。我们使用前几节中的新结构体 *MqlTradeRequestSync* 和 *MqlTradeResultSync* 来实现跟踪止损。

为了能够将该机制连接到任何 EA 交易, 我们将其声明为 *Trailing Stop* 类 (参见文件 *TrailingStop.mqh*)。我们将存储受控仓位的编号、其交易品种和价格点的大小、止损水平与当前价格的所需距离以及该类单个变量中的水平变化步长。

```

#include <MQL5Book/MqlTradeSync.mqh>

class TrailingStop
{
    const ulong ticket; // ticket of controlled position
    const string symbol; // position symbol
    const double point; // symbol price pip size
    const uint distance; // distance to the stop in points
    const uint step; // movement step (sensitivity) in points
    ...
}

```

只有基础类提供的标准仓位跟踪算法才需要距离。派生类能够根据其他原则移动保护水平, 如移动平均线、通道、SAR 指标等。在熟悉了基础类之后, 我们将给出一个带有移动平均线的派生类示例。

我们为当前止损价格水平创建 *level* 变量。在 *ok* 变量中, 我们将保持该仓位的当前状态: *true* 如果该仓位仍然存在, 则为 *true*; 如果出现错误且该仓位已平仓, 则为 *false*。

```
protected:
    double level;
    bool ok;
    virtual double detectLevel()
    {
        return DBL_MAX;
    }
```

虚方法 *detectLevel* 用于在子类中重写，其中止损价格应根据任意算法来计算。在这个实现中，会返回一个特殊值 DBL_MAX，表示根据标准算法所做的工作（见下文）。

在构造函数中，用相应参数的值填充所有字段。函数 *PositionSelectByTicket* 用于检查具有给定订单号的仓位是否存在，并在程序环境中分配它，以便 *PositionGetString* 的后续调用可返回其具有交易品种名称的字符串特性。

```
public:
    TrailingStop(const ulong t, const uint d, const uint s = 1) :
        ticket(t), distance(d), step(s),
        symbol(PositionSelectByTicket(t) ? PositionGetString(POSITION_SYMBOL) : NULL),
        point(SymbolInfoDouble(symbol, SYMBOL_POINT))
    {
        if(symbol == NULL)
        {
            Print("Position not found: " + (string)t);
            ok = false;
        }
        else
        {
            ok = true;
        }
    }

    bool isOK() const
    {
        return ok;
    }
```

现在我们考虑 *trail* 类的主要公共方法。MQL 程序需要在每个分时报价上或通过计时器调用该方法跟踪仓位。当仓位存在时，该方法返回 *true*。

```

virtual bool trail()
{
    if(!PositionSelectByTicket(ticket))
    {
        ok = false;
        return false; // position closed
    }

    // find out prices for calculations: current quote and stop level
    const double current = PositionGetDouble(POSITION_PRICE_CURRENT);
    const double sl = PositionGetDouble(POSITION_SL);
    ...
}

```

此处和下面，我们使用仓位特性来读取函数。仓位特性将在 单独的章节中详细讨论。特别是，我们需要找出交易方向（买入和卖出）才能知道应在哪个方向设置止损水平。

```

// POSITION_TYPE_BUY = 0 (false)
// POSITION_TYPE_SELL = 1 (true)
const bool sell = (bool)PositionGetInteger(POSITION_TYPE);
TU::TradeDirection dir(sell);
...

```

对于计算和检查，我们应使用辅助类 *TU::TradeDirection* 及其对象 *dir*。例如，其 *negative* 方法允许在亏损方向上计算位于当前价格指定距离处的价格，而无需考虑操作的类型。这样简化了代码，否则就必须为买入和卖出进行“镜像”计算。

```

level = detectLevel();
// we can't trail without a level: removing the stop level must be done by the
if(level == 0) return true;
// if there is a default value, make a standard offset from the current price
if(level == DBL_MAX) level = dir.negative(current, point * distance);
level = TU::NormalizePrice(level, symbol);

if(!dir.better(current, level))
{
    return true; // you can't set a stop level on the profitable side<
}
...

```

TU::TradeDirection 类的 *better* 方法用于检查收到的止损水平是否位于价格的右侧。如果没有这个方法，我们将需要再次写入两次检查（针对买入和卖出）。

我们可能会得到一个不正确的止损水平值，因为 *detectLevel* 方法可以在派生类中被重写。使用标准计算，这个问题就可以消除，因为水平是由 *dir* 对象计算的。

最后，在计算水平时，需要将其应用到仓位上。如果仓位还没有止损，任何有效水平均可以。如果已经设置了止损，那么新的值应比以前的值更好，并且相差超过指定的步长。

```

    if(sl == 0)
    {
        PrintFormat("Initial SL: %f", level);
        move(level);
    }
    else
    {
        if(dir.better(level, sl) && fabs(level - sl) >= point * step)
        {
            PrintFormat("SL: %f -> %f", sl, level);
            move(level);
        }
    }

    return true; // success
}

```

发送仓位修改请求是在 *move* 方法中实现的，该方法使用 *MqlTradeRequestSync* 结构体的常见 *adjust* 方法（参见 修改止损和/或止盈水平一节）。

```

bool move(const double sl)
{
    MqlTradeRequestSync request;
    request.position = ticket;
    if(request.adjust(sl, 0) && request.completed())
    {
        Print("OK Trailing: ", TU::StringOf(sl));
        return true;
    }
    return false;
}
};

```

现在一切准备就绪，可以向测试 EA 交易 *TrailingStop.mq5* 添加跟踪了。在输入参数中，你可以指定交易方向、到止损点的距离以及步长。默认情况下，*TrailingDistance* 参数等于 0，这意味着自动计算每日报价范围，并使用其一半作为距离。


```

#include <MQL5Book/MqlTradeSync.mqh>
#include <MQL5Book/TrailingStop.mqh>

enum ENUM_ORDER_TYPE_MARKET
{
    MARKET_BUY = ORDER_TYPE_BUY,    // ORDER_TYPE_BUY
    MARKET_SELL = ORDER_TYPE_SELL    // ORDER_TYPE_SELL
};

input int TrailingDistance = 0;    // Distance to Stop Loss in points (0 = autodetect)
input int TrailingStep = 10;      // Trailing Step in points
input ENUM_ORDER_TYPE_MARKET Type;
input string Comment;
input ulong Deviation;
input ulong Magic = 1234567890;

```

启动时，EA 交易将查找当前交易品种上是否存在具有指定 *Magic* 号的仓位，如果不存在，将创建该仓位。

跟踪将由 *TrailingStop* 类的对象执行（包装在智能指针 *AutoPtr* 中）。由于后者，当需要新的跟踪对象来替换该对象以获得正在创建的新仓位时，我们不需要手动删除旧对象。当新对象被赋予给智能指针时，旧对象会被自动删除。注意，取消对智能指针的引用，即访问存储在其中的工作对象，是使用重载的 `[]` 运算符完成的。

```

#include <MQL5Book/AutoPtr.mqh>

```

```

AutoPtr<TrailingStop> tr;

```

在 *OnTick* 处理程序中，我们会检查是否有对象。如果有，检查仓位是否存在（从 *trail* 方法返回其属性）。程序启动后，如果对象不存在，则指针为空。在这种情况下，你应创建一个新的仓位，或者找到一个已经打开的仓位，并为其创建一个 *Trailing Stop* 对象。这是由 *Setup* 函数完成的。在 *OnTick* 的后续调用中，对象开始并继续跟踪，防止程序在仓位“活动”时进入 *if* 块。

```

void OnTick()
{
    if(tr[] == NULL || !tr[].trail())
    {
        // if there is no trailing yet, create or find a suitable position
        Setup();
    }
}

```

这就是 *Setup* 函数。

```

void Setup()
{
    int distance = 0;
    const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);

    if(trailing distance == 0) // auto-detect the daily range of prices
    {
        distance = (int)((iHigh(_Symbol, PERIOD_D1, 1) - iLow(_Symbol, PERIOD_D1, 1))
            / point / 2);
        Print("Autodetected daily distance (points): ", distance);
    }
    else
    {
        distance = TrailingDistance;
    }

    // process only the position of the current symbol and our Magic
    if(GetMyPosition(_Symbol, Magic))
    {
        const ulong ticket = PositionGetInteger(POSITION_TICKET);
        Print("The next position found: ", ticket);
        tr = new TrailingStop(ticket, distance, TrailingStep);
    }
    else // there is no our position
    {
        Print("No positions found, lets open it...");
        const ulong ticket = OpenPosition();
        if(ticket)
        {
            tr = new TrailingStop(ticket, distance, TrailingStep);
        }
    }

    if(tr[] != NULL)
    {
        // Execute trailing for the first time immediately after creating or finding a
        tr[].trail();
    }
}

```

在 *GetMyPosition* 函数中实现对合适的开仓仓位的搜索，并且通过 *OpenPosition* 函数完成对新仓位的开仓。二者都在下文中介绍。无论如何，我们可以得到一张仓位订单号，并为其创建一个跟踪对象。

```

bool GetMyPosition(const string s, const ulong m)
{
    for(int i = 0; i < PositionsTotal(); ++i)
    {
        if(PositionGetSymbol(i) == s && PositionGetInteger(POSITION_MAGIC) == m)
        {
            return true;
        }
    }
    return false;
}

```

该算法的目的和一般含义从内置函数的名称中就能看得很清楚。在所有未结仓位 (*PositionsTotal*) 的循环中, 我们可使用 *PositionGetSymbol* 依次选择每个仓位并获取其交易品种。如果该交易品种与请求的交易品种匹配, 我们可读取仓位特性 *POSITION_MAGIC* 并与传递的 "magic" 进行比较。与处理仓位相关的所有函数将在 [单独的章节](#) 中讨论。

一旦找到第一个匹配仓位, 该函数将返回 *true*。同时, 在终端的交易环境中, 该仓位将保持选中状态, 这使得代码的其余部分可以在必要时读取其相关特性。

我们已经知道了开仓的算法。

```

ulong OpenPosition()
{
    MqlTradeRequestSync request;

    // default values
    const bool wantToBuy = Type == MARKET_BUY;
    const double volume = SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN);
    // optional fields are filled directly in the structure
    request.magic = Magic;
    request.deviation = Deviation;
    request.comment = Comment;
    ResetLastError();
    // execute the selected trade operation and wait for its confirmation
    if((bool)(wantToBuy ? request.buy(volume) : request.sell(volume))
        && request.completed())
    {
        Print("OK Order/Deal/Position");
    }

    return request.position; // non-zero value - sign of success
}

```

为了清楚起见, 我们看看这个程序是如何在测试程序中以可视模式工作的。

编译之后, 我们打开终端中的策略测试程序面板, 在 **Review** 选项卡上, 选择第一个选项: *Single test*。

在 **Settings** 选项卡中, 选择以下选项:

- 在下拉菜单 *Expert Advisor* 中: `MQL5Book\p6\TrailingStop`
- *Symbol*: EURUSD

- *Timeframe*: H1
- *Interval*: 去年、上个月或自定义
- *Forward*: 否
- *Delays*: 禁用
- *Modeling*: 基于真实或生成的分时报价
- *Optimization*: 禁用
- *Visual mode*: 启用

一旦按下 **Start**, 将在一个单独的测试程序窗口中看到类似这样的内容:



测试程序中的标准跟踪止损

日志将显示如下条目:

```

2022.01.10 00:02:00 Autodetected daily distance (points): 373
2022.01.10 00:02:00 No positions found, let's open it...
2022.01.10 00:02:00 instant buy 0.01 EURUSD at 1.13612 (1.13550 / 1.13612 / 1.13550)
2022.01.10 00:02:00 deal #2 buy 0.01 EURUSD at 1.13612 done (based on order #2)
2022.01.10 00:02:00 deal performed [#2 buy 0.01 EURUSD at 1.13612]
2022.01.10 00:02:00 order performed buy 0.01 at 1.13612 [#2 buy 0.01 EURUSD at 1.13612]
2022.01.10 00:02:00 Waiting for position for deal D=2
2022.01.10 00:02:00 OK Order/Deal/Position
2022.01.10 00:02:00 Initial SL: 1.131770
2022.01.10 00:02:00 position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13177]
2022.01.10 00:02:00 OK Trailing: 1.13177
2022.01.10 00:06:13 SL: 1.131770 -> 1.131880
2022.01.10 00:06:13 position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13188]
2022.01.10 00:06:13 OK Trailing: 1.13188
2022.01.10 00:09:17 SL: 1.131880 -> 1.131990
2022.01.10 00:09:17 position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13199]
2022.01.10 00:09:17 OK Trailing: 1.13199
2022.01.10 00:09:26 SL: 1.131990 -> 1.132110
2022.01.10 00:09:26 position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13211]
2022.01.10 00:09:26 OK Trailing: 1.13211
2022.01.10 00:09:35 SL: 1.132110 -> 1.132240
2022.01.10 00:09:35 position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13224]
2022.01.10 00:09:35 OK Trailing: 1.13224
2022.01.10 10:06:38 stop loss triggered #2 buy 0.01 EURUSD 1.13612 sl: 1.13224 [#3 se
2022.01.10 10:06:38 deal #3 sell 0.01 EURUSD at 1.13221 done (based on order #3)
2022.01.10 10:06:38 deal performed [#3 sell 0.01 EURUSD at 1.13221]
2022.01.10 10:06:38 order performed sell 0.01 at 1.13221 [#3 sell 0.01 EURUSD at 1.13
2022.01.10 10:06:38 Autodetected daily distance (points): 373
2022.01.10 10:06:38 No positions found, let's open it...

```

看看该算法如何在有利的价格变动中提升 SL 水平，直到止损平仓。清算仓位后，该程序会立即打开一个新的仓位。

为了检查使用非标准跟踪机制的可能性，我们实现了一个移动平均方法的示例。为此，我们回到文件 *TrailingStop.mqh* 并说明派生类 *TrailingStopByMA*。

```

class TrailingStopByMA: public TrailingStop
{
    int handle;

public:
    TrailingStopByMA(const ulong t, const int period,
        const int offset = 1,
        const ENUM_MA_METHOD method = MODE_SMA,
        const ENUM_APPLIED_PRICE type = PRICE_CLOSE): TrailingStop(t, 0, 1)
    {
        handle = iMA(_Symbol, PERIOD_CURRENT, period, offset, method, type);
    }

    virtual double detectLevel() override
    {
        double array[1];
        ResetLastError();
        if(CopyBuffer(handle, 0, 0, 1, array) != 1)
        {
            Print("CopyBuffer error: ", _LastError);
            return 0;
        }
        return array[0];
    }
};

```

该类可在构造函数中创建 *iMA* 指标实例：周期、平均方法和价格类型通过参数传递。

在被改写的 *detectLevel* 方法中，我们从指标缓冲区读取值，默认情况下，这是以 1 柱线的偏移量完成的，即，该柱线是关闭的，当分时报价到达时，其读数不会改变。如果愿意，可以从零柱线取值，但是这样的信号对于所有价格类型都是不稳定的，除了 PRICE_OPEN 之外。

为了在同一个测试 EA 交易 *TrailingStop.mq5* 中使用一个新类，我们添加另一个带有移动周期的输入参数 *MATrailingPeriod*（保持该指标的其他参数不变）。

```
input int MATrailingPeriod = 0; // Period for Trailing by MA (0 = disabled)
```

此参数中的 0 值可禁用跟踪移动平均线。如果启用，*TrailingDistance* 参数中的距离设置将被忽略。

根据这个参数，我们将创建一个标准的跟踪对象 *TrailingStop* 或一个来自 *iMA -TrailingStopByMA* 的派生对象。

```

...
tr = MATrailingPeriod > 0 ?
    new TrailingStopByMA(ticket, MATrailingPeriod) :
    new TrailingStop(ticket, distance, TrailingStep);
...

```

我们看看更新后的程序在测试程序中的表现。在 EA 交易设置中，为 MA 设置一个非零周期，例如 10。



测试程序中移动平均线上的跟踪止损

请注意，在平均线接近该价格的时刻，有一个频繁止损触发平仓的效应。当平均价格高于报价时，并没有设置保护水平，因为这对于买入是不正确的。这是因为我们的 EA 交易没有任何策略，不管市场情况如何，总是建立相同类型的未结仓位。对于卖出而言，当平均价格低于该价格时，也会偶尔出现相同的矛盾情况，这意味着市场正在增长，而机器人“顽固地”进入空头仓位。

在工作策略中，一般来说，仓位方向的选择要考虑到市场的波动，当移动平均线位于当前价格的右侧时，方可允许设置止损。

6.4.17 平仓：全部和部分

从技术上讲，平仓可以视为与开仓相反的交易操作。例如，要退出买入，需要进行卖出操作（`type` 字段中的 `ORDER_TYPE_SELL`），要退出卖出，需要进行卖出运算 `type` 字段中的 `ORDER_TYPE_BUY`）。

`MqlTradeTransaction` 结构体的 `action` 字段中的交易操作类型保持不变：`TRADE_ACTION_DEAL`。

在对冲账户上，必须在 `position` 字段中使用订单号指定要平仓的仓位。对于净额结算账户，只能在 `symbol` 字段中指定交易品种名称，因为它们只能有一个交易品种仓位。不过，也可以在此处用订单号平仓。

为了统一代码，无论账户类型如何，都应填写 `position` 和 `symbol` 字段。

此外，请确保在 `volume` 字段中设置交易量。如果等于持仓量，则完全平仓。但是，通过指定一个较低的值，可以部分平仓。

在下表中，所有必需的结构体字段都标有星号，可选字段标有加号。

字 段	净 额 结 算	对 冲
action	*	*
symbol	*	+
position	+	*
type	*	*
type_filling	*	*
volume	*	*
price	*'	*'
deviation	±	±
magic	+	+
comment	+	+

标记的 *price* 字段带有一个带勾号的星号，因为它仅对于具有 *Request* 和 *Instant* 执行模式的交易品种是必需的，而对于 *Exchange* 和 *Market* 执行，不会考虑结构体中的价格。

出于类似的原因，*deviation* 字段标有 '±'。该字段仅对 *Instant* 和 *Request* 模式有效。

为了简化平仓的编程实现，我们回到 *MqlTradeSync.mqh* 文件中的扩展结构体 *MqlTradeRequestSync*。按订单号的平仓方法有以下代码。

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
    double partial; // volume after partial closing
    ...
    bool close(const ulong ticket, const double lot = 0)
    {
        if(!PositionSelectByTicket(ticket)) return false;

        position = ticket;
        symbol = PositionGetString(POSITION_SYMBOL);
        type = (ENUM_ORDER_TYPE)(PositionGetInteger(POSITION_TYPE) ^ 1);
        price = 0;
        ...
    }
}
```

此处，我们首先检查仓位是否存在（通过调用 *PositionSelectByTicket* 函数）。此外，该调用可在终端的交易环境中选择仓位，这允许你使用 *后续函数* 读取其特性。具体来说，我们可从 *POSITION_SYMBOL* 特性中找出仓位的交易品种，并将其类型从 *POSITION_TYPE* “反转”到相反的类型，以获得所需的订单类型。

ENUM_POSITION_TYPE 枚举中的仓位类型为 POSITION_TYPE_BUY (值 0) 和 POSITION_TYPE_SELL (值 1)。在订单类型 ENUM_ORDER_TYPE 的枚举中, 市场操作占据完全相同的值: ORDER_TYPE_BUY 和 ORDER_TYPE_SELL。这就是为什么我们可以将第一个枚举带到第二个枚举的原因, 为了获得相反的交易方向, 只需使用异或运算 (^) 来切换 0 位: 我们可从 0 得到 1, 从 1 得到 0。

将 *price* 字段置零意味着在发送请求之前会自动选择正确的当前价格 (*Ask* 或 *Bid*): 这是稍后在辅助方法 *setVolumePrices* 中完成的, 该方法与来自 *market* 方法的算法一起调用。

_market 方法调用发生在下面几行。*_market* 方法可为整个或部分交易量生成市场订单, 并考虑结构体中所有已完成的字段。

```
const double total = lot == 0 ? PositionGetDouble(POSITION_VOLUME) : lot;
partial = PositionGetDouble(POSITION_VOLUME) - total;
return _market(symbol, total);
}
```

与当前源代码相比, 这个片段稍微简化了一些。完整的代码包含处理一种罕见但可能发生的情况, 即仓位交易量超过每个交易品种一个订单中允许的最大交易量 (SYMBOL_VOLUME_MAX 特性)。在这种情况下, 必须通过几个订单分批平仓。

还要注意的, 由于可以部分平仓, 我们必须向 *partial* 结构体添加一个字段, 在该字段中放置运算后的计划量余额。当然, 对于完全平仓则为 0。必须用此信息来进一步验证操作的完成。

对于净额结算账户, 有一个版本的 *close* 方法, 通过交易品种名称识别仓位。该方法通过交易品种选择一个仓位, 得到其订单号, 然后引用以前版本的 *close*。

```
bool close(const string name, const double lot = 0)
{
    if(!PositionSelect(name)) return false;
    return close(PositionGetInteger(POSITION_TICKET), lot);
}
```

在 *MqlTradeRequestSync* 结构体中, 我们有 *completed* 方法, 该方法在必要时为运算的完成提供同步等待。现在我们需要在 *action* 等于 TRADE_ACTION_DEAL 的分支中将其补充到平仓。我们将通过 *position* 字段的零值来区分开仓和平仓: 开仓时没有订单号, 平仓时有订单号。

```

bool completed()
{
    if(action == TRADE_ACTION_DEAL)
    {
        if(position == 0)
        {
            const bool success = result.opened(timeout);
            if(success) position = result.position;
            return success;
        }
        else
        {
            result.position = position;
            result.partial = partial;
            return result.closed(timeout);
        }
    }
}

```

为了检查实际平仓情况，我们可将 *closed* 方法添加到 *MqlTradeResultSync* 结构体中。在调用该方法之前，我们可在 *result.position* 字段中写入仓位订单号，以便产生的结构体可以跟踪相应订单号从终端的交易环境中消失的时刻，或者在部分平仓的情况下交易量等于 *result.partial* 的时刻。

此处是 *closed* 方法。该方法基于一个众所周知的原则：首先检查服务器返回代码是否成功，然后使用 *wait* 方法等待某个条件满足。

```

struct MqlTradeResultSync: public MqlTradeResult
{
    ...
    bool closed(const ulong msc = 1000)
    {
        if(retcode != TRADE_RETCODE_DONE)
        {
            return false;
        }
        if(!wait(positionRemoved, msc))
        {
            Print("Position removal timeout: P=" + (string)position);
        }

        return true;
    }
}

```

在这种情况下，为了检查仓位消失的条件，我们必须实现一个新的函数 *positionRemoved*。

```

static bool positionRemoved(MqlTradeResultSync &ref)
{
    if(ref.partial)
    {
        return PositionSelectByTicket(ref.position)
            && TU::Equal(PositionGetDouble(POSITION_VOLUME), ref.partial);
    }
    return !PositionSelectByTicket(ref.position);
}

```

我们可使用 EA 交易 *TradeClose.mq5* 来测试平仓操作，其实现了一个简单的交易策略：如果有两个连续的柱线同向，则进入市场，一旦下一个柱线以与前一个趋势相反的方向收盘，我们就退出市场。连续趋势期间的重复信号将被忽略，即市场中最多有一个仓位（最小手数）或没有。

EA 交易没有任何可调参数：只有 *(Deviation)* 和一个唯一的数字 *(Magic)*。隐含的参数是图表的时间范围和工作交易品种。

为了跟踪已经开仓的仓位是否存在，我们使用前面示例 *TradeTrailing.mq5* 中的 *GetMyPosition* 函数：该函数通过交易品种和 EA 交易号在仓位中搜索，如果找到合适的仓位，则返回逻辑 *true*。

我们还采用几乎没有变化的函数 *OpenPosition*：该函数可根据单个参数中传递的市场订单类型开仓。此处，该参数将来自趋势检测算法，在更早的时候（在 *TrailingStop.mq5* 中），订单类型是由用户通过输入变量设置的。

实现平仓的一个新函数是 *ClosePosition*。因为头文件 *MqlTradeSync.mqh* 接管了整个例程，所以我们只需要为提交的仓位订单号调用 *request.close(ticket)* 方法，然后等待 *request.completed()* 完成删除。

从理论上讲，如果 EA 交易在每一个分时报价处分析情况，后者是可以避免的。在这种情况下，删除仓位的潜在问题将在下一个分时报价处迅速暴露出来，并且 EA 交易可以再次尝试删除仓位。但是，该 EA 交易采用基于柱线的交易逻辑，因此分析每一个分时报价是没有意义的。接下来，我们实现了一个特殊的机制来完成逐柱线工作，在这方面，我们可同步地控制移除，否则，仓位将在整个柱线上保持“悬置”。

```

ulong LastErrorCode = 0;

ulong ClosePosition(const ulong ticket)
{
    MqlTradeRequestSync request; // empty structure

    // optional fields are filled directly in the structure
    request.magic = Magic;
    request.deviation = Deviation;

    ResetLastError();
    // perform close and wait for confirmation
    if(request.close(ticket) && request.completed())
    {
        Print("OK Close Order/Deal/Position");
    }
    else // print diagnostics in case of problems
    {
        Print(TU::StringOf(request));
        Print(TU::StringOf(request.result));
        LastErrorCode = request.result.retcode;
        return 0; // error, code to parse in LastErrorCode
    }

    return request.position; // non-zero value - success
}

```

我们可以强制 *ClosePosition* 函数在成功删除仓位的情况下返回 0，否则返回一个错误代码。这种看似有效的方法会使 *OpenPosition* 和 *ClosePosition* 两个函数的行为有所不同：在调用代码中，需要将这些函数的调用嵌套在意义相反的逻辑表达式中，而这会引入混乱。此外，我们在任何情况下都需要全局变量 *LastErrorCode*，以便在 *OpenPosition* 函数中添加有关错误的信息。同样，此外，*if(condition)* 检查比 *if(!condition)* 更容易被解释为成功。

根据上述策略生成交易信号的函数称为 *GetTradeDirection*。

```

ENUM_ORDER_TYPE GetTradeDirection()
{
    if(iClose(_Symbol, _Period, 1) > iClose(_Symbol, _Period, 2)
        && iClose(_Symbol, _Period, 2) > iClose(_Symbol, _Period, 3))
    {
        return ORDER_TYPE_BUY; // open a long position
    }

    if(iClose(_Symbol, _Period, 1) < iClose(_Symbol, _Period, 2)
        && iClose(_Symbol, _Period, 2) < iClose(_Symbol, _Period, 3))
    {
        return ORDER_TYPE_SELL; // open a short position
    }

    return (ENUM_ORDER_TYPE)-1; // close
}

```

该函数可返回一个 `ENUM_ORDER_TYPE` 类型的值，该值具有两个分别触发买入和卖出的标准元素 `ORDER_TYPE_BUY` 和 `ORDER_TYPE_SELL`。特殊值 `-1` (不在枚举中)可用作平仓信号。

为了激活基于交易算法的 EA 交易，我们使用了 *OnTick* 处理程序。我们都还记得，其他选项适用于其他策略，例如，计时器可用于新闻交易，市场深度事件可用于交易量。

首先，我们以简化的形式分析该函数，不处理潜在的错误。在最开始，有一个程序块可确保仅当一个新的柱线平仓时，才会触发进一步的算法。

```

void OnTick()
{
    static datetime lastBar = 0;
    if(iTime(_Symbol, _Period, 0) == lastBar) return;
    lastBar = iTime(_Symbol, _Period, 0);
    ...
}

```

接下来，我们从 *GetTradeDirection* 函数中获取当前信号。

```
const ENUM_ORDER_TYPE type = GetTradeDirection();
```

如果有一个仓位，我们可检查是否已经收到其平仓信号，并根据需要调用 *ClosePosition*。如果还没有仓位，但有进场信号，我们称之为 *OpenPosition*。

```

if(GetMyPosition(_Symbol, Magic))
{
    if(type != ORDER_TYPE_BUY && type != ORDER_TYPE_SELL)
    {
        ClosePosition(PositionGetInteger(POSITION_TICKET));
    }
}
else if(type == ORDER_TYPE_BUY || type == ORDER_TYPE_SELL)
{
    OpenPosition(type);
}
}

```

要分析错误，需要将 *OpenPosition* 和 *ClosePosition* 调用包含在条件语句中，并采取一些措施来恢复程序的工作状态。在最简单的情况下，只需要在下一个分时报价处重复该请求，但是最好只重复有限的次数。因此，我们将创建带有计数器和错误限制的静态变量。

```

void OnTick()
{
    static int errors = 0;
    static const int maxtrials = 10; // no more than 10 attempts per bar

    // expect a new bar to appear if there were no errors
    static datetime lastBar = 0;
    if(iTime(_Symbol, _Period, 0) == lastBar && errors == 0) return;
    lastBar = iTime(_Symbol, _Period, 0);
    ...
}

```

如果出现错误，则逐柱线机制会被暂时禁用，因为需要尽快处理这些错误。

在 *ClosePosition* 和 *OpenPosition* 的条件语句中，统计错误的数量。

```

const ENUM_ORDER_TYPE type = GetTradeDirection();

if(GetMyPosition(_Symbol, Magic))
{
    if(type != ORDER_TYPE_BUY && type != ORDER_TYPE_SELL)
    {
        if(!ClosePosition(PositionGetInteger(POSITION_TICKET)))
        {
            ++errors;
        }
        else
        {
            errors = 0;
        }
    }
}
else if(type == ORDER_TYPE_BUY || type == ORDER_TYPE_SELL)
{
    if(!OpenPosition(type))
    {
        ++errors;
    }
    else
    {
        errors = 0;
    }
}
// too many errors per bar
if(errors >= maxtrials) errors = 0;
// error serious enough to pause
if(IS_TANGIBLE(LastErrorCode)) errors = 0;
}

```

将 **errors** 变量设置为 0 将再次打开逐柱线机制，并停止重复请求的尝试，直到下一个柱线。

宏 **IS_TANGIBLE** 在 *TradeRetcode.mqh* 中定义为：

```
#define IS_TANGIBLE(T) ((T) >= TRADE_RETCODE_ERROR)
```

值较小的错误代码是可运行的，即在某种意义上是正常的。值较大的错误代码需要分析和不同的操作，具体取决于问题的原因：不正确的请求参数、交易环境中的永久或临时禁令、缺乏资金等等。我们将在 [挂单修改](#) 章节介绍了改进错误分类器。

我们从 2022 年初开始，在 XAUUSD,H1 的测试程序中运行 EA 交易，模拟真实的分时报价。下一幅图展示了一个包含交易的图表片段以及余额曲线。



对 XAUUSD,H1 运行 TradeClose 的测试结果

基于报告和日志，我们可以看到，我们简单的交易逻辑和开仓与平仓两个运算的结合在正常工作。

除了简单的平仓，该平台还支持对冲账户上 [两个相反仓位的相互平仓](#)。

6.4.18 全部和部分平仓（对冲）

在对冲账户上，允许同时开立几个仓位，在大多数情况下，这些仓位可以是相反的方向。在某些司法管辖区，对冲账户是受限制的：一次只能持有一个方向的仓位。在这种情况下，当试图执行相反的交易操作时，你将收到 `TRADE_RETCODE_HEDGE_PROHIBITED` 错误代码。此外，这种限制通常与 `ACCOUNT_FIFO_CLOSE` 账户特性设置为 `true` 相关。

当两个相反的仓位同时开仓时，平台使用 `TRADE_ACTION_CLOSE_BY` 运算支持其同时相互平仓的机制。要执行此操作，除了 `action` 字段之外，你还应在 `MqlTradeTransaction` 结构体中填充另外两个字段：`position` 并且 `position_by` 必须包含要平仓的仓位订单号。

此功能的可用性取决于金融工具的 `SYMBOL_ORDER_MODE` 特性：`SYMBOL_ORDER_CLOSEBY` (64) 必须出现在允许标志位掩码中。

这种操作不仅简化了平仓（一次操作而不是两次），而且节省了一个点差。

我们知道，任何新仓位开始交易时的损失都等于该点差。例如，当买入一种金融工具时，交易以 `Ask` 价格成交，但对于退出交易，即卖出，实际价格是 `Bid` 价格。对于空头仓位，情况正好相反：在以 `Bid` 价格进场后，为了潜在的离场操作，我们会立即开始跟踪 `Ask` 价格。

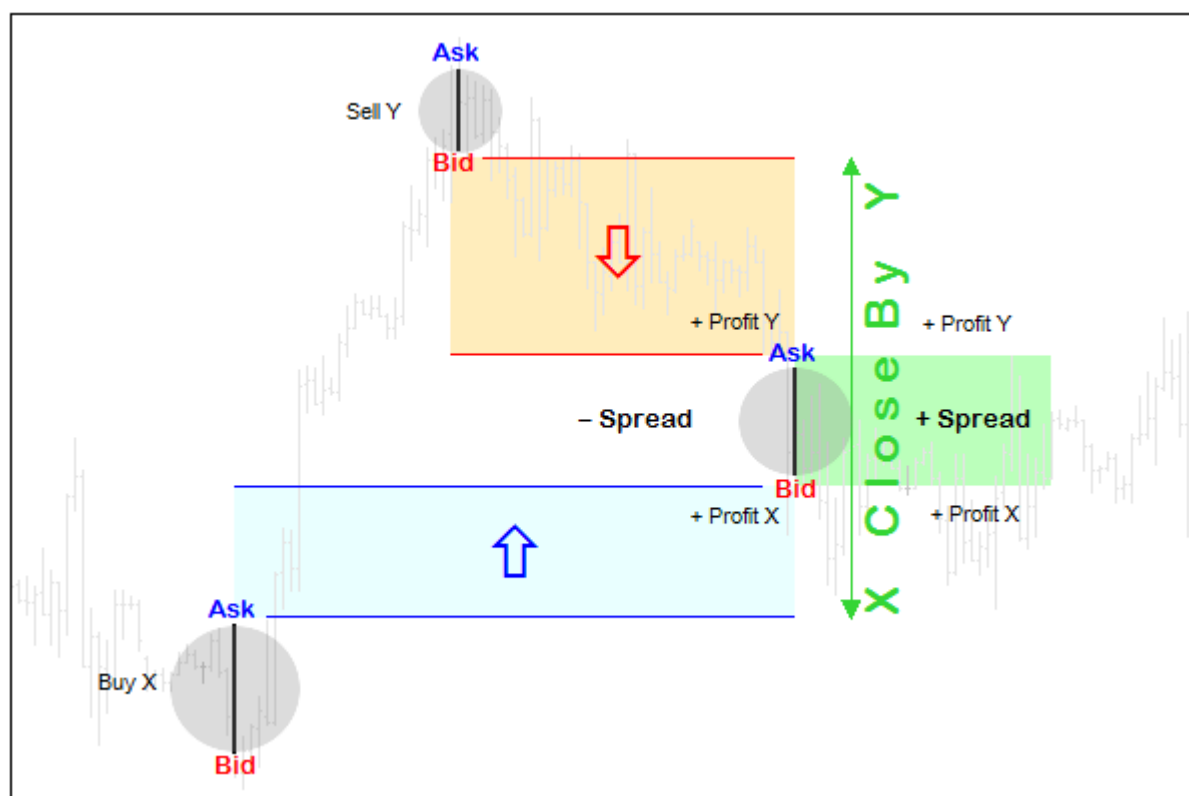
如果你定期在同一时间平仓，它们的退出价格将与当前点差相差一段距离。但是，如果你使用 `TRADE_ACTION_CLOSE_BY` 操作，那么这两个仓位均将被平仓，而不考虑当前价格。仓位抵消的

价格等于 `position_by` 仓位的开盘价 (在请求结构体中)。该开盘价在由 `TRADE_ACTION_CLOSE_BY` 请求生成的 `ORDER_TYPE_CLOSE_BY` 订单中指定。

遗憾的是,在交易和仓位背景下的报告中,相反仓位/交易的收盘价和开盘价会以镜像方向成对显示,这给人以双重止盈或止亏的印象。事实上,该运算的财务结果 (为该手数调整的价格之间的差异)仅针对第一个仓位退出交易 (请求结构体中的 `position` 字段)进行记录。第二次退出交易的结果总是为 0,而与价差无关。

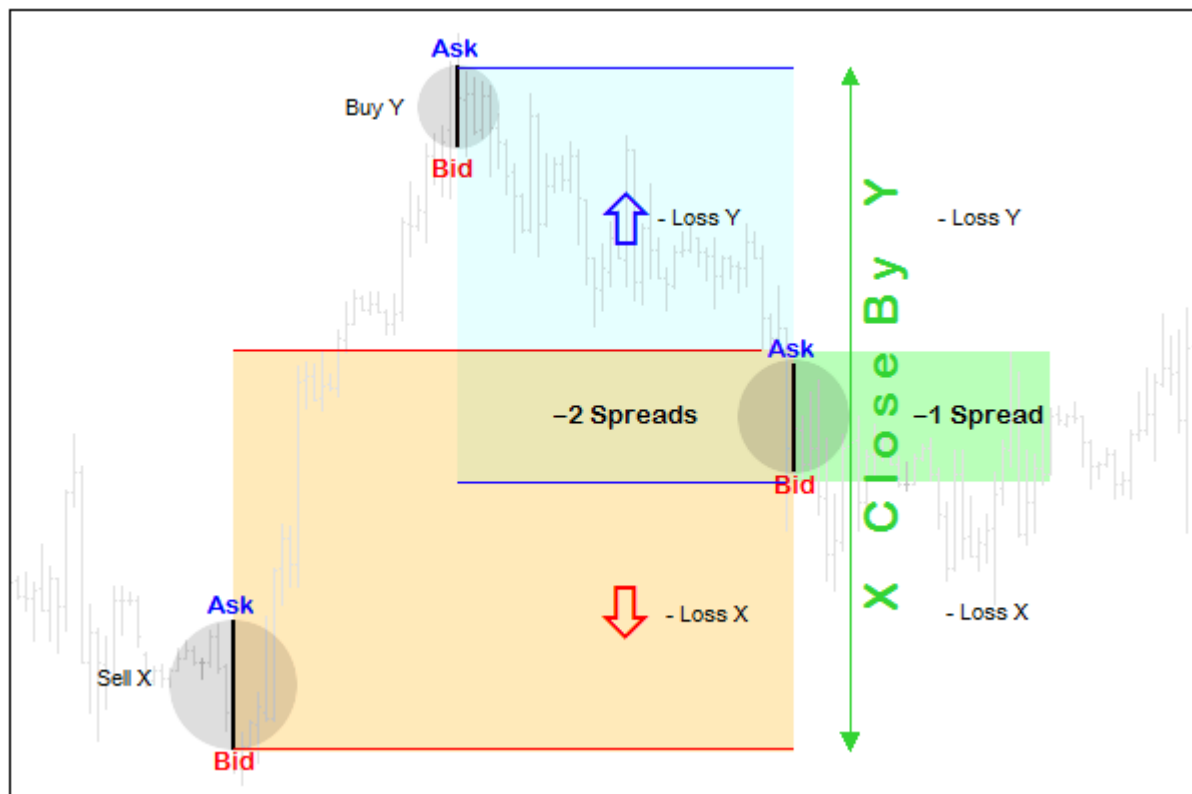
这种不对称的另一个结果是,通过改变在 `position` 和 `position_by` 字段中的订单号位置,在交易报告中的多头交易和空头交易背景下的损益统计会发生变化,例如,盈利的多头交易可以增加正好与盈利的空头交易的数量减少一样多。但如果我们假设订单执行的延迟不依赖于订单号的传输顺序,在理论上不会应影响整体结果。

下图显示了该过程的图解说明 (点差被有意夸大)。



平仓盈利仓位时的点差会计

这是一对盈利仓位的情况。如果仓位方向相反,处于亏损状态,那么当它们分别平仓时,点差将被考虑两次 (分别考虑)。对冲平仓可让你减少一个点差的损失。



平仓无盈利仓位时的点差会计

反向仓位无需大小相等。反向平仓操作将在两个交易量中的最小值上进行。

在 *MqlTradeSync.mqh* 文件中，使用带有两个仓位订单号参数的 *closeby* 方法来实现邻近运算。

```

struct MqlTradeRequestSync: public MqlTradeRequest
{
    ...
    bool closeby(const ulong ticket1, const ulong ticket2)
    {
        if(!PositionSelectByTicket(ticket1)) return false;
        double volume1 = PositionGetDouble(POSITION_VOLUME);
        if(!PositionSelectByTicket(ticket2)) return false;
        double volume2 = PositionGetDouble(POSITION_VOLUME);

        action = TRADE_ACTION_CLOSE_BY;
        position = ticket1;
        position_by = ticket2;

        ZeroMemory(result);
        if(volume1 != volume2)
        {
            // remember which position should disappear
            if(volume1 < volume2)
                result.position = ticket1;
            else
                result.position = ticket2;
        }
        return OrderSend(this, result);
    }
}

```

为了控制平仓结果，我们可在 *result.position* 变量中存储一个较小仓位的订单号。*completed* 方法和 *MqlTradeResultSync* 结构体中的一切都是为同步平仓跟踪做准备：相同的算法适用于正常的平仓。

```

struct MqlTradeRequestSync: public MqlTradeRequest
{
    ...
    bool completed()
    {
        ...
        else if(action == TRADE_ACTION_CLOSE_BY)
        {
            return result.closed(timeout);
        }
        return false;
    }
}

```

反向仓位通常被用来代替止损订单，或试图在短期修正时止盈，同时留在市场上并跟随主要走势。使用伪止损订单的选项允许你将实际平仓的决定推迟一段时间，继续分析市场走势，期待价格向正确的方向反转。但是，应记住，“锁定”仓位需要增加存款，并受掉期的影响。这就是为什么很难想象一个纯粹建立在反向仓位上的交易策略的原因，这可以作为本节的一个示例。

我们开发上一个示例中所述的基于价格-操作柱线的策略。新的 EA 交易为 *TradeCloseBy.mq5*。

我们将使用之前的信号，在检测到两根连续的柱线以相同的方向收盘时进场。负责形成这个信号的一个函数还是 *GetTradeDirection*。但是，如果这种趋势继续下去，我们允许重新进入。最大允许仓位总数将在 *PositionLimit* 输入变量中设置，默认值为 5。

GetMyPositions 函数将经历一些变化：它将有二个参数，这两个参数将是对接受仓位订单号数组的引用：分别买入和卖出。

```
#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1, ArraySize(A) * 2) - 1] = V)

int GetMyPositions(const string s, const ulong m,
    ulong &ticketsLong[], ulong &ticketsShort[])
{
    for(int i = 0; i < PositionsTotal(); ++i)
    {
        if(PositionGetSymbol(i) == s && PositionGetInteger(POSITION_MAGIC) == m)
        {
            if((ENUM_POSITION_TYPE)PositionGetInteger(POSITION_TYPE) == POSITION_TYPE_BUY
                PUSH(ticketsLong, PositionGetInteger(POSITION_TICKET));
            else
                PUSH(ticketsShort, PositionGetInteger(POSITION_TICKET));
        }
    }

    const int min = fmin(ArraySize(ticketsLong), ArraySize(ticketsShort));
    if(min == 0) return -fmax(ArraySize(ticketsLong), ArraySize(ticketsShort));
    return min;
}
```

该函数返回两个数组中最小数组的大小。当它大于零时，我们有机会反向平仓。

如果最小数组的大小为零，该函数将返回另一个数组的大小，但带有一个减号，只是为了告知调用代码：所有仓位都在同一方向。

如果两个方向都没有仓位，该函数将返回 0。

开仓仓位将保持在 *OpenPosition* 函数的控制之下 - 此处不变。

在新函数 *CloseByPosition* 中，只有在两个相反仓位的模式下才会执行平仓。换言之，该 EA 交易不能像通常那样一次平仓一单。当然，在真实的机器人中，这样的原则不太可能出现，但非常适合作为即将介绍的平仓示例。如果我们需平仓一单，为它开一个相反的仓位（这时浮动盈亏是固定的）并调用 *CloseByPosition* 两次就够了。

```

bool CloseByPosition(const ulong ticket1, const ulong ticket2)
{
    MqlTradeRequestSync request;
    request.magic = Magic;

    ResetLastError();
    // send a request and wait for it to complete
    if(request.closeby(ticket1, ticket2))
    {
        Print("Positions collapse initiated");
        if(request.completed())
        {
            Print("OK CloseBy Order/Deal/Position");
            return true; // success
        }
    }

    Print(TU::StringOf(request));
    Print(TU::StringOf(request.result));

    return false; // error
}

```

该代码使用了上述 `request.closeby` 方法。填充 `position` 和 `position_by` 字段，并调用 `OrderSend`。

`OnTick` 处理程序中说明了交易逻辑，该处理程序仅在新柱线形成时分析价格配置，并接收来自 `GetTradeDirection` 函数的信号。

```

void OnTick()
{
    static bool error = false;
    // waiting for the formation of a new bar, if there is no error
    static datetime lastBar = 0;
    if(iTime(_Symbol, _Period, 0) == lastBar && !error) return;
    lastBar = iTime(_Symbol, _Period, 0);

    const ENUM_ORDER_TYPE type = GetTradeDirection();
    ...
}

```

接下来，我们用工作交易品种的仓位标签和给定的 `Magic` 编号填充 `ticketsLong` 和 `ticketsShort` 数组。如果 `GetMyPositions` 函数返回一个大于零的值，它将给出已形成的相反仓位对数量。可以使用 `CloseByPosition` 函数将它们封闭在一个循环中。在这种情况下，仓位对的组合是随机选择的（按照在终端环境中的仓位顺序），但在实践中，务必通过交易量或以最盈利仓位对首先平仓的方式来选择仓位对。

```

ulong ticketsLong[], ticketsShort[];
const int n = GetMyPositions(_Symbol, Magic, ticketsLong, ticketsShort);
if(n > 0)
{
    for(int i = 0; i < n; ++i)
    {
        error = !CloseByPosition(ticketsShort[i], ticketsLong[i]) && error;
    }
}
...

```

对于 n 的任何其他值，都要检查是否有信号（可能是重复的）进场，并通过调用 *OpenPosition* 来执行。

```

else if(type == ORDER_TYPE_BUY || type == ORDER_TYPE_SELL)
{
    error = !OpenPosition(type);
}
...

```

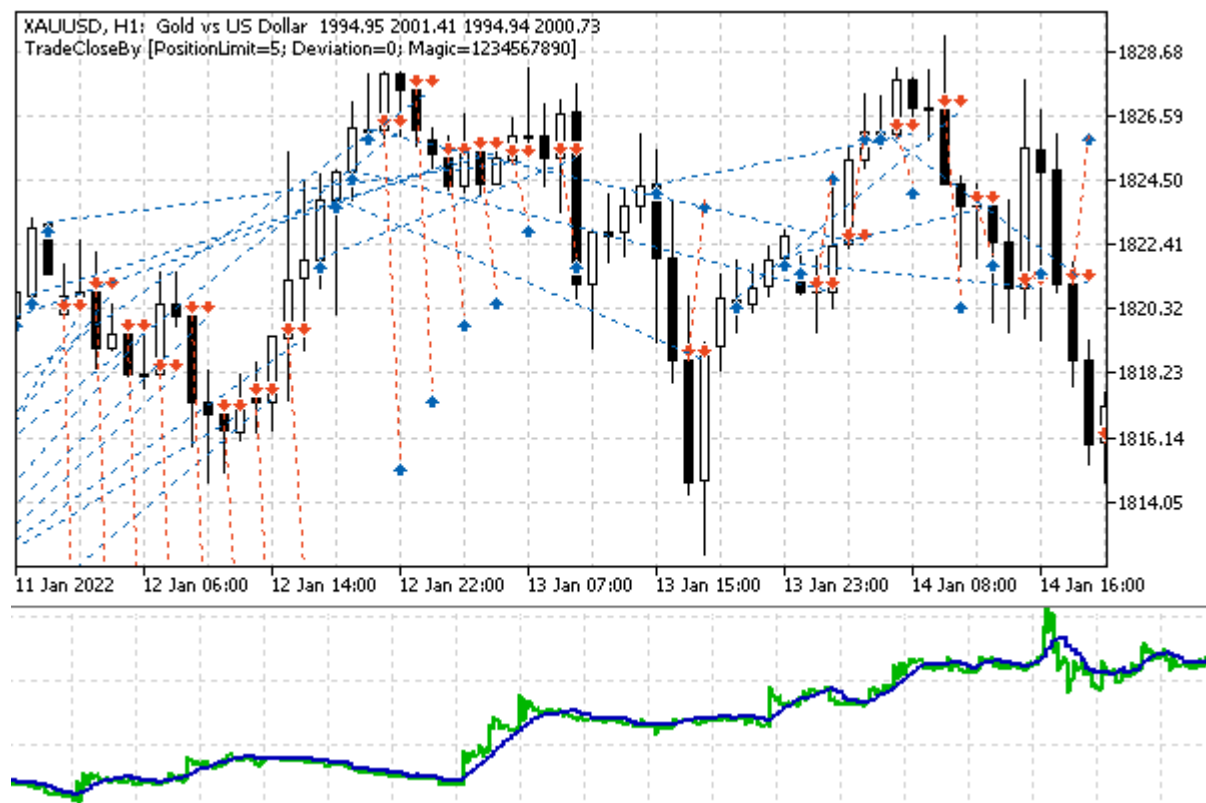
最后，如果仍然有未平仓仓位，但处于相同的方向，可检查其数量是否已经达到极限，在这种情况下，我们可形成相反的仓位，以便在下一根柱线上“折叠”其中的两个仓位（从而平仓旧仓位中的任何一个）。

```

else if(n < 0)
{
    if(-n >= (int)PositionLimit)
    {
        if(ArraySize(ticketsLong) > 0)
        {
            error = !OpenPosition(ORDER_TYPE_SELL);
        }
        else // (ArraySize(ticketsShort) > 0)
        {
            error = !OpenPosition(ORDER_TYPE_BUY);
        }
    }
}
}

```

我们从 2022 年初开始在 XAUUSD, H1 的测试程序中运行 EA 交易，使用默认设置。以下是该程序进程中的持仓图以及余额曲线。



对 XAUUSD,H1 运行 TradeCloseBy 的测试结果

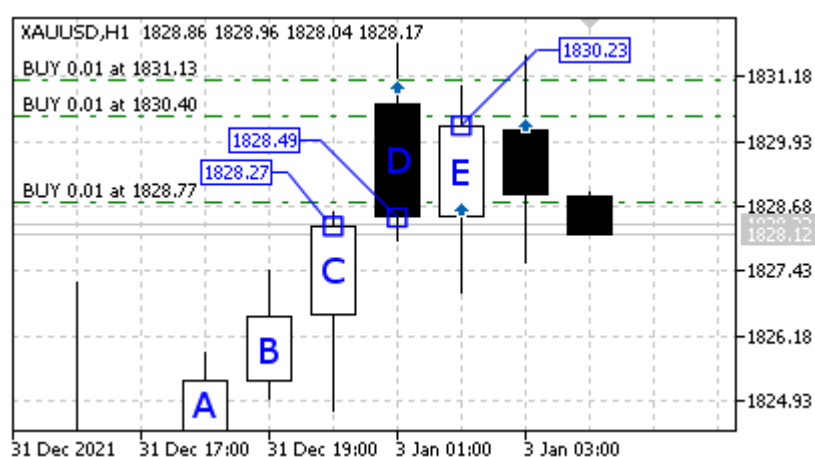
在日志中很容易找到一个趋势结束的时刻 (用 2 号至 4 号的订单号买入), 交易开始以相反的方向产生 (卖出 5 号), 之后触发对冲平仓。

```

2022.01.03 01:05:00 instant buy 0.01 XAUUSD at 1831.13 (1830.63 / 1831.13 / 1830.63)
2022.01.03 01:05:00 deal #2 buy 0.01 XAUUSD at 1831.13 done (based on order #2)
2022.01.03 01:05:00 deal performed [#2 buy 0.01 XAUUSD at 1831.13]
2022.01.03 01:05:00 order performed buy 0.01 at 1831.13 [#2 buy 0.01 XAUUSD at 1831.1]
2022.01.03 01:05:00 Waiting for position for deal D=2
2022.01.03 01:05:00 OK New Order/Deal/Position
2022.01.03 2:00:00 instant buy 0.01 XAUUSD at 1828.77 (1828.47 / 1828.77 / 1828.47)
2022.01.03 2:00:00 deal #3 buy 0.01 XAUUSD at 1828.77 done (based on order #3)
2022.01.03 2:00:00 deal performed [#3 buy 0.01 XAUUSD at 1828.77]
2022.01.03 2:00:00 order performed buy 0.01 at 1828.77 [#3 buy 0.01 XAUUSD at 1828.77]
2022.01.03 2:00:00 Waiting for position for deal D=3
2022.01.03 2:00:00 OK New Order/Deal/Position
2022.01.03 3:00:00 instant buy 0.01 XAUUSD at 1830.40 (1830.16 / 1830.40 / 1830.16)
2022.01.03 3:00:00 deal #4 buy 0.01 XAUUSD at 1830.40 done (based on order #4)
2022.01.03 3:00:00 deal performed [#4 buy 0.01 XAUUSD at 1830.40]
2022.01.03 3:00:00 order performed buy 0.01 at 1830.40 [#4 buy 0.01 XAUUSD at 1830.40]
2022.01.03 3:00:00 Waiting for position for deal D=4
2022.01.03 3:00:00 OK New Order/Deal/Position
2022.01.03 05:00:00 instant sell 0.01 XAUUSD at 1826.22 (1826.22 / 1826.45 / 1826.22)
2022.01.03 05:00:00 deal #5 sell 0.01 XAUUSD at 1826.22 done (based on order #5)
2022.01.03 05:00:00 deal performed [#5 sell 0.01 XAUUSD at 1826.22]
2022.01.03 05:00:00 order performed sell 0.01 at 1826.22 [#5 sell 0.01 XAUUSD at 1826]
2022.01.03 5:00:00 Waiting for position for deal D=5
2022.01.03 5:00:00 OK New Order/Deal/Position
2022.01.03 06:00:00 close position #5 sell 0.01 XAUUSD by position #2 buy 0.01 XAUUSD
2022.01.03 6:00:00 deal #6 buy 0.01 XAUUSD at 1831.13 done (based on order #6)
2022.01.03 06:00:00 deal #7 sell 0.01 XAUUSD at 1826.22 done (based on order #6)
2022.01.03 06:00:00 Positions collapse initiated
2022.01.03 06:00:00 OK CloseBy Order/Deal/Position

```

交易 #3 是个很有趣的异常情况。细心的读者会注意到，它的开盘价比前一个低，似乎违反了我们的策略。事实上，此处没有错误，这是因为信号的条件写得尽可能简单：仅基于该柱线的收盘价。因此，一根空头反转柱线 (D) 以向上跳空开盘，收于前一根多头柱线 (C) 的末端上方，产生了买入信号。下面的截图说明了这种情况。



收盘价呈上升趋势的交易

序列 A、B、C、D 和 E 中的所有柱线收盘都比前一根柱线高，鼓励继续买入。为了排除这种情况，应另外分析柱线本身的方向。

本例中最后需要注意的是 *OnInit* 函数。由于 EA 交易使用 `TRADE_ACTION_CLOSE_BY` 操作，因此需要在此检查相关账户和工作交易品种设置。

```
int OnInit()
{
    ...
    if(AccountInfoInteger(ACCOUNT_MARGIN_MODE) != ACCOUNT_MARGIN_MODE_RETAIL_HEDGING)
    {
        Alert("An account with hedging is required for this EA!");
        return INIT_FAILED;
    }

    if((SymbolInfoInteger(_Symbol, SYMBOL_ORDER_MODE) & SYMBOL_ORDER_CLOSEBY) == 0)
    {
        Alert("'Close By' mode is not supported for ", _Symbol);
        return INIT_FAILED;
    }

    return INIT_SUCCEEDED;
}
```

如果其中一个特性不支持交叉平仓，EA 交易将无法继续工作。创建工作机器人时，这些检查通常在交易算法内部进行，并将程序切换到替代模式，特别是单次平仓并在净额结算时保持总仓位的情况下。

6.4.19 下达挂单

在 `订单类型` 中，我们从理论上探讨了平台支持的所有挂单下达选项。从实用的角度来看，订单是使用 `OrderSend/OrderSendAsync` 函数创建的，根据特殊的规则预先填充了请求结构体 `MqlTradeRequest`。具体而言，`action` 字段必须包含 `TRADE_ACTION_PENDING` 值（在 `ENUM_TRADE_REQUEST_ACTIONS` 枚举中）。考虑到这一点，以下字段为必填：

- ⌚ `action`
- ⌚ `symbol`
- ⌚ `volume`
- ⌚ `price`
- ⌚ `type`（默认值 0 对应于 `ORDER_TYPE_BUY`）
- ⌚ `type_filling`（默认值 0 对应于 `ORDER_FILLING_FOK`）
- ⌚ `type_time`（默认值 0 对应于 `ORDER_TIME_GTC`）
- ⌚ `expiration`（默认值为 0，不适用于 `ORDER_TIME_GTC`）

如果零默认值适用于该任务，则可以跳过最后四个字段中的某些字段。

只有订单类型为 `ORDER_TYPE_BUY_STOP_LIMIT` 和 `ORDER_TYPE_SELL_STOP_LIMIT` 时，才必须填写 `stoplimit` 字段。

以下字段可选填：

- ⌚ `sl`
- ⌚ `tp`

⌚ magic

⌚ comment

sl 和 *tp* 中的零值表示没有保护水平。

我们在 *MqlTradeSync.mqh* 文件中将检查值和将字段填充方法添加到我们的结构体中。所有类型的订单的形成原理都是相同的，所以我们考虑一些下达限制买卖订单的特殊情况。其余类型的仅在字段类型的值方面有区别。具有全套必需字段以及保护水平的公共方法是根据类型命名的：*buyLimit* 和 *sellLimit*。

```
ulong buyLimit(const string name, const double lot, const double p,
               const double stop = 0, const double take = 0,
               ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0)
{
    type = ORDER_TYPE_BUY_LIMIT;
    return _pending(name, lot, p, stop, take, duration, until);
}

ulong sellLimit(const string name, const double lot, const double p,
                const double stop = 0, const double take = 0,
                ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0)
{
    type = ORDER_TYPE_SELL_LIMIT;
    return _pending(name, lot, p, stop, take, duration, until);
}
```

由于该结构体包含 *symbol* 字段，而该字段在构造函数中的初始化是可选的，所以有一些类似的无 *name* 参数的方法：它们通过传递 *symbol* 作为第一个参数来调用上述方法。因此，要以最少的工作量创建订单，请编写以下代码：

```
MqlTradeRequestSync request; // by default uses the current chart symbol
request.buyLimit(volume, price);
```

用于检查传递的值、规范化、并将其保存在结构体字段中以及创建挂单代码的常规部分已经被移到了辅助方法 *_pending* 中。如果成功，则返回订单号，如果失败，则返回 0。

```

ulong _pending(const string name, const double lot, const double p,
    const double stop = 0, const double take = 0,
    ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0,
    const double origin = 0)
{
    action = TRADE_ACTION_PENDING;
    if(!setSymbol(name)) return 0;
    if(!setVolumePrices(lot, p, stop, take, origin)) return 0;
    if(!setExpiration(duration, until)) return 0;
    if((SymbolInfoInteger(name, SYMBOL_ORDER_MODE) & (1 << (type / 2))) == 0)
    {
        Print(StringFormat("pending orders %s not allowed for %s",
            EnumToString(type), name));
        return 0;
    }
    ZeroMemory(result);
    if(OrderSend(this, result)) return result.order;
    return 0;
}

```

我们已经知道如何填充 `action` 字段以及如何从以前的交易操作中调用 `setSymbol` 和 `setVolumePrices` 的方法。

多字符串 `if` 运算符可确保正在准备的运算出现在允许交易品种运算中 (在 `SYMBOL_ORDER_MODE` 特性中指定)。整数型除法 `type` 除以二并将结果值移位 1, 用于在允许的订单类型掩码中设置正确的位。这是由于 `ENUM_ORDER_TYPE` 枚举中的常量和 `SYMBOL_ORDER_MODE` 特性的组合。例如, `ORDER_TYPE_BUY_STOP` 和 `ORDER_TYPE_SELL_STOP` 的值分别为 4 和 5, 除以 2 均为 2 (去掉小数)。运算 `1 << 2` 的结果 4 等于 `SYMBOL_ORDER_STOP`。

挂单的一个特殊功能是处理到期日期。`setExpiration` 方法用于处理到期日期。在此方法中, 应确保允许为该交易品种指定的到期模式 `ENUM_ORDER_TYPE_TIME` (`duration`) 并且正确填写 `until` 中的日期和时间。

```

bool setExpiration(ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until
{
    const int modes = (int)SymbolInfoInteger(symbol, SYMBOL_EXPIRATION_MODE);
    if(((1 << duration) & modes) != 0)
    {
        type_time = duration;
        if((duration == ORDER_TIME_SPECIFIED || duration == ORDER_TIME_SPECIFIED_DAY
            && until == 0)
        {
            Print(StringFormat("datetime is 0, "
                "but it's required for order expiration mode %s",
                EnumToString(duration)));
            return false;
        }
        if(until > 0 && until <= TimeTradeServer())
        {
            Print(StringFormat("expiration datetime %s is in past, server time is %s"
                TimeToString(until), TimeToString(TimeTradeServer())));
            return false;
        }
        expiration = until;
    }
    else
    {
        Print(StringFormat("order expiration mode %s is not allowed for %s",
            EnumToString(duration), symbol));
        return false;
    }
    return true;
}

```

允许模式的位掩码在 `SYMBOL_EXPIRATION_MODE` 特性中提供。掩码中的位和常量 `ENUM_ORDER_TYPE_TIME` 的组合是这样的，我们只需计算表达式 `1 << duration`，并将其叠加在掩码上：非零值表示模式存在。

对于 `ORDER_TIME_SPECIFIED` 和 `ORDER_TIME_SPECIFIED_DAY` 模式，具有特定 *datetime* 值的 *expiration* 字段不能为空。此外，指定的日期和时间不能是过去的时间。

由于前面介绍的 *_pending* 方法最终使用 *OrderSend* 向服务器发送请求，所以我们的程序必须确保实际创建了带有接收订单号的订单（这对于可以输出到外部交易系统的限价订单尤其重要）。因此，在用于“阻止”结果控制的 *completed* 方法中，我们将为 `TRADE_ACTION_PENDING` 操作添加一个分支。

```

bool completed()
{
    // old processing code
    // TRADE_ACTION_DEAL
    // TRADE_ACTION_SLTP
    // TRADE_ACTION_CLOSE_BY
    ...
    else if(action == TRADE_ACTION_PENDING)
    {
        return result.placed(timeout);
    }
    ...
    return false;
}

```

在 *MqlTradeResultSync* 结构体中，我们添加了 *placed* 方法。

```

bool placed(const ulong msc = 1000)
{
    if(retcode != TRADE_RETCODE_DONE
        && retcode != TRADE_RETCODE_DONE_PARTIAL)
    {
        return false;
    }

    if(!wait(orderExist, msc))
    {
        Print("Waiting for order: #" + (string)order);
        return false;
    }
    return true;
}

```

其主要任务是使用 *orderExist* 函数中的 *wait* 等待订单的出现：其已经被用于 开仓验证的第一阶段。

为了测试新功能，我们实现 EA 交易 *PendingOrderSend.mq5*。其支持使用输入变量选择挂单类型及其所有特性，之后执行确认请求。

```

enum ENUM_ORDER_TYPE_PENDING
{
    PENDING_BUY_STOP = ORDER_TYPE_BUY_STOP,           // UI interface strings
    PENDING_SELL_STOP = ORDER_TYPE_SELL_STOP,          // ORDER_TYPE_BUY_STOP
    PENDING_BUY_LIMIT = ORDER_TYPE_BUY_LIMIT,          // ORDER_TYPE_SELL_STOP
    PENDING_SELL_LIMIT = ORDER_TYPE_SELL_LIMIT,        // ORDER_TYPE_BUY_LIMIT
    PENDING_BUY_STOP_LIMIT = ORDER_TYPE_BUY_STOP_LIMIT, // ORDER_TYPE_SELL_LIMIT
    PENDING_SELL_STOP_LIMIT = ORDER_TYPE_SELL_STOP_LIMIT, // ORDER_TYPE_BUY_STOP_LIMIT
};

input string Symbol;           // Symbol (empty = current _Symbol)
input double Volume;          // Volume (0 = minimal lot)
input ENUM_ORDER_TYPE_PENDING Type = PENDING_BUY_STOP;
input int Distance2SLTP = 0;   // Distance to SL/TP in points (0 = no)
input ENUM_ORDER_TYPE_TIME Expiration = ORDER_TIME_GTC;
input datetime Until = 0;
input ulong Magic = 1234567890;
input string Comment;

```

每次启动或更改参数时，EA 交易都会创建一个新订单。自动 订单删除 尚未提供。我们将在后面讨论这种操作类型。在这方面，不要忘记手动删除订单。

与前面的一些示例一样，一次性订单下达是基于计时器执行的（因此，你应首先确保市场是开放的）。

```

void OnTimer()
{
    // execute once and wait for the user to change the settings
    EventKillTimer();

    const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
    if(PlaceOrder((ENUM_ORDER_TYPE)Type, symbol, Volume,
        Distance2SLTP, Expiration, Until, Magic, Comment))
    {
        Alert("Pending order placed - remove it manually, please");
    }
}

```

PlaceOrder 函数接受所有设置作为参数，发送一个请求，并返回一个成功指示符（非零订单号）。所有受支持类型的订单都提供了与当前价格的预先填充距离，该距离作为每日报价范围的一部分进行计算。

```

ulong PlaceOrder(const ENUM_ORDER_TYPE type,
    const string symbol, const double lot,
    const int sltp, ENUM_ORDER_TYPE_TIME expiration, datetime until,
    const ulong magic = 0, const string comment = NULL)
{
    static double coefficients[] = // indexed by order type
    {
        0 ,    // ORDER_TYPE_BUY - not used
        0 ,    // ORDER_TYPE_SELL - not used
        -0.5,  // ORDER_TYPE_BUY_LIMIT - slightly below the price
        +0.5,  // ORDER_TYPE_SELL_LIMIT - slightly above the price
        +1.0,  // ORDER_TYPE_BUY_STOP - far above the price
        -1.0,  // ORDER_TYPE_SELL_STOP - far below the price
        +0.7,  // ORDER_TYPE_BUY_STOP_LIMIT - average above the price
        -0.7,  // ORDER_TYPE_SELL_STOP_LIMIT - average below the price
        0 ,    // ORDER_TYPE_CLOSE_BY - not used
    };
    ...
}

```

例如, `ORDER_TYPE_BUY_LIMIT` 的系数为 `-0.5`, 表示订单将低于当前价格一半的日线范围下边(在该范围内反弹), `ORDER_TYPE_BUY_STOP` 的系数为 `+1.0`, 表示订单将位于该范围的上边界突破)。

日线范围本身的计算方式如下。

```

const double range = iHigh(symbol, PERIOD_D1, 1) - iLow(symbol, PERIOD_D1, 1);
Print("Autodetected daily range: ", (float)range);
...

```

我们找到了下面需要的交易量和点值。

```

const double volume = lot == 0 ? SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : lot;
const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);

```

根据总范围中给定的系数, 在 `price` 变量中计算下达订单的价格水平。

```

const double price = TU::GetCurrentPrice(type, symbol) + range * coefficients[type];

```

只能为 `*_STOP_LIMIT` 订单填写 `stoplimit` 字段。其值存储在 `origin` 变量中。

```

const bool stopLimit =
    type == ORDER_TYPE_BUY_STOP_LIMIT ||
    type == ORDER_TYPE_SELL_STOP_LIMIT;
const double origin = stopLimit ? TU::GetCurrentPrice(type, symbol) : 0;

```

当这两种类型的订单被触发时, 将以当前价格下达一个新的挂单。事实上, 在这种情况下, 价格从当前值移动到订单被激活的 `price` 水平, 因此“以前的当前”价格成为限价订单指示的正确反弹水平。我们将在下面说明这种情况。

保护水平是使用 `TU::TradeDirection` 对象确定的。对于限制止损订单, 我们从 `origin` 开始计算。

```

TU::TradeDirection dir(type);
const double stop = sltp == 0 ? 0 :
    dir.negative(stopLimit ? origin : price, sltp * point);
const double take = sltp == 0 ? 0 :

```

```
dir.positive(stopLimit ? origin : price, sltp * point);
```

接下来,说明其结构体并填写可选字段。

```
MqlTradeRequestSync request(symbol);

request.magic = magic;
request.comment = comment;
// request.type_filling = SYMBOL_FILLING_FOK;
```

此处,你可以选择填充模式。默认情况下, *MqlTradeRequestSync* 会自动选择第一个允许的模式 *ENUM_ORDER_TYPE_FILLING*。

根据用户选择的订单类型,我们可调用某一种交易方法。

```
ResetLastError();
// fill in and check the required fields, send the request
ulong order = 0;
switch(type)
{
case ORDER_TYPE_BUY_STOP:
    order = request.buyStop(volume, price, stop, take, expiration, until);
    break;
case ORDER_TYPE_SELL_STOP:
    order = request.sellStop(volume, price, stop, take, expiration, until);
    break;
case ORDER_TYPE_BUY_LIMIT:
    order = request.buyLimit(volume, price, stop, take, expiration, until);
    break;
case ORDER_TYPE_SELL_LIMIT:
    order = request.sellLimit(volume, price, stop, take, expiration, until);
    break;
case ORDER_TYPE_BUY_STOP_LIMIT:
    order = request.buyStopLimit(volume, price, origin, stop, take, expiration, until);
    break;
case ORDER_TYPE_SELL_STOP_LIMIT:
    order = request.sellStopLimit(volume, price, origin, stop, take, expiration, until);
    break;
}
...
```

如果收到订单号,我们应等待其出现在终端的交易环境中。


```

if(order != 0)
{
    Print("OK order sent: #=", order);
    if(request.completed()) // expect result (order confirmation)
    {
        Print("OK order placed");
    }
}
Print(TU::StringOf(request));
Print(TU::StringOf(request.result));
return order;
}

```

我们使用默认设置运行 EURUSD 图表上的 EA 交易，并另外选择到 1000 点保护水平的距离。我们将在日志中看到以下条目（假设默认设置与你账户中的 EURUSD 权限相匹配）。

```

Autodetected daily range: 0.01413
OK order sent: #=1282106395
OK order placed
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, »
    » @ 1.11248, SL=1.10248, TP=1.12248, ORDER_TIME_GTC, M=1234567890
DONE, #=1282106395, V=0.01, Request executed, Req=91
Alert: Pending order placed - remove it manually, please

```

以下是其在图表上的显示效果：



挂单 ORDER_TYPE_BUY_STOP

我们手动删除该订单，并将订单类型更改为 ORDER_TYPE_BUY_STOP_LIMIT。结果情况会更加复杂。



挂单 ORDER_TYPE_BUY_STOP_LIMIT

上面一对点划线所在的价格为订单触发价格，因此 `ORDER_TYPE_BUY_LIMIT` 订单将在当前价格水平下达，*Stop Loss* 和 *Take Profit* 值用红线标记。期货 `ORDER_TYPE_BUY_LIMIT` 订单的 *Take Profit* 水平实际上与新创建的初步订单 `ORDER_TYPE_BUY_STOP_LIMIT` 的激活水平一致。

作为自学的另一个示例，本书包含了一个 EA 交易 *AllPendingsOrderSend.mq5*，EA 交易一次设置 6 个挂单：每种类型 1 个。



所有类型的挂单

如果使用默认设置运行该 EA 交易，你可能会得到如下日志条目：

```
Autodetected daily range: 0.01413
OK order placed: #=1282032135
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, »
    » @ 1.08824, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032135, V=0.01, Request executed, Req=73
OK order placed: #=1282032136
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, »
    » @ 1.10238, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032136, V=0.01, Request executed, Req=74
OK order placed: #=1282032138
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, »
    » @ 1.10944, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032138, V=0.01, Request executed, Req=75
OK order placed: #=1282032141
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP, V=0.01, ORDER_FILLING_FOK, »
    » @ 1.08118, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032141, V=0.01, Request executed, Req=76
OK order placed: #=1282032142
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, »
    » @ 1.10520, X=1.09531, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032142, V=0.01, Request executed, Req=77
OK order placed: #=1282032144
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, »
    » @ 1.08542, X=1.09531, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032144, V=0.01, Request executed, Req=78
Alert: 6 pending orders placed - remove them manually, please
```

6.4.20 修改挂单

MetaTrader 5 允许你修改挂单的某些特性，包括激活价格、保护水平和到期日期。订单类型或交易量等主要特性不能更改。若需更改，你应 **删除** 该订单并将其替换为另一个订单。服务器本身可以更改订单类型的唯一情况是激活止损限价订单，该订单会变成相应的限价订单。

要以编程方式修改订单，需要通过 `TRADE_ACTION_MODIFY` 操作执行：需要将该常量写入 **action** 字段（属于 *MqlTradeRequest* 结构体），而且需要在通过 *OrderSend* 或 *OrderSendAsync* 函数将其发送到服务器之前执行。修改订单的订单号在 **order** 字段中指示。考虑到 **action** 和 **order**，此操作的完整必填字段列表包括：

- **action**
- **order**
- **price**
- **type_time**（默认值 0 对应于 `ORDER_TIME_GTC`）
- **expiration**（默认值 0 对于 `ORDER_TIME_GTC`）
- **type_filling**（默认值 0 对应于 `ORDER_FILLING_FOK`）
- **stoplimit**（仅适用于 `ORDER_TYPE_BUY_STOP_LIMIT` 和 `ORDER_TYPE_SELL_STOP_LIMIT` 类型的订单）

选填字段：

- **sl**
- **tp**

如果已经为订单设置了保护水平，则应注明保护水平，以便保存。零值表示删除了 **Stop Loss** 和/或 **Take Profit**。

在 *MqlTradeRequestSync* 结构体 (*MqlTradeSync.mqh*) 中，订单修改的实现包含在 **modify** 方法中。

```

struct MqlTradeRequestSync: public MqlTradeRequest
{
    ...
    bool modify(const ulong ticket,
        const double p, const double stop = 0, const double take = 0,
        ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0,
        const double origin = 0)
    {
        if(!OrderSelect(ticket)) return false;

        action = TRADE_ACTION_MODIFY;
        order = ticket;

        // the following fields are needed for checks inside subfunctions
        type = (ENUM_ORDER_TYPE)OrderGetInteger(ORDER_TYPE);
        symbol = OrderGetString(ORDER_SYMBOL);
        volume = OrderGetDouble(ORDER_VOLUME_CURRENT);

        if(!setVolumePrices(volume, p, stop, take, origin)) return false;
        if(!setExpiration(duration, until)) return false;
        ZeroMemory(result);
        return OrderSend(this, result);
    }
}

```

在 *if* 运算符的专用分支中，请求的实际执行再次在 *completed* 方法中完成。

```

bool completed()
{
    ...
    else if(action == TRADE_ACTION_MODIFY)
    {
        result.order = order;
        result.bid = sl;
        result.ask = tp;
        result.price = price;
        result.volume = stoplimit;
        return result.modified(timeout);
    }
    ...
}

```

为了让 *MqlTradeResultSync* 结构体知道已编辑订单特性的新值，并能够将其与结果进行比较，我们将这些新值写在自由字段中（在这种类型的请求中，服务器不会填充这些字段）。此外，在 *modified* 方法中，结果结构体正在等待应用修改。

```

struct MqlTradeResultSync: public MqlTradeResult
{
    ...
    bool modified(const ulong msc = 1000)
    {
        if(retcode != TRADE_RETCODE_DONE && retcode != TRADE_RETCODE_PLACED)
        {
            return false;
        }

        if(!wait(orderModified, msc))
        {
            Print("Order not found in environment: #" + (string)order);
            return false;
        }
        return true;
    }

    static bool orderModified(MqlTradeResultSync &ref)
    {
        if(!(OrderSelect(ref.order) || HistoryOrderSelect(ref.order)))
        {
            Print("OrderSelect failed: #" + (string)ref.order);
            return false;
        }
        return TU::Equal(ref.bid, OrderGetDouble(ORDER_SL))
            && TU::Equal(ref.ask, OrderGetDouble(ORDER_TP))
            && TU::Equal(ref.price, OrderGetDouble(ORDER_PRICE_OPEN))
            && TU::Equal(ref.volume, OrderGetDouble(ORDER_PRICE_STOPLIMIT));
    }
}

```

此处我们看到如何使用 `OrderGetDouble` 函数读取订单特性，并与指定的值进行比较。所有这一切都按照我们已经熟悉的过程发生，在等待函数内的一个循环中，在 `msc` 的某个超时时间内（默认为 1000 毫秒）。

例如，我们使用 EA 交易 `PendingOrderModify.mq5`，同时从 `PendingOrderSend.mq5` 继承一些代码片段。具体来说，是一组输入参数和用于创建新订单的 `PlaceOrder` 函数。如果给定的交易品种和 `Magic` 号组合没有订单，则在第一次启动时使用，从而确保 EA 交易有修改内容。

需要一个新函数来查找合适的订单：`GetMyOrder`。该函数非常类似于 `GetMyPosition` 函数，后者用于在 `仓位跟踪` 示例 (`TrailingStop.mq5`) 中查找合适的仓位。`GetMyOrder` 中使用了内置 MQL5 API 函数，从其名称中就能大致推断其用途，技术说明将在 [单独的章节](#) 中介绍。

```

ulong GetMyOrder(const string name, const ulong magic)
{
    for(int i = 0; i < OrdersTotal(); ++i)
    {
        ulong t = OrderGetTicket(i);
        if(OrderGetInteger(ORDER_MAGIC) == magic
            && OrderGetString(ORDER_SYMBOL) == name)
        {
            return t;
        }
    }

    return 0;
}

```

现在缺少输入参数 *Distance2SLTP*。取而代之的是，新的 EA 交易将自动计算价格的日线范围，并在该范围的一半处设置保护水平。每天开始时，将重新计算 *sl* 和 *tp* 字段中的范围和新水平。将根据新值生成订单修改请求。

那些触发并转为仓位的挂单将在达到 *Stop Loss* 或 *Take Profit* 时平仓。终端可以通知 MQL 程序关于挂单激活和平仓的信息，前提是你终端中说明了 *交易事件* 处理程序。例如，如果存在未结仓位，便可以避免创建新订单。但也可以使用当前策略。所以，我们以后再处理事件。

EA 交易的主要逻辑在 *OnTick* 处理程序中实现。

```

void OnTick()
{
    static datetime lastDay = 0;
    static const uint DAYLONG = 60 * 60 * 24; // number of seconds in a day
    //discard the "fractional" part, i.e. time
    if(TimeTradeServer() / DAYLONG * DAYLONG == lastDay) return;
    ...
}

```

该函数开头的两行确保算法在每天开始时运行一次。为此，我们计算不带时间的当前日期，并将其与包含上次成功日期的 *lastDay* 变量值进行比较。当然，在该函数结束时，成功或错误状态会变得很清楚，所以我们稍后再来讨论它。

接下来，计算前一天的价格范围。

```

const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
const double range = iHigh(symbol, PERIOD_D1, 1) - iLow(symbol, PERIOD_D1, 1);
Print("Autodetected daily range: ", (float)range);
...

```

根据 *GetMyOrder* 函数中是否有订单，我们将通过 *PlaceOrder* 创建一个新订单，或者使用 *ModifyOrder* 编辑一个现有订单。

```

uint retcode = 0;
ulong ticket = GetMyOrder(symbol, Magic);
if(!ticket)
{
    retcode = PlaceOrder((ENUM_ORDER_TYPE)Type, symbol, Volume,
        range, Expiration, Until, Magic);
}
else
{
    retcode = ModifyOrder(ticket, range, Expiration, Until);
}
...

```

PlaceOrder 和 *ModifyOrder* 这两个函数都基于 EA 交易的输入参数和找到的价格范围运行。这两个含税可返回请求状态，需要以某种方式对其进行分析，以决定采取何种操作：

- 如果请求成功，则更新 *lastDay* 变量（订单已更新，EA 交易休眠至第二天开始）
- 如果出现临时问题（例如，交易时段还没有开始），则将前一日期留在 *lastDay* 中，在下一个分时报价处再试一次
- 如果检测到严重问题，则停止 EA 交易（例如，所选订单类型或交易方向不允许出现在某个交易品种上）

```

...
if(/* some kind of retcode analysis */)
{
    lastDay = TimeTradeServer() / DAYLONG * DAYLONG;
}
}

```

在“平仓：全部和部分”一节中，我们使用了一个带有 `IS_TANGIBLE` 宏的简化分析，该宏给出了“是”和“否”类别的答案，以指示是否存在错误。显然，这种方法需要改进，我们将很快回到这个问题上来。现在，我们将重点介绍 EA 交易的主要功能。

PlaceOrder 函数的源代码与前一个示例相比几乎没有变化。*ModifyOrder* 如下所示。

注意，我们是根据日线范围来确定订单位置的，并对日线范围应用了系数表。但是，原则并没有改变，因为我们现在有两个处理订单的函数，*PlaceOrder* 和 *ModifyOrder*，*Coefficients* 表被放置在全局背景中。我们在此不再赘述，将直接讨论 *ModifyOrder* 函数。

```

uint ModifyOrder(const ulong ticket, const double range,
    ENUM_ORDER_TYPE_TIME expiration, datetime until)
{
    // default values
    const string symbol = OrderGetString(ORDER_SYMBOL);
    const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
    ...
}

```

价格水平的计算取决于订单类型和传递的范围。


```

const ENUM_ORDER_TYPE type = (ENUM_ORDER_TYPE)OrderGetInteger(ORDER_TYPE);
const double price = TU::GetCurrentPrice(type, symbol) + range * Coefficients[type];

// origin is filled only for orders *_STOP_LIMIT
const bool stopLimit =
    type == ORDER_TYPE_BUY_STOP_LIMIT ||
    type == ORDER_TYPE_SELL_STOP_LIMIT;
const double origin = stopLimit ? TU::GetCurrentPrice(type, symbol) : 0;

TU::TradeDirection dir(type);
const int sltp = (int)(range / 2 / point);
const double stop = sltp == 0 ? 0 :
    dir.negative(stopLimit ? origin : price, sltp * point);
const double take = sltp == 0 ? 0 :
    dir.positive(stopLimit ? origin : price, sltp * point);
...

```

计算完所有值后，我们将创建一个 *MqlTradeRequestSync* 结构体的对象并执行请求。

```

MqlTradeRequestSync request(symbol);

ResetLastError();
// pass the data for the fields, send the order and wait for the result
if(request.modify(ticket, price, stop, take, expiration, until, origin)
    && request.completed())
{
    Print("OK order modified: #=", ticket);
}

Print(TU::StringOf(request));
Print(TU::StringOf(request.result));
return request.result.retcode;
}

```

为了分析我们必须在 *OnTick* 内部的调用块中执行的 *retcode*，开发了一种新的机制来补充文件 *TradeRetcode.mqh*。所有服务器返回码都被分成几个“严重性”组，由 *TRADE_RETCODE_SEVERITY* 枚举的元素进行描述。

```
enum TRADE_RETCODE_SEVERITY
{
    SEVERITY_UNDEFINED,    // something non-standard - just output to the log
    SEVERITY_NORMAL,      // normal operation
    SEVERITY_RETRY,        // try updating environment/prices again (probably several t
    SEVERITY_TRY_LATER,    // we should wait and try again
    SEVERITY_REJECT,       // request denied, probably(!) you can try again
    //
    SEVERITY_INVALID,      // need to fix the request
    SEVERITY_LIMITS,       // need to check the limits and fix the request
    SEVERITY_PERMISSIONS,  // it is required to notify the user and change the program/
    SEVERITY_ERROR,        // stop, output information to the log and to the user
};
```

简言之，前半部分对应于可恢复的错误：通常只需要等待一段时间，然后重试请求。后半部分要求你更改请求的内容，检查账户或交易品种设置，程序的权限，最坏的情况下，需要停止交易。如果愿意的话，可以在 `SEVERITY_REJECT` 之前而不是之后画一条条件分隔线，因为它现在是高亮显示的。

通过 *TradeCodeSeverity* 函数 (提供了缩写版本) 将所有代码分组。

```
TRADE_RETCODE_SEVERITY TradeCodeSeverity(const uint retcode)
{
    static const TRADE_RETCODE_SEVERITY severities[] =
    {
        ...
        SEVERITY_RETRY,        // REQUOTE (10004)
        SEVERITY_UNDEFINED,
        SEVERITY_REJECT,       // REJECT (10006)
        SEVERITY_NORMAL,       // CANCEL (10007)
        SEVERITY_NORMAL,       // PLACED (10008)
        SEVERITY_NORMAL,       // DONE (10009)
        SEVERITY_NORMAL,       // DONE_PARTIAL (10010)
        SEVERITY_ERROR,        // ERROR (10011)
        SEVERITY_RETRY,        // TIMEOUT (10012)
        SEVERITY_INVALID,      // INVALID (10013)
        SEVERITY_INVALID,      // INVALID_VOLUME (10014)
        SEVERITY_INVALID,      // INVALID_PRICE (10015)
        SEVERITY_INVALID,      // INVALID_STOPS (10016)
        SEVERITY_PERMISSIONS,  // TRADE_DISABLED (10017)
        SEVERITY_TRY_LATER,    // MARKET_CLOSED (10018)
        SEVERITY_LIMITS,       // NO_MONEY (10019)
        ...
    };

    if(retcode == 0) return SEVERITY_NORMAL;
    if(retcode < 10000 || retcode > HEDGE_PROHIBITED) return SEVERITY_UNDEFINED;
    return severities[retcode - 10000];
}
```

借助这个功能，*OnTick* 处理程序可以补充“智能”错误处理。静态变量 *RetryFrequency* 存储了程序在出现非关键错误时尝试重复请求的频率。上次时间 (比如上次进行重试的时间) 存储在 *RetryRecordTime* 变量中。

```

void OnTick()
{
    ...
    const static int DEFAULT_RETRY_TIMEOUT = 1; // seconds
    static int RetryFrequency = DEFAULT_RETRY_TIMEOUT;
    static datetime RetryRecordTime = 0;
    if(TimeTradeServer() - RetryRecordTime < RetryFrequency) return;
    ...
}

```

一旦 *PlaceOrder* 或 *ModifyOrder* 函数返回 *retcode* 值,我们就知道其严重程度,并且根据严重程度,我们从三个选项选择一个:停止 EA 交易、等待超时或常规操作 (*lastDay* 的当天标记订单成功修改)。

```

const TRADE_RETCODE_SEVERITY severity = TradeCodeSeverity(retcode);
if(severity >= SEVERITY_INVALID)
{
    Alert("Can't place/modify pending order, EA is stopped");
    RetryFrequency = INT_MAX;
}
else if(severity >= SEVERITY_RETRY)
{
    RetryFrequency += (int)sqrt(RetryFrequency + 1);
    RetryRecordTime = TimeTradeServer();
    PrintFormat("Problems detected, waiting for better conditions "
        "(timeout enlarged to %d seconds)",
        RetryFrequency);
}
else
{
    if(RetryFrequency > DEFAULT_RETRY_TIMEOUT)
    {
        RetryFrequency = DEFAULT_RETRY_TIMEOUT;
        PrintFormat("Timeout restored to %d second", RetryFrequency);
    }
    lastDay = TimeTradeServer() / DAYLONG * DAYLONG;
}
}

```

在被分类为可解决重复问题的情况下, *RetryFrequency* 超时会随着每个后续错误逐渐增加,但是当请求被成功处理时重置为 1 秒。

应当注意,所应用的结构体 *MqlTradeRequestSync* 的方法进而检查大量参数组合的正确性,如果发现问题,会在 *SendRequest* 调用之前中断该过程。默认情况下启用此行为,但是可以通过在 *MqlTradeSync.mqh* 的 *#include* 指令之前定义一个空的 RETURN(X) 宏来禁用它。

```

#define RETURN(X)
#include <MQL5Book/MqlTradeSync.mqh>

```

使用这个宏定义,检查仅在日志中打印警告,但会继续执行方法,直到 *SendRequest* 调用。

在任何情况下,在调用 *MqlTradeResultSync* 结构体的一个或另一个方法后,错误代码将被添加到 *retcode* 中。这将由服务器或 *MqlTradeRequestSync* 结构体的检查算法来完成 (此处我们利用了 *MqlTradeResultSync* 实例包含在 *MqlTradeRequestSync* 中这一事实)。为了简洁起见,此处不提供对返回错误代码以及 *MqlTradeRequestSync* 方法中使用 RETURN 宏的说明。感兴趣的读者可以在 *MqlTradeSync.mqh* 文件中看到完整的源代码。

在启用可视化模式的情况下，我们在测试程序中运行 EA 交易 *PendingOrderModify.mq5*，并使用 XAUUSD, H1 的数据（所有分时报价或真实分时报价模式）。使用默认设置，EA 交易将以最小手数下达 ORDER_TYPE_BUY_STOP 类型的订单。我们从日志和交易历史可以确认，该程序在每天开始时下达挂单并进行修改。

```
2022.01.03 01:05:00 Autodetected daily range: 14.37
2022.01.03 01:05:00 buy stop 0.01 XAUUSD at 1845.73 sl: 1838.55 tp: 1852.91 (1830.63
2022.01.03 01:05:00 OK order placed: #=2
2022.01.03 01:05:00 TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_
» @ 1845.73, SL=1838.55, TP=1852.91, ORDER_TIME_GTC, M=1234567890
2022.01.03 01:05:00 DONE, #=2, V=0.01, Bid=1830.63, Ask=1831.36, Request executed
2022.01.04 01:05:00 Autodetected daily range: 33.5
2022.01.04 01:05:00 order modified [#2 buy stop 0.01 XAUUSD at 1836.56]
2022.01.04 01:05:00 OK order modified: #=2
2022.01.04 01:05:00 TRADE_ACTION_MODIFY, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_F
» @ 1836.56, SL=1819.81, TP=1853.31, ORDER_TIME_GTC, #=2
2022.01.04 01:05:00 DONE, #=2, @ 1836.56, Bid=1819.81, Ask=1853.31, Request executed,
2022.01.05 01:05:00 Autodetected daily range: 18.23
2022.01.05 01:05:00 order modified [#2 buy stop 0.01 XAUUSD at 1832.56]
2022.01.05 01:05:00 OK order modified: #=2
2022.01.05 01:05:00 TRADE_ACTION_MODIFY, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_F
» @ 1832.56, SL=1823.45, TP=1841.67, ORDER_TIME_GTC, #=2
2022.01.05 01:05:00 DONE, #=2, @ 1832.56, Bid=1823.45, Ask=1841.67, Request executed,
...
2022.01.11 01:05:00 Autodetected daily range: 11.96
2022.01.11 01:05:00 order modified [#2 buy stop 0.01 XAUUSD at 1812.91]
2022.01.11 01:05:00 OK order modified: #=2
2022.01.11 01:05:00 TRADE_ACTION_MODIFY, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_F
» @ 1812.91, SL=1806.93, TP=1818.89, ORDER_TIME_GTC, #=2
2022.01.11 01:05:00 DONE, #=2, @ 1812.91, Bid=1806.93, Ask=1818.89, Request executed,
2022.01.11 18:10:58 order [#2 buy stop 0.01 XAUUSD at 1812.91] triggered
2022.01.11 18:10:58 deal #2 buy 0.01 XAUUSD at 1812.91 done (based on order #2)
2022.01.11 18:10:58 deal performed [#2 buy 0.01 XAUUSD at 1812.91]
2022.01.11 18:10:58 order performed buy 0.01 at 1812.91 [#2 buy stop 0.01 XAUUSD at 1
2022.01.11 20:28:59 take profit triggered #2 buy 0.01 XAUUSD 1812.91 sl: 1806.93 tp:
» [#3 sell 0.01 XAUUSD at 1818.89]
2022.01.11 20:28:59 deal #3 sell 0.01 XAUUSD at 1818.91 done (based on order #3)
2022.01.11 20:28:59 deal performed [#3 sell 0.01 XAUUSD at 1818.91]
2022.01.11 20:28:59 order performed sell 0.01 at 1818.91 [#3 sell 0.01 XAUUSD at 1818
2022.01.12 01:05:00 Autodetected daily range: 23.28
2022.01.12 01:05:00 buy stop 0.01 XAUUSD at 1843.77 sl: 1832.14 tp: 1855.40 (1820.14
2022.01.12 01:05:00 OK order placed: #=4
2022.01.12 01:05:00 TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_
» @ 1843.77, SL=1832.14, TP=1855.40, ORDER_TIME_GTC, M=1234567890
2022.01.12 01:05:00 DONE, #=4, V=0.01, Bid=1820.14, Ask=1820.49, Request executed, Re
```

订单可以在任何时候触发，一段时间后通过止损或止盈进行平仓（如上面的代码所示）。

在某些情况下，可能会出现第二天开始时该仓位仍然存在的情况，然后除此之外还会创建一个新订单，如下图所示。



EA 交易根据测试程序中的挂单制定的交易策略

请注意，由于我们需要请求 PERIOD_D1 时间范围的报价以计算日线范围，因此除了当前的工作图表之外，可视化测试程序还会打开相应的图表。这种服务不仅适用于工作时间范围以外的时间范围，还适用于其他交易品种。这在开发 **多货币 EA 交易** 时特别有用。

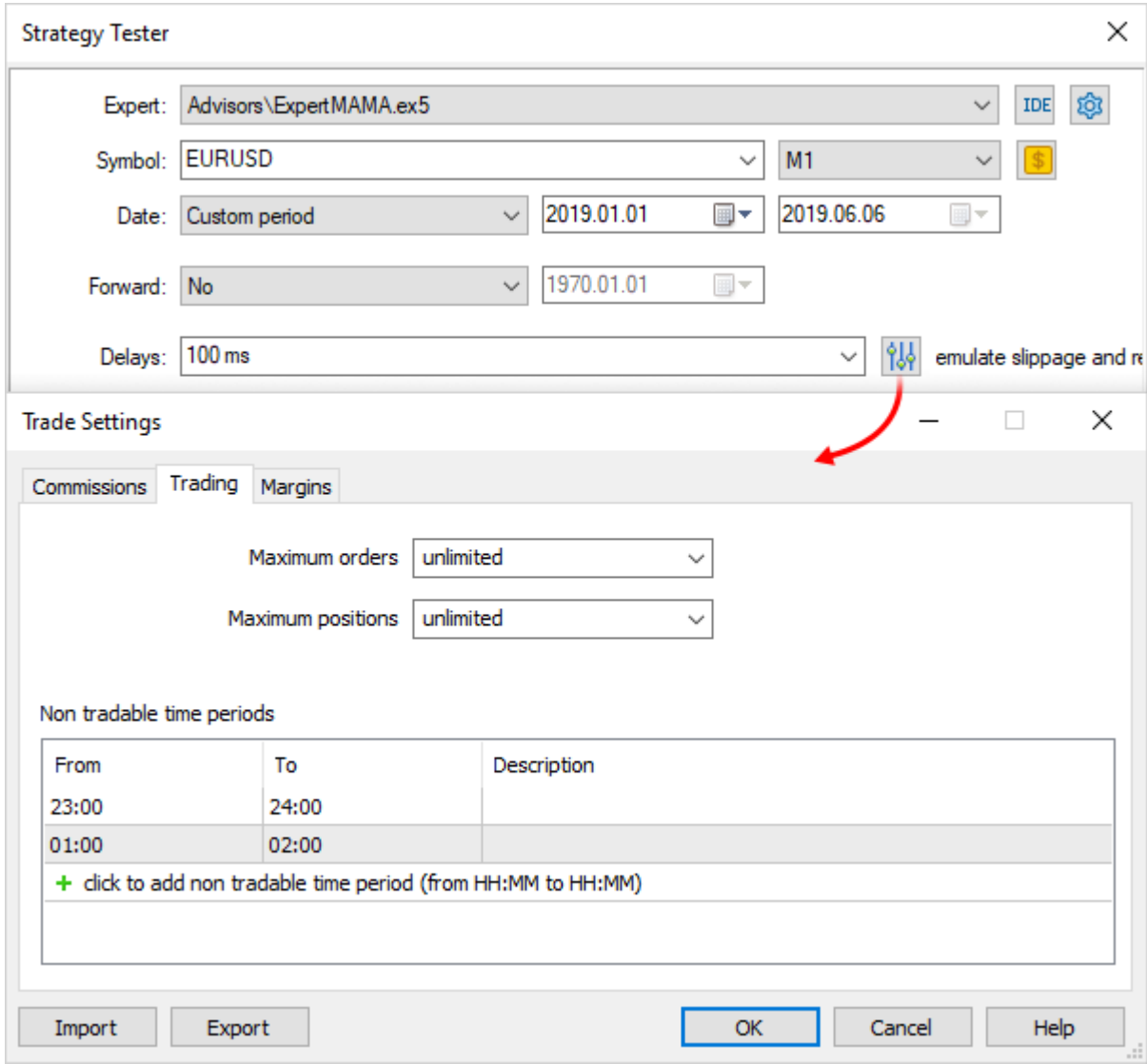
要检查错误处理是如何工作的，请尝试禁用 EA 交易类型的交易。日志将包含以下内容：

```
Autodetected daily range: 34.48
TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, »
» @ 1975.73, SL=1958.49, TP=1992.97, ORDER_TIME_GTC, M=1234567890
CLIENT_DISABLES_AT, AutoTrading disabled by client
Alert: Can't place/modify pending order, EA is stopped
```

此错误非常严重，EA 交易将停止工作。

为了演示一个更简单的错误，我们可以使用 **OnTimer** 处理程序来代替 **OnTick**。然后，若在交易时段仅占一天的一部分的交易品种上启动相同的 EA 交易，则将周期性地产生一系列关于关闭市场（“市场关闭”）的非关键错误。在这种情况下，EA 交易会不断尝试开始交易，不断增加等待时间。

特别是，很容易在测试程序中进行检查，它允许你为任何交易品种设置任意的交易时段。在 **Settings** 选项卡上 **Delays** 下拉菜单的右边，有一个按钮可打开 **Trade setup** 对话框。此处，应包括 **Use your settings** 选项，并在 **Trade** 选项卡上向 **Non-trading periods** 表中添加至少一条记录。



在测试程序中设置非交易时段

请注意，此处设置的是非交易时段，而非交易时段，即与交易品种规格相比，此设置的作用正好相反。

许多与交易限制相关的潜在错误基本上可以通过使用某个类进行环境分析来消除，比如 *Permissions* 类，之前在 [账户交易的限制和权限](#) 一节中介绍过。

6.4.21 删除挂单

使用 `TRADE_ACTION_REMOVE` 操作在程序水平执行挂单删除：应将该常量分配给 *action* 字段（属于 *MqlTradeRequest* 结构体，且需要在调用 *OrderSend* 函数的一个版本之前进行。除了 *action* 之外，唯一的必填字段是 *order*，用于指定要删除的订单号。

MqlTradeSync.mqh 文件中 *MqlTradeRequestSync* 应用程序结构体中的 *remove* 方法非常简单。

```

struct MqlTradeRequestSync: public MqlTradeRequest
{
    ...
    bool remove(const ulong ticket)
    {
        if(!OrderSelect(ticket)) return false;
        action = TRADE_ACTION_REMOVE;
        order = ticket;
        ZeroMemory(result);
        return OrderSend(this, result);
    }
}

```

订单删除的事实验证通常是在 *completed* 方法中完成的。

```

bool completed()
{
    ...
    else if(action == TRADE_ACTION_REMOVE)
    {
        result.order = order;
        return result.removed(timeout);
    }
    ...
}

```

等待订单的实际移除是在 *MqlTradeResultSync* 结构体的 *removed* 方法中执行的。

```

struct MqlTradeResultSync: public MqlTradeResult
{
    ...
    bool removed(const ulong msc = 1000)
    {
        if(retcode != TRADE_RETCODE_DONE)
        {
            return false;
        }

        if(!wait(orderRemoved, msc))
        {
            Print("Order removal timeout: #" + (string)order);
            return false;
        }

        return true;
    }

    static bool orderRemoved(MqlTradeResultSync &ref)
    {
        return !OrderSelect(ref.order) && HistoryOrderSelect(ref.order);
    }
}

```

EA 交易 (*PendingOrderDelete.mq5*) 的一个示例演示了我们将几乎完全基于 *PendingOrderSend.mq5* 构建的订单移除。这是因为在删除之前更便于保证订单存在。因此，在启

动后，EA 交易将立即使用指定的参数创建新订单。随后将在 *OnDeinit* 处理程序中删除订单。如果你更改 EA 交易的输入参数、交易品种或图表时间范围，旧订单也将被删除，并将创建一个新订单。

添加了 *OwnOrder* 全局变量，用于存储订单号。该变量是在调用 *PlaceOrder* 后生成的（函数本身不变）。

```
ulong OwnOrder = 0;

void OnTimer()
{
    // execute the code once for the current parameters
    EventKillTimer();

    const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
    OwnOrder = PlaceOrder((ENUM_ORDER_TYPE)Type, symbol, Volume,
        Distance2SLTP, Expiration, Until, Magic, Comment);
}
```

以下是一个简单的删除函数 *RemoveOrder*，其会创建 *request* 对象并依次调用 *remove* 和 *completed* 方法。

```
void OnDeinit(const int)
{
    if(OwnOrder != 0)
    {
        RemoveOrder(OwnOrder);
    }
}

void RemoveOrder(const ulong ticket)
{
    MqlTradeRequestSync request;
    if(request.remove(ticket) && request.completed())
    {
        Print("OK order removed");
    }
    Print(TU::StringOf(request));
    Print(TU::StringOf(request.result));
}
```

以下日志显示了将 EA 交易放置在 EURUSD 图表上的结果，在此之后，交易品种被切换为 XAUUSD，然后 EA 交易被删除。


```

(EURUSD,H1) Autodetected daily range: 0.0094
(EURUSD,H1) OK order placed: #=1284920879
(EURUSD,H1) TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLIN
» @ 1.11011, ORDER_TIME_GTC, M=1234567890
(EURUSD,H1) DONE, #=1284920879, V=0.01, Request executed, Req=1
(EURUSD,H1) OK order removed
(EURUSD,H1) TRADE_ACTION_REMOVE, EURUSD, ORDER_TYPE_BUY, ORDER_FILLING_FOK, #=12849
(EURUSD,H1) DONE, #=1284920879, Request executed, Req=2
(XAUUSD,H1) Autodetected daily range: 47.45
(XAUUSD,H1) OK order placed: #=1284921672
(XAUUSD,H1) TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLIN
» @ 1956.68, ORDER_TIME_GTC, M=1234567890
(XAUUSD,H1) DONE, #=1284921672, V=0.01, Request executed, Req=3
(XAUUSD,H1) OK order removed
(XAUUSD,H1) TRADE_ACTION_REMOVE, XAUUSD, ORDER_TYPE_BUY, ORDER_FILLING_FOK, #=12849
(XAUUSD,H1) DONE, #=1284921672, Request executed, Req=4

```

我们将在 [OnTrade](#) 事件一节中了解另一个删除订单以实现“互损订单”(OCO)策略的示例。

6.4.22 获取活跃订单列表

EA 交易程序通常需要列举现有的活跃订单并分析其特性。具体来说，在 [挂单修改](#) 一节中的 *PendingOrderModify.mq5* 示例中，我们创建了一个特殊函数 *GetMyOrder* 来查找属于该 EA 交易的订单，以便进行修改。此处，通过交易品种名称和 EA 交易 ID (*Magic*) 进行分析。理论上，其方法与上一节删除挂单 *PendingOrderDelete.mq5* 的示例中使用的方法相同。

在后一种情况下，为了简单起见，我们创建了一个订单并将其订单号存储在一个全局变量中。但这在一般情况下是做不到的，因为 EA 交易和整个终端可以随时停止或重启。因此，EA 交易必须包含用于恢复内部状态的算法，包括对整个交易环境的分析以及订单、交易、仓位、账户余额等。

在本节中，我们将研究 *OrdersTotal* 函数，该函数用于获取活跃订单列表，并在交易环境中选择其中任何一个订单，这样便可以读取其所有特性。

`int OrdersTotal()`

OrdersTotal 函数可返回当前活跃订单的数量。其中包括挂单以及尚未执行的市场订单。通常情况下，市场订单会被迅速执行，因此通常不可能在活动阶段捕捉到，但如果市场流动性不足，就会发生这种情况。一旦订单被执行（交易完成），它就会从活跃订单类别转移到历史中。我们将在单独的章节讨论如何使用订单历史。

请注意，只有订单可以分为活跃的和历史。这将订单与总是在历史中创建的交易以及仅在线存在的仓位显著区分开来。要恢复仓位的历史，应分析 [交易的历史](#)。

`ulong OrderGetTicket(uint index)`

OrderGetTicket 函数可根据终端交易环境中订单列表中的订单编号返回订单号。*index* 参数必须介于 0 到 *OrdersTotal()-1* 的值之间。订单的组织方式没有规定。

OrderGetTicket 函数可选择一个订单，也就是说，将其有关的数据复制到某个内部缓存中，以便 MQL 程序可以使用 *OrderGetDouble*、*OrderGetInteger* 或 *OrderGetString* 函数的后续调用来读取其所有特性，这将在 [单独的章节](#)中讨论。

存在这种缓存即表明从其接收的数据可能已过时：订单可能不再存在或可能已经被修改（例如，它可能具有不同的状态、开盘价、*Stop Loss* 或 *Take Profit* 水平以及到期）。因此，为了保证

收到订单的相关数据，建议在请求数据之前立即调用 `OrderGetTicket` 函数。此处是在 `PendingOrderModify.mq5` 示例中的实现方法。

```
ulong GetMyOrder(const string name, const ulong magic)
{
    for(int i = 0; i < OrdersTotal(); ++i)
    {
        ulong t = OrderGetTicket(i);
        if(OrderGetInteger(ORDER_MAGIC) == magic
            && OrderGetString(ORDER_SYMBOL) == name)
        {
            return t;
        }
    }
    return 0;
}
```

每个 MQL 程序都有自己的缓存（交易环境上下文），其中包括选定的订单。在以下章节中，我们将了解到，除了订单之外，MQL 程序还可以将交易和订单的仓位和历史片段选择到活动上下文中。

`OrderSelect` 函数可执行类似的订单选择，并将其数据复制到内部缓存中。

`bool OrderSelect(ulong ticket)`

该函数可检查订单是否存在，并为进一步读取其特性做好准备。在这种情况下，订单不是由序列号指定的，而是由 MQL 程序必须以某种方式提前收到的订单号指定的，特别是作为执行 `OrderSend/OrderSendAsync` 的结果。

如果成功，则该函数返回 `true`。如果收到 `false`，通常意味着没有指定订单号的订单。最常见的原因是当订单状态从活动变为历史时，例如，作为执行或取消的结果（我们将在后面学习如何确定确切的状态）。可以使用 [相关函数](#) 在历史中选择订单。

以前，我们使用 `MqlTradeResultSync` 结构体中的 `OrderSelect` 函数来跟踪挂单的 [创建](#) 和 [移除](#)。

6.4.23 订单特性（活跃和历史）

在与交易操作相关的章节中，特别是与 [买入/卖出](#)、[平仓](#) 和 [下达挂单](#) 相关的章节中，我们已经看到请求是根据 `MqlTradeRequest` 结构体中特定字段的填充发送到服务器的，其中大多数字段直接定义了生成订单的特性。MQL5 API 允许你了解这些特性以及由交易系统本身设置的一些其他特性，比如订单号、注册时间和状态。

值得注意的是，订单特性列表对于活跃订单和历史订单是通用的，当然，许多特性的值会有所不同。

在 MQL5 中，订单特性根据我们已经熟悉的基于值类型的原则进行了分组：整数型（与 `long/ulong` 兼容）、实数型（`double`）和字符串型。每种特性组都有自己的枚举。

整数型特性在 `ENUM_ORDER_PROPERTY_INTEGER` 中汇总，并在下表中列出。

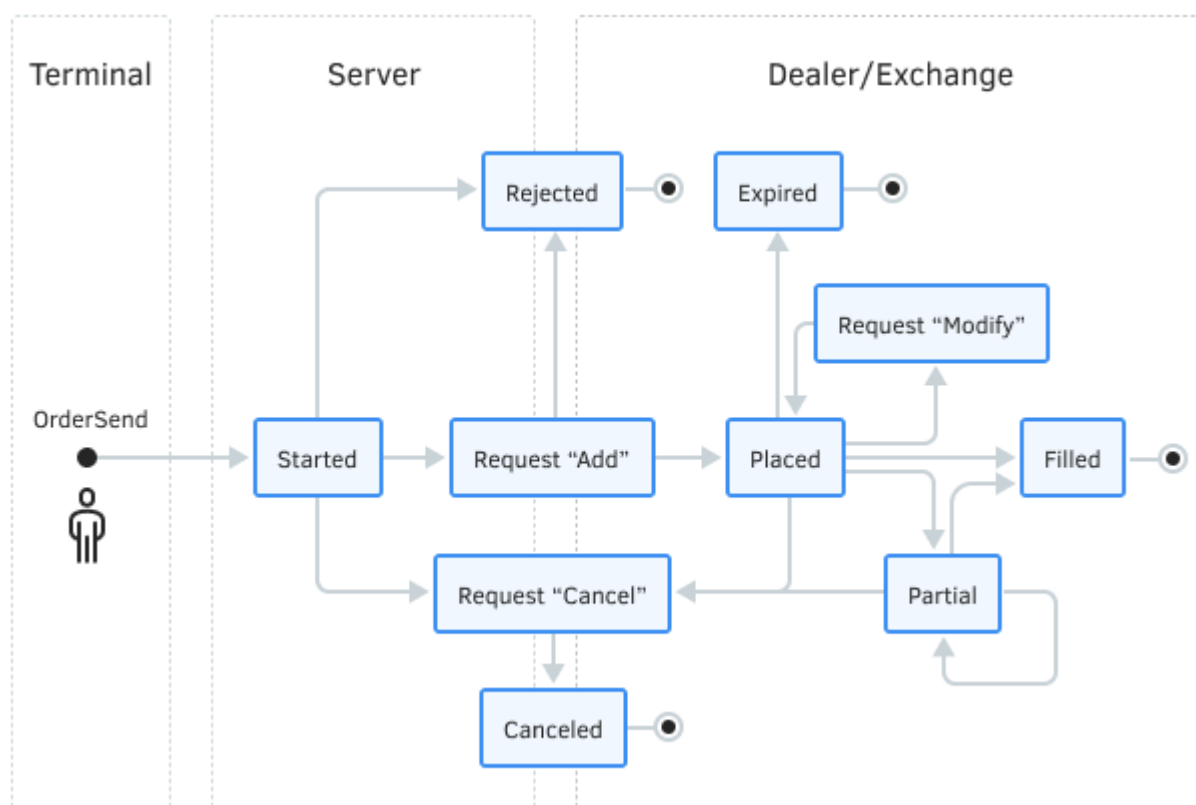
标识符	说明	类型
ORDER_TYPE	订单类型	ENUM_ORDER_TYPE

标识符	说明	类型
ORDER_TYPE_FILLING	按交易量分类的订单类型	ENUM_ORDER_TYPE_FILLING
ORDER_TYPE_TIME	订单生命周期 (挂单)	ENUM_ORDER_TYPE_TIME
ORDER_TIME_EXPIRATION	订单到期时间 (挂单)	datetime
ORDER_MAGIC	由下订单的 EA 交易设置的任意标识符	ulong
ORDER_TICKET	订单号; 服务器分配给每个订单的唯一编号	ulong
ORDER_STATE	订单状态	ENUM_ORDER_STATE (见下文)
ORDER_REASON	订单的原因或来源	ENUM_ORDER_REASON (见下文)
ORDER_TIME_SETUP	订单下达时间	datetime
ORDER_TIME_DONE	订单执行或撤销时间	datetime
ORDER_TIME_SETUP_MSC	下单执行时间 (毫秒)	ulong
ORDER_TIME_DONE_MSC	订单执行/撤回时间 (毫秒)	ulong
ORDER_POSITION_ID	订单执行时生成或执行修改的仓位 ID	ulong
ORDER_POSITION_BY_ID	ORDER_TYPE_CLOSE_BY 类型 订单的对应仓位标识符	ulong

每一笔执行的订单都会生成一笔交易, 用于开立新仓位或改变现有仓位。在 ORDER_POSITION_ID 特性中, 该仓位的 ID 被分配给已执行的订单。

ENUM_ORDER_STATE 枚举包含说明订单状态的元素。参见下面订单的简化方案 (状态图)。

标识符	说明
ORDER_STATE_STARTED	订单已经过正确性检查，但尚未被服务器接受
ORDER_STATE_PLACED	订单已被服务器接受
ORDER_STATE_CANCELED	订单已被客户（用户或 MQL 程序）取消
ORDER_STATE_PARTIAL	订单已部分执行
ORDER_STATE_FILLED	订单已全部履行
ORDER_STATE_REJECTED	订单已被服务器拒绝
ORDER_STATE_EXPIRED	订单已在到期时取消
ORDER_STATE_REQUEST_ADD	订单正在注册（正在交易系统中下单）
ORDER_STATE_REQUEST_MODIFY	订单正在被修改（其参数被更改）
ORDER_STATE_REQUEST_CANCEL	订单正在被删除（从交易系统中删除）



订单状态图

只有活跃订单才可以更改状态。对于历史订单 (已完成或已取消), 状态是固定的。

你可以取消已经部分履行的订单，然后其在历史中的状态将变为 ORDER_STATE_CANCELED。

ORDER_STATE_PARTIAL 仅适用于活跃订单。已执行 (历史) 订单的状态始终为 ORDER_STATE_FILLED。

ENUM_ORDER_REASON 枚举指定了可能的订单来源选项。

标识符	说明
ORDER_REASON_CLIENT	从桌面终端手动下单
ORDER_REASON_EXPERT	由 EA 交易或脚本从桌面终端下单
ORDER_REASON_MOBILE	从移动应用程序下单
ORDER_REASON_WEB	从 WEB 终端 (浏览器) 下单
ORDER_REASON_SL	作为触发止损的结果由服务器下单
ORDER_REASON_TP	作为触发止盈的结果由服务器下单
ORDER_REASON_SO	作为停止事件的结果由服务器下单

真实特性收集在 ENUM_ORDER_PROPERTY_DOUBLE 枚举中。

标识符	说明
ORDER_VOLUME_INITIAL	下单时的初始交易量
ORDER_VOLUME_CURRENT	当前交易量 (初始的或部分执行后剩余的交易量)
ORDER_PRICE_OPEN	订单中显示的价格
ORDER_PRICE_CURRENT	尚未执行订单的当前交易品种价格或执行价格
ORDER_SL	止损水平
ORDER_TP	止盈水平
ORDER_PRICE_STOPLIMIT	触发 StopLimit 订单时, 下达限价订单的价格

ORDER_PRICE_CURRENT 特性包含活动买入挂单的当前 **Ask** 价格或活动卖出挂单的 **Bid** 价格。“当前价格”是指使用 **OrderSelect** 或 **OrderGetTicket** 选择订单时交易环境中的已知价格。对于历史中已执行的订单, 此特性包含执行价格, 由于滑点, 该价格可能与订单中指定的价格不同。

仅当订单状态为 ORDER_STATE_PARTIAL 时, ORDER_VOLUME_INITIAL 和 ORDER_VOLUME_CURRENT 特性才不相等。

如果订单是部分完成的, 那么它在历史中的 ORDER_VOLUME_INITIAL 特性将等于最后完成的部分的大小, 并且与原始完整交易量相关的所有其他“完成”将作为单独的订单 (和交易) 来执行。

在 ENUM_ORDER_PROPERTY_STRING 枚举中说明了字符串特性。

标识符	说明
ORDER_SYMBOL	下单的交易品种
ORDER_COMMENT	注释
ORDER_EXTERNAL_ID	外部交易系统（交易所）中的订单 ID

要读取上述所有特性，有两组不同的函数：分别用于活跃订单和历史订单。首先，我们将考虑活跃订单的函数，我们将返回到历史订单（在我们熟悉选择 *历史* 中所需时期的原则后）。

6.4.24 用于读取活动订单特性的函数

对于活动订单和历史订单，可用于获取所有订单特性值的函数集并不相同。本节介绍读取活动订单特性的函数。有关访问历史订单特性的函数，请参见 *相关章节*。

可以使用 *OrderGetInteger* 函数读取整数型特性，该函数有两种形式：第一种直接返回特性值，第二种返回成功 (*true*) 或错误 (*false*) 的逻辑交易品种，第二种通过引用传递的参数用特性值填充。

```
long OrderGetInteger(ENUM_ORDER_PROPERTY_INTEGER property)
bool OrderGetInteger(ENUM_ORDER_PROPERTY_INTEGER property, long &value)
```

这两个函数都允许你获取请求的整数兼容类型的订单特性 (*datetime*、*long/ulong* 或 *listing*)。虽然原型提到了 *long*，但从技术角度来看，该值存储为一个 8 字节的单元，可以将其强制转换为兼容的类型，而无需对内部表示进行任何转换，特别是转换为 *ulong*，用于所有订单号。

一对类似的函数用于实数型 *double* 的特性。

```
double OrderGetDouble(ENUM_ORDER_PROPERTY_DOUBLE property)
bool OrderGetDouble(ENUM_ORDER_PROPERTY_DOUBLE property, double &value)
```

最后，字符串型特性可通过一对 *OrderGetString* 函数获得。

```
string OrderGetString(ENUM_ORDER_PROPERTY_STRING property)
bool OrderGetString(ENUM_ORDER_PROPERTY_STRING property, string &value)
```

所有函数的第一个参数都是我们感兴趣的特性标识符。这必须是下列其中一个枚举元素：ENUM_ORDER_PROPERTY_INTEGER、ENUM_ORDER_PROPERTY_DOUBLE 或 ENUM_ORDER_PROPERTY_STRING（在 *上一节* 中讨论过）。

请注意，在调用任何上述函数之前，应首先使用 *OrderSelect* 或 *OrderGetTicket* 选择一个订单。

为了读取特定订单的所有特性，我们将开发 *OrderMonitor* 类(*OrderMonitor.mqh*)，其工作原理与之前考虑的交易品种 (*SymbolMonitor.mqh*) 和交易账户 (*AccountMonitor.mqh*) 监视器相同。

本书中讨论的这些类和其他 *monitor* 类提供了一种统一的方法，通过重载版本的虚拟 *get* 方法来分析特性。

回顾一下，假设交易和仓位根据三种主要类型的值具有相同的特性分组，并且我们还需要为其实现监视器。在这方面，将通用算法将其分离到一个基础抽象类 *MonitorInterface* (*TradeBaseMonitor.mqh*) 中是有意义的。这是一个具有三个参数的模板类，用于为整数型 (I)、实数型 (D) 和字符串型 (S) 特性组指定特定枚举的类型。

```

#include <MQL5Book/EnumToArray.mqh>

template<typename I,typename D,typename S>
class MonitorInterface
{
protected:
    bool ready;
public:
    MonitorInterface(): ready(false) { }

    bool isReady() const
    {
        return ready;
    }
    ...

```

由于在交易环境中寻找订单（交易或仓位）可能会因各种原因而失败，该类有一个保留变量 **ready**，其中派生类必须写入成功初始化的标志，即选择一个对象来读取其特性。

几个纯虚方法声明对相应类型特性的访问。

```

virtual long get(const I property) const = 0;
virtual double get(const D property) const = 0;
virtual string get(const S property) const = 0;
virtual long get(const int property, const long) const = 0;
virtual double get(const int property, const double) const = 0;
virtual string get(const int property, const string) const = 0;
...

```

在前三种方法中，特性类型由模板参数之一指定。在另外三种方法中，类型由方法本身的第二个参数指定：这是必需的，因为最新方法不接受特定枚举的常量，而只接受一个整数作为第一个参数。一方面，这便于标识符的连续编号（三种类型的枚举常量不相交）。另一方面，我们需要另一个来源来确定值类型，因为函数/方法返回的类型不参与选择适当 [重载](#) 的过程。

这种方法允许你基于调用代码中可用的各种输入来获取特性。接下来，我们将基于 **OrderMonitor**（以及期货的 **DealMonitor** 和 **PositionMonitor**）创建类，以根据一组任意条件选择对象，所有这些方法都将是必需的。

通常，程序需要获得任何特性的字符串表示，例如，用于日志记录。在新的监视器中，这是通过 **stringify** 方法实现的。显然，它们可通过上面提到的 **get** 方法调用来获取所请求特性的值。

```

virtual string stringify(const long v, const I property) const = 0;

virtual string stringify(const I property) const
{
    return stringify(get(property), property);
}

virtual string stringify(const D property, const string format = NULL) const
{
    if(format == NULL) return (string)get(property);
    return StringFormat(format, get(property));
}

virtual string stringify(const S property) const
{
    return get(property);
}
...

```

唯一没有实现的方法是 *long* 类型的 *stringify* 的第一个版本。这是因为，正如我们在上一节中看到的，整数型特性组实际上包含不同的应用程序类型，包括日期和时间、枚举和整数。因此，只有派生类可以提供其到可理解字符串的转换。这种情况对于所有交易实体都是常见的，不仅是订单，还有交易和仓位，我们将在后面考虑其特性。

当整数型特性包含枚举元素（例如 `ENUM_ORDER_TYPE`、`ORDER_TYPE_FILLING` 等）时，应使用 *EnumToString* 函数将其转换为字符串。这个任务由一个辅助方法 *enumstr* 来完成。很快我们将看到该方法在特定监视器类中的广泛使用，在几个段落之后从 *OrderMonitor* 开始。

```

template<typename E>
static string enumstr(const long v)
{
    return EnumToString((E)v);
}

```

为了记录特定类型的所有特性，我们创建了 *list2log* 方法，该方法在循环中使用 *stringify*。

```

template<typename E>
void list2log() const
{
    E e = (E)0; // suppress warning 'possible use of uninitialized variable'
    int array[];
    const int n = EnumToArray(e, array, 0, USHORT_MAX);
    Print(typename(E), " Count=", n);
    for(int i = 0; i < n; ++i)
    {
        e = (E)array[i];
        PrintFormat("% 3d %s=%s", i, EnumToString(e), stringify(e));
    }
}

```

最后，为了更容易地记录所有三个组的特性，有一个 *print* 方法，它为每组特性调用 *list2log* 三次。


```

virtual void print() const
{
    if(!ready) return;

    Print(typename(this));
    list2log<I>();
    list2log<D>();
    list2log<S>();
}

```

利用一个基础模板类 *MonitorInterface*, 我们可以说明 *OrderMonitorInterface*, 其中为上一节中的订单指定了某些枚举类型, 并为订单的整数型特性提供了 *stringify* 的实现。

```

class OrderMonitorInterface:
public MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,
    ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>
{
public:
    // description of properties according to subtypes
    virtual string stringify(const long v,
        const ENUM_ORDER_PROPERTY_INTEGER property) const override
    {
        switch(property)
        {
            case ORDER_TYPE:
                return enumstr<ENUM_ORDER_TYPE>(v);
            case ORDER_STATE:
                return enumstr<ENUM_ORDER_STATE>(v);
            case ORDER_TYPE_FILLING:
                return enumstr<ENUM_ORDER_TYPE_FILLING>(v);
            case ORDER_TYPE_TIME:
                return enumstr<ENUM_ORDER_TYPE_TIME>(v);
            case ORDER_REASON:
                return enumstr<ENUM_ORDER_REASON>(v);

            case ORDER_TIME_SETUP:
            case ORDER_TIME_EXPIRATION:
            case ORDER_TIME_DONE:
                return TimeToString(v, TIME_DATE | TIME_SECONDS);

            case ORDER_TIME_SETUP_MSC:
            case ORDER_TIME_DONE_MSC:
                return STR_TIME_MSC(v);
        }

        return (string)v;
    }
};

```

以毫秒为单位显示时间的 STR_TIME_MSC 宏定义如下:

```
#define STR_TIME_MSC(T) (TimeToString((T) / 1000, TIME_DATE | TIME_SECONDS) \
    + StringFormat("%03d", (T) % 1000))
```

现在我们准备介绍用于读取任何订单特性的最终类: *OrderMonitor*, 派生自 *OrderMonitorInterface*。订单号被传递给构造函数, 并在交易环境中使用 *OrderSelect* 进行选择。

```
class OrderMonitor: public OrderMonitorInterface
{
public:
    const ulong ticket;
    OrderMonitor(const long t): ticket(t)
    {
        if(!OrderSelect(ticket))
        {
            PrintFormat("Error: OrderSelect(%lld) failed: %s",
                ticket, E2S(_LastError));
        }
        else
        {
            ready = true;
        }
    }
    ...
}
```

监视器的主要工作部分包括读取特性的虚函数的重新定义。此处我们看到了 *OrderGetInteger*、*OrderGetDouble* 和 *OrderGetString* 函数的调用。

```

virtual long get(const ENUM_ORDER_PROPERTY_INTEGER property) const override
{
    return OrderGetInteger(property);
}

virtual double get(const ENUM_ORDER_PROPERTY_DOUBLE property) const override
{
    return OrderGetDouble(property);
}

virtual string get(const ENUM_ORDER_PROPERTY_STRING property) const override
{
    return OrderGetString(property);
}

virtual long get(const int property, const long) const override
{
    return OrderGetInteger((ENUM_ORDER_PROPERTY_INTEGER)property);
}

virtual double get(const int property, const double) const override
{
    return OrderGetDouble((ENUM_ORDER_PROPERTY_DOUBLE)property);
}

virtual string get(const int property, const string) const override
{
    return OrderGetString((ENUM_ORDER_PROPERTY_STRING)property);
}
};

```

该代码片段以简短的形式呈现：用于处理历史中订单的运算符已删除。当我们在接下来的章节中探讨这个方面时，我们将看到 *OrderMonitor* 的完整代码。

值得注意的是，监视器对象不存储其特性的副本。因此，对 *get* 方法的访问必须在创建对象以及相应的调用 *OrderSelect* 之后立即执行。要在以后读取这些特性，需要在 MQL 程序的内部缓存中再次分配该订单，例如，通过调用方法 *refresh*。

```

void refresh()
{
    ready = OrderSelect(ticket);
}

```

我们通过将 *OrderMonitor* 添加到 EA 交易 *MarketOrderSend.mq5* 来测试其运行情况。一个名为 *MarketOrderSendMonitor.mq5* 的新版本通过指令 *#include* 连接文件 *OrderMonitor.mqh*，并在 *OnTimer* 函数体（在订单上成功确认开仓的块）中创建一个监视器对象并调用其 *print* 方法。

```

#include <MQL5Book/OrderMonitor.mqh>
...
void OnTimer()
{
    ...
    const ulong order = (wantToBuy ?
        request.buy(volume, Price) :
        request.sell(volume, Price));
    if(order != 0)
    {
        Print("OK Order: #=", order);
        if(request.completed())
        {
            Print("OK Position: P=", request.result.position);

            OrderMonitor m(order);
            m.print();
            ...
        }
    }
}

```

在日志中，我们可看到包含订单所有特性的新行。

```

OK Order: #=1287846602
Waiting for position for deal D=1270417032
OK Position: P=1287846602
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER, »
    » ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>
ENUM_ORDER_PROPERTY_INTEGER Count=14
    0 ORDER_TIME_SETUP=2022.03.21 13:28:59
    1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
    2 ORDER_TIME_DONE=2022.03.21 13:28:59
    3 ORDER_TYPE=ORDER_TYPE_BUY
    4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
    5 ORDER_TYPE_TIME=ORDER_TIME_GTC
    6 ORDER_STATE=ORDER_STATE_FILLED
    7 ORDER_MAGIC=1234567890
    8 ORDER_POSITION_ID=1287846602
    9 ORDER_TIME_SETUP_MSC=2022.03.21 13:28:59'572
   10 ORDER_TIME_DONE_MSC=2022.03.21 13:28:59'572
   11 ORDER_POSITION_BY_ID=0
   12 ORDER_TICKET=1287846602
   13 ORDER_REASON=ORDER_REASON_EXPERT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
    0 ORDER_VOLUME_INITIAL=0.01
    1 ORDER_VOLUME_CURRENT=0.0
    2 ORDER_PRICE_OPEN=1.10275
    3 ORDER_PRICE_CURRENT=1.10275
    4 ORDER_PRICE_STOPLIMIT=0.0
    5 ORDER_SL=0.0
    6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
    0 ORDER_SYMBOL=EURUSD
    1 ORDER_COMMENT=
    2 ORDER_EXTERNAL_ID=
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, »
    » @ 1.10275, P=1287846602, M=1234567890
DONE, D=1270417032, #=1287846602, V=0.01, @ 1.10275, Bid=1.10275, Ask=1.10275, »
    » Request executed, Req=3

```

第四行开始于 `print` 方法的输出,其中包括监视器对象 *MonitorInterface* 的全名以及参数类型(在本例中是三重 `ENUM_ORDER_PROPERTY`),然后是特定订单的所有特性。

但是,特性打印并不是监视器所能提供的最有趣操作。根据条件(任意特性值)选择订单的任务在 EA 交易中更受欢迎。使用监视器作为辅助工具,我们可创建一个筛选订单的机制,类似于我们对交易品种 *SymbolFilter.mqh* 所创建的机制。

6.4.25 按特性选择订单

在关于 *交易品种特性* 一节中,我们引入了 *SymbolFilter* 类来选择具有特定特性的金融工具。现在,我们将对订单应用相同的方法。

由于我们不仅要分析订单,还要以类似的方式分析交易和仓位,我们将把筛选算法的一般部分分离到基础类 *TradeFilter* (*TradeFilter.mqh*) 中。该部分几乎完全重复了 *SymbolFilter* 的源代码。所以此处不再解释。

如果愿意的话，可以对 *SymbolFilter.mqh* 和 *TradeFilter.mqh* 进行文件级上下文比较，看看它们相似程度，并对微小的编辑进行本地化。

主要区别在于 *TradeFilter* 类是一个模板，因为它必须处理不同对象的特性：订单、交易和仓位。

```

enum IS // supported comparison conditions in filters
{
    EQUAL,
    GREATER,
    NOT_EQUAL,
    LESS
};

enum ENUM_ANY // dummy enum to cast all enums to it
{
};

template<typename T,typename I,typename D,typename S>
class TradeFilter
{
protected:
    MapArray<ENUM_ANY,long> longs;
    MapArray<ENUM_ANY,double> doubles;
    MapArray<ENUM_ANY,string> strings;
    MapArray<ENUM_ANY,IS> conditions;
    ...

    template<typename V>
    static bool equal(const V v1, const V v2);

    template<typename V>
    static bool greater(const V v1, const V v2);

    template<typename V>
    bool match(const T &m, const MapArray<ENUM_ANY,V> &data) const;

public:
    // methods for adding conditions to the filter
    TradeFilter *let(const I property, const long value, const IS cmp = EQUAL);
    TradeFilter *let(const D property, const double value, const IS cmp = EQUAL);
    TradeFilter *let(const S property, const string value, const IS cmp = EQUAL);
    // methods for getting into arrays of records matching the filter
    template<typename E,typename V>
    bool select(const E property, ulong &tickets[], V &data[],
        const bool sort = false) const;
    template<typename E,typename V>
    bool select(const E &property[], ulong &tickets[], V &data[][],
        const bool sort = false) const;
    bool select(ulong &tickets[]) const;
    ...
}

```

模板参数 I、D、S 为三种主要类型（整数型、实数型、字符串型）特性组的枚举：对于订单，在前面的小节中已经说明过了，所以为了清晰起见，可以想象 I = ENUM_ORDER_PROPERTY_INTEGER, D = ENUM_ORDER_PROPERTY_DOUBLE, S = ENUM_ORDER_PROPERTY_STRING。

T 类型是为指定监视器类而设计的。目前，我们只有一个准备好的监视器 *OrderMonitor*。稍后我们将实现 *DealMonitor* 和 *PositionMonitor*。

之前，在 *SymbolFilter* 类中，我们没有使用模板参数，因为对于交易品种，所有类型的特性枚举都是已知的，并且只有一个类 *SymbolMonitor*。

回想一下筛选器类的结构体。一组 *let* 方法允许你在筛选器中注册 “property=value” 对的组合，该组合将用于在 *select* 方法中选择对象。ID 特性在 *property* 参数中指定，值在 *value* 参数中指定。

还有几种 *select* 方法。这些方法允许调用代码用选定订单号填充数组，如果需要，还可以用请求的对象特性值填充其他数组。所请求特性的特定标识符在 *select* 方法的第一个参数中设置；可以是一个或几个特性。因此，接收数组必须是一维的或二维的。

特性和值的组合不仅可以检查相等性 (EQUAL)，还可以检查大于/小于运算 (GREATER/LESS)。对于字符串型特性，可以指定一个搜索模式，用字符 “*” 表示任何字符序列（例如，ORDER_COMMENT 特性的 “[tp]*” 将匹配任何地方出现 “[tp]” 的所有注释，尽管这只是可能性的演示，而要搜索由触发 *Take Profit* 产生的订单，就应分析 ORDER_REASON）。

由于该算法要求实现一个循环，尽管所有对象和对象可以是不同类型（迄今为止，这些都是订单，但随后将出现对交易和仓位的支持），我们需要说明 *TradeFilter* 类中的两个抽象方法：*total* 和 *get*：

```
virtual int total() const = 0;
virtual ulong get(const int i) const = 0;
```

第一个方法返回对象的数量，第二个方法按其编号返回订单号。这样可以让你想起一对函数 *OrdersTotal* 和 *OrderGetTicket*。实际上，它们被用于筛选订单方法的特定实现中。

以下是 *OrderFilter* 类 (*OrderFilter.mqh*) 的完整版。

```
#include <MQL5Book/OrderMonitor.mqh>
#include <MQL5Book/TradeFilter.mqh>

class OrderFilter: public TradeFilter<OrderMonitor,
    ENUM_ORDER_PROPERTY_INTEGER,
    ENUM_ORDER_PROPERTY_DOUBLE,
    ENUM_ORDER_PROPERTY_STRING>
{
protected:
    virtual int total() const override
    {
        return OrdersTotal();
    }
    virtual ulong get(const int i) const override
    {
        return OrderGetTicket(i);
    }
};
```

这种简单性尤其重要，因为可以轻松地为交易和仓位创建类似的筛选器。

借助这个新类，我们可以更容易地检查属于我们的 EA 交易订单的存在，即替换 *GetMyOrder* 函数的任何自己编写版本（在示例 *PendingOrderModify.mq5* 中使用）。


```

OrderFilter filter;
ulong tickets[];

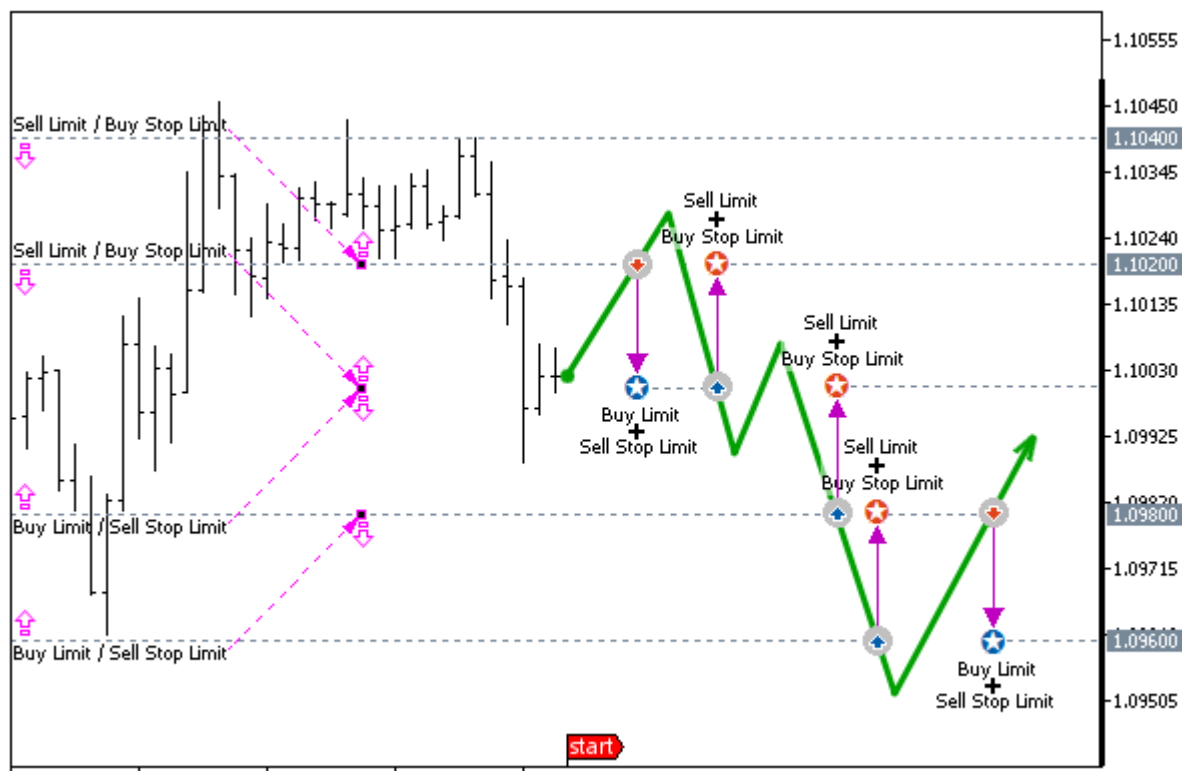
// set a condition for orders for the current symbol and our "magic" number
filter.let(ORDER_SYMBOL, _Symbol).let(ORDER_MAGIC, Magic);
// select suitable tickets in an array
if(filter.select(tickets))
{
    ArrayPrint(tickets);
}

```

此处的“任何版本”是指，由于有了 `filter` 类，我们可以创建任意条件来选择订单并“随时”更改它们（例如，根据用户的命令，而不是根据程序员的命令）。

作为如何使用筛选器的一个示例，我们使用一个 EA 交易，为从某个价格范围内水平反弹时的交易创建一个挂单净额结算网格，也就是说，针对波动的市场而设计。从这一节开始，在接下来的几节中，我们将在所研究材料的上下文中修改 EA 交易。

EA 交易的第一个版本 *PendingOrderGrid1.mq5* 根据限价订单和止损订单构建了一个给定大小的净额结算网格。这些参数将是价格水平的数量及其之间的步长（点数）。运算方案如下图所示。



步长为 200 点的 4 个水平的挂单网格

在某个初始时间（该时间可以由一天内的时间表确定，并且可以对应于“夜盘”），当前价格被向上舍入到净额结算网格步长的大小，并且从该水平上下设置指定数量的水平。

在每一个较高的水平，我们可下达限价卖出订单和 *stoplimit* 买入订单（期货限价订单的价格要低一个水平）。在每一个较低的水平，我们可下达限制买入订单和 *stoplimit* 卖出订单（期货限价订单的价格要高一个水平）。

当价格触及其中一个水平时，该水平的限价订单就变成了买入或卖出（仓位）。同时，同一水平的止损限价订单被系统自动转换为下一水平的反向限价订单。

例如，如果价格在向上移动的同时突破了该水平，我们将获得一个空头仓位，并在其下方的一步距离处创建一个买入限价订单。

EA 交易将监控每个水平是否有与限价订单配对的止损限价订单。因此，在检测到新的限价买入订单后，程序将在同一水平为其添加止损限价卖出订单，期货限价订单的目标价格为顶部的下一个水平，即开仓水平。

假设价格下跌，并激活限价订单至下方水平，我们将获得多头仓位。同时，止损限价订单转换为下一个上方水平的限价卖单。现在，EA 交易将再次检测到一个“裸”限价订单，并创建一个止损限价订单与其配对买入，比期货限价订单低一个水平。

如果有相反的仓位，我们会平仓。我们还将提供交易系统启用时的日内时段设置，在其余时间，所有订单和仓位都将被删除。这对于“收盘”尤其有用，因为此时市场的回报波动尤其明显。

当然，这只是网格策略的许多潜在实现之一，缺少了网格的很多定制项，但是我们不会使这个示例过于复杂。

EA 交易将分析每根柱线的情况（大概为 H1 时间或更短）。理论上，这个 EA 交易的运算逻辑需要通过及时响应 [交易事件](#) 来改进，但我们还没有探讨过这方面。因此，我们没有持续跟踪和即时“手动”恢复空缺网格水平的限价订单，而是通过使用止损限价订单将这项工作委托给服务器。但是，此处有一个细微的差别。

事实上，每个水平的限价订单和止损订单为相反的类型（*buy/sell*），因此由不同类型的价格激活。

造成的结果是，如果市价在网格的上半部分向上移动到下一个水平，*Ask* 价格可能会触及该水平并激活止损限价买入订单，*Bid* 价格不会达到该水平，卖出限价订单将保持原样（不会变为持仓）。在网格的下半部分，当市价向下移动时，情况是镜像的。任何水平首先被 *Bid* 价格触及，并激活止损限价订单卖出，只有在进一步下跌时，该水平才会被 *Ask* 价格触及。如果没有变动，买入限价订单将保持不变。

随着点差的增加，这个问题变得很关键。因此，EA 交易需要对“额外”限价订单进行额外控制。换言之，如果在其假定的目标价格（相邻水平）已经有限价订单，EA 交易将不会生成在该水平丢失的止损限价订单。

源代码包含在 *PendingOrderGrid1.mq5* 文件中。在输入参数中，你可以设置每次交易的 *Volume*（默认情况下，如果保留为 0，则采用图表交易品种的最小手数）、网格水平数 *GridSize*（必须是偶数）以及水平之间的步长 *GridStep*（点数）。在 *StartTime* 和 *StopTime* 参数中指定允许策略起作用的日内时段的开始和结束时间：在这两个参数中，只有时间是重要的。

```

#include <MQL5Book/MqlTradeSync.mqh>
#include <MQL5Book/OrderFilter.mqh>
#include <MQL5Book/MapArray.mqh>

input double Volume; // Volume (0 = minimal lot)
input uint GridSize = 6; // GridSize (even number c
input uint GridStep = 200; // GridStep (points)
input ENUM_ORDER_TYPE_TIME Expiration = ORDER_TIME_GTC;
input ENUM_ORDER_TYPE_FILLING Filling = ORDER_FILLING_FOK;
input datetime StartTime = D'1970.01.01 00:00:00'; // StartTime (hh:mm:ss)
input datetime StopTime = D'1970.01.01 09:00:00'; // StopTime (hh:mm:ss)
input ulong Magic = 1234567890;

```

工作时段可以在一天之内 (*StartTime* < *StopTime*), 也可以跨越一天的边界 (*StartTime* > *StopTime*), 例如, 从 22:00 到 09:00。如果两个时间相等, 则假定为全天交易。

在继续实现交易想法之前, 我们简化设置查询和将诊断信息输出到日志的任务。为此, 我们介绍了自己的结构体 *MqlTradeRequestSyncLog*, 其为 *MqlTradeRequestSync* 的派生结构体。

```

const ulong DAYLONG = 60 * 60 * 24; // length of the day in seconds

struct MqlTradeRequestSyncLog: public MqlTradeRequestSync
{
    MqlTradeRequestSyncLog()
    {
        magic = Magic;
        type_filling = Filling;
        type_time = Expiration;
        if(Expiration == ORDER_TIME_SPECIFIED)
        {
            expiration = (datetime)(TimeCurrent() / DAYLONG * DAYLONG
                + StopTime % DAYLONG);
            if(StartTime > StopTime)
            {
                expiration = (datetime)(expiration + DAYLONG);
            }
        }
    }
    ~MqlTradeRequestSyncLog()
    {
        Print(TU::StringOf(this));
        Print(TU::StringOf(this.result));
    }
};

```

在构造函数中, 我们用不变的值填充所有字段。在析构函数中, 我们记录有意义的查询和结果字段。显然, 自动对象的析构函数始终会在从形成和发送订单的代码块中退出时被调用, 也就是发送和接收的数据会被打印出来。

在 *OnInit* 中, 我们对输入变量的正确性进行一些检查, 特别是对于偶数网格大小。

```

int OnInit()
{
    if(GridSize < 2 || !(GridSize % 2))
    {
        Alert("GridSize should be 2, 4, 6+ (even number)");
        return INIT_FAILED;
    }
    return INIT_SUCCEEDED;
}

```

该算法的主要入口点是 `OnTick` 处理程序。为了简洁起见，我们将省略示例 *PendingOrderModify.mq5* 中基于 `TRADE_RETCODE_SEVERITY` 的错误处理机制。

对于逐柱线工作，该函数有一个静态变量 `lastBar`，其中存储了最后一个成功处理柱线的时间。同一根柱线上的所有后续分时报价都会被跳过。

```

void OnTick()
{
    static datetime lastBar = 0;
    if(iTime(_Symbol, _Period, 0) == lastBar) return;
    uint retcode = 0;

    ... // main algorithm (see further)

    const TRADE_RETCODE_SEVERITY severity = TradeCodeSeverity(retcode);
    if(severity < SEVERITY_RETRY)
    {
        lastBar = iTime(_Symbol, _Period, 0);
    }
}

```

为了系统化的目的，主算法将被分成几个辅助函数，而不是省略号。首先，我们来确定是否设置了当天的工作时段，如果设置了，当前是否启用了该策略。该属性存储在 `tradeScheduled` 变量中。

```

...
bool tradeScheduled = true;

if(StartTime != StopTime)
{
    const ulong now = TimeCurrent() % DAYLONG;

    if(StartTime < StopTime)
    {
        tradeScheduled = now >= StartTime && now < StopTime;
    }
    else
    {
        tradeScheduled = now >= StartTime || now < StopTime;
    }
}
...

```

启用交易后，首先使用 *CheckGrid* 函数检查是否已经有一个订单网络。如果没有网络，函数将返回 GRID_EMPTY 常量，我们应通过调用 *Setup Grid* 来创建网络。如果网络已经建立，检查是否有要平仓的相反仓位是有意义的：这是由 *CompactPositions* 函数完成的。

```

if(tradeScheduled)
{
    retcode = CheckGrid();

    if(retcode == GRID_EMPTY)
    {
        retcode = SetupGrid();
    }
    else
    {
        retcode = CompactPositions();
    }
}
...

```

交易期一结束，就必须删除订单并全部平仓（如果有）。这项工作分别由 *RemoveOrders* 和 *CompactPositions function* 函数来完成，这些函数带有一个布尔标志 (*true*)：这个单一可选自变量指示在相反平仓之后对剩余仓位进行简单平仓。

```

else
{
    retcode = CompactPositions(true);
    if(!retcode) retcode = RemoveOrders();
}

```

所有函数都返回一个服务器代码，用 *TradeCodeSeverity* 分析该代码以确认操作成功还是失败。根据 TRADE_RETCODE_SEVERITY，特殊应用代码 GRID_EMPTY和GRID_OK 也视为标准代码。

```

#define GRID_OK      +1
#define GRID_EMPTY   0

```

现在我们来逐个看下这些函数。

CheckGrid 函数使用本节开始时提供的 *OrderFilter* 类。筛选器请求当前交易品种和带有“我们的”标识号的所有挂单，找到的订单订单号存储在数组中。

```
uint CheckGrid()
{
    OrderFilter filter;
    ulong tickets[];

    filter.let(ORDER_SYMBOL, _Symbol).let(ORDER_MAGIC, Magic)
        .let(ORDER_TYPE, ORDER_TYPE_SELL, IS::GREATER)
        .select(tickets);
    const int n = ArraySize(tickets);
    if(!n) return GRID_EMPTY;
    ...
}
```

该网格的完整性是使用我们已经熟悉的 *MapArray* 类来分析的，该类存储“key=value”对。在这种情况下，key 表示水平（价格转换为点数），value 表示给定水平的订单类型的位掩码（叠加）。此外，限价和止损限价订单分别计入 *limits* 和 *stops* 变量。

```
// price levels => masks of types of orders existing there
MapArray<ulong,uint> levels;

const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
int limits = 0;
int stops = 0;

for(int i = 0; i < n; ++i)
{
    if(OrderSelect(tickets[i]))
    {
        const ulong level = (ulong)MathRound(OrderGetDouble(ORDER_PRICE_OPEN) / point);
        const ulong type = OrderGetInteger(ORDER_TYPE);
        if(type == ORDER_TYPE_BUY_LIMIT || type == ORDER_TYPE_SELL_LIMIT)
        {
            ++limits;
            levels.put(level, levels[level] | (1 << type));
        }
        else if(type == ORDER_TYPE_BUY_STOP_LIMIT || type == ORDER_TYPE_SELL_STOP_LIMIT)
        {
            ++stops;
            levels.put(level, levels[level] | (1 << type));
        }
    }
}
...
```

如果每种类型的订单数匹配并等于指定的网格大小，则一切正常。

```

if(limits == stops)
{
    if(limits == GridSize) return GRID_OK; // complete grid

    Alert("Error: Order number does not match requested");
    return TRADE_RETCODE_ERROR;
}
...

```

限价订单数量大于止损限价订单数量的情况是正常的：这意味着由于价格变动，一个或多个止损订单变成了限价订单。然后，该程序应在不足的水平上增加止损限价订单。特定水平的特定类型的单独订单可以通过 *RepairGridLevel* 函数进行设置。

```

if(limits > stops)
{
    const uint stopmask =
        (1 << ORDER_TYPE_BUY_STOP_LIMIT) | (1 << ORDER_TYPE_SELL_STOP_LIMIT);
    for(int i = 0; i < levels.getSize(); ++i)
    {
        if((levels[i] & stopmask) == 0) // there is no stop-limit order at this level
        {
            // the direction of the limit is required to set the reverse stop limit
            const bool buyLimit = (levels[i] & (1 << ORDER_TYPE_BUY_LIMIT));
            // checks for "extra" orders due to the spread are omitted here (see the
            ...
            // create a stop-limit order in the desired direction
            const uint retcode = RepairGridLevel(levels.getKey(i), point, buyLimit);
            if(TradeCodeSeverity(retcode) > SEVERITY_NORMAL)
            {
                return retcode;
            }
        }
    }
    return GRID_OK;
}
...

```

止损限价订单数量大于限价订单数量的情况视为错误（可能是服务器因为某种原因跳过了该价格）。

```

Alert("Error: Orphaned Stop-Limit orders found");
return TRADE_RETCODE_ERROR;
}

```

RepairGridLevel 函数执行以下操作。

```

uint RepairGridLevel(const ulong level, const double point, const bool buyLimit)
{
    const double price = level * point;
    const double volume = Volume == 0 ?
        SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;

    MqlTradeRequestSyncLog request;

    request.comment = "repair";

    // if there is an unpaired buy-limit, set the sell-stop-limit to it
    // if there is an unpaired sell-limit, set buy-stop-limit to it
    const ulong order = (buyLimit ?
        request.sellStopLimit(volume, price, price + GridStep * point) :
        request.buyStopLimit(volume, price, price - GridStep * point));
    const bool result = (order != 0) && request.completed();
    if(!result) Alert("RepairGridLevel failed");
    return request.result.retcode;
}

```

请注意，我们不需要实际填写该结构体（除了在必要时可以提供更多信息的注释之外），因为有些字段是由构造函数自动填写的，并且我们将交易量和价格直接传递给 *sellStopLimit* 或 *buyStopLimit* 方法。

在 *SetupGrid* 函数中使用了类似的方法，其创建了一个新的完整订单网络。在该函数的开始，我们准备了用于计算的变量，并说明了该结构体的 *MqlTradeRequestSyncLog* 数组。

```

uint SetupGrid()
{
    const double current = SymbolInfoDouble(_Symbol, SYMBOL_BID);
    const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
    const double volume = Volume == 0 ?
        SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;
    // central price of the range rounded to the nearest step,
    // from it up and down we identify the levels
    const double base = ((ulong)MathRound(current / point / GridStep) * GridStep)
        * point;
    const string comment = "G[" + DoubleToString(base,
        (int)SymbolInfoInteger(_Symbol, SYMBOL_DIGITS)) + "]";
    const static string message = "SetupGrid failed: ";
    MqlTradeRequestSyncLog request[][2]; // limit and stop-limit - one pair
    ArrayResize(request, GridSize);      // 2 pending orders per level
}

```

接下来，我们为该网格的下半部分和上半部分生成从中心向两侧发散的订单。


```

for(int i = 0; i < (int)GridSize / 2; ++i)
{
    const int k = i + 1;

    // bottom half of the grid
    request[i][0].comment = comment;
    request[i][1].comment = comment;

    if(!(request[i][0].buyLimit(volume, base - k * GridStep * point)))
    {
        Alert(message + (string)i + "/BL");
        return request[i][0].result.retcode;
    }
    if(!(request[i][1].sellStopLimit(volume, base - k * GridStep * point,
        base - (k - 1) * GridStep * point)))
    {
        Alert(message + (string)i + "/SSL");
        return request[i][1].result.retcode;
    }

    // top half of the grid
    const int m = i + (int)GridSize / 2;

    request[m][0].comment = comment;
    request[m][1].comment = comment;

    if(!(request[m][0].sellLimit(volume, base + k * GridStep * point)))
    {
        Alert(message + (string)m + "/SL");
        return request[m][0].result.retcode;
    }
    if(!(request[m][1].buyStopLimit(volume, base + k * GridStep * point,
        base + (k - 1) * GridStep * point)))
    {
        Alert(message + (string)m + "/BSL");
        return request[m][1].result.retcode;
    }
}

```

然后我们检查准备情况。

```

for(int i = 0; i < (int)GridSize; ++i)
{
    for(int j = 0; j < 2; ++j)
    {
        if(!request[i][j].completed())
        {
            Alert(message + (string)i + "/" + (string)j + " post-check");
            return request[i][j].result.retcode;
        }
    }
}
return GRID_OK;
}

```

尽管检查 (调用 `completed`) 与发送订单间隔开来, 我们的结构体仍然在内部使用了同步形式 `OrderSend`。事实上, 为了加快发送一批订单 (就像在我们的网格 EA 交易中一样), 最好使用异步版本 `OrderSendAsync`。但随后订单执行状态应从 `OnTradeTransaction` 的事件处理程序中启动。我们稍后将研究该程序。

发送任何订单时的错误都会导致提前退出循环并从服务器返回代码。如果出现错误, 这个测试 EA 交易就会直接停止后续工作。对于一个真实的机器人, 最好能提供对错误含义的智能分析, 如有必要, 删除所有订单并平仓。

挂单产生的仓位由函数 `CompactPositions` 平仓。

```

uint CompactPositions(const bool cleanup = false)

```

默认情况下, `cleanup` 参数等于 `false` 意味着在交易周期内定期 “清理” 仓位, 即对反向仓位 (如果有) 进行平仓。`cleanup=true` 的值用于在交易期末强制平仓所有仓位。

该函数通过一个辅助函数 `GetMyPositions`, 用多头和空头订单号填充 `ticketsLong` 和 `ticketsShort` 数组。我们已经在示例 `TradeCloseBy.mq5` 中使用了后者, 该示例在 [反向平仓: 全部和部分](#)一节中说明。该示例中的 `CloseByPosition` 函数在新的 EA 交易中经历了最小的变化: 它从服务器返回一个代码, 而不是成功或错误的逻辑指示符。

```

uint CompactPositions(const bool cleanup = false)
{
    uint retcode = 0;
    ulong ticketsLong[], ticketsShort[];
    const int n = GetMyPositions(_Symbol, Magic, ticketsLong, ticketsShort);
    if(n > 0)
    {
        Print("CompactPositions, pairs: ", n);
        for(int i = 0; i < n; ++i)
        {
            retcode = CloseByPosition(ticketsShort[i], ticketsLong[i]);
            if(retcode) return retcode;
        }
    }
    ...
}

```

`CompactPositions` 的第二部分仅在 `cleanup=true` 时有效。该函数远不完美, 很快我们会重新编写。

```

if(cleanup)
{
    if(ArraySize(ticketsLong) > ArraySize(ticketsShort))
    {
        retcode = CloseAllPositions(ticketsLong, ArraySize(ticketsShort));
    }
    else if(ArraySize(ticketsLong) < ArraySize(ticketsShort))
    {
        retcode = CloseAllPositions(ticketsShort, ArraySize(ticketsLong));
    }
}

return retcode;
}

```

对于所有找到的剩余仓位，通常通过调用 *CloseAllPositions* 来执行平仓。

```

uint CloseAllPositions(const ulong &tickets[], const int start = 0)
{
    const int n = ArraySize(tickets);
    Print("CloseAllPositions ", n);
    for(int i = start; i < n; ++i)
    {
        MqlTradeRequestSyncLog request;
        request.comment = "close down " + (string)(i + 1 - start)
            + " of " + (string)(n - start);
        if(!(request.close(tickets[i]) && request.completed()))
        {
            Print("Error: position is not closed ", tickets[i]);
            return request.result.retcode;
        }
    }
    return 0; // success
}

```

现在我们仅需考虑 *RemoveOrders* 函数。该函数还使用订单筛选器获取其列表，然后在循环中调用 *remove* 方法。

```

uint RemoveOrders()
{
    OrderFilter filter;
    ulong tickets[];
    filter.let(ORDER_SYMBOL, _Symbol).let(ORDER_MAGIC, Magic)
        .select(tickets);
    const int n = ArraySize(tickets);
    for(int i = 0; i < n; ++i)
    {
        MqlTradeRequestSyncLog request;
        request.comment = "removal " + (string)(i + 1) + " of " + (string)n;
        if(!(request.remove(tickets[i]) && request.completed()))
        {
            Print("Error: order is not removed ", tickets[i]);
            return request.result.retcode;
        }
    }
    return 0;
}

```

我们检查一下在默认设置下 (交易时段从 00:00 到 09:00), EA 交易在测试程序中是如何工作的。以下是在 EURUSD,H1 上发布的屏幕截图。



测试程序中的网格策略 PendingOrderGrid1.mq5

在日志中,除了关于批量创建的几个订单 (在一天的开始)及其在早上移除的周期性条目之外,我们还会定期看到网络的恢复 (添加订单而不是触发的订单)和平仓。

```

buy stop limit 0.01 EURUSD at 1.14200 (1.14000) (1.13923 / 1.13923)
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, »
» @ 1.14200, X=1.14000, ORDER_TIME_GTC, M=1234567890, repair
DONE, #=159, V=0.01, Bid=1.13923, Ask=1.13923, Request executed, Req=287
CompactPositions, pairs: 1
close position #152 sell 0.01 EURUSD by position #153 buy 0.01 EURUSD (1.13923 / 1.13
deal #18 buy 0.01 EURUSD at 1.13996 done (based on order #160)
deal #19 sell 0.01 EURUSD at 1.14202 done (based on order #160)
Positions collapse initiated
OK CloseBy Order/Deal/Position
TRADE_ACTION_CLOSE_BY, EURUSD, ORDER_TYPE_BUY, ORDER_FILLING_FOK, P=152, b=153, »
» M=1234567890, compacting
DONE, D=18, #=160, Request executed, Req=288

```

现在，我们来研究用于处理仓位的 MQL5 函数，并在我们的 EA 交易中改进其选择和分析。下面几节将讨论这个问题。

6.4.26 获取仓位列表

在许多 EA 交易示例中，我们已经使用了 MQL5 API 函数来分析未平仓交易仓位。本节给出了其正式说明。

值得注意的是，这组函数并不能创建、修改或删除仓位。正如我们前面看到的，所有这些操作都是通过发送订单间接执行的。如果成功执行，则进行交易，结果形成仓位。

另一个特点是这些函数只适用于在线仓位。为了恢复仓位的历史，就必须分析交易的历史。

PositionsTotal 函数允许你找出账户上持仓的总数（对于所有金融工具）。

int PositionsTotal()

对于仓位的净额结算会计（ACCOUNT_MARGIN_MODE_RETAIL_NETTING 和 ACCOUNT_MARGIN_MODE_EXCHANGE），任何时候每个交易品种只能有一个仓位。这个仓位可能来自一个或多个交易。

有了仓位的独立表示（ACCOUNT_MARGIN_MODE_RETAIL_HEDGING），每个交易品种可以同时开立几个仓位，包括多方向仓位。每笔进场交易都会创建一个单独的仓位，因此一个订单的部分分步执行可以生成几个仓位。

PositionGetSymbol 函数可根据仓位的编号返回其交易品种。

string PositionGetSymbol(int index)

索引必须介于 0 到 N-1 之间，其中 N 为预调用 **PositionsTotal** 收到的值。仓位的顺序没有规定。

如果没有找到仓位，那么将返回一个空字符串，错误代码将在 **_LastError** 中提供。

在几个测试 EA 交易中（[TrailingStop.mq5](#)、[TradeCloseBy.mq5](#)等）提供了使用这两个函数的示例，函数名为 **GetMyPosition/GetMyPositions**。

未平仓仓位的特性在于唯一订单号，即区别于其他仓位的编号，但在某些情况下，订单号在订单生命周期内可能会发生变化，例如一笔交易在净额结算模式下仓位反转，或者是服务器上服务操作的结果（为应计掉期、清算而重新开立）。

为了通过编号获取仓位订单号，我们使用了 **PositionGetTicket** 函数。

ulong PositionGetTicket(int index)

此外，该函数突出显示终端交易环境中的仓位，然后允许你使用一组特殊的 *PositionGet* 函数读取其特性。换言之，与订单类似，终端为每个 MQL 程序保持一个内部缓存来存储仓位的特性。为了突出显示仓位，除了 *PositionGetTicket* 之外，还有两个函数：*PositionSelect* *PositionSelect* 和 *PositionSelectByTicket*，我们将在下面讨论。

如果出现错误，*PositionGetTicket* 函数将返回 0。

不要将订单号与分配给每个仓位且永不改变的标识符相混淆。标识符用于将仓位与订单和交易联系起来。我们稍后会谈到这一点。

完成涉及仓位的请求需要订单号：订单号在 *MqlTradeRequest* 结构体的 *position* 和 *position_by* 字段中指定。此外，通过将订单号保存在变量中，程序随后可以使用 *PositionSelectByTicket* 函数（见下文）选择并使用特定的仓位，而无需在循环中重复枚举仓位。

当在净额结算账户上反向开仓时，POSITION_TICKET 将更改为发起此运算的订单订单号。但是，仍然可以使用 ID 来跟踪此类仓位。对冲模式不支持仓位反向。

bool PositionSelect(const string symbol)

该函数根据金融工具的名称选择持仓。

通过仓位的独立表示 (ACCOUNT_MARGIN_MODE_RETAIL_HEDGING)，每个交易品种可以同时有多个持仓。在这种情况下，*PositionSelect* 将选择订单号最小的仓位。

返回的结果表示函数执行成功 (*true*) 或不成功 (*false*)。

所选仓位的特性被缓存的事实意味着该仓位本身可能不再存在，或者如果程序在一段时间后读取其特性，它可能会被更改。建议在访问数据之前调用 *PositionSelect* 函数。

bool PositionSelectByTicket(ulong ticket)

该函数可选择的一个未平仓仓位，以便在指定的订单号上进一步工作。

我们将在后面看到使用该函数的示例（在学习 *特性* 及其相关的 *PositionGet* *PositionGet* 函数时）。

当使用 *PositionsTotal*、*OrdersTotal* 及类似函数构造算法时，应考虑终端运算的异步原则。在编写 *MqlTradeSync.mqh* 类和实现等待交易请求的执行结果时，我们已经谈到了这个主题。但是，这种等待在客户端不一定总是可行。具体来说，如果我们下了一个挂单，那么它到市场订单的转换和随后的执行将发生在服务器上。此时，订单可能不再列在活动订单中 (*OrdersTotal* 将返回 0)，但仓位尚未显示 (*PositionsTotal* 也等于 0)。因此，在没有仓位的情况下，具有下单条件的 MQL 程序可能会错误地启动新订单，其结果是仓位最终会翻倍。

为了解决这个问题，MQL 程序必须更深入地分析交易环境，而不仅仅是检查一次订单和仓位数量。例如，你可以保留交易环境最新正确状态的快照，并且不允许任何实体在没有某种确认的情况下消失。只有这样才能形成新的状态。因此，订单只能与仓位变更（创建、平仓）一起删除，或以取消状态移至历史。在 *TradeGuard.mqh* 文件中，以 *TradeGuard* 类的形式提出了一种可能的解决方案。本书还包括演示脚本 *TradeGuardExample.mq5*，你可以补充学习。

6.4.27 仓位特性

所有的仓位特性根据值的类型分为三组：整数型及其兼容型，实数型和字符串型。它们用于读取 *PositionGet* 函数，其类似于 *OrderGet* 函数。我们将在下一节介绍函数本身，此处我们将给出所有可用于在这些函数的第一个参数中指定的特性标识符。

整数型特性在 `ENUM_POSITION_PROPERTY_INTEGER` 枚举中提供。

标识符	说明	类型
<code>POSITION_TICKET</code>	仓位订单号	<code>ulong</code>
<code>POSITION_TIME</code>	开仓时间	<code>datetime</code>
<code>POSITION_TIME_MSC</code>	开仓时间 (毫秒)	<code>ulong</code>
<code>POSITION_TIME_UPDATE</code>	仓位变化 (交易量) 时间	<code>datetime</code>
<code>POSITION_TIME_UPDATE_MSC</code>	仓位变化 (交易量) 时间 (毫秒)	<code>ulong</code>
<code>POSITION_TYPE</code>	仓位类型	<code>ENUM_POSITION_TYPE</code>
<code>POSITION_MAGIC</code>	仓位 Magic 号 (基于 <code>ORDER_MAGIC</code>)	<code>ulong</code>
<code>POSITION_IDENTIFIER</code>	仓位标识符；分配给每个新开设仓位的唯一编号，在其整个寿命周期内不会改变。	<code>ulong</code>
<code>POSITION_REASON</code>	开仓原因	<code>ENUM_POSITION_REASON</code>

通常，`POSITION_IDENTIFIER` 对应于开仓订单的订单号。仓位标识符在每个开仓、更改或平仓的订单 (`ORDER_POSITION_ID`) 和交易 (`DEAL_POSITION_ID`) 中指明。因此，使用该标识符来搜索与仓位相关的订单和交易非常方便。

如果订单部分履行，那么剩余交易量的仓位和活动挂单以及匹配的订单号可以同时存在。而且此类仓位可以及时平仓，在下次补仓剩下挂单时，会再次出现同样订单号的仓位。

在净额结算模式下，用一笔交易反转仓位视为仓位变更，而不是新的变更，因此保留 `POSITION_IDENTIFIER`。只有在零交易量中平仓前一个仓位后，交易品种上的新仓位才是可能的。

`POSITION_TIME_UPDATE` 特性仅响应交易量的变化 (例如，部分平仓或持仓增加的结果)，而不响应其他参数，如 *Stop Loss/Take Profit* 水平或掉期费用。

仓位只有两种类型 (`ENUM_POSITION_TYPE`)。

标识符	说明
POSITION_TYPE_BUY	买入
POSITION_TYPE_SELL	卖出

ENUM_POSITION_REASON 枚举中提供了仓位原点的选项，即仓位如何打开。

标识符	说明
POSITION_REASON_CLIENT	触发从桌面终端下达的订单
POSITION_REASON_MOBILE	触发从移动应用程序下达的订单
POSITION_REASON_WEB	触发从 WEB 平台 (浏览器) 下达的订单
POSITION_REASON_EXPERT	触发由 EA 交易或脚本下达的订单

实数型特性收集在 ENUM_POSITION_PROPERTY_DOUBLE 中。

标识符	说明
POSITION_VOLUME	仓位交易量
POSITION_PRICE_OPEN	仓位价格
POSITION_SL	止损价格
POSITION_TP	止盈价格
POSITION_PRICE_CURRENT	当前交易品种价格
POSITION_SWAP	累计掉期
POSITION_PROFIT	当前盈利

当前价格类型对应于平仓运算。例如，多头仓位必须通过卖出平仓，因此在 POSITION_PRICE_CURRENT 中跟踪其 *Bid* 价格。

最后，仓位支持以下字符串型特性 (ENUM_POSITION_PROPERTY_STRING)。

标识符	说明
POSITION_SYMBOL	开仓的交易品种
POSITION_COMMENT	仓位注释
POSITION_EXTERNAL_ID	外部系统中的仓位 ID (交易所)

在看了仓位特性列表之后，我们可以开始来看读取这些特性的函数。

6.4.28 用于读取仓位特性的函数

根据特性类型，MQL 程序可以使用几个 *PositionGet* 函数来获取仓位特性。在所有函数中，被请求的特定特性在第一个参数中定义，该参数采用上一节中讨论的 `ENUM_POSITION_PROPERTY` 枚举之一的 ID。

对于每种类型的特性，都存在函数的简短形式和完整形式：第一种形式直接返回特性的值，第二种形式将其写入第二个参数，通过引用传递。

整数型特性及其兼容类型 (*datetime* 和枚举)特性可以通过 *PositionGetInteger* 函数获得。

```
long PositionGetInteger(ENUM_POSITION_PROPERTY_INTEGER property)
bool PositionGetInteger(ENUM_POSITION_PROPERTY_INTEGER property, long &value)
```

如果失败，该函数将返回 0 或 *false*。

PositionGetDouble 函数用于获取实数型特性。

```
double PositionGetDouble(ENUM_POSITION_PROPERTY_DOUBLE property)
bool PositionGetDouble(ENUM_POSITION_PROPERTY_DOUBLE property, double &value)
```

最后，字符串型特性由 *PositionGetString* 函数返回。

```
string PositionGetString(ENUM_POSITION_PROPERTY_STRING property)
bool PositionGetString(ENUM_POSITION_PROPERTY_STRING property, string &value)
```

如果失败，函数的第一种形式将返回一个空字符串。

为了读取仓位特性，我们已经有了一个现成的摘要界面 *MonitorInterface* (*TradeBaseMonitor.mqh*)，我们用其来编写订单监视器。现在很容易实现一个类似的仓位监视器。结果附在 *PositionMonitor.mqh* 文件中。

PositionMonitorInterface 类继承自 *MonitorInterface*，其用于为所考虑的 `ENUM_POSITION_PROPERTY` 枚举的模板类型 ID 和 S 赋值，并且考虑到仓位特性的具体情况，重写了两个 *stringify* 方法。

```

class PositionMonitorInterface:
    public MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER,
        ENUM_POSITION_PROPERTY_DOUBLE,ENUM_POSITION_PROPERTY_STRING>
{
public:
    virtual string stringify(const long v,
        const ENUM_POSITION_PROPERTY_INTEGER property) const override
    {
        switch(property)
        {
            case POSITION_TYPE:
                return enumstr<ENUM_POSITION_TYPE>(v);
            case POSITION_REASON:
                return enumstr<ENUM_POSITION_REASON>(v);

            case POSITION_TIME:
            case POSITION_TIME_UPDATE:
                return TimeToString(v, TIME_DATE | TIME_SECONDS);

            case POSITION_TIME_MSC:
            case POSITION_TIME_UPDATE_MSC:
                return STR_TIME_MSC(v);
        }

        return (string)v;
    }

    virtual string stringify(const ENUM_POSITION_PROPERTY_DOUBLE property,
        const string format = NULL) const override
    {
        if(format == NULL &&
            (property == POSITION_PRICE_OPEN || property == POSITION_PRICE_CURRENT
            || property == POSITION_SL || property == POSITION_TP))
        {
            const int digits = (int)SymbolInfoInteger(PositionGetString(POSITION_SYMBOL)
                SYMBOL_DIGITS);
            return DoubleToString(PositionGetDouble(property), digits);
        }
        return MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER,
            ENUM_POSITION_PROPERTY_DOUBLE,ENUM_POSITION_PROPERTY_STRING>
            ::stringify(property, format);
    }
}

```

准备好查看仓位的特定监视器类是继承链中的下一个类，其基于 *PositionGet* 函数。通过订单号选择仓位是在构造函数中完成的。

```

class PositionMonitor: public PositionMonitorInterface
{
public:
    const ulong ticket;
    PositionMonitor(const ulong t): ticket(t)
    {
        if(!PositionSelectByTicket(ticket))
        {
            PrintFormat("Error: PositionSelectByTicket(%lld) failed: %s",
                ticket, E2S(_LastError));
        }
        else
        {
            ready = true;
        }
    }

    virtual long get(const ENUM_POSITION_PROPERTY_INTEGER property) const override
    {
        return PositionGetInteger(property);
    }

    virtual double get(const ENUM_POSITION_PROPERTY_DOUBLE property) const override
    {
        return PositionGetDouble(property);
    }

    virtual string get(const ENUM_POSITION_PROPERTY_STRING property) const override
    {
        return PositionGetString(property);
    }
    ...
};

```

一个简单脚本将允许你记录第一个仓位的所有特性（如果至少有一个可用）。

```

void OnStart()
{
    PositionMonitor pm(PositionGetTicket(0));
    pm.print();
}

```

在日志中，我们应会看到此类内容。

```

MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER, »
    » ENUM_POSITION_PROPERTY_DOUBLE,ENUM_POSITION_PROPERTY_STRING>
ENUM_POSITION_PROPERTY_INTEGER Count=9
0 POSITION_TIME=2022.03.24 23:09:45
1 POSITION_TYPE=POSITION_TYPE_BUY
2 POSITION_MAGIC=0
3 POSITION_IDENTIFIER=1291755067
4 POSITION_TIME_MSC=2022.03.24 23:09:45'261
5 POSITION_TIME_UPDATE=2022.03.24 23:09:45
6 POSITION_TIME_UPDATE_MSC=2022.03.24 23:09:45'261
7 POSITION_TICKET=1291755067
8 POSITION_REASON=POSITION_REASON_EXPERT
ENUM_POSITION_PROPERTY_DOUBLE Count=8
0 POSITION_VOLUME=0.01
1 POSITION_PRICE_OPEN=1.09977
2 POSITION_PRICE_CURRENT=1.09965
3 POSITION_SL=0.00000
4 POSITION_TP=1.10500
5 POSITION_COMMISSION=0.0
6 POSITION_SWAP=0.0
7 POSITION_PROFIT=-0.12
ENUM_POSITION_PROPERTY_STRING Count=3
0 POSITION_SYMBOL=EURUSD
1 POSITION_COMMENT=
2 POSITION_EXTERNAL_ID=

```

如果目前没有未结仓位，我们会看到一条错误消息。

```
Error: PositionSelectByTicket(0) failed: TRADE_POSITION_NOT_FOUND
```

但是，监视器的用处不仅仅在于将特性输出到日志中。基于 *PositionMonitor*，我们创建了一个按条件选择仓位的类，类似于我们对订单 (*OrderFilter*) 所创建的。最终目标是为了改进我们的网格 EA 交易。

有了 OOP，创建新的筛选器类就变得非常简单。以下是完整的源代码（文件 *PositionFilter.mqh*）。

```

class PositionFilter: public TradeFilter<PositionMonitor,
    ENUM_POSITION_PROPERTY_INTEGER,
    ENUM_POSITION_PROPERTY_DOUBLE,
    ENUM_POSITION_PROPERTY_STRING>
{
protected:
    virtual int total() const override
    {
        return PositionsTotal();
    }
    virtual ulong get(const int i) const override
    {
        return PositionGetTicket(i);
    }
};

```

例如，现在我们可以编写这样一个脚本，在给定魔术编号的仓位上获得特定的盈利。

```

input ulong Magic;

void OnStart()
{
    PositionFilter filter;

    ENUM_POSITION_PROPERTY_DOUBLE properties[] =
        {POSITION_PROFIT, POSITION_VOLUME};

    double profits[][2];
    ulong tickets[];
    string symbols[];

    filter.let(POSITION_MAGIC, Magic).select(properties, tickets, profits);
    filter.select(POSITION_SYMBOL, tickets, symbols);

    for(int i = 0; i < ArraySize(symbols); ++i)
    {
        PrintFormat("%s[%lld]=%f",
            symbols[i], tickets[i], profits[i][0] / profits[i][1]);
    }
}

```

在这种情况下，我们必须调用 *select* 方法两次，因为我们感兴趣的特性类型是不同的：真实盈利和手数以及工具的字符串名称。在本章开头的其中一节中，当我们开发交易品种的筛选器类时，对 *元组* 的概念进行了说明。在 MQL5 中，我们可以将其实现为具有任意类型字段的结构体模板。这种元组对于最终确定筛选器类的层次结构体非常方便，因为这样就能说明用任何类型的字段填充元组数组的 *select* 方法。

元组在 *Tuples.mqh* 文件中说明。其中的所有结构体都具有名称 *TupleN<T1,...>*，其中 N 为 2 到 8 之间的数字，并且对应于模板参数 (Ti 类型) 的数量。例如，*Tuple2*：

```

template<typename T1,typename T2>
struct Tuple2
{
    T1 _1;
    T2 _2;

    static int size() { return 2; };

    // M - order, position, deal monitor class, any MonitorInterface<>
    template<typename M>
    void assign(const int &properties[], M &m)
    {
        if(ArraySize(properties) != size()) return;
        _1 = m.get(properties[0], _1);
        _2 = m.get(properties[1], _2);
    }
};

```

在 *TradeFilter* (*TradeFilter.mqh*) 类中，我们添加一个带有元组的 *select* 函数版本。

```

template<typename T,typename I,typename D,typename S>
class TradeFilter
{
    ...
    template<typename U> // type U must be Tuple<>, e.g. Tuple3<T1,T2,T3>
    bool select(const int &property[], U &data[], const bool sort = false) const
    {
        const int q = ArraySize(property);
        static const U u; // PRB: U::size() does not compile
        if(q != u.size()) return false; // required condition

        const int n = total();
        // cycle through orders/positions/deals
        for(int i = 0; i < n; ++i)
        {
            const ulong t = get(i);
            // access to properties via monitor T
            T m(t);
            // check all filter conditions for different types of properties
            if(match(m, longs)
            && match(m, doubles)
            && match(m, strings))
            {
                // for a suitable object, store the properties in an array of tuples
                const int k = EXPAND(data);
                data[k].assign(property, m);
            }
        }

        if(sort)
        {
            sortTuple(data, u._1);
        }

        return true;
    }
}

```

元组数组可以根据第一个 `field_1` 进行排序，因此你可以另外研究 *sortTuple* 辅助方法。

使用元组，你可以在一次 *select* 调用中查询一个筛选对象的三种不同类型的特性。

下面有一些显示 *Magic* 编号的仓位，对于每个额外获得一个交易品种和订单号，按盈利排序。

```

input ulong Magic;

void OnStart()
{
    int props[] = {POSITION_PROFIT, POSITION_SYMBOL, POSITION_TICKET};
    Tuple3<double,string,ulong> tuples[];
    PositionFilter filter;
    filter.let(POSITION_MAGIC, Magic).select(props, tuples, true);
    ArrayPrint(tuples);
}

```

当然，元组数组（本例中为 *Tuple3<double,string,ulong>*）说明中的参数类型必须与请求的特性枚举类型（*POSITION_PROFIT*, *POSITION_SYMBOL*, *POSITION_TICKET*）相匹配。

现在我们可以稍微简化一下网格 EA 交易（这不仅意味着代码更短，而且更容易理解）。新版本名为 *PendingOrderGrid2.mq5*。这些变化将会影响与仓位管理相关的所有函数。

GetMyPositions 函数用于填充通过引用传递的元组的 *types4tickets* 数组。在每个 *Tuple2* 元组中，应存储仓位的类型和标签。在这种特殊情况下，我们可以只用一个二维数组 *ulong* 来代替元组，因为两个特性都是相同的基础类型。但是，我们使用元组来演示如何在调用代码中使用它们。

```

#include <MQL5Book/Tuples.mqh>
#include <MQL5Book/PositionFilter.mqh>

int GetMyPositions(const string s, const ulong m,
    Tuple2<ulong,ulong> &types4tickets[])
{
    int props[] = {POSITION_TYPE, POSITION_TICKET};
    PositionFilter filter;
    filter.let(POSITION_SYMBOL, s).let(POSITION_MAGIC, m)
        .select(props, types4tickets, true);
    return ArraySize(types4tickets);
}

```

注意，*select* 方法的最后一个参数（第三个）等于 *true*，它指示按照第一个字段（即仓位类型）对数组进行排序。因此，我们将在开始时买入，在结束时卖出。这是对冲平仓所必需的。

CompactPositions 方法的循环如下。

```

uint CompactPositions(const bool cleanup = false)
{
    uint retcode = 0;
    Tuple2<ulong,ulong> types4tickets[];
    int i = 0, j = 0;
    int n = GetMyPositions(_Symbol, Magic, types4tickets);
    if(n > 0)
    {
        Print("CompactPositions: ", n);
        for(i = 0, j = n - 1; i < j; ++i, --j)
        {
            if(types4tickets[i]._1 != types4tickets[j]._1) // as long as the types are d
            {
                retcode = CloseByPosition(types4tickets[i]._2, types4tickets[j]._2);
                if(retcode) return retcode; // error
            }
            else
            {
                break;
            }
        }
    }

    if(cleanup && j < n)
    {
        retcode = CloseAllPositions(types4tickets, i, j + 1);
    }

    return retcode;
}

```

CloseAllPositions 函数几乎相同：


```

uint CloseAllPositions(const Tuple2<ulong,ulong> &types4tickets[],
    const int start = 0, const int end = 0)
{
    const int n = end == 0 ? ArraySize(types4tickets) : end;
    Print("CloseAllPositions ", n - start);
    for(int i = start; i < n; ++i)
    {
        MqlTradeRequestSyncLog request;
        request.comment = "close down " + (string)(i + 1 - start)
            + " of " + (string)(n - start);
        const ulong ticket = types4tickets[i]._2;
        if(!(request.close(ticket) && request.completed()))
        {
            Print("Error: position is not closed ", ticket);
            return request.result.retcode; // error
        }
    }
    return 0; // success
}

```

你可以在测试程序中比较 EA 交易 *PendingOrderGrid1.mq5* 和 *PendingOrderGrid2.mq5* 的效果。

报告将略有不同，因为如果有几个仓位，它们会以相反的组合平仓，因此其他不成对的仓位平仓是相对于它们各自的点差进行的。

6.4.29 交易特性

交易反映了一个事实，即交易操作是在订单的基础上进行的。由于分批执行或反向平仓，一个订单可以产生几笔交易。

交易的特性在于三种基本类型的特性：整数型（及其兼容性）、实数型和字符串型。每种特性由其自己的常数在枚举之一中说明： `ENUM_DEAL_PROPERTY_INTEGER`、`ENUM_DEAL_PROPERTY_DOUBLE`、`ENUM_DEAL_PROPERTY_STRING`。

要读取交易特性，可使用 *HistoryDealGet* 函数。所有这些方法都假设之前使用特殊函数 *从历史中选择订单和交易* 请求了必要的部分历史。

整数型特性在 `ENUM_DEAL_PROPERTY_INTEGER` 枚举中说明。

标识符	说明	类型
DEAL_TICKET	交易订单号；分配给每笔交易的唯一编号	ulong
DEAL_ORDER	执行交易时所依据的订单号	ulong
DEAL_TIME	交易时间	datetime
DEAL_TIME_MSC	交易时间（毫秒）	ulong
DEAL_TYPE	交易类型	ENUM_DEAL_TYPE（参见下文）
DEAL_ENTRY	交易方向；市场进入，市场退出或反向	ENUM_DEAL_ENTRY（参见下文）
DEAL_MAGIC	交易的魔术数字（基于ORDER_MAGIC）	ulong
DEAL_REASON	交易原因或来源	ENUM_DEAL_REASON（参见下文）
DEAL_POSITION_ID	由交易开立、修改或平仓的仓位标识符	ulong

由 ENUM_DEAL_TYPE 枚举表示的可能交易类型

标识符	说明
DEAL_TYPE_BUY	买入
DEAL_TYPE_SELL	卖出
DEAL_TYPE_BALANCE	应计余额
DEAL_TYPE_CREDIT	应计信贷
DEAL_TYPE_CHARGE	附加费用
DEAL_TYPE_CORRECTION	更正
DEAL_TYPE_BONUS	奖金
DEAL_TYPE_COMMISSION	额外佣金
DEAL_TYPE_COMMISSION_DAILY	在交易日结束时收取的佣金
DEAL_TYPE_COMMISSION_MONTHLY	在月末收取佣金
DEAL_TYPE_COMMISSION_AGENT_DAILY	在交易日结束时收取的代理佣金
DEAL_TYPE_COMMISSION_AGENT_MONTHLY	在月末收取的代理佣金

标识符	说明
DEAL_TYPE_INTEREST	自由资金的应计利息
DEAL_TYPE_BUY_CANCELED	已取消的买入交易
DEAL_TYPE_SELL_CANCELED	已取消的卖出交易
DEAL_DIVIDEND	应计红利
DEAL_DIVIDEND_FRANKED	应计已付股息 (免税)
DEAL_TAX	应计税

DEAL_TYPE_BUY_CANCELED 和 DEAL_TYPE_SELL_CANCELED 选项反映了取消早期交易的情况。在这种情况下, 先前执行的交易类型 (DEAL_TYPE_BUY 或 DEAL_TYPE_SELL) 被更改为 DEAL_TYPE_BUY_CANCELED 或 DEAL_TYPE_SELL_CANCELED, 并且其利润/损失重置为零。之前收到的利润/损失作为单独的余额运算从账户中贷记/借记。

交易在改变仓位的方式上有所不同。可以是简单的开仓 (入市), 增加先前开仓的交易量, 以相反方向的交易平仓, 或者当相反的交易覆盖先前开仓的交易量时反转仓位。只有净额结算账户支持后一种运算。

所有这些情况都由 ENUM_DEAL_ENTRY 枚举的元素说明。

标识符	说明
DEAL_ENTRY_IN	进入市场
DEAL_ENTRY_OUT	退出市场
DEAL_ENTRY_INOUT	反转
DEAL_ENTRY_OUT_BY	以相反仓位平仓

在 ENUM_DEAL_REASON 枚举中汇总了交易的原因。

标识符	说明
DEAL_REASON_CLIENT	触发从桌面终端下达的订单
DEAL_REASON_MOBILE	触发从移动应用程序下达的订单
DEAL_REASON_WEB	触发从WEB平台下达的订单
DEAL_REASON_EXPERT	触发由EA交易或脚本下达的订单
DEAL_REASON_SL	触发止损订单
DEAL_REASON_TP	触发止盈订单
DEAL_REASON_SO	强行平仓事件
DEAL_REASON_ROLLOVER	仓位展期至新交易日
DEAL_REASON_VMARGIN	添加/扣除变动保证金
DEAL_REASON_SPLIT	分割(低价)有仓位的金融工具

实数型特性由 ENUM_DEAL_PROPERTY_DOUBLE 枚举表示。

标识符	说明
DEAL_VOLUME	交易量
DEAL_PRICE	交易价格
DEAL_COMMISSION	交易佣金
DEAL_SWAP	收盘时的累计掉期
DEAL_PROFIT	交易的财务结果
DEAL_FEE	交易后立即收取的交易费用
DEAL_SL	止损水平
DEAL_TP	止盈水平

最后两个特性填写如下：对于进场或反向交易，**Stop Loss/Take Profit** 值取自开仓或平仓的订单。对于退出交易，**Stop Loss/Take Profit** 值取自平仓时的仓位。

字符串型交易特性可通过 ENUM_DEAL_PROPERTY_STRING 枚举常量获得。

标识符	说明
DEAL_SYMBOL	达成交易的交易品种名称
DEAL_COMMENT	交易注释
DEAL_EXTERNAL_ID	外部交易系统(交易所)中的交易标识符

我们将在 *HistoryDealGet* 函数 章节中测试如何通过 *DealMonitor* 和 *DealFilter* 类来读取特性。

6.4.30 从历史中选择订单和交易

MetaTrader 5 允许你为 EA 交易或脚本创建特定时段的历史快照。该快照为订单和交易列表，可以通过适当的函数进一步访问。此外，可以请求与特定订单、交易或仓位相关的历史。

可通过 *HistorySelect* 函数显式 (按日期) 选择所需的时段。之后，可以分别使用 *HistoryDealsTotal* 和 *HistoryOrdersTotal* 函数找到交易和订单列表的大小。可以使用 *HistoryOrderGetTicket* 函数检查订单列表的元素；使用 *HistoryDealGetTicket* 函数检查交易列表的元素。

必须区分活动 (工作) 订单和历史订单，即已执行、已取消或已拒绝的订单。要分析活动订单，请使用与 *获取活动订单列表* 和 *读取其特性* 相关章节中讨论的函数。

bool HistorySelect(datetime from, datetime to)

该函数可请求指定服务器时段 (*from* 和 *to* (含), *to* >= *from*) 内的交易和订单历史，如果成功，则返回 *true*。

即使在请求的时段内没有订单和交易，如果没有错误，该函数也将返回 *true*。例如，错误可能是缺少用于构建订单或交易列表的内存。

请注意，订单有两个时间：设置 (ORDER_TIME_SETUP) 和执行 (ORDER_TIME_DONE)。函数 *HistorySelect* 可根据执行时间选择订单。

要提取整个账户历史，可以使用句法 *HistorySelect(0, LONG_MAX)*。

访问部分历史的另一种方式是通过仓位 ID。

bool HistorySelectByPosition(ulong positionID)

该函数可请求 ORDER_POSITION_ID 和 DEAL_POSITION_ID 特性中具有指定仓位 ID 的交易和订单历史。

注意！该函数不会按结算运算的相反仓位的 ID 选择订单。换言之，尽管订单数据参与了仓位的形成，ORDER_POSITION_BY_ID 特性被忽略了。

例如，EA 交易可以在启用的对冲账户上完成买入 (订单 #1) 和卖出 (订单 #2)。这将导致仓位 #1 和 #2 的形成。相反仓位平仓需要 ORDER_TYPE_CLOSE_BY (#3) 订单。因此，*HistorySelectByPosition(#1)* 调用将选择订单 #1 和 #3，这是我们所期望的。但是，*HistorySelectByPosition(#2)* 调用将仅选择订单 #2 (尽管订单 #3 在 ORDER_POSITION_BY_ID 特性中有 #2，严格而言，订单 #3 参与了仓位 #2 的平仓)。

在成功执行 *HistorySelect* 或 *HistorySelectByPosition* 这两个函数中的任何一个后，终端将为 MQL 程序生成一个内部订单和交易列表。你还可以使用函数 *HistoryOrderSelect* 和 *HistoryDealSelect* 来更改历史上下文，为此你需要预先知道相应对象的订单号 (例如，从请求结果中保存该订单号)。

需要注意的是，*HistoryOrderSelect* 仅影响订单列表，而 *HistoryDealSelect* 仅影响交易列表。

所有上下文选择函数都会返回一个 *bool* 值，表示成功 (*true*) 或错误 (*false*)。可以在内置的 *_LastError* 变量中读取错误代码。

bool HistoryOrderSelect(ulong ticket)

HistoryOrderSelect 函数可根据订单号选择历史中的订单。然后，该订单将用于交易的进一步操作 (读取特性)。

在应用 *HistoryOrderSelect* 函数的过程中，如果按订单号搜索订单成功，历史中选择的新订单列表将包含刚刚找到的唯一订单。换言之，先前选择的订单列表（如果有的话）会被重置。但是，该函数不会重置先前选择的交易历史，即，不选择与改订单相关联的交易。

bool HistoryDealSelect(ulong ticket)

函数 *HistoryDealSelect* 可选择历史中的交易，以便通过适当的函数进一步访问。该函数不会重置订单历史，即，不会选择与所选交易相关的订单。

通过调用上述函数之一在历史中选择了某个上下文之后，MQL 程序可以调用这些函数来迭代落入该上下文的订单和交易，并读取其特性。

int HistoryOrdersTotal()

HistoryOrdersTotal 函数可返回历史中的订单数（在选择中）。

ulong HistoryOrderGetTicket(int index)

HistoryOrderGetTicket 函数允许你在选定的历史上下文中通过订单序列号来获取订单。该索引必须介于 0 到 N-1 之间，其中 N 是从 *HistoryOrdersTotal* 函数中获得的。

知道了订单号，可使用 *HistoryOrderGet* 函数很容易获得其所有必要特性。历史订单的特性与现有订单的特性完全相同。

处理交易也有类似的一对函数。

int HistoryDealsTotal()

HistoryDealsTotal 函数可返回历史中的交易数（在选择中）。

ulong HistoryDealGetTicket(int index)

HistoryDealGetTicket 函数允许你在选定的历史上下文中通过序列号获取交易单。这是使用 *HistoryDealGet* 函数进一步处理交易所必需的。通过这些函数可访问的 *交易特性* 列表在上一节中已有说明。

我们在学习 *HistoryOrderGet* 和 *HistoryDealGet* 函数后探讨一个使用这些函数的示例。

6.4.31 用于从历史中读取订单特性的函数

根据特性值的基本类型，按照将可用特性的标识符划分为 3 种枚举，这与将可用属性标识符划分为三个枚举的方式一致：*ENUM_ORDER_PROPERTY_INTEGER*、*ENUM_ORDER_PROPERTY_DOUBLE* 和 *ENUM_ORDER_PROPERTY_STRING*，这在前面研究活动订单时的 *单独的章节* 中讨论过。

在调用这些函数之前，你需要以某种方式 *在历史中选择一组合适的订单号*。

如果你尝试读取其订单号不在所选历史上下文内的订单或交易的特性，环境可能会生成 *WRONG_INTERNAL_PARAMETER (4002)* 错误，可以通过 *_LastError* 对其进行分析。

对于每个基础特性类型，有两种函数形式：一种直接返回所请求特性的值，第二种将它写入通过引用传递的参数，并返回成功指示符 (*true*) 或错误 (*false*)。

对于整数型及其兼容型 (*datetime* 和枚举) 特性，有一个专用函数 *HistoryOrderGetInteger*。

```
long HistoryOrderGetInteger(ulong ticket, ENUM_ORDER_PROPERTY_INTEGER property)
bool HistoryOrderGetInteger(ulong ticket, ENUM_ORDER_PROPERTY_INTEGER property,
    long &value)
```

该函数允许你通过订单号从所选历史中找出订单 *property*。

对于实数型特性，分配了 *HistoryOrderGetDouble* 函数

```
double HistoryOrderGetDouble(ulong ticket, ENUM_ORDER_PROPERTY_DOUBLE property)
bool HistoryOrderGetDouble(ulong ticket, ENUM_ORDER_PROPERTY_DOUBLE property,
    double &value)
```

最后，可以使用 *HistoryOrderGetString* 读取字符串型特性。

```
string HistoryOrderGetString(ulong ticket, ENUM_ORDER_PROPERTY_STRING property)
bool HistoryOrderGetString(ulong ticket, ENUM_ORDER_PROPERTY_STRING property,
    string &value)
```

现在我们可以补充 *OrderMonitor* 类 (*OrderMonitor.mqh*) 来处理历史订单。首先，让我们将一个布尔变量添加到 *history* 类中，我们将根据选择带有传递订单号订单的片段将该变量填充到构造函数中：在活跃订单中 (*OrderSelect*) 或在历史 (*HistoryOrderSelect*) 中。

```
class OrderMonitor: public OrderMonitorInterface
{
    bool history;

public:
    const ulong ticket;
    OrderMonitor(const long t): ticket(t), history(!OrderSelect(t))
    {
        if(history && !HistoryOrderSelect(ticket))
        {
            PrintFormat("Error: OrderSelect(%lld) failed: %s", ticket, E2S(_LastError));
        }
        else
        {
            ResetLastError();
            ready = true;
        }
    }
    ...
}
```

我们需要在成功的 *if* 分支中调用 *ResetLastError* 函数，以便重置可能由 *OrderSelect* 函数设置的错误 (如果订单在历史中)。

事实上，这个版本的构造函数包含了一个严重的逻辑错误，我们将在几段后返回来研究。

为了读取 *get* 方法中的特性，我们现在调用不同的内置函数，具体取决于 *history* 变量的值。

```

virtual long get(const ENUM_ORDER_PROPERTY_INTEGER property) const override
{
    return history?HistoryOrderGetInteger(ticket, property) : OrderGetInteger(property);
}

virtual double get(const ENUM_ORDER_PROPERTY_DOUBLE property) const override
{
    return history?HistoryOrderGetDouble(ticket, property) : OrderGetDouble(property);
}

virtual string get(const ENUM_ORDER_PROPERTY_STRING property) const override
{
    return history?HistoryOrderGetString(ticket, property) : OrderGetString(property);
}
...

```

OrderMonitor 类的主要用途是向其他分析类提供数据。*OrderMonitor* 对象用于筛选 *OrderFilter* 类中的活动订单，我们需要一个类似的类来根据历史中的任意条件选择订单：*HistoryOrderFilter*。

让我们在同一个文件 *OrderFilter.mqh* 中编写这个类。其使用两个新函数来处理历史：*HistoryOrdersTotal* 和 *HistoryOrderGetTicket*。

```

class HistoryOrderFilter: public TradeFilter<OrderMonitor,
    ENUM_ORDER_PROPERTY_INTEGER,
    ENUM_ORDER_PROPERTY_DOUBLE,
    ENUM_ORDER_PROPERTY_STRING>
{
protected:
    virtual int total() const override
    {
        return HistoryOrdersTotal();
    }
    virtual ulong get(const int i) const override
    {
        return HistoryOrderGetTicket(i);
    }
};

```

这个简单的代码继承自模板类 *TradeFilter*，其中该类作为模板 *OrderMonitor* 的第一个参数传递，以读取相应对象的特性（我们看到了一个用于仓位的模拟，很快也会为交易创建一个模拟）。

这就是 *OrderMonitor* 构造函数的问题所在。正如我们在 [从历史中选择订单和交易](#) 一节中了解到的，为了分析账户，我们必须首先用一个函数（如 *HistorySelect*）设置上下文。因此，在源代码 *HistoryOrderFilter* 中，假设 MQL 程序已经选择了所需的历史片段。但是，*OrderMonitor* 构造函数的新中间版本可使用 *HistoryOrderSelect* 调用来检查订单号在历史中是否存在。同时，该函数还可以重置历史订单的先前上下文并选择单个订单。

因此，我们需要一个辅助方法 *historyOrderSelectWeak*，以一种“软”的方式验证订单号，而不破坏现有的上下文。为此，我们可以简单地检查 *ORDER_TICKET* 特性是否等于传递的订单号 *t*：*(HistoryOrderGetInteger(t, ORDER_TICKET) == t)*。如果已经选择了这样的订单号（可用），则检查成功，并且监视器不需要操作历史。


```

class OrderMonitor: public OrderMonitorInterface
{
    bool historyOrderSelectWeak(const ulong t) const
    {
        return (((HistoryOrderGetInteger(t, ORDER_TICKET) == t) ||
            (HistorySelect(0, LONG_MAX) && (HistoryOrderGetInteger(t, ORDER_TICKET) == t
        }
    bool history;

public:
    const ulong ticket;
    OrderMonitor(const long t): ticket(t), history(!OrderSelect(t))
    {
        if(history && !historyOrderSelectWeak(ticket))
        {
            PrintFormat("Error: OrderSelect(%lld) failed: %s", ticket, E2S(_LastError));
        }
        else
        {
            ResetLastError();
            ready = true;
        }
    }
}

```

在我们为交易准备了类似的功能后，下一节将考虑对历史应用订单筛选的示例。

6.4.32 用于从历史中读取交易特性的函数

为了读取交易特性，函数组按 **特性类型** 组织，具体分为整数型、实数型和字符串型。在调用函数之前，你需要选择所需的 **历史时期**，从而确保所有函数的第一个参数 (**ticket**) 中传递的订单号交易的可用性。

每种类型的特性都有两种形式：直接返回值和通过引用写入变量。第二种形式返回 **true** 表示成功。第一种形式仅在出错时返回 0。错误代码在 **_LastError** 变量中。

整数型及其兼容的特性类型 (**datetime** 和枚举) 可以使用 **HistoryDealGetInteger** 函数获得。

```

long HistoryDealGetInteger(ulong ticket, ENUM_DEAL_PROPERTY_INTEGER property)
bool HistoryDealGetInteger(ulong ticket, ENUM_DEAL_PROPERTY_INTEGER property,
    long &value)

```

实数型特性由 **HistoryDealGetDouble** 函数读取。

```

double HistoryDealGetDouble(ulong ticket, ENUM_DEAL_PROPERTY_DOUBLE property)
bool HistoryDealGetDouble(ulong ticket, ENUM_DEAL_PROPERTY_DOUBLE property,
    double &value)

```

对于字符串型特性，提供了 **HistoryDealGetString** 函数。

```
string HistoryDealGetString(ulong ticket, ENUM_DEAL_PROPERTY_STRING property)
bool HistoryDealGetString(ulong ticket, ENUM_DEAL_PROPERTY_STRING property,
    string &value)
```

DealMonitor 类 (*DealMonitor.mqh*) 可提供交易特性的统一读数，其组织方式与 *OrderMonitor* 和 *PositionMonitor* 完全相同。基础类为 *DealMonitorInterface*，继承自模板 *MonitorInterface*（我们在读取活动订单特性的函数一节中进行说明）。正是在这个水平，ENUM_DEAL_PROPERTY 枚举的特定类型被指定为模板参数和 *stringify* 方法的特定实现。

```
#include <MQL5Book/TradeBaseMonitor.mqh>

class DealMonitorInterface:
    public MonitorInterface<ENUM_DEAL_PROPERTY_INTEGER,
        ENUM_DEAL_PROPERTY_DOUBLE, ENUM_DEAL_PROPERTY_STRING>
{
public:
    // property descriptions taking into account integer subtypes
    virtual string stringify(const long v,
        const ENUM_DEAL_PROPERTY_INTEGER property) const override
    {
        switch(property)
        {
            case DEAL_TYPE:
                return enumstr<ENUM_DEAL_TYPE>(v);
            case DEAL_ENTRY:
                return enumstr<ENUM_DEAL_ENTRY>(v);
            case DEAL_REASON:
                return enumstr<ENUM_DEAL_REASON>(v);

            case DEAL_TIME:
                return TimeToString(v, TIME_DATE | TIME_SECONDS);

            case DEAL_TIME_MSC:
                return STR_TIME_MSC(v);
        }

        return (string)v;
    }
};
```

下面的 *DealMonitor* 类有点类似于最近为了处理历史数据而修改的 *OrderMonitor* 类。除了应用 *HistoryDeal* 函数而不是 *HistoryOrder* 函数之外，应注意的是，对于交易，不需要在在线环境中检查订单号，因为交易只存在于历史中。

```

class DealMonitor: public DealMonitorInterface
{
    bool historyDealSelectWeak(const ulong t) const
    {
        return ((HistoryDealGetInteger(t, DEAL_TICKET) == t) ||
            (HistorySelect(0, LONG_MAX) && (HistoryDealGetInteger(t, DEAL_TICKET) == t)))
    }
public:
    const ulong ticket;
    DealMonitor(const long t): ticket(t)
    {
        if(!historyDealSelectWeak(ticket))
        {
            PrintFormat("Error: HistoryDealSelect(%lld) failed", ticket);
        }
        else
        {
            ready = true;
        }
    }

    virtual long get(const ENUM_DEAL_PROPERTY_INTEGER property) const override
    {
        return HistoryDealGetInteger(ticket, property);
    }

    virtual double get(const ENUM_DEAL_PROPERTY_DOUBLE property) const override
    {
        return HistoryDealGetDouble(ticket, property);
    }

    virtual string get(const ENUM_DEAL_PROPERTY_STRING property) const override
    {
        return HistoryDealGetString(ticket, property);
    }
    ...
};

```

基于 *DealMonitor* 和 *TradeFilter*, 很容易创建交易筛选器 (*DealFilter.mqh*)。注意, *TradeFilter* 作为许多实体的基础类, 在 [按特性选择订单](#) 一节中进行了说明。

```

#include <MQL5Book/DealMonitor.mqh>
#include <MQL5Book/TradeFilter.mqh>

class DealFilter: public TradeFilter<DealMonitor,
    ENUM_DEAL_PROPERTY_INTEGER,
    ENUM_DEAL_PROPERTY_DOUBLE,
    ENUM_DEAL_PROPERTY_STRING>
{
protected:
    virtual int total() const override
    {
        return HistoryDealsTotal();
    }
    virtual ulong get(const int i) const override
    {
        return HistoryDealGetTicket(i);
    }
};

```

作为使用历史的通用示例，我们来考虑仓位历史恢复脚本 *TradeHistoryPrint.mq5*。

TradeHistoryPrint

该脚本可为当前图表交易品种构建历史。

我们首先需要交易和订单的筛选器。

```

#include <MQL5Book/OrderFilter.mqh>
#include <MQL5Book/DealFilter.mqh>

```

我们将从交易中提取仓位 ID，并基于仓位 ID 来请求订单的详细信息。

可以查看整个历史，也可以查看特定仓位的历史，为此，我们将提供一个模式选择和一个输入字段，用作输入变量中的标识符。

```

enum SELECTOR_TYPE
{
    TOTAL,    // Whole history
    POSITION,  // Position ID
};

input SELECTOR_TYPE Type = TOTAL;
input ulong PositionID = 0; // Position ID

```

必须要记住，对久远的账户历史进行采样可能是一种开销，因此我们希望在处理的 EA 交易中提供对所获得的历史处理结果以及最后处理时间戳的缓存。后续每次分析历史订单时，可以不从头开始，而是从一个记忆中的时刻开始这个过程。

为了以视觉上吸引人的方式显示以列对齐的历史信息，将其表示为一个结构数组是有意义的。但是，我们的筛选器已经支持查询存储在特殊结构中的数据，也就是元组。因此，我们可应用一个技巧：描述我们的应用程序结构体，并观察元组的规则：

- 第一个字段的名称必须为 `_1`；该名称可在排序算法中选择性使用。
- 返回字段数量的 `size` 函数必须在结构体中说明。

- 该结构体应具有一个模板方法 *assign*，用于根据 *MonitorInterface* 派生的已传递监视器对象的特性填充字段。

在标准元组中，*assign* 方法的说明如下：

```
template<typename M>
void assign(const int &properties[], M &m);
```

作为第一个参数，可接收一个数组，其特性 ID 对应于我们关注的字段。事实上，这是由调用代码传递给筛选器的 *select* 方法 (*TradeFilter::select*) 的数组，然后通过引用传递给 *assign*。但是，由于我们现在创建的不是一些标准元组，而是我们自己的结构体，这些结构“知道”其字段的应用特性，我们可以将带有特性标识符的数组留在结构本身内，而不是在筛选器与同一结构体的 *assign* 方法之间来回传递。

具体来说，为了请求交易，我们用 8 个字段对 *DealTuple* 结构体进行了说明。其标识符将在 *fields* 静态数组中指定。

```
struct DealTuple
{
    datetime _1;    // deal time
    ulong deal;    // deal ticket
    ulong order;    // order ticket
    string type;    // ENUM_DEAL_TYPE as string
    string in_out;  // ENUM_DEAL_ENTRY as string
    double volume;
    double price;
    double profit;

    static int size() { return 8; }; // number of properties
    static const int fields[]; // identifiers of the requested deal properties
    ...
};

static const int DealTuple::fields[] =
{
    DEAL_TIME, DEAL_TICKET, DEAL_ORDER, DEAL_TYPE,
    DEAL_ENTRY, DEAL_VOLUME, DEAL_PRICE, DEAL_PROFIT
};
```

这种方法将标识符和字段放在一起，将相应的值存储在一个地方，这样更容易理解和维护源代码。

用特性值填充字段需要稍微修改（简化）一下 *assign* 方法，该方法用于从 *fields* 数组（而不是从输入参数）获取 ID。

```

struct DealTuple
{
    ...
    template<typename M> // M is derived from MonitorInterface<>
    void assign(M &m)
    {
        static const int DEAL_TYPE_ = StringLen("DEAL_TYPE_");
        static const int DEAL_ENTRY_ = StringLen("DEAL_ENTRY_");
        static const ulong L = 0; // default type declaration (dummy)

        _l = (datetime)m.get(fields[0], L);
        deal = m.get(fields[1], deal);
        order = m.get(fields[2], order);
        const ENUM_DEAL_TYPE t = (ENUM_DEAL_TYPE)m.get(fields[3], L);
        type = StringSubstr(EnumToString(t), DEAL_TYPE_);
        const ENUM_DEAL_ENTRY e = (ENUM_DEAL_ENTRY)m.get(fields[4], L);
        in_out = StringSubstr(EnumToString(e), DEAL_ENTRY_);
        volume = m.get(fields[5], volume);
        price = m.get(fields[6], price);
        profit = m.get(fields[7], profit);
    }
};

```

同时，我们将 `ENUM_DEAL_TYPE` 和 `ENUM_DEAL_ENTRY` 枚举的数字元素转换为用户友好的字符串。当然，这只在日志记录时需要。对于编程分析，这些类型应保持原样。

因为我们已经在其元组中开发了一个新版本的 `assign` 方法，所以你需要在 `TradeFilter` 类中为其添加一个新版本的 `select` 方法。这项创新肯定会对其他程序有用，因此我们可将其直接引入 `TradeFilter`，而不是引入某个新的派生类。

```

template<typename T,typename I,typename D,typename S>
class TradeFilter
{
    ...
    template<typename U> // U must have first field _1 and method assign(T)
    bool select(U &data[], const bool sort = false) const
    {
        const int n = total();
        // loop through the elements
        for(int i = 0; i < n; ++i)
        {
            const ulong t = get(i);
            // read properties through the monitor object
            T m(t);
            // check all filtering conditions
            if(match(m, longs)
            && match(m, doubles)
            && match(m, strings))
            {
                // for a suitable object, add its properties to an array
                const int k = EXPAND(data);
                data[k].assign(m);
            }
        }

        if(sort)
        {
            static const U u;
            sortTuple(data, u._1);
        }

        return true;
    }
}

```

注意，在用特定类型的代码调用模板方法之前，编译器并不会实现所有模板方法。因此，*TradeFilter* 中存在这种模式并不意味着必须包含任何元组头文件或必须描述类似的结构（除非你要使用）。

因此，如果在更早的时候，要使用标准元组选择交易，我们必须这样编写：

```

#include <MQL5Book/Tuples.mqh>
...
DealFilter filter;
int properties[] =
{
    DEAL_TIME, DEAL_TICKET, DEAL_ORDER, DEAL_TYPE,
    DEAL_ENTRY, DEAL_VOLUME, DEAL_PRICE, DEAL_PROFIT
};
Tuple8<ulong,ulong,ulong,ulong,ulong,double,double,double> tuples[];
filter.let(DEAL_SYMBOL, _Symbol).select(properties, tuples);

```

有了定制的结构体，一切都简单多了：

```
DealFilter filter;
DealTuple tuples[];
filter.let(DEAL_SYMBOL, _Symbol).select(tuples);
```

类似于 *DealTuple* 结构体，我们来描述一下 *OrderTuple* 的 10 字段结构体。

```
struct OrderTuple
{
    ulong _1;          // ticket (also used as 'ulong' prototype)
    datetime setup;
    datetime done;
    string type;
    double volume;
    double open;
    double current;
    double sl;
    double tp;
    string comment;

    static int size() { return 10; }; // number of properties
    static const int fields[]; // identifiers of requested order properties

    template<typename M> // M is derived from MonitorInterface<>
    void assign(M &m)
    {
        static const int ORDER_TYPE_ = StringLen("ORDER_TYPE_");

        _1 = m.get(fields[0], _1);
        setup = (datetime)m.get(fields[1], _1);
        done = (datetime)m.get(fields[2], _1);
        const ENUM_ORDER_TYPE t = (ENUM_ORDER_TYPE)m.get(fields[3], _1);
        type = StringSubstr(EnumToString(t), ORDER_TYPE_);
        volume = m.get(fields[4], volume);
        open = m.get(fields[5], open);
        current = m.get(fields[6], current);
        sl = m.get(fields[7], sl);
        tp = m.get(fields[8], tp);
        comment = m.get(fields[9], comment);
    }
};

static const int OrderTuple::fields[] =
{
    ORDER_TICKET, ORDER_TIME_SETUP, ORDER_TIME_DONE, ORDER_TYPE, ORDER_VOLUME_INITIAL,
    ORDER_PRICE_OPEN, ORDER_PRICE_CURRENT, ORDER_SL, ORDER_TP, ORDER_COMMENT
};
```

现在，实现脚本的主要函数 *-OnStart* 准备就绪。首先，我们将描述交易和订单筛选器的对象。


```

void OnStart()
{
    DealFilter filter;
    HistoryOrderFilter subfilter;
    ...

```

根据输入变量，我们可选择整个历史或特定仓位。

```

if(PositionID == 0 || Type == TOTAL)
{
    HistorySelect(0, LONG_MAX);
}
else if(Type == POSITION)
{
    HistorySelectByPosition(PositionID);
}
...

```

接下来，我们将在一个数组中收集所有仓位标识符，或者保留一个由用户指定的标识符。

```

ulong positions[];
if(PositionID == 0)
{
    ulong tickets[];
    filter.let(DEAL_SYMBOL, _Symbol)
        .select(DEAL_POSITION_ID, tickets, positions, true); // true - sorting
    ArrayUnique(positions);
}
else
{
    PUSH(positions, PositionID);
}

const int n = ArraySize(positions);
Print("Positions total: ", n);
if(n == 0) return;
...

```

辅助函数 *ArrayUnique* 可在数组中留下不重复的元素。前提是对源数组进行排序。

此外，在遍历仓位过程中，我们可请求与每个仓位相关的交易和订单。交易按 *DealTuple* 结构体的第一个字段进行排序，即按时间排序。也许最有趣的是计算仓位的盈利/亏损。为此，我们对所有交易的 *profit* 字段的值进行求和。

```

for(int i = 0; i < n; ++i)
{
    DealTuple deals[];
    filter.let(DEAL_POSITION_ID, positions[i]).select(deals, true);
    const int m = ArraySize(deals);
    if(m == 0)
    {
        Print("Wrong position ID: ", positions[i]);
        break; // invalid id set by user
    }
    double profit = 0; // TODO: need to take into account commissions, swaps and fee
    for(int j = 0; j < m; ++j) profit += deals[j].profit;
    PrintFormat("Position: % 8d %16lld Profit:%f", i + 1, positions[i], (profit));
    ArrayPrint(deals);

    Print("Order details:");
    OrderTuple orders[];
    subfilter.let(ORDER_POSITION_ID, positions[i], IS::OR_EQUAL)
        .let(ORDER_POSITION_BY_ID, positions[i], IS::OR_EQUAL)
        .select(orders);
    ArrayPrint(orders);
}
}

```

此代码不会分析交易特性中的佣金 (DEAL_COMMISSION)、掉期 (DEAL_SWAP) 和费用 (DEAL_FEE)。在真正的 EA 交易中, 可能需要进行这些分析 (取决于策略的需求)。我们将在 [测试多币种 EA 交易](#) 一节中看到交易历史分析的另一个示例, 到时我们会考虑这一要求。

你可以将脚本的结果与终端中的历史选项卡上的表进行比较: 其“盈利”列显示了每个仓位的净利润 (掉期、佣金和费用在相邻的列内, 但需要手动将其包括在内)。

务必要注意的是, 仅当在设置中选择了整个历史时, ORDER_TYPE_CLOSE_BY 类型的订单才会在两个仓位中显示。如果选择了一个特定的仓位, 系统只会将这样的订单计入其中一个仓位 (在交易请求中首先指定的仓位, 在 *position* 字段内), 而不会计入第二个仓位 (在 *position_by* 中指定)。

以下是一个历史较短的交易品种脚本结果示例。

```

Positions total: 3
Position: 1 1253500309 Profit:238.150000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.02.04 17:34:57 1236049891 1253500309 "BUY" "IN" 1.00000 76.23900 0.00000
[1] 2022.02.14 16:28:41 1242295527 1259788704 "SELL" "OUT" 1.00000 76.42100 238.15000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1253500309 2022.02.04 17:34:57 2022.02.04 17:34:57 "BUY" 1.00000 76.23900 76.2390
» 0.00 0.00 ""
[1] 1259788704 2022.02.14 16:28:41 2022.02.14 16:28:41 "SELL" 1.00000 76.42100 76.421
» 0.00 0.00 ""
Position: 2 1253526613 Profit:878.030000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.02.07 10:00:00 1236611994 1253526613 "BUY" "IN" 1.00000 75.75000 0.00000
[1] 2022.02.14 16:28:40 1242295517 1259788693 "SELL" "OUT" 1.00000 76.42100 878.03000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1253526613 2022.02.04 17:55:18 2022.02.07 10:00:00 "BUY_LIMIT" 1.00000 75.75000 7
» 0.00 0.00 ""
[1] 1259788693 2022.02.14 16:28:40 2022.02.14 16:28:40 "SELL" 1.00000 76.42100 76.421
» 0.00 0.00 ""
Position: 3 1256280710 Profit:4449.040000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.02.09 13:17:52 1238797056 1256280710 "BUY" "IN" 2.00000 74.72100 0.00000
[1] 2022.02.14 16:28:39 1242295509 1259788685 "SELL" "OUT" 2.00000 76.42100 4449.0400
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1256280710 2022.02.09 13:17:52 2022.02.09 13:17:52 "BUY" 2.00000 74.72100 74.7210
» 0.00 0.00 ""
[1] 1259788685 2022.02.14 16:28:39 2022.02.14 16:28:39 "SELL" 2.00000 76.42100 76.421
» 0.00 0.00 ""

```

在净额结算账户上增加仓位（两笔“IN”交易）及其冲销（一笔较大数量的“NOUT”交易）的情况如下片段所示。

```

Position: 5 219087383 Profit:0.170000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.03.29 08:03:33 215612450 219087383 "BUY" "IN" 0.01000 1.10011 0.00000
[1] 2022.03.29 8:04:05 215612451 219087393 "BUY" "IN" 0.01000 1.10009 0.00000
[2] 2022.03.29 08:04:29 215612457 219087400 "SELL" "INOUT" 0.03000 1.10018 0.16000
[3] 2022.03.29 08:04:34 215612460 219087403 "BUY" "OUT" 0.01000 1.10017 0.01000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 219087383 2022.03.29 08:03:33 2022.03.29 08:03:33 "BUY" 0.01000 0.0000 1.10011 »
» 0.00 0.00 ""
[1] 219087393 2022.03.29 8:04:05 2022.03.29 8:04:05 "BUY" 0.01000 0.0000 1.10009 »
» 0.00 0.00 ""
[2] 219087400 2022.03.29 08:04:29 2022.03.29 08:04:29 "SELL" 0.03000 0.0000 1.10018 »
» 0.00 0.00 ""
[3] 219087403 2022.03.29 8:04:34 2022.03.29 8:04:34 "BUY" 0.01000 0.0000 1.10017 »
» 0.00 0.00 ""

```

我们将考虑部分历史，以对冲账户反向平仓的特定仓位为例。首先可以单独查看第一个仓位：PositionID=1276109280。无论 *Type* 输入参数如何，该仓位都将完整显示。

```

Positions total: 1
Position: 1 1276109280 Profit:-0.040000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.03.07 12:20:53 1258725455 1276109280 "BUY" "IN" 0.01000 1.08344 0.00000
[1] 2022.03.07 12:20:58 1258725503 1276109328 "SELL" "OUT_BY" 0.01000 1.08340 -0.04000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1276109280 2022.03.07 12:20:53 2022.03.07 12:20:53 "BUY" 0.01000 1.08344 1.08344
» 0.00 0.00 ""
[1] 1276109328 2022.03.07 12:20:58 2022.03.07 12:20:58 "CLOSE_BY" 0.01000 1.08340 1.0
» 0.00 0.00 "#1276109280 by #1276109283"

```

还可以看到第二个仓位：PositionID=1276109283。但是，如果 *Type* 等于 “*position*”，为了选择历史的片段，可使用 *HistorySelectByPosition* 函数，结果是只有一个退出订单（尽管事实上有两个交易）。

```

Positions total: 1
Position: 1 1276109283 Profit:0.000000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.03.07 12:20:53 1258725458 1276109283 "SELL" "IN" 0.01000 1.08340 0.00000
[1] 2022.03.07 12:20:58 1258725504 1276109328 "BUY" "OUT_BY" 0.01000 1.08344 0.00000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1276109283 2022.03.07 12:20:53 2022.03.07 12:20:53 "SELL" 0.01000 1.08340 1.08340
» 0.00 0.00 ""

```

如果我们将 *Type* 设置为 “整个历史”，将会出现一个 “CLOSE_BY” 订单。

```

Positions total: 1
Position: 1 1276109283 Profit:0.000000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.03.07 12:20:53 1258725458 1276109283 "SELL" "IN" 0.01000 1.08340 0.00000
[1] 2022.03.07 12:20:58 1258725504 1276109328 "BUY" "OUT_BY" 0.01000 1.08344 0.00000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1276109283 2022.03.07 12:20:53 2022.03.07 12:20:53 "SELL" 0.01000 1.08340 1.08340
» 0.00 0.00 ""
[1] 1276109328 2022.03.07 12:20:58 2022.03.07 12:20:58 "CLOSE_BY" 0.01000 1.08340 1.0
» 0.00 0.00 "#1276109280 by #1276109283"

```

使用这样的设置，可完整选择历史，但是筛选器只留下那些在 `ORDER_POSITION_ID` 或 `ORDER_POSITION_BY_ID` 特性中找到指定仓位标识符的订单。为了用逻辑 `OR` 构成条件，在 *TradeFilter* 类中添加了 `IS::OR_EQUAL` 元素。你可以另外研究一下。

6.4.33 交易事务类型

除了执行交易操作，MQL 程序还可以响应交易事件。务必要注意的是，此类事件的发生不仅是程序操作的结果，还有其他原因，例如，当用户手动管理或在服务器上执行自动操作时（激活挂单、*Stop Loss*、*Take Profit*、*Stop Out*、仓位转移到新的一天、从账户存取款等）。

不管操作的发起者是谁，都会导致账户上交易事务的执行。交易事务是一些不可见的步骤，包括：

- ①处理交易请求
- ①更改活动订单列表（包括添加新订单、执行和删除已触发订单）
- ①更改订单历史
- ①更改交易历史
- ①更改仓位

根据操作的特性，有些步骤可能是可选的。例如，修改一个仓位的保护水平将错过三个中间点。当发出买入订单时，市场将经历一个完整的周期：处理请求，为账户创建相应的订单，执行订单，将其从活动列表中删除，添加到订单历史中，然后将相应的交易添加到历史中，并创建新的仓位。所有这些操作都是交易事务。

要接收有关此类事件的通知，应在 EA 交易或指标中说明特殊的 *OnTradeTransaction* 处理程序函数。我们将在下一节详细讨论。事实上，其中的一个参数，第一个也是最重要的参数，具有预定义结构体 *MqlTradeTransaction* 的类型。所以我们先来谈谈事务本身。

```

struct MqlTradeTransaction
{
    ulong          deal;           // Deal ticket
    ulong          order;          // Order ticket
    string         symbol;         // Name of the trading instrument
    ENUM_TRADE_TRANSACTION_TYPE type; // Trade transaction type
    ENUM_ORDER_TYPE order_type;    // Order type
    ENUM_ORDER_STATE order_state;  // Order state
    ENUM_DEAL_TYPE deal_type;      // Deal type
    ENUM_ORDER_TYPE_TIME time_type; // Order type by duration
    datetime       time_expiration; // Order expiration date
    double         price;          // Price
    double         price_trigger;  // Stop limit order trigger price
    double         price_sl;       // Stop Loss Level
    double         price_tp;       // Take Profit Level
    double         volume;         // Volume in lots
    ulong          position;       // Position ticket
    ulong          position_by;    // Opposite position ticket
};

```

下表对每个结构体字段进行说明。

字段	说明
deal	交易票
order	订单号
symbol	发生事务的交易金融工具名称
type	交易事务类型 ENUM_TRADE_TRANSACTION_TYPE (见下文)
order_type	订单类型 ENUM_ORDER_TYPE
order_state	订单状态 ENUM_ORDER_STATE
deal_type	交易类型 ENUM_DEAL_TYPE
time_type	按到期日期分类的订单类型 ENUM_ORDER_TYPE_TIME
time_expiration	挂单到期日期
price	订单、交易或仓位的价格，具体取决于事务
price_trigger	限制止损订单的止损价格 (触发价格)
price_sl	Stop Loss 价格；可能指订单、交易或仓位，具体取决于事务
price_tp	Take Profit 价格；可能指订单、交易或仓位，具体取决于事务
volume	每手交易量；可能指示订单、交易或仓位的当前交易量，具体取决于事务
position	受事务影响的仓位订单号

字段	说明
position_by	反向仓位订单号

有些字段仅在特定情况下有意义。具体来说，*time_expiration* 字段是为 *time_type* 等于 ORDER_TIME_SPECIFIED 或 ORDER_TIME_SPECIFIED_DAY 到期类型的订单填充的。*price_trigger* 字段仅用于限价止损订单 (ORDER_TYPE_BUY_STOP_LIMIT 和 ORDER_TYPE_SELL_STOP_LIMIT)。

同样显而易见的是，仓位修改在仓位订单号 (*position* 字段) 上进行操作，但不使用订单或交易订单号。此外，*position_by* 字段专门保留用于平仓对冲仓位，也就是为同一工具但方向相反而打开的仓位。

事务分析的定义特性是其类型 (*type* 字段)。为了进行说明，MQL5 API 引入了一个特殊的枚举 ENUM_TRADE_TRANSACTION_TYPE，其包含所有可能的事务类型。

标识符	说明
TRADE_TRANSACTION_ORDER_ADD	添加新订单
TRADE_TRANSACTION_ORDER_UPDATE	更改活动订单
TRADE_TRANSACTION_ORDER_DELETE	删除活动订单
TRADE_TRANSACTION_DEAL_ADD	将交易添加至历史
TRADE_TRANSACTION_DEAL_UPDATE	在历史中更改交易
TRADE_TRANSACTION_DEAL_DELETE	从历史中删除交易
TRADE_TRANSACTION_HISTORY_ADD	将订单作为执行或取消的结果添加到历史
TRADE_TRANSACTION_HISTORY_UPDATE	在历史中更改订单
TRADE_TRANSACTION_HISTORY_DELETE	从历史中删除订单
TRADE_TRANSACTION_POSITION	更改仓位
TRADE_TRANSACTION_REQUEST	通知服务器已经处理了交易请求，并且已经收到其处理结果

我们提供一些解释说明。

在 TRADE_TRANSACTION_ORDER_UPDATE 类型的事务中，订单变化不仅包括客户端或交易服务器方面的显式变化，还包括其状态的变化 (例如，从状态 ORDER_STATE_STARTED 到 ORDER_STATE_PLACED 或从 ORDER_STATE_PLACED 到 ORDER_STATE_PARTIAL 等的变化)。

在 TRADE_TRANSACTION_ORDER_DELETE 事务期间，订单可以作为服务器上相应的显式请求或执行 (填充) 的结果而删除。在这两种情况下，该订单均会被传送到历史，并且交易 TRADE_TRANSACTION_HISTORY_ADD 也必须发生。

TRADE_TRANSACTION_DEAL_ADD 交易不仅是订单执行的结果，也是账户余额事务的结果。

有些事务 (如 `TRADE_TRANSACTION_DEAL_UPDATE`、`TRADE_TRANSACTION_DEAL_DELETE` 和 `TRADE_TRANSACTION_HISTORY_DELETE`) 非常罕见, 因为它们说明了历史中的交易或订单在服务器上被追溯更改或删除的情况。通常, 这是与外部交易系统 (交易所) 同步的结果。

需要注意的是, 增加或清算仓位并不会导致出现 `TRADE_TRANSACTION_POSITION` 事务。这种类型的事务会通知交易服务器端的仓位已被用户手动更改或者被程序更改。具体来说, 仓位可以经历交易量 (部分反向平仓、冲销)、开盘价以及 *Stop Loss* 和 *Take Profit* 水平的变化。某些操作, 如重新填写, 不会触发此事件。

MQL 程序发出的所有交易请求均反映在 `TRADE_TRANSACTION_REQUEST` 交易中, 这允许以延迟的方式分析其执行。这种情况在使用 *OrderSendAsync* 函数时尤其重要, 该函数会立即将控制权返回给调用代码, 因此结果是未知的。同时, 在使用同步 *OrderSend* 函数时, 会以相同的方式生成事务。

此外, 使用 `TRADE_TRANSACTION_REQUEST` 事务, 可以从终端界面分析用户的交易行为。

6.4.34 OnTradeTransaction 事件

如果 EA 交易和指标的代码包含 *OnTradeTransaction* 特殊处理函数, 他们可以收到关于交易事件的通知。

```
void OnTradeTransaction(const MqlTradeTransaction &trans,
    const MqlTradeRequest &request, const MqlTradeResult &result)
```

第一个参数为 *MqlTradeTransaction* 结构体, 在 上一节中介绍过。第二个和第三个参数为 *MqlTradeRequest* 和 *MqlTradeResult* 结构体, 在前面的相关章节中介绍过。

MqlTradeTransaction 结构体可描述交易, 其填充方式根据 *type* 字段中指定的交易类型而不同。例如, 对于 `TRADE_TRANSACTION_REQUEST` 类型的交易, 所有其他字段都不重要, 为了获得附加信息, 需要分析该函数的第二个和第三个参数 (*request* 和 *result*)。但对于所有其他类型的交易, 应忽略该函数的最后两个参数。

在 `TRADE_TRANSACTION_REQUEST` 的情况下, *result* 变量中的 *request_id* 字段包含一个标识符 (通过序列号), 交易 *request* 在终端中以该标识符注册。该编号与订单和交易订单号以及仓位标识符无关。在与终端的每次会话期间, 编号从头开始 (从 1 开始)。请求标识符的存在允许你将执行的动作 (调用 *OrderSend* 或 *OrderSendAsync* 函数) 与传递给 *OnTradeTransaction* 的动作结果相关联。我们稍后会看示例。

对于与活动订单 (`TRADE_TRANSACTION_ORDER_ADD`、`TRADE_TRANSACTION_ORDER_UPDATE` 和 `TRADE_TRANSACTION_ORDER_DELETE`) 及其订单历史 (`TRADE_TRANSACTION_HISTORY_ADD`、`TRADE_TRANSACTION_HISTORY_UPDATE`、`TRADE_TRANSACTION_HISTORY_DELETE`) 相关的交易, 在 *MqlTradeTransaction* 结构体中填写以下字段:

- ① order - 订单号
- ② symbol - 订单中的金融工具名称
- ③ type - 交易类型
- ④ order_type - 订单类型
- ⑤ orders_state - 当前订单状态
- ⑥ time_type - 订单到期类型
- ⑦ time_expiration - 订单到期时间 (适用于到期类型为 `ORDER_TIME_SPECIFIED` 和 `ORDER_TIME_SPECIFIED_DAY` 的订单)

- ①price - 客户/程序指定的订单价格
- ①Price_trigger - 触发限价止损订单的止损价格 (仅适用于 ORDER_TYPE_BUY_STOP_LIMIT 和 ORDER_TYPE_SELL_STOP_LIMIT)
- ①price_sl - **Stop Loss** 订单价格 (如果订单中指定则成交)
- ①price_tp - **Take Profit** 订单价格 (如果订单中指定则成交)
- ①volume - 当前订单交易量 (未执行), 初始订单交易可从订单历史中找到
- ①position - 未平仓、已修改或已平仓的订单号
- ①Position_by - 反向仓位订单号 (仅用于反向仓位平仓的订单)

对于与交易相关的事务 (TRADE_TRANSACTION_DEAL_ADD、TRADE_TRANSACTION_DEAL_UPDATE 和 TRADE_TRANSACTION_DEAL_DELETE), 在 *MqlTradeTransaction* 结构体中填写以下字段:

- ①Deal - 交易订单号
- ①order - 作为交易成交依据的订单号
- ①symbol - 交易的金融工具名称
- ①type - 交易类型
- ①deal_type - 交易类型
- ①price - 交易价格
- ①price_sl - **Stop Loss** 价格 (如果在作为交易依据的订单中指定则成交)
- ①price_tp - **Take Profit** 价格 (如果在作为交易依据的订单中指定则成交)
- ①volume - 交易量
- ①position - 未平仓、已修改或已平仓的订单号
- ①position_by - 反向仓位订单号 (对于反向仓位平仓交易)

对于与仓位变更相关的交易 (TRADE_TRANSACTION_POSITION), 在 *MqlTradeTransaction* 结构体中填写以下字段:

- ①symbol - 仓位的金融工具名称
- ①type - 交易类型
- ①deal_type - 仓位类型 (DEAL_TYPE_BUY 或 DEAL_TYPE_SELL)
- ①price - 加权平均开盘价
- ①price_sl - **Stop Loss** 价格
- ①price_tp - **Take Profit** 价格
- ①volume - 每手仓位交易量
- ①position - 仓位订单号

并非所有关于订单、交易和仓位的可用信息 (例如, 注释) 都会在交易的说明中传输。有关更多信息, 请使用相关函数: *OrderGet*、*HistoryOrderGet*、*HistoryDealGet* 和 *PositionGet*。

从终端手动或通过交易函数 *OrderSend*/*OrderSendAsync* 发送的一个交易请求可以在交易服务器上生成几个连续的交易。同时, 不能保证这些交易的通知到达终端的顺序, 因此你不能依赖特定交易事务的等待顺序来构建交易算法。

交易事件是异步处理的, 也就是说, 相对于生成时刻有所延迟。每个交易事件都会被发送到 MQL 程序的队列中, 程序会按照队列顺序依次选取它们。

当 EA 交易在 *OnTradeTransaction* 处理器内处理交易时，终端会继续接受传入的交易。因此，在 *OnTradeTransaction* 运行时，交易账户的状态可能会发生变化。将来，程序将按照事件出现的顺序得到所有这些事件的通知。

交易队列的长度为 1024 个元素。如果 *OnTradeTransaction* 处理下一个交易的时间过长，队列中的旧交易可能会被新交易取代。

由于终端与交易对象的并行多线程操作，在调用 *OnTradeTransaction* 处理程序时，其中提到的所有实体，包括订单、交易和仓位，可能已经处于与交易特性中指定的不同状态。要获得它们的当前状态，应在当前环境或历史中选择它们，并使用适当的 MQL5 函数请求其特性。

我们从一个简单的 EA 交易示例 *TradeTransactions.mq5* 开始，其记录了所有的 *OnTradeTransaction* 交易事件。其唯一参数 *DetailedLog* 允许你有选择地使用 *OrderMonitor*、*DealMonitor* 和 *PositionMonitor* 类来显示所有特性。默认情况下，EA 交易只显示 *MqlTradeTransaction*、*MqlTradeRequest* 和 *MqlTradeResult* 结构体的填充字段内容，这些内容以参数形式到达处理程序；同时，只为 `TRADE_TRANSACTION_REQUEST` 交易处理 *request* 和 *result*。

```

input bool DetailedLog = false; // DetailedLog ('true' shows order/deal/position deta

void OnTradeTransaction(const MqlTradeTransaction &transaction,
    const MqlTradeRequest &request,
    const MqlTradeResult &result)
{
    static ulong count = 0;
    PrintFormat(">>>% 6d", ++count);
    Print(TU::StringOf(transaction));

    if(transaction.type == TRADE_TRANSACTION_REQUEST)
    {
        Print(TU::StringOf(request));
        Print(TU::StringOf(result));
    }

    if(DetailedLog)
    {
        if(transaction.order != 0)
        {
            OrderMonitor m(transaction.order);
            m.print();
        }
        if(transaction.deal != 0)
        {
            DealMonitor m(transaction.deal);
            m.print();
        }
        if(transaction.position != 0)
        {
            PositionMonitor m(transaction.position);
            m.print();
        }
    }
}

```

让我们在 EURUSD 图表上运行该算法，并手动执行几个操作，相应的条目将出现在日志中（为了实验的纯粹性，假设没有人和其他程序对交易账户执行操作，特别是，没有其他 EA 交易在运行）。

我们用最小手数开一个多头仓位。

```

>>> 1
TRADE_TRANSACTION_ORDER_ADD, #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD
» @ 1.10947, V=0.01
>>> 2
TRADE_TRANSACTION_DEAL_ADD, D=1279627746(DEAL_TYPE_BUY), »
» #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10947, V=0.01, P=1296
>>> 3
TRADE_TRANSACTION_ORDER_DELETE, #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURU
» @ 1.10947, P=1296991463
>>> 4
TRADE_TRANSACTION_HISTORY_ADD, #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURUS
» @ 1.10947, P=1296991463
>>> 5
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10947, #=12
DONE, D=1279627746, #=1296991463, V=0.01, @ 1.10947, Bid=1.10947, Ask=1.10947, Req=7

```

我们将卖出最低手数的两倍。

```

>>> 6
TRADE_TRANSACTION_ORDER_ADD, #=1296992157(ORDER_TYPE_SELL/ORDER_STATE_STARTED), EURUS
» @ 1.10964, V=0.02
>>> 7
TRADE_TRANSACTION_DEAL_ADD, D=1279628463(DEAL_TYPE_SELL), »
» #=1296992157(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10964, V=0.02, P=1296
>>> 8
TRADE_TRANSACTION_ORDER_DELETE, #=1296992157(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EUR
» @ 1.10964, P=1296992157
>>> 9
TRADE_TRANSACTION_HISTORY_ADD, #=1296992157(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EURU
» @ 1.10964, P=1296992157
>>> 10
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.02, ORDER_FILLING_FOK, @ 1.10964, #=1
DONE, D=1279628463, #=1296992157, V=0.02, @ 1.10964, Bid=1.10964, Ask=1.10964, Req=8

```

让我们执行计数器平仓操作。

```

>>> 11
TRADE_TRANSACTION_ORDER_ADD, #=1296992548(ORDER_TYPE_CLOSE_BY/ORDER_STATE_STARTED), E
» @ 1.10964, V=0.01, P=1296991463, b=1296992157
>>> 12
TRADE_TRANSACTION_DEAL_ADD, D=1279628878(DEAL_TYPE_SELL), »
» #=1296992548(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10964, V=0.01, P=1296
>>> 13
TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10947, P=1296991463
>>> 14
TRADE_TRANSACTION_DEAL_ADD, D=1279628879(DEAL_TYPE_BUY), »
» #=1296992548(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10947, V=0.01, P=1296
>>> 15
TRADE_TRANSACTION_ORDER_DELETE, #=1296992548(ORDER_TYPE_CLOSE_BY/ORDER_STATE_FILLED),
» @ 1.10964, P=1296991463, b=1296992157
>>> 16
TRADE_TRANSACTION_HISTORY_ADD, #=1296992548(ORDER_TYPE_CLOSE_BY/ORDER_STATE_FILLED),
» @ 1.10964, P=1296991463, b=1296992157
>>> 17
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_CLOSE_BY, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, #=129699254
» P=1296991463, b=1296992157
DONE, D=1279628878, #=1296992548, V=0.01, @ 1.10964, Bid=1.10961, Ask=1.10965, Req=9

```

我们仍然有最小手数的空头仓位。我们进行平仓。

```

>>> 18
TRADE_TRANSACTION_ORDER_ADD, #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD
» @ 1.10964, V=0.01, P=1296992157
>>> 19
TRADE_TRANSACTION_ORDER_DELETE, #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURU
» @ 1.10964, P=1296992157
>>> 20
TRADE_TRANSACTION_HISTORY_ADD, #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURUS
» @ 1.10964, P=1296992157
>>> 21
TRADE_TRANSACTION_DEAL_ADD, D=1279639132(DEAL_TYPE_BUY), »
» #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10964, V=0.01, P=1296
>>> 22
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10964, #=12
» P=1296992157
DONE, D=1279639132, #=1297002683, V=0.01, @ 1.10964, Bid=1.10964, Ask=1.10964, Req=10

```

如果你愿意，可以启用 **DetailedLog** 选项，在事件处理时记录交易对象的所有特性。在详细日志中，你能注意到存储在交易结构体中的对象状态（在其启动时）和当前状态之间的差异。例如，当添加平仓订单（反向或正常）时，在交易中指定一张订单号，根据该订单号，监控对象将不再能够读取任何内容，因为该仓位已被删除。因此，我们将在日志中看到这样的行：

```

TRADE_TRANSACTION_ORDER_ADD, #=1297777749(ORDER_TYPE_CLOSE_BY/ORDER_STATE_STARTED), E
» @ 1.10953, V=0.01, P=1297774881, b=1297776850
...
Error: PositionSelectByTicket(1297774881) failed: TRADE_POSITION_NOT_FOUND

```

我们重新启动 EA 交易 *TradeTransaction.mq5* 来重置记录的事件以便进行下一次测试。这一次我们将使用默认设置（没有详细说明）。

现在我们尝试在新的 EA 交易 *OrderSendTransaction1.mq5* 中以编程方式执行交易动作，同时在其中说明我们的 *OnTradeTransaction* 处理程序（与前面的示例相同）。

这个 EA 交易允许你选择交易方向和交易量：如果你将其设置为零，默认使用当前交易品种的最小手数。同样，在参数中，有一个以点为单位的到保护水平的距离。使用指定的参数进入市场，在设置 *Stop Loss* 和 *Take Profit* 之间有 5 秒钟的暂停，然后平仓，以便用户可以干预（例如，手动编辑止损），尽管这不是必需的，因为我们已经确保手动操作可被程序拦截。

```
enum ENUM_ORDER_TYPE_MARKET
{
    MARKET_BUY = ORDER_TYPE_BUY,    // ORDER_TYPE_BUY
    MARKET_SELL = ORDER_TYPE_SELL    // ORDER_TYPE_SELL
};

input ENUM_ORDER_TYPE_MARKET Type;
input double Volume;                // Volume (0 - minimal lot)
input uint Distance2SLTP = 1000;
```

该策略将启动一次，为此使用一个 1 秒钟的计时器，该计时器会在其自己的处理程序中关闭。

```
int OnInit()
{
    EventSetTimer(1);
    return INIT_SUCCEEDED;
}

void OnTimer()
{
    EventKillTimer();
    ...
}
```

所有操作都是通过一个我们已经熟悉的具有高级功能的 *MqlTradeRequestSync* 结构体 (*MqlTradeSync.mqh*) 来执行的：用正确的值隐式初始化字段、用于市场订单的 *buy/sell* 方法、用于保护水平的 *adjust* 以及用于平仓的 *close*。

第 1 步：

```
MqlTradeRequestSync request;

const double volume = Volume == 0 ?
    SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;

Print("Start trade");
const ulong order = (Type == MARKET_BUY ? request.buy(volume) : request.sell(volume));
if(order == 0 || !request.completed())
{
    Print("Failed Open");
    return;
}

Print("OK Open");
```

第 2 步：

```

Sleep(5000); // wait 5 seconds (user can edit position)
Print("SL/TP modification");
const double price = PositionGetDouble(POSITION_PRICE_OPEN);
const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
TU::TradeDirection dir((ENUM_ORDER_TYPE)Type);
const double SL = dir.negative(price, Distance2SLTP * point);
const double TP = dir.positive(price, Distance2SLTP * point);
if(request.adjust(SL, TP) && request.completed())
{
    Print("OK Adjust");
}
else
{
    Print("Failed Adjust");
}

```

第 3 步:

```

Sleep(5000); // wait another 5 seconds
Print("Close down");
if(request.close(request.result.position) && request.completed())
{
    Print("Finish");
}
else
{
    Print("Failed Close");
}
}

```

通过中间等待，不仅可以有时间考虑进程，还展示了 MQL5 编程的一个重要方面，即单线程。当我们的 EA 交易在 *OnTimer* 中时，终端生成的交易事件可在其队列中累积，并且只有在退出 *OnTimer* 后才会以延迟的方式转发给内部 *OnTradeTransaction* 处理程序。

同时，并行运行的 *TradeTransactions* EA 交易不会参与任何计算，并将尽快接收交易事件。

两个 EA 交易的执行结果显示在下面的日志中，并显示了时间（为简单起见，*OrderSendTransaction1* 标记为 *OS1*，*Trade Transactions* 标记为 *TTs*）。

```

19:09:08.078 OS1 Start trade
19:09:08.109 TTs >>> 1
19:09:08.125 TTs TRADE_TRANSACTION_ORDER_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STATE
EURUSD, @ 1.10913, V=0.01
19:09:08.125 TTs >>> 2
19:09:08.125 TTs TRADE_TRANSACTION_DEAL_ADD, D=1280661362(DEAL_TYPE_BUY), »
#=1298021794(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10913, V=0.01, »
P=1298021794
19:09:08.125 TTs >>> 3
19:09:08.125 TTs TRADE_TRANSACTION_ORDER_DELETE, #=1298021794(ORDER_TYPE_BUY/ORDER_ST
EURUSD, @ 1.10913, P=1298021794
19:09:08.125 TTs >>> 4
19:09:08.125 TTs TRADE_TRANSACTION_HISTORY_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STA
EURUSD, @ 1.10913, P=1298021794
19:09:08.125 TTs >>> 5
19:09:08.125 TTs TRADE_TRANSACTION_REQUEST
19:09:08.125 TTs TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK
D=10, #=1298021794, M=1234567890
19:09:08.125 TTs DONE, D=1280661362, #=1298021794, V=0.01, @ 1.10913, Bid=1.10913, As
Req=9
19:09:08.125 OS1 Waiting for position for deal D=1280661362
19:09:08.125 OS1 OK Open
19:09:13.133 OS1 SL/TP modification
19:09:13.164 TTs >>> 6
19:09:13.164 TTs TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10913, SL=1.09913, TP=1.1191
P=1298021794
19:09:13.164 OS1 OK Adjust
19:09:13.164 TTs >>> 7
19:09:13.164 TTs TRADE_TRANSACTION_REQUEST
19:09:13.164 TTs TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK
TP=1.11913, D=10, P=1298021794, M=1234567890
19:09:13.164 TTs DONE, Req=10
19:09:18.171 OS1 Close down
19:09:18.187 OS1 Finish
19:09:18.218 TTs >>> 8
19:09:18.218 TTs TRADE_TRANSACTION_ORDER_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_STAT
EURUSD, @ 1.10901, V=0.01, P=1298021794
19:09:18.218 TTs >>> 9
19:09:18.218 TTs TRADE_TRANSACTION_DEAL_ADD, D=1280661967(DEAL_TYPE_SELL), »
#=1298022443(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10901, »
SL=1.09913, TP=1.11913, V=0.01, P=1298021794
19:09:18.218 TTs >>> 10
19:09:18.218 TTs TRADE_TRANSACTION_ORDER_DELETE, #=1298022443(ORDER_TYPE_SELL/ORDER_S
EURUSD, @ 1.10901, P=1298021794
19:09:18.218 TTs >>> 11
19:09:18.218 TTs TRADE_TRANSACTION_HISTORY_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_ST
EURUSD, @ 1.10901, P=1298021794
19:09:18.218 TTs >>> 12
19:09:18.218 TTs TRADE_TRANSACTION_REQUEST
19:09:18.218 TTs TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_FO
D=10, #=1298022443, P=1298021794, M=1234567890
19:09:18.218 TTs DONE, D=1280661967, #=1298022443, V=0.01, @ 1.10901, Bid=1.10901, As
Req=11
19:09:18.218 OS1 >>> 1

```



```

19:09:18.218 OS1 TRADE_TRANSACTION_ORDER_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STATE
EURUSD, @ 1.10913, V=0.01
19:09:18.218 OS1 >>> 2
19:09:18.218 OS1 TRADE_TRANSACTION_DEAL_ADD, D=1280661362(DEAL_TYPE_BUY), »
#=1298021794(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, »
@ 1.10913, V=0.01, P=1298021794
19:09:18.218 OS1 >>> 3
19:09:18.218 OS1 TRADE_TRANSACTION_ORDER_DELETE, #=1298021794(ORDER_TYPE_BUY/ORDER_ST
EURUSD, @ 1.10913, P=1298021794
19:09:18.218 OS1 >>> 4
19:09:18.218 OS1 TRADE_TRANSACTION_HISTORY_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STA
EURUSD, @ 1.10913, P=1298021794
19:09:18.218 OS1 >>> 5
19:09:18.218 OS1 TRADE_TRANSACTION_REQUEST
19:09:18.218 OS1 TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK
D=10, #=1298021794, M=1234567890
19:09:18.218 OS1 DONE, D=1280661362, #=1298021794, V=0.01, @ 1.10913, Bid=1.10913, As
Req=9
19:09:18.218 OS1 >>> 6
19:09:18.218 OS1 TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10913, SL=1.09913, TP=1.1191
P=1298021794
19:09:18.218 OS1 >>> 7
19:09:18.218 OS1 TRADE_TRANSACTION_REQUEST
19:09:18.218 OS1 TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK
SL=1.09913, TP=1.11913, D=10, P=1298021794, M=1234567890
19:09:18.218 OS1 DONE, Req=10
19:09:18.218 OS1 >>> 8
19:09:18.218 OS1 TRADE_TRANSACTION_ORDER_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_STAT
EURUSD, @ 1.10901, V=0.01, P=1298021794
19:09:18.218 OS1 >>> 9
19:09:18.218 OS1 TRADE_TRANSACTION_DEAL_ADD, D=1280661967(DEAL_TYPE_SELL), »
#=1298022443(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10901, »
SL=1.09913, TP=1.11913, V=0.01, P=1298021794
19:09:18.218 OS1 >>> 10
19:09:18.218 OS1 TRADE_TRANSACTION_ORDER_DELETE, #=1298022443(ORDER_TYPE_SELL/ORDER_S
EURUSD, @ 1.10901, P=1298021794
19:09:18.218 OS1 >>> 11
19:09:18.218 OS1 TRADE_TRANSACTION_HISTORY_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_ST
EURUSD, @ 1.10901, P=1298021794
19:09:18.218 OS1 >>> 12
19:09:18.218 OS1 TRADE_TRANSACTION_REQUEST
19:09:18.218 OS1 TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_FO
D=10, #=1298022443, P=1298021794, M=1234567890
19:09:18.218 OS1 DONE, D=1280661967, #=1298022443, V=0.01, @ 1.10901, Bid=1.10901, As
Req=11

```

程序中事件的编号是相同的（假设它们按照建议明确地启动）。注意，相同的事件在请求执行后会立即从 *TTs* 中打印出来，第二次仅在测试结束时打印出来，实际上，所有的事件都从队列输出到 *OS1*。

如果我们消除了人为延迟，该脚本当然会运行得更快，但是 *OnTradeTransaction* 处理程序仍然会在所有三个步骤之后（多次）收到通知，而不是在每个请求之后。这有多重要？

现在，示例使用我们对 *MqlTradeRequestSync* 结构体的修改，特意使用同步选项 *OrderSend*，还实现了一个通用的 *completed* 方法，该方法用于检查请求是否成功完成。有了这种控制，我们可

以为仓位设置保护水平，因为我们知道如何等待其订单号出现。在这样一个同步概念的框架内（为了方便起见而采用），我们不需要在 `OnTradeTransaction` 中分析查询结果。但是，情况并非总是如此。

当 EA 交易需要一次发送许多请求时，例如在设置订单网格 `PendingOrderGrid2.mq5` 的示例中，如 [仓位特性](#) 一节所述，等待每个仓位或等待订单就绪可能会降低 EA 交易的整体性能。在这种情况下，建议使用 `OrderSendAsync` 函数。但是如果成功，该函数仅填充 `MqlTradeResult` 中的 `request_id` 字段，然后需要用其来跟踪 `OnTradeTransaction` 中订单、交易和仓位的出现。

实现这种方案的一个最明显但不够优雅的一个技巧是，在全局上下文中，将请求的标识符或发送请求的整个结构体存储在一个数组中。然后可以在 `OnTradeTransaction` 中的传入交易中查找这些标识符，可以在 `MqlTradeResult` 参数中找到其订单号，并可以采取进一步操作。因此，交易逻辑被分成不同的函数。例如，在最后一个 EA 交易 `OrderSendTransaction1.mq5` 的上下文中，这种“分散处理”体现在：在发送第一个订单后，代码片段必须被传输到 `OnTradeTransaction` 并检查以下内容：

⌚ `MqlTradeTransaction` 中的交易类型 (`transaction type`);

⌚ `MqlTradeRequest` 中的请求类型 (`request action`);

⌚ `MqlTradeResult` 中的请求 ID (`result.request_id`);

所有这些都应辅以特定的应用逻辑（例如，检查仓位是否存在），其提供了按交易策略状态的分支。稍后，我们将在不同的编号下对 `OrderSendTransaction` EA 交易进行类似的修改，以直观地显示额外源代码的数量。然后我们将提供一种更线性的程序组织方式，但不会放弃交易性事件。

目前，我们只注意到开发者选择是否应围绕 `OnTradeTransaction` 来构建算法。在许多情况下，若不需要批量发送订单，可以保持同步编程模式。但是，控制挂单和触发保护水平以及服务器生成其他事件，最实用的方法就是使用 `OnTradeTransaction`。稍作准备后，我们将呈现两个相关的示例：网格 EA 交易的最终修改版本和两个 OCO（互损订单）订单的流行设置实现（参见 [交易](#) 一节）。

若不想使用 `OnTradeTransaction`，一种替代方法是定期分析交易环境，实质上就是记住订单和仓位的数量，并寻找它们之间的变化。这种方法适用于基于时间表或允许一定时间延迟的策略。

我们再次强调，`OnTradeTransaction` 的使用并不意味着程序必须从 `OrderSend` 切换到 `OrderSendAsync`：你可以使用任意一个或同时使用两者。注意，`OrderSend` 函数也不是完全同步的，因为它最多返回订单和交易订单号，但不会返回仓位。很快，我们将能够测量同一网格策略中一批订单的执行时间，使用的两个函数变体为：`OrderSend` 和 `OrderSendAsync`。

为了统一同步和异步程序的开发，要是能在我们的结构体 `MqlTradeRequestSync` 中支持 `OrderSendAsync` 就太好了（不考虑其名称）。这可以通过几处修改来完成。首先，需要将所有当前存在的调用 `OrderSend` 替换为你自己的方法 `orderSend`，并在其中根据一个标志改为调用 `OrderSend` 或 `OrderSendAsync`。

```

struct MqlTradeRequestSync: public MqlTradeRequest
{
    ...
    static bool AsyncEnabled;
    ...
private:
    bool orderSend(const MqlTradeRequest &req, MqlTradeResult &res)
    {
        return AsyncEnabled ? ::OrderSendAsync(req, res) : ::OrderSend(req, res);
    }
};

```

通过将 *AsyncEnabled* 公共变量设置为 *true* 或 *false*, 可以从一种模式切换到另一种模式, 例如, 在发送批量订单的代码片段中。

第二, 那些返回订单号 (例如, 进场) 的结构方法应返回 *request_id* 字段, 而不是 *order*。例如, 在 *_pending* 和 *_market* 方法中, 我们有以下运算符:

```
if(OrderSend(this, result)) return result.order;
```

现在将其替换为:

```
if(orderSend(this, result)) return result.order ? result.order :
    (result.retcode == TRADE_RETCODE_PLACED ? result.request_id : 0);
```

当然, 当启用异步模式时, 我们不能再使用 *completed* 方法来等待查询结果在发送后立即准备好。但是这个方法本质上是可选的: 即使是通过 *OrderSend* 工作, 也可以放弃这个方法。

因此, 考虑到对 *MqlTradeSync.mqh* 文件的新修改, 我们来创建 *OrderSendTransaction2.mq5*。

该 EA 交易将像以前一样从 *OnTimer* 发送初始请求, 同时在 *OnTradeTransaction* 中逐步设置保护水平并平仓。虽然我们这次不会在各个阶段之间有人为延迟, 但状态的顺序本身是许多 EA 交易的标准: 开仓、修改、平仓 (如果满足某些市场条件, 这些条件此处不做赘述)。

两个全局变量将允许你跟踪状态: *RequestID* 变量包含发送的最后一个请求的 ID (我们期望的结果), *Position Ticket* 变量包含未平仓仓位订单号。当该仓位尚未出现或不再存在时, 该订单号等于 0。

```

uint RequestID = 0;
ulong PositionTicket = 0;

```

OnInit 处理程序中启用了异步模式。

```

int OnInit()
{
    ...
    MqlTradeRequestSync::AsyncEnabled = true;
    ...
}

```

OnTimer 函数现在更简短了。

```

void OnTimer()
{
    ...
    // send a request TRADE_ACTION_DEAL (asynchronously!)
    const ulong order = (Type == MARKET_BUY ? request.buy(volume) : request.sell(volume)
    if(order) // in asynchronous mode this is now request_id
    {
        Print("OK Open?");
        RequestID = request.result.request_id; // same as order
    }
    else
    {
        Print("Failed Open");
    }
}

```

成功完成请求后，我们只获得 *request_id*，并将其存储在 *RequestID* 变量中。状态打印现在包含一个问号，如 “OK Open?”，因为实际结果还不知道。

OnTradeTransaction 由于结果的验证以及根据条件执行后续交易订单，OnTradeTransaction 变得更加复杂。我们逐一来看吧。

在这种情况下，整个交易逻辑已经转移到 TRADE_TRANSACTION_REQUEST 类型的分支中。当然，如果需要的话，开发人员也可以使用其他类型，但是我们使用这种类型，因为其包含熟悉的 *MqlTradeResult* 结构体形式的信息，也就是说，这种类型表示异步调用 *OrderSendAsync* 的延迟结束。

```

void OnTradeTransaction(const MqlTradeTransaction &transaction,
    const MqlTradeRequest &request,
    const MqlTradeResult &result)
{
    static ulong count = 0;
    PrintFormat(">>>% 6d", ++count);
    Print(TU::StringOf(transaction));

    if(transaction.type == TRADE_TRANSACTION_REQUEST)
    {
        Print(TU::StringOf(request));
        Print(TU::StringOf(result));

        ...
        // here is the whole algorithm
    }
}

```

我们应只对具有我们期望的 ID 的请求感兴趣。所以下一条语句是嵌套 *if*。在这个块中，我们预先描述了 *MqlTradeRequestSync* 对象，因为根据计划需要发送常规的交易请求。

```

if(result.request_id == RequestID)
{
    MqlTradeRequestSync next;
    next.magic = Magic;
    next.deviation = Deviation;
    ...
}

```

我们只有两种工作请求类型，所以我们为其增加了另一个嵌套的 *if*。

```

if(request.action == TRADE_ACTION_DEAL)
{
    ... // here is the reaction to opening and closing a position
}
else if(request.action == TRADE_ACTION_SLTP)
{
    ... // here is the reaction to setting SLTP for an open position
}

```

请注意，`TRADE_ACTION_DEAL` 用于开仓和平仓，因此还需要一个 *if*，其中我们可根据 *PositionTicket* 变量的值来区分这两种状态。

```

if(PositionTicket == 0)
{
    ... // there is no position, so this is an opening notification
}
else
{
    ... // there is a position, so this is a closure
}

```

在所考虑的交易策略中没有增加仓位 (用于净额结算)或多个仓位 (用于对冲)，这就是为什么这一部分在逻辑上很简单的原因。真正的 EA 交易需要更多不同的中间状态估计。

接收到开仓通知时，代码块如下所示：

```

if(PositionTicket == 0)
{
    // trying to get results from the transaction: select an order by tick
    if(!HistoryOrderSelect(result.order))
    {
        Print("Can't select order in history");
        RequestID = 0;
        return;
    }
    // get position ID and ticket
    const ulong posid = HistoryOrderGetInteger(result.order, ORDER_POSITIC
    PositionTicket = TU::PositionSelectById(posid);
    ...
}

```

为了简单起见，我们此处省略了错误和重新报价检查。你可以在附加的源代码中看到它们的处理示例。回想一下，所有这些检查都已经在 *MqlTradeRequestSync* 结构体的方法中实现了，但是它们只在同步模式下工作，因此我们必须显式地重复它们。

下一个用于设置保护水平的代码片段变化不大。

```

if(PositionTicket == 0)
{
    ...
    const double price = PositionGetDouble(POSITION_PRICE_OPEN);
    const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
    TU::TradeDirection dir((ENUM_ORDER_TYPE)Type);
    const double SL = dir.negative(price, Distance2SLTP * point);
    const double TP = dir.positive(price, Distance2SLTP * point);
    // sending TRADE_ACTION_SLTP request (asynchronously!)
    if(next.adjust(PositionTicket, SL, TP))
    {
        Print("OK Adjust?");
        RequestID = next.result.request_id;
    }
    else
    {
        Print("Failed Adjust");
        RequestID = 0;
    }
}

```

此处唯一的区别是：我们用新的 TRADE_ACTION_SLTP 请求的 ID 来填充 *RequestID* 变量。

收到关于非零 *PositionTicket* 交易的通知意味着该仓位已平仓。

```

if(PositionTicket == 0)
{
    ... // see above
}
else
{
    if(!PositionSelectByTicket(PositionTicket))
    {
        Print("Finish");
        RequestID = 0;
        PositionTicket = 0;
    }
}

```

如果成功删除，则不能使用 *PositionSelectByTicket* 选择仓位，因此我们应重置 *RequestID* 和 *PositionTicket*。然后，EA 交易会返回到其初始状态，并准备进行下一个买入/卖出-修改-平仓循环。

我们仍需考虑发送平仓请求。在我们简化到最低限度的策略中，这应发生在成功修改保护水平之后。

```

if(request.action == TRADE_ACTION_DEAL)
{
    ... // see above
}
else if(request.action == TRADE_ACTION_SLTP)
{
    // send a TRADE_ACTION_DEAL request to close (asynchronously!)
    if(next.close(PositionTicket))
    {
        Print("OK Close?");
        RequestID = next.result.request_id;
    }
    else
    {
        PrintFormat("Failed Close %lld", PositionTicket);
    }
}
}

```

这就是整个 *OnTradeTransaction* 函数。EA 交易准备就绪。

我们使用 EURUSD 的默认设置运行 *OrderSendTransaction2.mq5*。以下是一个日志示例。

```

Start trade
OK Open?
>>> 1
TRADE_TRANSACTION_ORDER_ADD, #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD
» @ 1.10640, V=0.01
>>> 2
TRADE_TRANSACTION_DEAL_ADD, D=1282135720(DEAL_TYPE_BUY), »
» #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10640, V=0.01, P=1299
>>> 3
TRADE_TRANSACTION_ORDER_DELETE, #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURU
» @ 1.10640, P=1299508203
>>> 4
TRADE_TRANSACTION_HISTORY_ADD, #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURUS
» @ 1.10640, P=1299508203
>>> 5
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10640, D=10
» #=1299508203, M=1234567890
DONE, D=1282135720, #=1299508203, V=0.01, @ 1.1064, Bid=1.1064, Ask=1.1064, Req=7
OK Adjust?
>>> 6
TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10640, SL=1.09640, TP=1.11640, V=0.01, P=1299
>>> 7
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, SL=1.09640, TP=
» D=10, P=1299508203, M=1234567890
DONE, Req=8
OK Close?
>>> 8
TRADE_TRANSACTION_ORDER_ADD, #=1299508215(ORDER_TYPE_SELL/ORDER_STATE_STARTED), EURUS
» @ 1.10638, V=0.01, P=1299508203
>>> 9
TRADE_TRANSACTION_ORDER_DELETE, #=1299508215(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EUR
» @ 1.10638, P=1299508203
>>> 10
TRADE_TRANSACTION_HISTORY_ADD, #=1299508215(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EURU
» @ 1.10638, P=1299508203
>>> 11
TRADE_TRANSACTION_DEAL_ADD, D=1282135730(DEAL_TYPE_SELL), »
» #=1299508215(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10638, »
» SL=1.09640, TP=1.11640, V=0.01, P=1299508203
>>> 12
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_FOK, @ 1.10638, D=1
» #=1299508215, P=1299508203, M=1234567890
DONE, D=1282135730, #=1299508215, V=0.01, @ 1.10638, Bid=1.10638, Ask=1.10638, Req=9
Finish

```

交易逻辑按预期工作，交易事件严格按订单发送顺序到达。如果我们现在并行运行新的 EA 交易和交易拦截器 *TradeTransactions.mq5*，来自两个 EA 交易的日志消息将同步出现。

但是，从第一个直接版本 *OrderSendTransaction1.mq5* 重构为第二个异步版本 *OrderSendTransaction2.mq5* 需要更复杂的代码。问题出现了：有没有可能以某种方式将顺序说明交易逻辑（代码透明度）和并行处理（速度）的原则结合起来呢？

这在理论上是可能的，但是需要在某个节点花时间去创建某种辅助机制。

6.4.35 同步和异步请求

在学习细节之前，请注意，每个 MQL 程序都是在自己的线程中执行的，因此，交易（和其他事件）的并行异步处理之所以可能，是因为另一个 MQL 程序也会这样做。同时，要保证程序之间的信息交换。我们已经了解了几种方法，包括终端的 [全局变量](#) 和 [文件](#)。在本书的第 7 章，我们将探讨其他特性，如 [图形资源](#) 和 [数据库](#)。

实际上，设想一个类似于 *TradeTransactions.mq5* 的 EA 交易与交易中 EA 交易并行运行，并在全局变量中保存接收到的交易（不一定是所有字段，只是影响决策的可选字段）。然后，EA 交易可以在发送下一个请求后立即检查全局变量，并在不离开当前函数的情况下从中读取结果。此外，它不需要自己的 *OnTradeTransaction* 处理程序。

但是，组织第三方 EA 交易的运行并不容易。从技术角度来看，这可以通过创建一个 [图表对象](#) 并应用一个 [模板](#)（带有预定义交易监视器的 EA 交易）来实现。但是有一个更简单的方法。关键是，*OnTradeTransaction* 的事件不仅可转化为 EA 交易，还可转化为指标。反过来，指标是最容易启动的 MQL 程序类型：只需调用 *iCustom* 即可。

此外，使用指标还有一个好处：它可以说明通过 *CopyBuffer* 从外部程序获得的指标缓冲区，并在其中安排一个 *ring buffer* 来存储来自终端的交易（请求结果）。因此，没有必要弄乱全局变量。

注意！测试程序中的指标并不会生成 *OnTradeTransaction* 事件，因此你只能在线检查 EA 交易/指标对的运行情况。

让我们将这个指标称为 *TradeTransactionRelay.mq5*，并描述其中的一个缓冲区。该缓冲区可以是不可见的，因为其将写入无法呈现的数据，但我们让其保持可见，以证明这个概念。

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1

double Buffer[];

void OnInit()
{
    SetIndexBuffer(0, Buffer, INDICATOR_DATA);
}
```

OnCalculate 处理程序为空。

```
int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &price[])
{
    return rates_total;
}
```

在代码中，我们需要一个现成 [转换器](#)（*double* 和 *ulong* 的相互转换），因为如果使用简单的类型转换将缓冲区单元写入 *ulong*，它们可能会破坏大的 *ulong* 值（参见 [实数型](#)）。

```
#include <MQL5Book/ConverterT.mqh>
Converter<ulong,double> cnv;
```

以下是 *OnTradeTransaction* 函数。

```
#define FIELD_NUM 6 // the most important fields in MqlTradeResult

void OnTradeTransaction(const MqlTradeTransaction &transaction,
    const MqlTradeRequest &request,
    const MqlTradeResult &result)
{
    if(transaction.type == TRADE_TRANSACTION_REQUEST)
    {
        ArraySetAsSeries(Buffer, true);

        // store FIELD_NUM result fields into consecutive buffer cells
        const int offset = (int)((result.request_id * FIELD_NUM)
            % (Bars(_Symbol, _Period) / FIELD_NUM * FIELD_NUM));
        Buffer[offset + 1] = result.retcode;
        Buffer[offset + 2] = cnv[result.deal];
        Buffer[offset + 3] = cnv[result.order];
        Buffer[offset + 4] = result.volume;
        Buffer[offset + 5] = result.price;
        // this assignment must come last,
        // because it is the result ready flag
        Buffer[offset + 0] = result.request_id;
    }
}
```

我们决定仅保留 *MqlTradeResult* 结构体中最重要的六个字段。如果需要，可以将该机制扩展到整个结构体，但是要传递 *comment* 字符串字段，需要一个字符数组，因此必须为该数组保留相当多的元素。

因此，每个结果现在占据六个连续的缓冲区单元。这六个单元中的第一个单元的索引是根据请求 ID 确定的：将这个数字乘以 6 即可。因为可能有许多请求，所以条目根据环形缓冲区的原理工作，即，通过用余数（%）除以指示符缓冲区的大小（向上舍入到 6 的柱线数量）来规范化结果索引。当请求数量超过该大小时，记录将从初始元素开始循环。

由于柱线编号受新柱线形成的影响，建议将指标放在大的时间框架上，如 D1。然后，只有在一天开始的时候，才有可能（但概率较低）出现这样的情况：在处理下一个交易的过程中，指标中的柱线编号会直接发生变化，然后 EA 交易不会读取该指标记录的结果（可能会遗漏一个交易）。

指标准备就绪。现在让我们开始实现测试 EA 交易 *OrderSendTransaction3.mq5* 的新修改版本（这是其最新版本）。我们来介绍指标句柄的 *handle* 变量，并在 *OnInit* 中创建指标。

```

int handle = 0;

int OnInit()
{
    ...
    const static string indicator = "MQL5Book/p6/TradeTransactionRelay";
    handle = iCustom(_Symbol, PERIOD_D1, indicator);
    if(handle == INVALID_HANDLE)
    {
        Alert("Can't start indicator ", indicator);
        return INIT_FAILED;
    }
    return INIT_SUCCEEDED;
}

```

为了从指标缓冲区读取查询结果，我们来准备一个辅助函数 *AwaitAsync*。作为其第一个参数，它用于接收对 *MqlTradeRequestSync* 结构体的引用。如果成功，则从带有 *handle* 的指标缓冲区获得的结果将被写入该结构体。我们感兴趣的请求标识符已经在嵌套结构体中，即 *result.request_id* 字段中。当然，此处必须按照同样的原理来读取数据，也就是六条柱线。

```

#define FIELD_NUM 6 // the most important fields in MqlTradeResult
#define TIMEOUT 1000 // 1 second

bool AwaitAsync(MqlTradeRequestSync &r, const int _handle)
{
    Converter<ulong,double> cnv;
    const int offset = (int)((r.result.request_id * FIELD_NUM)
        % (Bars(_Symbol, _Period) / FIELD_NUM * FIELD_NUM));
    const uint start = GetTickCount();
    // wait for results or timeout
    while(!IsStopped() && GetTickCount() - start < TIMEOUT)
    {
        double array[];
        if((CopyBuffer(_handle, 0, offset, FIELD_NUM, array)) == FIELD_NUM)
        {
            ArraySetAsSeries(array, true);
            // when request_id is found, fill other fields with results
            if((uint)MathRound(array[0]) == r.result.request_id)
            {
                r.result.retcode = (uint)MathRound(array[1]);
                r.result.deal = cnv[array[2]];
                r.result.order = cnv[array[3]];
                r.result.volume = array[4];
                r.result.price = array[5];
                PrintFormat("Got Req=%d at %d ms",
                    r.result.request_id, GetTickCount() - start);
                Print(TU::StringOf(r.result));
                return true;
            }
        }
    }
    Print("Timeout for: ");
    Print(TU::StringOf(r));
    return false;
}

```

现在我们有了这个函数，让我们以异步/同步的方式编写一个交易算法：作为一个直接的步骤序列，每个步骤都会等待前一个步骤准备就绪，因为通知来自并行指标程序，同时保留在一个函数中。

```

void OnTimer()
{
    EventKillTimer();

    MqlTradeRequestSync::AsyncEnabled = true;

    MqlTradeRequestSync request;
    request.magic = Magic;
    request.deviation = Deviation;

    const double volume = Volume == 0 ?
        SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;
    ...

```

第 1 步.

```

Print("Start trade");
ResetLastError();
if((bool)(Type == MARKET_BUY ? request.buy(volume) : request.sell(volume)))
{
    Print("OK Open?");
}

if(!(AwaitAsync(request, handle) && request.completed()))
{
    Print("Failed Open");
    return;
}
...

```

第 2 步.

```

Print("SL/TP modification");
...
if(request.adjust(SL, TP))
{
    Print("OK Adjust?");
}

if(!(AwaitAsync(request, handle) && request.completed()))
{
    Print("Failed Adjust");
}

```

第 3 步.

```

Print("Close down");
if(request.close(request.result.position))
{
    Print("OK Close?");
}

if(!(AwaitAsync(request, handle) && request.completed()))
{
    Print("Failed Close");
}

Print("Finish");
}

```

请注意，现在 *completed* 方法调用不是在发送请求之后，而是在 *AwaitAsync* 函数收到结果之后。

除此之外，一切都与该算法的第一个版本非常相似，但现在起建立在异步函数调用的基础上，并可对异步事件做出反应。

在这个特定的示例中，对单个仓位的一系列操作可能看起来并不重要。但是，我们可以使用相同的技术来发送和控制一批订单。优势就显而易见了。稍后，我们将在网格 EA 交易的帮助下演示这一点，同时比较两个函数的性能：*OrderSend* 和 *OrderSendAsync*。

但是现在，当我们完成 *OrderSendTransaction* EA 交易的序列时，让我们运行最新版本，并在日志中查看所有步骤的常规线性执行。

```

Start trade
OK Open?
Got Req=1 at 62 ms
DONE, D=1282677007, #=1300045365, V=0.01, @ 1.10564, Bid=1.10564, Ask=1.10564, Order
Waiting for position for deal D=1282677007
SL/TP modification
OK Adjust?
Got Req=2 at 63 ms
DONE, Order placed, Req=2
Close down
OK Close?
Got Req=3 at 78 ms
DONE, D=1282677008, #=1300045366, V=0.01, @ 1.10564, Bid=1.10564, Ask=1.10564, Order
Finish

```

响应延迟的计时可能主要取决于服务器、时间和交易品种。当然，此处的部分时间不是花在带确认的交易请求上，而是花在 *CopyBuffer* 函数的执行上。根据我们的观察结果，该时间不会超过 16 毫秒（在标准系统计时器的一个周期内，希望使用高精度计时器 *GetMicrosecondCount* 的人可以分析该程序）。

忽略状态 (DONE) 和字符串说明 ("Order placed") 之间的差异。事实上，从 *OrderSendAsync* 函数发送注释（以及 *ask/bid* 字段）的那一刻起，注释就一直保留在该结构体中，*retcode* 字段中的最终状态由 *AwaitAsync* 函数写入。对我们而言很重要的一点是，在包含结果的结构体中，订单号 (*deal* 和 *order*)、行权价格 (*price*) 和交易量 (*volume*) 是最新的。

基于前面考虑的 *OrderSendTransaction3.mq5* 示例，让我们创建一个新版本的网格 EA 交易 *PendingOrderGrid3.mq5*（以前的版本在 [用于读取仓位特性的函数](#)一节中提供）。该 EA 交易能够

按用户选择在同步或异步模式下设置一个完整的订单网格。我们还可以检测设置完整网格的时间，以便进行比较。

该模式由输入变量 *EnableAsyncSetup* 控制。*handle* 变量被分配给指标句柄。

```
input bool EnableAsyncSetup = false;

int handle;
```

在初始化期间，在异步模式的情况下，我们可创建一个 *TradeTransactionRelay* 指标的实例。

```
int OnInit()
{
    ...
    if(EnableAsyncSetup)
    {
        const uint start = GetTickCount();
        const static string indicator = "MQL5Book/p6/TradeTransactionRelay";
        handle = iCustom(_Symbol, PERIOD_D1, indicator);
        if(handle == INVALID_HANDLE)
        {
            Alert("Can't start indicator ", indicator);
            return INIT_FAILED;
        }
        PrintFormat("Started in %d ms", GetTickCount() - start);
    }
    ...
}
```

为了简化编码，我们在 *SetupGrid* 函数中用一维数组替换了二维 *request* 数组。

```
uint SetupGrid()
{
    ...
    // prev:
    MqlTradeRequestSyncLog request[]; // MqlTradeRequestSyncLog request[][2];
    ArrayResize(request, GridSize * 2); // ArrayResize(request, GridSize);
    ...
}
```

在数组的循环中，我们使用寻址请求 *request[i * 2 + 1]*，而不是请求 *request[i][1]* 类型的调用。

出于以下原因，需要进行这个小的转换。因为我们在创建网格时使用这个结构体数组进行查询，并且我们需要等待所有的结果，所以 *AwaitAsync* 函数现在应引用该数组作为其第一个参数。一维数组更容易处理。

对于每个请求，根据其 *request_id* 计算其在指标缓冲区中的偏移量：所有偏移量都放入 *offset* 数组中。当收到请求确认时，通过在数组中写入值 -1，将数组的相应元素标记为已处理。执行的请求数在 *done* 变量中统计。当这个数量等于数组的大小时，整个网格准备就绪。

```

bool AwaitAsync(MqlTradeRequestSyncLog &r[], const int _handle)
{
    Converter<ulong,double> cnv;
    int offset[];
    const int n = ArraySize(r);
    int done = 0;
    ArrayResize(offset, n);

    for(int i = 0; i < n; ++i)
    {
        offset[i] = (int)((r[i].result.request_id * FIELD_NUM)
            % (Bars(_Symbol, _Period) / FIELD_NUM * FIELD_NUM));
    }

    const uint start = GetTickCount();
    while(!IsStopped() && done < n && GetTickCount() - start < TIMEOUT)
        for(int i = 0; i < n; ++i)
        {
            if(offset[i] == -1) continue; // skip empty elements
            double array[];
            if((CopyBuffer(_handle, 0, offset[i], FIELD_NUM, array)) == FIELD_NUM)
            {
                ArraySetAsSeries(array, true);
                if((uint)MathRound(array[0]) == r[i].result.request_id)
                {
                    r[i].result.retcode = (uint)MathRound(array[1]);
                    r[i].result.deal = cnv[array[2]];
                    r[i].result.order = cnv[array[3]];
                    r[i].result.volume = array[4];
                    r[i].result.price = array[5];
                    PrintFormat("Got Req=%d at %d ms", r[i].result.request_id,
                        GetTickCount() - start);
                    Print(TU::StringOf(r[i].result));
                    offset[i] = -1; // mark processed
                    done++;
                }
            }
        }
    }
    return done == n;
}

```

回到 *SetupGrid* 函数，让我们展示一下在请求发送循环之后 *AwaitAsync* 是如何被调用的。


```

uint SetupGrid()
{
    ...
    const uint start = GetTickCount();
    for(int i = 0; i < (int)GridSize / 2; ++i)
    {
        // calls of buyLimit/sellStopLimit/sellLimit/buyStopLimit
    }

    if(EnableAsyncSetup)
    {
        if(!AwaitAsync(request, handle))
        {
            Print("Timeout");
            return TRADE_RETCODE_ERROR;
        }
    }

    PrintFormat("Done %d requests in %d ms (%d ms/request)",
        GridSize * 2, GetTickCount() - start,
        (GetTickCount() - start) / (GridSize * 2));
    ...
}

```

如果在设置网格时发生超时（并非所有请求都会在分配的时间内收到确认），我们将返回 `TRADE_RETCODE_ERROR` 代码，EA 交易将尝试“回滚”其创建的内容。

需要注意的是，异步模式只是为了在我们需要发送一批请求时建立一个完整的网格。否则，仍会使用同步模式。因此，我们必须在发送循环之前将 `MqlTradeRequestSync::AsyncEnabled` 标志设置为 `true`，之后再将其设置回 `false`。但是，请注意以下几点。循环内部可能会出现错误，因此循环会过早终止，从服务器返回最后一个代码。因此，如果我们在循环之后进行异步复位，就不能保证其能够复位。

为了解决这个问题，在 `MqlTradeSync.mqh` 文件中添加了一个小的 `AsyncSwitcher` 类。该类可通过其构造函数和析构函数来控制异步模式的启用和禁用。这与 [文件描述符管理](#) 一节中讨论的 RAII 资源管理概念一致。

```

class AsyncSwitcher
{
public:
    AsyncSwitcher(const bool enabled = true)
    {
        MqlTradeRequestSync::AsyncEnabled = enabled;
    }
    ~AsyncSwitcher()
    {
        MqlTradeRequestSync::AsyncEnabled = false;
    }
};

```

现在，为了安全地临时激活异步模式，我们可以简单地在 `SetupGrid` 函数中描述本地 `AsyncSwitcher` 对象。退出该函数时，代码将自动返回到同步模式。

```

uint SetupGrid()
{
    ...
    AsyncSwitcher sync(EnableAsyncSetup);
    ...
    for(int i = 0; i < (int)GridSize / 2; ++i)
    {
        ...
    }
    ...
}

```

EA 交易准备就绪。让我们尝试运行两次：对于足够大的网格（10 个层级，网格步长 200），以同步和异步模式运行。

对于 10 层级的网格，我们将得到 20 个查询，所以以下是一些日志。首先，使用同步模式。让我们澄清一下，关于请求准备就绪的标志显示在关于请求的消息之前，因为后者是在该函数退出时由结构析构函数生成的。每个请求的处理速度为 51 毫秒。

```

Start setup at 1.10379
Done 20 requests in 1030 ms (51 ms/request)
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978336, V=0.01, Request executed, Req=1
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978337, V=0.01, Request executed, Req=2
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978343, V=0.01, Request executed, Req=5
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978344, V=0.01, Request executed, Req=6
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.09
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978348, V=0.01, Request executed, Req=9
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10000, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978350, V=0.01, Request executed, Req=10
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978339, V=0.01, Request executed, Req=3
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978340, V=0.01, Request executed, Req=4
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978345, V=0.01, Request executed, Req=7
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.10600, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978347, V=0.01, Request executed, Req=8
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978365, V=0.01, Request executed, Req=19
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.11200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978366, V=0.01, Request executed, Req=20

```

网格中间匹配了 1.10400 的价格。该系统按照收到请求的顺序为请求分配编号，其在数组中的编号对应于我们下订单的顺序：从中心基础层开始，逐渐向两侧发散。因此，不要惊讶在一对编号 1 和 2（对于层级 1.10200）之后是 5 和 6（1.10000），因为 3 和 4（1.10600）发送地更早。

在异步模式下，析构函数前面是关于在 *AwaitAsync* 中实时接收的特定请求准备就绪消息，不一定按照请求发送的顺序（例如，第 49 个和第 50 个请求可能先于第 47 个和第 48 个请求）。

```

Started in 16 ms
Start setup at 1.10356
Got Req=41 at 109 ms
DONE, #=1300979180, V=0.01, Order placed, Req=41
Got Req=42 at 109 ms
DONE, #=1300979181, V=0.01, Order placed, Req=42
Got Req=43 at 125 ms
DONE, #=1300979182, V=0.01, Order placed, Req=43
Got Req=44 at 140 ms
DONE, #=1300979183, V=0.01, Order placed, Req=44
Got Req=45 at 156 ms
DONE, #=1300979184, V=0.01, Order placed, Req=45
Got Req=46 at 172 ms
DONE, #=1300979185, V=0.01, Order placed, Req=46
Got Req=49 at 172 ms
DONE, #=1300979188, V=0.01, Order placed, Req=49
Got Req=50 at 172 ms
DONE, #=1300979189, V=0.01, Order placed, Req=50
Got Req=47 at 172 ms
DONE, #=1300979186, V=0.01, Order placed, Req=47
Got Req=48 at 172 ms
DONE, #=1300979187, V=0.01, Order placed, Req=48
Got Req=51 at 172 ms
DONE, #=1300979190, V=0.01, Order placed, Req=51
Got Req=52 at 203 ms
DONE, #=1300979191, V=0.01, Order placed, Req=52
Got Req=55 at 203 ms
DONE, #=1300979194, V=0.01, Order placed, Req=55
Got Req=56 at 203 ms
DONE, #=1300979195, V=0.01, Order placed, Req=56
Got Req=53 at 203 ms
DONE, #=1300979192, V=0.01, Order placed, Req=53
Got Req=54 at 203 ms
DONE, #=1300979193, V=0.01, Order placed, Req=54
Got Req=57 at 218 ms
DONE, #=1300979196, V=0.01, Order placed, Req=57
Got Req=58 at 218 ms
DONE, #=1300979198, V=0.01, Order placed, Req=58
Got Req=59 at 218 ms
DONE, #=1300979199, V=0.01, Order placed, Req=59
Got Req=60 at 218 ms
DONE, #=1300979200, V=0.01, Order placed, Req=60
Done 20 requests in 234 ms (11 ms/request)
...

```

由于所有请求都是并行执行的，所以总的发送时间（234 毫秒）只比单个请求的时间（此处约为 100 毫秒，但是你可以自己计时）多一点。结果，我们获得了每个请求 11 毫秒的速度，速度是使用同步方法的 5 倍。由于请求几乎是同时发送的，所以我们无法知道每个请求的执行时间，毫秒表示从一般的组发送开始时特定请求结果的到达时间。

与前一种情况一样，更多的日志包含所有从结构体析构函数打印的查询和结果字段。在 `OrderSendAsync` 之后，“Order placed” 行保持不变，因为我们的辅助指标

TradeTransactionRelay.mq5 没有通过 `TRADE_TRANSACTION_REQUEST` 消息完整地发布 *MqlTradeResult* 结构体。

```
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979180, V=0.01, Order placed, Req=41
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979181, V=0.01, Order placed, Req=42
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979184, V=0.01, Order placed, Req=45
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979185, V=0.01, Order placed, Req=46
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.09
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979188, V=0.01, Order placed, Req=49
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10000, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979189, V=0.01, Order placed, Req=50
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979182, V=0.01, Order placed, Req=43
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979183, V=0.01, Order placed, Req=44
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979186, V=0.01, Order placed, Req=47
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.10600, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979187, V=0.01, Order placed, Req=48
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979199, V=0.01, Order placed, Req=59
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.11200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979200, V=0.01, Order placed, Req=60
```

到现在为止，我们的网格 EA 交易在每个层级都有一对挂单：限价和止损限价。为了避免这种重复，我们只留下限价订单。这将是 *PendingOrderGrid4.mq5* 的最终版本，它也可以在同步和异步模式下运行。我们不会详细讨论源代码，只会指出与前一版本的主要区别。

在 *SetupGrid* 函数中，我们需要一个大小等于 *GridSize* 的结构体数组，不再翻倍。请求数量也将减少到原来的一半：用于这些请求的方法仅限于 *buyLimit* 和 *sellLimit*。

CheckGrid 函数可以不同的方式检查网格的完整性。以前，在有限制的层级上没有成对的止损限价订单被认为是错误的。当相邻层级在服务器上触发限价止损订单时，可能会发生这种情况。但是，如果一根柱线上出现强烈的双向价格波动（尖峰），这种方案就无法恢复网格：它不仅会取消原始限价订单，还会取消止损限价订单产生的新限价订单。现在，该算法会如实检查当

前价格两侧的空缺层级,并使用 *RepairGridLevel* 在该处创建限价订单。这个辅助函数以前下达过限价止损订单。

最后, *OnTradeTransaction* 处理程序出现在 *PendingOrderGrid4.mq5* 中。触发挂单会导致一个交易的执行 (以及需要纠正的网格配置改变),所以我们通过给定的交易品种和魔术数字来控制交易。当检测到一个交易时, *CheckGrid* 函数会被立即调用,除此之外,它还会在每个柱线的开始处执行。

```
void OnTradeTransaction(const MqlTradeTransaction &transaction,
    const MqlTradeRequest &,
    const MqlTradeResult &)
{
    if(transaction.type == TRADE_TRANSACTION_DEAL_ADD)
    {
        if(transaction.symbol == _Symbol)
        {
            DealMonitor dm(transaction.deal); // select the deal
            if(dm.get(DEAL_MAGIC) == Magic)
            {
                CheckGrid();
            }
        }
    }
}
```

应注意的是, *OnTradeTransaction* 的存在并不足以编写能够抵抗不可预见外部影响的 EA 交易。当然,许多事件允许你对情况做出快速响应,但是我们不能保证 EA 交易在一段时间内不会因为某种原因而平仓 (或离线)并跳过某个交易。因此, *OnTradeTransaction* 处理程序只能帮助加速程序在无事件时也能执行的流程。特别是在启动后正确恢复其状态。

但是,除了 *OnTradeTransaction* 事件之外, MQL5 还提供了另一个更简单的事件: *OnTrade*。

6.4.36 OnTrade 事件

OnTrade 事件发生在更改已下订单和未结仓位列表、订单历史和交易历史时。任何交易行为 (下单/激活/删除挂单、开仓/平仓、设置保护水平等)都会相应地改变订单和交易的历史和/或仓位和当前订单列表。动作的发起者可以是用户、程序或服务器。

要在程序中接收事件,应说明相应的处理程序。

void OnTrade (void)

在使用 *OrderSend/OrderSendAsync* 发送交易请求的情况下,一个请求会触发多个 *OnTrade* 事件,因为处理通常发生在几个阶段,每个运算都可以改变订单、仓位和交易历史的状态。

一般情况下, *OnTrade* 和 *OnTradeTransaction* 的调用次数没有确切比例。 *OnTrade* 在相应的 *OnTradeTransaction* 调用之后被调用。

由于 *OnTrade* 事件具有一般化的特性,并且不指定操作的本质,所以 MQL 程序开发人员不太喜欢使用。通常需要检查代码中交易账户状态的所有方面,并将其与一些保存的状态进行比较,即与交易策略中使用的交易实体的应用缓存进行比较。例如,在最简单的情况下,你可以在 *OnTrade* 处理程序中记住已创建订单的订单号,以查询其所有特性。但是,这可能意味着对大量与特定订单无关的附带事件进行“不必要的”分析。

我们将在关于 **多货币 EA 交易** 的章节中介绍交易环境和历史的应用缓存。

为了进一步研究 **OnTrade**，我们来看一个 EA 交易对两个 OCO（“互损订单”）挂单实现策略。该策略会下达一对突破止损订单，等待其中一个触发，之后第二个会被删除。为了清楚起见，我们可提供对两种类型交易事件 **OnTrade** 和 **OnTradeTransaction** 的支持，这样工作逻辑将根据用户的选择从一个处理程序或另一个处理程序运行。

OCO2.mq5 文件中提供了源代码。其输入参数包括手数大小 **Volume**（默认值为 0 表示最小）和在每个订单处的 **Distance2SLTP** 距离（以点为单位），它还可以确定保护水平、从设置时间开始的到期日期 **Expiration**（以秒为单位）以及事件切换器 **ActivationBy**（默认为 **OnTradeTransaction**）。由于 **Distance2SLTP** 设置了当前价格的偏移量和到止损点的距离，两个订单的止损是相同的，等于设置时的价格。

```
enum EVENT_TYPE
{
    ON_TRANSACTION, // OnTradeTransaction
    ON_TRADE         // OnTrade
};

input double Volume;           // Volume (0 - minimal lot)
input uint Distance2SLTP = 500; // Distance Indent/SL/TP (points)
input ulong Magic = 1234567890;
input ulong Deviation = 10;
input ulong Expiration = 0;    // Expiration (seconds in future, 3600 - 1 hour, etc)
input EVENT_TYPE ActivationBy = ON_TRANSACTION;
```

为了简化请求结构体的初始化，我们将说明从 **MqlTradeRequestSync** 派生的我们自己的 **MqlTradeRequestSyncOCO** 结构体。

```
struct MqlTradeRequestSyncOCO: public MqlTradeRequestSync
{
    MqlTradeRequestSyncOCO()
    {
        symbol = _Symbol;
        magic = Magic;
        deviation = Deviation;
        if(Expiration > 0)
        {
            type_time = ORDER_TIME_SPECIFIED;
            expiration = (datetime)(TimeCurrent() + Expiration);
        }
    }
};
```

在全局水平，我们介绍几个对象和变量。

```
OrderFilter orders;           // object for selecting orders
PositionFilter trades;        // object for selecting positions
bool FirstTick = false;       // or single processing of OnTick at start
ulong ExecutionCount = 0;     // counter of trading strategy calls RunStrategy()
```

所有的交易逻辑，除了开始时刻，均会被交易事件触发。在 **OnInit** 处理程序中，我们设置筛选对象并等待第一个分时报价（将 **FirstTick** 设置为 **true**）。

```

int OnInit()
{
    FirstTick = true;

    orders.let(ORDER_MAGIC, Magic).let(ORDER_SYMBOL, _Symbol)
        .let(ORDER_TYPE, (1 << ORDER_TYPE_BUY_STOP) | (1 << ORDER_TYPE_SELL_STOP),
            IS::OR_BITWISE);
    trades.let(POSITION_MAGIC, Magic).let(POSITION_SYMBOL, _Symbol);

    return INIT_SUCCEEDED;
}

```

我们只对止损订单（买入/卖出）、有特定魔术编号以及当前交易品种的仓位感兴趣。

在 *OnTick* 函数中，我们曾经调用设计为 *RunStrategy* 的算法的主要部分（在下文中说明）。此外，只能从 *OnTrade* 或 *OnTradeTransaction* 调用此函数。

```

void OnTick()
{
    if(FirstTick)
    {
        RunStrategy();
        FirstTick = false;
    }
}

```

例如，当 *OnTrade* 模式被启用时，这个片段起作用。

```

void OnTrade()
{
    static ulong count = 0;
    PrintFormat("OnTrade(%d)", ++count);
    if(ActivationBy == ON_TRADE)
    {
        RunStrategy();
    }
}

```

注意，不管此处是否激活了该策略，*OnTrade* 处理程序的调用均进行计数。类似地，在 *OnTradeTransaction* 处理程序中会对相关事件进行计数（即使它们的发生无效）。这是为了能够同时在日志中看到事件及其计数器。

当 *OnTradeTransaction* 模式打开时，显然 *RunStrategy* 就是从那里启动的。


```

void OnTradeTransaction(const MqlTradeTransaction &transaction,
    const MqlTradeRequest &request,
    const MqlTradeResult &result)
{
    static ulong count = 0;
    PrintFormat("OnTradeTransaction(%d)", ++count);
    Print(TU::StringOf(transaction));

    if(ActivationBy != ON_TRANSACTION) return;

    if(transaction.type == TRADE_TRANSACTION_ORDER_DELETE)
    {
        // why not here? for answer, see the text
        /* // this won't work online: m.isReady() == false because order temporarily lc
        OrderMonitor m(transaction.order);
        if(m.isReady() && m.get(ORDER_MAGIC) == Magic && m.get(ORDER_SYMBOL) == _Symbol
        {
            RunStrategy();
        }
        */
    }
    else if(transaction.type == TRADE_TRANSACTION_HISTORY_ADD)
    {
        OrderMonitor m(transaction.order);
        if(m.isReady() && m.get(ORDER_MAGIC) == Magic && m.get(ORDER_SYMBOL) == _Symbol
        {
            // the ORDER_STATE property does not matter - in any case, you need to remov
            // if(transaction.order_state == ORDER_STATE_FILLED
            // || transaction.order_state == ORDER_STATE_CANCELED ...)
            RunStrategy();
        }
    }
}

```

应当注意的是，在线交易时，由于从现有订单转移到历史订单，触发的挂单可能会从交易环境中消失一段时间。当我们收到 `TRADE_TRANSACTION_ORDER_DELETE` 事件时，该订单已经从活跃订单中删除但还没有在历史订单中出现。只有当我们接收到 `TRADE_TRANSACTION_HISTORY_ADD` 事件时，该订单才会出现在历史订单中。在测试程序中不会观察到这种行为，即，被删除的订单会立即添加到历史中，并可用于选择和读取已经处于 `TRADE_TRANSACTION_ORDER_DELETE` 阶段的特性。

在这两个交易事件处理程序中，我们可计数并记录调用次数。对于使用 *OnTrade* 的情况，调用次数必须匹配 *ExecutionCount*，我们将很快在 *RunStrategy* 中看到这一点。但是，对于 *OnTradeTransaction*，其计数器和 *ExecutionCount* 会有很大差异，因为此处的策略是针对一种类型的事件非常有选择性地调用的。基于此，我们可以得出结论，*OnTradeTransaction* 通过仅在适当的时候调用算法，允许更有效地使用资源。

卸载 EA 交易时，*ExecutionCount* 计数器会输出到日志中。

```

void OnDeinit(const int r)
{
    Print("ExecutionCount = ", ExecutionCount);
}

```

现在，最后，我们介绍一下 *RunStrategy* 函数。承诺计数器从一开始就递增计数。

```

void RunStrategy()
{
    ExecutionCount++;
    ...
}

```

接下来，说明两个用于从 *orders* 筛选对象接收订单号及其状态的数组。

```

ulong tickets[];
ulong states[];

```

首先，我们可请求符合我们条件的订单。如果有两个订单，一切都好，什么都不用做。

```

orders.select(ORDER_STATE, tickets, states);
const int n = ArraySize(tickets);
if(n == 2) return; // OK - standard state
...

```

如果剩余一个订单，则另一个订单会被触发，剩余的订单必须删除。

```

if(n > 0) // 1 or 2+ orders is an error, you need to delete everything
{
    // delete all matching orders, except for partially filled ones
    MqlTradeRequestSyncOCO r;
    for(int i = 0; i < n; ++i)
    {
        if(states[i] != ORDER_STATE_PARTIAL)
        {
            r.remove(tickets[i]) && r.completed();
        }
    }
}
...

```

否则，没有订单。因此，需要检查是否有未结仓位：为此，我们使用另一个 *trades* 筛选对象，但是结果被添加到同一个接收数组 *tickets* 中。如果没有仓位，我们可下达一对新订单。

```

else // n == 0
{
    // if there are no open positions, place 2 orders
    if(!trades.select(tickets))
    {
        MqlTradeRequestSyncOCO r;
        SymbolMonitor sm(_Symbol);

        const double point = sm.get(SYMBOL_POINT);
        const double lot = Volume == 0 ? sm.get(SYMBOL_VOLUME_MIN) : Volume;
        const double buy = sm.get(SYMBOL_BID) + point * Distance2SLTP;
        const double sell = sm.get(SYMBOL_BID) - point * Distance2SLTP;

        r.buyStop(lot, buy, buy - Distance2SLTP * point,
            buy + Distance2SLTP * point) && r.completed();
        r.sellStop(lot, sell, sell + Distance2SLTP * point,
            sell - Distance2SLTP * point) && r.completed();
    }
}
}

```

我们使用默认设置在 EURUSD 对上运行测试程序中的 EA 交易。下图显示了测试过程。



测试程序中具有一对基于 OCO 策略的止损挂单的 EA 交易

在下达一对订单的阶段，我们将在日志中看到以下条目。

```

buy stop 0.01 EURUSD at 1.11151 sl: 1.10651 tp: 1.11651 (1.10646 / 1.10683)
sell stop 0.01 EURUSD at 1.10151 sl: 1.10651 tp: 1.09651 (1.10646 / 1.10683)
OnTradeTransaction(1)
TRADE_TRANSACTION_ORDER_ADD, #=2(ORDER_TYPE_BUY_STOP/ORDER_STATE_PLACED), ORDER_TIME_
» @ 1.11151, SL=1.10651, TP=1.11651, V=0.01
OnTrade(1)
OnTradeTransaction(2)
TRADE_TRANSACTION_REQUEST
OnTradeTransaction(3)
TRADE_TRANSACTION_ORDER_ADD, #=3(ORDER_TYPE_SELL_STOP/ORDER_STATE_PLACED), ORDER_TIME
» @ 1.10151, SL=1.10651, TP=1.09651, V=0.01
OnTrade(2)
OnTradeTransaction(4)
TRADE_TRANSACTION_REQUEST

```

一旦其中一个订单被触发,就会发生以下情况:

```

order [#3 sell stop 0.01 EURUSD at 1.10151] triggered
deal #2 sell 0.01 EURUSD at 1.10150 done (based on order #3)
deal performed [#2 sell 0.01 EURUSD at 1.10150]
order performed sell 0.01 at 1.10150 [#3 sell stop 0.01 EURUSD at 1.10151]
OnTradeTransaction(5)
TRADE_TRANSACTION_DEAL_ADD, D=2(DEAL_TYPE_SELL), #=3(ORDER_TYPE_BUY/ORDER_STATE_START
» EURUSD, @ 1.10150, SL=1.10651, TP=1.09651, V=0.01, P=3
OnTrade(3)
OnTradeTransaction(6)
TRADE_TRANSACTION_ORDER_DELETE, #=3(ORDER_TYPE_SELL_STOP/ORDER_STATE_FILLED), ORDER_T
» EURUSD, @ 1.10151, SL=1.10651, TP=1.09651, V=0.01, P=3
OnTrade(4)
OnTradeTransaction(7)
TRADE_TRANSACTION_HISTORY_ADD, #=3(ORDER_TYPE_SELL_STOP/ORDER_STATE_FILLED), ORDER_TI
» EURUSD, @ 1.10151, SL=1.10651, TP=1.09651, P=3
order canceled [#2 buy stop 0.01 EURUSD at 1.11151]
OnTrade(5)
OnTradeTransaction(8)
TRADE_TRANSACTION_ORDER_DELETE, #=2(ORDER_TYPE_BUY_STOP/ORDER_STATE_CANCELED), ORDER_
» EURUSD, @ 1.11151, SL=1.10651, TP=1.11651, V=0.01
OnTrade(6)
OnTradeTransaction(9)
TRADE_TRANSACTION_HISTORY_ADD, #=2(ORDER_TYPE_BUY_STOP/ORDER_STATE_CANCELED), ORDER_T
» EURUSD, @ 1.11151, SL=1.10651, TP=1.11651, V=0.01
OnTrade(7)
OnTradeTransaction(10)
TRADE_TRANSACTION_REQUEST

```

订单 #3 被自己删除,订单 #2 被我们的 EA 交易删除 (取消)。

如果我们仅通过在设置中更改的 **OnTrade** 事件运行 EA 交易,我们应得到完全相似的财务结果 (其他条件相同,例如,如果不包括分时报价生成中的随机延迟)。唯一不同的是 **RunStrategy** 函数调用次数。例如,对于 2022 年 4 个月 EURUSD,H1 的 88 笔交易,我们将获得以下 **ExecutionCount** 的近似指标 (重要的是比率,而不是与你的经纪商的分时报价相关的绝对值):

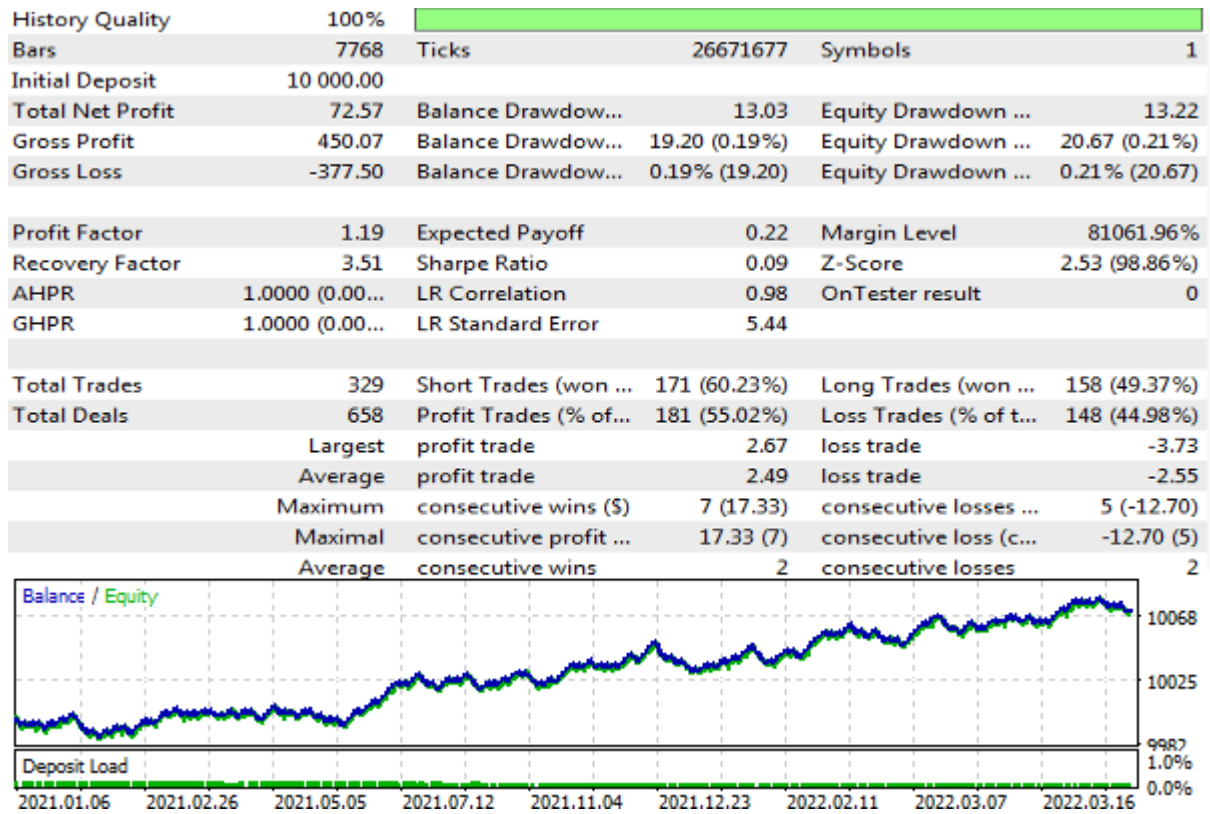
- OnTradeTransaction - 132
- OnTrade - 438

这是一个实际证明，与 *OnTrade* 相比，基于 *OnTradeTransaction* 可以构建更具选择性的算法。

这个 *OCO2.mq5* EA 交易版本对订单和仓位的反应相当简单。特别是之前的仓位被止损或者止盈平仓后，就会下达两个新订单。如果你手动删除其中一个订单，EA 交易将立即删除第二个订单，然后用当前价格的偏移量重新创建一对新订单。你可以通过嵌入一个类似于网格 EA 交易所做的调度来改进这种行为，并且不会对历史中被取消的订单作出反应（但是，请注意，MQL5 没有提供查明订单是手动取消还是编程取消的方法）。在探讨 *财经日历 API* 时，我们将提供一个不同的方向来改进这个 EA 交易。

此外，在当前版本中已经提供了一个有趣的模式，与在输入变量 *Expiration* 中设置挂单到期日期相关。如果一对订单没有触发，然后，在它们到期之后，会立即相对于改变后的新当前价格下达一对新订单。作为一次独立的练习，你可以尝试通过更改 *Expiration* 和 *Distance2SLTP* 来优化测试程序中的 EA 交易。测试程序的编程工作，包括优化模式，将在 [下一章](#) 中介绍。

以下是从 2021 年初开始的 16 个月内发现的 EURUSD 对的设置选项之一 (*Distance2SLTP=250*, *Expiration=5000*)。



OCO2 EA 交易的测试运行结果

6.4.37 监测交易环境变化

在前面关于 *OnTrade* 事件的章节中，我们提到一些交易策略编程方法可能需要你生成环境的快照，并随着时间的推移将它们相互比较。这是使用 *OnTrade* 时的一种常见做法，但也可以按时间表、每根柱线甚至分时报价激活。我们的监视器类虽然可以读取订单、交易和仓位的特性，但不具备保存状态的能力。在本节中，我们将介绍一种交易环境缓存选项。

所有交易对象的特性按类型分为三组：整数型、实数型和字符串型。每个对象类都有自己的组（例如，对于订单，整数型特性在 *ENUM_ORDER_PROPERTY_INTEGER* 枚举中描述，对于仓位，其特性在 *ENUM_POSITION_PROPERTY_INTEGER* 中描述），但分组的本质是相同的。因此，我们将引入 *PROP_TYPE* 枚举，借助该枚举，可以描述一个对象特性属于哪种类型。由于存储和处理相同类型

特性的机制应是相同的，无论特性是订单的、仓位的还是交易的，所以这种通用性是自然而然出现的。

```
enum PROP_TYPE
{
    PROP_TYPE_INTEGER,
    PROP_TYPE_DOUBLE,
    PROP_TYPE_STRING,
};
```

数组是存储特性值的最简单方式。显然，由于存在三种基础类型，我们将需要三个不同的数组。让我们在 *MonitorInterface (TradeBaseMonitor.mqh)* 中嵌套的新类 *TradeState* 中进行说明。

基础模板 *MonitorInterface<I,D,S>* 构成了所有应用监视器类 (*OrderMonitor*、*DealMonitor* 和 *PositionMonitor*) 的基础。此处的类型 I、D 和 S 对应于整数、实数和字符串型特性的具体枚举。

在基础监视器中包含存储机制是非常合理的，特别是因为创建的特性缓存应通过从监视器对象读取特性来填充数据。

```
template<typename I,typename D,typename S>
class MonitorInterface
{
    ...
    class TradeState
    {
    public:
        ...
        long ulongs[];
        double doubles[];
        string strings[];
        const MonitorInterface *owner;

        TradeState(const MonitorInterface *ptr) : owner(ptr)
        {
            ...
        }
    };
};
```

整个 *TradeState* 类已被公开，因为其字段需要从父监视器对象（作为指针传递给构造函数）中访问，而且 *TradeState* 仅在监视器的受保护部分使用（不能从外部访问）。

为了用三种不同类型的特性值填充三个数组，必须首先找出每个特定数组中按类型和索引的特性分布。

对于每个交易对象类型（订单、交易和仓位），具有不同类型特性的 3 个对应枚举标识符不相交，并形成连续编号。我们来演示一下。

在 [枚举](#) 一章中，我们考虑了脚本 *ConversionEnum.mq5*，其实现了 *process* 函数来记录特定枚举的所有元素。该脚本检查了 `ENUM_APPLIED_PRICE` 枚举。现在我们可以创建该脚本的副本，并分析其他三个枚举。例如，如下所示：

```

void OnStart()
{
    process((ENUM_POSITION_PROPERTY_INTEGER)0);
    process((ENUM_POSITION_PROPERTY_DOUBLE)0);
    process((ENUM_POSITION_PROPERTY_STRING)0);
}

```

执行该脚本后，我们可得到以下日志。左边的列包含枚举内部的编号，右边的值（在 “ ” 号之后）为元素的内置常量（标识符）。

```

ENUM_POSITION_PROPERTY_INTEGER Count=9
0 POSITION_TIME=1
1 POSITION_TYPE=2
2 POSITION_MAGIC=12
3 POSITION_IDENTIFIER=13
4 POSITION_TIME_MSC=14
5 POSITION_TIME_UPDATE=15
6 POSITION_TIME_UPDATE_MSC=16
7 POSITION_TICKET=17
8 POSITION_REASON=18
ENUM_POSITION_PROPERTY_DOUBLE Count=8
0 POSITION_VOLUME=3
1 POSITION_PRICE_OPEN=4
2 POSITION_PRICE_CURRENT=5
3 POSITION_SL=6
4 POSITION_TP=7
5 POSITION_COMMISSION=8
6 POSITION_SWAP=9
7 POSITION_PROFIT=10
ENUM_POSITION_PROPERTY_STRING Count=3
0 POSITION_SYMBOL=0
1 POSITION_COMMENT=11
2 POSITION_EXTERNAL_ID=19

```

例如，常量为 0 的特性为字符串型 POSITION_SYMBOL，常量为 1 和 2 的特性为整数型 POSITION_TIME 和 POSITION_TYPE，常量为 3 的特性为实数型 POSITION_VOLUME 等。

因此，常量是所有类型特性的端到端索引系统，我们可以使用相同的算法（基于 *EnumToArray.mqh*）来获取它们。

对于每个特性，需要记住其类型（这决定了三个数组中的哪一个应存储该值）和同一类型特性中的序列号（这是相应数组中元素的索引）。例如，我们看到仓位只有 3 个字符串型特性，因此一个仓位快照中的 *strings* 数组必须具有相同的大小，POSITION_SYMBOL (0)、POSITION_COMMENT (11) 和 POSITION_EXTERNAL_ID (19) 将被写入其索引 0、1 和 2。

特性的端到端索引到其类型（PROP_TYPE 之一）以及到相应类型数组中序数的转换可以在程序开始时完成一次，因为具有特性的枚举为常量（内置于系统中）。我们可将得到的间接寻址表写入一个静态二维 *indices* 数组。其第一维度的大小动态确定为特性总数（所有 3 种类型）。我们将其大小写入 *limit* 静态变量。为第二维度分配了几个单元：*indices[i][0]* - 类型为 PROP_TYPE、*indices[i][1]* - *ulongs*、*doubles* 或 *strings* 数组之一的索引（具体取决于 *indices[i][0]*）。

```

class TradeState
{
    ...
    static int indices[][2];
    static int j, d, s;
public:
    const static int limit;

    static PROP_TYPE type(const int i)
    {
        return (PROP_TYPE)indices[i][0];
    }

    static int offset(const int i)
    {
        return indices[i][1];
    }
    ...
}

```

变量 *j*、*d* 和 *s* 用于按顺序对 3 种不同类型的特性建立索引。以下是其在静态方法 *calcIndices* 中完成的方式。


```

static int calcIndices()
{
    const int size = fmax(boundary<I>(),
        fmax(boundary<D>(), boundary<S>())) + 1;
    ArrayResize(indices, size);
    j = d = s = 0;
    for(int i = 0; i < size; ++i)
    {
        if(detect<I>(i))
        {
            indices[i][0] = PROP_TYPE_INTEGER;
            indices[i][1] = j++;
        }
        else if(detect<D>(i))
        {
            indices[i][0] = PROP_TYPE_DOUBLE;
            indices[i][1] = d++;
        }
        else if(detect<S>(i))
        {
            indices[i][0] = PROP_TYPE_STRING;
            indices[i][1] = s++;
        }
        else
        {
            Print("Unresolved int value as enum: ", i, " ", typename(TradeState));
        }
    }
    return size;
}

```

boundary 方法可返回给定枚举 E 的所有元素中的最大常量。

```

template<typename E>
static int boundary(const E dummy = (E)NULL)
{
    int values[];
    const int n = EnumToArray(dummy, values, 0, 1000);
    ArraySort(values);
    return values[n - 1];
}

```

所有三种类型枚举的最大值确定了应根据其所属特性类型排序的整数范围。

此处我们使用 *detect* 方法，如果该整数是枚举的一个元素，则该方法返回 *true*。

```

template<typename E>
static bool detect(const int v)
{
    ResetLastError();
    const string s = EnumToString((E)v); // result is not used
    if(_LastError == 0) // only the absence of an error is important
    {
        return true;
    }
    return false;
}

```

最后一个是程序启动时如何运行该计算。这是通过利用变量和方法的静态特性来实现的。

```

template<typename I,typename D,typename S>
static int MonitorInterface::TradeState::indices[][2];
template<typename I,typename D,typename S>
static int MonitorInterface::TradeState::j,
    MonitorInterface::TradeState::d,
    MonitorInterface::TradeState::s;
template<typename I,typename D,typename S>
const static int MonitorInterface::TradeState::limit =
    MonitorInterface::TradeState::calcIndices();

```

注意, *limit* 通过调用 *calcIndices* 函数进行初始化。

获得了一个带索引的表之后, 我们便可以在 *cache* 方法中实现用特性值填充数组。

```

class TradeState
{
    ...
    TradeState(const MonitorInterface *ptr) : owner(ptr)
    {
        cache(); // when creating an object, immediately cache the properties
    }

    template<typename T>
    void _get(const int e, T &value) const // overload with record by reference
    {
        value = owner.get(e, value);
    }

    void cache()
    {
        ArrayResize(ulongs, j);
        ArrayResize(doubles, d);
        ArrayResize(strings, s);
        for(int i = 0; i < limit; ++i)
        {
            switch(indices[i][0])
            {
                case PROP_TYPE_INTEGER: _get(i, ulongs[indices[i][1]]); break;
                case PROP_TYPE_DOUBLE: _get(i, doubles[indices[i][1]]); break;
                case PROP_TYPE_STRING: _get(i, strings[indices[i][1]]); break;
            }
        }
    }
};

```

我们遍历从 0 到 *limit* 的整个特性范围，根据 *indices[i][0]* 中的特性类型，将其值写入 *ulongs*、*doubles* 或 *strings* 数组中编号为 *indices[i][1]* 的元素（数组的相应元素通过引用传递给 *_get* 方法）。

对 *owner.get(e, value)* 的调用引用了监视器类的一个标准方法（此处其作为抽象指针 *MonitorInterface* 可见）。特别是，对于 *PositionMonitor* 类中的仓位，这将导致 *PositionGetInteger*、*PositionGetDouble*，或 *PositionGetString* 调用。编译器可选择正确的类型。订单和交易监视器均有它们各自的类似实现，这些实现自动包含在这个基础代码中。

从监视器类继承一个交易对象的快照说明是合乎逻辑的。因为我们必须缓存订单、交易和仓位，所以将新类作为模板并集中适用于所有对象的所有通用算法是一种合理方案。我们姑且称之为 *TradeBaseState* (*TradeState.mqh* 文件)。

```

template<typename M,typename I,typename D,typename S>
class TradeBaseState: public M
{
    M::TradeState state;
    bool cached;

public:
    TradeBaseState(const ulong t) : M(t), state(&this), cached(true)
    {
    }

    void passthrough(const bool b)    // enable/disable cache as desired
    {
        cached = b;
    }
    ...

```

前面介绍的特定监视器类中，部分以字母 **M** 隐藏了具体细节 (*OrderMonitor.mqh*、*PositionMonitor.mqh*和 *DealMonitor.mqh*)。其依据是新引入的 **M::TradeState** 类的 **state** 缓存对象。根据 **M**，可在内部形成一个特定的索引表（一个 **M** 类对应一个），并分配特性数组（每个 **M** 实例拥有一个，即每个订单、交易、仓位都有一个）。

cached 变量包含一个标志，用于表明 **state** 中的数组是否填充有特性值以及是否查询对象特性以从缓存中返回值。然后，需要将保存状态与当前状态进行比较。

换言之，当 **cached** 设置为 **false** 时，对象会像常规监视器一样从交易环境中读取特性。当 **cached** 等于 **true** 时，对象将从内部数组返回以前存储的值。

```

virtual long get(const I property) const override
{
    return cached ? state.ulongs[M::TradeState::offset(property)] : M::get(property)
}

virtual double get(const D property) const override
{
    return cached ? state.doubles[M::TradeState::offset(property)] : M::get(property)
}

virtual string get(const S property) const override
{
    return cached ? state.strings[M::TradeState::offset(property)] : M::get(property)
}
...

```

当然，默认是启用缓存的。

我们还必须提供一个直接执行缓存（填充数组）的方法。为此，只需调用 **state** 对象的 **cache** 方法。

```

bool update()
{
    if(refresh())
    {
        cached = false; // disable reading from the cache
        state.cache(); // read real properties and write to cache
        cached = true; // enable external cache access back
        return true;
    }
    return false;
}

```

refresh 方法有什么作用呢？

迄今为止，我们一直都是在简单模式下使用监视器对象：创建、读取和删除特性。同时，特性读取假设在交易上下文（在构造函数内部）中选择了相应的订单、交易或仓位。因为我们现在正在改进监视器以支持内部状态，所以必须确保重新分配所需的元素，以便在不确定时间之后读取特性（当然，同时还会检查元素是否仍然存在）。为了实现这一点，我们向模板 *MonitorInterface* 类添加了 *refresh* 虚方法。

```

// TradeBaseMonitor.mqh
template<typename I,typename D,typename S>
class MonitorInterface
{
    ...
    virtual bool refresh() = 0;
}

```

成功分配订单、交易或仓位后，该方法必须返回 *true*。如果结果为 *false*，则内置 *_LastError* 变量中应包含以下错误之一：

- 4753 ERR_TRADE_POSITION_NOT_FOUND;
- 4754 ERR_TRADE_ORDER_NOT_FOUND;
- 4755 ERR_TRADE_DEAL_NOT_FOUND;

在这种情况下，在派生类中实现此方法时，表示对象可用性的 *ready* 元素变量必须重置为 *false*。

例如，在 *PositionMonitor* 构造函数中，我们曾经并且现在仍然有这样的初始化。这种情况类似于订单和交易监视器。

```
// PositionMonitor.mqh
const ulong ticket;
PositionMonitor(const ulong t): ticket(t)
{
    if(!PositionSelectByTicket(ticket))
    {
        PrintFormat("Error: PositionSelectByTicket(%lld) failed: %s", ticket,
            E2S(_LastError));
    }
    else
    {
        ready = true;
    }
}
...
```

现在，我们把 *refresh* 方法添加到这种类型的所有特定类中（参见示例 *PositionMonitor*）：

```
// PositionMonitor.mqh
virtual bool refresh() override
{
    ready = PositionSelectByTicket(ticket);
    return ready;
}
```

但是用特性值填充缓存数组只是成功了一半。另一半是将这些值与订单、交易或仓位的实际状态进行比较。

为了识别差异并将已更改特性的索引写入 *changes* 数组，生成的 *TradeBaseState* 类提供了 *getChanges* 方法。检测到更改时，该方法返回 *true*。

```
template<typename M,typename I,typename D,typename S>
class TradeBaseState: public M
{
    ...
    bool getChanges(int &changes[])
    {
        const bool previous = ready;
        if(refresh())
        {
            // element is selected in the trading environment = properties can be read a
            cached = false;    // read directly
            const bool result = M::diff(state, changes);
            cached = true;    // turn cache back on by default
            return result;
        }
        // no longer "ready" = most likely deleted
        return previous != ready; // if just deleted, this is also a change
    }
}
```

可以看到，主要工作委托给了 *M* 类中的某个方法 *diff*，这是一个新方法，我们需要自己编写。幸运的是，由于有了 OOP，你可以在基础模板 *MonitorInterface* 中实现一次，该方法将会立即出现在订单、交易和仓位中。

```
// TradeBaseMonitor.mqh
template<typename I,typename D,typename S>
class MonitorInterface
{
    ...
    bool diff(const TradeState &that, int &changes[])
    {
        ArrayResize(changes, 0);
        for(int i = 0; i < TradeState::limit; ++i)
        {
            switch(TradeState::indices[i][0])
            {
                case PROP_TYPE_INTEGER:
                    if(this.get((I)i) != that.ulongs[TradeState::offset(i)])
                    {
                        PUSH(changes, i);
                    }
                    break;
                case PROP_TYPE_DOUBLE:
                    if(!TU::Equal(this.get((D)i), that.doubles[TradeState::offset(i)]))
                    {
                        PUSH(changes, i);
                    }
                    break;
                case PROP_TYPE_STRING:
                    if(this.get((S)i) != that.strings[TradeState::offset(i)])
                    {
                        PUSH(changes, i);
                    }
                    break;
            }
        }
        return ArraySize(changes) > 0;
    }
}
```

所以，一切准备就绪，可以为订单、交易和仓位形成特定的缓存类。例如，仓位将存储在基于 *PositionMonitor* 的扩展监视器 *PositionState* 中。

```
class PositionState: public TradeBaseState<PositionMonitor,
    ENUM_POSITION_PROPERTY_INTEGER,
    ENUM_POSITION_PROPERTY_DOUBLE,
    ENUM_POSITION_PROPERTY_STRING>
{
public:
    PositionState(const long t): TradeBaseState(t) { }
};
```

类似地，在 *TradeState.mqh* 文件中定义了交易的缓存类。

```

class DealState: public TradeBaseState<DealMonitor,
    ENUM_DEAL_PROPERTY_INTEGER,
    ENUM_DEAL_PROPERTY_DOUBLE,
    ENUM_DEAL_PROPERTY_STRING>
{
public:
    DealState(const long t): TradeBaseState(t) { }
};

```

对于订单，情况稍微复杂一点，因为它们可以分为活跃订单和历史订单。目前，我们已经有了一个通用的订单监视器类 **OrderMonitor**。其会试图在活跃订单和历史订单中查找提交的订单号。这种方法不适合缓存，因为 EA 交易需要跟踪订单从一种状态到另一种状态的转换。

出于这个原因，我们向 **OrderMonitor.mqh** 文件添加了另外两个特定的类：**ActiveOrderMonitor** 和 **HistoryOrderMonitor**。

```

// OrderMonitor.mqh
class ActiveOrderMonitor: public OrderMonitor
{
public:
    ActiveOrderMonitor(const ulong t): OrderMonitor(t)
    {
        if(history) // if the order is in history, then it is already inactive
        {
            ready = false; // reset ready flag
            history = false; // this object is only for active orders by definition
        }
    }

    virtual bool refresh() override
    {
        ready = OrderSelect(ticket);
        return ready;
    }
};

class HistoryOrderMonitor: public OrderMonitor
{
public:
    HistoryOrderMonitor(const ulong t): OrderMonitor(t) { }

    virtual bool refresh() override
    {
        history = true; // work only with history
        ready = historyOrderSelectWeak(ticket);
        return ready; // readiness is determined by the presence of a ticket in the his
    }
};

```

二者均仅在自己的区域内搜索订单号。基于这些监视器，即可创建缓存类。

// TradeState.mqh

```

class OrderState: public TradeBaseState<ActiveOrderMonitor,
    ENUM_ORDER_PROPERTY_INTEGER,
    ENUM_ORDER_PROPERTY_DOUBLE,
    ENUM_ORDER_PROPERTY_STRING>
{
public:
    OrderState(const long t): TradeBaseState(t) { }
};

class HistoryOrderState: public TradeBaseState<HistoryOrderMonitor,
    ENUM_ORDER_PROPERTY_INTEGER,
    ENUM_ORDER_PROPERTY_DOUBLE,
    ENUM_ORDER_PROPERTY_STRING>
{
public:
    HistoryOrderState(const long t): TradeBaseState(t) { }
};

```

为了方便起见，添加到 *TradeBaseState* 类的最后一点是将特性值转换为字符串的特殊方法。虽然监视器中有几个版本的 *stringify* 方法，但它们都是“打印”来自缓存的值（如果元素变量 *cached* 等于 *true*）或来自交易环境原始对象的值（如果 *cached* 等于 *false*）。为了可视化缓存和更改对象之间的差异（若发现这些差异），我们需要同时从缓存中读取值并绕过缓存。在这方面，我们添加了 *stringifyRaw* 方法，其始终直接与特性一起进行处理（因为 *cached* 变量被临时重置并重新安装）。

```

// get the string representation of the property 'i' bypassing the cache
string stringifyRaw(const int i)
{
    const bool previous = cached;
    cached = false;
    const string s = stringify(i);
    cached = previous;
}

```

我们使用一个简单的 EA 交易示例来检查缓存监视器的性能，该监视器可监视活跃订单的状态 (*OrderSnapshot.mq5*)。后续我们将扩展这个思路，实现对一组订单、交易或仓位进行缓冲，即创建一个功能完整的缓存。

EA 交易将尝试查找活跃订单列表中的最后一个订单，并为其创建 *OrderState* 对象。如果没有订单，将提示用户创建订单或开仓（后者与在市場上下达并执行订单相关）。只要发现订单，就会检查订单状态是否已经改变。这项检查是在 *OnTrade* 处理程序中执行的。EA 交易将继续监控该订单，直到订单被卸载为止。

```

int OnInit()
{
    if(OrdersTotal() == 0)
    {
        Alert("Please, create a pending order or open/close a position");
    }
    else
    {
        OnTrade(); // self-invocation
    }
    return INIT_SUCCEEDED;
}

void OnTrade()
{
    static int count = 0;
    // object pointer is stored in static AutoPtr
    static AutoPtr<OrderState> auto;
    // get a "clean" pointer (so as not to dereference auto[] everywhere)
    OrderState *state = auto[];

    PrintFormat(">>> OnTrade(%d)", count++);

    if(OrdersTotal() > 0 && state == NULL)
    {
        const ulong ticket = OrderGetTicket(OrdersTotal() - 1);
        auto = new OrderState(ticket);
        PrintFormat("Order picked up: %lld %s", ticket,
            auto[].isReady() ? "true" : "false");
        auto[].print(); // initial state at the time of "capturing" the order
    }
    else if(state)
    {
        int changes[];
        if(state.getChanges(changes))
        {
            Print("Order properties changed:");
            ArrayPrint(changes);
            ...
        }
        if(_LastError != 0) Print(E2S(_LastError));
    }
}

```

除了显示已更改特性的数组之外，最好也能显示变更本身。因此，我们将添加这样一个片段，而不是省略号（该片段在未来功能完整的缓存类中非常有用）。

```

for(int k = 0; k < ArraySize(changes); ++k)
{
    switch(OrderState::TradeState::type(changes[k]))
    {
        case PROP_TYPE_INTEGER:
            Print(EnumToString((ENUM_ORDER_PROPERTY_INTEGER)changes[k]), ": ",
                state.stringify(changes[k]), " -> ",
                state.stringifyRaw(changes[k]));
            break;
        case PROP_TYPE_DOUBLE:
            Print(EnumToString((ENUM_ORDER_PROPERTY_DOUBLE)changes[k]), ": ",
                state.stringify(changes[k]), " -> ",
                state.stringifyRaw(changes[k]));
            break;
        case PROP_TYPE_STRING:
            Print(EnumToString((ENUM_ORDER_PROPERTY_STRING)changes[k]), ": ",
                state.stringify(changes[k]), " -> ",
                state.stringifyRaw(changes[k]));
            break;
    }
}

```

此处我们使用了新的 *stringifyRaw* 方法。显示变更后，不要忘记更新缓存状态。

```
state.update();
```

如果你在没有活跃订单的账户上运行 EA 交易，并下了一个新订单，你将在日志中看到以下条目（此处创建的 EURUSD *buy limit* 低于当前市场价格）。

```

Alert: Please, create a pending order or open/close a position
>>> OnTrade(0)
Order picked up: 1311736135 true
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
0 ORDER_TIME_SETUP=2022.04.11 11:42:39
1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
2 ORDER_TIME_DONE=1970.01.01 00:00:00
3 ORDER_TYPE=ORDER_TYPE_BUY_LIMIT
4 ORDER_TYPE_FILLING=ORDER_FILLING_RETURN
5 ORDER_TYPE_TIME=ORDER_TIME_GTC
6 ORDER_STATE=ORDER_STATE_STARTED
7 ORDER_MAGIC=0
8 ORDER_POSITION_ID=0
9 ORDER_TIME_SETUP_MSC=2022.04.11 11:42:39'729
10 ORDER_TIME_DONE_MSC=1970.01.01 00:00:00'000
11 ORDER_POSITION_BY_ID=0
12 ORDER_TICKET=1311736135
13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
0 ORDER_VOLUME_INITIAL=0.01
1 ORDER_VOLUME_CURRENT=0.01
2 ORDER_PRICE_OPEN=1.087
3 ORDER_PRICE_CURRENT=1.087
4 ORDER_PRICE_STOPLIMIT=0.0
5 ORDER_SL=0.0
6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
0 ORDER_SYMBOL=EURUSD
1 ORDER_COMMENT=
2 ORDER_EXTERNAL_ID=
>>> OnTrade(1)
Order properties changed:
10 14
ORDER_PRICE_CURRENT: 1.087 -> 1.09073
ORDER_STATE: ORDER_STATE_STARTED -> ORDER_STATE_PLACED
>>> OnTrade(2)
>>> OnTrade(3)
>>> OnTrade(4)

```

此处，你可以看到订单状态是如何从 `STARTED` (已开始)变为 `PLACED` (已下单)。如果我们在市场上以少的交易量开盘而不是挂单，我们可能没有时间接收这些变化，因为这样的订单通常设置为非常快的执行速度，并且其观察状态会立即从 `STARTED` (已开始)变为 `FILLED` (已成交)。而后者意味着，订单已经被移至历史订单。因此，需要同步监视历史对其进行跟踪。我们将在下一个示例中展示这一点。

请注意，可能会有许多 *OnTrade* 事件，但并非所有事件都与我们的订单相关。

我们来尝试设置 *Take Profit* 水平，并查看日志。

```
>>> OnTrade(5)
Order properties changed:
10 13
ORDER_PRICE_CURRENT: 1.09073 -> 1.09079
ORDER_TP: 0.0 -> 1.097
>>> OnTrade(6)
>>> OnTrade(7)
```

下一步，改变到期日期：从 GTC 改为一天。

```
>>> OnTrade(8)
Order properties changed:
10
ORDER_PRICE_CURRENT: 1.09079 -> 1.09082
>>> OnTrade(9)
>>> OnTrade(10)
Order properties changed:
2 6
ORDER_TIME_EXPIRATION: 1970.01.01 00:00:00 -> 2022.04.11 00:00:00
ORDER_TYPE_TIME: ORDER_TIME_GTC -> ORDER_TIME_DAY
>>> OnTrade(11)
```

此处，在更改订单的过程中，价格有足够的时间进行更改，因此我们在 `ORDER_PRICE_CURRENT` 中“挂入”了一个关于新值的中间通知。只有在此之后，`ORDER_TYPE_TIME` 和 `ORDER_TIME_EXPIRATION` 中的预期更改才会记录到日志中。

接下来，我们移除了该订单。

```
>>> OnTrade(12)
TRADE_ORDER_NOT_FOUND
```

现在，对于导致 *OnTrade* 事件的任何账户操作，我们的 EA 交易将输出 `TRADE_ORDER_NOT_FOUND`，因为它是为跟踪单个订单而设计的。如果 EA 交易重新启动，它将“缓存”另一个订单（如果有）。但是我们将停止 EA 交易，并开始准备一个更紧急的任务。

通常，缓存和控制更改不是针对单个订单或仓位，而是针对根据特定条件选择的所有订单或仓位或其中一部分。出于此目的，我们将开发一个基础模板类 *TradeCache* (*TradeCache.mqh*)，并据此为订单、交易和仓位列表创建应用类。

```

template<typename T,typename F,typename E>
class TradeCache
{
    AutoPtr<T> data[];
    const E property;
    const int NOT_FOUND_ERROR;

public:
    TradeCache(const E id, const int error): property(id), NOT_FOUND_ERROR(error) { }

    virtual string rtti() const
    {
        return typename(this); // will be redefined in derived classes for visual output
    }
    ...
}

```

在这个模板中，字母 T 表示 *TradeState* 系列的一个类。可以看出，这种对象的数组以自动指针的形式保留，名称为 *data*。

字母 F 描述了用于选择缓存项的筛选器类的类型 *OrderFilter.mqh*，包括 *HistoryOrderFilter*、*DealFilter.mqh*和 *PositionFilter.mqh*。在最简单的情况下，当筛选器不包含 *let* 条件时，所有元素都将被缓存（相对于来自历史的对象的 *采样历史*）。

字母 E 对应于标识对象的 *property* 所在的枚举。由于这个特性通常为 *SOME_TICKET*，所以枚举假定为整数型 *ENUM_SOMETHING_PROPERTY_INTEGER*。

NOT_FOUND_ERROR 变量专用于处理尝试分配不存在的对象进行读取操作时出现的错误代码，例如，针对仓位的 *ERR_TRADE_POSITION_NOT_FOUND* 错误。

在参数中，主要的类方法 *scan* 用于扫描对已配置筛选器的引用（应由调用代码配置）。

```

void scan(F &f)
{
    const int existedBefore = ArraySize(data);

    ulong tickets[];
    ArrayResize(tickets, existedBefore);
    for(int i = 0; i < existedBefore; ++i)
    {
        tickets[i] = data[i][].get(property);
    }
    ...
}

```

在该方法的开头，我们会将已经缓存的对象标识符收集到 *tickets* 数组中。显然第一次运行时，数组为空。

接下来，我们使用筛选器用相关对象的订单号填充 *objects* 数组。对于每个新订单号，我们都会创建一个缓存监视器对象 T，并将其添加到 *data* 数组中。对于旧对象，我们可通过调用 *data[j][].getChanges(changes)* 来分析是否存在报告，然后通过调用 *data[j][].update()* 来更新缓存。

```

ulong objects[];
f.select(objects);
for(int i = 0, j; i < ArraySize(objects); ++i)
{
    const ulong ticket = objects[i];
    for(j = 0; j < existedBefore; ++j)
    {
        if(tickets[j] == ticket)
        {
            tickets[j] = 0; // mark as found
            break;
        }
    }

    if(j == existedBefore) // this is not in the cache, you need to add
    {
        const T *ptr = new T(ticket);
        PUSH(data, ptr);
        onAdded(*ptr);
    }
    else
    {
        ResetLastError();
        int changes[];
        if(data[j][].getChanges(changes))
        {
            onUpdated(data[j][], changes);
            data[j][].update();
        }
        if(_LastError) PrintFormat("%s: %lld (%s)", rtti(), ticket, E2S(_LastErrc
    }
}
...

```

可以看到，在变更的每个阶段，也就是添加对象时或更改对象后，*onAdded* 和 *onUpdated* 方法都会被调用。这些都是虚拟存根方法，扫描可以使用它们向程序通知适当的事件。应用程序代码需要使用这些方法的重写版本来实现派生类。我们稍后会谈到这个问题，但是现在我们将继续考虑 *scan* 方法。

在上面的循环中，*tickets* 数组中所有找到的订单号均被设置为零，因此剩余的元素对应于交易环境中缺失的对象。接下来，我们调用 *getChanges* 并将错误代码与 *NOT_FOUND_ERROR* 进行比较，以此来检查这些对象。如果结果相符，则调用 *onRemoved* 虚方法。然后，会返回一个布尔标志（由你的应用程序代码提供），表示该项是否应从缓存中移除。

```

for(int j = 0; j < existedBefore; ++j)
{
    if(tickets[j] == 0) continue; // skip processed elements

    // this ticket was not found, most likely deleted
    int changes[];
    ResetLastError();
    if(data[j][].getChanges(changes))
    {
        if(_LastError == NOT_FOUND_ERROR) // for example, ERR_TRADE_POSITION_NOT_FOUND
        {
            if(onRemoved(data[j]{}))
            {
                data[j] = NULL; // release the object and array element
            }
            continue;
        }

        // NB! Usually we shouldn't fall here
        PrintFormat("Unexpected ticket: %lld (%s) %s", tickets[j],
            E2S(_LastError), rtti());
        onUpdated(data[j][], changes, true);
        data[j][].update();
    }
    else
    {
        PrintFormat("Orphaned element: %lld (%s) %s", tickets[j],
            E2S(_LastError), rtti());
    }
}
}

```

在 *scan* 方法的最后，*data* 数组中的空元素会被清除，但为了简洁起见，此处省略了这个片段。

基础类可提供 *onAdded*、*onRemoved* 和 *onUpdated* 方法的标准实现，这些方法用于显示日志中事件的要点。在包含头文件 *TradeCache.mqh* 之前，在代码中定义 *PRINT_DETAILS* 宏，这样便可以要求打印每个新对象的所有特性。


```

virtual void onAdded(const T &state)
{
    Print(rtti(), " added: ", state.get(property));
    #ifdef PRINT_DETAILS
    state.print();
    #endif
}

virtual bool onRemoved(const T &state)
{
    Print(rtti(), " removed: ", state.get(property));
    return true; // allow the object to be removed from the cache (false to save)
}

virtual void onUpdated(T &state, const int &changes[],
    const bool unexpected = false)
{
    ...
}

```

我们不再介绍 *onUpdated* 方法，因为其实际上与上面显示的从 EA 交易 *OrderSnapshot.mq5* 输出变更的代码重复。

当然，该基础类具备获取缓存大小和通过编号访问特定对象的功能。

```

int size() const
{
    return ArraySize(data);
}

T *operator[](int i) const
{
    return data[i][]; // return pointer (T*) from AutoPtr object
}

```

基于基础类 *TradeCache*，我们可以轻松地创建某些类来缓存仓位、活跃订单和历史订单列表。交易缓存则留作一项独立任务。

```

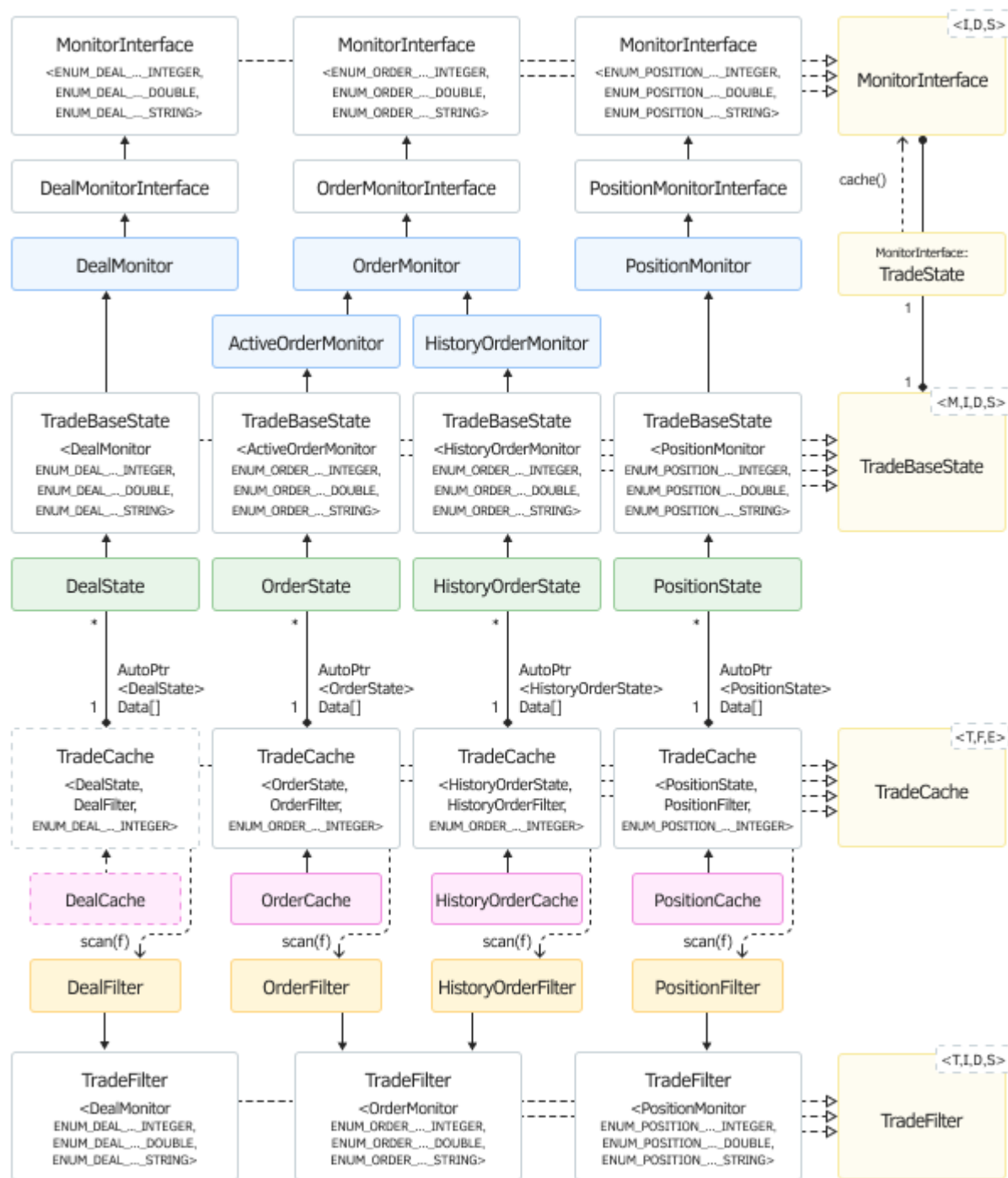
class PositionCache: public TradeCache<PositionState,PositionFilter,
    ENUM_POSITION_PROPERTY_INTEGER>
{
public:
    PositionCache(const ENUM_POSITION_PROPERTY_INTEGER selector = POSITION_TICKET,
        const int error = ERR_TRADE_POSITION_NOT_FOUND): TradeCache(selector, error) {
};

class OrderCache: public TradeCache<OrderState,OrderFilter,
    ENUM_ORDER_PROPERTY_INTEGER>
{
public:
    OrderCache(const ENUM_ORDER_PROPERTY_INTEGER selector = ORDER_TICKET,
        const int error = ERR_TRADE_ORDER_NOT_FOUND): TradeCache(selector, error) { }
};

class HistoryOrderCache: public TradeCache<HistoryOrderState,HistoryOrderFilter,
    ENUM_ORDER_PROPERTY_INTEGER>
{
public:
    HistoryOrderCache(const ENUM_ORDER_PROPERTY_INTEGER selector = ORDER_TICKET,
        const int error = ERR_TRADE_ORDER_NOT_FOUND): TradeCache(selector, error) { }
};

```

为了总结所展示功能的开发过程，我们展示了一个主要类的图表。这是 UML 图的简化版本，在 MQL5 中设计复杂程序时会很有用。



交易对象的监视器、筛选器和缓存的类图

模板用黄色标记，抽象类用白色标记，某些实现用彩色显示。实线箭头（尖端填充）表示继承，虚线箭头（尖端空心）表示模板类型。虚线箭头（尖端开放）表示类之间对彼此指定方法的使用。与菱形标志的连接表示一种组合（将一些对象包含到其他对象中）。

为了举例说明缓存的使用，我们来创建一个 EA 交易 *TradeSnapshot.mq5*，该 EA 交易将会响应 *OnTrade* 处理程序响应交易环境中的任何变化。对于筛选和缓存，该代码说明了 6 个对象，每种类型的元素有 2 个（筛选和缓存）：仓位、活跃订单和历史订单。

```

PositionFilter filter0;
PositionCache positions;

OrderFilter filter1;
OrderCache orders;

HistoryOrderFilter filter2;
HistoryOrderCache history;

```

通过调用 `let` 方法，没有为筛选器设置任何条件，因此所有发现的在线对象都将进入缓存。历史订单还有一个附加设置。

或者，在启动时，可以将过去的订单加载到给定历史深度的缓存中。这可以通过 `HistoryLookup` 输入变量来完成。在此变量中，你可以选择最后一天、上周（按持续时间，而不是日历）、月（30 天）或年（360 天）。默认情况下，不加载过去的历史（更准确地说，仅加载 1 秒的历史）。因为 `PRINT_DETAILS` 宏是在 EA 交易中定义的，所以对于有大量历史的账户要格外小心：如果没有限制时段，会生成大量日志。

```

enum ENUM_HISTORY_LOOKUP
{
    LOOKUP_NONE = 1,
    LOOKUP_DAY = 86400,
    LOOKUP_WEEK = 604800,
    LOOKUP_MONTH = 2419200,
    LOOKUP_YEAR = 29030400,
    LOOKUP_ALL = 0,
};

input ENUM_HISTORY_LOOKUP HistoryLookup = LOOKUP_NONE;

datetime origin;

```

在 `OnInit` 处理程序中，我们可以重置缓存（以防 EA 交易使用新参数重启），计算 `origin` 变量中历史的开始日期，并第一次调用 `OnTrade`。

```

int OnInit()
{
    positions.reset();
    orders.reset();
    history.reset();
    origin = HistoryLookup ? TimeCurrent() - HistoryLookup : 0;

    OnTrade(); // self start
    return INIT_SUCCEEDED;
}

```

`OnTrade` 处理程序非常简单，因为所有的复杂性现在都隐藏在类中。

```

void OnTrade()
{
    static int count = 0;

    PrintFormat(">>> OnTrade(%d)", count++);
    positions.scan(filter0);
    orders.scan(filter1);
    // make a history selection just before using the filter
    // inside the 'scan' method
    HistorySelect(origin, LONG_MAX);
    history.scan(filter2);
    PrintFormat(">>> positions: %d, orders: %d, history: %d",
        positions.size(), orders.size(), history.size());
}

```

在空白账户上启动 EA 交易后，我们将会立即看到以下消息：

```

>>> OnTrade(0)
>>> positions: 0, orders: 0, history: 0

```

让我们尝试执行最简单的测试案例：在一个空账户（该账户没有未平仓仓位和挂单）上买入或卖出。该日志将记录以下事件（几乎立即发生）。

首先将检测到一个活跃订单。

```

>>> OnTrade(1)
OrderCache added: 1311792104
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
0 ORDER_TIME_SETUP=2022.04.11 12:34:51
1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
2 ORDER_TIME_DONE=1970.01.01 00:00:00
3 ORDER_TYPE=ORDER_TYPE_BUY
4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
5 ORDER_TYPE_TIME=ORDER_TIME_GTC
6 ORDER_STATE=ORDER_STATE_STARTED
7 ORDER_MAGIC=0
8 ORDER_POSITION_ID=0
9 ORDER_TIME_SETUP_MSC=2022.04.11 12:34:51'096
10 ORDER_TIME_DONE_MSC=1970.01.01 00:00:00'000
11 ORDER_POSITION_BY_ID=0
12 ORDER_TICKET=1311792104
13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
0 ORDER_VOLUME_INITIAL=0.01
1 ORDER_VOLUME_CURRENT=0.01
2 ORDER_PRICE_OPEN=1.09218
3 ORDER_PRICE_CURRENT=1.09218
4 ORDER_PRICE_STOPLIMIT=0.0
5 ORDER_SL=0.0
6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
0 ORDER_SYMBOL=EURUSD
1 ORDER_COMMENT=
2 ORDER_EXTERNAL_ID=

```

然后，这个订单会被移到历史中（同时，至少状态、执行时间和仓位 ID 会发生变化）。

```

HistoryOrderCache added: 1311792104
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
0 ORDER_TIME_SETUP=2022.04.11 12:34:51
1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
2 ORDER_TIME_DONE=2022.04.11 12:34:51
3 ORDER_TYPE=ORDER_TYPE_BUY
4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
5 ORDER_TYPE_TIME=ORDER_TIME_GTC
6 ORDER_STATE=ORDER_STATE_FILLED
7 ORDER_MAGIC=0
8 ORDER_POSITION_ID=1311792104
9 ORDER_TIME_SETUP_MSC=2022.04.11 12:34:51'096
10 ORDER_TIME_DONE_MSC=2022.04.11 12:34:51'097
11 ORDER_POSITION_BY_ID=0
12 ORDER_TICKET=1311792104
13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
0 ORDER_VOLUME_INITIAL=0.01
1 ORDER_VOLUME_CURRENT=0.0
2 ORDER_PRICE_OPEN=1.09218
3 ORDER_PRICE_CURRENT=1.09218
4 ORDER_PRICE_STOPLIMIT=0.0
5 ORDER_SL=0.0
6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
0 ORDER_SYMBOL=EURUSD
1 ORDER_COMMENT=
2 ORDER_EXTERNAL_ID=
>>> positions: 0, orders: 1, history: 1

```

注意，这些修改发生在对 *OnTrade* 的同一次调用中。换言之，当我们的程序正在分析新订单的特性（通过调用 *orders.scan*）时，订单被终端并行处理，而在检查历史（通过调用 *history.scan*）时，该订单已经被移入历史。这就是为什么根据这个日志片段的最后一行，两处都列出了该订单的原因。这种行为对于多线程程序而言是正常的，在设计多线程程序时应考虑到这一点。但也并不总是如此。此处，我们只是提请注意这种情况。快速执行 MQL 程序时，这种情况通常不会发生。

如果我们先检查历史，然后再检查在线订单，那么在第一阶段，我们可能会发现订单还不在历史中，而在第二阶段，订单不再在线。也就是说，理论上它可能会丢失一会儿。更现实的情况是，由于历史同步，会跳过处于活跃阶段的订单，即，直接在历史中首次捕获到该订单。

注意，MQL5 不允许你将交易环境作为一个整体进行同步，而只是部分同步：

- 在活跃订单中，信息与刚刚调用 *OrderSelect* 或 *OrderGetTicket* 函数的订单相关
- 在仓位中，信息与刚刚调用函数 *PositionSelect*、*PositionSelectByTicket* 或 *PositionGetTicket* 的仓位相关
- 对于历史中的订单和交易，信息可在最后一次调用 *HistorySelect*、*HistorySelectByPosition* 和 *HistoryOrderSelect* 的上下文中获得 *HistoryDealSelect*

此外，请注意，交易事件（像任何 MQL5 事件一样）是关于已经发生的变化的消息，这些变更被放入队列中，并会以延迟的方式从队列中检索，并非在发生变更时立即能够检索到。此外，*OnTrade* 事件发生在相关的 *OnTradeTransaction* 事件之后。

尝试不同的程序配置，调试，并生成详细日志，可为你的交易系统选择最可靠的算法。

让我们回到我们的日志。在下次触发 *OnTrade* 时，情况已经得到修正：活跃订单的缓存检测到订单被删除。在这个过程中，仓位缓存可看到一个未平仓仓位。

```
>>> OnTrade(2)
PositionCache added: 1311792104
MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER,ENUM_POSITION_PROPERTY_DOUBLE,ENUM_PC
ENUM_POSITION_PROPERTY_INTEGER Count=9
0 POSITION_TIME=2022.04.11 12:34:51
1 POSITION_TYPE=POSITION_TYPE_BUY
2 POSITION_MAGIC=0
3 POSITION_IDENTIFIER=1311792104
4 POSITION_TIME_MSC=2022.04.11 12:34:51'097
5 POSITION_TIME_UPDATE=2022.04.11 12:34:51
6 POSITION_TIME_UPDATE_MSC=2022.04.11 12:34:51'097
7 POSITION_TICKET=1311792104
8 POSITION_REASON=POSITION_REASON_CLIENT
ENUM_POSITION_PROPERTY_DOUBLE Count=8
0 POSITION_VOLUME=0.01
1 POSITION_PRICE_OPEN=1.09218
2 POSITION_PRICE_CURRENT=1.09214
3 POSITION_SL=0.00000
4 POSITION_TP=0.00000
5 POSITION_COMMISSION=0.0
6 POSITION_SWAP=0.00
7 POSITION_PROFIT=-0.04
ENUM_POSITION_PROPERTY_STRING Count=3
0 POSITION_SYMBOL=EURUSD
1 POSITION_COMMENT=
2 POSITION_EXTERNAL_ID=
OrderCache removed: 1311792104
>>> positions: 1, orders: 0, history: 1
```

一段时间后，我们进行平仓。因为在我们的代码中，首先检查仓位缓存 (*positions.scan*)，所以对平仓仓位的更改会包含在日志中。

```
>>> OnTrade(8)
PositionCache changed: 1311792104
POSITION_PRICE_CURRENT: 1.09214 -> 1.09222
POSITION_PROFIT: -0.04 -> 0.04
```

此外，在同一次 *OnTrade* 调用中，我们可检测到一个成交订单的出现及其向历史的瞬时转移（同样，由于终端的快速并行处理）。


```

OrderCache added: 1311796883
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
0 ORDER_TIME_SETUP=2022.04.11 12:39:55
1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
2 ORDER_TIME_DONE=1970.01.01 00:00:00
3 ORDER_TYPE=ORDER_TYPE_SELL
4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
5 ORDER_TYPE_TIME=ORDER_TIME_GTC
6 ORDER_STATE=ORDER_STATE_STARTED
7 ORDER_MAGIC=0
8 ORDER_POSITION_ID=1311792104
9 ORDER_TIME_SETUP_MSC=2022.04.11 12:39:55'710
10 ORDER_TIME_DONE_MSC=1970.01.01 00:00:00'000
11 ORDER_POSITION_BY_ID=0
12 ORDER_TICKET=1311796883
13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
0 ORDER_VOLUME_INITIAL=0.01
1 ORDER_VOLUME_CURRENT=0.01
2 ORDER_PRICE_OPEN=1.09222
3 ORDER_PRICE_CURRENT=1.09222
4 ORDER_PRICE_STOPLIMIT=0.0
5 ORDER_SL=0.0
6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
0 ORDER_SYMBOL=EURUSD
1 ORDER_COMMENT=
2 ORDER_EXTERNAL_ID=
HistoryOrderCache added: 1311796883
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
0 ORDER_TIME_SETUP=2022.04.11 12:39:55
1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
2 ORDER_TIME_DONE=2022.04.11 12:39:55
3 ORDER_TYPE=ORDER_TYPE_SELL
4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
5 ORDER_TYPE_TIME=ORDER_TIME_GTC
6 ORDER_STATE=ORDER_STATE_FILLED
7 ORDER_MAGIC=0
8 ORDER_POSITION_ID=1311792104
9 ORDER_TIME_SETUP_MSC=2022.04.11 12:39:55'710
10 ORDER_TIME_DONE_MSC=2022.04.11 12:39:55'711
11 ORDER_POSITION_BY_ID=0
12 ORDER_TICKET=1311796883
13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
0 ORDER_VOLUME_INITIAL=0.01
1 ORDER_VOLUME_CURRENT=0.0
2 ORDER_PRICE_OPEN=1.09222
3 ORDER_PRICE_CURRENT=1.09222
4 ORDER_PRICE_STOPLIMIT=0.0
5 ORDER_SL=0.0
6 ORDER_TP=0.0

```

```

ENUM_ORDER_PROPERTY_STRING Count=3
0 ORDER_SYMBOL=EURUSD
1 ORDER_COMMENT=
2 ORDER_EXTERNAL_ID=
>>> positions: 1, orders: 1, history: 2

```

历史缓存中已有 2 个订单，但在历史缓存之前分析的仓位和活跃订单缓存尚未应用这些更改。

但是在下一个 *OnTrade* 事件中，我们可看到该仓位被平仓，市场订单已经消失。

```

>>> OnTrade(9)
PositionCache removed: 1311792104
OrderCache removed: 1311796883
>>> positions: 0, orders: 0, history: 2

```

如果我们在每一个分时报价处监控缓存（或者每秒一次，但不仅限于 *OnTrade* 事件），我们将随时看到 *ORDER_PRICE_CURRENT* 和 *POSITION_PRICE_CURRENT* 特性的变更。*POSITION_PROFIT* 也会变更。

我们的类没有 *persistence*，也就是说，它们只存在于 RAM 中，不知道如何在任何长期存储（比如文件）中保存和恢复它们的状态。这意味着程序可能会错过终端会话之间发生的变更。如果需要此类功能，应自己实现。接下来，在本书的第 7 章，我们将研究 MQL5 中内置的 SQLite 数据库支持，其提供了存储交易环境缓存和类似表格数据的最高效、最方便方式。

6.4.38 创建多交易品种 EA 交易

迄今为止，在本书的框架内，我们主要分析了在图表的当前工作交易品种上进行的 EA 交易示例。但是，MQL5 允许你为任何 *Market Watch* 交易品种生成交易订单，而与图表的工作交易品种无关。

事实上，前面几节中的许多示例都有一个 *symbol* 输入参数，你可以在其中指定一个任意交易品种。默认情况下，有一个空字符串，视为图表的当前交易品种。所以，我们已经考虑过下面的示例：

- *CustomOrderSend.mq5* 用于 发送交易请求
- *MarketOrderSend.mq5* 用于 买入和卖出运算
- *MarketOrderSendMonitor.mq5* 用于 读取活动订单特性的函数
- *PendingOrderSend.mq5* 用于 设置挂单
- *PendingOrderModify.mq5* 用于 修改挂单
- *PendingOrderDelete.mq5* 用于 删除挂单

你可以尝试使用不同的交易品种运行这些示例，并确保交易操作的执行与本地交易品种完全相同。

此外，正如我们在 *OnBookEven* 和 *OnTradeTransaction* 事件的说明中所看到的，它们是通用的，用于告知有关任意交易品种的交易环境变化。但对于 *OnTick* 事件而言，情况并非如此，仅在当前交易品种的新价格发生变化时才会生成该事件。通常这没有问题，但高频多币种交易需要采取额外的技术步骤，例如为其他交易品种订阅 *OnBookEvent* 事件或设置高频计时器。另一个绕过这种限制的选项是间谍指标 *EventTickSpy.mq5*，我们曾经在 *生成自定义事件* 中介绍过。

在讨论多交易品种交易支持的上下文中，应注意的是，多时间范围 EA 交易的类似概念并不完全正确。在新的柱线打开时间进行交易只是根据任意时段对分时报价进行分组的特例，不一定

是标准交易。当然，由于 `iTime(_Symbol, PERIOD_XX, 0)` 之类的函数，系统内核简化了对特定时间范围内出现的新柱线分析，但这种分析无论如何都是基于分时报价的。

你可以在你的 EA 交易中按分时报价数量，价格范围等构建虚拟柱线。在某些情况下，包括为了清楚起见，在 EA 交易之外以 **自定义交易品种** 显式生成此类“时间范围”也是合理的。但是这种方法也有其局限性：我们将在本书的下一部分讨论。

但是，如果交易系统仍然需要基于柱线开盘价的报价分析或者使用多币种指标，那么就应等待所有相关工具上的柱线同步。我们在 **跟踪柱线形成** 一节中提供了一个能执行次任务的类的示例。

开发多交易品种 EA 交易时，当务之急是将通用的交易算法分成不同的块。这些块可以随后应用于具有不同设置的各种交易品种。实现这一点的最合理的方法是在面向对象的编程 (OOP) 概念的框架内阐明一个或多个类。

我们用一个 EA 交易示例来说明这个方法，EA 交易采用了众所周知的鞅策略。众所周知，鞅策略本身是有风险的，因为它在每次交易失败后都会翻倍，以弥补之前的损失。降低这种风险至关重要，一个有效的方法是同时交易多个交易品种，最好是相关性弱的交易品种。通过这种方式，一种金融工具的暂时亏损可能会被其他工具的收益所抵消。

将各种金融工具（或单个交易系统内的不同设置，或甚至不同的交易系统）合并到 EA 交易中，有助于减少单个交易失败的整体影响。本质上，工具或系统越是多样化，其交易组成的孤立失败对最终结果的影响就越小。

我们调用一个新的 EA 交易 *MultiMartingale.mq5*。交易算法设置包括：

- **UseTime** UseTime - 启用/禁用预定交易的逻辑标志
- **HourStart** HourStart 和 **Hour End** - 如果 **UseTime** 为真，允许交易的时间范围（小时）*true*
- **Lots** Lots - 系列中第一笔交易的交易量
- **Factor** Factor - 损失后后续交易量增加的系数
- **Limit** Limit - 交易量倍增时亏损系列的最大交易量（之后，返回初始手数）
- **Stop Loss** Stop Loss 和 **Take Profit** - 到保护水平的距离（点数）
- **StartType** StartType - 第一笔交易的类型（买入或卖出）
- **Trailing** Trailing - 跟踪止损的指示

在源代码中，其说明如上所示。

```
input bool UseTime = true;           // UseTime (hourStart and hourEnd)
input uint HourStart = 2;            // HourStart (0...23)
input uint HourEnd = 22;             // HourEnd (0...23)
input double Lots = 0.01;            // Lots (initial)
input double Factor = 2.0;           // Factor (lot multiplication)
input uint Limit = 5;                // Limit (max number of multiplications)
input uint StopLoss = 500;           // StopLoss (points)
input uint TakeProfit = 500;         // TakeProfit (points)
input ENUM_POSITION_TYPE StartType = 0; // StartType (first order type: BUY or SELL)
input bool Trailing = true;          // Trailing
```

理论上，合理的做法是建立保护水平，不是用点数，而是用平均真实范围指标 (ATR) 的份额。但目前这并不是首要任务。

此外，EA 交易还集成了一种机制，用于在出现错误的情况下，在用户指定的时间内（由参数 *SkipTimeOnError* 控制）暂停交易操作。我们在此处省略这方面的详细讨论，因为它可以在源代码中引用。

为了将整个配置集合并到一个统一的实体中，定义了一个名为 *Settings* 的结构体。该结构体具有镜像输入变量的字段。此外，该结构体包括 *symbol* 字段，解决了其策略的多币种特性问题。换言之，交易品种可以是任意的，可与图表上的工作交易品种不同。

```
struct Settings
{
    bool useTime;
    uint hourStart;
    uint hourEnd;
    double lots;
    double factor;
    uint limit;
    uint stopLoss;
    uint takeProfit;
    ENUM_POSITION_TYPE startType;
    ulong magic;
    bool trailing;
    string symbol;
    ...
};
```

在最初的开发阶段，我们用输入变量填充该结构体。但是，这只能够交易单一的交易品种。随后，当我们扩展算法以包含多个交易品种时，就需要读取不同的设置集（使用不同的方法）并将它们附加到一个结构体数组中。

该结构体还包含几种有益的方法。具体而言，*validate* 方法用于验证设置的正确性，确认指定交易品种的存在，并返回一个成功指示符（*true*）。

```
struct Settings
{
    ...
    bool validate()
    {
        ...// checking the lot size and protective levels (see the source code)

        double rates[1];
        const bool success = CopyClose(symbol, PERIOD_CURRENT, 0, 1, rates) > -1;
        if(!success)
        {
            Print("Unknown symbol: ", symbol);
        }
        return success;
    }
    ...
};
```

调用 *CopyClose* 不仅可以检查 *Market Watch* 中的交易品种是否在线，还会在测试程序中加载其报价（所需时间范围的）和分时报价。如果不这样做，默认情况下，测试程序中仅能提供当前所

选工具和时间范围的报价和分时报价（在真实分时报价模式下）。由于我们正在编写一个多币种 EA 交易，将需要第三方报价和分时报价。

```
struct Settings
{
    ...
    void print() const
    {
        Print(symbol, (startType == POSITION_TYPE_BUY ? "+" : "-"), (float)lots,
            "*", (float)factor,
            "^", limit,
            "(", stopLoss, ",", takeProfit, ")",
            useTime ? "[" + (string)hourStart + "," + (string)hourEnd + "]" : "");
    }
};
```

`print` 方法可在一行中以缩写形式将所有字段输出到日志中。例如，

```
EURUSD+0.01*2.0^5(500,1000)[2,22]
|      | | | | | | | |
|      | | | | | | | `until this hour trading is allowed
|      | | | | | | | `from this hour trading is allowed
|      | | | | | | | `take profit in points
|      | | | | | | | `stop loss in points
|      | | | | | | | `maximum size of a series of losing trades (after '^')
|      | | | | | | | `lot multiplication factor (after '*')
|      | | | | | | | `initial lot in series
|      | | | | | | | `+ start with Buy
|      | | | | | | | `- start with Sell
|      | | | | | | | `instrument
```

当我们转向多币种时，需要 **Settings** 结构体中的其他方法。现在，我们想象一个简化版的 EA 交易在一个交易品种上进行交易的处理程序 *OnInit*。

```

int OnInit()
{
    Settings settings =
    {
        UseTime, HourStart, HourEnd,
        Lots, Factor, Limit,
        StopLoss, TakeProfit,
        StartType, Magic, SkipTimeOnError, Trailing, _Symbol
    };

    if(settings.validate())
    {
        settings.print();
        ...
        // here you will need to initialize the trading algorithm with these settings
    }
    ...
}

```

遵循面向对象的程序设计，广义的交易系统应描述为一个软件接口。同样，为了简化该示例，我们将在这个接口上仅使用一种方法：*trade*。

```

interface TradingStrategy
{
    virtual bool trade(void);
};

```

该算法的主要任务是交易，我们决定从何处调用这个方法并不重要：在 *OnTick* 的每个分时报价处，在柱线打开时，或者可能在计时器上。

你的工作 EA 交易很可能需要额外的接口方法来设置和支持各种模式。但是在这个示例中不需要。

我们开始创建一个基于接口的特定交易系统的类。在我们的示例中，所有的实例都是 *SimpleMartingale* 类。但是，也可以在一个 EA 交易中实现许多继承接口的不同类，然后按照任意组合方式统一使用它们。策略组合（最好在特性上区别很大）通常以财务表现的稳定性增加为特性。

```

class SimpleMartingale: public TradingStrategy
{
protected:
    Settings settings;
    SymbolMonitor symbol;
    AutoPtr<PositionState> position;
    AutoPtr<TrailingStop> trailing;
    ...
};

```

在这个类中，我们看到了一个熟悉的 *Settings* 结构体和工作交易品种监视器 *SymbolMonitor*。此外，我们还需要控制仓位的存在，并遵循其止损水平，为此我们引入了带有自动指针的变量，这些自动指针指向对象 *PositionState* 和 *TrailingStop*。使用自动指针，我们在代码中不必担心对象的显式删除，因为当控件退出该范围时，或者当新的指针被赋给自动指针时，可自动完成。

TrailingStop 类是一个基础类, 具有最简单的价格跟踪实现, 从中可以继承许多更复杂的算法, 我们认为其是一个派生的 *TrailingStopByMA*。因此, 为了让程序在将来具有灵活性, 最好确保调用代码可以传递自己特定的、定制的跟踪对象, 该对象是从 *TrailingStop* 派生的。例如, 这可以通过向构造函数传递一个指针或者将 *SimpleMartingale* 转换成一个模板类 (然后跟踪类将由模版参数设置) 来实现。

OOP 的这一原则被称为 *dependency injection*, 其和许多其他原则一起被广泛使用 (我们在 OOP 的理论基础: 组合中简要介绍过)。

这些设置作为构造函数参数传递给策略类。基于此, 我们可为所有内部变量赋值。

```
class SimpleMartingale: public TradingStrategy
{
    ...
    double lotsStep;
    double lotsLimit;
    double takeProfit, stopLoss;
public:
    SimpleMartingale(const Settings &state) : symbol(state.symbol)
    {
        settings = state;
        const double point = symbol.get(SYMBOL_POINT);
        takeProfit = settings.takeProfit * point;
        stopLoss = settings.stopLoss * point;
        lotsLimit = settings.lots;
        lotsStep = symbol.get(SYMBOL_VOLUME_STEP);

        // calculate the maximum lot in the series (after a given number of multiplicat
        for(int pos = 0; pos < (int)settings.limit; pos++)
        {
            lotsLimit = MathFloor((lotsLimit * settings.factor) / lotsStep) * lotsStep;
        }

        double maxLot = symbol.get(SYMBOL_VOLUME_MAX);
        if(lotsLimit > maxLot)
        {
            lotsLimit = maxLot;
        }
        ...
    }
}
```

接下来, 我们使用 *PositionFilter* 对象来搜索现有的“自己的”仓位 (通过魔术编号和交易品种)。如果找到了此类仓位, 即可创建 *PositionState* 对象, 如果需要, 还可以创建 *TrailingStop* 对象。

```

PositionFilter positions;
ulong tickets[];
positions.let(POSITION_MAGIC, settings.magic).let(POSITION_SYMBOL, settings.symbol)
    .select(tickets);
const int n = ArraySize(tickets);
if(n > 1)
{
    Alert(StringFormat("Too many positions: %d", n));
}
else if(n > 0)
{
    position = new PositionState(tickets[0]);
    if(settings.stopLoss && settings.trailing)
    {
        trailing = new TrailingStop(tickets[0], settings.stopLoss,
            ((int)symbol.get(SYMBOL_SPREAD) + 1) * 2);
    }
}
}

```

时间表操作将暂时留在 *trade* 方法 (*useTime*、*hourStart* 和 *hourEnd* 参数字段)的“幕后”。我们直接来看交易算法。

如果不存在并且还没有任何仓位, *PositionState* 指针将为零, 需要根据选择的方向 *startType* 建立多头或空头仓位。

```

virtual bool trade() override
{
    ...
    ulong ticket = 0;

    if(position[] == NULL)
    {
        if(settings.startType == POSITION_TYPE_BUY)
        {
            ticket = openBuy(settings.lots);
        }
        else
        {
            ticket = openSell(settings.lots);
        }
    }
    ...
}

```

此处使用了辅助方法 *openBuy* 和 *openSell*。我们将在几个段落中进行讨论。现在, 我们只需要知道他们是否在成功时返回订单号, 失败时返回 0。

如果 *position* 对象已经包含关于被跟踪仓位的信息, 可通过调用 *refresh* 来检查其是否是活动的。如果成功 (*true*), 则通过调用 *update* 更新仓位信息, 还可以根据设置的要求来跟踪止损。


```

else // position[] != NULL
{
    if(position[].refresh()) // does position still exists?
    {
        position[].update();
        if(trailing[]) trailing[].trail();
    }
    ...

```

如果仓位被平仓, *refresh* 将返回 *false*, 我们可在另一个 *if* 分支中打开一个新的仓位: 如果利润是固定的, 则在相同的方向上打开; 如果发生了亏损, 则在相反的方向上打开。请注意, 我们在缓存中仍然有先前仓位的快照。

```

else // the position is closed - you need to open a new one
{
    if(position[].get(POSITION_PROFIT) >= 0.0)
    {
        // keep the same direction:
        // BUY in case of profitable previous BUY
        // SELL in case of profitable previous SELL
        if(position[].get(POSITION_TYPE) == POSITION_TYPE_BUY)
            ticket = openBuy(settings.lots);
        else
            ticket = openSell(settings.lots);
    }
    else
    {
        // increase the lot within the specified limits
        double lots = MathFloor((position[].get(POSITION_VOLUME) * settings.fa

        if(lotsLimit < lots)
        {
            lots = settings.lots;
        }

        // change the trade direction:
        // SELL in case of previous unprofitable BUY
        // BUY in case of previous unprofitable SELL
        if(position[].get(POSITION_TYPE) == POSITION_TYPE_BUY)
            ticket = openSell(lots);
        else
            ticket = openBuy(lots);
    }
}
}
...

```

在这个最后阶段出现非零订单号意味着我们必须开始用新对象 *PositionState* 和 *TrailingStop* 进行控制。

```

    if(ticket > 0)
    {
        position = new PositionState(ticket);
        if(settings.stopLoss && settings.trailing)
        {
            trailing = new TrailingStop(ticket, settings.stopLoss,
                ((int)symbol.get(SYMBOL_SPREAD) + 1) * 2);
        }
    }

    return true;
}

```

我们现在用一些缩写来表示 *openBuy* 方法 (*openSell* 也是一样的)。有三个步骤：

- 使用 *prepare* 方法准备 *MqlTradeRequestSync* 结构体 (此处未显示, 其填充了 *deviation* 和 *magic*)
- 使用 *request.buy* 方法调用发送订单
- 用 *postprocess* 方法检查结果 (此处未显示, 其调用 *request.completed*, 如果出现错误, 交易暂停期并从预期的更好条件开始);

```

ulong openBuy(double lots)
{
    const double price = symbol.get(SYMBOL_ASK);

    MqlTradeRequestSync request;
    prepare(request);
    if(request.buy(settings.symbol, lots, price,
        stopLoss ? price - stopLoss : 0,
        takeProfit ? price + takeProfit : 0))
    {
        return postprocess(request);
    }
    return 0;
}

```

通常情况下, 仓位会被止损或止盈平仓。但是, 我们支持可能导致平仓的预定操作。我们回到预定工作的 *trade* 方法的开始。

```

virtual bool trade() override
{
    if(settings.useTime && !scheduled(TimeCurrent())) // time out of schedule?
    {
        // if there is an open position, close it
        if(position[] && position[].isReady())
        {
            if(close(position[].get(POSITION_TICKET)))
            {
                // at the request of the designer:
                position = NULL; // clear the cache or we could...
                // do not do this zeroing, that is, save the position in the cache,
                // to transfer the direction and lot of the next trade to a new series
            }
            else
            {
                position[].refresh(); // guaranteeing reset of the 'ready' flag
            }
        }
        return false;
    }
    ...// opening positions (given above)
}

```

工作方法 *close* 与 *openBuy* 高度相似，所以我们此处不予考虑。另一种方法 *scheduled* 仅返回 *true* 或 *false*，具体取决于当前时间是否在指定的工作时间范围内 (*hourStart* 和 *hourEnd*)。

到此，交易类准备就绪。但是对于多币种工作，你需要创建几个副本。*TradingStrategyPool* 类负责对其管理，其中我们说明了指向 *TradingStrategy* 的指针数组及其补充方法：参数型构造函数 和 *push*。

```

class TradingStrategyPool: public TradingStrategy
{
private:
    AutoPtr<TradingStrategy> pool[];
public:
    TradingStrategyPool(const int reserve = 0)
    {
        ArrayResize(pool, 0, reserve);
    }

    TradingStrategyPool(TradingStrategy *instance)
    {
        push(instance);
    }

    void push(TradingStrategy *instance)
    {
        int n = ArraySize(pool);
        ArrayResize(pool, n + 1);
        pool[n] = instance;
    }

    virtual bool trade() override
    {
        for(int i = 0; i < ArraySize(pool); i++)
        {
            pool[i][].trade();
        }
        return true;
    }
};

```

没有必要使策略池从 *TradingStrategy* 接口派生，但是如果我们这样做，则允许将来将策略池打包到其他更大的策略池中。*trade* 方法仅为在所有数组对象上调用相同的方法。

在全局上下文中，我们向交易池添加一个自动指针，在 *OnInit* 处理程序中，我们应确保其填充。我们可以从一个单一的策略开始（稍后我们将处理多币种问题）。

```

AutoPtr<TradingStrategyPool> pool;

int OnInit()
{
    ... // settings initialization was given earlier
    if(settings.validate())
    {
        settings.print();
        pool = new TradingStrategyPool(new SimpleMartingale(settings));
        return INIT_SUCCEEDED;
    }
    else
    {
        return INIT_FAILED;
    }
    ...
}

```

要开始交易，我们只需要编写以下小处理程序 *OnTick*。

```

void OnTick()
{
    if(pool[] != NULL)
    {
        pool[].trade();
    }
}

```

但是多币种支持怎么处理呢？

当前的输入参数集仅适用于一种金融工具。我们可以利用这一点来测试并优化单个交易品种的 EA 交易，但在为所有交易品种找到最佳设置后，需要以某种方式将它们组合起来并传递给算法。

在这种情况下，我们应用最简单的解决方案。上面的代码包含一行由 *Settings* 结构体生成的 *print* 方法形成的设置。我们可在 *parse* 结构体中实现该方法，该结构体可执行相反操作：通过行说明恢复字段的状态。此外，由于我们需要连接不同字符的几个设置，因此同意可以通过一个特殊的分隔符将它们连接成一个长字符串，例如 ‘;’。这样一来，编写 *parseAll* 静态方法来读取合并后的设置集就很简单了，该方法将调用 *parse* 来填充通过引用传递的 *Settings* 结构体数组。这些方法的完整源代码可以在附件中找到。

```

struct Settings
{
    ...
    bool parse(const string &line);
    void static parseAll(const string &line, Settings &settings[])
    ...
};

```

例如，以下串联字符串包含三个交易品种的设置。

```
EURUSD+0.01*2.0^7(500,500)[2,22];AUDJPY+0.01*2.0^8(300,500)[2,22];GBPCHF+0.01*1.7^8(1
```

`parseAll` 方法可以解析的就是这类行。为了将此类字符串输入到 EA 交易中，我们定义了输入变量 `WorkSymbols`。

```
input string WorkSymbols = ""; // WorkSymbols (name±lots*factor^limit(sl,tp)[start,stop];...)
```

如果为空，EA 交易将使用之前提供的各个输入变量的设置。如果指定了字符串，`OnInit` 处理程序将根据解析该行的结果填充交易系统池。

```
int OnInit()
{
    if(WorkSymbols == "")
    {
        ...// work with the current single character, as before
    }
    else
    {
        Print("Parsed settings:");
        Settings settings[];
        Settings::parseAll(WorkSymbols, settings);
        const int n = ArraySize(settings);
        pool = new TradingStrategyPool(n);
        for(int i = 0; i < n; i++)
        {
            settings[i].trailing = Trailing;
            // support multiple systems on one symbol for hedging accounts
            settings[i].magic = Magic + i; // different magic numbers for each subsystem
            pool[i].push(new SimpleMartingale(settings[i]));
        }
    }
    return INIT_SUCCEEDED;
}
```

需要注意的是，在 MQL5 中，输入字符串的长度被限制为 250 个字符。此外，在测试程序的优化过程中，字符串被进一步截断到最多 63 个字符。因此，为了优化跨多个交易品种的同步交易，必须设计一种替代方法来加载该设置，比如从文本文件中检索设置。如果用文件名而不是包含设置的字符串来指定输入变量，这可以通过使用相同的输入变量轻松实现。

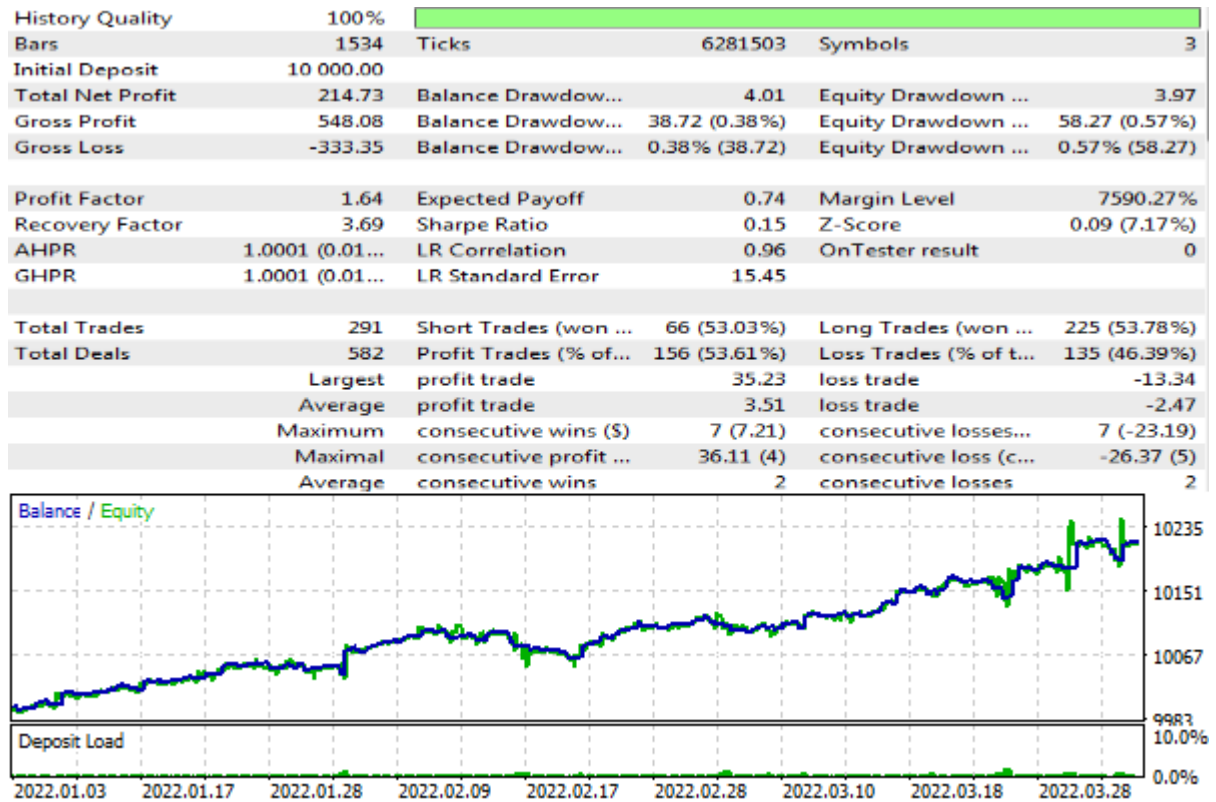
这种方法在上面提到的 `Settings::parseAll` 方法中实现。根据适用于所有类似情况的通用原则来设置文本文件的名称，在该文本文件中，输入字符串可无长度限制地传递给 EA 交易：文档名以 EA 交易名称开始，然后，在连字符之后，必须有文件包含其数据的变量名称。例如，就我们的情况而言，在 `WorkSymbols` 输入变量中，你可以选择指定文件名 `"MultiMartingale-WorkSymbols.txt"`。然后 `parseAll` 方法将尝试从该文件中读取文本（应在标准 `MQL5/Files` 沙箱中）。

在输入参数中传递文件名需要采取额外的步骤来进一步测试和优化此类 EA 交易：应将 `#property tester_file "MultiMartingale-WorkSymbols.txt"` 指令添加到源代码中。这将在 [测试程序预处理器指令](#) 一节中详细讨论。添加该指令后，EA 交易将要求该文件存在，并且不会在测试程序中没有该文件的情况下启动！

EA 交易准备就绪。我们可以分别针对不同的交易品种进行测试，为每个交易品种选择最佳设置，并建立一个交易组合。在下一章中，我们将研究测试程序的 API，包括优化，这个 EA 交易马上就能使用了。同时，我们检查一下其多币种运算。

```
WorkSymbols=EURUSD+0.01*1.2^4(300,600)[9,11];GBPCHF+0.01*2.0^7(300,400)[14,16];AUDJPY
```

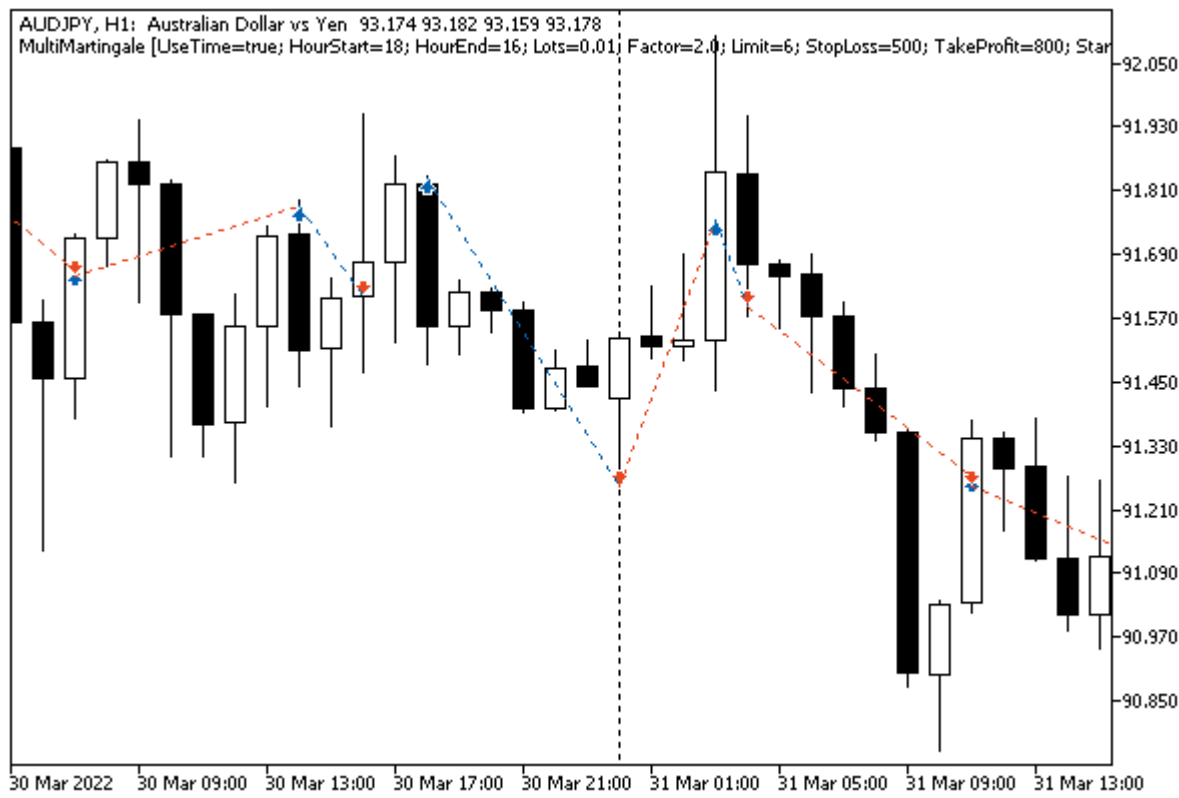
在 2022 年第一季度，我们将收到以下报告 MetaTrader 5 报告不提供按交易品种细分的统计数据，因此仅通过交易/订单/仓位表即可区分单币种报告和多币种报告）。



多币种策略 EA 交易的测试程序报告

需要注意的是，由于策略是从 *OnTick* 处理程序中启动的，所以在不同的主交易品种（即在测试程序设置下拉菜单中选择的交易品种）上运行将会产生略微不同的结果。在我们的测试中，我们简单地将 EURUSD 作为流动性最强、最常用的分时报价金融工具，这对于大多数应用而言已经足够了。但是，如果你想要对所有工具的分时报价做出反应，你可以使用类似 *EventTickSpy.mq5* 的指标。或者，你可以在计时器上运行交易逻辑，而不必绑定到特定工具的分时报价。

这是一个单一交易品种的交易策略，在这个示例中为 AUDJPY。



多币种策略 EA 交易测试图

顺便说一下，对于所有的多币种 EA 交易而言，还有一个重要的问题没有被注意到。我们讨论的是选择手数大小的方法，例如，基于存款或风险的负载。之前，我们在非交易性 EA 交易 [LotMarginExposureTable.mq5](#) 中演示过此类计算的例子。在 [MultiMartingale.mq5](#) 中，我们通过选择并在每个交易品种的设置中显示固定手数来简化任务。但是，在进行多币种 EA 交易中，根据工具价值（通过保证金或波动性）按比例选择手数是有益的。

总之，我想指出的是，多币种策略可能需要不同的优化原则。所考虑的策略使得有可能分别找到交易品种的参数，然后将它们组合起来。但是，一些套利和集群策略（例如，配对交易）是基于所有工具的同步分析来制定交易决策的。在这种情况下，与所有交易品种相关的设置均应单独包含在输入参数中。

6.4.39 EA 交易的优势和局限性

由于 EA 交易进行某些特定操作，因此存在一定的局限性，但相比其他类型 MQL 程序也有一些优势。特别是，所有用于指标的函数在 EA 交易中都被禁用了：

- ⌚ [SetIndexBuffer](#)
- ⌚ [IndicatorSetDouble](#)
- ⌚ [IndicatorSetInteger](#)
- ⌚ [IndicatorSetString](#)
- ⌚ [PlotIndexSetDouble](#)
- ⌚ [PlotIndexSetInteger](#)
- ⌚ [PlotIndexSetString](#)
- ⌚ [PlotIndexGetInteger](#)

此外，EA 交易不应描述通常为其他类型程序特有的事件处理程序：*OnStart*（脚本和服务）和 *OnCalculate*（指标）。

与指标不同，每个图表上只能设置一个 EA 交易。

同时，EA 交易是唯一一种除了可以进行测试（我们已经对指标和 EA 交易进行了测试），还可以进行优化的 MQL 程序。优化程序允许根据各种标准（包括交易和抽象数学标准），找到最佳输入参数。出于这些目的，API 包含了额外的一些函数和几个特定的事件处理程序。我们将在下一章学习这方面的内容。

此外，EA 交易（以及脚本和服务，即除指标之外的所有类型的程序）可以使用多组内置 MQL5 函数，用于在套接字级别和各种互联网协议（HTTP、FTP 和 SMTP）下使用网络。我们将在本书的第七部分讨论这些内容。

6.4.40 在 MQL 向导中创建 EA 交易

至此，我们即将学完有关开发 EA 交易的交易 API。在这一章中，我们探讨了各种示例，你可以将其作为你自己项目的起点。但是，如果你想从零开始创建 EA 交易，也不必真得“从零开始”。MetaEditor 提供了内置的 MQL 向导，可用于创建 EA 交易模板。此外，在 EA 交易的情况下，该向导提供了两种不同的方法来生成源代码。

我们已经在 [MQL 向导和程序草稿](#) 一节中熟悉了该向导的第 1 步。显然，在第 1 步中，我们应选择要创建的项目类型。在上一章中，我们创建了一个脚本模板。之后，在关于指标的一章中，我们介绍了如何 [创建指标模板](#)。现在我们将考虑以下两个选项：

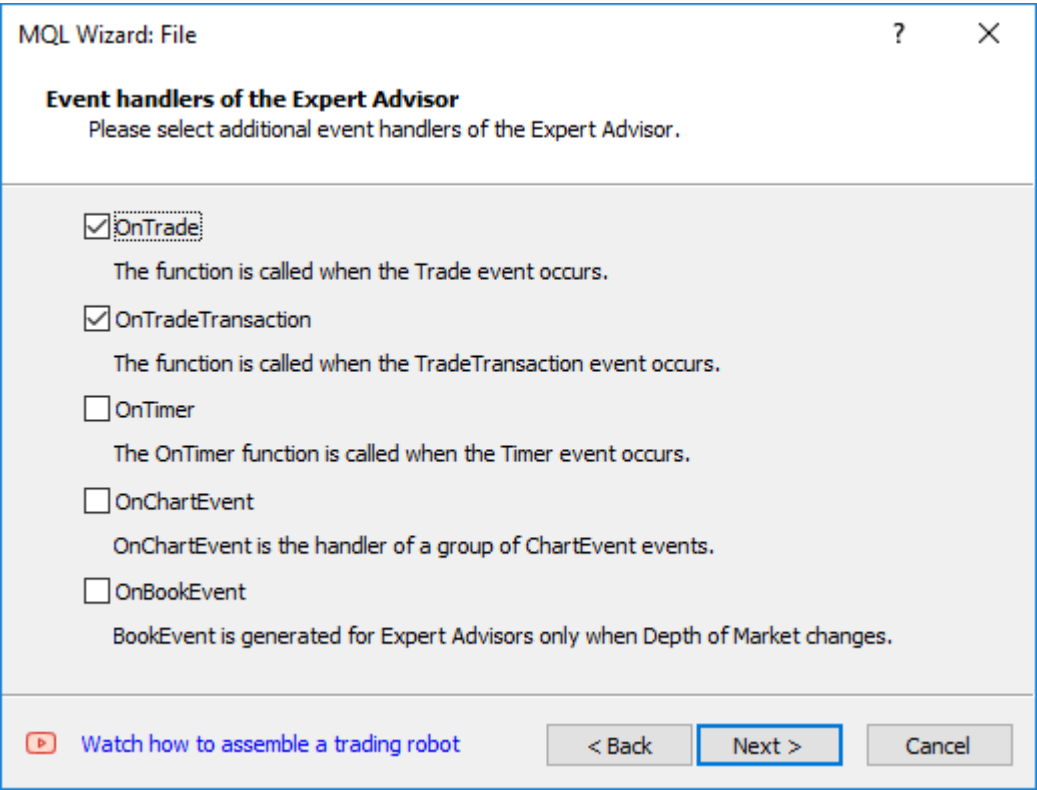
- EA 交易（模板）
- EA 交易（生成）

第一个选项更简单。你可以选择一个名称、输入参数及其所需的事件处理程序，如下面的截屏所示，但是在产生的源文件中不会有交易逻辑和现成的算法。

第二个选项更复杂。选择该选项将生成一个基于标准库的现成 EA 交易，该标准库在标准 MetaTrader 5 包中提供了头文件中的一组类。文件位于 *MQL5/Include/Expert/*、*MQL5/Include/Trade*、*MQL5/Include/Indicators* 及其他几个文件夹中。库类实现了最流行的指标信号、基于信号组合执行交易操作的机制以及资金管理和跟踪止损算法。对标准库的详细研究超出了本书的范围。

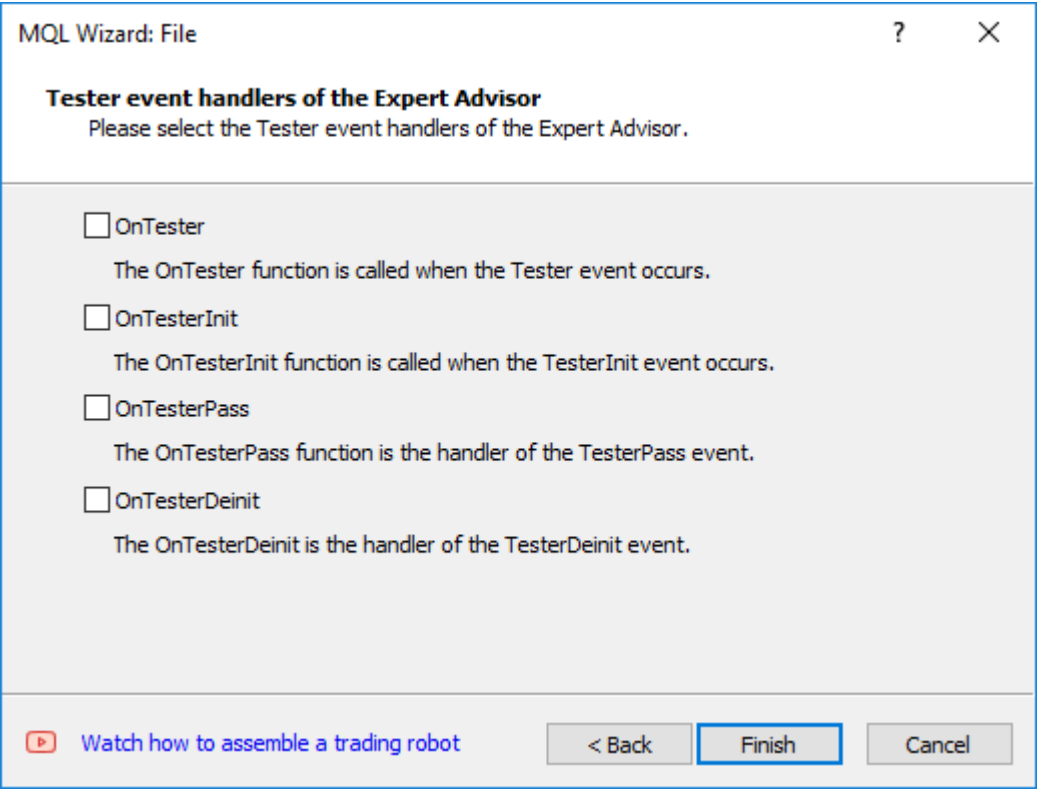
无论你选择哪个选项，在向导的第 2 步，都需要输入 EA 交易名称和输入参数。该步骤的外观与 [MQL 向导和程序草稿](#) 一节中显示的内容相似。唯一需要注意的是，基于标准库的 EA 交易必须有两个强制参数（不可移除）：*Symbol* 和 *TimeFrame*。

对于一个简单的模板，在第 3 步，除了 *OnTick*（始终会插入 *OnTick*）之外，建议选择将要添加到源代码的附加事件处理程序。



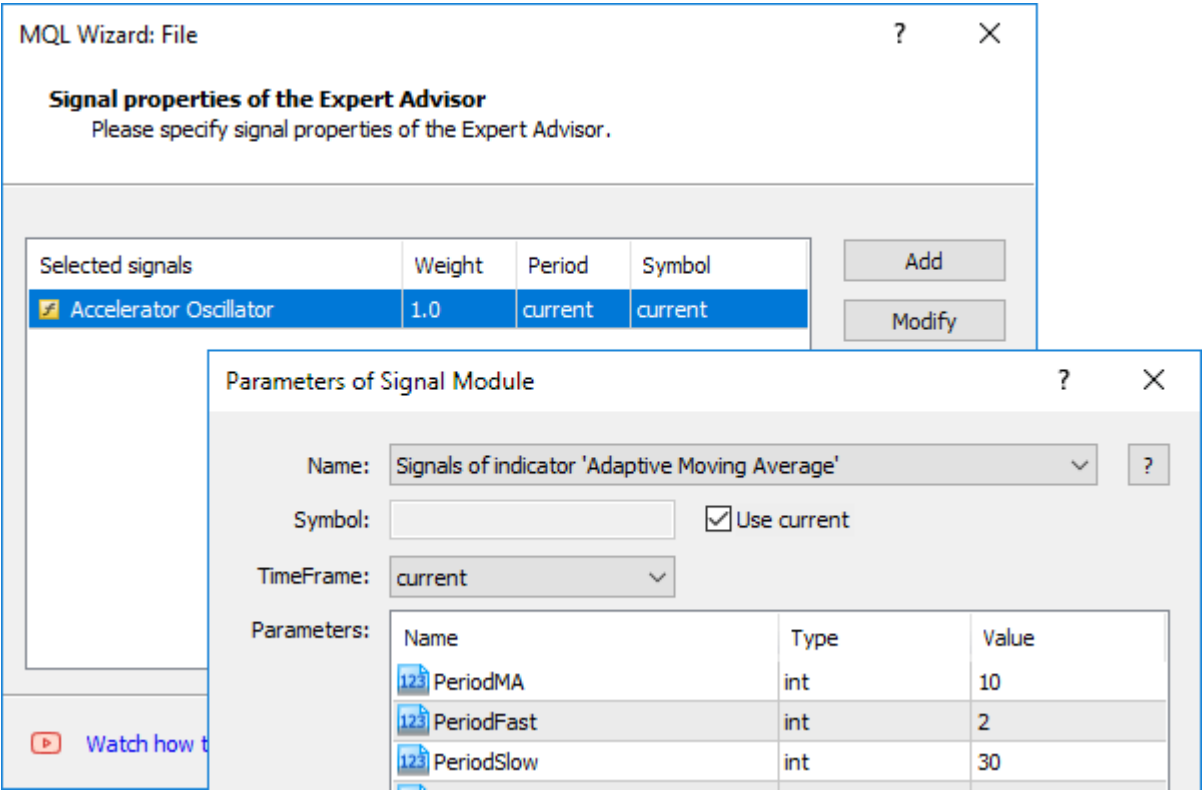
创建 EA 交易模板。第 3 步.其他事件处理程序

最后的第 4 步允许你为测试程序指定一个或多个可选的事件处理程序。将在下一章讨论这些事件处理程序。



创建 EA 交易模板。第 4 步.测试程序的事件处理程序

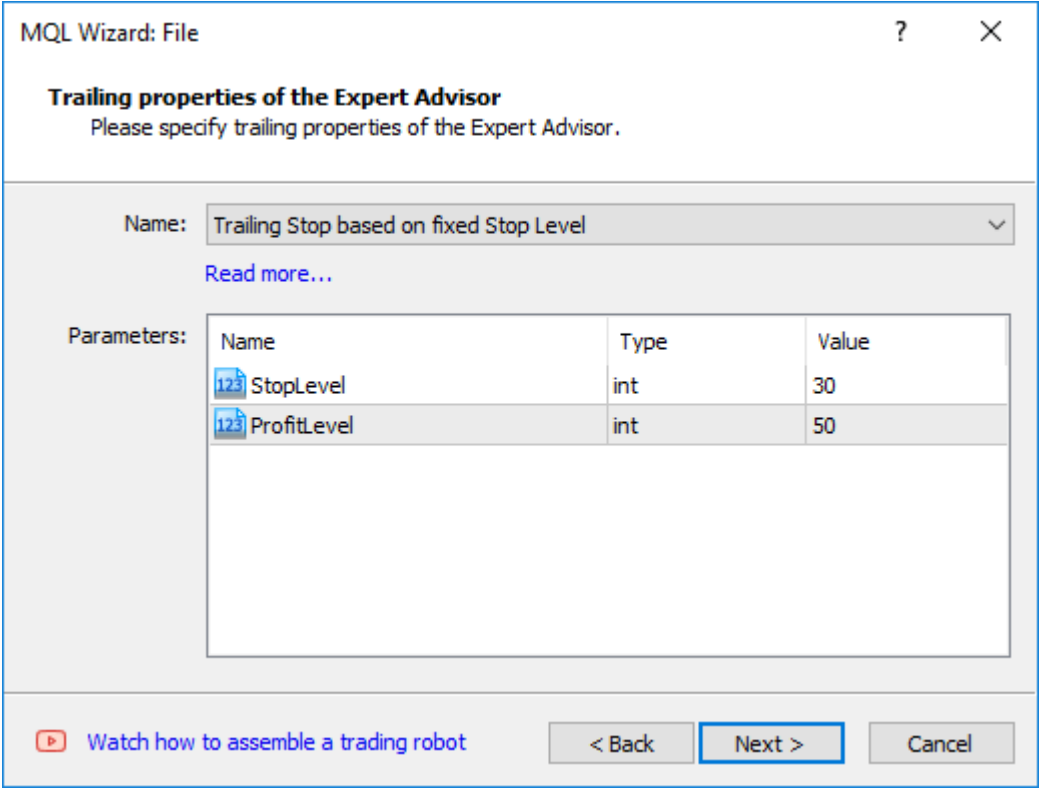
如果用户在该向导的第 1 步选择生成一个基于标准库的程序，那么第 3 步为设置交易信号。



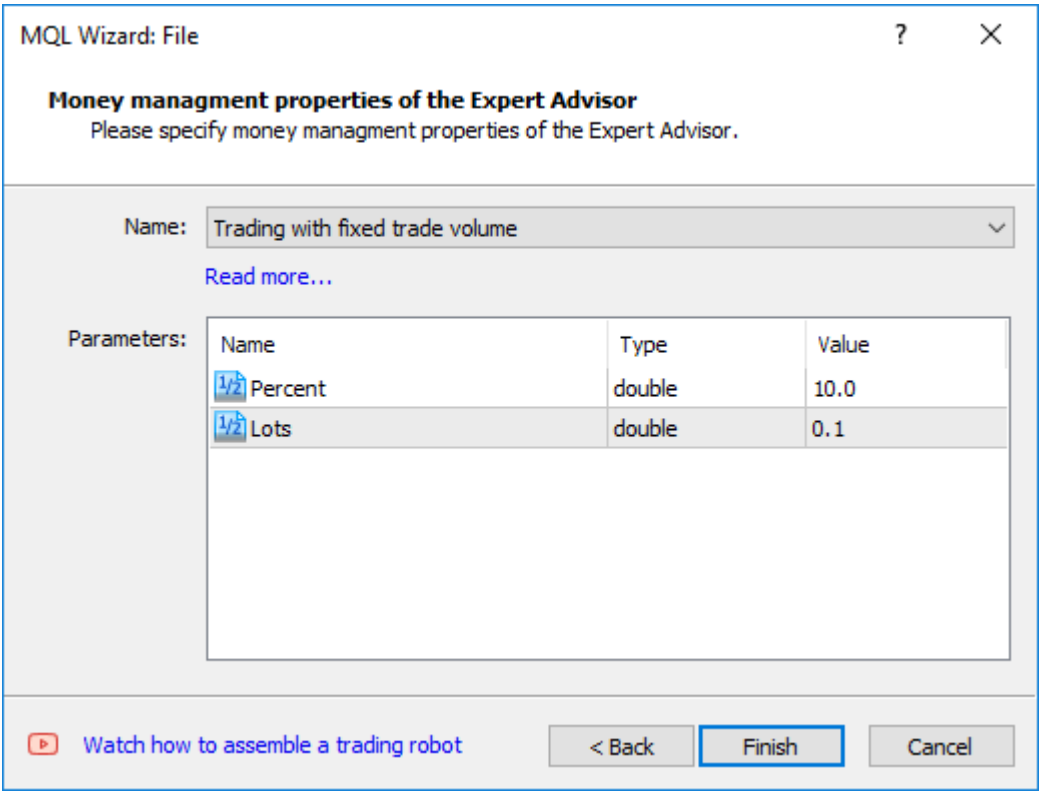
生成现成的 EA 交易。第 3 步.设置交易信号

你可以在本 文档中了解更多信息。

第 4 步和第 5 步旨在将跟踪纳入到 EA 交易中，并根据预定义的方法之一自动选择手数。



生成现成的 EA 交易。第 4 步.选择跟踪止损方法



生成现成的 EA 交易。第 5 步.选择手数

当然，该向导并非万能工具，生成的程序原型通常也需要改进。但是，本章学到的知识会让你对生成的源代码更有信心，并可根据需要扩展。

6.5 EA 交易的测试和优化

EA 交易的开发不仅意味着可在 MQL5 中实现交易策略，而且在更大程度上意味着可测试其财务表现，寻找最佳设置，并在各种情况下进行调试（寻找并纠正错误）。所有这些都可以在集成的 MetaTrader 5 测试程序中完成。

该测试程序适用于各种货币，支持多种逐笔交易明细生成模式：基于所选时段的开盘价、基于 M1 时段的 OHLC 价、基于人工生成的逐笔交易明细，以及基于真实的逐笔交易明细历史记录。这样您可以选择交易模拟的速度和精度的最佳比例。

测试程序设置允许您设置过去的测试时间间隔、存款规模和杠杆；测试程序可用于模拟重新报价和特定账户特性（包括佣金规模、保证金、会话时间表、限制手数）。从用户的角度来看，使用测试程序的所有详细信息都可以在 终端文档 中找到。

前面，我们已经简要地讨论了测试程序的操作，特别是在 测试指标 一节。让我们回忆一下，与 EA 交易不同，测试程序控制函数及其优化对于指标是不可用的。但是，我个人认为，我希望看到一个自适应的指标自我调整选项：所需要的就是在这些选项中支持 *OnTester* 处理程序，我们将在 单独的章节 中介绍。

如您所知，有多种模式可用于优化，例如 EA 交易输入参数组合的直接枚举、加速遗传算法、数学计算或 *Market Watch* 中符号的按顺序运行。作为一条优化标准，您可以使用众所周知的指标，如盈利能力、夏普比率、恢复系数和预期回报，以及由 EA 交易开发人员嵌入到源代码中的“自定义”变量。在本书的上下文中，假设读者已经熟悉了设置、运行和解释优化结果的原

则，因为在本章中我们将开始学习测试程序控制 API。感兴趣的人可以在 [文档](#) 相关章节的帮助下更新他们的知识。

测试程序的一个特别重要的函数是多线程优化，可以使用本地和分布式（网络）智能体程序来执行，包括 MQL5 云网络中的程序。由用户手动启动的单个测试运行（具有特定的输入参数），或者在优化期间调用的许多运行之一（当我们在给定范围内实现参数值的枚举时），是在一个单独的程序 - 智能体中执行的。从技术上讲，这是一个 `metatester64.exe` 文件，在测试和优化过程中，可以在 Windows 任务管理器中看到其进程的副本。因此，该测试程序是多线程的。

终端为将任务分配给本地和远程智能体的调度程序。如有必要，可启动本地智能体。优化时，默认启动几个智能体；其数量对应于处理器内核的数量。在使用指定参数执行下一个测试 EA 交易的任务后，智能体可将结果返回给终端。

每个智能体可创建自己的交易和软件环境。所有智能体相互隔离，并与客户终端隔离。

特别是，智能体有自己的全局变量和自己的 [文件沙箱](#)，包括写入详细智能体日志的文件夹：**Tester/Agent-IPaddress-Port/Logs**。此处，**Tester** 为测试程序安装目录（在与 MetaTrader 5 一起进行标准安装时，指安装终端的子文件夹）。目录 **Agent-IPaddress-Port** 的名称包含用于与终端通信的特定网络地址和端口值，而不是 **IPaddress** 和 **Port**。对于本地智能体，默认情况下，为地址 127.0.0.1 和从 3000 开始的端口范围（例如，在具有 4 个内核的计算机上，我们将看到端口 3000、3001、3002 和 3003 上的智能体）。

测试 EA 交易时，所有文件操作都在 **Tester/Agent-IPaddress-Port/MQL5/Files** 文件夹中执行。但是，可以通过 [共享文件夹](#) 实现本地智能体和客户终端之间（以及同一台计算机上终端的不同副本之间）的交互。为此，当使用 [FileOpen](#) 函数打开文件时，必须指定 `FILE_COMMON` 标志。将数据从智能体传送到终端的另一种方式由 [帧](#) 机制提供。

出于安全原因，智能体的本地沙箱会在每次测试前自动清空（以防止不同的 EA 交易读取彼此的数据）。

在每个智能体的文件沙箱旁边创建一个包含报价历史记录的文件夹：**Tester/Agent-IPaddress-Port/bases/ServerName/Symbol/**。在下一节中，我们简要地告诉您它是如何形成的。

各个测试运行和优化的结果由终端存储在一个特殊的缓存中，该缓存可以在安装目录的 **Tester/cache/** 子文件夹中找到。测试结果存储在扩展名为 **tst** 的文件中，优化结果存储扩展名为 **opt** 的文件中。这两种格式均为 MetaQuotes 开发者开源的，所以您可以实现自己的批量分析数据处理，或者使用 [mql5.com](#) 网址上代码库中现成的源代码。

在本章中，首先，我们将考虑 MQL 程序如何在测试程序中工作的基本原理，然后我们将学习如何在实践中与其进行交互。

6.5.1 在测试程序中生成分时报价

不强制要求 EA 交易中必须存在 **OnTick** 处理程序才能在测试程序中进行测试。EA 交易可以使用一个或多个其他熟悉的函数：

- **OnTick** - 新分时报价到达的事件处理程序
- **OnTrade** - 交易事件处理程序
- **OnTradeTransaction** - 交易处理程序
- **OnTimer** - 计时器信号处理程序
- **OnChartEvent** - 图表（包括自定义图表）上的事件处理程序

同时，在测试程序内部，时间过程的主要等价指标是一串分时报价，其中不仅包含价格变化，还包含精确到毫秒的时间。因此，为了测试 EA 交易，有必要生成分时报价序列。MetaTrader 5 测试程序有 4 种分时报价生成模式：

- 真实分时报价（如果经纪商提供了其历史）
- 每条分时报价（基于可用的 M1 时段报价的模拟）
- 分钟柱线的 OHLC 价格（1 分钟 OHLC）
- 仅开盘价（每根柱线 1 笔分时报价）

另一种操作模式是数学计算，我们将在后面分析，因为这种模式与报价和分时报价无关。

无论用户选择 4 种模式中的哪一种，终端都会加载可用的历史数据进行测试。如果选择真实分时报价模式，并且经纪商没有该金融工具的真实分时报价，则使用“所有分时报价”模式。测试程序在其报告中以图形和百分比形式指示分时报价生成的特性（其中 100% 表示所有分时报价都是真实的）。

在开始测试过程之前，终端会从交易服务器同步并下载测试程序设置中所选金融工具的历史。同时，终端会第一次从交易服务器下载历史到需要的深度（留有一定裕量，视时间范围而定，至少在测试开始前 1 年），以免以后申请。之后，只会下载新数据。所有这些都会在测试程序日志中生成相应消息。

测试代理在测试开始后应立即从客户终端接收被测金融工具的历史。如果测试过程使用其他金融工具上的数据（例如，这是一个多币种 EA 交易），那么在这种情况下，测试代理会在第一次调用时从客户终端请求所需的历史。如果终端上有可用的历史数据，这些数据会立即传输到测试代理。如果数据丢失，终端将请求并从服务器下载数据，然后将其传送到测试代理。

在交易操作期间计算交叉汇率价格时，还会使用其他金融工具。例如，当测试存款货币为美元的 EURCHF 策略时，在处理第一笔交易操作之前，测试代理将从客户终端请求 EURUSD 和 USDCHF 的历史，尽管该策略并不直接引用这些金融工具。

在这方面，在测试多币种策略之前，建议首先将所有必要的历史数据下载到客户终端。这将有助于避免与恢复数据相关的测试/优化延迟。例如，你可以通过打开相应的图表并将其滚动到历史的开头来下载历史。

现在我们更详细地了解一下分时报价生成模式。

来自历史数据的真实分时报价

对真实分时报价的测试和优化应尽可能接近真实条件。这些是交易所和流动性提供商的报价。

如果在交易品种的历史中有一个分钟柱线，但没有该分钟的分时报价数据，测试程序将在“每条分时报价”模式下生成分时报价（详见下文）。这允许你在经纪商的不完整分时报价数据的情况下，在测试程序中构建正确的图表。此外，由于各种原因，分时报价数据可能与分钟柱线不匹配。例如，由于从源到客户终端的数据传输断开或其他故障。测试时，分钟数据被认为更可靠。

分时报价存储在策略测试程序的交易品种缓存中。缓存大小为不超过 128,000 条分时报价。当新的分时报价到达时，最旧的数据会被推出。但是，使用 **CopyTicks** 函数，你可以获得缓存之外的分时报价（仅当使用真实分时报价进行测试时）。在这种情况下，将从测试程序的分时报价数据库请求数据，该数据库完全对应于客户终端的类似数据库。不会对该基数进行分钟柱线调整。因此，其中的分时报价可能与缓存中的分时报价不同。

每条分时报价 (模拟)

如果没有真实分时报价历史,或者如果你需要最小化网络流量(因为实际分时报价的存档会消耗大量资源),你可以选择基于 M1 时间范围的可用报价人工生成分时报价。

金融工具的报价历史可以紧密打包的分钟柱线块形式从交易服务器传输到 MetaTrader 5 的客户终端。历史查询过程和所需时间范围的构造在 [时间序列的组织 and 存储技术特性](#) 章节中详细介绍过。

价格历史的最小元素是一个分钟柱线,从中可以获得关于四个 OHLC 价格值的信息:开盘价、最高价、最低价和收盘价。

新的分钟柱线不是在新的一分钟开始时(秒钟数变为 0)打开,而是在出现分时报价(至少一个点的价格发生变化)时打开。类似地,我们不能从一秒精度的柱线确定对应于该分钟柱线收盘价的分时报价到达时间:我们只知道一分钟柱线的最新价,记录为收盘价。

因此,对于每个分钟柱线,我们知道 4 个控制点,我们可以肯定地说价格已经存在了。如果该柱线只有 4 条分时报价,那么这个信息就足够测试了,但是通常分时报价的数量都超过 4 个。这意味着需要为 **Open**、**High**、**Low** 和 **Close** 价格之间的分时报价生成额外的检查点。在“每条分时报价”模式下生成分时报价的基础知识在 [文档](#) 中介绍过。

在“每条分时报价”模式下测试时,将对每个生成的分时报价调用 EA 交易的 **OnTick** 函数。EA 交易会以与在线工作时相同的方式接收时间和 **Ask/Bid/Last** 价格。

“每条分时报价”测试模式是最准确的(在真实分时报价模式之后),但也是最耗时的。对于大多数交易策略的主要估算,使用这两种简化测试模式中的一种就足够了:OHLC M1 价格或所选时段的开盘价。

1 分钟 OHLC

在“1分钟 OHLC”模式下,分时报价序列仅由分钟柱线的 OHLC 价格构建,**OnTick** 函数调用次数显著减少;因此,测试时间也减少。这是一种非常有效且有用的模式,在测试精度和速度之间提供了一种折衷。但是,当涉及到别的 EA 交易时,需要特别小心。

如果拒绝在 **Open**、**High**、**Low** 和 **Close** 价格之间生成额外的中间分时报价,会导致从 **Open** 价格被定义的那一刻起,价格的发展就出现了硬性决定。这样便可以创建一个“Testing Grail”,在测试时显示一个漂亮的上升趋势平衡图。

对于一个分钟柱线,已知 4 个价格,其中第一个为 **Open** 价格,最后一个为 **Close** 价格。它们之间记录的价格有 **High** 和 **Low** 价格,关于它们出现顺序信息会丢失,但我们知道 **High** 价格大于等于 **Open** 价格, **Low** 价格小于或等于 **Open** 价格。

收到 **Open** 价格后,我们只需要分析下一条分时报价,就可以确定它是 **High** 还是 **Low** 价格。如果该价格低于 **Open** 价格,即在这个分时报价上的 **Low** 价格 - 买入价,因为下一个分时报价将对应于 **High** 价格,在此处我们应关闭买入交易并开设卖出交易。下一个分时报价是柱线的最后一个,即 **Close** 价格,在此处我们将关闭卖出交易。

如果一个价格高于开盘价的分时报价出现在我们的价格之后,那么交易顺序就颠倒了。看起来,在这种模式下,可以在任何一根柱线上进行交易。当在历史上测试这样的 EA 交易时,一切都很完美,但在线测试却会失败。

由于计算算法(例如,统计计算)和分时报价生成的函数组合,可能会无意中产生类似的影响。

因此,在粗略测试模式(“1 分钟 OHLC”“仅开盘价”)下找到最佳的 EA 交易设置后,在“每条分时报价”模式下进行测试,或者更好地基于真实分时报价进行测试,总是非常重要的。

仅开盘价

在这种模式下，分时报价是使用所选测试时间范围的 OHLC 价格生成的。在这种情况下，**OnTick** 函数仅在每个柱线的开头运行一次。由于这个特性，止损水平和挂单可能以不同于请求的价格触发（特别是在更高的时间范围上测试时）。作为交换，我们有机会快速对 EA 交易进行评估测试。

例如，EA 交易在“仅开盘价”模式下进行 EURUSD H1 测试。在这种情况下，分时报价（控制点）的总数将是测试间隔内小时柱线数量的 4 倍。但在这种情况下，**OnTick** 处理程序只会在小时柱线打开时调用。对于其余的分时报价（对 EA 交易“隐藏”的），会执行正确测试所需的以下检查：

- 保证金要求的计算
- 触发 **Stop Loss** 和止盈 **Take Profit**
- 触发挂单
- 过期时删除挂单

如果没有未结仓位或挂单，就不需要对隐藏的分时报价进行这些检查，速度会显著提高。

在“仅开盘价”模式下生成分时报价时的一个例外是 W1 和 MN1 时段：对于这些时段，分时报价是为每天的 OHLC 价格生成的，而不是分别为每周或每月生成的。

这种“仅开盘价”模式非常适合测试只在柱线打开时、不使用挂单以及不使用 **Stop Loss** 和 **Take Profit** 水平的交易策略。对于这类策略，保留了所有必要的测试精度。

MT5 API 不允许程序找出其在测试程序中运行的模式。同时，这可能对 EA 交易或其使用的指标很重要，例如，它们并未设计为正确处理开盘价或 OHLC。在这方面，我们实现了一个简单的模式检测机制。源代码附在 **TickModel.mqh** 文件中。

我们用现有模式来说明我们的枚举。

```
enum TICK_MODEL
{
    TICK_MODEL_UNKNOWN = -1,    /*Unknown (any)*/    // unknown/not yet defined
    TICK_MODEL_REAL = 0,        /*Real ticks*/      // best quality
    TICK_MODEL_GENERATED = 1,    /*Generated ticks*/  // good quality
    TICK_MODEL_OHLC_M1 = 2,      /*OHLC M1*/          // acceptable quality and fast
    TICK_MODEL_OPEN_PRICES = 3,  /*Open prices*/      // worse quality, but very fast
    TICK_MODEL_MATH_CALC = 4,    /*Math calculations*/ // no ticks (not defined)
};
```

除了第一个元素（保留用于模式尚未确定或由于某种原因无法确定的情况）之外，所有其他元素都以模拟质量的降序排列，从真实价格开始，以开盘价结束（对于这些元素，开发人员必须检查其交易仅在新柱线打开时进行这一事实的兼容性策略）。最后一种模式 **_MODEL_MATH_CALC** 运行时完全没有分时报价；我们会单独介绍。

模式检测原理基于在开始测试时，对分时报价的可用性及其前两条分时报价时间的检查。检查本身被包装在 **getTickModel** 函数中，EA 交易应从 **OnTick** 处理程序中调用这个函数。由于检查只进行一次，静态变量模型在最初设置为分时报价 **TICK_MODEL_UNKNOWN** 的函数中进行说明。该模型可存储并切换检查的当前状态，这将需要区分 OHLC 模式和开盘价。


```
TICK_MODEL getTickModel()
{
    static TICK_MODEL model = TICK_MODEL_UNKNOWN;
    ...
}
```

在第一个分析的分时报价上，该模型等于 `TICK_MODEL_UNKNOWN`，并试图通过调用 `CopyTicks` 来获得真实分时报价。

```
if(model == TICK_MODEL_UNKNOWN)
{
    MqlTick ticks[];
    const int n = CopyTicks(_Symbol, ticks, COPY_TICKS_ALL, 0, 10);
    if(n == -1)
    {
        switch(_LastError)
        {
            case ERR_NOT_ENOUGH_MEMORY: // emulate ticks
                model = TICK_MODEL_GENERATED;
                break;

            case ERR_FUNCTION_NOT_ALLOWED: // prices of opening and OHLC
                if(TimeCurrent() != iTime(_Symbol, _Period, 0))
                {
                    model = TICK_MODEL_OHLC_M1;
                }
                else if(model == TICK_MODEL_UNKNOWN)
                {
                    model = TICK_MODEL_OPEN_PRICES;
                }
                break;
        }

        Print(E2S(_LastError));
    }
    else
    {
        model = TICK_MODEL_REAL;
    }
}
...
```

如果成功，检测立即结束，将该模型设置为 `TICK_MODEL_REAL`。如果没有真实分时报价，系统会返回某种错误代码，根据这个错误代码我们可以得出以下结论。错误代码 `ERR_NOT_ENOUGH_MEMORY` 对应于分时报价模拟模式。为什么该代码是这样的还不完全清楚，但是这是一个典型的特性，我们只是此处使用该代码。在其他两种分时报价生成模式中，我们将得到 `ERR_FUNCTION_NOT_ALLOWED` 错误。

你可以通过分时报价时间来区分这两种模式。如果结果不是一条分时报价时间范围的倍数，那么我们就在讨论 `OHLC` 模式。但这里的问题是，两种模式中的第一条分时报价都可以与柱线打开时间对齐。因此，我们将获得 `TICK_MODEL_OPEN_PRICES` 值，但需要进行指定。因此，对于最终结论，应再分析一条分时报价（如果之前收到了 `TICK_MODEL_OPEN_PRICES`，则再次调用其函数）。对于这种情况，在函数内部提供了以下 `if` 分支。

```

else if(model == TICK_MODEL_OPEN_PRICES)
{
    if(TimeCurrent() != iTime(_Symbol, _Period, 0))
    {
        model = TICK_MODEL_OHLC_M1;
    }
}
return model;
}

```

我们在一个简单的 EA 交易 *TickModel.mq5* 中检查检测程序的运行。在 *TickCount* 输入参数中，我们指定了分析的分时报价的最大数量，即 *getTickModel* 将被调用的次数。我们知道两条分时报价就足够了，但是为了确保模型随后不会改变，默认情况下，建议使用 5 条分时报价。我们还提供了 *RequireTickModel* 参数，如果模拟水平低于要求的水平，该参数会指示 EA 交易终止操作。默认情况下，其值为 *TICK_MODEL_UNKNOWN*，这意味着没有模式限制。

```

input int TickCount = 5;
input TICK_MODEL RequireTickModel = TICK_MODEL_UNKNOWN;

```

在 *OnTick* 处理程序中，我们只在测试程序中运行代码时才运行它。

```

void OnTick()
{
    if(MQLInfoInteger(MQL_TESTER))
    {
        static int count = 0;
        if(count++ < TickCount)
        {
            // output tick information for reference
            static MqlTick tick[1];
            SymbolInfoTick(_Symbol, tick[0]);
            ArrayPrint(tick);
            // define and display the model (preliminarily)
            const TICK_MODEL model = getTickModel();
            PrintFormat("%d %s", count, EnumToString(model));
            // if the tick counter is 2+, the conclusion is final and we act based on it
            if(count >= 2)
            {
                if(RequireTickModel != TICK_MODEL_UNKNOWN
                && RequireTickModel < model) // quality less than requested
                {
                    PrintFormat("Tick model is incorrect (%s %sis required), terminating",
                        EnumToString(RequireTickModel),
                        (RequireTickModel != TICK_MODEL_REAL ? "or better " : ""));
                    ExpertRemove(); // end operation
                }
            }
        }
    }
}

```

我们通过选择 EURUSD H1 的常见组合，尝试在测试程序中以不同的分时报价生成模式运行 EA 交易。

EA 交易中的 *RequireTickModel* 参数被设置为 OHLC M1。如果测试程序模式是“每条分时报价”，我们将在日志中看到相应的消息，EA 交易将继续工作。

```
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:00:30 1.10656 1.10679 1.10656 0 1648771230000 14 0.00000
NOT_ENOUGH_MEMORY
1 TICK_MODEL_GENERATED
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:01:00 1.10656 1.10680 1.10656 0 1648771260000 12 0.00000
2 TICK_MODEL_GENERATED
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:01:30 1.10608 1.10632 1.10608 0 1648771290000 14 0.00000
3 TICK_MODEL_GENERATED
```

OHLC M1 和真实分时报价模式也适用，在后一种情况下，将不会有错误代码。

```
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:00:00 1.10656 1.10687 0.0000 0 1648771200122 134 0.00000
1 TICK_MODEL_REAL
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:00:00 1.10656 1.10694 0.0000 0 1648771200417 4 0.00000
2 TICK_MODEL_REAL
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:00:00 1.10656 1.10691 0.0000 0 1648771200816 4 0.00000
3 TICK_MODEL_REAL
```

但是，如果你将测试程序中的模式更改为“仅开盘价”，EA 交易将在第二条分时报价后停止。

```
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:00:00 1.10656 1.10679 1.10656 0 1648771200000 14 0.00000
FUNCTION_NOT_ALLOWED
1 TICK_MODEL_OPEN_PRICES
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 1:00:00 1.10660 1.10679 1.10660 0 1648774800000 14 0.00000
2 TICK_MODEL_OPEN_PRICES
Tick model is incorrect (TICK_MODEL_OHLC_M1 or better is required), terminating
ExpertRemove() function called
```

这种方法需要运行一次测试并等待几条分时报价来确定模式。换言之，我们不能通过从 *OnInit* 返回一个错误来提前停止测试。更重要的是，当使用错误类型的分时报价生成开始优化时，我们将无法停止优化，只能通过 *OnTesterInit* 函数来完成。因此，测试程序将尝试在优化过程中完成所有传递，尽管它们会在最开始时停止。这是当前平台的局限性。

6.5.2 测试程序中的时间管理：计时器、睡眠和 GMT

在开发 EA 交易时，应考虑到测试程序有一些基于 [生成的分时报价](#) 和时间相关函数的操作来模拟时间流逝的细节。

测试时，根据 *TimeTradeServer*, *TimeLocal* 函数返回的本地时间始终等于服务器时间。反过来，服务器时间总是等于 GMT *TimeGMT*。因此，测试时，所有这些函数给出相同的时间。这是该平台的一个技术特性，这是因为其决定不在本地存储关于服务器时间的信息，而是总是从服务器获取，在特定时刻可能与服务器没有关系的信息。

这一特性给实施与全球时间有关的策略造成了困难，特别是在新闻发布方面。在这种情况下，有必要在被测试的 EA 交易设置中指定报价的时区，或者发明自动检测时区的方法（参见 [夏令时](#) 一节）。

现在我们来看看其他与时间相关的函数。

众所周知，在 MQL5 中可以处理计时器事件。无论测试模式如何，[OnTimer](#) 处理程序都会被调用。这意味着，如果在 H4 时段以“仅开盘价”模式启动测试，并且在 EA 交易中设置了计时器，每秒钟有一次调用，那么 [OnTick](#) 处理程序将在每个 H4 柱线打开时被调用一次，然后，在柱线内，[OnTimer](#) 处理程序将被调用 14400 次（3600 秒乘以 4 小时）。在这种情况下，EA 交易测试时间增加的程度取决于其算法。

另一个影响程序时间进程的函数是 [Sleep](#) 函数。该函数允许你将 EA 交易的执行暂停一段时间。请求任何在请求时尚未准备好的数据时，这可能是必要的，并且必须等待直到该数据准备好。

理解 [Sleep](#) 只会影响其调用程序，而不会延迟测试过程，这一点很重要。事实上，当调用 [Sleep](#) 时，生成的分时报价会在指定的延迟内“播放”，其结果是挂单、止损水平等均会被触发。调用 [Sleep](#) 后，测试程序中模拟的时间会增加函数参数中指定的间隔。

稍后，在 [测试多币种 EA 交易](#) 一节，我们将展示如何使用计时器和 [Sleep](#) 函数来同步柱线。

6.5.3 测试可视化对象：图表、对象和指标

测试程序允许以两种不同的方式进行测试：可视化和非可视化。通过在测试程序的主设置选项卡上选择相应的选项来选择测试方法。

启用可视化时，测试程序会打开一个单独的窗口，在其中再现交易操作并显示指标和对象。虽然测试程序是可视化的，但我们并不需要在每种情况下都能看到，只需要在有用户接口的程序中看到即可（例如，交易面板或由图形对象制作的受控标记）。对于其他 EA 交易而言，只有根据既定策略执行算法才是重要的。这可以在没有可视化的情况下进行检查，进而可以显著加快该过程。顺便说一下，在优化过程中，测试就是在这种模式下运行的。

在这样的“后台”测试和优化过程中，不会构建任何图形对象。因此，当访问对象的特性时，EA 交易将接收到零值。因此，只有在可视化模式下测试时，你才能检查对象和图表的工作。

之前，在 [测试指标](#) 一节，我们已经看到了指标在测试程序中的具体行为。为了提高非可视化测试和 EA 交易优化（使用指标）的效率，指标可以不在每条分时报价处进行计算，而是仅在我们向他们请求数据时进行计算。仅当指标中有 [EventChartCustom](#)、[OnChartEvent](#) 和 [OnTimer](#) 函数或 [tester_everytick_calculate](#) 指令时，才会对每条分时报价进行重新计算（请参阅 [测试程序的预处理指令](#)）。在可视化测试程序窗口中，在线指标总是在每条分时报价处接收 [OnCalculate](#) 事件。

如果测试是在非可视模式下进行的，则在测试完成后，符号图表会自动在终端中打开，显示已完成的交易和 EA 交易中使用的指标。这有助于将市场进入和退出时刻与指标值联系起来。但是，此处我们指的只是对测试的交易品种和时间范围起作用的指标。如果 EA 交易在其他交易品种或时间范围上创建了指标，则不会显示。

务必注意的是，测试完成后自动打开的图表上显示的指标会在测试结束后重新计算。即使这些指标在测试的 EA 交易中使用过，并且在柱线形成时已经计算过，也会发生这种情况。

在某些情况下，程序员可能需要隐藏交易算法中使用了哪些指标的信息，因此它们在图表上的可视化是不可取的。为此，可以使用 [IndicatorRelease](#) 函数。

IndicatorRelease 函数最初用于在不再需要时释放指标的计算部分。这样可以节省内存和处理器的资源。其第二个目的是在完成一次运行后，禁止在测试图表上显示指标。

要在测试结束时禁用图表上的指标显示，只需用 **OnDeinit** 处理程序中的指标句柄调用 **IndicatorRelease** 即可。在完成之后和显示测试图表之前，总会在 EA 交易中调用 **OnDeinit** 函数。无论是 **OnDeinit** 还是全局和静态对象的析构函数都不会在测试程序的指标本身中被调用 – 这是 MetaTrader 5 的开发人员商定的。

此外，MQL5 API 包括一个具有类似目的的特殊函数 **TesterHideIndicators**，我们将在后面探讨。

同时，应注意 **tpl** 模板（如果已创建）会额外影响测试图表的外部表示。

因此，如果 **MQL5/Profiles/Templates** 目录中有 **tester.tpl** 模板，该模板将被应用到打开的图表中。如果 EA 交易在其工作中使用了其他指标，并且没有禁止显示这些指标，则来自模板和 EA 交易的指标将会在图表上合并。

当 **tester.tpl** 不存在时，应用默认模板（**default.tpl**）。

如果 **MQL5/Profiles/Templates** 文件夹包含一个与 EA 交易同名的 **tpl** 模板（例如，**ExpertMACD.tpl**），则在可视化测试期间或在测试后打开的图表上，只会显示该模板中的指标。在这种情况下，不会显示测试的 EA 交易中使用的任何指标。

6.5.4 多币种测试

我们知道，MetaTrader 5 测试程序允许你测试交易多种金融工具的策略。从纯技术上讲，受制于计算机硬件资源，模拟所有可用金融工具同时交易是可能的。

测试这种策略对测试程序提出了几个额外的技术要求：

- 生成所有金融工具的分时报价序列
- 计算所有金融工具的指标
- 计算保证金要求并模拟所有金融工具的其他交易条件

首次访问历史时，测试程序会自动从终端下载所需金融工具的历史。如果终端不包含所需的历史，则将依次向交易服务器请求。因此，在测试多币种 EA 交易之前，建议在终端的 **Market Watch** 中选择所需的金融工具，并下载所需的数据量。

代理应上传缺失的历史，并留有少量保证金，以便在测试时为计算指标或由 EA 交易复制提供必要的历史数据。从交易服务器下载的历史的最小数量取决于时间范围。例如，对于 D1 和更短的时间，为 1 年。换言之，初始历史是从相对于测试程序开始日期的前一年开始下载的。如果要求从 1 月 1 日开始测试，将提供至少 1 年的历史，如果要求从 12 月开始测试，则最多可提供近 2 年的历史。对于周时间范围，需要 100 根柱线的历史，即大约 2 年（一年有 52 周）。对于每月一次的测试，代理可请求 100 个月的历史（等于约 8 年：12 个月 * 8 年 = 96）。在任何情况下，在低于工作时间的范围内，可用的柱线数量将按比例增加。如果现有数据不足以满足预定义深度的初步历史，这一事实将记录在测试日志中。

无法配置（更改）此行为。因此，如果你需要从一开始就提供当前时间范围的特定数量的历史柱线，你应为测试设置一个更早的开始日期，然后在 EA 交易代码中“等待”所需的交易开始日期或足够数量的柱线。在此之前，你应跳过所有事件。

测试程序还可以模拟自己的 **Market Watch**，该程序可以从中获得金融工具信息。默认情况下，在测试开始时，测试程序的 **Market Watch** 只包含一个交易品种：测试开始时的交易品种。通过 API 函数访问所有附加交易品种时，这些交易品种会自动添加到测试程序的 **Market Watch** 中。

在第一次访问来自 MQL 程序的“第三方”交易品种时，测试代理应使交易品种数据与终端同步。

在下列情况下，可以访问附加交易品种的数据：

- 使用交易品种/时间范围对的技术指标 *iCustom* 或 *IndicatorCreate*
- 查询另一个交易品种的 *Market Watch*:
 - *SeriesInfoInteger*
 - *Bars*
 - *SymbolSelect*
 - *SymbolsSynchronized*
 - *SymbolInfoDouble*
 - *SymbolInfoInteger*
 - *SymbolInfoString*
 - *SymbolInfoTick*
 - *SymbolInfoSessionQuote*
 - *SymbolInfoSessionTrade*
 - *MarketBookAdd*
 - *MarketBookGet*
- 使用以下函数查询交易品种/时间范围对时间序列：
 - *CopyBuffer*
 - *CopyRates*
 - *CopyTime*
 - *CopyOpen*
 - *CopyHigh*
 - *CopyLow*
 - *CopyClose*
 - *CopyTickVolume*
 - *CopyRealVolume*
 - *CopySpread*

此外，你可以通过调用 *OnInit* 处理程序中的 *SymbolSelect* 函数来显式请求所需交易品种的历史。历史数据会在 EA 交易测试开始之前提前加载。

第一次访问另一个交易品种时，测试过程会停止，交易品种/周期对历史可从终端下载给测试代理。此时也会启用分时报价序列的生成。

每个金融工具根据设定的分时报价生成模式生成自己的分时报价序列。

实现多币种 EA 交易时，不同交易品种的柱线同步特别重要，因为这会影响计算的正确性。当所有使用交易品种的最后一个柱线具有相同的打开时间时，状态被认为是同步的。

测试程序可为每种金融工具生成并播放其分时报价序列。同时，不管其他金融工具上的柱线是如何打开的，每个金融工具上都会打开一个新的柱线。这意味着在测试多币种 EA 交易时，可

能会出现一种情况 (这种情况经常发生), 即一个新的柱线已经在一个金融工具上打开, 但在另一个金融工具上还没有打开。

例如, 如果我们正在测试一个使用 EURUSD 交易品种数据的 EA 交易, 并且为该交易品种打开了一个新的每小时 K 线, 我们将会收到 **OnTick** 事件。但与此同时, 不能保证 GBPUSD 会有一个我们可能也会使用的新 K 线。

因此, 同步算法意味着你需要检查所有金融工具的报价, 并等待最后几根柱线的开盘时间相等。

只要使用了真实分时报价、所有分时报价的模拟或 OHLC M1 测试模式, 就不会引起任何问题。使用这些模式, 可在一个柱线上生成足够数量的分时报价, 以等待来自不同交易品种的 K 线同步时刻。只需完成 **OnTick** 函数, 并在下一条分时报价处检查 GBPUSD 上一个新柱线是否出现。但是当在“仅开盘价”模式下进行测试时, 将不会有其他分时报价, 因为每个柱线只调用一次 EA 交易, 并且这种模式似乎不适合测试多币种 EA 交易。事实上, 测试程序允许你使用 **Sleep** 函数 (循环) 或计时器来检测一个新柱线打开另一个交易品种的时刻。

首先, 我们考虑一个 EA 交易示例 **SyncBarsBySleep.mq5**, 这个示例演示了通过 **Sleep** 来同步柱线。

一对输入参数允许你以秒为单位设置等待其他交易品种柱线以及其他交易品种 (**OtherSymbol**) 名称的 **Pause** 时长, 该交易品种必须不同于图表交易品种。

```
input uint Pause = 1; // Pause (seconds)
input string OtherSymbol = "USDJPY";
```

为了确定柱线打开时间的延迟模式, 我们介绍一个简单的类 **BarTimeStatistics**。该类包含一个字段, 用于计算柱线总数 (**total**) 和最初没有同步的柱线数 (**late**), 即另一个交易品种的延迟。

```
class BarTimeStatistics
{
public:
    int total;
    int late;

    BarTimeStatistics(): total(0), late(0) { }

    ~BarTimeStatistics()
    {
        PrintFormat("%d bars on %s was late among %d total bars on %s (%2.1f%%)",
            late, OtherSymbol, total, _Symbol, late * 100.0 / total);
    }
};
```

这个类的对象可在其析构函数中打印接收到的统计数据。因为我们将使这个对象成为静态的, 所以可在测试结束时打印出报告。

如果在测试程序中选择的分时报价生成模式与开盘价不同, 我们可使用之前介绍的 **getTickModel** 函数检测到这一点, 并将返回一个警告。

```
void OnTick()
{
    const TICK_MODEL model = getTickModel();
    if(model != TICK_MODEL_OPEN_PRICES)
    {
        static bool shownOnce = false;
        if(!shownOnce)
        {
            Print("This Expert Advisor is intended to run in \"Open Prices\" mode");
            shownOnce = true;
        }
    }
}
```

接下来，*OnTick* 提供了有效的同步算法。


```

// time of the last known bar for _Symbol
static datetime lastBarTime = 0;
// attribute of synchronization
static bool synchronized = false;
// bar counters
static BarTimeStatistics stats;

const datetime currentTime = iTime(_Symbol, _Period, 0);

// if it is executed for the first time or the bar has changed, save the bar
if(lastBarTime != currentTime)
{
    stats.total++;
    lastBarTime = currentTime;
    PrintFormat("Last bar on %s is %s", _Symbol, TimeToString(lastBarTime));
    synchronized = false;
}

// time of the last known bar for another symbol
datetime otherTime;
bool late = false;

// wait until the times of two bars become the same
while(currentTime != (otherTime = iTime(OtherSymbol, _Period, 0)))
{
    late = true;
    PrintFormat("Wait %d seconds...", Pause);
    Sleep(Pause * 1000);
}
if(late) stats.late++;

// here we are after synchronization, save the new status
if(!synchronized)
{
    // use TimeTradeServer() because TimeCurrent() does not change in the absence c
    Print("Bars are in sync at ", TimeToString(TimeTradeServer(),
        TIME_DATE | TIME_SECONDS));
    // no longer print a message until the next out of sync
    synchronized = true;
}
// here is your synchronous algorithm
// ...
}

```

让我们设置测试程序来对 EURUSD, H1 运行 EA 交易, 其为流动性最强的金融工具。让我们使用默认的 EA 交易参数, 即 USDJPY 为“其他”交易品种。

作为测试的结果, 日志将包含以下条目 (我们有意显示与 USDJPY 历史下载相关的日志, 这发生在第一次 *iTime* 调用期间)。

```

2022.04.15 00:00:00 Last bar on EURUSD is 2022.04.15 00:00
USDJPY: load 27 bytes of history data to synchronize in 0:00:00.001
USDJPY: history synchronized from 2020.01.02 to 2022.04.20
USDJPY,H1: history cache allocated for 8109 bars and contains 8006 bars from 2021.01.
USDJPY,H1: 1 bar from 2022.04.15 00:00 added
USDJPY,H1: history begins from 2021.01.04 00:00
2022.04.15 00:00:00 Bars are in sync at 2022.04.15 00:00:00
2022.04.15 1:00:00 Last bar on EURUSD is 2022.04.15 1:00
2022.04.15 01:00:00 Wait 1 seconds...
2022.04.15 1:00:01 Bars are in sync at 2022.04.15 1:00:01
2022.04.15 2:00:00 Last bar on EURUSD is 2022.04.15 2:00
2022.04.15 2:00:00 Wait 1 seconds...
2022.04.15 2:00:01 Bars are in sync at 2022.04.15 2:00:01
...
2022.04.20 23:59:59 95 bars on USDJPY was late among 96 total bars on EURUSD (99.0%)

```

你可以看到 USDJPY 柱线有规律的延迟情况。如果你在测试程序设置中选择 USDJPY, H1, 在 EA 交易参数中选择 EURUSD, 将得到相反的数据。

```

2022.04.15 00:00:00 Last bar on USDJPY is 2022.04.15 00:00
EURUSD: load 27 bytes of history data to synchronize in 0:00:00.002
EURUSD: history synchronized from 2018.01.02 to 2022.04.20
EURUSD,H1: history cache allocated for 8109 bars and contains 8006 bars from 2021.01.
EURUSD,H1: 1 bar from 2022.04.15 00:00 added
EURUSD,H1: history begins from 2021.01.04 00:00
2022.04.15 00:00:00 Bars are in sync at 2022.04.15 00:00:00
2022.04.15 1:00:00 Last bar on USDJPY is 2022.04.15 1:00
2022.04.15 01:00:00 Wait 1 seconds...
2022.04.15 1:00:01 Bars are in sync at 2022.04.15 1:00:01
2022.04.15 2:00:00 Last bar on USDJPY is 2022.04.15 2:00
2022.04.15 2:00:00 Wait 1 seconds...
2022.04.15 2:00:01 Bars are in sync at 2022.04.15 2:00:01
...
2022.04.20 23:59:59 23 bars on EURUSD was late among 96 total bars on USDJPY (24.0%)

```

此处, 在大多数情况下没有必要等待: EURUSD 的柱线在 USDJPY 的柱线形成时就已经存在了。

还有另一种方法来同步柱线: 使用计时器。这种 EA 交易的一个示例 *SyncBarsByTimer.mq5* 也包含在本书中。请注意, 通常情况下, 计时器事件发生在柱线内 (因为恰好到达起点的概率非常低)。因此, 柱线几乎总是同步的。

我们还可以使用间谍指标 *EventTickSpy.mq5* 提醒你同步柱线的可能性, 但是该指标是基于自定义事件的, 只有在可视化测试时才起作用。此外, 对于需要对每条分时报价做出响应的指标, 使用 *#property tester_everytick_calculate* 指令非常重要。我们已经在 [测试指标](#) 一节讨论过了, 我们将在具体的 [测试程序指令](#) 章节中再次回顾。

6.5.5 优化准则

优化准则是定义输入参数测试集质量的某个度量指标。优化准则的值越大, 对使用给定参数集的测试结果估算就越好。在“优化”字段右侧的“设置”选项卡上选择该参数。

该准则不只是对用户能否比较结果很重要。没有优化准则, 就不可能使用遗传算法, 因为基于该准则, 优化准则可“决定”如何为新一代选择候选算法。在对所有可能变量进行完全迭代的完全优化过程中, 不使用该准则。

测试程序中提供了以下内置优化准则：

- ①最大余额
- ①最大利润
- ①最大预期盈利（每笔交易的平均盈利/亏损）
- ①最小提款额占权益的百分比
- ①最大恢复系数
- ①最大夏普比率
- ①定制优化准则

当选择后一个选项时，在 EA 交易中实现的 **OnTester** 函数的值将作为一个优化准则予以考虑 - 我们将在 后面介绍。此参数允许程序员使用任何自定义索引进行优化。

MetaTrader 5 中还提供了一个特殊的“复杂准则”。这是测试轮次质量的一个整体指标，它同时考虑了几个参数：

- ①交易次数
- ①提款
- ①恢复系数
- ①盈利的数学预期
- ①夏普比率

开发者没有公开该公式，但是已知可能的取值范围为 0 到 100。重要的是，该复杂参数的值会影响优化表中 **Result** 列的单元格颜色，而与准则无关，即，即使在 **Result** 列中选择了另一个准则进行显示时，遵循该方案的高亮显示仍然有效。低于 20 的弱组合以红色突出显示，高于 80 的强组合以深绿色突出显示。

对于大多数交易者而言，寻找交易系统质量因子的通用准则是一项紧迫而困难的任务，因为根据一个准则（例如，利润）的最大值来选择设置，一般来说，在可预见的将来，就 EA 交易的稳定和可预测行为而言肯定不是最佳选择。

复杂指标的存在使你能够消除每个单独指标的弱点（这些指标必须可用且广为人知），并在开发你自己的定制变量以用于 **OnTester** 计算时提供指南。我们将尽快处理这个问题。

6.5.6 获取测试财务统计数据：TesterStatistics

我们通常根据交易报告来评估 EA 交易的质量，交易报告类似于用测试程序进行交易的测试报告。其包含了大量表征交易风格、稳定性以及盈利能力的变量。除了少数例外情况，所有这些指标均可通过特殊函数 **TesterStatistics** 用于 MQL 程序。因此，EA 交易开发人员能够分析代码中的单个变量，并根据这些变量构建他们自己的组合优化质量标准。

double TesterStatistics(ENUM_STATISTICS statistic)

TesterStatistics 函数可返回指定统计变量的值，该值是根据测试程序中 EA 交易的单独运行结果计算得出。可以在 **OnDeinit** 或 **OnTester** 处理程序中调用函数，这将在后面讨论。

ENUM_STATISTICS 枚举中总结了所有可用的统计变量。其中一些变量具有定性特征，即实数型（通常包括利润总额、提款、比率等），另一些变量具有定量特征，即整数型（例如交易次数）。但是，两组变量都由具有 **double** 结果的相同函数控制。

下表显示了实数型指标（货币金额和系数）。所有货币金额都以存款货币表示。

标识符	说明
STAT_INITIAL_DEPOSIT	初始存款
STAT_WITHDRAWAL	从账户中提取的金额
STAT_PROFIT	测试结束时的净利润或亏损, 即 STAT_GROSS_PROFIT 和 STAT_GROSS_LOSS 的总和
STAT_GROSS_PROFIT	总利润, 即所有盈利交易的总和 (大于等于零)
STAT_GROSS_LOSS	总亏损, 即所有亏损交易的总和 (小于或等于零)
STAT_MAX_PROFITTRADE	最大利润: 所有盈利交易中的最大值 (大于等于零)
STAT_MAX_LOSTRADE	最大亏损: 所有亏损交易中的最小值 (小于或等于零)
STAT_CONPROFITMAX	一系列止盈交易的最大总利润 (大于等于零)
STAT_MAX_CONWINS	最长止盈交易系列的总利润
STAT_CONLOSSMAX	一系列亏损交易的最大总亏损 (小于或等于零)
STAT_MAX_CONLOSSES	最长亏损交易系列的总亏损
STAT_BALANCEMIN	最小余额值
STAT_BALANCE_DD	最大余额提款
STAT_BALANCEDD_PERCENT	在最大余额提款 (STAT_BALANCE_DD) 时记录的以百分比表示的余额提款
STAT_BALANCE_DDREL_PERCENT	最大余额提取百分比
STAT_BALANCE_DD_RELATIVE	在最大余额提款百分比 (STAT_BALANCE_DDREL_PERCENT) 时记录的以等值货币表示的余额提款
STAT_EQUITYMIN	最小净值
STAT_EQUITY_DD	最大提款金额
STAT_EQUITYDD_PERCENT	在货币中资金的最大提款时 (STAT_EQUITY_DD) 记录的以百分比表示的提款
STAT_EQUITY_DDREL_PERCENT	最大提款百分比
STAT_EQUITY_DD_RELATIVE	最大提款时记录的以百分比表示的提款 (STAT_EQUITY_DDREL_PERCENT)
STAT_EXPECTED_PAYOFF	盈利的数学期望 (总利润和交易次数的算术平均值)
STAT_PROFIT_FACTOR	盈利能力, 即 $\text{STAT\_GROSS\_PROFIT} / \text{STAT\_GROSS\_LOSS}$ 的比率 (如果 $\text{STAT\_GROSS\_LOSS} = 0$; 盈利能力取值 DBL_MAX)
STAT_RECOVERY_FACTOR	恢复系数: $\text{STAT\_PROFIT} / \text{STAT\_BALANCE\_DD}$ 的比率

标识符	说明
STAT_SHARPE_RATIO	夏普比率
STAT_MIN_MARGINLEVEL	达到的最小保证金水平
STAT_CUSTOM_ONTESTER	OnTester 函数返回的自定义优化准则的值

下表显示了整数型指标（金额）。

标识符	说明
STAT_DEALS	已完成的交易次数
STAT_TRADES	交易次数（退出市场的交易）
STAT_PROFIT_TRADES	盈利交易
STAT_LOSS_TRADES	亏损交易
STAT_SHORT_TRADES	空头交易
STAT_LONG_TRADES	多头交易
STAT_PROFIT_SHORTTRADES	空头盈利交易
STAT_PROFIT_LONGTRADES	多头盈利交易
STAT_PROFITTRADES_AVGCON	一系列盈利交易的平均长度
STAT_LOSSTRADES_AVGCON	一系列亏损交易的平均长度
STAT_CONPROFITMAX_TRADES	构成 STAT_CONPROFITMAX 的交易次数（盈利交易序列中的最大利润）
STAT_MAX_CONPROFIT_TRADES	最长盈利交易系列的交易次数
STAT_CONLOSSMAX_TRADES	构成 STAT_CONLOSSMAX 的交易次数（亏损交易序列中的最大亏损）
STAT_MAX_CONLOSS_TRADES	最长亏损交易系列的交易次数 STAT_MAX_CONLOSSES

我们尝试使用给出的指标来创建我们自己的复杂 EA 交易质量标准。要做到这一点，我们需要某种 MQL 程序的“实验”示例。我们以 EA 交易 [MultiMartingale.mq5](#) 为起点，但是我们将对其进行简化：删除多币种、内置错误处理和时间安排。此外，我们将选择一个信号交易策略以及一个单一的计算柱线，即开盘价。这将加快优化速度，并扩大实验领域。

该策略将基于 0sMA 指标确定的超买和超卖条件。叠加在 0sMA 上的 Bollinger 带指标可帮你动态找到波动幅度过大的边界，即交易信号。

当 0sMA 从下向上穿越通道下边界并回到通道内时，我们将开启买入交易。当 0sMA 从上到下以同样的方式越过上边界时，我们则开启卖出交易。为了退出仓位，我们使用了移动平均线，也适用于 0sMA。如果 0sMA 显示反向移动（多头向下或空头向上）并触及 MA，该仓位将被平仓。下面的截图展示了这种策略。



基于 OsMA、BBands 和 MA 指标的交易策略

蓝色竖线对应的是开仓买入时的柱线，因为在前两根柱线上，下方的 Bollinger 带被 OsMA 柱状图从下向上穿过（这个仓位在子窗口内用蓝色空心箭头标出）。红色竖线是反转符号的位置，所以买入交易关闭，卖出交易打开。在子窗口中，在这个位置（或者更确切地说，在前面两根柱线上，空心红色箭头所在的位置），OsMA 柱状图自上而下穿过上 Bollinger 带。最后，绿线表示成交，因为柱状图开始高于红色均线。

我们将 EA 交易命名为 **BandOsMA.mq5**。一般设置将包括一个魔术编号、一个固定手数和一个以点为单位的止损距离。对于止损，我们将使用前面示例中的 **TrailingStop**。此处不用止盈。

```
input group "COMMON SETTINGS"
input ulong Magic = 1234567890;
input double Lots = 0.01;
input int StopLoss = 1000;
```

三组设置可用于这些指标。

```

input group "O S M A   S E T T I N G S"
input int FastOsMA = 12;
input int SlowOsMA = 26;
input int SignalOsMA = 9;
input ENUM_APPLIED_PRICE PriceOsMA = PRICE_TYPICAL;

input group "B B A N D S   S E T T I N G S"
input int BandsMA = 26;
input int BandsShift = 0;
input double BandsDeviation = 2.0;

input group "M A   S E T T I N G S"
input int PeriodMA = 10;
input int ShiftMA = 0;
input ENUM_MA_METHOD MethodMA = MODE_SMA;

```

在 *MultiMartingale.mq5* EA 交易中，我们没有交易信号，而开仓方向是由用户设定的。此处我们有交易信号，合理做法是将其单独列为一类。首先，我们说明一下抽象接口 *TradingSignal*。

```

interface TradingSignal
{
    virtual int signal(void);
};

```

该接口和我们的其他接口 *TradingStrategy* 一样简单。这样很好。接口和对象越简单，就越有可能执行同样的操作，这是一种很好的编程风格，因为可最大限度地减少错误，并使大型软件项目更容易理解。由于在任何使用 *TradingSignal* 的程序中都有抽象概念，便可以用一个信号替换另一个信号。我们也可以替换策略。我们的策略现在负责准备和发送订单，并根据市场分析发出启动订单符号。

就我们的情况下而言，我们将 *TradingSignal* 的具体实现打包到 *BandOsMaSignal* 类中。当然，我们需要变量来存储 3 个指标的说明符。指标实例分别在构造函数和析构函数中创建和删除。所有参数都将通过输入变量传递。注意，*iBands* 和 *iMA* 是基于 *hOsMA* 处理程序构建的。

```

class BandOsMaSignal: public TradingSignal
{
    int hOsMA, hBands, hMA;
    int direction;
public:
    BandOsMaSignal(const int fast, const int slow, const int signal,
        const ENUM_APPLIED_PRICE price,
        const int bands, const int shift, const double deviation,
        const int period, const int x, ENUM_MA_METHOD method)
    {
        hOsMA = iOsMA(_Symbol, _Period, fast, slow, signal, price);
        hBands = iBands(_Symbol, _Period, bands, shift, deviation, hOsMA);
        hMA = iMA(_Symbol, _Period, period, x, method, hOsMA);
        direction = 0;
    }

    ~BandOsMaSignal()
    {
        IndicatorRelease(hMA);
        IndicatorRelease(hBands);
        IndicatorRelease(hOsMA);
    }
    ...
}

```

当前交易信号的方向放在 *direction* 变量内: 0 - 无信号 (未定义情况), +1 - 买入, -1 - 卖出。我们将在 *signal* 方法中填充这个变量。其代码重复了上面 MQL5 中信号的口头说明。


```

virtual int signal(void) override
{
    double osma[2], upper[2], lower[2], ma[2];
    // get two values of each indicator on bars 1 and 2
    if(CopyBuffer(hOsMA, 0, 1, 2, osma) != 2) return 0;
    if(CopyBuffer(hBands, UPPER_BAND, 1, 2, upper) != 2) return 0;
    if(CopyBuffer(hBands, LOWER_BAND, 1, 2, lower) != 2) return 0;
    if(CopyBuffer(hMA, 0, 1, 2, ma) != 2) return 0;

    // if there was a signal already, check if it has ended
    if(direction != 0)
    {
        if(direction > 0)
        {
            if(osma[0] >= ma[0] && osma[1] < ma[1])
            {
                direction = 0;
            }
        }
        else
        {
            if(osma[0] <= ma[0] && osma[1] > ma[1])
            {
                direction = 0;
            }
        }
    }

    // in any case, check if there is a new signal
    if(osma[0] <= lower[0] && osma[1] > lower[1])
    {
        direction = +1;
    }
    else if(osma[0] >= upper[0] && osma[1] < upper[1])
    {
        direction = -1;
    }

    return direction;
}
};

```

可以看到，指标值是为第 1 和第 2 根柱线读取的，因为我们将打开一根柱线，而第 0 根柱线在我们调用 *signal* 方法时刚刚打开。

实现 *TradingStrategy* 接口的新类将被命名为 *SimpleStrategy*。

该类提供了一些新特性，同时还使用了一些以前存在的部分。特别是，其为 *PositionState* 和 *TrailingStop* 保留了自动指针，并为 *TradingSignal* 信号提供了一个新的自动指针。此外，因为我们只在柱线打开时交易，所以我们需要 *lastBar* 变量，其用于存储最后一个处理柱线的时间。

```

class SimpleStrategy: public TradingStrategy
{
protected:
    AutoPtr<PositionState> position;
    AutoPtr<TrailingStop> trailing;
    AutoPtr<TradingSignal> command;

    const int stopLoss;
    const ulong magic;
    const double lots;

    datetime lastBar;
    ...

```

全局参数被传递给 *SimpleStrategy* 构造函数。我们还传递一个指向 *TradingSignal* 对象的指针：在这种情况下，为 *BandOsMaSignal*，该指针必须通过调用代码来创建。接下来，构造函数试图在现有仓位中找到具有所需 *magic* 号和符号的仓位，如果成功，则添加一个跟踪止损。如果 EA 交易因为这样或那样的原因而中止运行，并且仓位已经被打开，这将是有益的。

```

public:
    SimpleStrategy(TradingSignal *signal, const ulong m, const int sl, const double v)
        command(signal), magic(m), stopLoss(sl), lots(v), lastBar(0)
    {
        // select "our" position among the existing ones (if there is a suitable one)
        PositionFilter positions;
        ulong tickets[];
        positions.let(POSITION_MAGIC, magic).let(POSITION_SYMBOL, _Symbol).select(tickets);
        const int n = ArraySize(tickets);
        if(n > 1)
        {
            Alert(StringFormat("Too many positions: %d", n));
            // TODO: close extra positions - this is not allowed by the strategy
        }
        else if(n > 0)
        {
            position = new PositionState(tickets[0]);
            if(stopLoss)
            {
                trailing = new TrailingStop(tickets[0], stopLoss, stopLoss / 50);
            }
        }
    }
}

```

trade 方法的实现类似于鞅的示例。但是，我们已经删除了许多乘法，并增加了 *signal* 方法调用。

```

virtual bool trade() override
{
    // we work only once when a new bar appears
    if(lastBar == iTime(_Symbol, _Period, 0)) return false;

    int s = command[].signal(); // getting a signal

    ulong ticket = 0;

    if(position[] != NULL)
    {
        if(position[].refresh()) // position exists
        {
            // the signal has changed to the opposite or disappeared
            if((position[].get(POSITION_TYPE) == POSITION_TYPE_BUY && s != +1)
            || (position[].get(POSITION_TYPE) == POSITION_TYPE_SELL && s != -1))
            {
                PrintFormat("Signal lost: %d for position %d %lld",
                    s, position[].get(POSITION_TYPE), position[].get(POSITION_TICKET));
                if(close(position[].get(POSITION_TICKET)))
                {
                    position = NULL;
                }
                else
                {
                    // update internal flag 'ready'
                    // according to whether or not there was a closure
                    position[].refresh();
                }
            }
            else
            {
                position[].update();
                if(trailing[]) trailing[].trail();
            }
        }
        else // position is closed
        {
            position = NULL;
        }
    }

    if(position[] == NULL && s != 0)
    {
        ticket = (s == +1) ? openBuy() : openSell();
    }

    if(ticket > 0) // new position just opened
    {
        position = new PositionState(ticket);
        if(stopLoss)

```

```

        {
            trailing = new TrailingStop(ticket, stopLoss, stopLoss / 50);
        }
    }
    // store the current bar
    lastBar = iTime(_Symbol, _Period, 0);

    return true;
}

```

辅助方法 *openBuy*、*openSell* 等都做了极小的改动，就不一一列举了（完整源代码附后）。

因为在这个 EA 交易中我们总是只有一个策略，与多币种鞅相反，在多币种鞅中每个符号都需要自己的设置，所以我们可以排除策略池，直接管理策略对象。

```

AutoPtr<TradingStrategy> strategy;

int OnInit()
{
    if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;
    strategy = new SimpleStrategy(
        new BandOsMaSignal(FastOsMA, SlowOsMA, SignalOsMA, PriceOsMA,
            BandsMA, BandsShift, BandsDeviation,
            PeriodMA, ShiftMA, MethodMA),
        Magic, StopLoss, Lots);
    return INIT_SUCCEEDED;
}

void OnTick()
{
    if(strategy[] != NULL)
    {
        strategy[].trade();
    }
}

```

我们现在有了一个现成的 EA 交易，可以用来作为研究测试程序的工具。首先，我们创建一个辅助结构体 *TesterRecord*，用于查询和存储所有统计数据。

```

struct TesterRecord
{
    string feature;
    double value;

    static void fill(TesterRecord &stats[])
    {
        ResetLastError();
        for(int i = 0; ; ++i)
        {
            const double v = TesterStatistics((ENUM_STATISTICS)i);
            if(_LastError) return;
            TesterRecord t = {EnumToString((ENUM_STATISTICS)i), v};
            PUSH(stats, t);
        }
    }
};

```

在这种情况下，只有信息日志输出才需要 *feature* 字符串型字段。要保存所有指标（例如，为了能够在以后生成你自己的报告），只需要一个适当长度的 *double* 类型的简单数组。

使用 *OnDeinit* 处理程序中的结构体，我们可确保 MQL5 API 返回与测试程序报告相同的值。

```

void OnDeinit(const int)
{
    TesterRecord stats[];
    TesterRecord::fill(stats);
    ArrayPrint(stats, 2);
}

```

例如，在存款为 10000 且未进行任何优化（使用默认设置）的情况下，在 EURUSD,H1 上运行 EA 交易时，对于 2021 年，我们将获得大约以下值（片段）：

	[feature]	[value]
[0]	"STAT_INITIAL_DEPOSIT"	10000.00
[1]	"STAT_WITHDRAWAL"	0.00
[2]	"STAT_PROFIT"	6.01
[3]	"STAT_GROSS_PROFIT"	303.63
[4]	"STAT_GROSS_LOSS"	-297.62
[5]	"STAT_MAX_PROFITTRADE"	15.15
[6]	"STAT_MAX_LOSSTRADE"	-10.00
...		
[27]	"STAT_DEALS"	476.00
[28]	"STAT_TRADES"	238.00
...		
[37]	"STAT_CONLOSSMAX_TRADES"	8.00
[38]	"STAT_MAX_CONLOSS_TRADES"	8.00
[39]	"STAT_PROFITTRADES_AVGCON"	2.00
[40]	"STAT_LOSSTRADES_AVGCON"	2.00

知道了所有这些值，我们可以编写我们自己的公式，用于 EA 交易质量的组合度量指标，同时用于目标优化函数。但是在任何情况下，这个指标的值都需要向测试程序报告。这就是 *OnTester* 函数的作用。

6.5.7 OnTester 事件

在完成对历史数据的 EA 交易测试 (由用户启动的单独测试运行和优化期间由测试程序自动启动的多次运行之一)后,将生成 **OnTester** 事件。要处理 **OnTester** 事件,MQL 程序必须在其源代码中有相应的函数,但这不是必需的。即使没有 **OnTester** 函数,也可以依据标准准则成功优化 EA 交易。

该函数只能在 EA 交易中使用。

double OnTester()

该函数旨在计算 **double** 类型的某个值,用作自定义优化准则 (**Custom max**)。准则选择对于成功的遗传优化是重要的,同时它也允许用户估算和比较不同设置的效果。

在遗传优化中,结果在一代内按准则降序排列。也就是说,根据优化准则,具有最高值的结果认为是最好的。这种排序中最差的值随后被弃用,并且不参与下一代的形成。

请注意,只有在测试程序设置中选择了自定义准则时,才会考虑 **OnTester** 函数返回的值。**OnTester** 函数的可用性并不自动意味着其可被遗传算法使用。

无法通过 MQL5 API 来以编程方式判断用户在测试程序设置中选择了哪个优化准则。有时候,了解这一点非常重要,以便将自己的分析算法应用到后处理优化结果中。

在调用 **OnDeinit** 函数之前,内核仅在测试程序中调用此函数。

为了计算返回值,我们可以使用通过 **TesterStatistics** 函数及其任意计算获得的标准统计数据。

在 **BandOsMA.mq5** EA 交易中,我们创建了 **OnTester** 处理程序,其考虑了几个指标:利润、止盈能力、交易次数和夏普比率。接下来,我们在计算每个指标的平方根后,将所有指标相乘。当然,在构建此类通用质量准则方面,每个开发人员可能有他们自己的偏好和想法。

```
double sign(const double x)
{
    return x > 0 ? +1 : (x < 0 ? -1 : 0);
}

double OnTester()
{
    const double profit = TesterStatistics(STAT_PROFIT);
    return sign(profit) * sqrt(fabs(profit))
        * sqrt(TesterStatistics(STAT_PROFIT_FACTOR))
        * sqrt(TesterStatistics(STAT_TRADES))
        * sqrt(fabs(TesterStatistics(STAT_SHARPE_RATIO)));
}
```

单元测试日志会显示包含 **OnTester** 函数值的行。

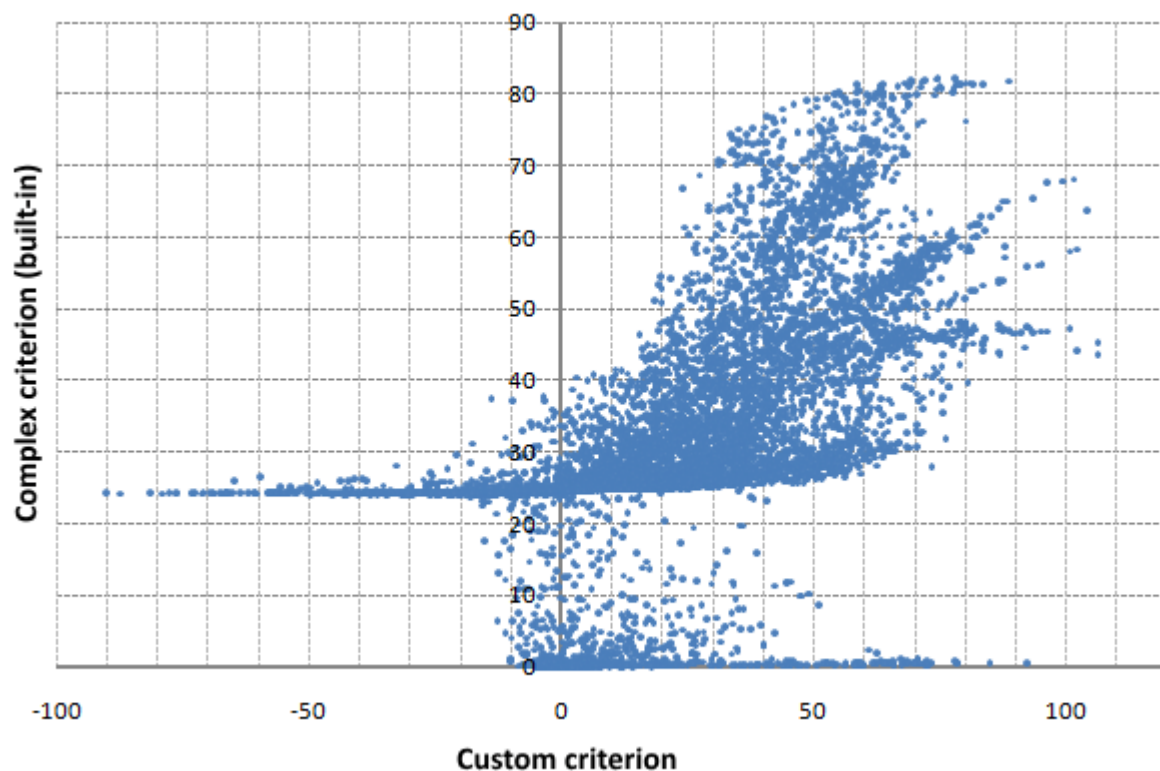
我们启动 2021 年 EURUSD,H1 的 EA 交易遗传优化,选择指标参数和止损大小 (本书附带的 **MQL5/Presets/MQL5Book/BandOsMA.set** 文件)。为了检查优化的质量,我们还将包括从 2022 年初 (6 个月)开始的递进测试。

首先,我们根据我们的准则进行优化。

如你所知,除了优化过程中使用的当前准则,MetaTrader 5 还在优化结果中保存了所有标准准则。这样,在优化完成后,可通过从带有表格的面板右上角的下拉菜单中选择某些准则来分析

不同点的结果。因此，尽管我们根据自己的标准进行了优化，但我们也可以使用最有趣的内置复杂准则。

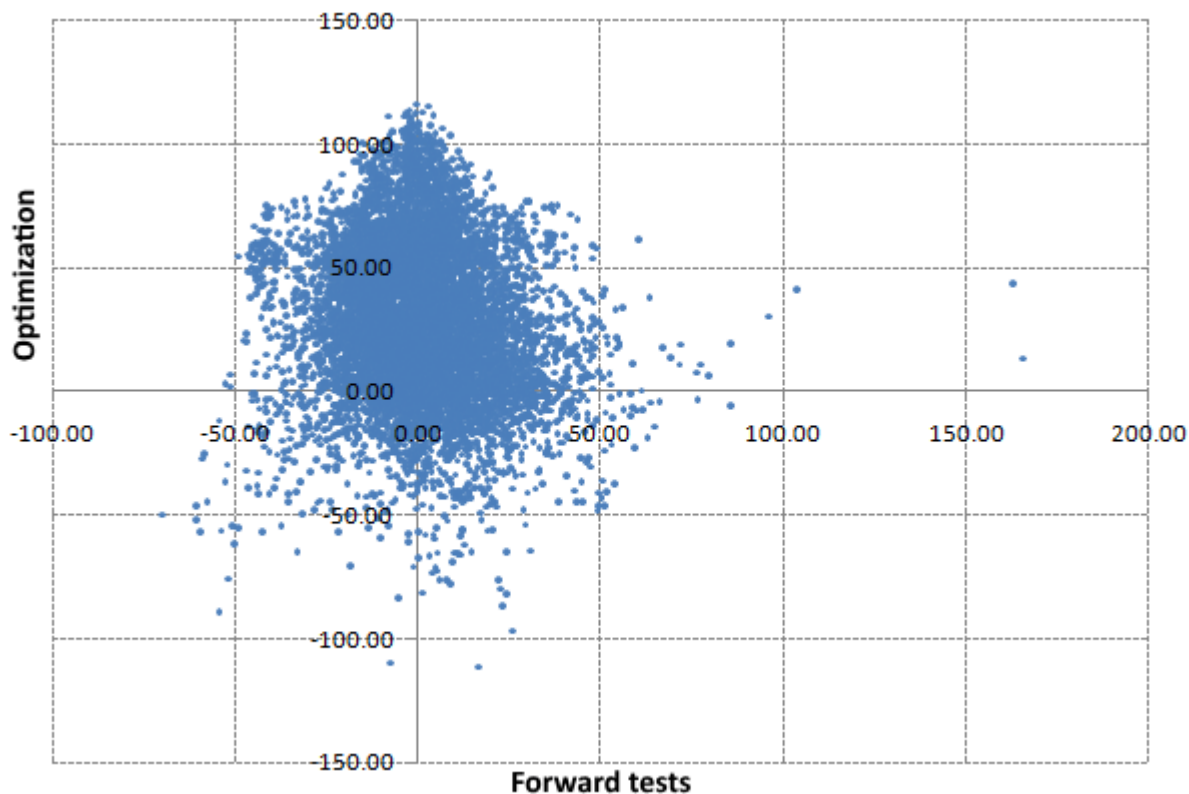
我们可以将优化表导出到一个 XML 文件中，首先使用我们选择的准则，然后使用复杂准则为文件命名（遗憾的是，只有一个准则被写入导出文件；重要的是不要改变两次导出之间的排序）。这使得在外部程序中可组合两个表并构建一个图表成为可能，在该图表上沿轴绘制两个准则；每个点表示一次运行中的准则组合。



自定义准则和复杂优化准则的比较

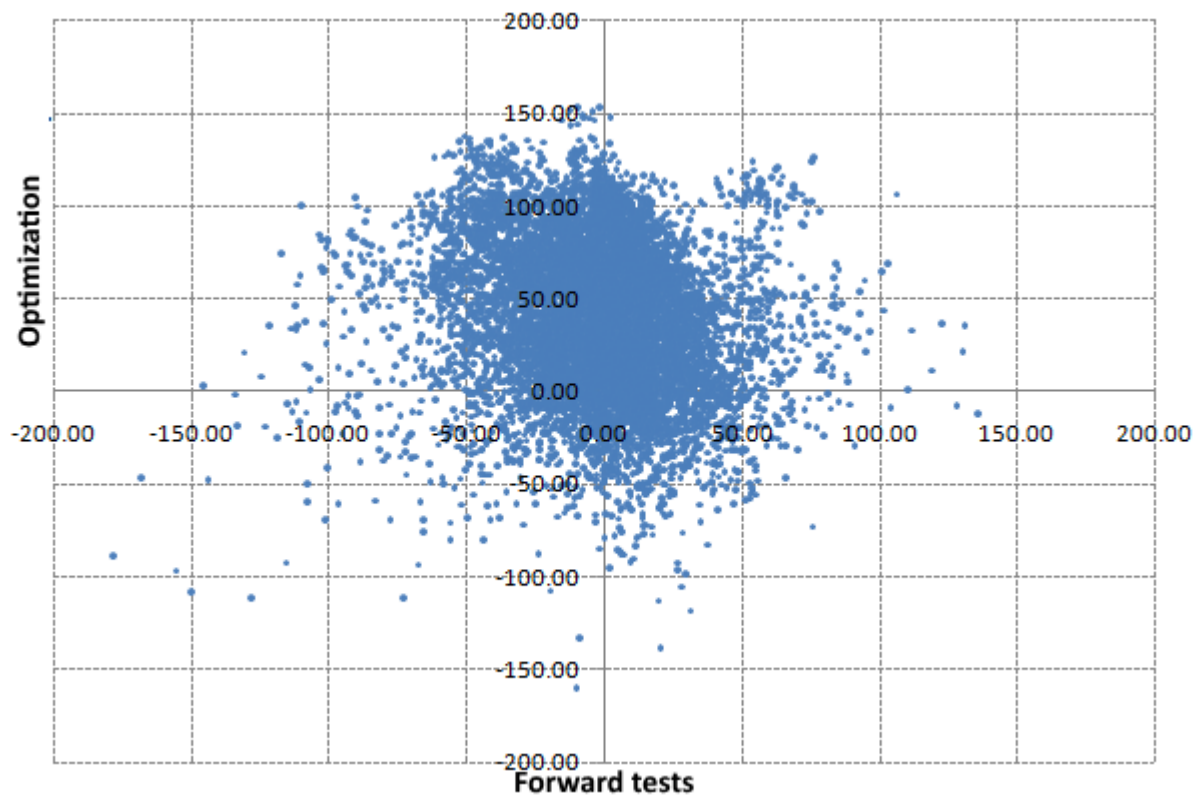
在一个复杂准则中，我们观察到一个多层次结构体，因为它是一个有条件的公式计算的：在某处一个分支工作，在另一处另一个分支工作。我们的自定义准则总是使用相同的公式计算。我们还注意到在我们的准则中存在负值（这是意料之中的），并且复杂准则的声明范围为 0-100。

我们通过分析递进测试的值来检查我们的准则是否完善。



优化和递进测试期间的自定义准则值

正如预期的那样，只有一部分良好的优化指标保持继续使用。但我们更感兴趣的不是准则，而是利润。我们来看看在优化与远期测试过程中盈利的分布情况。



优化和远期测试期间的利润

此处的图呈现类似规律。在优化期盈利的 6850 轮测试中，有 3123 轮在远期测试中也实现了盈利（占比 45%）。而在前 1000 轮最优测试结果中，仅有 323 轮在远期测试中盈利，这一结果并不理想。因此，该 EA 交易需要做大量的工作来确定稳定的盈利设置。但也许是优化准则的问题呢？

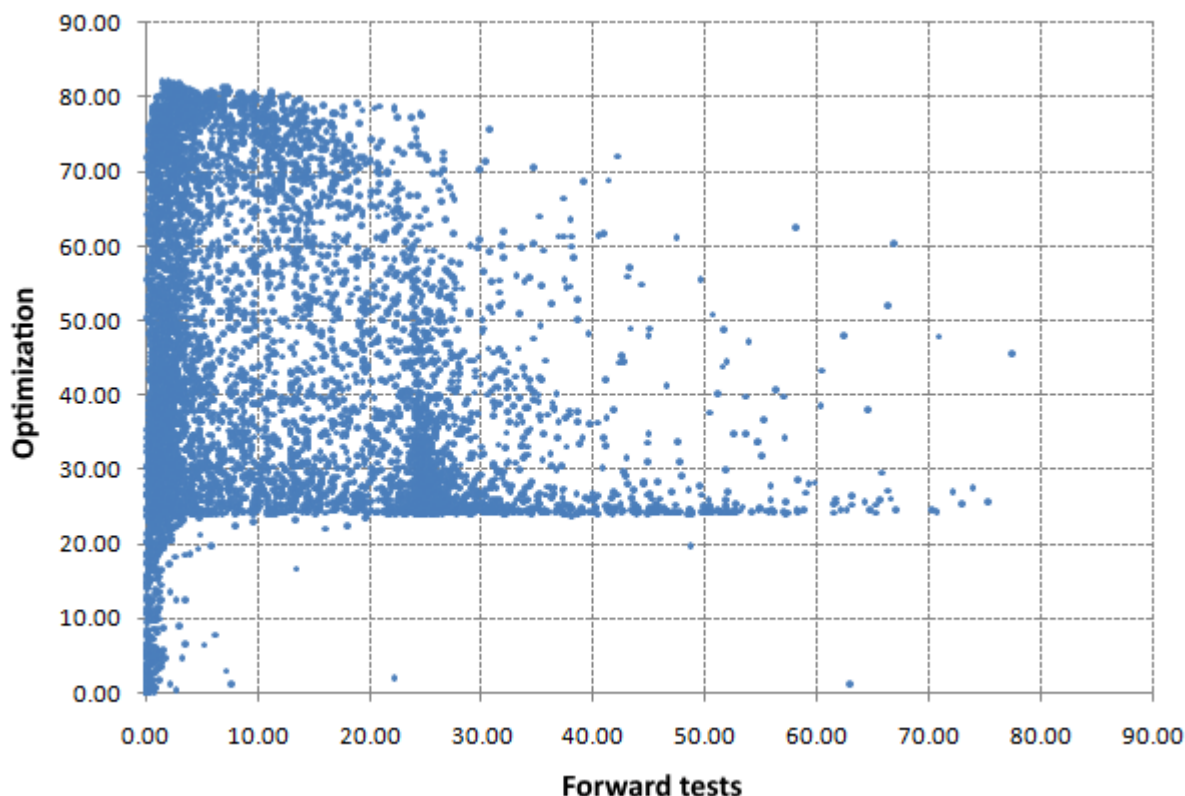
我们重复优化，这次使用内置的复杂准则。

注意！MetaTrader 5 在优化过程中会在以下目录中生成 **Tester/cache** 优化缓存，即 `opt` 文件。当开始下一次优化时，它会寻找合适的缓存来继续优化。如果存在具有先前设置的缓存文件，则该过程不会从头开始，而是会考虑先前的结果。这允许你建立遗传优化链，假设你找到了最好的结果（毕竟，每个遗传优化都是一个随机过程）。

MetaTrader 5 没有将优化准则作为设置中的一个区分因素。根据前面所述，这在某些情况下可能是有用的，但它会干扰我们当前的任务。要进行纯实验，我们需要从头开始优化。因此，在使用我们的标准进行第一次优化之后，我们不能立即使用复杂准则启动第二次优化。

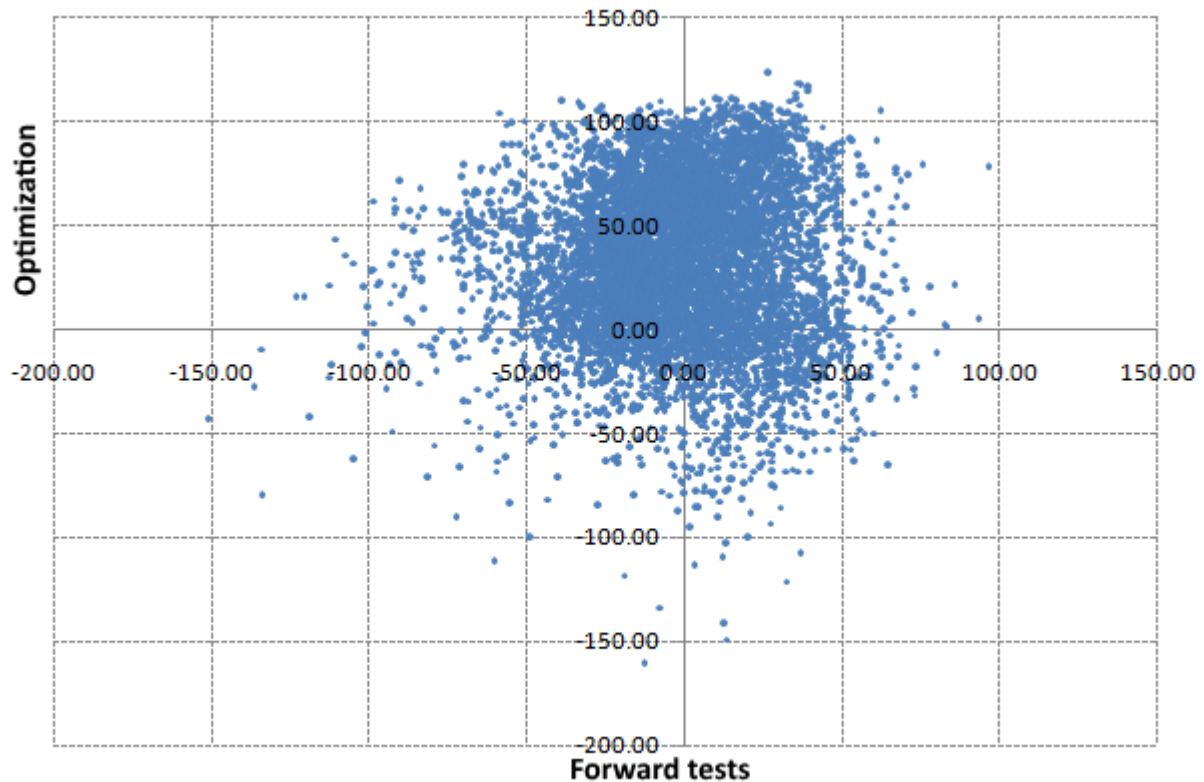
无法从终端界面禁用当前行为。因此，你应在任何文件管理器中手动删除或重命名（更改扩展名）以前的 `opt` 文件。稍后，我们将熟悉测试程序的预处理程序指令 `tester_no_cache`，该指令可以在特定 EA 交易的源代码中指定，允许你禁用缓存读取。

优化和远期测试期间的复合准则值比较采用以下形式。



优化和远期测试期间的复杂准则

以下是远期测试期间利润的稳定性。



优化和远期测试期间的利润

在历史内的 5952 个正向结果中，只有 2655 个（也是约 45%）在远期测试中保持盈利但是在最初的 1000 个正向结果中，有 581 在远期测试中表现成功。

因此，我们已经看到，从技术角度来看，使用 *OnTester* 非常简单，但我们的准则比内置准则（其他条件不变）更差，尽管内置准则也并非完美。因此，无论是探索评估准则的公式本身，还是不预知未来的情况下合理选择参数，*OnTester* 自身还有很多值得探索的地方。

至此，编程已逐渐演变为研究与科学探索，这超出了本书的范畴。但是我们将给出一个根据我们自己的度量指标（而不是现成的度量值）比较计算的准则示例：*TesterStatistics*。我们将讨论 R^2 准则，也称为决策系数 (*RSquared.mqh*)。

我们创建一个通过余额曲线计算 R^2 的函数。众所周知，当以固定手数交易时，理想的交易系统应以直线的形式显示余额。我们现在使用固定手数，因此更适合我们目前的情况。对于 R^2 ，在可变手数的情况下，我们将稍后再做讨论。

最后，相对于建立在数据基础上的线性回归， R^2 为数据方差的倒数。 R^2 值的范围为从负无穷大到 +1（尽管在我们的示例中不太可能出现大的负值）。显然，找到的线同时具有斜率的特性，因此，为了使代码通用化，我们将保存 R^2A 结构体中的 R^2 和角度的正切值作为中间结果。

```
struct R2A
{
    double r2;    // square of correlation coefficient
    double angle; // tangent of the slope
    R2A(): r2(0), angle(0) { }
};
```

指标的计算是在 *RSquared* 函数中执行的，该函数将一个数据数组作为输入，并返回一个 R^2A 结构体。

```

R2A RSquared(const double &data[])
{
    int size = ArraySize(data);
    if(size <= 2) return R2A();
    double x, y, div;
    int k = 0;
    double Sx = 0, Sy = 0, Sxy = 0, Sx2 = 0, Sy2 = 0;
    for(int i = 0; i < size; ++i)
    {
        if(data[i] == EMPTY_VALUE
        || !MathIsValidNumber(data[i])) continue;
        x = i + 1;
        y = data[i];
        Sx += x;
        Sy += y;
        Sxy += x * y;
        Sx2 += x * x;
        Sy2 += y * y;
        ++k;
    }
    size = k;
    const double Sx22 = Sx * Sx / size;
    const double Sy22 = Sy * Sy / size;
    const double SxSy = Sx * Sy / size;
    div = (Sx2 - Sx22) * (Sy2 - Sy22);
    if(fabs(div) < DBL_EPSILON) return R2A();
    R2A result;
    result.r2 = (Sxy - SxSy) * (Sxy - SxSy) / div;
    result.angle = (Sxy - SxSy) / (Sx2 - Sx22);
    return result;
}

```

为了优化，我们需要一个标准值，此处的角度是很重要的，因为负斜率的平滑资金曲线可能仍会获得较高 R2 值。因此我们将再编写一个函数，该函数将对任何具有负角度的 R2 估算值取相反数。我们采用 R2 模的值，因为在非常差的（分散的）数据不适合我们的线性模型的情况下，该值本身可以是负的。因此，我们必须防止出现负负得正的情况。

```

double RSquaredTest(const double &data[])
{
    const R2A result = RSquared(data);
    const double weight = 1.0 - 1.0 / sqrt(ArraySize(data) + 1);
    if(result.angle < 0) return -fabs(result.r2) * weight;
    return result.r2 * weight;
}

```

此外，我们的准则考虑了系列的大小，其对应于交易次数。因此，交易次数的增加需要增加指标。

有了这个工具，我们将在 EA 交易中实现资金曲线计算函数，并为该函数找到 R2。最后，我们将该值乘以 100，从而将标度转换为内置复杂准则的范围。

```

#define STAT_PROPS 4

double GetR2onBalanceCurve()
{
    HistorySelect(0, LONG_MAX);

    const ENUM_DEAL_PROPERTY_DOUBLE props[STAT_PROPS] =
    {
        DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE
    };
    double expenses[][STAT_PROPS];
    ulong tickets[]; // only needed because of the 'select' prototype, but useful for

    DealFilter filter;
    filter.let(DEAL_TYPE, (1 << DEAL_TYPE_BUY) | (1 << DEAL_TYPE_SELL), IS::OR_BITWISE
        .let(DEAL_ENTRY,
            (1 << DEAL_ENTRY_OUT) | (1 << DEAL_ENTRY_INOUT) | (1 << DEAL_ENTRY_OUT_BY),
            IS::OR_BITWISE)
        .select(props, tickets, expenses);

    const int n = ArraySize(tickets);

    double balance[];

    ArrayResize(balance, n + 1);
    balance[0] = TesterStatistics(STAT_INITIAL_DEPOSIT);

    for(int i = 0; i < n; ++i)
    {
        double result = 0;
        for(int j = 0; j < STAT_PROPS; ++j)
        {
            result += expenses[i][j];
        }
        balance[i + 1] = result + balance[i];
    }
    const double r2 = RSquaredTest(balance);
    return r2 * 100;
}

```

在 *OnTester* 处理程序中，我们可使用条件编译指令下的新标准，因此我们需要取消源代码开头的 *#define USE_R2_CRITERION* 指令的注释。

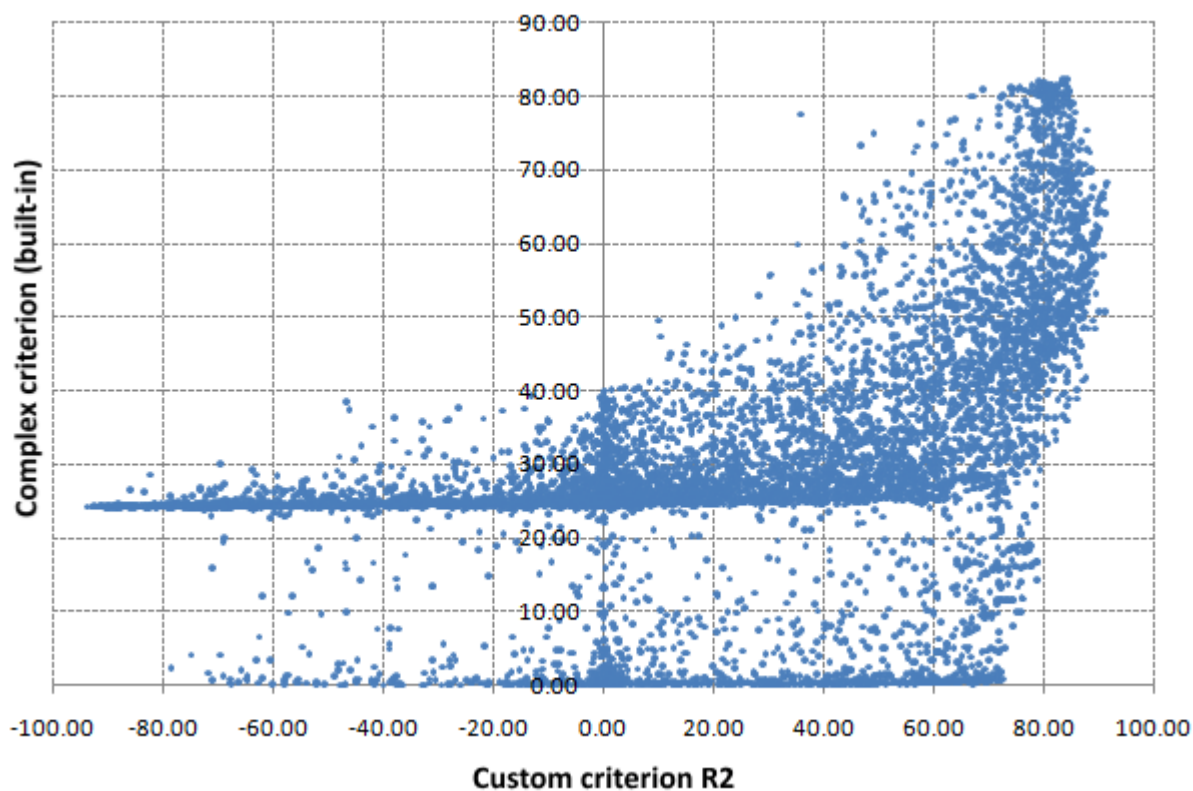
```

double OnTester()
{
#ifdef USE_R2_CRITERION
    return GetR2onBalanceCurve();
#else
    const double profit = TesterStatistics(STAT_PROFIT);
    return sign(profit) * sqrt(fabs(profit))
        * sqrt(TesterStatistics(STAT_PROFIT_FACTOR))
        * sqrt(TesterStatistics(STAT_TRADES))
        * sqrt(fabs(TesterStatistics(STAT_SHARPE_RATIO)));
#endif
}

```

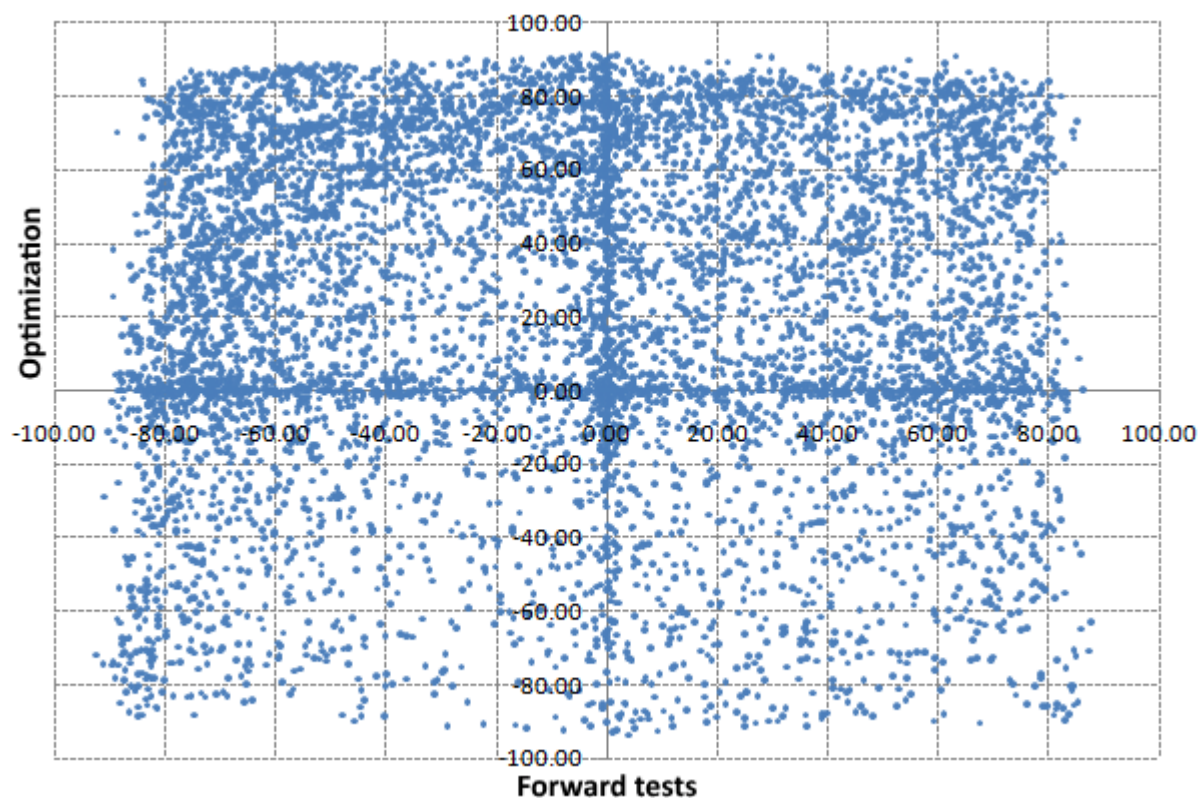
我们删除之前的优化结果（带缓存的 opt 文件），并启动 EA 交易的新优化：根据 R2 准则。

当比较 R2 准则和复杂准则的值时，我们可以说它们之间的“趋同性”提高了。



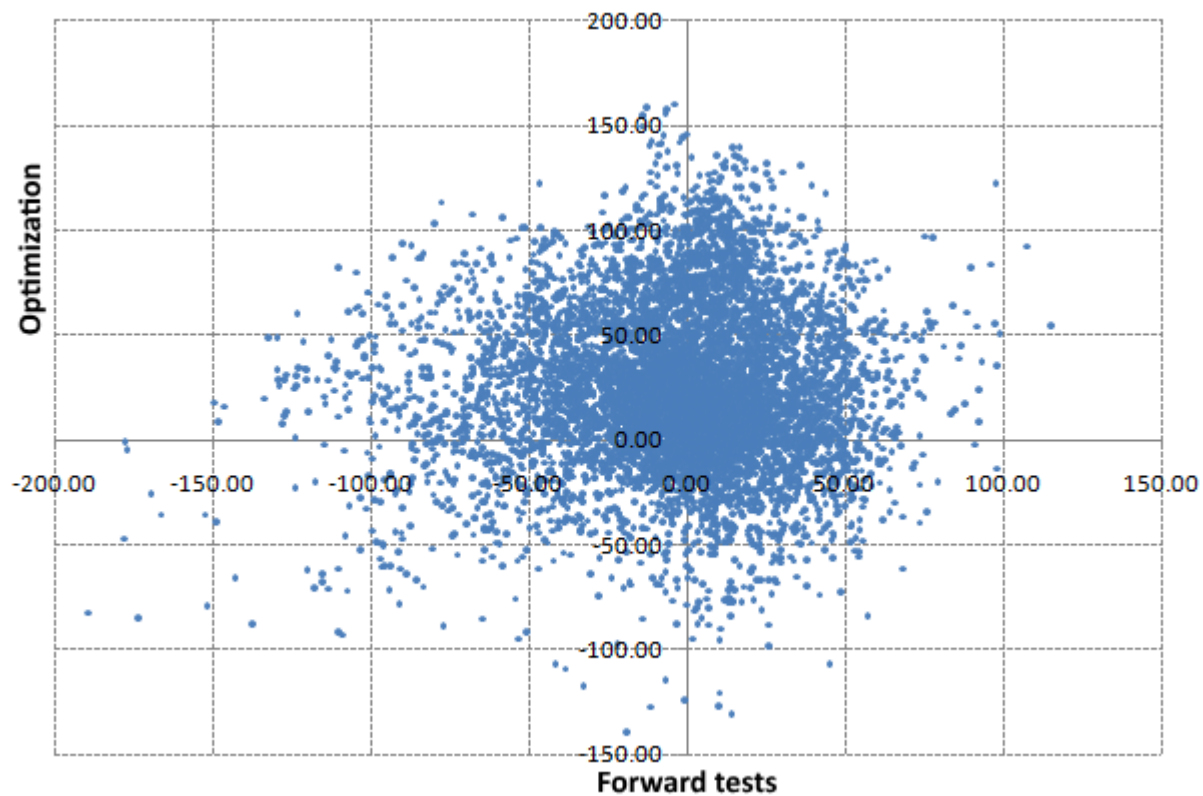
自定义准则 R2 与复杂内置准则的比较

对于相应的参数组，优化窗口中和远期测试的 R2 标准的值如下所示。



优化和远期测试期间的 R2 标准

此处是如何将过去和未来的利润结合在一起的说明。



R2 优化和远期测试期间的利润

统计数据如下：在最近 5582 轮盈利的测试中，有 2638 次 (47%) 在远期测试中仍保持盈利；而在前 1000 次盈利最高的测试中，有 566 次在远期测试中盈利，这一表现与内置复杂准则相当。

如上所述，统计数据为下一次更智能的优化阶段提供了原始资料，这不仅仅是一项编程任务。我们将专注于优化的其他纯编程方面。

6.5.8 自动调优：ParameterGetRange 和 ParameterSetRange

上一节，我们学习了如何将一个优化标准传递给测试程序。但是，我们忽略了重要的一点。如果你查看我们的优化日志，会看到许多错误消息，如下所示。

```
...
Best result 90.61004580175876 produced at generation 25. Next generation 26
genetic pass (26, 388) tested with error "incorrect input parameters" in 0:00:00.021
genetic pass (26, 436) tested with error "incorrect input parameters" in 0:00:00.007
genetic pass (26, 439) tested with error "incorrect input parameters" in 0:00:00.007
genetic pass (26, 363) tested with error "incorrect input parameters" in 0:00:00.008
genetic pass (26, 365) tested with error "incorrect input parameters" in 0:00:00.008
...
```

换言之，每经过几次测试传递，输入参数就会有问题，这样的传递就无法执行。*OnInit* 处理程序包含以下检查：

```
if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;
```

对我们而言，施加这样的限制是很合理的，即慢速 MA 的周期应大于快速 MA 的周期。但是，测试程序并不知道我们算法的此类要求，因此会尝试对各种周期组合进行排序，包括不正确的组合。对于优化而言，这可能是一种常见的情况，但是却有负面后果。

由于我们应用了遗传优化，每一代中有几个被拒绝的样本不会参与进一步的突变。MetaTrader 5 优化器不会弥补这些损失，也就是说，不会为这些被拒绝的样本生成替代品。然后，样本数量较小会对质量产生负面影响。因此，有必要考虑一种方法来确保输入设置仅在正确的组合中被枚举。此处有两个 MQL5 API 函数可提供帮助：*ParameterGetRange* 和 *ParameterSetRange*。

这两个函数都有两个参数类型不同的重载原型：*long* 和 *double*。这是对 *ParameterGetRange* 函数的两种变体说明。

```
bool ParameterGetRange(const string name, bool &enable, long &value, long &start, long &step, long &stop)
```

```
bool ParameterGetRange(const string name, bool &enable, double &value, double &start, double &step, double &stop)
```

对于按名称指定的输入变量，函数在优化过程中接收有关其当前值 (*value*)、值范围 (*start*, *stop*) 和变化步长 (*step*) 的信息。此外，还会向 *enable* 变量写入一个特性，说明是否是名为 'name' 的输入变量启用了优化。

该函数返回成功 (*true*) 或错误 (*false*) 的指示。

该函数只能通过三个与优化相关的特殊处理程序中调用：*OnTesterInit*、*OnTesterPass* 和 *OnTesterDeinit*。我们将在 下一节 中介绍。正如你从名称中可以猜到的，在优化开始之前调用 *OnTesterInit*，在优化完成之后调用 *OnTesterDeinit*，在优化过程中的每一次传递之后调用 *OnTesterPass*。目前，我们只对 *OnTesterInit* 感兴趣。就像其他两个函数一样，该函数没有参数，可以用 *void* 类型进行说明，也就是说，不会返回任何值。

ParameterSetRange 函数的两个版本具有相似的原型，并执行相反的操作：它们可设置 EA 交易的输入参数的优化特性。

```
bool ParameterSetRange(const string name, bool enable, long value, long start, long step, long stop)
bool ParameterSetRange(const string name, bool enable, double value, double start, double step,
double stop)
```

该函数在优化当前值、变化步长和值范围时为名为 *name* 的 *input* 变量设置修改规则。

在策略测试程序中启动优化时，只能通过 *OnTesterInit* 处理程序调用此函数。

因此，使用 *ParameterGetRange* 和 *ParameterSetRange* 函数，你可以分析并设置新的范围和步长值以及从优化中完全排除或包括某些参数，而不管策略测试程序中的设置如何。这允许你创建自己的脚本来管理优化期间的输入参数空间。

该函数甚至允许你在优化中使用那些用 *sinput* 修饰符声明的变量（用户没有将这些变量包含在优化中）。

注意！在调用 *ParameterSetRange* 并更改特定输入变量的设置之后，对 *ParameterGetRange* 的后续调用将不会“看到”这些更改，仍然会返回到原始设置。因此在复杂软件产品中不能一起使用这些函数，在复杂软件产品中，设置可以由独立开发人员的不同类和库来处理。

我们使用新函数来改进 *BandOsMA* EA 交易。更新后的版本命名为 *BandOsMApro.mq5*（“pro”可以有条件解码为“参数范围优化”）。

因此，我们有 *OnTesterInit* 处理程序，在其中我们可以读取 *FastOsMA* 和 *SlowOsMA* 参数的设置，并检查它们是否包含在优化中。如果包含，则需要关掉它们，并提供一些参数作为回报。


```

void OnTesterInit()
{
    bool enabled1, enabled2;
    long value1, start1, step1, stop1;
    long value2, start2, step2, stop2;
    if(ParameterGetRange("FastOsMA", enabled1, value1, start1, step1, stop1)
    && ParameterGetRange("SlowOsMA", enabled2, value2, start2, step2, stop2))
    {
        if(enabled1 && enabled2)
        {
            if(!ParameterSetRange("FastOsMA", false, value1, start1, step1, stop1)
            || !ParameterSetRange("SlowOsMA", false, value2, start2, step2, stop2))
            {
                Print("Can't disable optimization by FastOsMA and SlowOsMA: ",
                    E2S(_LastError));
                return;
            }
            ...
        }
    }
    else
    {
        Print("Can't adjust optimization by FastOsMA and SlowOsMA: ", E2S(_LastError));
    }
}

```

不幸的是，由于添加了 *OnTesterInit*，编译器也会要求添加 *OnTesterDeinit*，虽然我们并不需要这个函数。但是我们被迫同意并添加一个空白处理程序。

```

void OnTesterDeinit()
{
}

```

代码中存在 *OnTesterInit/OnTesterDeinit* 函数将导致这样一个事实，即当优化开始时，将在终端中打开一个额外的图表，在这个图表上运行我们的 EA 交易副本。该副本会以一种特殊的模式工作，允许你从代理的测试副本接收额外的数据（所谓的 *frames*），但我们将在后面探讨这种可能性。现在，我们需要注意的是，所有文件、日志、图表和对象的操作都像往常一样直接在终端（而不是代理）的 EA 交易的辅助副本中进行。特别是，所有错误消息和 *Print* 调用都将显示在终端的 *Experts* 选项卡上的日志中。

我们关于这些参数变化范围和步骤的信息，我们可以准确地重新计算所有正确的组合。这个任务被分配给一个单独的 *Iterate* 函数，因为类似的操作必须由 *OnInit* 处理程序中代理的 EA 交易副本来重现。

在 *Iterate* 函数中，我们在快速和慢速 MA 周期上有两个嵌套循环，在这两个循环中，我们可统计有效组合的数量，比如，当 *i* 周期小于 *j* 时。当从 *OnInit* 调用 *Iterate* 时，我们需要可选的 *find* 参数，以通过组合 *i* 和 *j* 的序列号返回该组合对。由于需要返回 2 个数字，因此我们为其说明了 *PairOfPeriods* 结构体。

```

struct PairOfPeriods
{
    int fast;
    int slow;
};

PairOfPeriods Iterate(const long start1, const long stop1, const long step1,
    const long start2, const long stop2, const long step2,
    const long find = -1)
{
    int count = 0;
    for(int i = (int)start1; i <= (int)stop1; i += (int)step1)
    {
        for(int j = (int)start2; j <= (int)stop2; j += (int)step2)
        {
            if(i < j)
            {
                if(count == find)
                {
                    PairOfPeriods p = {i, j};
                    return p;
                }
                ++count;
            }
        }
    }
    PairOfPeriods p = {count, 0};
    return p;
}

```

从 *OnTesterInit*调用 *Iterate* 时，我们不使用 *find* 参数，而是一直计数到最后，并在结构体的第一个字段中返回得到的数量。这将是一些新的影子参数的值范围，我们必须为此启用优化。姑且称之为 *FastSlowCombo4Optimization*，然后加入新的一组辅助输入参数。此处，更多参数将很快添加进去。

```

input group "A U X I L I A R Y"
sinput int FastSlowCombo4Optimization = 0;    // (reserved for optimization)
...

```

我们回到 *OnTesterInit*，使用 *ParameterSetRange* 在所需的范围内通过 *FastSlowCombo4Optimization* 参数组织一次 MQL5 优化。

```

void OnTesterInit()
{
    ...
    PairOfPeriods p = Iterate(start1, stop1, step1, start2, stop2, step2);
    const int count = p.fast;
    ParameterSetRange("FastSlowCombo4Optimization", true, 0, 0, 1, count);
    PrintFormat("Parameter FastSlowCombo4Optimization is enabled with maximum: %
        count);
    ...
}

```

请注意，新参数的迭代次数应显示在终端日志中。

在代理上进行测试时，使用 *FastSlowCombo4Optimization* 中的数字，通过再次调用 *Iterate* 可获得几个周期，这次可使用填充的 *find* 参数。但问题是，对于该操作，需要知道初始范围以及 *FastOsMA* 和 *SlowOsMA* 参数的变化步长。该信息仅存在于终端中。所以，我们需要以某种方式转移给代理。

现在我们将应用迄今为止我们知道的唯一解决方案：增加 3 个影子优化参数，并为其设置一些值。将来，我们将熟悉将文件传输给代理的技术（参见 [测试程序的预处理指令](#)）。然后便可以把 *Iterate* 函数计算的整个索引数组写入文件，并将其发送给代理。这样可避免使用三个额外的影子优化参数。

因此，我们添加三个输入参数：

```

input ulong FastShadow4Optimization = 0;    // (reserved for optimization)
input ulong SlowShadow4Optimization = 0;    // (reserved for optimization)
input ulong StepsShadow4Optimization = 0;    // (reserved for optimization)

```

我们使用 *ulong* 类型更为合理：将 2 个 *int* 数打包到每个值中。这是它们在 *OnTesterInit* 中的填写方式。

```

void OnTesterInit()
{
    ...
    const ulong fast = start1 | (stop1 << 16);
    const ulong slow = start2 | (stop2 << 16);
    const ulong step = step1 | (step2 << 16);
    ParameterSetRange("FastShadow4Optimization", false, fast, fast, 1, fast);
    ParameterSetRange("SlowShadow4Optimization", false, slow, slow, 1, slow);
    ParameterSetRange("StepsShadow4Optimization", false, step, step, 1, step);
    ...
}

```

所有 3 个参数都是不可优化的（第二个自变量为 *false*）。

对 *OnTesterInit* 函数的操作到此结束。我们来看接收方：*OnInit* 处理程序。

```

int OnInit()
{
    // keep the check for single tests
    if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;

    // when optimizing, we require the presence of shadow parameters
    if(MQLInfoInteger(MQL_OPTIMIZATION) && StepsShadow4Optimization == 0)
    {
        return INIT_PARAMETERS_INCORRECT;
    }

    PairOfPeriods p = {FastOsMA, SlowOsMA}; // by default we work with normal parameters
    if(FastShadow4Optimization && SlowShadow4Optimization && StepsShadow4Optimization)
    {
        // if the shadow parameters are full, decode them into periods
        int FastStart = (int)(FastShadow4Optimization & 0xFFFF);
        int FastStop = (int)((FastShadow4Optimization >> 16) & 0xFFFF);
        int SlowStart = (int)(SlowShadow4Optimization & 0xFFFF);
        int SlowStop = (int)((SlowShadow4Optimization >> 16) & 0xFFFF);
        int FastStep = (int)(StepsShadow4Optimization & 0xFFFF);
        int SlowStep = (int)((StepsShadow4Optimization >> 16) & 0xFFFF);

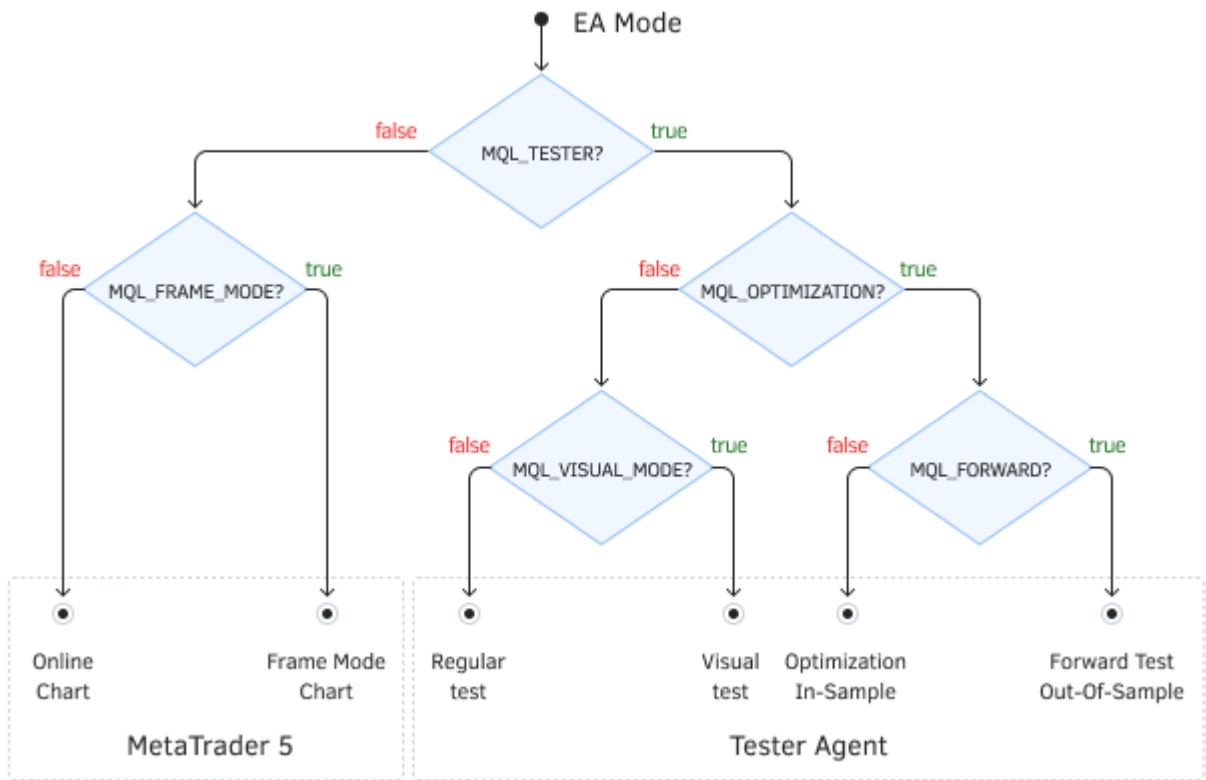
        p = Iterate(FastStart, FastStop, FastStep,
            SlowStart, SlowStop, SlowStep, FastSlowCombo4Optimization);
        PrintFormat("MA periods are restored from shadow: FastOsMA=%d SlowOsMA=%d",
            p.fast, p.slow);
    }

    strategy = new SimpleStrategy(
        new BandOsMaSignal(p.fast, p.slow, SignalOsMA, PriceOsMA,
            BandsMA, BandsShift, BandsDeviation,
            PeriodMA, ShiftMA, MethodMA),
        Magic, StopLoss, Lots);
    return INIT_SUCCEEDED;
}

```

使用 *MQLInfoInteger* 函数，我们可以确定所有的 EA 交易模式，包括那些与测试和优化相关的模式。将 *ENUM_MQL_INFO_INTEGER* 枚举的一个元素指定为参数后，我们将得到一个逻辑符号作为结果 (*true/false*):

- *MQL_TESTER* – 该程序在测试程序中运行
- *MQL_VISUAL_MODE* – 测试程序在可视化模式下运行
- *MQL_OPTIMIZATION* – 测试过程在优化过程中进行 (不是单独运行)
- *MQL_FORWARD* – 在优化后的递进期间执行试过程 (如果由优化设置指定)
- *MQL_FRAME_MODE* – EA 交易在终端图表上 (而不是在代理上) 以特殊服务模式运行，以控制优化 (下一节将详细介绍)



MQL 程序的测试模式

一切准备就绪，开始优化。一旦启动，使用所述的设置 *Presets/MQL5Book/BandOsMA.set*，我们将在终端的 *Experts* 日志中看到一条消息：

```
Parameter FastSlowCombo4Optimization is enabled with maximum: 698
```

这一次，优化日志中应没有错误，正常生成所有内容，不会崩溃。

```
...
Best result 91.02452934181422 produced at generation 39. Next generation 42
Best result 91.56338892567393 produced at generation 42. Next generation 43
Best result 91.71026391877101 produced at generation 43. Next generation 44
Best result 91.71026391877101 produced at generation 43. Next generation 45
Best result 92.48460871443507 produced at generation 45. Next generation 46
...
```

这甚至可以通过增加的整体优化时间来确定：早先，一些传递在早期阶段被拒绝，现在它们都得到了完全处理。

但是我们的解决方案也有一个缺点。现在，EA 交易的工作设置不仅包括 *FastOsMA* 和 *SlowOsMA* 参数中的几个周期，还包括它们在所有可能组合中的序号 (*FastSlowCombo4Optimization*)。我们唯一能做的就是输出 *Onlnit* 函数中解码的周期，这在上面已经演示过了。

因此，在优化的帮助下找到好的设置后，用户将像往常一样执行一次运行来完善交易系统的行为。在测试日志的开头，应出现以下形式的内容：

```
MA periods are restored from shadow: FastOsMA=27 SlowOsMA=175
```

然后，你可以在同名参数中输入指定的周期，并重置所有影子参数。

6.5.9 用于优化控制的一组 OnTester 事件

MQL5 中有三个特殊事件，用于管理优化过程，并将任意应用的结果（除了交易指标之外）从代理传输到终端：**OnTesterInit**、**OnTesterDeinit** 和 **OnTesterPass**。在代码中说明了其处理程序之后，程序员能够在开始优化之前、优化完成之后以及每个单独的优化传递结束时执行其所需的操作（如果已经从代理接收到应用程序数据，下面将详细介绍）。

所有处理程序都是可选的。正如我们已经看到的，优化并不依赖于处理程序。还需要理解的是，所有三个事件只在优化期间起作用，在单次测试中不起作用。

带有这些处理程序的 EA 交易可自动加载到终端的单独图表上，并带有测试程序中指定的符号和周期。这种 EA 交易实例不进行交易，只执行服务操作。所有其他事件处理程序，如 **OnInit**、**OnDeinit** 和 **OnTick** 在其中不起作用。

要确定 EA 交易是在代理的常规交易模式下执行还是在终端的服务模式下执行，请在其代码中调用函数 **MQLInfoInteger(MQL_FRAME_MODE)**，并获取 **true** 或 **false**。这种服务模式也称为“帧”模式，适用于可以从代理上的 EA 交易实例发送到终端的数据包。稍后我们将看到该过程是如何完成的。

在优化过程中，只有一个 EA 交易实例在终端中工作，并且在必要时接收输入帧。不要忘记，只有当 EA 交易代码包含三个所述的事件处理程序之一时，才会启动这种实例。

在第一轮之前，在策略测试程序中启动优化时，会生成 **OnTesterInit** 事件。该处理程序有两个版本：返回类型为 **int** 和 **void**。

int OnTesterInit(void)
void OnTesterInit(void)

在 **int** 返回版本中，零值 (**INIT_SUCCEEDED**) 意味着在终端中图表上启动的 EA 交易成功初始化，允许开始优化。任何其他值都意味着返回错误代码，并且不会开始优化。

该函数的第二个版本始终表明 EA 交易为优化做好了准备。

为执行 **OnTesterInit** 提供了有限时间，在此之后，EA 交易将被强制终止，优化本身将被取消。在这种情况下，测试程序的日志中会显示相应的消息。

在上一节中，我们看到了如何使用 **OnTesterInit** 处理程序通过 **ParameterGetRange/ParameterSetRange** 函数修改优化参数的示例。

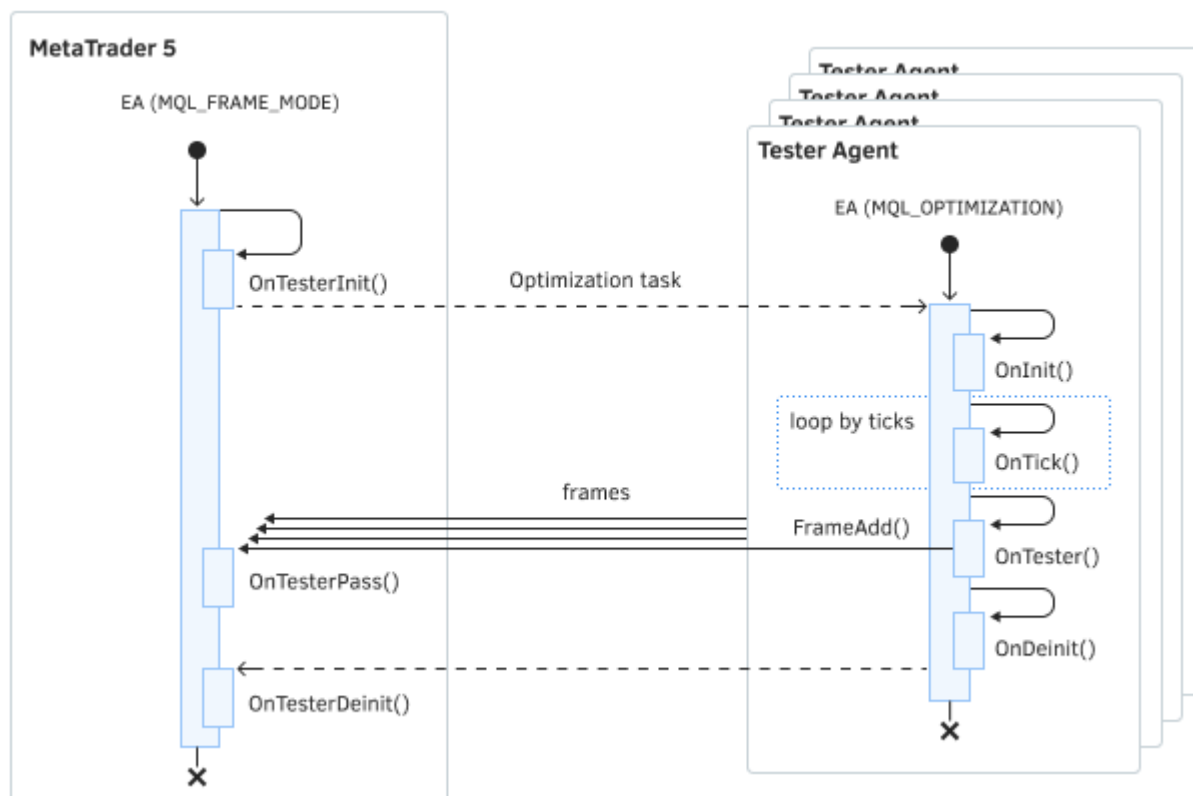
void OnTesterDeinit(void)

EA 交易优化完成后，将调用 **OnTesterDeinit** 函数。

该函数旨在对应用的优化结果进行最终处理。例如，如果在 **OnTesterInit** 中打开一个文件来写入帧的内容，那么需要在 **OnTesterDeinit** 中关闭。

void OnTesterPass(void)

在优化过程中，当数据帧到达时，会自动生成 **OnTesterPass** 事件。该函数允许在优化期间处理从运行在代理上的 EA 交易实例接收的应用程序数据。测试代理的帧必须通过 **OnTester** 处理程序，使用 **FrameAdd** 函数发送。



该图显示了优化 EA 交易时的事件顺序

关于每轮测试的标准财务统计数据会从代理自动发送到终端。如果不需要，EA 交易不需要使用 **FrameAdd** 发送任何内容。如果没有使用帧，就不会调用 **OnTesterPass** 处理程序。

使用 **OnTesterPass**，你可以“随时”动态处理优化结果，例如，将它们显示在终端中的图表上，或者将它们添加到文件中，以便进行后续的批处理。

为了演示 **OnTester** 事件处理程序的功能，我们首先需要学习处理帧的函数。这些函数将在后面的章节中介绍。

6.5.10 从代理向终端发送数据帧

除了标准的财务指标和统计数据之外，MQL5 还提供了一组函数，用于组织传输和处理你自己的（应用的）优化结果。其中一个 **FrameAdd**，用于发送来自测试代理的数据。其他函数用于接收终端中的数据。

数据交换格式基于帧。这是一种特殊的内部结构体，EA 交易可以基于简单类型的数组（不包含字符串、类对象或动态数组）或使用具有指定名称的文件（该文件必须首先在代理的沙箱中创建）来填充测试程序。通过多次调用 **FrameAdd** 函数，EA 交易可以向终端发送一系列帧。帧的数量没有限制。

FrameAdd 函数有两个版本。

```
bool FrameAdd(const string name, ulong id, double value, const string filename)
bool FrameAdd(const string name, ulong id, double value, const void &data[])
```

该函数可将数据帧添加到缓冲区，以发送到终端。**name** 和 **id** 参数为公共标签，用于在 **FrameFilter** 函数中筛选帧。**value** 参数允许你传递一个任意数值，当一个值足够大时即可使用。更大批量的数据显示在 **data** 数组（可能是简单结构体的数组）或名为 **filename** 的文件中。

如果没有要传输的批量数据（例如，只需要传输该过程的状态），请使用函数的第一种形式并指定 `NULL`，而不是使用带文件名的字符串，或者使用大小为零的虚拟数组的第二种形式。

如果成功，则该函数返回 `true`。

该函数只能在 `OnTester` 处理程序中调用。

该函数在简单测试期间（即未经优化）调用时没有任何效果。

只能从代理向终端发送数据。MQL5 中没有在优化期间以相反方向发送数据的机制。在开始优化之前，EA 交易想要发送给代理的所有数据都必须准备好并可用（以输入参数或由指令连接的文件形式）。

在下一节我们熟悉了主机的函数之后，我们将探讨一个使用 `FrameAdd` 的示例。

6.5.11 在终端获取数据帧

通过 `FrameAdd` 函数从测试代理发送的帧被传送到终端，并按照接收顺序写入 `mqd` 文件，该文件以 EA 交易的名称命名，位于文件夹 `terminal_directory/MQL5/Files/Tester` 中。一个或多个帧同时到达会生成 `OnTesterPass` 事件。

MQL5 API 提供了下列 4 个用于分析和读取帧的函数：`FrameFirst`、`FrameFilter`、`FrameNext` 和 `FrameInputs`。所有函数都会返回一个布尔值，指示成功 (`true`) 或错误 (`false`)。

为了访问现有的帧，内核采用了指向当前帧的内部指针这一隐喻机制。当 `FrameNext` 函数读取下一帧时，指针会自动向前移动，但可以使用 `FrameFirst` 或 `FrameFilter` 返回到所有帧的开头。因此，MQL 程序可以在一个循环中组织帧的迭代，直到检查完所有的帧。如果需要，可以重复这个过程，例如，通过在 `OnTesterDeinit` 中应用不同的筛选器。

`bool FrameFirst()`

`FrameFirst` 函数可将内部帧读取指针设置为开始，并重置筛选器（如果之前使用 `FrameFilter` 函数设置了筛选器）。

理论上，对于所有帧的单次接收和处理，没有必要调用 `FrameFirst`，因为当优化开始时，指针已经在开头位置。

`bool FrameFilter(const string name, ulong id)`

该函数可设置帧读取筛选器并将内部帧指针设置为开头位置。筛选器将影响 `FrameNext` 的后续调用中包含哪些帧。

如果一个空字符串作为第一个参数传递，筛选器将只通过一个数字参数工作，即所有具有指定 `id` 的帧。如果第二个参数的值等于 `ULONG_MAX`，那么只有文本筛选器起作用。

调用 `FrameFilter("", ULONG_MAX)` 相当于调用 `FrameFirst()`，相当于没有筛选器。

如果你在 `OnTesterPass` 中调用 `FrameFirst` 或 `FrameFilter`，请确保确实是你所需要的：其代码可能包含逻辑错误，因为它可能会循环、读取相同帧或以指数方式增加计算负载。

`bool FrameNext(ulong &pass, string &name, ulong &id, double &value)`

`bool FrameNext(ulong &pass, string &name, ulong &id, double &value, void &data[])`

`FrameNext` 函数可读取一帧并将指针移动到下一帧。`pass` 参数中可记录优化测试轮次编号。`name`、`id` 和 `value` 参数可接收在 `FrameAdd` 函数的相应参数中传递的值。

值得注意的是，当没有更多帧需要读取时，该函数在正常运行时可能会返回 *false*。在这种情况下，内置变量 *_LastError* 包含值 4000（没有内置符号）。

无论使用哪种形式的 *FrameAdd* 函数来发送数据，文件或数组的内容都会放在接收 *data* 数组中。接收数组的类型必须与发送数组的类型相匹配，在发送文件的情况下有一些细微差别。

二进制文件（FILE_BIN）最好在字节数组 *uchar* 中接受，以确保与任何大小兼容（因为其他更大的类型可能不是文件大小的倍数）。如果文件大小（实际上为接收帧中数据块的大小）不是接收数组类型大小的倍数，*FrameNext* 函数将不会读取数据，并将返回 INVALID_ARRAY (4006) 错误。

Unicode 文本文件（没有 FILE_ANSI 修饰符的 FILE_TXT 或 FILE_CSV）应被 *ushort* 类型的数组接受，然后通过调用 *ShortArrayToString* 转换为字符串。ANSI 文本文件应以 *uchar* 数组的形式接收，并使用 *CharArrayToString* 进行转换。

bool FrameInputs(ulong pass, string &parameters[], uint &count)

FrameInputs 函数允许你获取 EA 交易 *input* 参数的说明和值，形成具有指定传递编号的测试轮次。*parameters* 字符串数组用类似 “ParameterName=ValueParameterN” 的行填充。*count* 参数用 *parameters* 数组中的元素数填充。

这四个函数只允许在 *OnTesterPass* 和 *OnTesterDeinit* 处理程序内部进行调用。

帧可以成批到达终端，在这种情况下，其需要时间传送。因此，没有必要让它们都有时间生成 *OnTesterPass* 事件，并一直处理到优化结束。在这方面，为了保证接收到所有迟到的帧，有必要在 *OnTesterDeinit* 中放置一个代码块对其进行处理（使用 *FrameNext* 函数）。

考虑一个简单的示例 *FrameTransfer.mq5*。

EA 交易有四个测试参数。除了最后一个字符串，所有这些参数都可以包含在优化中。

```
input bool Parameter0;
input long Parameter1;
input double Parameter2;
input string Parameter3;
```

但是，为了简化该示例，参数 *Parameter1* 和 *Parameter2* 的步骤数被限制为 10 步（对于每个参数）。因此，如果不使用 *Parameter0*，最大传递次数为 121 次。*Parameter3* 是一个不能包含在优化中的参数示例。

EA 交易不进行交易，而是生成模拟任意应用数据的随机数据。切勿在你的工作项目中像这样使用随机化：其仅适用于演示。

```
ulong startup; // track the time of one run (just like demo data)

int OnInit()
{
    startup = GetMicrosecondCount();
    MathSrand((int)startup);
    return INIT_SUCCEEDED;
}
```

数据以两种类型的帧发送：文件和数组。每种类型都有自己的标识符。

```

#define MY_FILE_ID 100
#define MY_TIME_ID 101

double OnTester()
{
    // send file in one frame
    const static string filename = "binfile";
    int h = FileOpen(filename, FILE_WRITE | FILE_BIN | FILE_ANSI);
    FileWriteString(h, StringFormat("Random: %d", MathRand()));
    FileClose(h);
    FrameAdd(filename, MY_FILE_ID, MathRand(), filename);

    // send array in another frame
    ulong dummy[1];
    dummy[0] = GetMicrosecondCount() - startup;
    FrameAdd("timing", MY_TIME_ID, 0, dummy);

    return (Parameter2 + 1) * (Parameter1 + 2);
}

```

该文件以二进制格式编写，包含简单的字符串。*OnTester* 的结果（准则）是一个简单的算术表达式，包含 *Parameter1* 和 *Parameter2*。

在接收端，在终端图表上以服务模式运行的 EA 交易实例中，我们可从所有带文件的帧中收集数据，并将它们放入一个公共 CSV 文件中。该文件可在 *OnTesterInit* 处理程序中打开。

```

int handle; // file for collecting applied results
void OnTesterInit()
{
    handle = FileOpen("output.csv", FILE_WRITE | FILE_CSV | FILE_ANSI, ",");
}

```

如前所述，不一定所有帧都有时间进入 *OnTesterPass* 处理程序，需要在 *OnTesterDeinit* 中进行额外检查。因此，我们实现了一个辅助函数 *ProcessFileFrames*，我们将通过 *OnTesterPass* 和 *OnTesterDeinit* 调用该函数。

在 *ProcessFileFrames* 中，我们保存了已处理帧的内部计数器 *framecount*。以此为例，我们将确保帧的到达顺序和测试轮次编号通常不会匹配。

```

void ProcessFileFrames()
{
    static ulong framecount = 0;
    ...
}

```

为了在函数中接收帧，说明了根据原型 *FrameNext* 所需的变量。此处，接收数据数组描述为 *uchar*。如果我们要写一些结构体到我们的二进制文件中，我们可以把它们直接放入一个相同类型的结构体数组中。

```

ulong   pass;
string  name;
long    id;
double  value;
uchar   data[];
...

```

下面介绍了几个变量，用于获得帧所属的当前测试轮次的 EA 交易输入。

```

string  params[];
uint    count;
...

```

然后我们用 *FrameNext* 循环读取帧。请注意，几个帧可以同时进入该处理程序，所以需要有一个循环。对于每个帧，我们向终端日志输出轮次编号、帧名和生成的 *double* 值。我们可跳过 ID 不同于 MY_FILE_ID 的帧，稍后再处理它们。

```

ResetLastError();

while(FrameNext(pass, name, id, value, data))
{
    PrintFormat("Pass: %lld Frame: %s Value:%f", pass, name, value);
    if(id != MY_FILE_ID) continue;
    ...
}

if(_LastError != 4000 && _LastError != 0)
{
    Print("Error: ", E2S(_LastError));
}
}

```

对于具有 MY_FILE_ID 的帧，我们执行以下操作：查询输入变量，找出哪些变量包含在优化中，并将它们的值与来自帧的信息一起保存到公共 CSV 文件中。当帧数为 0 时，我们可在 *header* 变量中形成 CSV 文件的标题。在所有帧中，CSV 文件的当前（新）记录是在 *record* 变量中形成的。

```

void ProcessFileFrames()
{
    ...
    if(FrameInputs(pass, params, count))
    {
        string header, record;
        if(framecount == 0) // prepare CSV header
        {
            header = "Counter,Pass ID,";
        }
        record = (string)framecount + "," + (string)pass + ",";
        // collect optimized parameters and their values
        for(uint i = 0; i < count; i++)
        {
            string name2value[];
            int n = StringSplit(params[i], '=', name2value);
            if(n == 2)
            {
                long pvalue, pstart, pstep, pstop;
                bool enabled = false;
                if(ParameterGetRange(name2value[0],
                    enabled, pvalue, pstart, pstep, pstop))
                {
                    if(enabled)
                    {
                        if(framecount == 0) // prepare CSV header
                        {
                            header += name2value[0] + ",";
                        }
                        record += name2value[1] + ","; // data field
                    }
                }
            }
        }
        if(framecount == 0) // prepare CSV header
        {
            FileWriteString(handle, header + "Value,File Content\n");
        }
        // write data to CSV
        FileWriteString(handle, record + DoubleToString(value) + ","
            + CharArrayToString(data) + "\n");
    }
    framecount++;
    ...
}

```

调用 *ParameterGetRange* 也可以更有效地完成，只是 *framecount* 的值为零。你可以尝试这样做。

在 *OnTesterPass* 处理程序中，我们只调用 *ProcessFileFrames*。

```

void OnTesterPass()
{
    ProcessFileFrames(); // standard processing of frames on the go
}

```

此外，我们从 *OnTesterDeinit* 调用相同的函数并关闭 CSV 文件。

```

void OnTesterDeinit()
{
    ProcessFileFrames(); // pick up late frames
    FileClose(handle);   // close the CSV file
    ..
}

```

在 *OnTesterDeinit* 中，我们用 *MY_TIME_ID* 处理帧。测试轮次的持续时间在这些帧中提供，并且一个轮次的平均持续时间在此处计算。理论上，只在程序中进行分析时这样做是有意义的，因为对于用户而言，测试程序已经在日志中显示了测试轮次的持续时间。

```

void OnTesterDeinit()
{
    ...
    ulong   pass;
    string   name;
    long     id;
    double   value;
    ulong    data[]; // same array type as sent

    FrameFilter("timing", MY_TIME_ID); // rewind to the first frame

    ulong count = 0;
    ulong total = 0;
    // cycle through 'timing' frames only
    while(FrameNext(pass, name, id, value, data))
    {
        if(ArraySize(data) == 1)
        {
            total += data[0];
        }
        else
        {
            total += (ulong)value;
        }
        ++count;
    }
    if(count > 0)
    {
        PrintFormat("Average timing: %lld", total / count);
    }
}

```

EA 交易准备就绪。我们为其启用完全优化（因为选项的总数是人为限制的，对于遗传算法而言太小了）。我们只能选择开盘价，因为 EA 交易并不执行交易。因此，你应选择一个自定义准则

所有其他准则将给出 0)。例如，我们以单步长从 1 到 10 设置范围 *Parameter1*，以 0.1 步长从 -0.5 到 +0.5 设置 *Parameter2*。

我们运行优化。在终端的专家日志中，我们将看到关于接收帧的条目，格式如下：

```
Pass: 0 Frame: binfile Value:5105.000000
Pass: 0 Frame: timing Value:0.000000
Pass: 1 Frame: binfile Value:28170.000000
Pass: 1 Frame: timing Value:0.000000
Pass: 2 Frame: binfile Value:17422.000000
Pass: 2 Frame: timing Value:0.000000
...
Average timing: 1811
```

带有轮次编号、参数值和帧内容的相应行将出现在 output.csv 文件中：

```
Counter,Pass ID,Parameter1,Parameter2,Value,File Content
0,0,0,-0.5,5105.00000000,Random: 87
1,1,1,-0.5,28170.00000000,Random: 64
2,2,2,-0.5,17422.00000000,Random: 61
...
37,35,2,-0.2,6151.00000000,Random: 68
38,62,7,0.0,17422.00000000,Random: 61
39,36,3,-0.2,16899.00000000,Random: 71
40,63,8,0.0,17422.00000000,Random: 61
...
117,116,6,0.5,27648.00000000,Random: 74
118,117,7,0.5,16899.00000000,Random: 71
119,118,8,0.5,17422.00000000,Random: 61
120,119,9,0.5,28170.00000000,Random: 64
```

显然，我们的内部编号 (*Count* 列)是按顺序进行的，轮次编号 *Pass ID* 可以混合 (具体取决于代理并行处理批任务的许多因素)。特别是，该批任务可以是第一次分配完成具有较高序号任务的代理的任务：在这种情况下，文件中的编号将从较高的轮次开始。

在测试程序的日志中，你可以按帧检查服务的统计数据。

```
242 frames (42.78 Kb total, 181 bytes per frame) received
local 121 tasks (100%), remote 0 tasks (0%), cloud 0 tasks (0%)
121 new records saved to cache file 'tester\cache\Frametransfer.EURUSD.H1. »
» 20220101.20220201.20.9E2DE099D4744A064644F6BB39711DE8.opt'
```

需要注意的是，在遗传优化过程中，运行编号在优化报告中是成对 (*generation number, copy number*) 出现的，而在 *FrameNext* 函数中获得的传递编号为 *ulong*。事实上，其为当前优化运行环境中批次作业的轮次编号。MQL5 没有提供将轮次编号与遗传报告相匹配的方法。为此，应计算每个轮次输入参数的校验和。具有优化缓存的 *Opt* 文件已经包含这样一个具有 MD5 散列的字段。

6.5.12 测试程序的预处理指令

在 *程序的通用特性* 一节中，我们首先熟悉了 MQL 程序中的 *#property* 指令。然后，我们认识了用于 *脚本*、*服务* 和 *指标* 的指令。还有一组用于测试程序的指令。我们已经提到了其中一些。例如，*tester_everytick_calculate* 会影响指标的计算。

下表列出了所有测试程序指令及其解释。

指令	说明
<code>tester_indicator "string"</code>	自定义指标名称，格式为 <code>"indicator_name.ex5"</code>
<code>tester_file "string"</code>	文件名，格式为 <code>"file_name.extension"</code> ，包含程序测试所需的初始数据
<code>tester_library "string"</code>	库名，扩展名为 <code>"library.ex5"</code> 或 <code>"library.dll"</code>
<code>tester_set "string"</code>	文件名，格式为 <code>"file_name.set"</code> ，带有程序输入参数优化值及其范围的设置
<code>tester_no_cache</code>	禁止读取先前优化的现有缓存（ <code>opt</code> 文件）
<code>tester_everytick_calculate</code>	禁用测试程序中计算指标的资源节约模式

最后两个指令没有自变量。所有其他类型都需要一个带双引号的字符串，其中包含某种类型的文件名。由此还可以得出结论，这些指令可以在不同的文件中重复，也就是说，可以包括几个设置文件或几个指标。

tester_indicator 指令用于以常量字符串（文字）的形式将测试程序的源代码中没有提到的那些指标连接到测试过程。通常，如果在相应的参数中明确指定了指标名称，则编译器可以从 *iCustom* 调用中自动确定所需的指标，例如，*iCustom(symbol, period, "indicator_name",...)*。但是，情况并非总是如此。

假设我们正在编写一个通用的 EA 交易，其可以使用不同的移动平均线指标，而不仅仅是标准的内置指标。然后，我们可以创建一个输入变量，由用户指定指标名称。然后，*iCustom* 调用会变为 *iCustom(symbol, period, CustomIndicatorName,...)*，其中 *CustomIndicatorName* 为 EA 交易的输入变量，其内容在编译时是未知的。此外，在这种情况下，开发人员可能会应用 *IndicatorCreate* 而不是 *iCustom*，因为还必须配置指标参数的数量和类型。在这种情况下，为了调试程序或者用特定的指标演示程序，我们应使用 **tester_indicator** 指令向测试程序提供名称。

在源代码中报告指标名称的需要极大地限制了测试这种可以在线连接各种指标的通用程序的能力。

如果没有 **tester_indicator** 指令，终端将无法向源代码中未明确声明的代理发送指标，其结果是相关程序将失去部分或全部函数。

tester_file 指令允许你指定一个文件，该文件将在测试之前被传输到代理并放置在沙箱中。文件的内容和类型没有规定。例如，可以是预先训练的神经网络权重、预先收集的市场深度数据（因为此类数据不能被测试程序再现）等等。

注意，**tester_file** 指令中的文件只有在编译时存在时才会被读取。如果源代码是在没有对应文件的情况下编译的，那么其在未来的出现将不再有帮助：编译后的程序将在没有辅助文件的情况下发送给代理。因此，例如，如果 **tester_file** 中指定的文件是在 *OnTesterInit* 中生成的，应确保具有给定名称的文件在编译时已经存在，即使它是空的。我们将在下面进行演示。

请注意，如果 **tester_file** 指令中指定的文件不存在，编译器不会生成警告。

连接的文件必须在终端的沙箱 *MQL5/Files/* 中。

tester_library 指令用于通知测试程序需要将库转移到代理，库是一个辅助程序，只能在另一个 MQL 程序的上下文中工作。我们将在单独的 章节中详细介绍库。

测试所需的库由源代码中的 **#import** 指令自动确定。但是，如果外部指标使用任何库，则必须启用该特性。该库既可以带有 **dll** 扩展名，也可以带有 **ex5** 扩展名。

tester_set 指令用于操作带有 MQL 程序设置的 **set** 文件。该指令中指定的文件将从测试程序的上下文菜单中提供，并允许用户快速应用设置。

如果指定名称时没有包含路径，则 **set** 文件必须与 EA 交易位于同一目录中。这有点出乎意料，因为 **set** 文件的默认目录是 **Presets**，这是终端接口命令保存它们的位置。要从给定目录连接 **set** 文件，必须在指令中显式指定，并在其前面加一个斜杠，该斜杠指示 MQL5 文件夹内的绝对路径。

```
#property tester_set "/Presets/xyz.set"
```

当没有前斜杠时，该路径相对于放置源文本的位置。

在添加文件并重新编译程序后，需要立即在测试程序中重新选择 EA 交易；否则文件不会被选取！

如果你将 EA 交易名称和版本号指定为类型为 **set** 文件的 **<expert_name>_<number>.set**，那么它会自动添加到版本号 **<number>** 下的参数版本下载菜单中。例如，名称 **"MACD Sample_4.set"** 表示它是 EA 交易 **"MACD Sample.mq5"** 的 **set** 文件，版本号为 4。

如果感兴趣，可以研究 **set** 文件的格式：为此，可在策略测试程序中手动保存测试/优化设置，然后在文本编辑器中打开以这种方式创建的文件。

现在我们来查看 **tester_no_cache** 指令。执行优化时，策略测试程序会将所有执行过的测试轮次结果保存到优化缓存（扩展名为 **opt** 的文件）中，其中存储了每组输入参数的测试结果。这使得在对相同参数进行重新优化时，无需重新计算和浪费时间即可获得现成的结果。

但是，对于某些任务，如数学计算，无论优化缓存中是否存在现成的结果，都可能需要执行计算。在这种情况下，源代码必须包括特性 **tester_no_cache**。同时，测试结果本身仍将存储在缓存中，这样就可以在策略测试程序中看到所有已完成传递的数据。

tester_everytick_calculate 指令用于在测试程序的每条分时报价上启用指标计算模式。

默认情况下，仅在访问指标以获取数据时，即当请求指标缓冲区的值时，才会在测试程序中计算指标。如果你不需要在每个分时报价上获取指标值，则会大大加快测试和优化的速度。

但是，一些程序可能需要在每条分时报价上重新计算指标。在这种情况下，特性 **tester_everytick_calculate** 非常有用。

在以下情况下，策略测试程序中的指标也必须在每条分时报价上进行计算：

- 在可视化模式下测试时
- 如果指标中有 **EventChartCustom**、**OnChartEvent** 或 **OnTimer** 函数

此特性仅适用于策略测试程序中的操作。在终端中，指标总是在每个传入的分时报价上计算。

该指令实际上已经在 **FrameTransfer.mq5** EA 交易

```
#property tester_set "FrameTransfer.set"
```

中使用过，只是我们没有关注。**"FrameTransfer.set"** 文件位于源代码旁边。在同一个 EA 交易中，我们还需要上表中的另一个指令：


```
#property tester_no_cache
```

此外，我们考虑 *tester_file* 指令的一个示例。在前面关于优化时 *自动调优 EA 交易参数* 一节中，我们介绍了 *BandOsMApro.mq5*，其中需要引入几个影子参数来将优化范围传递给在代理上运行的源代码。

tester_file 指令可允许我们省略这些多余的参数。我们将新版本命名为 *BandOsMAprofile.mq5*。

既然我们现在已经熟悉了 *tester_set* 指令，那么我们将前面提到的文件 */Presets/MQL5Book/BandOsMA.set* 添加到新版本中。

```
#property tester_set "/Presets/MQL5Book/BandOsMA.set"
```

关于 *FastOsMA* 和 *SlowOsMA* 的变更周期的范围和步长信息将被保存到文件 *BandOsMAprofile.csv* 中，而非保存到三个附加的输入参数 *FastShadow4Optimization*、*SlowShadow4Optimization* 和 *StepsShadow4Optimization* 中。

```
#define SETTINGS_FILE "BandOsMAprofile.csv"
#property tester_file SETTINGS_FILE
```

```
const string SettingsFile = SETTINGS_FILE;
```

对于允许的周期组合的完整枚举，仍然需要影子设置 *FastSlowCombo4Optimization*。

```
input group "A U X I L I A R Y"
sinput int FastSlowCombo4Optimization = 0; // (reserved for optimization)
```

注意，我们在 *Iterate* 函数中找到了其优化范围。我们第一次使用快慢周期组合的完整枚举在 *OnTesterInit* 中调用该参数。

基本上，我们可以将所有有效的组合存储在 *PairOfPeriods* 结构体数组中，并将其写入二进制文件，以便传输给代理。然后，在代理上，我们的 EA 交易可以从该文件中读取就需的数组，并通过 *FastSlowCombo4Optimization* 索引从数组中提取相应的 *FastOsMA* 和 *SlowOsMA* 对。

相反，我们将关注程序工作逻辑中的最小变化：继续恢复由于 *OnInit* 处理程序中的第二次调用 *Iterate* 而导致的几个周期。这一次，我们不是通过影子参数，而是通过 CSV 文件，获取周期值的枚举范围和步长。

以下是对 *OnTesterInit* 的更改。

```

int OnTesterInit()
{
    ...
    // check if the file already exists before compiling
    // - if not, the tester will not be able to send it to agents
    const bool preExisted = FileIsExist(SettingsFile);

    // write the settings to a file for transfer to copy programs on agents
    int handle = FileOpen(SettingsFile, FILE_WRITE | FILE_CSV | FILE_ANSI, ",");
    FileWrite(handle, "FastOsMA", start1, step1, stop1);
    FileWrite(handle, "SlowOsMA", start2, step2, stop2);
    FileClose(handle);

    if(!preExisted)
    {
        PrintFormat("Required file %s is missing. It has been just created."
            " Please restart again.",
            SettingsFile);
        ChartClose();
        return INIT_FAILED;
    }
    ...
    return INIT_SUCCEEDED;
}

```

注意，我们使用返回类型 `int` 创建了 `OnTesterInit` 处理程序，如果该文件不存在，就可以取消优化。但是，在任何情况下，实际数据都会被写入文件，所以如果它不存在，现在就应创建，并且随后的优化启动就肯定会成功。

如果想跳过这一步，可以事先创建一个空文件 `MT5/Files/BandOsMAprofile.csv`。

`OnInit` 处理程序更改如下。

```

int OnInit()
{
    if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;

    PairOfPeriods p = {FastOsMA, SlowOsMA}; // default initial parameters
    int handle = FileOpen(SettingsFile, FILE_READ | FILE_TXT | FILE_ANSI);

    // during optimization, a file with shadow parameters is needed
    if(MQLInfoInteger(MQL_OPTIMIZATION) && handle == INVALID_HANDLE)
    {
        return INIT_PARAMETERS_INCORRECT;
    }

    if(handle != INVALID_HANDLE)
    {
        if(FastSlowCombo4Optimization != -1)
        {
            // if there is a shadow copy, read the period values from it
            const string line1 = FileReadString(handle);
            string settings[];
            if(StringSplit(line1, ',', settings) == 4)
            {
                int FastStart = (int)StringToInteger(settings[1]);
                int FastStep = (int)StringToInteger(settings[2]);
                int FastStop = (int)StringToInteger(settings[3]);
                const string line2 = FileReadString(handle);
                if(StringSplit(line2, ',', settings) == 4)
                {
                    int SlowStart = (int)StringToInteger(settings[1]);
                    int SlowStep = (int)StringToInteger(settings[2]);
                    int SlowStop = (int)StringToInteger(settings[3]);
                    p = Iterate(FastStart, FastStop, FastStep,
                        SlowStart, SlowStop, SlowStep, FastSlowCombo4Optimization);
                    PrintFormat("MA periods are restored from shadow: FastOsMA=%d SlowOsMA=%d",
                        p.fast, p.slow);
                }
            }
        }
        FileClose(handle);
    }
}

```

在优化后运行单次测试时，我们将在日志中看到基于优化值 *FastSlowCombo4Optimization* 解码的 *FastOsMA* 和 *SlowOsMA* 周期值。将来，我们可以在周期参数中替换这些值，并删除 csv 文件。我们还规定，如果 *FastSlowCombo4Optimization* 设置为 -1，则不会考虑该文件。

6.5.13 管理指标的可见性: TesterHideIndicators

默认情况下，可见性测试图表显示了正在测试的 EA 交易中创建的所有指标。此外，这些指标也会显示在图表上，图表会在测试结束时自动打开。所有这些仅适用于那些直接在代码中创建的指标：可以在主指标计算中使用的嵌套指标此处并不适用。

从开发人员的角度来看，指标的可见性并不总是令人满意的，他们可能希望隐藏 EA 交易的实现细节。在这种情况下，函数 **TesterHideIndicators** 可禁用图表上已用指标的显示。

void TesterHideIndicators(bool hide)

布尔参数 **hide** 可指示隐藏 (值 **true**) 或显示 (值 **false**) 指标。设置的状态可被 MQL 程序执行环境记住，直到通过用相反的参数值再次调用函数才会改变。此设置的当前状态会影响所有新创建的指标。

换言之，在创建相应指标的说明符之前，应调用具有所需标志值 **hide** 的函数 **TesterHideIndicators**。特别是，在使用 **true** 参数调用该函数后，新的指标将被标记为隐藏标志，并且不会在可见性测试期间和图表上显示，图表将在测试完成时自动打开。

要禁用隐藏新创建指标的模式，请使用 **false** 调用 **TesterHideIndicators**。

该函数仅适用于测试程序。

如果在 **/MQL5/Profiles/Templates** 文件夹中为测试程序或 EA 交易创建了特殊的 **tpl** 模板，那么这个函数就有一些与其性能相关的细节。

如果文件夹 **<expert_name>.tpl** 中有特殊模板，则在可见性测试期间和测试图表上，将仅显示来自该模板的指标。在这种情况下，即使在 EA 交易代码 **TesterHideIndicators** 中调用的函数为 **false**，也不会显示测试的 EA 交易中使用的任何指标。

如果 **tester.tpl** 文件夹中有模板，则在可见性测试期间和测试图表上，将显示 **tester.tpl** 模板的指标以及 EA 交易中未被 **TesterHideIndicators** 调用禁止的指标。**TesterHideIndicators** 函数不会影响模板中的指标。

如果没有 **tester.tpl** 模板，但是有 **default.tpl** 模板，那么该模板的指标就会按照类似的原理进行处理。

稍后，我们将在 [大型 EA 交易示例](#) 中演示该函数是如何工作的。

6.5.14 模拟存取款

MetaTrader 5 测试程序允许你模拟存取款操作。你可以体验一些资金管理系统的操作。

bool TesterDeposit(double money)

TesterDeposit 函数用于在测试资金参数中的存款金额过程中补充账户。金额以测试存款货币表示。

bool TesterWithdrawal(double money)

TesterWithdrawal 函数可使提款等于 **money**。

两个函数均返回 **true** 来表示成功。

例如，我们考虑一个基于“利差交易”策略的 EA 交易。对于该交易，我们需要在其中一个交易方向上选择一个有较大正掉期的符号，例如，买入 AUDUSD。EA 交易会在指定方向开立一个或多个仓位。无盈利仓位将被持有，以便积累掉期交易。当每手交易达到预定利润时，盈利仓位将被平仓。赚取的掉期可从账户中提取。**CrazyCarryTrade.mq5** 文件中提供了源代码。

在输入参数中，用户可以选择交易方向、交易规模 (默认为 0，表示最小手数) 以及每手的最小利润，在该点将平仓止盈的仓位。

```
enum ENUM_ORDER_TYPE_MARKET
{
    MARKET_BUY = ORDER_TYPE_BUY,
    MARKET_SELL = ORDER_TYPE_SELL
};

input ENUM_ORDER_TYPE_MARKET Type;
input double Volume;
input double MinProfitPerLot = 1000;
```

首先，我们在 *OnInit* 处理程序中测试 *TesterWithdrawal* 和 *TesterDeposit* 函数的性能。具体来说，试图提取双倍余额会导致错误 10019。

```
int OnInit()
{
    PRTF(TesterWithdrawal(AccountInfoDouble(ACCOUNT_BALANCE) * 2));
    /*
    not enough money for 20 000.00 withdrawal (free margin: 10 000.00)
    TesterWithdrawal(AccountInfoDouble(ACCOUNT_BALANCE)*2)=false / MQL_ERROR::10019(10
    */
    ...
}
```

但是随后的 100 单位账户货币的提款和贷记将会成功。

```
PRTF(TesterWithdrawal(100));
/*
deal #2 balance -100.00 [withdrawal] done
TesterWithdrawal(100)=true / ok
*/
PRTF(TesterDeposit(100)); // return the money
/*
deal #3 balance 100.00 [deposit] done
TesterDeposit(100)=true / ok
*/
return INIT_SUCCEEDED;
}
```

在 *OnTick* 处理程序中，我们使用 *PositionFilter* 检查仓位的可用性，并用其当前的利润/损失和累计掉期来填充 *values* 数组。

```
void OnTick()
{
    const double volume = Volume == 0 ?
        SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;
    ENUM_POSITION_PROPERTY_DOUBLE props[] = {POSITION_PROFIT, POSITION_SWAP};
    double values[][2];
    ulong tickets[];
    PositionFilter pf;
    pf.select(props, tickets, values, true);
    ...
}
```

没有仓位时，我们可在预先定义的方向打开一个仓位。

```

if(ArraySize(tickets) == 0) // no positions
{
    MqlTradeRequestSync request1;
    (Type == MARKET_BUY ? request1.buy(volume) : request1.sell(volume));
}
else
{
    ... // there are positions - see the next box
}

```

有仓位时，我们在一个周期内进行处理，并平仓那些有足够利润的仓位（根据掉期调整）。如此操作的同时，我们也可以总结平仓和总损失的掉期。由于掉期与时间成比例增长，我们可用其作为平仓“旧”仓位的放大因子。因此，有可能以亏损平仓。

```

double loss = 0, swaps = 0;
for(int i = 0; i < ArraySize(tickets); ++i)
{
    if(values[i][0] + values[i][1] * values[i][1] >= MinProfitPerLot * volume)
    {
        MqlTradeRequestSync request0;
        if(request0.close(tickets[i]) && request0.completed())
        {
            swaps += values[i][1];
        }
    }
    else
    {
        loss += values[i][0];
    }
}
...

```

如果总损失增加，我们会定期开设额外的仓位，但当有更多的仓位时，我们会减少开设仓位的频率，以便以某种方式控制风险。

```

if(loss / ArraySize(tickets) <= -MinProfitPerLot * volume * sqrt(ArraySize(tickets)))
{
    MqlTradeRequestSync request1;
    (Type == MARKET_BUY ? request1.buy(volume) : request1.sell(volume));
}
...

```

最后，我们从账户中移除掉期。

```

if(swaps >= 0)
{
    TesterWithdrawal(swaps);
}

```

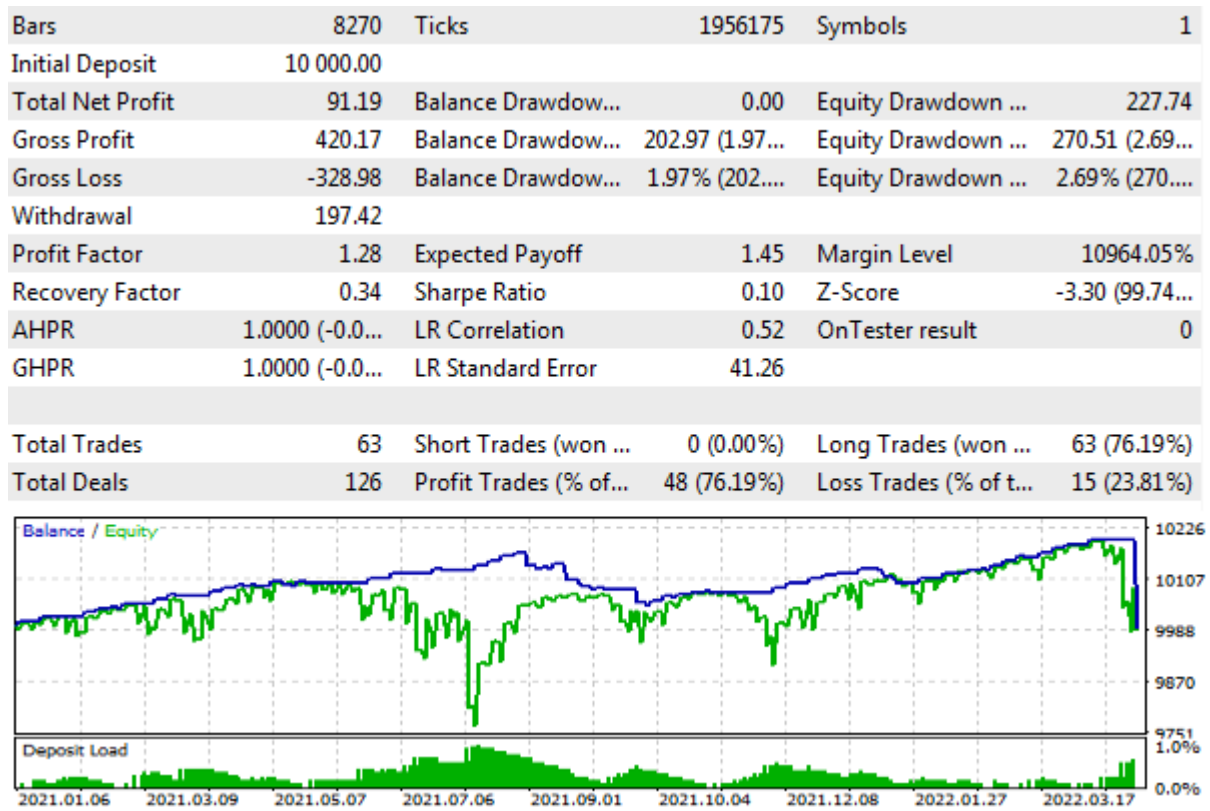
在 *OnDeinit* 处理程序中，显示了关于扣除的统计数据。

```
void OnDeinit(const int)
{
    PrintFormat("Deposit: %.2f Withdrawals: %.2f",
        TesterStatistics(STAT_INITIAL_DEPOSIT),
        TesterStatistics(STAT_WITHDRAWAL));
}
```

例如，在 2021 年至 2022 年初期间使用默认设置运行 EA 交易时，我们会得到以下 AUDUSD 结果：

```
final balance 10091.19 USD
Deposit: 10000.00 Withdrawals: 197.42
```

报告和图表如下所示。



账户提款的 EA 交易报告

因此，在一年多一点的时间里，当交易一个最小手数并存入不超过 1% 的保证金时，我们设法提取了约 200 美元。

6.5.15 强制停止测试：TesterStop

如果有必要，根据观察到的情况，开发人员可以提前停止测试 EA 交易。例如，当达到特定数量的失败交易或提款水平时，可以如此操作。为此，API 提供了 *TesterStop* 函数。

```
void TesterStop()
```

该函数给出一个终止测试程序的命令，即当在程序将控制返回到执行环境后，就会停止。

调用 *TesterStop* 被认为是测试的正常结束，因此将会调用 *OnTester* 函数，并将所有累积的交易统计数据 and 优化准则的值返回给策略测试程序。

还有一种替代的常规方法可中断测试：使用之前探讨过的 `ExpertRemove` 函数。调用 `ExpertRemove` 也会返回调用该函数时收集的交易统计数据。但是，也有一些不同之处。

作为 `ExpertRemove` 调用的结果，EA 交易会从代理的内存中卸载。因此，如果你需要使用一组新参数运行新的轮次，就需要一些时间来重新加载 MQL 程序。当使用 `TesterStop` 时，这种情况不会发生，而且从性能方面而言，这种方法更可取。

另一方面，`ExpertRemove` 调用会在 MQL 程序中设置 `_IsStopped` 停止标志，其可以在程序的不同部分以标准方式用于结束（“清理”资源）。但是，调用 `TesterStop` 不会设置这个标志，因此开发人员可能需要引入他们自己的全局变量来指示提前终止，并以特定的方式对其进行处理。

值得注意的是，`TesterStop` 仅用于停止测试程序的一轮。

MQL5 不提供提前终止优化的函数。因此，例如，如果你的 EA 交易检测到优化是在错误的分时报价生成模型上启动的，并且只能在优化启动后才能检测到 `OnTesterInit`（此处没有作用），那么 `TesterStop` 或 `ExpertRemove` 调用将中断新的轮次，但是轮次本身将继续被启动，生成大量空结果。我们将在 [大型 EA 交易示例](#) 一节中看到该实例，该示例使用保护来防止以开盘价发布。

可以假设在终端中运行并实际服务于优化管理器的 EA 交易实例中的 `ExpertRemove` 调用会停止优化。但事实并非如此。即使在 EA 交易以帧模式工作的情况下关闭图表，也不会停止优化。

建议亲身实践来尝试这些函数。

6.5.16 大型 EA 交易示例

为了总结和巩固关于测试程序能力方面的知识，我们一步一步地考虑一个大型 EA 交易示例。在本示例中，我们将总结以下几个方面：

- 使用多种交易品种，包括柱线同步
- 使用 EA 交易的指标
- 使用事件
- 独立计算主要交易统计数据
- 计算针对可变手数调整的 R2 定制优化标准
- 发送和处理带有应用数据的帧（按交易品种细分的交易报告）

我们将使用 `MultiMartingale.mq5` 作为 EA 交易的技术基础，但将通过切换到交易多币种的超买/超卖信号和作为可选添加项增加手数来降低风险。之前，在 `BandOsMA.mq5` 中，我们已经看到了如何基于指标交易信号进行操作。这次我们将使用 `UseUnityPercentPro.mq5` 作为信号指标。但是，我们需要先修改一下。我们姑且称新版本为 `UnityPercentEvent.mq5`。

UnityPercentEvent.mq5

回想一下 `Unity` 指标的本质。该指标用于计算一组给定金融工具中包含的货币或报价机的相对强度（假设所有金融工具都有一个共同的货币，通过该货币可以进行转换）。在每根柱线上，所有货币的读数均已形成：一些更贵，一些更便宜，这两个极端元素处于一种边界状态。进一步讲，可以为其考虑两种本质上相反的策略：

- 进一步分解（确认并继续向两侧强烈移动）
- 回调（由于超买和超卖而向中心移动的反转）

要交易这些信号中的任何一个,我们必须做一个两种货币的工作交易品种(或交易代码),如果市场报价中有适合这种组合的交易品种。例如,如果指标的上线属于 EUR,下线属于 USD,则二者对应的是 EURUSD 对,根据突破策略,我们应买入;但是根据反弹策略,我们应卖出。

在更一般情况下,例如,当具有共同报价货币的差价合约或大宗商品被列入指标的一篮子工作金融工具时,并不总是能够创造一个真正的金融工具。对于这种情况,必须通过引入交易合成物(复合仓位)使 EA 交易变得更加复杂,但此时我们不会这样做,而是将我们自己限制在外汇市场,在外汇市场上,几乎所有的交叉汇率通常都是可用的。

因此,EA 交易不仅要读取所有的指标缓冲区,还要找出对应于最大值和最小值的货币名称。此处我们有一个小障碍。

MQL5 不允许读取第三方指示符缓冲区的名称,通常也不允许读取除整数以外的任何行特性。有三个函数可设置特性: *PlotIndexSetInteger*、*PlotIndexSetDouble* 和 *PlotIndexSetString*,但是读取特性的函数只有一个: *PlotIndexGetInteger*。

理论上,如果编译成一个交易综合体的 MQL 程序是由一个开发人员创建的,这个问题不大。具体而言,我们可以将指标的一部分源代码分离到一个头文件中,并将其不仅纳入指标中,还纳入 EA 交易中。然后,在 EA 交易中,可以重复分析指标的输入参数,并恢复货币列表,与用指标创建的货币列表完全相似。复制计算虽然不是很优雅,但管用。但是,若指标有不同的开发人员,并且他们不想公开算法或计划在将来改变算法(指标和 EA 交易的编译版本将变得不兼容),也需要更通用的解决方案。这种将别人的指标与自己的指标“对接”或者向别人定制 EA 交易的做法非常普遍。因此,指标开发人员应使指标尽可能可友好集成。

一种可能的解决方案是,指标在初始化后可发送带有缓冲区编号和名称的消息。

这是如何在 *UnityPercentEvent.mq5* 指标的 *OnInit* 处理程序中完成的呢(以下代码以简短形式显示,因为几乎没有什么变化)。

```

int OnInit()
{
    // find the common currency for all pairs
    const string common = InitSymbols();
    ...
    // set up the displayed lines in the currency cycle
    int replaceIndex = -1;
    for(int i = 0; i <= SymbolCount; i++)
    {
        string name;
        // change the order so that the base (common) currency goes under index 0,
        // the rest depends on the order in which the pairs are entered by the user
        if(i == 0)
        {
            name = common;
            if(name != workCurrencies.getKey(i))
            {
                replaceIndex = i;
            }
        }
        else
        {
            if(common == workCurrencies.getKey(i) && replaceIndex > -1)
            {
                name = workCurrencies.getKey(replaceIndex);
            }
            else
            {
                name = workCurrencies.getKey(i);
            }
        }
    }

    // set up rendering of buffers
    PlotIndexSetString(i, PLOT_LABEL, name);
    ...
    // send indexes and buffer names to programs where they are needed
    EventChartCustom(0, (ushort)BarLimit, i, SymbolCount + 1, name);
}
...
}

```

与原始版本相比，此处只增加了一行。该行包含 *EventChartCustom* 调用。*BarLimit* 输入变量用作指标复本（可能是几个）的标识符。由于该指标将从 EA 交易中调用，并且不会显示给用户，所以只需指定一个小的正数，至少为 1，但是也可能是其它数字，例如 10。

现在，指标已经准备好，其信号可以用于第三方 EA 交易。我们开始开发 EA 交易 *UnityMartingale.mq5*。为了简化演示，我们将其分为 4 个阶段，逐步添加新的块。我们将有三个初步版本和一个最终版本。

UnityMartingaleDraft1.mq5

第 1 阶段，对于 *UnityMartingaleDraft1.mq5* 版本，我们先以 *MultiMartingale.mq5* 为基础进行修改。

我们将之前的 *StartType* 输入变量重命名为 *SignalType*，该变量决定了系列中第一笔交易的方向。该变量被用来在 BREAKOUT 和 PULLBACK 之间做出选择。

```
enum SIGNAL_TYPE
{
    BREAKOUT,
    PULLBACK
};
...
input SIGNAL_TYPE StartType = 0; // SignalType
```

要设置指标，我们需要一组单独的输入变量。

```
input group "U N I T Y   S E T T I N G S"
input string UnitySymbols = "EURUSD,GBPUSD,USDCHF,USDJPY,AUDUSD,USDCAD,NZDUSD";
input int UnityBarLimit = 10;
input ENUM_APPLIED_PRICE UnityPriceType = PRICE_CLOSE;
input ENUM_MA_METHOD UnityPriceMethod = MODE_EMA;
input int UnityPricePeriod = 1;
```

请注意，*UnitySymbols* 参数包含用于构建指标的聚类金融工具列表，通常不同于我们想要交易的工作金融工具列表。交易金融工具仍在 *WorkSymbols* 参数中设置。

例如，默认情况下，我们可将一组主要 *Forex* 货币对传递给该指标，因此我们不仅可以指定主要货币对为交易货币，还可以将任何交叉货币对指定为交易货币。一般来说，合理的做法是将这一组货币对限制在交易条件最好的金融工具（特别是小或中等点差）。此外，我们通常希望避免扭曲，即，在所有货币对中保持每种货币的等额，从而在统计上抵消为其中一种货币选择方向不成功的潜在风险。

接下来，我们将指标控件打包在 *UnityController* 类中。除了指标 *handle*，该类字段还存储以下数据：

- *buffers* 指标的数量，其在初始化后从指标的消息中接收
- 正在读取数据的 *bar* 编号（通常，0 表示当前未完成，1 表示上次完成）
- 包含从指定柱线上的指标缓冲区读取的值的 *data* 数组
- 上次读取时间 *lastRead*
- 按分时报价或柱线显示的操作标志 *tickwise*

此外，该类使用 *MultiSymbolMonitor* 对象来同步所有相关交易品种的柱线。

```
class UnityController
{
    int handle;
    int buffers;
    const int bar;
    double data[];
    datetime lastRead;
    const bool tickwise;
    MultiSymbolMonitor sync;
    ...
}
```

在通过自变量接受指标所有参数的构造函数中，我们创建了指标并设置了 *sync* 对象。

```

public:
    UnityController(const string symbolList, const int offset, const int limit,
        const ENUM_APPLIED_PRICE type, const ENUM_MA_METHOD method, const int period):
        bar(offset), tickwise(!offset)
    {
        handle = iCustom(_Symbol, _Period, "MQL5Book/p6/UnityPercentEvent",
            symbolList, limit, type, method, period);
        lastRead = 0;

        string symbols[];
        const int n = StringSplit(symbolList, ',', symbols);
        for(int i = 0; i < n; ++i)
        {
            sync.attach(symbols[i]);
        }
    }

    ~UnityController()
    {
        IndicatorRelease(handle);
    }
    ...

```

缓冲区的数量由 *attached* 方法设置。我们将在收到来自指标的消息时调用该方法。

```

void attached(const int b)
{
    buffers = b;
    ArrayResize(data, buffers);
}

```

当所有交易品种的最后一条柱线具有相同的时间时，一种特殊的方法 *isReady* 返回 *true*。只有在这种同步状态下，我们才能获得指标的正确值。应注意的是，此处假设所有金融工具交易时段的时间表相同。如果情况并非如此，则需要更改时序分析。

```

bool isReady()
{
    return sync.check(true) == 0;
}

```

根据指标操作模式的不同，我们可以不同的方式定义当前时间：当在每条分时报价重新计算时 (*tickwise* 等于 *true*)，我们使用服务器时间，当每个柱线重新计算时，我们使用最后一个柱线的开盘时间。

```

datetime lastTime() const
{
    return tickwise ? TimeTradeServer() : iTime(_Symbol, _Period, 0);
}

```

如果当前时间没有改变，相应地，存储在 *data* 缓冲器中的最后读数据仍然相关，则该方法的存在将允许我们排除读取指标。这就是如何在 *read* 方法中组织对指示符缓冲区的读取。对于具有 *bar* 索引的柱线，我们只需要每个缓冲区的一个值。

```

bool read()
{
    if(!buffers) return false;
    for(int i = 0; i < buffers; ++i)
    {
        double temp[1];
        if(CopyBuffer(handle, i, bar, 1, temp) == 1)
        {
            data[i] = temp[0];
        }
        else
        {
            return false;
        }
    }
    lastRead = lastTime();
    return true;
}

```

最后，我们只需将读取时间保存到 *lastRead* 变量中。如果该变量为空或者不等于新的当前时间，用下面的方法访问控制器数据会导致使用 *read* 来读取指标缓冲区。

控制器的主要外部方法是用于获取最大值和最小值索引的 *getOuterIndices* 和读取这些值的操作符 '[]'。

```

bool isNewTime() const
{
    return lastRead != lastTime();
}

bool getOuterIndices(int &min, int &max)
{
    if(isNewTime())
    {
        if(!read()) return false;
    }
    max = ArrayMaximum(data);
    min = ArrayMinimum(data);
    return true;
}

double operator[](const int buffer)
{
    if(isNewTime())
    {
        if(!read())
        {
            return EMPTY_VALUE;
        }
    }
    return data[buffer];
}
};

```

之前，EA 交易 *BandOsMA.mq5* 引入了 *TradingSignal* 接口的概念。

```

interface TradingSignal
{
    virtual int signal(void);
};

```

在此基础上，我们将使用 *UnityPercentEvent* 指标来说明信号的实现。控制器对象 *UnityController* 被传递给构造函数。其还显示了货币指数（缓冲区），即我们想要跟踪的信号。我们能够为选定的工作交易品种创建任意一组不同的交易品种。

```

class UnitySignal: public TradingSignal
{
    UnityController *controller;
    const int currency1;
    const int currency2;

public:
    UnitySignal(UnityController *parent, const int c1, const int c2):
        controller(parent), currency1(c1), currency2(c2) { }

    virtual int signal(void) override
    {
        if(!controller.isReady()) return 0; // waiting for bars synchronization
        if(!controller.isNewTime()) return 0; // waiting for time to change

        int min, max;
        if(!controller.getOuterIndices(min, max)) return 0;

        // overbought
        if(currency1 == max && currency2 == min) return +1;
        // oversold
        if(currency2 == max && currency1 == min) return -1;
        return 0;
    }
};

```

signal 方法在不确定的情况下返回 0，在两种特定货币的超买和超卖状态下返回 +1 或 -1。

为了最终形成交易策略，我们使用了 *TradingStrategy* 接口。

```

interface TradingStrategy
{
    virtual bool trade(void);
};

```

在这种情况下，*UnityMartingale* 类是在其基础上创建的，这在很大程度上与 *MultiMartingale.mq5* 中的 *SimpleMartingale* 一致。我们将只展示其不同之处。

```

class UnityMartingale: public TradingStrategy
{
protected:
    ...
    AutoPtr<TradingSignal> command;

public:
    UnityMartingale(const Settings &state, TradingSignal *signal)
    {
        ...
        command = signal;
    }
    virtual bool trade() override
    {
        ...
        int s = command[].signal(); // get controller signal
        if(s != 0)
        {
            if(settings.startType == PULLBACK) s *= -1; // reverse logic for bounce
        }
        ulong ticket = 0;
        if(position[] == NULL) // clean start - there were (and is) no positions
        {
            if(s == +1)
            {
                ticket = openBuy(settings.lots);
            }
            else if(s == -1)
            {
                ticket = openSell(settings.lots);
            }
        }
        else
        {
            if(position[].refresh()) // position exists
            {
                if((position[].get(POSITION_TYPE) == POSITION_TYPE_BUY && s == -1)
                || (position[].get(POSITION_TYPE) == POSITION_TYPE_SELL && s == +1))
                {
                    // signal in the other direction - we need to close
                    PrintFormat("Opposite signal: %d for position %d %lld",
                                s, position[].get(POSITION_TYPE), position[].get(POSITION_TICKET));
                    if(close(position[].get(POSITION_TICKET)))
                    {
                        // position = NULL; - save the position in the cache
                    }
                    else
                    {
                        position[].refresh(); // control possible closing errors
                    }
                }
            }
        }
    }
}

```



```

        else
        {
            // the signal is the same or absent - "trailing"
            position[].update();
            if(trailing[]) trailing[].trail();
        }
    }
else // no position - open a new one
{
    if(s == 0) // no signals
    {
        // here is the full logic of the old Expert Advisor:
        // - reversal for martingale loss
        // - continuation by the initial lot in a profitable direction
        ...
    }
else // there is a signal
{
    double lots;
    if(position[].get(POSITION_PROFIT) >= 0.0)
    {
        lots = settings.lots; // initial lot after profit
    }
else // increase the lot after the loss
    {
        lots = MathFloor((position[].get(POSITION_VOLUME) * settings.factor

            if(lotsLimit < lots)
            {
                lots = settings.lots;
            }
        }

        ticket = (s == +1) ? openBuy(lots) : openSell(lots);
    }
}
}
}
...
}

```

交易部分准备就绪。现在开始考虑其初始化。*UnityController* 对象的自动指针和带有货币名称的数组在全局水平进行说明。交易系统的池完全与之前开发完全一致。

```

AutoPtr<TradingStrategyPool> pool;
AutoPtr<UnityController> controller;

int currenciesCount;
string currencies[];

```

在 *OnInit* 处理程序中，我们可创建 *UnityController* 对象，并等待指标通过缓冲区索引发送货币分布。

```

int OnInit()
{
    currenciesCount = 0;
    ArrayResize(currencies, 0);

    if(!Startup(true)) return INIT_PARAMETERS_INCORRECT;

    const bool barwise = UnityPriceType == PRICE_CLOSE && UnityPricePeriod == 1;
    controller = new UnityController(UnitySymbols, barwise,
        UnityBarLimit, UnityPriceType, UnityPriceMethod, UnityPricePeriod);
    // waiting for messages from the indicator on currencies in buffers
    return INIT_SUCCEEDED;
}

```

如果在指标输入参数中选择了价格类型 `PRICE_CLOSE` 和单个周期，则控制器中的计算将对每个柱线执行一次。在所有其他情况下，信号将按分时报价进行更新，但不会超过每秒一次（回想一下控制器中 *lastTime* 方法的实现）。

辅助方法 *Startup* 通常与 EA 交易 *MultiMartingale* 中的旧 *OnInit* 处理程序执行相同的操作。其用设置填充 *Settings* 结构体，检查其正确性并创建交易系统 *TradingStrategyPool* 的池，该池由用于不同交易品种 *WorkSymbols* 的 *UnityMartingale* 类对象组成。但是，由于我们需要等待有关缓冲区中货币分布的信息，现在这个过程分为两个阶段。因此，*Startup* 函数有一个输入参数，表示来自 *OnInit* 以及之后来自 *OnChartEvent* 的调用。

当分析 *Startup* 源代码时，重要的是要记住，当我们只交易一个与当前图表匹配的金融工具时和当指定了一篮子金融工具时，初始化是不同的。当 *WorkSymbols* 为空行时，第一种模式处于活动状态。为特定金融工具优化 EA 交易非常方便。找到几种金融工具的设置后，我们可以在 *WorkSymbols* 中将它们组合起来。

```

bool StartUp(const bool init = false)
{
    if(WorkSymbols == "")
    {
        Settings settings =
        {
            UseTime, HourStart, HourEnd,
            Lots, Factor, Limit,
            StopLoss, TakeProfit,
            StartType, Magic, SkipTimeOnError, Trailing, _Symbol
        };

        if(settings.validate())
        {
            if(init)
            {
                Print("Input settings:");
                settings.print();
            }
        }
        else
        {
            if(init) Print("Wrong settings, please fix");
            return false;
        }
        if(!init)
        {
            ...// creating a trading system based on the indicator
        }
    }
    else
    {
        Print("Parsed settings:");
        Settings settings[];
        if(!Settings::parseAll(WorkSymbols, settings))
        {
            if(init) Print("Settings are incorrect, can't start up");
            return false;
        }
        if(!init)
        {
            ...// creating a trading system based on the indicator
        }
    }
    return true;
}

```

OnInit 中的 *StartUp* 函数可用 *true* 参数调用，这意味着只检查设置的正确性。交易系统对象的创建会被延迟，直到从 *OnChartEvent* 中的指标接收到消息为止。

```

void OnChartEvent(const int id,
    const long &lparam, const double &dparam, const string &sparam)
{
    if(id == CHARTEVENT_CUSTOM + UnityBarLimit)
    {
        PrintFormat("%lld %f '%s'", lparam, dparam, sparam);
        if(lparam == 0) ArrayResize(currencies, 0);
        currenciesCount = (int)MathRound(dparam);
        PUSH(currencies, sparam);
        if(ArraySize(currencies) == currenciesCount)
        {
            if(pool[] == NULL)
            {
                start up(); // indicator readiness confirmation
            }
            else
            {
                Alert("Repeated initialization!");
            }
        }
    }
}

```

此处我们记住了全局变量 *currenciesCount* 中的货币数量，并将它们存储在 *currencies* 数组中，之后我们可用 *false* 参数（默认值，因此省略）调用 *StartUp*。消息会按照其在指标缓冲区中的存在顺序从队列中到达。因此，我们可得到指数和货币名称之间的匹配。

再次调用 *StartUp* 时，会执行一个附加代码：

```

bool StartUp(const bool init = false)
{
    if(WorkSymbols == "") // one current symbol
    {
        ...
        if(!init) // final initialization after OnInit
        {
            controller[].attached(currenciesCount);
            // split _Symbol into 2 currencies from the currencies array []
            int first, second;
            if(!SplitSymbolToCurrencyIndices(_Symbol, first, second))
            {
                PrintFormat("Can't find currencies (%s %s) for %s",
                    (first == -1 ? "base" : ""), (second == -1 ? "profit" : ""), _Symbol);
                return false;
            }
            // create a pool from a single strategy
            pool = new TradingStrategyPool(new UnityMartingale(settings,
                new UnitySignal(controller[], first, second)));
        }
    }
    else // symbol basket
    {
        ...
        if(!init) // final initialization after OnInit
        {
            controller[].attached(currenciesCount);

            const int n = ArraySize(settings);
            pool = new TradingStrategyPool(n);
            for(int i = 0; i < n; i++)
            {
                ...
                // split settings[i].symbol into 2 currencies from currencies[]
                int first, second;
                if(!SplitSymbolToCurrencyIndices(settings[i].symbol, first, second))
                {
                    PrintFormat("Can't find currencies (%s %s) for %s",
                        (first == -1 ? "base" : ""), (second == -1 ? "profit" : ""),
                        settings[i].symbol);
                }
                else
                {
                    // add a strategy to the pool on the next trading symbol
                    pool[].push(new UnityMartingale(settings[i],
                        new UnitySignal(controller[], first, second)));
                }
            }
        }
    }
}

```

辅助函数 *SplitSymbolToCurrencyIndices* 用于选择传递交易品种的基础货币和盈利货币，并在 *currencies* 数组中找到其索引。因此，我们可获得在 *UnitySignal* 对象中生成信号的参考数据。每种货币都有自己的一对货币索引。

```
bool SplitSymbolToCurrencyIndices(const string symbol, int &first, int &second)
{
    const string s1 = SymbolInfoString(symbol, SYMBOL_CURRENCY_BASE);
    const string s2 = SymbolInfoString(symbol, SYMBOL_CURRENCY_PROFIT);
    first = second = -1;
    for(int i = 0; i < ArraySize(currencies); ++i)
    {
        if(currencies[i] == s1) first = i;
        else if(currencies[i] == s2) second = i;
    }

    return first != -1 && second != -1;
}
```

总之，EA 交易准备就绪。

你可以看到，在 EA 交易的最后示例中，我们有策略类和交易信号类。我们特意使其成为通用接口 *TradingStrategy* 和 *TradingSignal* 的子节点，以便随后能够收集兼容但不同的实现，这些实现可以在期货 EA 交易的开发中进行组合。这种统一的具体类通常应被分离到单独的头文件中。在我们的示例中，为了简化逐步修改，我们没有这样做。

但是，所说明的方法为面向对象编程的标准。特别是，正如我们在 [创建 EA 交易草案](#) 一节中所述，MetaTrader 5 附带了一个头文件 *framework*，其中包含 MQL 向导中使用的交易操作、交易品种指标和资金管理的标准类。其他类似的解决方案发布在 [mql5.com](#) 网站的文章和代码库版块中。

你可以使用现成的类结构体作为项目的基础，前提是它们在功能和易用性方面合适。

为了完成该图，我们想在 EA 交易中引入我们自己的基于 R2 的优化标准。为了避免 R2 计算公式中的线性回归与我们策略中包含的可变手数之间发生矛盾，我们计算系数不是用于通常的结余线，而是用于每次交易中按手数规范化的累积增量。

为此，在 *OnTester* 处理程序中，我们选择类型为 DEAL_TYPE_BUY 和 DEAL_TYPE_SELL 以及方向为 OUT 的交易。我们将会请求构成财务结果 (利润/损失) 的所有交易特性，即 DEAL_PROFIT、DEAL_SWAP、DEAL_COMMISSION、DEAL_FEE 及其 DEAL_VOLUME 交易量。

```

#define STAT_PROPS 5 // number of requested deal properties

double OnTester()
{
    HistorySelect(0, LONG_MAX);

    const ENUM_DEAL_PROPERTY_DOUBLE props[STAT_PROPS] =
    {
        DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE, DEAL_VOLUME
    };
    double expenses[][STAT_PROPS];
    ulong tickets[]; // needed because of 'select' method prototype, but useful for debuggi

    DealFilter filter;
    filter.中介绍过。let(DEAL_TYPE, (1<<DEAL_TYPE_BUY) | (1<<DEAL_TYPE_SELL), IS::OR_BITWISE
    中介绍过。let(DEAL_ENTRY, (1<<DEAL_ENTRY_OUT) | (1<<DEAL_ENTRY_INOUT) | (1<<DEAL_ENTR
    IS::OR_BITWISE)
    中介绍过。select(props, tickets, expenses);
    ...

```

接下来，在 *balance* 数组中，我们可累计按交易量规范化的盈利/亏损，并计算其标准 R2。

```

const int n = ArraySize(tickets);
double balance[];
ArrayResize(balance, n + 1);
balance[0] = TesterStatistics(STAT_INITIAL_DEPOSIT);

for(int i = 0; i < n; ++i)
{
    double result = 0;
    for(int j = 0; j < STAT_PROPS - 1; ++j)
    {
        result += expenses[i][j];
    }
    result /= expenses[i][STAT_PROPS - 1]; // normalize by volume
    balance[i + 1] = result + balance[i];
}
const double r2 = RSquaredTest(balance);
return r2 * 100;
}

```

EA 交易的第一版本基本准备就绪。我们没有纳入对使用 *TickModel.mqh* 的分时报价模型的检查。假设在 OHLC M1 模式或更好的模式下生成分时报价时，将测试 EA 交易。当检测到“仅开盘价”模式时，EA 交易将向终端发送一个带有错误状态的特殊帧，并将其从测试程序中卸载。不幸的是，这样只会停止本轮测试，但优化将会继续进行。因此，在终端中运行的 EA 交易的副本会向用户发出“警报”，以使用户手动中断优化。

```

void OnTesterPass()
{
    ulong   pass;
    string   name;
    long     id;
    double   value;
    uchar    data[];
    while(FrameNext(pass, name, id, value, data))
    {
        if(name == "status" && id == 1)
        {
            Alert("Please stop optimization!");
            Alert("Tick model is incorrect: OHLC M1 or better is required");
            // it would be logical if the next call would stop all optimization,
            // but it is not
            ExpertRemove();
        }
    }
}

```

你可以优化任何交易品种的 `SYMBOL SETTINGS` 参数，并对不同的交易品种重复优化。同时，`COMMON SETTINGS` 和 `UNITY SETTINGS` 组应包含相同的设置，因为它们适用于交易系统的所有交易品种和实例。例如，所有优化都必须启用或禁用 *Trailing*。另请注意，单个交易品种的输入变量（即交易品种设置组）仅在 *WorkSymbols* 包含空字符串时有效。所以，在优化阶段，要保持该输入变量为空。

例如，为了分散风险，你可以在完全独立的货币对上持续优化 EA 交易：EURUSD、AUDJPY、GBPCHF、NZDCAD 或其他组合。三个带有私有设置示例的设置文件可连接到源代码。

```

#property tester_set "UnityMartingale-eurusd.set"
#property tester_set "UnityMartingale-gbpchf.set"
#property tester_set "UnityMartingale-audjpy.set"

```

要想一次交易三个交易品种，这些设置应该“打包”到一个公共参数 *WorkSymbols*：

```
EURUSD+0.01*1.6^5(200,200)[17,21];GBPCHF+0.01*1.2^8(600,800)[7,20];AUDJPY+0.01*1.2^8(
```

该设置也包含在一个单独的文件中。

```
#property tester_set "UnityMartingale-combo.set"
```

当前版本的 EA 交易的一个问题是，测试报告将提供所有交易品种的一般统计数据（更准确地说，是所有交易策略的统计数据，因为我们可以池中包含不同的类别），而对我们而言，单独监控和估算系统的每个组成部分很有意义。

要做到这一点，需要学习如何通过类比测试程序的执行结果独立计算交易的主要财务指标。我们将在 EA 交易开发的第 2 阶段处理这个问题。

UnityMartingaleDraft2.mq5

可能经常需要统计数据计算，所以我们将在一个单独的头文件 *TradeReport.mqh* 中实现该计算，因此我们将源代码组织到适当的类中。

我们调用主类 *TradeReport*。许多交易变量取决于结余和自由保证金（净值）曲线。因此，该类包含用于跟踪当前结余和利润的变量，并包含结余历史的不断更新的数组。我们不会存储净值的

历史，因为它可能会在每一条分时报价上发生变化，因此，最好在移动中进行计算。稍后我们会看到为什么要有结余曲线。

```
class TradeReport
{
    double balance;    // current balance
    double floating;    // current floating profit
    double data[];      // full balance curve - prices
    datetime moments[]; // and date/time
    ...
}
```

更改和读取类字段是使用方法完成的，包括构造函数，其中结余由 `ACCOUNT_BALANCE` 特性初始化。

```
TradeReport()
{
    balance = AccountInfoDouble(ACCOUNT_BALANCE);
}

void resetFloatingPL()
{
    floating = 0;
}

void addFloatingPL(const double pl)
{
    floating += pl;
}

void addBalance(const double pl)
{
    balance += pl;
}

double getCurrent() const
{
    return balance + floating;
}
...
}
```

这些方法需要迭代计算净值提取（动态）。对于结余提取的一次性计算，需要使用 `data` 结余数组（我们将在测试结束时进行计算）。

基于曲线的波动（结余或净值无关紧要），应使用相同的算法计算绝对和相对提款。因此，该算法和存储中间状态所需的其内部变量在嵌套结构体 *DrawDown* 中实现。以下代码显示了其主要方法和特性。

```

struct DrawDown
{
    double
        series_start,
        series_min,
        series_dd,
        series_dd_percent,
        series_dd_relative_percent,
        series_dd_relative;
    ...
    void reset();
    void calcDrawdown(const double &data[]);
    void calcDrawdown(const double amount);
    void print() const;
};

```

当我们知道整个数组时，第一个 `calcDrawdown` 方法用于计算提款，可用于结余计算。第二个 `calcDrawdown` 方法用于迭代计算提款：每次调用时，都会告知其序列的下一个值，可用于净值计算。

我们知道，除了提款之外，还有报告的大量标准统计数据，但我们将首先支持其中的一小部分。为此，我们在另一个嵌套结构体 `GenericStats` 中描述相应的字段。这是从 `DrawDown` 继承而来的，因为我们仍然需要在报告中提款。

```

struct GenericStats: public DrawDown
{
    long deals;
    long trades;
    long buy_trades;
    long wins;
    long buy_wins;
    long sell_wins;

    double profits;
    double losses;
    double net;
    double pf;
    double average_trade;
    double recovery;
    double max_profit;
    double max_loss;
    double sharpe;
    ...
};

```

根据变量的名称，很容易猜出其对应于什么标准度量指标。一些度量指标是多余的，因此可省略。例如，给定总交易次数 (`trades`) 和其中的买入交易次数 (`buy_trades`)，我们可以很容易地求出卖出交易次数 (`trades - sell_trades`)。互补的盈利/亏损统计数据也是如此。连续盈利和亏损不进行统计。如果愿意，可以用这些指标来补充我们的报告。

为了与测试程序的一般统计数据保持一致，提供了一个 `fillByTester` 方法，可通过 `TesterStatistics` 函数填充所有字段。我们稍后会用到该方法。

```

void fillByTester()
{
    deals = (long)TesterStatistics(STAT_DEALS);
    trades = (long)TesterStatistics(STAT_TRADES);
    buy_trades = (long)TesterStatistics(STAT_LONG_TRADES);
    wins = (long)TesterStatistics(STAT_PROFIT_TRADES);
    buy_wins = (long)TesterStatistics(STAT_PROFIT_LONGTRADES);
    sell_wins = (long)TesterStatistics(STAT_PROFIT_SHORTTRADES);

    profits = TesterStatistics(STAT_GROSS_PROFIT);
    losses = TesterStatistics(STAT_GROSS_LOSS);
    net = TesterStatistics(STAT_PROFIT);
    pf = TesterStatistics(STAT_PROFIT_FACTOR);
    average_trade = TesterStatistics(STAT_EXPECTED_PAYOFF);
    recovery = TesterStatistics(STAT_RECOVERY_FACTOR);
    sharpe = TesterStatistics(STAT_SHARPE_RATIO);
    max_profit = TesterStatistics(STAT_MAX_PROFITTRADE);
    max_loss = TesterStatistics(STAT_MAX_LOSSTRADE);

    series_start = TesterStatistics(STAT_INITIAL_DEPOSIT);
    series_min = TesterStatistics(STAT_EQUITYMIN);
    series_dd = TesterStatistics(STAT_EQUITY_DD);
    series_dd_percent = TesterStatistics(STAT_EQUITYDD_PERCENT);
    series_dd_relative_percent = TesterStatistics(STAT_EQUITY_DDREL_PERCENT);
    series_dd_relative = TesterStatistics(STAT_EQUITY_DD_RELATIVE);
}
};

```

当然，我们需要为测试程序无法计算的交易系统的那些单独结余和净值实现我们自己的计算。上面介绍了 *calcDrawdown* 方法的原型。在操作期间，它们可用 “series_dd”前缀填充最后一组字段。此外，*TradeReport* 类还包含一个计算夏普比率的方法。作为输入，它需要一系列数字和无风险融资利率。完整的源代码可以在随附的文件中找到。

```
static double calcSharpe(const double &data[], const double riskFreeRate = 0);
```

你可能也猜到了，当调用这个方法时，带有结余的 *TradeReport* 类相关元素数组会在 *data* 参数中传递。填充该数组并为特定指标调用上述方法的过程发生在 *calcStatistics* 方法中（参见下文）。交易的对象筛选器可作为输入 (*filter*)、初始存款 (*start*) 和时间 (*origin*) 传递给该数组。我们假设调用代码会以这样一种方式设置筛选器，即只有我们感兴趣的交易系统的交易才会进行筛选。

该方法返回一个填充的结构体 *GenericStats*，此外，还用结余值和变化的时间引用分别填充 *TradeReport* 对象内的 *data* 和 *moments* 数组。我们将在 EA 交易的最终版本中需要该对象。

```

GenericStats calcStatistics(DealFilter &filter,
    const double start = 0, const datetime origin = 0,
    const double riskFreeRate = 0)
{
    GenericStats stats;
    ArrayResize(data, 0);
    ArrayResize(moments, 0);
    ulong tickets[];
    if(!filter.select(tickets)) return stats;

    balance = start;
    PUSH(data, balance);
    PUSH(moments, origin);

    for(int i = 0; i < ArraySize(tickets); ++i)
    {
        DealMonitor m(tickets[i]);
        if(m.get(DEAL_TYPE) == DEAL_TYPE_BALANCE) //deposit/withdrawal
        {
            balance += m.get(DEAL_PROFIT);
            PUSH(data, balance);
            PUSH(moments, (datetime)m.get(DEAL_TIME));
        }
        else if(m.get(DEAL_TYPE) == DEAL_TYPE_BUY
            || m.get(DEAL_TYPE) == DEAL_TYPE_SELL)
        {
            const double profit = m.get(DEAL_PROFIT) + m.get(DEAL_SWAP)
                + m.get(DEAL_COMMISSION) + m.get(DEAL_FEE);
            balance += profit;

            stats.deals++;
            if(m.get(DEAL_ENTRY) == DEAL_ENTRY_OUT
                || m.get(DEAL_ENTRY) == DEAL_ENTRY_INOUT
                || m.get(DEAL_ENTRY) == DEAL_ENTRY_OUT_BY)
            {
                PUSH(data, balance);
                PUSH(moments, (datetime)m.get(DEAL_TIME));
                stats.trades++; // trades are counted by exit deals
                if(m.get(DEAL_TYPE) == DEAL_TYPE_SELL)
                {
                    stats.buy_trades++; // closing with a deal in the opposite directic
                }
                if(profit >= 0)
                {
                    stats.wins++;
                    if(m.get(DEAL_TYPE) == DEAL_TYPE_BUY)
                    {
                        stats.sell_wins++; // closing with a deal in the opposite direct
                    }
                    else
                    {

```

```

        stats.buy_wins++;
    }
}
else if(!TU::Equal(profit, 0))
{
    PUSH(data, balance); // entry fee (if any)
    PUSH(moments, (datetime)m.get(DEAL_TIME));
}

if(profit >= 0)
{
    stats.profits += profit;
    stats.max_profit = fmax(profit, stats.max_profit);
}
else
{
    stats.losses += profit;
    stats.max_loss = fmin(profit, stats.max_loss);
}
}
}

if(stats.trades > 0)
{
    stats.net = stats.profits + stats.losses;
    stats.pf = -stats.losses > DBL_EPSILON ?
        stats.profits / -stats.losses : MathExp(10000.0); // NaN(+inf)
    stats.average_trade = stats.net / stats.trades;
    stats.sharpe = calcSharpe(data, riskFreeRate);
    stats.calcDrawdown(data); // fill in all fields of the DrawDown substruct
    stats.recovery = stats.series_dd > DBL_EPSILON ?
        stats.net / stats.series_dd : MathExp(10000.0);
}
return stats;
}
};

```

此处可以看到我们是如何调用 *calcSharpe* 和 *calcDrawdown* 来获取 *data* 数组上的相应指标。其余指标可在 *calcStatistics* 内部的循环中直接计算。

TradeReport 类准备就绪，我们可以将 EA 交易的功能扩展到 *UnityMartingaleDraft2.mq5* 版本。

我们向 *UnityMartingale* 类添加新成员。

```

class UnityMartingale: public TradingStrategy
{
protected:
    ...
    TradeReport report;
    TradeReport::DrawDown equity;
    const double deposit;
    const datetime epoch;
    ...

```

我们需要 *report* 对象来调用 *calcStatistics*，其中将包括结余提取。*equity* 对象是独立计算净值提取所必需的。初始结余和日期以及净值提取计算的开始，都在构造函数中进行设置。

```

public:
    UnityMartingale(const Settings &state, TradingSignal *signal):
        symbol(state.symbol), deposit(AccountInfoDouble(ACCOUNT_BALANCE)),
        epoch(TimeCurrent())
    {
        ...
        equity.calcDrawdown(deposit);
        ...
    }

```

随着每次调用 *trade* 方法，按净值提取的计算会持续进行。

```

virtual bool trade() override
{
    ...
    if(MQLInfoInteger(MQL_TESTER))
    {
        if(position[])
        {
            report.resetFloatingPL();
            // after reset, sum all floating profits
            // why we call addFloatingPL for each existing position,
            // but this strategy has a maximum of 1 position at a time
            report.addFloatingPL(position[].get(POSITION_PROFIT)
                + position[].get(POSITION_SWAP));
            // after taking into account all the amounts - update the drawdown
            equity.calcDrawdown(report.getCurrent());
        }
    }
    ...
}

```

正确计算所需要的还不止于此。除了结余之外，我们还应该考虑到浮动盈亏。上面的代码章节只显示了 *addFloatingPL* 调用，但是 *TradeReport* 类也有一种修改结余的方法：*addBalance* 中介绍过。但是，只有当仓位平仓时，结余才会改变。

由于有 OOP 的概念，在我们的情况下平仓一个仓位相当于删除 *PositionState* 类的 *position* 对象。那么为什么不能拦截呢？

PositionState 类没有为此提供任何方法，但是我们可以用特殊的构造函数和析构函数声明一个派生类 *PositionStateWithEquity*。

创建对象时，不仅将仓位标识符传递给构造函数，还要传递一个指向报告对象的指针（信息将发送给改报告对象）

```
class PositionStateWithEquity: public PositionState
{
    TradeReport *report;

public:
    PositionStateWithEquity(const long t, TradeReport *r):
        PositionState(t), report(r) { }
    ...
}
```

在析构函数中，我们可通过平仓 ID 找到所有交易，计算总的财务结果（以及佣金和其他扣除额），然后调用 *addBalance* 来找出关联 *report* 对象。

```
~PositionStateWithEquity()
{
    if(HistorySelectByPosition(get(POSITION_IDENTIFIER)))
    {
        double result = 0;
        DealFilter filter;
        int props[] = {DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE};
        Tuple4<double, double, double, double> overheads[];
        if(filter.select(props, overheads))
        {
            for(int i = 0; i < ArraySize(overheads); ++i)
            {
                result += NormalizeDouble(overheads[i]._1, 2)
                    + NormalizeDouble(overheads[i]._2, 2)
                    + NormalizeDouble(overheads[i]._3, 2)
                    + NormalizeDouble(overheads[i]._4, 2);
            }
        }
        if(CheckPointer(report) != POINTER_INVALID) report.addBalance(result);
    }
};
```

还有一点需要澄清：如何为仓位而不是 *PositionState* 创建 *PositionStateWithEquity* 类对象。为此，只需对 *TradingStrategy* 类中调用 *new* 运算符的几处进行更改即可。

```
position=MQLInfoInteger(MQL_TESTER) ?
new PositionStateWithEquity(tickets[0], &report) : new PositionState(tickets[0]);
```

至此，我们已经实现了数据收集。现在我们需要直接生成报告，也就是调用 *calcStatistics*。此处，我们需要扩展我们的 *TradingStrategy* 接口：添加 *statement* 方法。

```
interface TradingStrategy
{
    virtual bool trade(void);
    virtual bool statement();
};
```

然后，在这个当前的实现中，为了我们的策略，我们应该工作满足其逻辑结论。

```

class UnityMartingale: public TradingStrategy
{
    ...
    virtual bool statement() override
    {
        if(MQLInfoInteger(MQL_TESTER))
        {
            Print("Separate trade report for ", settings.symbol);
            // equity drawdown should already be calculated on the fly
            Print("Equity DD:");
            equity.print();

            // balance drawdown is calculated in the resulting report
            Print("Trade Statistics (with Balance DD):");
            // configure the filter for a specific strategy
            DealFilter filter;
            filter.let(DEAL_SYMBOL, settings.symbol)
                .let(DEAL_MAGIC, settings.magic, IS::EQUAL_OR_ZERO);
            // zero "magic" number is needed for the last exit deal
            // - it is done by the tester itself
            HistorySelect(0, LONG_MAX);
            TradeReport::GenericStats stats =
                report.calcStatistics(filter, deposit, epoch);
            stats.print();
        }
        return false;
    }
    ...
}

```

新方法可直接在日志中打印所有计算出的指标。通过在交易系统 *TradingStrategyPool* 的池中转发相同的方法，我们向 *OnTester* 处理程序请求所有交易品种的单独报告。

```

double OnTester()
{
    ...
    if(pool[] != NULL)
    {
        pool[].statement(); // ask all trading systems to display their results
    }
    ...
}

```

我们检查一下我们报告的正确性。为此，我们在测试程序中运行 EA 交易，一次一个交易品种，并将标准报告与我们的计算进行比较。例如，要建立 *UnityMartingale-eurusd.set*，在 EURUSD H1 交易中，我们将得到 2021 年的此类指标。

Bars	6232	Ticks	1474964	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	18.35	Balance Drawdown...	1.49	Equity Drawdown ...	1.80
Gross Profit	57.97	Balance Drawdown...	5.73 (0.06%)	Equity Drawdown ...	6.23 (0.06%)
Gross Loss	-39.62	Balance Drawdown...	0.06% (5.73)	Equity Drawdown ...	0.06% (6.23)
Profit Factor	1.46	Expected Payoff	0.19	Margin Level	81232.33%
Recovery Factor	2.95	Sharpe Ratio	0.15	Z-Score	1.22 (77.75%)
AHPR	1.0000 (0.00...	LR Correlation	0.95	OnTester result	80.38985394...
GHPR	1.0000 (0.00...	LR Standard Error	2.13		
Total Trades	97	Short Trades (won ...	54 (42.59%)	Long Trades (won ...	43 (44.19%)
Total Deals	194	Profit Trades (% of...	42 (43.30%)	Loss Trades (% of t...	55 (56.70%)
	Largest	profit trade	2.00	loss trade	-2.01
	Average	profit trade	1.38	loss trade	-0.72
	Maximum	consecutive wins (\$)	5 (4.37)	consecutive losses ...	7 (-4.77)
	Maximal	consecutive profit ...	6.00 (3)	consecutive loss (c...	-4.77 (7)
	Average	consecutive wins	2	consecutive losses	2

2021年测试程序报告，EURUSD H1

在日志中，我们的版本显示为两种结构体：带有净值提取的 *DrawDown* 和带有结余提取指标和其他统计数据的 *GenericStats*。

Separate trade report for EURUSD

Equity DD:

```
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10022.48 10017.03 10000.00 9998.20 6.23 0.06 »
» [series_dd_relative_percent] [series_dd_relative]
» 0.06 6.23
```

Trade Statistics (with Balance DD):

```
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10022.40 10017.63 10000.00 9998.51 5.73 0.06 »
» [series_dd_relative_percent] [series_dd_relative] »
» 0.06 5.73 »
» [deals] [trades] [buy_trades] [wins] [buy_wins] [sell_wins] [profits] [losses] [net
» 194 97 43 42 19 23 57.97 -39.62 18.35 1.46 »
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
» 0.19 3.20 2.00 -2.01 0.15
```

很容易验证这些数字是否与测试程序的报告相符。

现在我们一次性开始三个交易品种的同期交易（设置 *UnityMartingale-combo.set*）。

除了 EURUSD 条目，GBPCHF 和 AUDJPY 结构体将会出现在日志中。

```

Separate trade report for GBPCHF
Equity DD:
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10029.50 10000.19 10000.00 9963.65 62.90 0.63 »
» [series_dd_relative_percent] [series_dd_relative]
» 0.63 62.90
Trade Statistics (with Balance DD):
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10023.68 9964.28 10000.00 9964.28 59.40 0.59 »
» [series_dd_relative_percent] [series_dd_relative] »
» 0.59 59.40 »
» [deals] [trades] [buy_trades] [wins] [buy_wins] [sell_wins] [profits] [losses] [net
» 600 300 154 141 63 78 394.53 -389.33 5.20 1.01 »
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
» 0.02 0.09 9.10 -6.73 0.01

Separate trade report for AUDJPY
Equity DD:
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10047.14 10041.53 10000.00 9961.62 48.20 0.48 »
» [series_dd_relative_percent] [series_dd_relative]
» 0.48 48.20
Trade Statistics (with Balance DD):
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10045.21 10042.75 10000.00 9963.62 44.21 0.44 »
» [series_dd_relative_percent] [series_dd_relative] »
» 0.44 44.21 »
» [deals] [trades] [buy_trades] [wins] [buy_wins] [sell_wins] [profits] [losses] [net
» 332 166 91 89 54 35 214.79 -170.20 44.59 1.26 »
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
» 0.27 1.01 7.58 -5.17 0.09

```

在这种情况下，测试程序报告将包含一般化数据，借助我们的类，我们已经收到了以前无法获得的细节。

但是，在日志中查看伪报告不是很方便。此外，我希望至少能看到结余线的图示，因为其表现通常比纯统计数据更能说明系统的适用性。

我们来改进 EA 交易，使其能够生成 HTML 格式可视化报告：毕竟，随着时间的推移，测试程序的报告也可以导出到 HTML，保存，并进行比较。此外，这种报告今后可以在优化过程中以帧的形式传输到终端，用户甚至可以在整个过程完成之前就开始研究特定传递的报告。

这将是示例 *UnityMartingaleDraft3.mq5* 的倒数第二个版本。

UnityMartingaleDraft3.mq5

该交易报告的可视化包括结余线和带有统计指标的表格。我们不会生成类似于测试程序报告的完整报告，而是将我们自己限制在选定的最重要值上。我们的目的是实现一种工作机制，然后可以根据个人需求进行定制。

我们将以 *TradeReportWriter* 类 (*TradeReportWriter.mqh*) 的形式安排算法的依据。该类将能够存储来自不同交易系统的任意报告数量：每个报告都存储在一个单独的对象 *DataHolder* 中，其中包括结余值和时间戳（分别是 *data* 和 *when*）、包含统计信息的 *stats* 结构体以及要显示的行标题、颜色和宽度。

```

class TradeReportWriter
{
protected:
    class DataHolder
    {
    public:
        double data[];           // balance changes
        datetime when[];         // balance timestamps
        string name;             // description
        color clr;               // color
        int width;               // line width
        TradeReport::GenericStats stats; // trading indicators
    };
    ...

```

我们为 *DataHolder* 类对象分配了一个自动指针数组 *curves*。此外，我们需要对金额和期限施加通用限制，以匹配图中所有交易系统的行。这将由 *lower*、*upper*、*start* 和 *stop* 变量提供。

```

    AutoPtr<DataHolder> curves[];
    double lower, upper;
    datetime start, stop;

public:
    TradeReportWriter(): lower(DBL_MAX), upper(-DBL_MAX), start(0), stop(0) { }
    ...

```

addCurve 方法可添加结余线。

```

virtual bool addCurve(double &data[], datetime &when[], const string name,
    const color clr = clrNONE, const int width = 1)
{
    if(ArraySize(data) == 0 || ArraySize(when) == 0) return false;
    if(ArraySize(data) != ArraySize(when)) return false;
    DataHolder *c = new DataHolder();
    if(!ArraySwap(data, c.data) || !ArraySwap(when, c.when))
    {
        delete c;
        return false;
    }

    const double max = c.data[ArrayMaximum(c.data)];
    const double min = c.data[ArrayMinimum(c.data)];

    lower = fmin(min, lower);
    upper = fmax(max, upper);
    if(start == 0) start = c.when[0];
    else if(c.when[0] != 0) start = fmin(c.when[0], start);
    stop = fmax(c.when[ArraySize(c.when) - 1], stop);

    c.name = name;
    c.clr = clr;
    c.width = width;
    ZeroMemory(c.stats); // no statistics by default
    PUSH(curves, c);
    return true;
}

```

`addCurve` 方法的第二个版本不仅可添加结余线, 还在 `GenericStats` 结构体中添加了一组金融变量。

```

virtual bool addCurve(TradeReport::GenericStats &stats,
    double &data[], datetime &when[], const string name,
    const color clr = clrNONE, const int width = 1)
{
    if(addCurve(data, when, name, clr, width))
    {
        curves[ArraySize(curves) - 1][].stats = stats;
        return true;
    }
    return false;
}

```

使报告可视化的最重要类方法是抽象的。

```
virtual void render() = 0;
```

这样便有可能实现多种显示报告的方式, 例如, 既可以记录到不同格式的文件, 也可以直接在图表上绘图。我们现在来关注在 HTML 文件的格式上, 因为这是技术上最先进和最普遍的方法。

新类 *HTMLReportWriter* 有一个构造函数，其参数可指定文件名称以及带有结余曲线的图大小。我们可以众所周知的 SVG 矢量图形格式生成图像本身：在这种情况下这是理想的，因为其为 XML 语言的子集，而 XML 语言就是 HTML 本身。

```
class HTMLReportWriter: public TradeReportWriter
{
    int handle;
    int width, height;

public:
    HTMLReportWriter(const string name, const int w = 600, const int h = 400):
        width(w), height(h)
    {
        handle = FileOpen(name,
            FILE_WRITE | FILE_TXT | FILE_ANSI | FILE_REWRITE);
    }

    ~HTMLReportWriter()
    {
        if(handle != 0) FileClose(handle);
    }

    void close()
    {
        if(handle != 0) FileClose(handle);
        handle = 0;
    }
    ...
}
```

在探讨主要的公共 *render* 方法之前，有必要向读者介绍一种技术，将在本书的最后第 7 部分详细说明。我们谈论的是 *资源*：是指连接到 MQL 程序、用于处理多媒体（声音和图像）的文件和任意数据数组、嵌入编译过的指示符，或者仅仅作为应用程序信息的存储库。我们现在使用后一种选项。

重点是，最好不要完全用 MQL 代码生成 HTML 页面，而是基于一个模板（页面模板），MQL 代码只会在其中插入一些变量的值。这是编程中一项众所周知的技术，允许将算法和程序的外部表示（或其工作结果）分开。因此，我们可以分别试验 HTML 模板和 MQL 代码，在熟悉的环境中使用每个组件。具体而言，MetaEditor 仍然不太适合编辑和查看网页，就像标准浏览器对 MQL5 一无所知一样（虽然这是可以修复的）。

我们将 HTML 报告模板存储在连接到 MQL5 源代码的文本文件中，作为资源。使用特殊指令 *#resource* 建立连接。例如，在 *TradeReportWriter.mqh* 文件中有下面的一行。

```
#resource "TradeReportPage.htm" as string ReportPageTemplate
```

这意味着在源代码旁边应有 *TradeReportPage.htm* 文件，其将作为字符串 *ReportPageTemplate* 在 MQL 代码中提供。根据扩展名，你可以理解为文件是一个网页。下面是这个文件的内容及其缩写（我们的任务不是教读者网页开发，尽管，显然这方面的知识对交易者也很有用）。添加缩进是为了直观地表示 HTML 标记的嵌套结构体；文件中没有缩进。

```

<!DOCTYPE html>
<html>
  <head>
    <title>Trade Report</title>
    <style>
      *{font: 9pt "Segoe UI";}
      .center{width:fit-content;margin:0 auto;}
      ...
    </style>
  </head>
  <body>
    <div class="center">
      <h1>Trade Report</h1>
      ~
    </div>
  </body>
  <script>
    ...
  </script>
</html>

```

模板的基础由开发人员选择。有大量现成的 HTML 模板系统，但是它们提供了很多冗余的特性，因此对于我们的示例而言太复杂了。我们将开发我们自己的概念。

首先，我们注意到大多数网页都有一个开头部分（页眉），一个结尾部分（页脚），有用的信息位于中间。在这个意义上，上述报告草案也不例外。其使用字符“~”来表示有用的内容。相反，MQL 代码将不得不插入一个结余图像和一个指标表格。但是“~”的存在并不是必须的，因为页面可以是一个整体，也就是非常有用的中间部分：毕竟，如果需要，MQL 代码可以将处理一个模板的结果插入到另一个模板中。

为了结束关于 HTML 模板的题外话，我们要注意另外一件事。理论上，一个网页由执行不同函数的标签组成。标准的 HTML 标签会告诉浏览器应显示什么。除此之外，还有级联样式（CSS），用于说明显示方式。最后，网页可以有一个 JavaScript 脚本形式的动态组件，用于交互地控制第一个和第二个网页。

通常，这三个组件独立地进行模板化处理，例如，一个 HTML 模板，严格地说，应只包含 HTML 而不包含 CSS 或 JavaScript。这允许 *“解除绑定”网页的内容、外观和行为*，从而方便开发（建议在 MQL5 中遵循相同的方法！）。

但是，在我们的示例中，我们已经在模板中包含了所有组件。特别是，在上面模板中，我们看到了带有 CSS 样式的标签 <style> 和带有一些 JavaScript 函数的标签 <script>，这些标签都省略了。这样做是为了简化示例，重点是 MQL5 特性而不是网页开发。

将 *ReportPageTemplate* 变量中的网页模板作为资源连接起来，我们就可以编写 *render* 方法了。

```

virtual void render() override
{
    string headerAndFooter[2];
    StringSplit(ReportPageTemplate, '~', headerAndFooter);
    FileWriteString(handle, headerAndFooter[0]);
    renderContent();
    FileWriteString(handle, headerAndFooter[1]);
}
...

```

该方法实际上通过 `~` 字符将页面分成上下两半，按原样显示它们，并在它们之间调用辅助方法 `renderContent`。

我们已经介绍过，报告由一个带有结余曲线的总体图和带有交易系统指标的表格组成，因此实现 `renderContent` 是自然的。

```

private:
void renderContent()
{
    renderSVG();
    renderTables();
}

```

`renderSVG` 中的图像生成基于另一个模板文件 `TradeReportSVG.htm`，该文件绑定到一个字符串变量 `SVGBoxTemplate`：

```
#resource "TradeReportSVG.htm" as string SVGBoxTemplate
```

这个模板的内容是我们此处列出的最后一个。如果愿意，可以自行查看模板其余部分的源代码。

```

<span id="params" style="display:block;width:%WIDTH%px;text-align:center;"></span>
<a id="main" style="display:block;text-align:center;">
    <svg width="%WIDTH%" height="%HEIGHT%" xmlns="http://www.w3.org/2000/svg">
        <style>.legend {font: bold 11px Consolas;}</style>
        <rect x="0" y="0" width="%WIDTH%" height="%HEIGHT%"
            style="fill:none; stroke-width:1; stroke: black;" />
        ~
    </svg>
</a>

```

在 `renderSVG` 方法的代码中，我们将看到一个常用技巧是：将内容分成交易品种 `~` 之前和之后，但此处有点儿新东西。

```

void renderSVG()
{
    string headerAndFooter[2];
    if(StringSplit(SVGBoxTemplate, '~', headerAndFooter) != 2) return;
    StringReplace(headerAndFooter[0], "%WIDTH%", (string)width);
    StringReplace(headerAndFooter[0], "%HEIGHT%", (string)height);
    FileWriteString(handle, headerAndFooter[0]);

    for(int i = 0; i < ArraySize(curves); ++i)
    {
        renderCurve(i, curves[i][].data, curves[i][].when,
            curves[i][].name, curves[i][].clr, curves[i][].width);
    }

    FileWriteString(handle, headerAndFooter[1]);
}

```

在页面顶部的字符串 `headerAndFooter[0]` 中，我们正在寻找特殊形式 `%WIDTH%` 和 `%HEIGHT%` 的子字符串，并用图像所需的宽度和高度替换它们。正是根据这一原则，可以在我们的模板中进行值替换。例如，在此模板中，这些子字符串实际出现在 `rect` 标记中：

```
<rect x="0" y="0" width="%WIDTH%" height="%HEIGHT%" style="fill:none; stroke-width:1;
```

因此，如果定制的报告大小为 600 x 400，该行将转换为以下内容：

```
<rect x="0" y="0" width="600" height="400" style="fill:none; stroke-width:1; stroke:
```

这将在浏览器中显示指定尺寸为 1 像素宽的黑色边框。

绘制特定结余线的标记的生成通过 `renderCurve` 方法处理，我们可将所有必要的数组和其他设置（名称、颜色和厚度）传递给该方法。我们将此方法和其他高度专业化方法（`renderTables` 和 `renderTable`）留给独立研究。

我们回到 `UnityMartingaleDraft3.mq5` EA 交易的主模块。设置结余图的大小，并连接 `TradeReportWriter.mqh`。

```

#define MINIWIDTH 400
#define MINIHEIGHT 200

#include <MQL5Book/TradeReportWriter.mqh>

```

为了将策略与报告生成器“连接”起来，需要修改 `TradingStrategy` 接口中的 `statement` 方法：传递一个指向 `TradeReportWriter` 对象的指针，调用代码可以创建并配置该对象。

```

interface TradingStrategy
{
    virtual bool trade(void);
    virtual bool statement(TradeReportWriter *writer = NULL);
};

```

现在我们在我们的 `UnityMartingale` 策略类中，为该方法的具体实现添加一些行。


```

class UnityMartingale: public TradingStrategy
{
    ...
    TradeReport report;
    ...
    virtual bool statement(TradeReportWriter *writer = NULL) override
    {
        if(MQLInfoInteger(MQL_TESTER))
        {
            ...
            // it's already been done
            DealFilter filter;
            filter.let(DEAL_SYMBOL, settings.symbol)
                .let(DEAL_MAGIC, settings.magic, IS::EQUAL_OR_ZERO);
            HistorySelect(0, LONG_MAX);
            TradeReport::GenericStats stats =
                report.calcStatistics(filter, deposit, epoch);
            ...
            // adding this
            if(CheckPointer(writer) != POINTER_INVALID)
            {
                double data[];           // balance values
                datetime time[];         // balance points time to synchronize curves
                report.getCurve(data, time); // fill in the arrays and transfer to write
                return writer.addCurve(stats, data, time, settings.symbol);
            }
            return true;
        }
        return false;
    }
}

```

这一切都归结于从 *report* 对象 (*TradeReport* 类) 获得一个结余数组和一个带有指标的结构体，并传递给 *TradeReportWriter* 对象，以调用 *addCurve*。

当然，交易策略池可确保将同一个对象 *TradeReportWriter* 转移到所有策略，以生成合并报告。

```

class TradingStrategyPool: public TradingStrategy
{
    ...
    virtual bool statement(TradeReportWriter *writer = NULL) override
    {
        bool result = false;
        for(int i = 0; i < ArraySize(pool); i++)
        {
            result = pool[i].statement(writer) || result;
        }
        return result;
    }
}

```

最后，*OnTester* 处理程序经历了最大的修改。下面几行足以生成交易策略的 HTML 报告。

```

double OnTester()
{
    ...
    const static string tempfile = "temp.html";
    HTMLReportWriter writer(tempfile, MINIWIDTH, MINIHEIGHT);
    if(pool[] != NULL)
    {
        pool[].statement(&writer); // ask strategies to report their results
    }
    writer.render(); // write the received data to a file
    writer.close();
}

```

但是，为了清晰起见和方便用户查看，最好在报告中添加一条总体结余曲线以及一个包含总体指标的表格。仅当在 EA 交易设置中指定了几个交易品种时，其输出才有意义，如若不然，一个策略的报告会与文件中的一般策略一致。

这需要更多一点的代码。

```

double OnTester()
{
    ...
    // had it before
    DealFilter filter;
    // set up the filter and fill in the array of deals based on it tickets
    ...
    const int n = ArraySize(tickets);

    // add this
    const bool singleSymbol = WorkSymbols == "";
    double curve[]; // total balance curve
    datetime stamps[]; // date and time of total balance points

    if(!singleSymbol) // the total balance is displayed only if there are several symbols
    {
        ArrayResize(curve, n + 1);
        ArrayResize(stamps, n + 1);
        curve[0] = TesterStatistics(STAT_INITIAL_DEPOSIT);

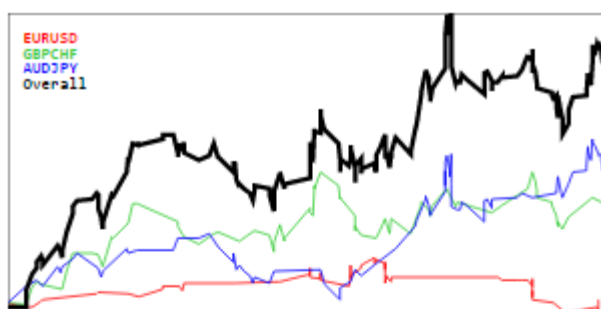
        // MQL5 does not allow to know the test start time,
        // this could be found out from the first transaction,
        // but it is outside the filter conditions of a specific system,
        // so let's just agree to skip time 0 in calculations
        stamps[0] = 0;
    }

    for(int i = 0; i < n; ++i) // deal cycle
    {
        double result = 0;
        for(int j = 0; j < STAT_PROPS - 1; ++j)
        {
            result += expenses[i][j];
        }
        if(!singleSymbol)
        {
            curve[i + 1] = result + curve[i];
            stamps[i + 1] = (datetime)HistoryDealGetInteger(tickets[i], DEAL_TIME);
        }
        ...
    }
    if(!singleSymbol) // send the tester's statistics and the overall curve to the reporter
    {
        TradeReport::GenericStats stats;
        stats.fillByTester();
        writer.addCurve(stats, curve, stamps, "Overall", clrBlack, 3);
    }
    ...
}

```

我们看看我们得到了什么。如果我们使用设置 *UnityMartingale-combo.set* 运行 EA 交易，将在其中一个代理的 *MQL5/Files* 文件夹中获得 *temp.html* 文件。以下是其在浏览器中的效果。

Trade Report



EURUSD

Profit>	45.09	1.03	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-43.99	0.01	<Sharpe	Buy	16	18	34
Net>	1.10	0.07	<Recovery	Sell	17	21	38
Deposit>	10000.00	144	<Deals	Total	33	39	72
DD	Absolute	Relative	Deposit	Max.	Average	Max.	
Money	16.83	16.83	-0.00	Profit		Loss	
Percent	0.17%	0.17%	-0.00%	2.00	0.02	-2.00	

GBPCHF

Profit>	131.26	1.25	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-105.30	0.09	<Sharpe	Buy	17	18	35
Net>	25.96	1.32	<Recovery	Sell	18	22	40
Deposit>	10000.00	150	<Deals	Total	35	40	75
DD	Absolute	Relative	Deposit	Max.	Average	Max.	
Money	19.74	19.74	-0.00	Profit		Loss	
Percent	0.20%	0.20%	-0.00%	8.78	0.35	-5.55	

AUDJPY

Profit>	145.96	1.38	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-106.06	0.13	<Sharpe	Buy	30	19	49
Net>	39.90	1.98	<Recovery	Sell	21	22	43
Deposit>	10000.00	184	<Deals	Total	51	41	92
DD	Absolute	Relative	Deposit	Max.	Average	Max.	
Money	20.19	20.19	-0.00	Profit		Loss	
Percent	0.20%	0.20%	-0.00%	7.03	0.43	-4.90	

Overall

Profit>	322.31	1.26	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-255.35	0.09	<Sharpe	Buy	63	55	118
Net>	66.96	1.65	<Recovery	Sell	56	65	121
Deposit>	10000.00	478	<Deals	Total	119	120	239
DD	Absolute	Relative	Deposit	Max.	Average	Max.	
Money	40.68	40.68	-0.00	Profit		Loss	
Percent	0.40%	0.40%	-0.00%	8.78	0.28	-5.55	

具有多种交易策略/交易品种的 EA 交易 HTML 报告

现在，我们知道了如何在一次试传递中生成报告，我们可以在优化期间将其发送到终端，在运行中选择最佳报告，并在整个过程结束之前呈现给用户。所有报告都放在终端的 *MQL5/Files* 内的一个单独文件夹中。该文件夹将收到一个名称，其中包含测试程序设置中的交易品种和时间范围以及 EA 交易名称。

UnityMartingale.mq5

我们知道，要向终端发送一个文件，只需要调用 *FrameAdd* 函数。我们已经在先前版本的框架内生成了该文件。

```
double OnTester()
{
    ...
    if(MQLInfoInteger(MQL_OPTIMIZATION))
    {
        FrameAdd(tempfile, 0, r2 * 100, tempfile);
    }
}
```

在接收 EA 交易实例中，我们将执行必要的准备工作。我们用每次优化传递的主要参数来说明 *Pass* 结构体。

```
struct Pass
{
    ulong id;           // pass number
    double value;       // optimization criterion value
    string parameters;  // optimized parameters as list 'name=value'
    string preset;      // text to generate set-file (with all parameters)
};
```

在 *parameters* 字符串中，“名称=值”对用 `&` 符号连接。这对以后报告的网页交互很有用（`&` 符号是网页地址中组合参数的标准符号）。我们没有介绍设置文件的格式，但是下面形成 *preset* 字符串的源代码允许你在实践中研究这个问题。

当帧到达时，我们将根据优化标准向 *TopPasses* 数组写入改进。当前最佳传递将始终是数组中的最后一轮测试，也可以在 *BestPass* 变量中提供。

```
Pass TopPasses[];    // stack of constantly improving passes (last one is best)
Pass BestPass;       // current best pass
string ReportPath;   // dedicated folder for all html files of this optimization
```

在 *OnTesterInit* 处理程序中，我们创建一个文件夹名。

```
void OnTesterInit()
{
    BestPass.value = -DBL_MAX;
    ReportPath = _Symbol + "-" + PeriodToString(_Period) + "-"
        + MQLInfoString(MQL_PROGRAM_NAME) + "/";
}
```

在 *OnTesterPass* 处理程序中，我们将按顺序只选择那些指标有所改善的帧，为其找到优化的以及其他参数的值，并将所有这些信息添加到结构体的 *Pass* 数组中。

```

void OnTesterPass()
{
    ulong   pass;
    string  name;
    long    id;
    double  value;
    uchar   data[];

    // input parameters for the pass corresponding to the current frame
    string  params[];
    uint    count;

    while(FrameNext(pass, name, id, value, data))
    {
        // collect passes with improved stats
        if(value > BestPass.value && FrameInputs(pass, params, count))
        {
            BestPass.preset = "";
            BestPass.parameters = "";
            // get optimized and other parameters for generating a set-file
            for(uint i = 0; i < count; i++)
            {
                string name2value[];
                int n = StringSplit(params[i], '=', name2value);
                if(n == 2)
                {
                    long pvalue, pstart, pstep, pstop;
                    bool enabled = false;
                    if(ParameterGetRange(name2value[0], enabled, pvalue, pstart, pstep, pstop)
                    {
                        if(enabled)
                        {
                            if(StringLen(BestPass.parameters)) BestPass.parameters += "&";
                            BestPass.parameters += params[i];
                        }

                        BestPass.preset += params[i] + "||" + (string)pstart + "||"
                            + (string)pstep + "||" + (string)psstop + "||"
                            + (enabled ? "Y" : "N") + "<br>\n";
                    }
                    else
                    {
                        BestPass.preset += params[i] + "<br>\n";
                    }
                }
            }

            BestPass.value = value;
            BestPass.id = pass;
            PUSH(TopPasses, BestPass);
            // write the frame with the report to the HTML file

```

```

        const string text = CharArrayToString(data);
        int handle = FileOpen(StringFormat(ReportPath + "%06.3f-%lld.htm", value, pa
            FILE_WRITE | FILE_TXT | FILE_ANSI);
        FileWriteString(handle, text);
        FileClose(handle);
    }
}
}

```

改进后的结果报告保存在文件中，文件名包括优化标准值和传递编号。

以下是最有趣的事情。在 *OnTesterDeinit* 处理程序中，我们可以形成一个通用的 HTML 文件 (*overall.htm*)，该文件允许你一次看到所有的报告 (比如，前 100 个)。它对模板使用了我们前面提到的相同方案。

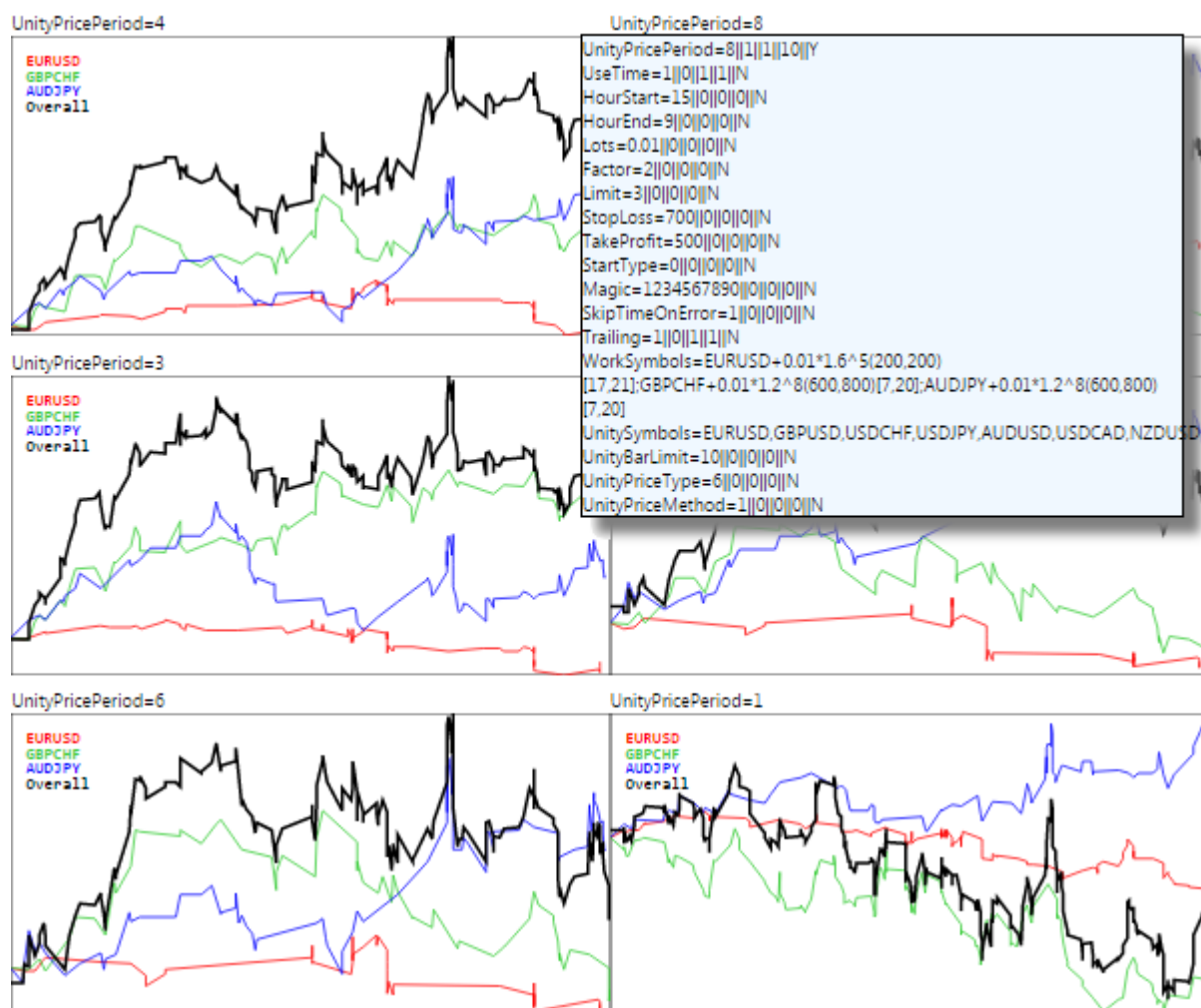
```

#resource "OptReportPage.htm" as string OptReportPageTemplate
#resource "OptReportElement.htm" as string OptReportElementTemplate

void OnTesterDeinit()
{
    int handle = FileOpen(ReportPath + "overall.htm",
        FILE_WRITE | FILE_TXT | FILE_ANSI, 0, CP_UTF8);
    string headerAndFooter[2];
    StringSplit(OptReportPageTemplate, '~', headerAndFooter);
    StringReplace(headerAndFooter[0], "%MINIWIDTH%", (string)MINIWIDTH);
    StringReplace(headerAndFooter[0], "%MINIHEIGHT%", (string)MINIHEIGHT);
    FileWriteString(handle, headerAndFooter[0]);
    // read no more than 100 best records from TopPasses
    for(int i = ArraySize(TopPasses) - 1, k = 0; i >= 0 && k < 100; --i, ++k)
    {
        string p = TopPasses[i].parameters;
        StringReplace(p, "&", " ");
        const string filename = StringFormat("%06.3f-%lld.htm",
            TopPasses[i].value, TopPasses[i].id);
        string element = OptReportElementTemplate;
        StringReplace(element, "%FILENAME%", filename);
        StringReplace(element, "%PARAMETERS%", TopPasses[i].parameters);
        StringReplace(element, "%PARAMETERS_SPACED%", p);
        StringReplace(element, "%PASS%", IntegerToString(TopPasses[i].id));
        StringReplace(element, "%PRESET%", TopPasses[i].preset);
        StringReplace(element, "%MINIWIDTH%", (string)MINIWIDTH);
        StringReplace(element, "%MINIHEIGHT%", (string)MINIHEIGHT);
        FileWriteString(handle, element);
    }
    FileWriteString(handle, headerAndFooter[1]);
    FileClose(handle);
}

```

下图显示了在多币种模式下通过 *UnityPricePeriod* 参数优化 *UnityMartingale.mq5* 后的网页概览外观。



带有最佳优化测试轮次交易报告的网页概览

对于每个报告，我们只显示上半部分，即结余图所在位置。这部分是最方便的，一眼就能看到估算结果。

优化参数列表 (“name=value&name=value...”) 显示在每个图的上方。按下一行，可打开一个块，其中包含该传递所有设置的设置文件文本。如果在块内单击，其内容将被复制到剪贴板上。其可以保存在文本编辑器中，从而得到一个现成的设置文件。

单击图表可进入特定的报告页面以及记分卡 (如上所述)。

在这一节的最后，我们遇到了另一个问题。之前我们承诺展示 `TesterHideIndicators` 函数的效果。`UnityMartingale.mq5` EA 交易当前使用 `UnityPercentEvent.mq5` 指标。在任何测试之后，该指标会显示在打开的图表上。我们假设我们想要对用户隐藏 EA 交易的工作机制并从中获取交易品种。然后，在创建通过 `iCustom` 接收说明符的对象 `UnityController` 之前，可以在 `OnInit` 处理程序中调用 `TesterHideIndicators` 函数 (带有 `true` 参数)。


```

int OnInit()
{
    ...
    TesterHideIndicators(true);
    ...
    controller = new UnityController(UnitySymbols, barwise,
        UnityBarLimit, UnityPriceType, UnityPriceMethod, UnityPricePeriod);
    return INIT_SUCCEEDED;
}

```

此版本的 EA 交易将不再在图表上显示指标。但是，隐藏的也不是很好。如果我们查看测试程序的日志，我们将会在许多有用的信息中看到关于加载程序的行：首先是关于加载 EA 交易本身的消息，之后是关于加载指标的消息。

```

...
expert file added: Experts\MQL5Book\p6\UnityMartingale.ex5.
...
program file added: \Indicators\MQL5Book\p6\UnityPercentEvent.ex5.
...

```

因此，细心的用户可以找到指标名称。这种可能性可以通过资源机制来消除，我们已经在网页空白上下文的传递中提到了这一点。事实证明，已编译的指标也可以作为资源嵌入到 MQL 程序中（EA 交易或其他指标中）。而这样的资源程序在测试程序的日志中也不再提及。我们将在本书的第七章详细研究这些资源，现在我们将在我们的 EA 交易的最终版本中显示与其相关的行。

首先，我们用 `#resource` 指标指令来描述资源。事实上，该指令只包含编译指示符文件的路径（显然，该路径肯定事先已经编译过），此处强制使用双反斜杠作为分隔符，因为前单斜杠在资源路径中不支持。

```
#resource "\\Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5"
```

然后，在 `iCustom` 调用的行中，替换之前的运算符：

```

UnityController(const string symbolList, const int offset, const int limit,
    const ENUM_APPLIED_PRICE type, const ENUM_MA_METHOD method, const int period):
    bar(offset), tickwise(!offset)
{
    handle = iCustom(_Symbol, _Period,
        "MQL5Book/p6/UnityPercentEvent", // <---
        symbolList, limit, type, method, period);
    ...
}

```

完全相同，但是有一个资源的链接（注意以双冒号 '::' 开头的语法，这是区分文件系统中的一般路径和资源中的路径所必需的）。

```

UnityController(const string symbolList, const int offset, const int limit,
    const ENUM_APPLIED_PRICE type, const ENUM_MA_METHOD method, const int period):
    bar(offset), tickwise(!offset)
{
    handle = iCustom(_Symbol, _Period,
        "::.Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5", // <---
        symbolList, limit, type, method, period);
    ...
}

```

现在，EA 交易的编译版本可以单独交付给用户，而无单独指标，因为其隐藏在 EA 交易内部。这不会以任何方式影响其性能，但考虑到 *TesterHideIndicators* 的挑战，内部设备是隐藏的。切记，如果指标随后更新，EA 交易也需要重新编译。

6.5.17 数学计算

MetaTrader 5 终端中的测试程序不仅可以用于测试交易策略，还可以用于数学计算。为此，请在测试程序设置的模拟下拉菜单中选择适当的模式。这与我们选择分时报价生成方法的列表相同，但是在这种情况下，测试程序不会生成分时报价或报价，甚至不会连接交易环境（交易账户以及交易品种）。

参数的完全枚举和遗传算法之间的选择取决于搜索空间的大小。对于优化准则，请选择“自定义最大值”。测试程序设置中的其他输入字段（如日期范围或延迟）并不重要，因此会自动禁用。

在“数学计算”模式下，每次测试代理运行均只调用三个函数：*OnInit*、*OnTester* 和 *OnDeinit*。

在 MetaTrader 5 测试程序中需要解决的一个典型数学问题是寻找一个多变量函数的极值。要解决这个问题，必须以输入变量的形式说明函数参数，并将用于计算其值的块放在 *OnTester* 中。

一组特定输入变量的函数值作为 *OnTester* 的输出值返回。不要在计算中使用除数学函数以外的任何内置函数。

必须记住，优化时，总是寻求 *OnTester* 函数的最大值。因此，如果要求最小值，应返回相反数值或乘以 -1 的值。

为了理解这是如何工作的，我们以一个相对简单的具有一个最大值的二元函数为例。我们在 *MathCalc.mq5* EA 交易算法中进行说明。

通常假设我们不知道函数的解析表达式，否则就有可能计算出其极值。但是现在我们用一个众所周知的公式来确定答案是正确的。

```

input double X1;
input double X2;

double OnTester()
{
    const double r = 1 + sqrt(X1 * X1 + X2 * X2);
    return sin(r) / r;
}

```

EA 交易附带有 *MathCalc.set* 文件，其中包含用于优化的参数：自变量 X1 和 X2 在范围 [-15, +15] 内迭代，步长为 0.5。

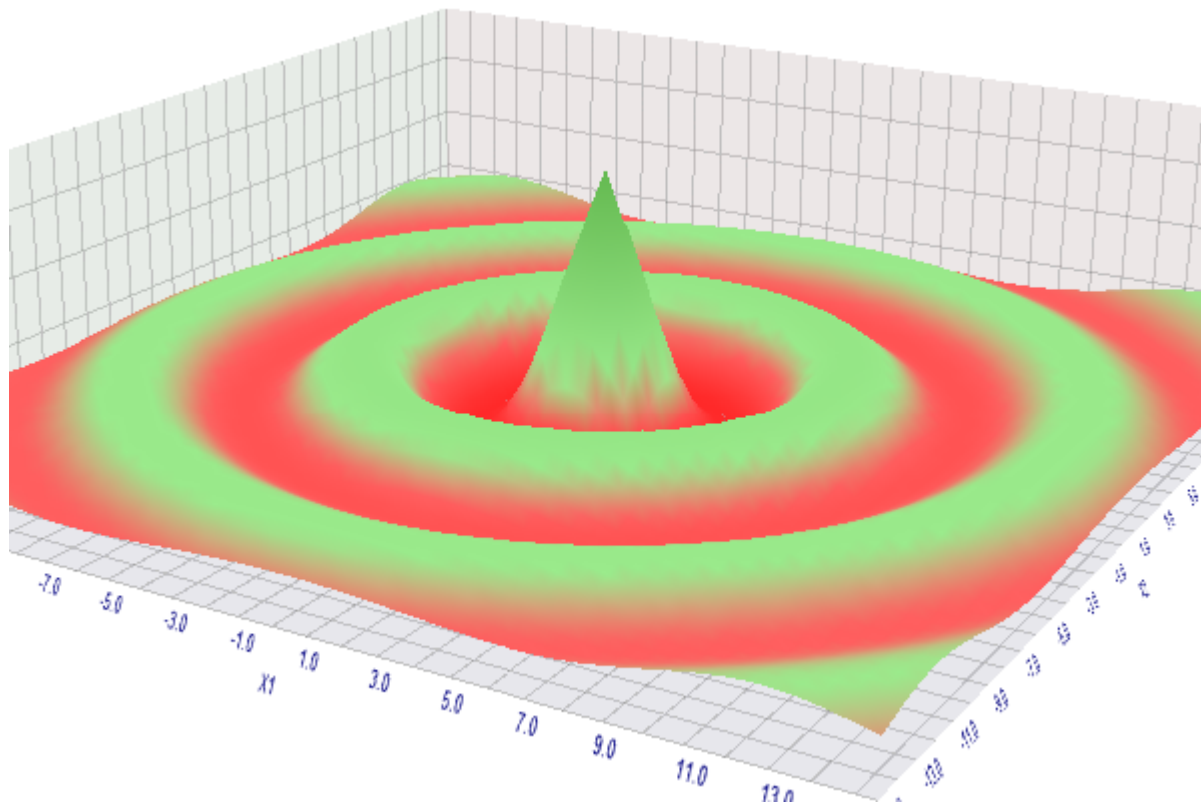
我们运行优化，并在优化表中查看解。最佳轮次可给出正确的结果：

```

X1=0.0
X2=0.0
OnTester result 0.8414709848078965

```

在优化图表上，你可以打开 3D 模式并直观地访问曲面的形状。



数学计算模式下函数优化（最大化）的结果

同时，测试程序在数学计算模式中的使用不限于纯粹的科学研究。具体来说，在此基础上，可以使用替代的已知优化方法（如“粒子群”或“模拟退火”方法）来组织交易系统的优化。当然，要做到这一点，需要将报价或分时报价的历史上传到文件中，并将它们连接到经过测试的 EA 交易以及模拟交易的执行，计算仓位和资金。由于可以自由定制优化过程（与遗传算法的内置“黑盒子”相反）和控制资源（主要是 RAM），所以这种日常工作可能很有意义。

6.5.18 调试和概要分析

MetaTrader 5 测试程序不仅适用于测试交易策略的盈利能力，也适用于调试 MQL 程序。错误检测主要与再现问题情况的能力相关。如果我们只能在线运行 MQL 程序，调试和分析源代码执行将需要不切实际的工作量。但是，测试程序允许你“运行”程序的任意部分的历史，更改账户设置和交易信号。

回想一下，在 MetaEditor 中的 **Debug** 菜单中有 2 个命令：

- 🕒 **Start/Continue on real data (F5)**
- 🕒 **Start/Continue on historical data (Ctrl-F5)**

在这两种情况下，程序会立即以特殊的方式重新编译，并在 `ex5` 文件中添加额外的调试信息，然后直接在终端（第一个选项）或测试程序（第二个选项）中启动。

在测试程序中调试时，可以使用快速（后台）模式和可见模式。此设置在 *Debug/Profile* 选项卡上的 *Setting* 对话框中提供：启用或禁用 *Use visual mode for debugging on history* 标志。调试程序的环境和设置可以直接从测试程序中获取（就像上次为该程序设置的一样），或者在同一对话框中 *Use specified settings* 标志下的输入字段中获取（要使其工作，必须启用该标志）。

你可以在操作符上预设断点（F9），在该操作符上，某些要素可能开始出错。当到达源代码中的指定位置时，测试程序将暂停该过程。

请注意，在测试程序中，启动时加载的历史柱线数量取决于不同的因素（包括时间范围、一年中的天数等），并且可能有很大差异。如有必要，将测试的开始时间向后移动。

除了导致程序停止或明显故障的明显错误之外，还有一类对性能有负面影响的微小错误。通常这些错误并不明显，但是随着处理的数据量增加，就会演变成严重问题，例如，在具有很长历史的交易账户上，或者在具有大量标记对象的图表上。

为了找到性能方面的“瓶颈”，调试人员提供了源代码概要分析机制。该分析机制也可以在线执行或在测试程序中执行，后者尤其有价值，因为它允许大大压缩时间。相应的命令也可以在调试菜单中找到。

🕒 *Start profiling on real data*

🕒 *Start profiling on historical data*

对于概要分析，程序也是用特殊设置预编译的，所以不要忘记在调试或概要分析完成后，在正常模式下再次编译程序（特别是如果你计划将其发送到客户终端或上传到 MQL5 市场）。

作为 MetaEditor 中的概要分析结果，你将收到代码执行的按行和函数（方法）细分的时间统计数据。这样，导致程序运行速度变慢的具体原因就很清楚了。开发的下一个阶段通常是 *refactoring* 源代码，即使用改进的算法、数据结构体或模块（组件）的其他建设性组织原则进行重写。不幸的是，编程中很大一部分时间花在重写现有代码、查找和修复错误上。

如有必要，程序本身可以找出其运行模式，并使其行为适应环境（例如，在测试程序中运行时间，不会尝试从互联网下载数据，因为此函数已禁用，但会从某个文件中读取数据）。

在编译阶段，由于预处理器宏 `_DEBUG` 和 `_RELEASE` 的原因，程序的调试版本和生产版本可能有所不同。

在程序执行阶段，可以使用 *MQLInfoInteger* 函数选项来区分其模式。

下表总结了影响运行时细节的所有可用组合。

运行时间 \ 标志	MQL_DEBUG	MQL_PROFILER	Normal(release)
在线	+	+	+
测试程序 (MQL_TESTER)	+	+	+
测试程序 (MQL_TESTER+MQL_VISUAL_MODE)	+	-	+

只有在没有可见模式的情况下，才能在测试程序中进行概要分析，因此你应使用图表和在线对象来测量操作。

优化过程中不允许进行调试，包括特殊处理程序 *OnTesterInit*、*OnTesterDeinit* 和 *OnTesterPass*。如果需要检查它们的性能，可以考虑在其他条件下调用其代码。

6.5.19 测试程序函数的局限性

使用测试程序时，应注意对内置函数的一些限制。有些 MQL5 API 函数在策略测试程序中从未执行过，有些函数只在单次传递中有效，但在优化过程中无效。

因此，为了提高优化 EA 交易时的性能，不会执行 *Comment*、*Print* 和 *PrintFormat* 函数。

但在 *OnInit* 处理程序中使用这些函数时例外，这是为了更容易找到初始化错误的可能原因。

提供与“世界”交互的函数不会在策略测试程序中执行。这些函数包括 *MessageBox*、*PlaySound*、*SendFTP*、*SendMail*、*SendNotification*、*WebRequest* 以及处理套接字的函数。

此外，许多用于处理图表和对象的函数也不起作用。特别是，将无法通过调用 *ChartSetSymbolPeriod* 来更改当前图表的交易品种或周期，使用 *ChartIndicatorGet* 列出所有指标（包括从属指标），使用模板 *ChartSaveTemplate* 等。

在测试程序中，即使在可见模式下，也不会为 *OnChartEvent* 处理程序生成交互式图表、对象、键盘和鼠标事件。

第七章高级 MQL5 工具

在本书的这一部分中，我们将学习为 MetaTrader 5 环境开发程序时可能需要的各个领域的其他 MQL5 API 功能。其中一些具有应用交易性质，例如：[自定义金融工具](#) 或 [内置经济日历](#)。其他则代表可在任何地方使用的通用技术：[网络函数](#)、[数据库](#)、[加密](#)等。

此外，我们还将考虑使用 [资源](#) 扩展 MQL 程序，这些资源是可嵌入代码中的任意类型文件，可包含多媒体、来自外部程序的“重”设置（例如，现成的机器学习模型或神经网络配置）或以编译形式存在的其他 MQL 程序（指标）。

有几个章节将专门讨论 MQL 程序的模块化开发。在这种情况下，我们将考虑一种特殊的程序类型——[库](#)，它可以连接到其他 MQL 程序，以封闭形式提供现成的特定 API 集，但不能独立使用。我们还将探讨组织软件综合体开发过程并将逻辑相关的程序组合成 [项目](#)。

最后，我们将介绍与其他软件环境的集成，特别是与 [Python](#) 的集成。

本书不涵盖一些高级用户可能感兴趣的高度专业化主题，例如使用 [OpenCL](#) 进行并行计算的硬件功能，以及基于 [DirectX](#) 的 2D 和 3D 图形。建议你使用 [mql5.com](#) 网站上的文档和文章来熟悉这些技术。

MQL5 Programming for Traders - Source Codes from the Book.Part 7

 本书中的示例也可以在 [公共项目](#) \MQL5\Shared Projects\MQL5Book 中找到

7.1 资源

MQL 程序的运行可能需要许多辅助资源，这些资源是应用程序数据数组或各种类型的文件，包括图像、声音和字体。MQL 开发环境允许在编译阶段将所有此类资源包含在可执行文件中。这样便无需与主程序一起并行传输和安装，让主程序成为一个完整、自给自足的产品，方便最终用户使用。

在本章中，我们将学习如何描述不同类型的资源以及用于后续操作已连接资源的内置函数。

光栅图像，以广泛认可的 BMP 格式表示为点（像素）数组，在资源中占有独特的地位。MQL5 API 允许在图表上创建、操作和动态显示这些图形资源。

早些时候，我们已经讨论了图形对象，特别是对于设计用户界面有用的 [OBJ_BITMAP](#) 和 [OBJ_BITMAP_LABEL](#) 类型的对象。对于这些对象，存在 [OBJPROP_BITMAPFILE](#) 属性，该属性将图像指定为文件或资源。以前，我们只考虑了使用文件的示例。现在我们将学习如何使用资源图像。

7.1.1 使用 #resource 指令描述资源

要在编译后的程序版本中包含资源文件，请在源代码中使用 **#resource** 指令。根据文件类型的不同，该指令有不同的形式。无论如何，该指令都包含 **#resource** 关键字，后面跟一个常量字符串。

```
#resource "path_file_name"
```

#resource 命令指示编译器在生成的可执行程序中包含一个文件（二进制格式 **ex5**），该文件具有指定的名称和可选位置（编译时）。路径是可选的：如果字符串中只包含文件名，则会在编译后的源代码旁边的目录中搜索到。如果字符串中包含路径，则适用以下规则。

编译器会按以下顺序在指定路径下查找资源：

- 如果路径前有反斜杠 '\\' 必须是双斜杠，因为单反斜杠是控制字符；特别是，'\' 用于换行符 '\r'、'\n' 和制表符 '\t'，则从终端数据目录内的 MQL5 文件夹开始搜索资源。
- 如果没有反斜杠，则相对于注册该资源的源文件的位置搜索资源。

请注意，在包含资源路径的常量字符串中，必须使用双反斜杠作为分隔符。与文件系统中的路径不同，这里不支持正向单斜杠。

例如：

```
#resource "\\Images\\euro.bmp" // euro.bmp is in /MQL5/Images/
#resource "picture.bmp"         // picture.bmp is in the same directory,
                                // where the source file is (mq5 or mqh)
#resource "Resource\\map.bmp"   // map.bmp is in the Resource subfolder of the directc
                                // where the source file is (mq5 or mqh)
```

如果资源在 mqh 头文件中声明了相对路径，则该路径将被视为相对于该 mqh 文件，而不是相对于正在编译的程序的 mq5 文件。

资源路径中不允许使用子字符串 "..\\" 和 ":\\"。

例如，使用一些指令，你可以将所有必要的图片和声音直接放入 ex5 文件中。这样，在另一个终端运行这样的程序时，就不需要单独传输它们了。我们将在接下来的各节中讨论从 MQL5 中访问资源的编程方法。

常量字符串 "path_file_name" 长度不得超过 63 个字符。资源文件大小不能超过 128 Mb。资源文件在纳入可执行文件中之前会自动压缩。

资源由 **#resource** 指令声明后，可在程序的任何部分使用。资源名称将变为指令中指定的常量字符串，开头不加斜线（如果有），字符串内容前应加上资源的特殊符号（两个冒号， "::"）。

下面我们将举例说明资源，并在注释中列出其名称。

```
#resource "\\Images\\euro.bmp"           // resource name - ::Images\\euro.bmp
#resource "picture.bmp"                   // resource name - ::picture.bmp
#resource "Resource\\map.bmp"             // resource name - ::Resource\\map.bmp
#resource "\\Files\\Pictures\\good.bmp"  // resource name - ::Files\\Pictures\\good.br
#resource "\\Files\\demo.wav";            // resource name - ::Files\\demo.wav"
#resource "\\Sounds\\thrill.wav";         // resource name - ::Sounds\\thrill.wav"
```

在后续的 MQL 代码中，可以按如下方式引用这些资源（在此，我们只知道 **ObjectSetString** 和 **PlaySound** 函数，但还有其他选项，如 **ResourceReadImage**，将在后续各节中介绍）。


```

ObjectSetString(0, bitmap_name, OBJPROP_BITMAP, 0, "::Images\\euro.bmp");
...
ObjectSetString(0, my_bitmap, OBJPROP_BITMAP, 0, "::picture.bmp");
...
ObjectSetString(0, bitmap_label, OBJPROP_BITMAP, 0, "::Resource\\map.bmp");
ObjectSetString(0, bitmap_label, OBJPROP_BITMAP, 1, "::Files\\Pictures\\good.bmp");
...
PlaySound("::Files\\demo.wav");
...
PlaySound("::Sounds\\thrill.wav");

```

需要注意的是，将资源中的图像设置为 OBJ_BITMAP 和 OBJ_BITMAP_LABEL 对象时，不能手动更改 OBJPROP_BITMAP 属性的值（在对象的属性对话框中）。

请注意，默认情况下，*PlaySound* 函数会相对于终端数据目录下的 *Sounds* 文件夹（或其子文件夹）设置 wav 文件。同时，如果资源（包括声音资源）在路径中带有斜线，则会在 MQL5 目录中搜索。因此，在上面的示例中，“\\Sounds\\thrill.wav”字符串指的是 *MQL5/Sounds/thrill.wav* 文件，而不是相对于数据目录的 *Sounds/thrill.wav*（确实有一个包含标准终端声音的 *Sounds* 目录）。

上面讨论的 *#resource* 指令的简单语法只允许描述图像资源（BMP 格式）和声音资源（WAV 格式）。试图将不同类型的文件描述为资源时，会出现“未知资源类型”错误。

经过 *#resource* 指令处理后，文件实际上已嵌入可执行二进制程序，并可通过资源名称访问。此外，还应注意此类资源的一个特殊属性，即它们可以从其他程序中公开获取（下一节将详细介绍）。

MQL5 还支持在程序中嵌入文件的另一种方式：以 *资源变量* 一节中显示了相关示例。此方法使用 *#resource* 指令的扩展语法，不仅可以连接 BMP 或 WAV 文件，还可以连接其他文件，例如文本或结构体数组。

我们将在后面几节分析连接资源的实际例子。

7.1.2 共用不同 MQL 程序的资源

资源名称在整个终端中都是唯一的。稍后我们将学习如何不在编译阶段创建资源（通过 *#resource* 指令），而是使用 *ResourceCreate* 函数检查是否支持特定文件。无论如何，资源都是在创建它的程序上下文中声明的，因此全名的唯一性是通过绑定到文件系统（特定文件 *ex5* 的路径和名称）自动提供的。

除了包含和使用资源外，MQL 程序还可以访问另一个已编译程序（*ex5* 文件）的资源。只要使用资源的程序知道另一个包含所需资源的程序的位置路径和名称，以及该资源的名称，就可以做到这一点。

因此，终端提供了资源的一个重要特性，即共用：一个 *ex5* 文件中的资源可以在许多其他程序中使用。

要使用第三方 *ex5* 文件中的资源，必须以 “path_file_name.ex5::resource_name” 的形式指定。例如，假设 *DrawingScript.mq5* 脚本引用了 *triangle.bmp* 文件中的指定图像资源：

```
#resource "\\Files\\triangle.bmp"
```

那么它在实际脚本中的名称将是 “::Files\\triangle.bmp”。

要使用其他程序 (例如智能交易系统) 中的相同资源, 资源名称前应加上相对于终端数据目录中 MQL5 文件夹的 ex5 脚本文件路径, 以及脚本本身的名称 (编译形式为 **DrawingScript.ex5**)。将脚本添加到标准 **MQL5/Scripts/** 文件夹中。在本例中, 应使用 “\\Scripts\\DrawingScript.ex5::Files\\triangle.bmp” 字符串访问图像。“ex5” 扩展名是可选的。

如果在访问另一个 ex5 文件的资源时, 没有指定该文件的路径, 则会在请求资源的程序所在的同一文件夹中搜索该文件。例如, 如果我们假设同一智能交易系统位于标准 **MQL5/Experts/** 文件夹中, 并且查询资源时未指明路径 (例如, “DrawingScript.ex5::Files\\triangle.bmp”), 则将在 **MQL5/Experts/** 文件夹中搜索 **DrawingScript.ex5**。

由于资源是共用的, 它们的动态创建和更新可用于在 MQL 程序之间交换数据。这直接发生在内存中, 因此不失为文件或全局变量的理想替代选择。

请注意, 要从 MQL 程序中加载资源, 并不需要运行该程序: 要读取资源, 只需拥有一个包含资源的 ex5 文件即可。

当资源以 [资源变量](#) 一节中显示了相关示例。

7.1.3 资源变量

#resource 指令有一种特殊形式, 可以将外部文件声明为资源变量, 并在程序中作为相应类型的普通变量进行访问。声明格式为:

```
#resource "path_file_name" as resource_variable_type resource_variable_name
```

下面是一些声明示例:

```
#resource "data.bin" as int Data[]           //array of int type with data from the f
#resource "rates.dat" as MqlRates Rates[]    // array of MqlRates structures from the
#resource "data.txt" as string Message       // line with the contents of the file da
#resource "image.bmp" as bitmap Bitmap1[]    // one-dimensional array with image pixe
                                              // from file image.bmp
#resource "image.bmp" as bitmap Bitmap2[][]  // two-dimensional array with the same i
```

我们来解释一下。资源变量是常量 (不能在 MQL5 代码中修改)。例如, 要在显示在屏幕上之前编辑图像, 应创建资源数组变量的副本。

对于文本文件 (**string** 类型的资源), 编码会根据是否存在 BOM 头。如果没有 BOM, 则编码由文件内容决定。支持 ANSI、UTF-8 和 UTF-16 编码。从文件读取数据时, 所有字符串都会转换为 Unicode。

使用资源字符串变量不仅能极大地方便基于纯 MQL5 的程序编写, 还能极大地方便基于其他技术的程序编写。例如, 可以在单独文件中编写 OpenCL 代码 (MQL5 支持其作为扩展), 然后将其作为字符串包含在 MQL 程序的资源中。在 [大型智能交易系统示例](#) 中, 我们已经使用资源字符串包含 HTML 模板。

对于图像, 我们引入了一种特殊的 **bitmap** 类型; 这种类型有几个特点。

bitmap 类型描述图像中的单个点或像素, 用 4 字节无符号整数 (**uint**) 表示。像素包含 4 个字节, 分别对应 ARGB 或 XRGB 格式的颜色分量 (一个字母 = 一个字节), 其中 R 代表红色, G 代表绿色, B 代表蓝色, A 代表透明度 (alpha 通道), X 代表忽略字节 (无透明度)。当在图表上叠加图像或图像之间相互叠加时, 透明度可用于产生各种效果。

我们将在有关动态创建图形资源一节中研究 ARGB 和 XRGB 格式的定义 (参见 [ResourceCreate](#))。例如, 对于 ARGB, 十六进制数 0xFFFF0000 表示完全不透明的红色像素 (最高字节为 0xFF) (下一个字节也是 0xFF), 绿色和蓝色分量的下一个字节为 0。

需要注意的是, 像素彩色编码不同于 *color*。我们回顾一下, *color* 类型的值可以用十六进制写成如下形式: 0x00BBGGRR, 其中 BB、GG、RR 分别表示蓝色、绿色和红色分量 (在每个字节中, 数值 255 表示分量的最大强度)。对于类似的像素记录, 字节顺序则相反: 0xAARRGGBB。当高字节 (此处表示 AA) 为 0, 数值 255 为纯色时, 就会实现完全透明。ColorToARGB 函数可用于将 *color* 转换为 ARGB。

BMP 文件可以有多种编码方式 (如果使用任何编辑器创建或编辑 BMP 文件, 请在本程序的文档中查看此问题)。MQL5 资源不支持所有现有的编码方法。可以使用 [ResourceCreate](#) 函数检查是否支持特定文件。在指令中指定不支持的 BMP 格式文件将导致编译错误。

加载 24 位颜色编码的文件时, alpha 通道组件的所有像素都会设置为 255 (不透明)。加载不含 alpha 通道的 32 位彩色编码文件时, 也意味着不透明, 即所有图像像素的 alpha 通道分量都设置为 255。加载含有 alpha 通道的 32 位彩色编码文件时, 不会对像素进行任何操作。

图像可以用一维数组和二维数组来描述。这会影响寻址方法, 而占用的内存量将保持不变。在这两种情况下, 数组的大小都是根据 BMP 文件的数据自动设置的。一维数组的大小等于图像高度和宽度的乘积 ($height * width$), 二维数组则会获得独立的维度 $[height][width]$: 第一个索引是行号, 第二个索引是行中的点。

注意! 在声明链接到资源变量的资源时, 访问该资源的唯一方式是通过该变量, 而通过名称 "::resource_name" (或更一般的 "path_file_name.ex5::resource_name") 读取的标准方式不再有效。这也意味着此类资源不能作为其他程序的共享资源使用。

我们以两个指标为例; 这两个指标都是无缓冲区的。选择这种 MQL 程序类型只是为了方便, 因为除了其他指标外, 它还可以应用于图表而不会发生冲突, 而智能交易系统则需要一个没有其他智能交易系统的图表。此外, 与脚本不同的是, 它们保留在图表上, 可用于后续设置更改。

BmpOwner.mq5 指标包含三个资源的说明:

- "search1.bmp" 图像, 带有简单的 **#resource** 指令, 可从其他程序访问
- "search2.bmp" 图像, 作为 **bitmap** 类型的资源数组变量, 无法从外部访问
- "message.txt" 文本文件, 作为向用户显示警告的资源字符串

这两个图像在该指标中均未以任何方式使用。在调用 **Alert** 的 **OnInit** 函数中需要使用警告行, 因为该指标并非旨在独立使用, 而只是作为图像资源的提供者。

如果在源代码中没有使用资源变量, 编译器可能根本不会在程序的二进制代码中包含该资源, 但这不适用于图像。

```
#resource "search1.bmp"
#resource "search2.bmp" as bitmap image[]
#resource "message.txt" as string Message
```

这三个文件都位于指标源代码所在的同一目录中: **MQL5/Indicators/MQL5Book/p7/**。

如果用户尝试运行该指标, 它会显示警告并立即停止工作。该警告包含在 Message 资源字符串变量中。

```

int OnInit()
{
    Alert(Message); // equivalent to the following line of the code
    // Alert("This indicator is not intended to run, it holds a bitmap resource");

    // remove the indicator explicitly, because otherwise it remains "hanging" on the
    ChartIndicatorDelete(0, 0, MQLInfoString(MQL_PROGRAM_NAME));
    return INIT_FAILED;
}

```

在第二个 *BmpUser.mq5* 指标中，我们将尝试使用输入变量 *ResourceOff* 和 *ResourceOn* 中指定的外部资源，以在 OBJ_BITMAP_LABEL 对象中显示。

```

input string ResourceOff = "BmpOwner.ex5::search1.bmp";
input string ResourceOn = "BmpOwner.ex5::search2.bmp";

```

默认情况下，该对象的状态为禁用/释放 (Off)，其图像取自先前的指标 "BmpOwner.ex5::search1.bmp"。该路径和资源名称类似于完整符号 "\\Indicators\\MQL5Book\\p7\\BmpOwner.ex5::search1.bmp"。在这里，简短的形式是可以接受的，因为这些指标彼此相邻。如果随后打开对象特性对话框，就会在 *Bitmap file (On/Off)* 字段中看到完整的符号。

对于按下状态，我们应在 *ResourceOn* 中读取资源 "BmpOwner.ex5::search2.bmp" (我们看看会发生什么)。

在其他输入变量中，可以选择图表的角，相对于该角设置图像的定位，以及水平和垂直缩进。

```

input int X = 25;
input int Y = 25;
input ENUM_BASE_CORNER Corner = CORNER_RIGHT_LOWER;

```

OBJ_BITMAP_LABEL 对象的创建及其特性的设置 (包括作为 OBJPROP_BITMAP 图片的资源名称) 均在 *OnInit* 中执行。

```

const string Prefix = "BMP_";
const ENUM_ANCHOR_POINT Anchors[] =
{
    ANCHOR_LEFT_UPPER,
    ANCHOR_LEFT_LOWER,
    ANCHOR_RIGHT_LOWER,
    ANCHOR_RIGHT_UPPER
};

void OnInit()
{
    const string name = Prefix + "search";
    ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);

    ObjectSetString(0, name, OBJPROP_BMPFILE, 0, ResourceOn);
    ObjectSetString(0, name, OBJPROP_BMPFILE, 1, ResourceOff);
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, X);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, Y);
    ObjectSetInteger(0, name, OBJPROP_CORNER, Corner);
    ObjectSetInteger(0, name, OBJPROP_ANCHOR, Anchors[(int)Corner]);
}

```

请注意，在 OBJPROP_BMPFILE 中指定图片时，按下状态用修饰符 0 表示，而释放（未按下）状态（默认情况下）用修饰符 1 表示，这有点出乎意料。

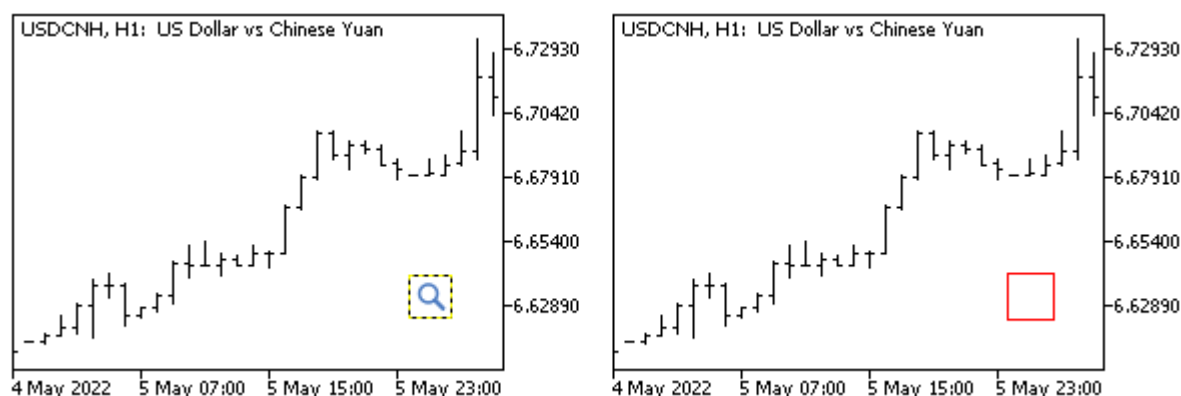
卸载指标时，*OnDeinit* 处理程序会删除对象。

```

void OnDeinit(const int)
{
    ObjectsDeleteAll(0, Prefix);
}

```

我们来编译这两个指标，并以默认设置运行 *BmpUser.ex5*。图形文件 *search1.bmp* 的图像应出现在图表上（见左图）。



图表上对象中图形资源的正常显示（左）和不正确显示（右）

如果点击图片，也就是将其切换到按下状态，程序将尝试访问 “BmpOwner.ex5::search2.bmp” 资源（由于资源数组 *bitmap* 附加在该资源上，因此该资源不可用）。结果，我们将看到一个红色方块，表示一个没有图片的空对象（见右上图）。如果输入参数指定了一个已知不存在或未共享资源的路径或名称，那么总是会发生类似的情形。可以创建自己的程序，在程序中描述指向某个

现有 bmp 文件的资源，然后在指标输入参数 *BmpUser* 中指定。在这种情况下，指标将能够在图表上显示图片。

7.1.4 以资源形式连接自定义指标

为便于运算，MQL 程序可能需要一个或多个自定义指标。所有这些都可以作为资源包含在 ex5 可执行文件中，使其易于分发和安装。

包含嵌套指标说明的 *#resource* 指令格式如下：

```
#resource "path_indicator_name.ex5"
```

设置和搜索指定文件的规则，与一般所有资源 的处理方式相同。

我们已经在 大型智能交易系统示例的 *UnityMartingale.mq5* 最终版本中使用了这一功能。

```
#resource "\\Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5"
```

在该智能交易系统中，传递给 *iCustom* 函数的不是指标名称，而是该资源：“::Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5”。

关于 *OnInit* 函数里的自定义指标创建自身一个或多个实例的情况，需要特别加以考虑（如果这个技术方案本身看起来有点奇怪，我们会在介绍性范例之后提供一个实际应用的例子）。

我们知道，要使用 MQL 程序中的资源，必须以如下形式指定：
path_file_name.ex5::resource_name。例如，如果 *EmbeddedIndicator.ex5* 指标作为资源包含在另一个指标 *MainIndicator.mq5*（更准确地说，包含在其二进制映像 *MainIndicator.ex5*）中，那么通过 *iCustom* 调用时指定的名称就不能再是简短的无路径名称，路径必须包含 MQL5 文件夹中“父”指标的位置。否则，系统将无法找到嵌套指标。

事实上，在正常情况下，指标可以使用 *iCustom(_Symbol, _Period, myself,...)* 运算符调用自身，其中 *myself* 是一个字符串，等于 *MQLInfoString(MQL_PROGRAM_NAME)* 或之前在代码中分配给 INDICATOR_SHORTNAME 特性的名称。但是，当指标作为资源位于另一个 MQL 程序中时，名称就不再指向相应的文件，因为作为资源原型的文件仍保留在进行编译的计算机上，而在用户的计算机上只有 *MainIndicator.ex5* 文件。这就需要在启动程序时对程序环境进行一些分析。

我们来实际操作一下。

首先，我们来创建一个 *NonEmbeddedIndicator.mq5* 指标。需要注意的是，它位于 *MQL5/Indicators/MQL5Book/p7/SubFolder/* 文件夹中，即相对于本书这章所有指标所在的 *p7* 文件夹下的 *SubFolder* 中。这样做的目的是模拟用户计算机上没有编译文件的情况。现在让我们看看它是如何运作的（或者说，演示了这个问题）。

该指标只有一个输入参数 *Reference*。它的作用是计算自身副本的数量：在首次创建时，参数值为 0，指标将创建参数值为 1 的自身副本。第二个副本在“检测到”数值 1 之后，就不会再建立新的副本了（否则，如果没有停止复制的边界条件，我们的资源很快就会用尽）。

```
input int Reference = 0;
```

handle 变量保留给复制指标的句柄。

```
int handle = 0;
```

在处理程序 *OnInit* 中，为了清晰起见，我们首先显示 MQL 程序的名称和路径。

```

int OnInit()
{
    const string name = MQLInfoString(MQL_PROGRAM_NAME);
    const string path = MQLInfoString(MQL_PROGRAM_PATH);
    Print(Reference);
    Print("Name: " + name);
    Print("Full path: " + path);
    ...
}

```

接下来是适合自启动单独指标的代码（以我们熟悉的文件 *NonEmbeddedIndicator.ex5* 的形式存在）。

```

if(Reference == 0)
{
    handle = iCustom(_Symbol, _Period, name, 1);
    if(handle == INVALID_HANDLE)
    {
        return INIT_FAILED;
    }
}
Print("Success");
return INIT_SUCCEEDED;
}

```

我们可以成功地在图表上放置这样一个指标，并在日志中接收以下类型的条目（你将拥有自己的文件系统路径）：

```

0
Name: NonEmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\NonEmbedded
Success
1
Name: NonEmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\NonEmbedded
Success

```

只需使用“NonEmbeddedIndicator”这一名称，复制就能成功启动。

我们现在暂时先不讨论这个指标，来创建第二个指标 *FaultyIndicator.mq5*。我们会将第一个指标作为资源包含进这个新指标中（请注意资源相对路径中 *subfolder* 的指定；这是必要的，因为 *FaultyIndicator.mq5* 指标位于上一层文件夹：*MQL5/Indicators/MQL5Book/p7/*）。

```
// FaultyIndicator.mq5
#resource "SubFolder\\NonEmbeddedIndicator.ex5"

int handle;

int OnInit()
{
    handle = iCustom(_Symbol, _Period, "::SubFolder\\NonEmbeddedIndicator.ex5");
    if(handle == INVALID_HANDLE)
    {
        return INIT_FAILED;
    }
    return INIT_SUCCEEDED;
}
```

如果你尝试运行已编译的 *FaultyIndicator.ex5*, 则会出现错误:

```
0
Name: NonEmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\FaultyIndicator.ex5 »
» ::SubFolder\NonEmbeddedIndicator.ex5
cannot load custom indicator 'NonEmbeddedIndicator' [4802]
```

当启动一个内嵌指标的副本时, 系统会在描述该资源的那个主指标所在的文件夹中搜寻它。但是并没有 *NonEmbeddedIndicator.ex5* 文件, 因为所需资源在 *FaultyIndicator.ex5* 中。

为了解决这个问题, 我们需要修改 *NonEmbeddedIndicator.mq5*。首先, 我们给它取一个更合适的名字, 即 *EmbeddedIndicator.mq5*。在源代码中, 我们需要添加一个辅助函数 *GetMQL5Path*, 它可以从已启动的 MQL 程序的常规路径中分离出 MQL5 文件夹内的相对部分 (如果指标是从资源启动的, 这部分还将包含资源名称)。

```
// EmbeddedIndicator.mq5
string GetMQL5Path()
{
    static const string MQL5 = "\\MQL5\\";
    static const int length = StringLen(MQL5) - 1;
    static const string path = MQLInfoString(MQL_PROGRAM_PATH);
    const int start = StringFind(path, MQL5);
    if(start != -1)
    {
        return StringSubstr(path, start + length);
    }
    return path;
}
```

考虑到新函数, 我们将更改 *OnInit* 处理程序中的 *iCustom* 调用。

```

int OnInit()
{
    ...
    const string location = GetMQL5Path();
    Print("Location in MQL5:" + location);
    if(Reference == 0)
    {
        handle = iCustom(_Symbol, _Period, location, 1);
        if(handle == INVALID_HANDLE)
        {
            return INIT_FAILED;
        }
    }
    return INIT_SUCCEEDED;
}

```

我们确保这次修改不会破坏指标的启动。在图表上叠加后，日志中会出现预期的文本行：

```

0
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\EmbeddedInd
Location in MQL5:\Indicators\MQL5Book\p7\SubFolder\EmbeddedIndicator.ex5
Success
1
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\EmbeddedInd
Location in MQL5:\Indicators\MQL5Book\p7\SubFolder\EmbeddedIndicator.ex5
Success

```

我们在这里添加了 `GetMQL5Path` 函数接收到的相对路径的调试输出。这一行现在用于 `iCustom`，并在此模式下运作：创建了一个副本。

现在，我们将该指标作为资源嵌入 `MQL5Book/p7` 文件夹中的另一个指标，其名称为 `MainIndicator.mq5`。除了连接的资源外，`MainIndicator.mq5` 与 `FaultyIndicator.mq5` 完全相同。

```

// MainIndicator.mq5

#resource "SubFolder\\EmbeddedIndicator.ex5"

...
int OnInit()
{
    handle = iCustom(_Symbol, _Period, "::SubFolder\\EmbeddedIndicator.ex5");
    ...
}

```

我们来编译并运行它。日志中出现的条目会带有一个包含嵌套资源的新相对路径。


```

0
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\MainIndicator.ex5 »
» ::SubFolder\EmbeddedIndicator.ex5
Location in MQL5:\Indicators\MQL5Book\p7\MainIndicator.ex5::SubFolder\EmbeddedIndicat
Success
1
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\MainIndicator.ex5 »
» ::SubFolder\EmbeddedIndicator.ex5
Location in MQL5:\Indicators\MQL5Book\p7\MainIndicator.ex5::SubFolder\EmbeddedIndicat
Success

```

我们可以看到，这次嵌套指标成功地创建了自己的副本，因为它使用了带有相对路径的限定名和资源名 “\\Indicators\\MQL5Book\\p7\\MainIndicator.ex5::SubFolder\\EmbeddedIndicator.ex5”。

在启动该指标的多次实验中，请注意在移除主指标后，嵌套副本不会立即从图表中卸载。因此，只有在等待卸载后才能重新启动；否则，仍在运行的副本将被重复使用，上述初始化行将不会出现在日志中。为了控制卸载，我们在 *OnDeinit* 处理程序中添加了 *Reference* 值的打印输出。

我们承诺向大家展示，创建指标的副本并不是什么特别的事情。作为该技术的应用演示，我们使用指标 *DeltaPrice.mq5*，它可以计算给定阶数的价格增量差。阶数 0 表示无差异（仅检查原始时间序列），1 表示单差异，2 表示双差异，以此类推。

阶数在 *Differentiating* 输入参数中指定。

```
input int Differencing = 1;
```

差值序列将显示在子窗口的单个缓冲区中。

```

#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots 1

#property indicator_type1 DRAW_LINE
#property indicator_color1 clrDodgerBlue
#property indicator_width1 2
#property indicator_style1 STYLE_SOLID

double Buffer[];

```

在 *OnInit* 处理程序中，我们将设置缓冲区并创建相同的指标，在输入参数中传递减 1 的值。

```

#include <MQL5Book/AppliedTo.mqh> // APPLIED_TO_STR macro

int handle = 0;

int OnInit()
{
    const string label = "DeltaPrice (" + (string)Differencing + "/"
        + APPLIED_TO_STR() + ")";
    IndicatorSetString(INDICATOR_SHORTNAME, label);
    PlotIndexSetString(0, PLOT_LABEL, label);

    SetIndexBuffer(0, Buffer);
    if(Differencing > 1)
    {
        handle = iCustom(_Symbol, _Period, GetMQL5Path(), Differencing - 1);
        if(handle == INVALID_HANDLE)
        {
            return INIT_FAILED;
        }
    }
    return INIT_SUCCEEDED;
}

```

为了避免将指标作为资源嵌入时可能出现的问题，我们使用了已证明有效的函数 *GetMQL5Path*。

在 *OnCalculate* 函数中，我们执行减去时间序列相邻值的运算。当 *Differentiating* 等于 1 时，操作数是 *price* 数组的元素。如果 *Differentiating* 值较大，我们将读取为前一个阶数创建的指标副本的缓冲区。

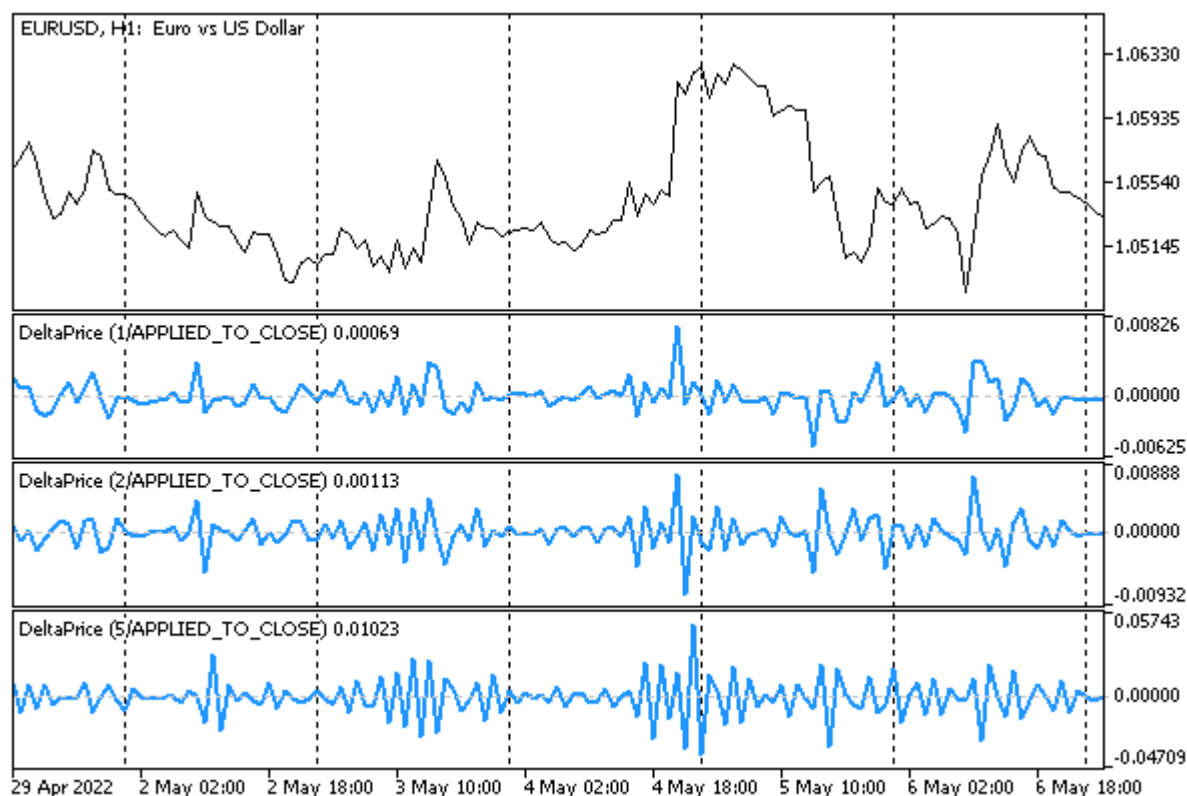
```

int OnCalculate(const int rates_total,
               const int prev_calculated,
               const int begin,
               const double &price[])
{
    for(int i = fmax(prev_calculated - 1, 1); i < rates_total; ++i)
    {
        if(Differencing > 1)
        {
            static double value[2];
            CopyBuffer(handle, 0, rates_total - i - 1, 2, value);
            Buffer[i] = value[1] - value[0];
        }
        else if(Differencing == 1)
        {
            Buffer[i] = price[i] - price[i - 1];
        }
        else
        {
            Buffer[i] = price[i];
        }
    }
    return rates_total;
}

```

差分价格的初始类型在指标设置对话框的 **Apply to** 下拉列表中设置。默认为 *Close* 价。

这就是在图表上，同一个指标的数个副本以不同的差分阶数呈现出来的样子。



不同差分阶数的收盘价差异

7.1.5 动态资源创建: ResourceCreate

#resource 指令在编译阶段就将资源嵌入程序, 因此可以称为静态资源。然而, 在程序执行阶段经常需要生成资源 (创建全新资源或修改现有资源)。为此, MQL5 提供了 **ResourceCreate** 函数。借助该函数创建的资源称为动态资源。

该函数有两种形式: 第一种允许从文件中加载图片和声音, 第二种用于根据内存中准备好的像素数组创建位图图像。

bool ResourceCreate(const string resource, const string filepath)

该函数从位于 **filepath** 的文件中加载名为 **resource** 的资源。如果路径以反斜杠 `\` 开头 (在常量字符串中应是两个: `\\path\\name.ext`), 那么将在该路径下相对于终端数据目录下的 MQL5 文件夹搜索文件 (例如, `\\Files\\Custom Sounds\\Hello.wav` 指的是 **MQL5/Files/CustomSounds/Hello.wav**)。如果没有反斜杠, 则从我们调用函数的可执行文件所在的文件夹开始搜索资源。

路径可以指向硬连接到第三方或当前 MQL 程序中的静态资源。例如, 某个脚本可以根据 **BmpOwner.mq5** 指标 (在关于 **资源变量** 的一节中讨论) 中的图片创建资源。

```
ResourceCreate("::MyImage", "\\Indicators\\MQL5Book\\p7\\BmpOwner.ex5::search1.bmp");
```

resource 参数中的资源名称可以包含首部的双冒号 (但这并不是必需的, 因为如果没有双冒号, 名称中将自动添加 `::` 前缀)。这确保了在 **ResourceCreate** 调用中声明资源和后续访问资源 (例如, 在设置 **OBJPROP_BMPFILE** 属性时) 时统一使用一行。

当然, 如果我们只是想在图表上将第三方图像资源加载到我们的对象中, 那么上述创建动态资源的语句就是多余的, 因为直接将字符串 `\\Indicators\\MQL5Book\\p7\\BmpOwner.ex5::` 赋值给 **OBJPROP_BMPFILE** 属性 **search1.bmp** 就足够了。但是, 如果需要编辑图像, 动态资源是必不可少的。接下来, 我们将在 **读取和修改资源数据** 一节中展示一个示例。

其他 MQL 程序可以通过动态资源的全名 (包括创建资源的程序的路径和名称) 公开获取动态资源。例如, 如果之前的 **ResourceCreate** 调用是由脚本 **MQL5/Scripts/MyExample.ex5** 生成的, 那么另一个 MQL 程序就可以使用全链接 `\\Scripts\\MyExample.ex5::MyImage` 访问相同的资源, 而同一文件夹中的任何其他脚本都可以访问简写 `MyExample.ex5::MyImage` (这里的相对路径只是退化)。上文给出了完整路径 (从 MQL5 根目录文件夹) 和相对路径的编写规则。

ResourceCreate 函数执行后会返回成功 (**true**) 或错误 (**false**) 的布尔指标。与往常一样, 错误代码可以在 **_LastError** 变量中找到。具体来说, 你可能会收到以下错误:

- **ERR_RESOURCE_NAME_DUPLICATED** (4015) - 动态资源和静态资源的名称不匹配
- **ERR_RESOURCE_NOT_FOUND** (4016) - 未找到 **filepath** 参数中的指定资源/文件
- **ERR_RESOURCE_UNSUPPORTED_TYPE** (4017) - 不支持的资源类型或大小超过 2 GB
- **ERR_RESOURCE_NAME_IS_TOO_LONG** (4018) - 资源名称超过 63 个字符

所有这些不仅适用于函数的第一种形式, 也适用于第二种形式。

bool ResourceCreate(const string resource, const uint &data[], uint img_width, uint img_height, uint data_xoffset, uint data_yoffset, uint data_width, ENUM_COLOR_FORMAT color_format)

resource 参数仍表示新资源的名称, 图像内容由其参数给出。

data 数组可以是一维数组 (**data[]**) 或二维数组 (**data[][]**): 传递栅格的点 (像素)。参数 **img_width** 和 **img_height** 设置所显示图像的尺寸 (以像素为单位)。这些尺寸可能小于 **data** 数组中图像的

物理尺寸，因此当只输出原始图像的一部分时，就能达到取景的效果。`data_xoffset` 和 `data_yoffset` 参数决定了“帧”左上角的坐标。

`data_width` 参数表示原始图像的全宽（在 `data` 数组中）。值为 0 意味着该宽度与 `img_width` 相同。只有在 `data` 参数中指定一维数组时，`data_width` 参数才有意义，因为二维数组的两个维度都是已知的（在这种情况下，`data_width` 参数将被忽略，并被假定等于 `data[][]` 数组的第二个维度）。

在最常见的情况下，如果要完整显示图像（按原样），请使用以下语法：

```
ResourceCreate(name, data, width, height, 0, 0, 0, ...);
```

例如，如果程序有一个描述为一个二维 `bitmap` 数组的静态资源：

```
#resource "static.bmp" as bitmap data[][]
```

那么在此基础上创建动态资源的方法如下：

```
ResourceCreate("dynamic", data, ArrayRange(data, 1), ArrayRange(data, 0), 0, 0, 0, ..
```

在静态资源的基础上创建动态资源不仅是为了直接编辑，还可以在显示资源时控制颜色的处理方式。可以使用函数的最后一个参数来选择这种模式：`color_format`。它使用 `ENUM_COLOR_FORMAT` 枚举。

标识符	说明
COLOR_FORMAT_XRGB_NOALPHA	忽略 alpha 通道分量（透明度）
COLOR_FORMAT_ARGB_RAW	终端不处理颜色分量
COLOR_FORMAT_ARGB_NORMALIZE	终端会处理颜色分量（参见下文）

在 `COLOR_FORMAT_XRGB_NOALPHA` 模式下，图像显示时不带效果：每个点以纯色显示（这是最快的绘制方式）。其他两种模式在显示像素时会考虑每个像素高字节的透明度，但效果不同。对于 `COLOR_FORMAT_ARGB_NORMALIZE`，终端在调用 `ResourceCreate` 时准备光栅时，会对每个点的颜色分量进行以下转换：

```
R = R * A / 255
G = G * A / 255
B = B * A / 255
A = A
```

`#resource` 指令中的静态图像资源借助 `COLOR_FORMAT_ARGB_NORMALIZE` 进行连接。

在动态资源中，数组大小受 `INT_MAX` 字节值（2147483647，2 Gb）的限制，这大大超出了编译器在处理静态指令 `#resource` 时施加的限制：文件大小不能超过 128 Mb。

如果调用第二个版本的函数来创建名称相同的资源，但更改了其他参数（像素数组的内容、宽度、高度或偏移量），则不会重新创建新资源，而只是更新现有资源。只有拥有资源的程序（最初创建资源的程序）才能以这种方式修改资源。

如果在不同图表上运行的不同程序副本中创建动态资源时，每个副本中都需要自己的资源，则应在资源名称中添加 `ChartID`。

为了演示动态创建各种配色方案的图像，我们建议拆解脚本 `ARGBbitmap.mq5`。

图像 "argb.bmp" 被静态附加到该脚本中。

```
#resource "argb.bmp" as bitmap Data[][]
```

用户通过 *ColorFormat* 参数选择颜色格式化方法。

```
input ENUM_COLOR_FORMAT ColorFormat = COLOR_FORMAT_XRGB_NOALPHA;
```

显示图像的对象名称和动态资源名称由变量 *BitmapObject* 和 *ResName* 描述。

```
const string BitmapObject = "BitmapObject";
const string ResName = "::image";
```

下面介绍该脚本的主要功能。

```
void OnStart()
{
    ResourceCreate(ResName, Data, ArrayRange(Data, 1), ArrayRange(Data, 0),
        0, 0, 0, ColorFormat);

    ObjectCreate(0, BitmapObject, OBJ_BITMAP_LABEL, 0, 0, 0);
    ObjectSetInteger(0, BitmapObject, OBJPROP_XDISTANCE, 50);
    ObjectSetInteger(0, BitmapObject, OBJPROP_YDISTANCE, 50);
    ObjectSetString(0, BitmapObject, OBJPROP_BMPFILE, ResName);

    Comment("Press ESC to stop the demo");
    const ulong start = TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE);
    while(!IsStopped() // waiting for the user's command to end the demo
        && TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE) == start)
    {
        Sleep(1000);
    }

    Comment("");
    ObjectDelete(0, BitmapObject);
    ResourceFree(ResName);
}
```

该脚本在指定的颜色模式下创建一个新资源，并将其分配给 `OBJ_BITMAP_LABEL` 类型对象的 `OBJPROP_BMPFILE` 属性。接下来，脚本等待用户明确停止脚本或按 *Esc*，然后 通过调用 *ObjectDelete* 删除对象，并使用 *ResourceFree* 函数删除资源。请注意，删除对象不会自动删除资源。这就是为什么我们需要 *ResourceFree* 函数，我们将在 下一节 中讨论。

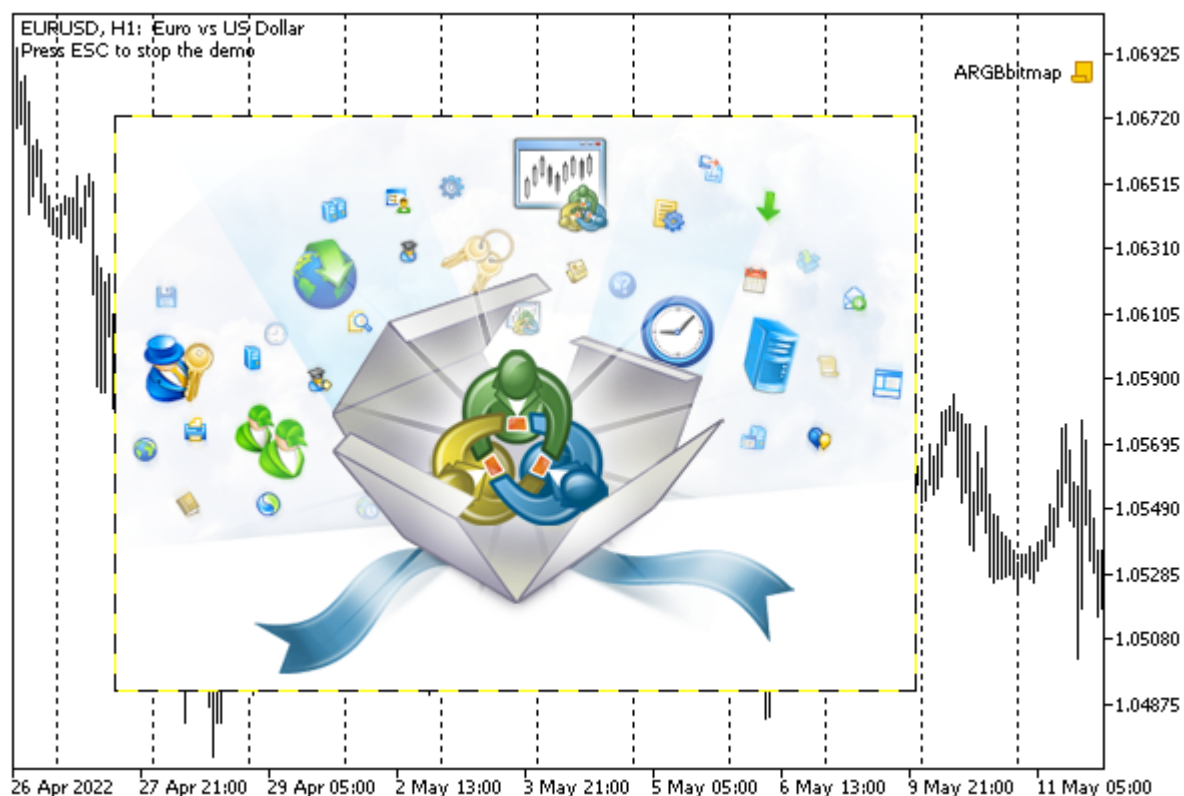
如果我们不调用 *ResourceFree*，那么即使在 MQL 程序终止后，动态资源仍会保留在终端的内存中，直到终端关闭为止。这使得可以将它们用作存储库或 MQL 程序之间交换信息的一种方式。

使用 *ResourceCreate* 的第二种形式创建的动态资源不必携带图像。如果我们不将其用于渲染，*data* 数组可以包含任意数据。在这种情况下，设置 `COLOR_FORMAT_XRGB_NOALPHA` 色彩格式非常重要。我们将在某个时候展示这样一个例子。

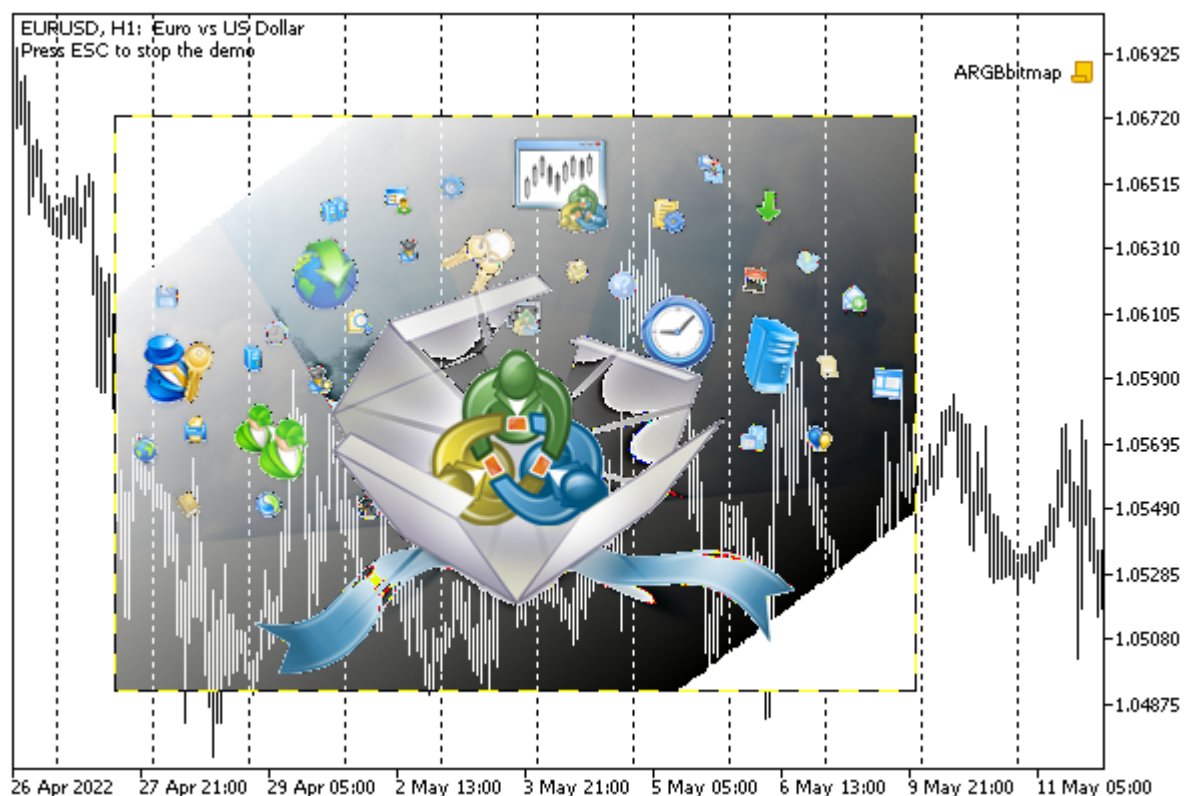
与此同时，我们检查一下 *ARGBbitmap.mq5* 脚本是如何工作的。

上述图片 "argb.bmp" 包含有关透明度的信息：左上角具有完全透明的背景，透明度沿对角线向右下角逐渐消失。

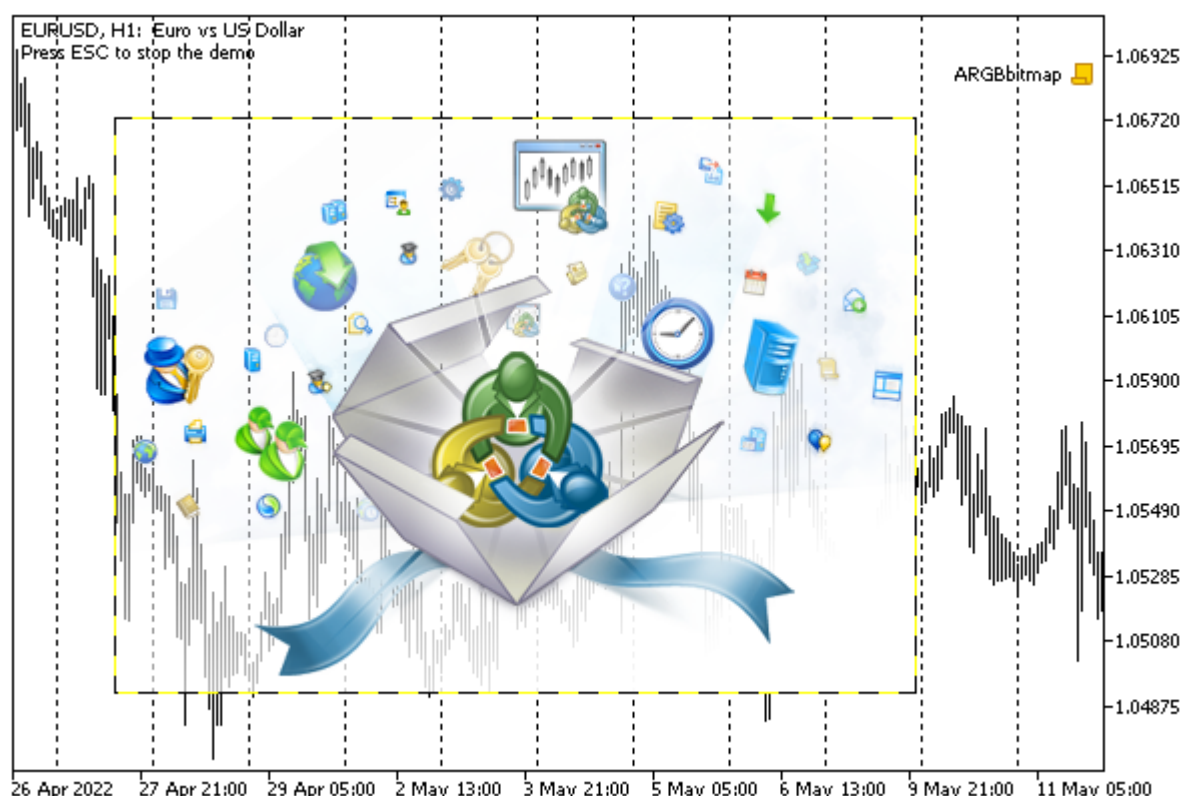
以下图片显示了在三种不同模式下运行脚本的结果。



颜色格式 COLOR_FORMAT_XRGB_NOALPHA 下的图像输出



颜色格式 COLOR_FORMAT_ARGB_RAW 下的图像输出



颜色格式 COLOR_FORMAT_ARGB_NORMALIZE 下的图像输出

7.1.6 删除动态资源: ResourceFree

ResourceFree 函数会删除之前创建的动态资源，并释放其占用的内存。如果不调用 **ResourceFree**，动态资源将保留在内存中，直到当前终端会话结束。这可以作为一种方便的数据存储方式，但对于常规的图像处理工作，建议在对图像的需求消失时释放它们。

与被删除资源相连的图形对象即使在删除后也能正确显示。但是，新创建的图形对象 (**OBJ_BITMAP** 和 **OBJ_BITMAP_LABEL**) 将无法再使用已删除的资源。

bool ResourceFree(const string resource)

资源名称在 **resource** 参数中设置，必须以 “:” 开头。

该函数返回操作状态标识：成功 (**true**) 或错误 (**false**)。

该函数只删除给定 MQL 程序创建的动态资源，而不删除 “第三方” 资源。

在上一节中，我们看到了脚本 **ARGBbitmap.mq5** 的示例，该脚本在完成操作后调用了 **ResourceFree**。

7.1.7 读取和修改资源数据: ResourceReadImage

ResourceReadImage 函数允许读取 **ResourceCreate** 函数创建的资源数据，或者在编译时嵌入可执行文件中的资源数据 (根据 **#resource** 指令)。尽管名称中带有后缀 “Image”，但该函数可用于任何数据数组，包括自定义数组 (请参见下面的 **Reservoir.mq5** 示例)。


```
bool ResourceReadImage(const string resource, uint &data[], uint &width, uint &height)
```

资源名称在 **resource** 参数中指定。要访问自己的资源,使用简短的 `::resource_name` 即可。要从另一个编译文件中读取资源,则需要全名,然后根据 **资源**中所述的路径解析规则确定路径。特别是,以反斜杠开头的路径表示来自 `MQ5` 根文件夹的路径(这样,“\path\filename.ex5::resource_name”就会在 `/MQ5/path/filename.ex5` 文件中以 `resource_name` 的名称被搜索到),而不带前导字符的路径表示相对于执行程序所在文件夹的路径。

资源的内部信息将写入接收 **data** 数组, **width** 和 **height** 参数将分别间接接收宽度和高度,即数组的大小 (**width*height**)。另外,只有当图像存储在资源中时, **width** 和 **height** 才有意义。数组必须是动态或固定的,但必须有足够的大小。否则,我们将收到 `SMALL_ARRAY (5052)` 错误。

如果将来要根据数据 **data** 创建图形资源,那么源资源应使用 `COLOR_FORMAT_ARGB_NORMALIZE` 或 `COLOR_FORMAT_XRGB_NOALPHA` 颜色格式。如果 **data** 数组包含任意应用程序数据,请使用 `COLOR_FORMAT_XRGB_NOALPHA`。

作为第一个例子,我们来看一下 **ResourceReadImage.mq5** 脚本。它演示了图形资源操作的几个方面:

- 从外部文件创建图像资源
- 在另一个动态创建的资源中读取并修改此图像的数据
- 在脚本两次启动之间,在终端内存中保留已创建的资源
- 在图表上的对象中使用资源
- 删除对象和资源

在本特定案例中,图像修改指的是反转所有颜色(这是最具视觉效果的方式)。

上述所有工作方法分三个阶段执行:每个阶段在脚本的一次运行中执行。脚本通过分析可用资源和对象来确定当前阶段:

1. 在缺少所需图形资源的情况下,脚本将创建它们(一个原始图像和一个反转图像)。
2. 如果存在资源但没有图形对象,脚本将使用第一步中的两个图像创建一个对象,用于表示开/关状态(可以通过鼠标点击切换它们)。
3. 如果存在对象,脚本将删除该对象和资源。

脚本的主函数首先定义图表上资源和对象的名称。

```
void OnStart()
{
    const static string resource = "::Images\\pseudo.bmp";
    const static string inverted = resource + "_inv";
    const static string object = "object";
    ...
}
```

请注意,我们为原始资源选择了一个名称,该名称看起来像标准 **Images** 文件夹中 **bmp** 文件的位置,但实际上并不存在这样的文件。这强调了资源的虚拟特性,并允许进行替换以满足技术要求,或者增加程序被逆向工程的难度。

接下来的 **ResourceReadImage** 调用用于检查资源是否已存在。在初始状态下(首次运行时),我们将得到否定结果 (**false**),然后开始第一步:我们从文件 `"\\Images\\dollar.bmp"` 创建原始资源,然后在一个带有 `"_inv"` 后缀的新资源中将其反转。

```

uint data[], width, height;
// check for resource existence
if(!PRTF(ResourceReadImage(resource, data, width, height)))
{
    Print("Initial state: Creating 2 bitmaps");
    PRTF(ResourceCreate(resource, "\\Images\\dollar.bmp")); // try "argb.bmp"
    ResourceCreateInverted(resource, inverted);
}
...

```

辅助函数 *ResourceCreateInverted* 的源代码将在下面给出。

如果找到资源（第二次运行时），脚本会检查对象是否存在，并在必要时创建它，包括在 *ShowBitmap* 函数中（见下文）设置带有图像资源的属性。

```

else
{
    Print("Resources (bitmaps) are detected");
    if(PRTF(ObjectFind(0, object) < 0))
    {
        Print("Active state: Creating object to draw 2 bitmaps");
        ShowBitmap(object, resource, inverted);
    }
    ...
}

```

如果资源和对象都已在图表上，那么我们就处于最后阶段，必须移除所有资源。

```

else
{
    Print("Cleanup state: Removing object and resources");
    PRTF(ObjectDelete(0, object));
    PRTF(ResourceFree(resource));
    PRTF(ResourceFree(inverted));
}
}
}

```

ResourceCreateInverted 函数使用 *ResourceReadImage* 调用来获取像素数组，然后使用 '^'（异或）运算符和一个在颜色分量中所有位都为 1 的操作数来反转其中的颜色。

```

bool ResourceCreateInverted(const string resource, const string inverted)
{
    uint data[], width, height;
    PRTF(ResourceReadImage(resource, data, width, height));
    for(int i = 0; i < ArraySize(data); ++i)
    {
        data[i] = data[i] ^ 0x00FFFFFF;
    }
    return PRTF(ResourceCreate(inverted, data, width, height, 0, 0, 0,
        COLOR_FORMAT_ARGB_NORMALIZE));
}

```

新数组 *data* 被传递给 *ResourceCreate* 以创建第二个图像。

ShowBitmap 函数以常规方式创建一个图形对象 (在图表的右下角), 并将其开和关状态的属性分别设置为原始图像和反转图像。

```
void ShowBitmap(const string name, const string resourceOn, const string resourceOff)
{
    ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);

    ObjectSetString(0, name, OBJPROP_BMPFILE, 0, resourceOn);
    if(resourceOff != NULL) ObjectSetString(0, name, OBJPROP_BMPFILE, 1, resourceOff);
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, 50);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, 50);
    ObjectSetInteger(0, name, OBJPROP_CORNER, CORNER_RIGHT_LOWER);
    ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_RIGHT_LOWER);
}
```

由于新创建的对象默认是关闭状态, 我们首先会看到反转的图像, 并且我们可以通过鼠标点击将其切换到原始图像。但我们要提醒你, 我们的脚本是逐步执行的, 因此, 在图像出现在图表上之前, 脚本必须运行两次。在所有阶段, 当前状态和执行的操作 (以及成功或错误指示) 都会被记录到日志中。

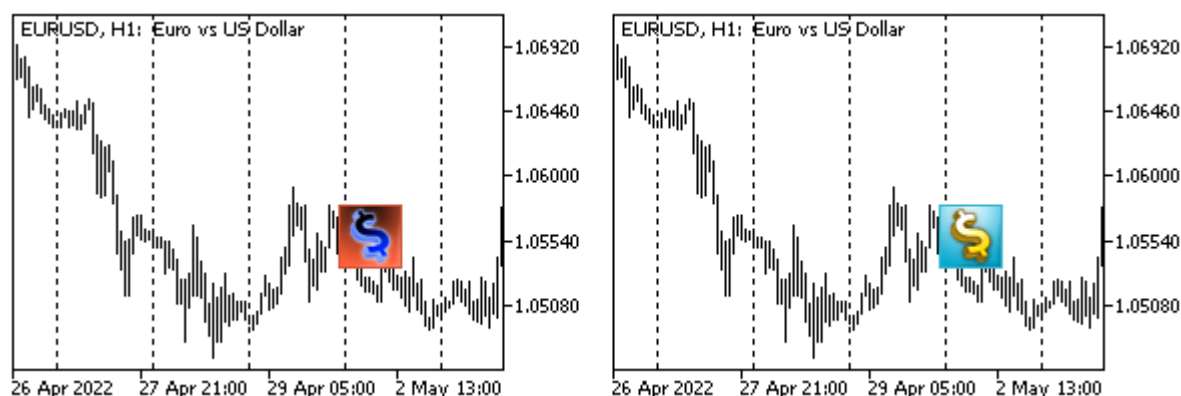
首次启动后, 日志中将出现以下条目:

```
ResourceReadImage(resource,data,width,height)=false / RESOURCE_NOT_FOUND(4016)
Initial state: Creating 2 bitmaps
ResourceCreate(resource,\\Images\\dollar.bmp)=true / ok
ResourceReadImage(resource,data,width,height)=true / ok
ResourceCreate(inverted,data,width,height,0,0,0,COLOR_FORMAT_XRGB_NOALPHA)=true / ok
```

日志显示, 由于找不到资源, 脚本创建了一些资源。在第二次运行时, 日志会显示找到了资源 (这些资源来自脚本的前一次运行并保留在内存中), 但对象仍然存在, 脚本将基于这些资源创建它。

```
ResourceReadImage(resource,data,width,height)=true / ok
Resources (bitmaps) are detected
ObjectFind(0,object)<0=true / OBJECT_NOT_FOUND(4202)
Active state: Creating object to draw 2 bitmaps
```

我们将在图表上看到一个对象和一张图片。可以通过鼠标点击来切换状态 (这里不处理关于状态变化的事件)。



对象在图表上的反转和原始图像。

最后, 在第三次运行时, 脚本将检测到该对象并删除其所有开发成果。

```

ResourceReadImage(resource,data,width,height)=true / ok
Resources (bitmaps) are detected
ObjectFind(0,object)<0=false / ok
Cleanup state: Removing object and resources
ObjectDelete(0,object)=true / ok
ResourceFree(resource)=true / ok
ResourceFree(inverted)=true / ok

```

然后，你就可以重复这个循环。

本节的第二个例子将考虑使用资源来存储任意应用程序数据，也就是说，作为终端内部的一种剪贴板（理论上，可以有任意数量的此类缓冲区，因为它们每一个都是一个独立的命名资源）。由于这个问题的普遍性，我们将创建具有主要功能的 *Reservoir* 类（在 *Reservoir.mqh* 文件中），并在此基础上编写一个演示脚本（*Reservoir.mq5*）。

在直接“深入”*Reservoir* 之前，我们来介绍一个辅助联合体 *ByteOverlay*，它会经常用到。联合体允许任何简单的内置类型（包括简单结构体）转换成字节数组，反之亦然。所谓“简单”，我们指的是所有内置的数值类型、日期和时间、枚举、颜色以及布尔标志。然而，对象和动态数组不再是简单的，并且我们的新存储将不支持它们（由于平台的技术限制）。字符串也不被认为是简单的，但对于它们，我们将作一个例外，并以特殊方式处理它们。

```

template<typename T>
union ByteOverlay
{
    uchar buffer[sizeof(T)];
    T value;

    ByteOverlay(const T &v)
    {
        value = v;
    }

    ByteOverlay(const uchar &bytes[], const int offset = 0)
    {
        ArrayCopy(buffer, bytes, 0, offset, sizeof(T));
    }
};

```

我们知道，资源是建立在 *uint* 类型数组的基础上的，因此我们在 *Reservoir* 类中描述了这样一个数组（*storage*）。我们将把所有随后要写入资源的数据添加到那里。数组中写入或读取数据的当前位置存储在 *offset* 字段中。

```

class Reservoir
{
    uint storage[];
    int offset;
public:
    Reservoir(): offset(0) { }
    ...
}

```

要将任意类型的数据数组放入 *storage* 中，可以使用模板方法 *packArray*。在其前半部分，我们使用 *ByteOverlay* 将传递的数组转换为字节数组。

```

template<typename T>
int packArray(const T &data[])
{
    const int bytesize = ArraySize(data) * sizeof(T); // TODO: check for overflow
    uchar buffer[];
    ArrayResize(buffer, bytesize);
    for(int i = 0; i < ArraySize(data); ++i)
    {
        ByteOverlay<T> overlay(data[i]);
        ArrayCopy(buffer, overlay.buffer, i * sizeof(T));
    }
    ...
}

```

在其后半部分，我们将字节数组转换为 *uint* 值的序列，这些值带有 *offset* 写入 *storage* 中。所需 *uint* 元素的数量是通过考虑数据大小（以字节为单位）除以 *uint* 大小后是否有余数来确定的：我们可以选择性地添加一个额外的元素。

```

    const int size = bytesize / sizeof(uint) + (bool)(bytesize % sizeof(uint));
    ArrayResize(storage, offset + size + 1);
    storage[offset] = bytesize; // write the size of the data before the data
    for(int i = 0; i < size; ++i)
    {
        ByteOverlay<uint> word(buffer, i * sizeof(uint));
        storage[offset + i + 1] = word.value;
    }

    offset = ArraySize(storage);

    return offset;
}

```

在数据本身之前，我们写入数据的字节大小：这是用于恢复数据时进行错误检查的最小可行协议。将来，也可以将 *typename(T)* 数据写入 *storage* 中。

该方法返回写入后存储中的当前位置。

基于 *packArray*，很容易实现一个保存字符串的方法：

```

int packString(const string text)
{
    uchar data[];
    StringToCharArray(text, data, 0, -1, CP_UTF8);
    return packArray(data);
}

```

还有一个存储单个数字的选项：

```

template<typename T>
int packNumber(const T number)
{
    T array[1] = {number};
    return packArray(array);
}

```

一个从 `uint` 类型存储中恢复任意类型 `T` 数组的方法，会反向“回放”所有操作。如果在可读类型和数据量与存储不一致时发现问题，该方法返回 0（错误标志）。在正常模式下，返回 `storage` 数组中的当前位置（如果成功读取了某些内容，它总是大于 0）。

```

template<typename T>
int unpackArray(T &output[])
{
    if(offset >= ArraySize(storage)) return 0; // out of array bounds
    const int bytesize = (int)storage[offset];
    if(bytesize % sizeof(T) != 0) return 0; // wrong data type
    if(bytesize > (ArraySize(storage) - offset) * sizeof(uint)) return 0;

    uchar buffer[];
    ArrayResize(buffer, bytesize);
    for(int i = 0, k = 0; i < ArraySize(storage) - 1 - offset
        && k < bytesize; ++i, k += sizeof(uint))
    {
        ByteOverlay<uint> word(storage[i + 1 + offset]);
        ArrayCopy(buffer, word.buffer, k);
    }

    int n = bytesize / sizeof(T);
    n = ArrayResize(output, n);
    for(int i = 0; i < n; ++i)
    {
        ByteOverlay<T> overlay(buffer, i * sizeof(T));
        output[i] = overlay.value;
    }

    offset += 1 + bytesize / sizeof(uint) + (bool)(bytesize % sizeof(uint));

    return offset;
}

```

解包字符串和数字是通过调用 `unpackArray` 来完成的。

```

int unpackString(string &output)
{
    uchar bytes[];
    const int p = unpackArray(bytes);
    if(p == offset)
    {
        output = CharArrayToString(bytes, 0, -1, CP_UTF8);
    }
    return p;
}

template<typename T>
int unpackNumber(T &number)
{
    T array[1] = {};
    const int p = unpackArray(array);
    number = array[0];
    return p;
}

```

简单的辅助方法可让你找出存储的大小和当前位置，并清除它。

```

int size() const
{
    return ArraySize(storage);
}

int cursor() const
{
    return offset;
}

void clear()
{
    ArrayFree(storage);
    offset = 0;
}

```

现在我们来看最有趣的部分：与资源的交互。

用应用程序数据填充 *storage* 数组后，很容易将其“移动”到提供的资源中。

```

bool submit(const string resource)
{
    return ResourceCreate(resource, storage, ArraySize(storage), 1,
        0, 0, 0, COLOR_FORMAT_XRGB_NOALPHA);
}

```

同样，我们也可以直接从资源中读取数据到内部数组 *storage* 中。

```

bool acquire(const string resource)
{
    uint width, height;
    if(ResourceReadImage(resource, storage, width, height))
    {
        return true;
    }
    return false;
}

```

我们将在 *Reservoir.mq5* 脚本中展示如何使用它。

在 *OnStart* 的前半部分，我们描述了存储资源的名称和 *Reservoir* 类对象，然后依次将一个字符串、*MqlTick* 结构体和一个 *double* 数字“打包”到这个对象中。该结构体被“包装”在一个含有一个元素的数组中，以明确演示 *packArray* 方法。此外，我们稍后需要将恢复的数据与原始数据进行比较，而 MQL5 不为结构体提供 ‘==’ 运算符。因此，使用 *ArrayCompare* 函数会更方便。

```

#include <MQL5Book/Reservoir.mqh>
#include <MQL5Book/PRTF.mqh>

void OnStart()
{
    const string resource = "::reservoir";

    Reservoir res1;
    string message = "message1";    // string to write to the resource
    PRTF(res1.packString(message));

    MqlTick tick1[1];                // add a simple structure
    SymbolInfoTick(_Symbol, tick1[0]);
    PRTF(res1.packArray(tick1));
    PRTF(res1.packNumber(DBL_MAX)); // real number
    ...
}

```

当所有必要的数都“打包”到对象中后，将其写入资源并清除该对象。

```

res1.submit(resource);    // create a resource with storage data
res1.clear();             // clear the object, but not the resource

```

在 *OnStart* 的后半部分，让我们执行从资源中读取数据的反向操作。


```

string reply;                // new variable for message
MqlTick tick2[1];           // new structure for tick
double result;              // new variable for number

PRTF(res1.acquire(resource)); // connect the object to the given resource
PRTF(res1.unpackString(reply)); // read line
PRTF(res1.unpackArray(tick2)); // read simple structure
PRTF(res1.unpackNumber(result)); // read number

// output and compare data element by element
PRTF(reply);
PRTF(ArrayCompare(tick1, tick2));
ArrayPrint(tick2);
PRTF(result == DBL_MAX);

// make sure the storage is read completely
PRTF(res1.size());
PRTF(res1.cursor());
...

```

最后，我们清理资源，因为这是一个测试。在实际任务中，MQL 程序很可能会将创建的资源留在内存中，以便其他程序可以读取它。在命名层级中，资源被声明为嵌套在创建它们的程序中。因此，要从其他程序访问，必须指定资源的名称以及程序的名称，还可以选择性地指定路径（如果程序创建者和程序读取者位于不同的文件夹中）。例如，要从外部读取新创建的资源，完整路径 “\Scripts\MQL5Book\p7\Reservoir\ex5::reservoir” 就能胜任。

```

PrintFormat("Cleaning up local storage '%s'", resource);
ResourceFree(resource);
}

```

由于所有主要的方法调用都由 PRTF 宏控制，当我们运行脚本时，我们将在日志中看到详细的进度“报告”。

```

res1.packString(message)=4 / ok
res1.packArray(tick1)=20 / ok
res1.packNumber(DBL_MAX)=23 / ok
res1.acquire(resource)=true / ok
res1.unpackString(reply)=4 / ok
res1.unpackArray(tick2)=20 / ok
res1.unpackNumber(result)=23 / ok
reply=message1 / ok
ArrayCompare(tick1,tick2)=0 / ok
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.05.19 23:09:32 1.05867 1.05873 0.0000 0 1653001772050 6 0.00000
result==DBL_MAX=true / ok
res1.size()=23 / ok
res1.cursor()=23 / ok
Cleaning up local storage '::reservoir'

```

数据已成功复制到资源中，然后从该资源恢复。

程序可以利用这种方法来交换无法放入自定义消息（事件 `CHARTEVENT_CUSTOM+`）的批量数据。只需在 *sparam* 字符串参数中发送要读取的资源名称即可。要回传数据，请创建你自己的资源，在其中包含该数据，并发送响应消息。

7.1.8 保存图像到文件: ResourceSave

借助 MQL5 API, 你可以使用 *ResourceSave* 函数将资源写入 BMP 文件。该框架目前仅支持图像资源。

bool ResourceSave(const string resource, const string filename)

resource 和 *filename* 参数分别指定资源和文件的名称。资源名称必须以 “::” 开头。文件名可以包含相对于 *MQL5/Files* 文件夹的路径。如有必要, 该函数将创建所有中间子目录。如果指定的文件存在, 它将被覆盖。

函数成功时返回 *true*。

为了测试此函数的操作, 最好创建一个原始图像。我们正好有适合这个的图像。

作为 OOP 学习的一部分, 在 [类和接口](#) 一章中, 我们开始了一系列关于图形形状的示例: 从 [类定义](#) 一节中的第一个版本 *Shapes1.mq5* 到 [嵌套类型](#) 中的最后一个版本 *Shapes6.mq5*。那时我们还无法进行绘图, 直到关于图形对象的章节, 我们才能够在 [ObjectShapesDraw.mq5](#) 脚本中实现可视化。现在, 在学习了图形资源之后, 是时候进行另一次 “升级” 了。

在新版本的 *ResourceShapesDraw.mq5* 脚本中, 我们将绘制这些形状。为了更容易地与先前版本比较更改, 我们将保留相同的形状集: 矩形、正方形、椭圆形、圆形和三角形。这样做是为了举例说明, 而不是因为有什么限制我们绘图: 相反, 在扩展形状集、视觉效果和标签方面存在潜力。我们将通过几个示例来了解这些特性, 从当前的这个开始。但是, 请注意, 本书的范围无法演示所有应用。

在生成并绘制形状后, 我们将生成的资源保存到文件中。

形状类层次结构的基础是具有 *draw* 方法的 *Shape* 类。

```
class Shape
{
public:
    ...
    virtual void draw() = 0;
    ...
}
```

在派生类中, 它是基于图形对象实现的, 通过调用 *ObjectCreate* 并随后使用 *ObjectSet* 函数设置对象。这种绘图的共享画布是图表本身。

现在我们需要根据特定的形状在某个共享资源中绘制像素。最好将公共资源和修改其中像素的方法分配到一个单独的类中, 或者更好的是, 一个接口中。

抽象实体将使我们不必与创建和配置资源的方法产生联系。特别是, 我们的下一个实现会将资源放置在 OBJ_BITMAP_LABEL 对象中 (正如我们在本章中已经做过的那样), 而对于某些情况, 可能只需要在内存中生成图像并保存到磁盘而无需绘制 (许多交易者喜欢定期捕获图表状态)。

我们称这个接口为 *Drawing*。

```

interface Drawing
{
    void point(const float x1, const float y1, const uint pixel);
    void line(const int x1, const int y1, const int x2, const int y2, const color clr)
    void rect(const int x1, const int y1, const int x2, const int y2, const color clr)
};

```

这里只有三个最基本的绘图方法，对于本例来说已经足够了。

point 方法是公共的（这使得可以绘制一个单独的点），但在某种意义上它是低级的，因为所有其他方法都将通过它来实现。这就是为什么其中的坐标是真实的，并且像素的内容是 **uint** 类型的现成值。这将允许在必要时应用各种抗锯齿算法，以使形状看起来不会因像素化而出现阶梯状。这里我们不讨论这个问题。

考虑到接口，**Shape::draw** 方法变为以下形式：

```

virtual void draw(Drawing *drawing) = 0;

```

然后，在 **Rectangle** 类中，很容易将矩形的绘制委托给新的接口。

```

class Rectangle : public Shape
{
protected:
    int dx, dy; // size (width, height)
    ...
public:
    void draw(Drawing *drawing) override
    {
        // x, y - anchor point (center) in Shape
        drawing.rect(x - dx / 2, y - dy / 2, x + dx / 2, y + dy / 2, backgroundColor);
    }
};

```

绘制椭圆需要更多的努力。

```

class Ellipse : public Shape
{
protected:
    int dx, dy; // large and small radii
    ...
public:
    void draw(Drawing *drawing) override
    {
        // (x, y) - center
        const int hh = dy * dy;
        const int ww = dx * dx;
        const int hhww = hh * ww;
        int x0 = dx;
        int step = 0;

        // main horizontal diameter
        drawing.line(x - dx, y, x + dx, y, backgroundColor);

        // horizontal lines in the upper and lower half, symmetrically decreasing in length
        for(int j = 1; j <= dy; j++)
        {
            for(int x1 = x0 - (step - 1); x1 > 0; --x1)
            {
                if(x1 * x1 * hh + j * j * ww <= hhww)
                {
                    step = x0 - x1;
                    break;
                }
            }
            x0 -= step;
            drawing.line(x - x0, y - j, x + x0, y - j, backgroundColor);
            drawing.line(x - x0, y + j, x + x0, y + j, backgroundColor);
        }
    }
};

```

最后，对于三角形，渲染实现如下。

```

class Triangle: public Shape
{
protected:
    int dx; // one size, because triangles are equilateral
    ...
public:
    virtual void draw(Drawing *drawing) override
    {
        // (x, y) - center
        // R = a * sqrt(3) / 3
        // p0: x, y + R
        // p1: x - R * cos(30), y - R * sin(30)
        // p2: x + R * cos(30), y - R * sin(30)
        // Pythagorean height: dx * dx = dx * dx / 4 + h * h
        // sqrt(dx * dx * 3/4) = h
        const double R = dx * sqrt(3) / 3;
        const double H = sqrt(dx * dx * 3 / 4);
        const double angle = H / (dx / 2);

        // main vertical line (triangle height)
        const int base = y + (int)(R - H);
        drawing.line(x, y + (int)R, x, base, backgroundColor);

        // smaller vertical lines left and right, symmetrical
        for(int j = 1; j <= dx / 2; ++j)
        {
            drawing.line(x - j, y + (int)(R - angle * j), x - j, base, backgroundColor);
            drawing.line(x + j, y + (int)(R - angle * j), x + j, base, backgroundColor);
        }
    }
};

```

现在，我们转向派生自 *Drawing* 接口的 *MyDrawing* 类。正是 *MyDrawing* 必须在形状中接口方法调用的指导下，确保某个资源显示在位图中。因此，该类描述了图形对象 (*object*) 和资源 (*sheet*) 名称的变量，以及用于存储图像的 *uint* 类型 *data* 数组。此外，我们移动了先前在 *OnStart* 处理程序中声明的形状数组 *shapes*。由于 *MyDrawing* 负责绘制所有形状，因此最好在这里管理它们的集合。

```

class MyDrawing: public Drawing
{
    const string object; // object with bitmap
    const string sheet; // resource
    uint data[]; // pixels
    int width, height; // dimensions
    AutoPtr<Shape> shapes[]; // figures/shapes
    const uint bg; // background color
    ...
}

```

在构造函数中，我们为整个图表的大小创建一个图形对象，并为 *data* 数组分配内存。画布用零（表示“黑色透明”）或 *background* 参数中传递的任何值填充，然后基于它创建一个资源。默认情况下，资源名称以字母 *D* 开头并包含当前图表的 ID，但你可以指定其他名称。

```

public:
    MyDrawing(const uint background = 0, const string s = NULL) :
        object((s == NULL ? "Drawing" : s)),
        sheet("::" + (s == NULL ? "D" + (string)ChartID() : s)), bg(background)
    {
        width = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
        height = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);
        ArrayResize(data, width * height);
        ArrayInitialize(data, background);

        ResourceCreate(sheet, data, width, height, 0, 0, width, COLOR_FORMAT_ARGB_NORMA

        ObjectCreate(0, object, OBJ_BITMAP_LABEL, 0, 0, 0);
        ObjectSetInteger(0, object, OBJPROP_XDISTANCE, 0);
        ObjectSetInteger(0, object, OBJPROP_YDISTANCE, 0);
        ObjectSetInteger(0, object, OBJPROP_XSIZE, width);
        ObjectSetInteger(0, object, OBJPROP_YSIZE, height);
        ObjectSetString(0, object, OBJPROP_BMPFILE, sheet);
    }

```

调用代码可以使用 *resource* 方法找出资源的名称。

```

string resource() const
{
    return sheet;
}

```

资源和对象在析构函数中被移除。

```

~MyDrawing()
{
    ResourceFree(sheet);
    ObjectDelete(0, object);
}

```

push 方法填充形状数组。

```

Shape *push(Shape *shape)
{
    shapes[EXPAND(shapes)] = shape;
    return shape;
}

```

draw 方法绘制形状。它只是在循环中调用每个形状的 *draw* 方法，然后更新资源和图表。

```

void draw()
{
    for(int i = 0; i < ArraySize(shapes); ++i)
    {
        shapes[i][].draw(&this);
    }
    ResourceCreate(sheet, data, width, height, 0, 0, width, COLOR_FORMAT_ARGB_NORMAL,
    ChartRedraw());
}

```

下面是最重要的方法，它们是 *Drawing* 接口的方法，并且实际实现绘图。

我们从 *point* 方法开始，我们暂时以简化形式呈现它（我们稍后会处理改进）。

```

virtual void point(const float x1, const float y1, const uint pixel) override
{
    const int x_main = (int)MathRound(x1);
    const int y_main = (int)MathRound(y1);
    const int index = y_main * width + x_main;
    if(index >= 0 && index < ArraySize(data))
    {
        data[index] = pixel;
    }
}

```

基于 *point*，很容易实现直线绘制。当起点和终点的坐标在某个维度上匹配时，我们使用 *rect* 方法进行绘制，因为直线是单位厚度矩形的退化情况。

```

virtual void line(const int x1, const int y1, const int x2, const int y2, const cc
{
    if(x1 == x2) rect(x1, y1, x1, y2, clr);
    else if(y1 == y2) rect(x1, y1, x2, y1, clr);
    else
    {
        const uint pixel = ColorToARGB(clr);
        double angle = 1.0 * (y2 - y1) / (x2 - x1);
        if(fabs(angle) < 1) // step along the axis with the largest distance, x
        {
            const int sign = x2 > x1 ? +1 : -1;
            for(int i = 0; i <= fabs(x2 - x1); ++i)
            {
                const float p = (float)(y1 + sign * i * angle);
                point(x1 + sign * i, p, pixel);
            }
        }
        else // or y-step
        {
            const int sign = y2 > y1 ? +1 : -1;
            for(int i = 0; i <= fabs(y2 - y1); ++i)
            {
                const float p = (float)(x1 + sign * i / angle);
                point(p, y1 + sign * i, pixel);
            }
        }
    }
}
}

```

这是 *rect* 方法。

```

virtual void rect(const int x1, const int y1, const int x2, const int y2, const cc
{
    const uint pixel = ColorToARGB(clr);
    for(int i = fmin(x1, x2); i <= fmax(x1, x2); ++i)
    {
        for(int j = fmin(y1, y2); j <= fmax(y1, y2); ++j)
        {
            point(i, j, pixel);
        }
    }
}

```

现在我们需要修改 *OnStart* 处理程序，脚本就准备好了。

首先，我们设置图表（隐藏所有元素）。理论上，这不是必需的：这样做是为了与原型脚本匹配。


```

void OnStart()
{
    ChartSetInteger(0, CHART_SHOW, false);
    ...

```

接下来,我们描述 *MyDrawing* 类的对象,生成预定数量的随机形状(这里一切保持不变,包括 *addRandomShape* 生成器和等于 21 的 FIGURES 宏),在资源中绘制它们,并在图表上的对象中显示它们。

```

MyDrawing raster;

for(int i = 0; i < FIGURES; ++i)
{
    raster.push(addRandomShape());
}

raster.draw(); // display the initial state
...

```

在 *ObjectShapesDraw.mq5* 示例中,我们启动了一个无限循环,在其中随机移动图形块。我们在这里重复这个技巧。这里我们需要添加 *MyDrawing* 类,因为形状数组存储在其中。我们编写一个简单的 *shake* 方法。

```

class MyDrawing: public Drawing
{
public:
    ...
    void shake()
    {
        ArrayInitialize(data, bg);
        for(int i = 0; i < ArraySize(shapes); ++i)
        {
            shapes[i][].move(random(20) - 10, random(20) - 10);
        }
    }
    ...
};

```

然后,在 *OnStart* 中,我们可以在循环中使用新方法,直到用户停止动画。

```

void OnStart()
{
    ...
    while(!IsStopped())
    {
        Sleep(250);
        raster.shake();
        raster.draw();
    }
    ...
}

```

至此,先前示例的功能实际上已经重复了。但是我们需要添加将图像保存到文件的功能。所以,我们添加一个输入参数 *SaveImage*。

```
input bool SaveImage = false;
```

当它设置为 true 时，检查 *ResourceSave* 函数的性能。

```
void OnStart()
{
    ...
    if(SaveImage)
    {
        const string filename = "temp.bmp";
        if(ResourceSave(raster.resource(), filename))
        {
            Print("Bitmap image saved: ", filename);
        }
        else
        {
            Print("Can't save image ", filename, ", ", E2S(_LastError));
        }
    }
}
```

另外，既然我们谈论的是输入变量，让用户选择一个背景并将结果值传递给 *MyDrawing* 构造函数。

```
input color BackgroundColor = clrNONE;
void OnStart()
{
    ...
    MyDrawing raster(BackgroundColor != clrNONE ? ColorToARGB(BackgroundColor) : 0);
    ...
}
```

那么，一切都为第一次测试做好了准备。

如果运行 *ResourceShapesDraw.mq5* 脚本，图表将形成如下图所示的图像。



带有随机形状集的资源位图

将此图像与我们在 [ObjectShapesDraw.mq5](#) 示例中看到的图像进行比较时，结果表明我们的新渲染方式与终端显示对象的方式有所不同。尽管形状和颜色是正确的，但形状重叠的地方以不同的方式指示。

我们的脚本用指定的颜色绘制形状，按照它们在数组中出现的顺序将它们叠加在一起。后面的形状会覆盖前面的形状。另一方面，终端在重叠的地方应用某种颜色混合（反转）。

两种方法都有其存在的理由，这里没有错误。然而，在绘图时是否可以达到类似的效果？

我们完全控制绘图过程，因此不仅可以应用终端的效果，还可以应用任何效果。

除了原始的简单绘图方式外，让我们再实现几种模式。所有这些都总结在 `COLOR_EFFECT` 枚举中。

```
enum COLOR_EFFECT
{
    PLAIN,           // simple drawing with overlap (default)
    COMPLEMENT,      // draw with a complementary color (like in the terminal)
    BLENDING_XOR,    // mixing colors with XOR '^'
    DIMMING_SUM,     // "darken" colors with '+'
    LIGHTEN_OR,      // "lighten" colors with '|'
};
```

我们添加一个输入变量来选择模式。

```
input COLOR_EFFECT ColorEffect = PLAIN;
```

我们在 `MyDrawing` 类中支持这些模式。首先，我们描述相应的字段和方法。

```

class MyDrawing: public Drawing
{
    ...
    COLOR_EFFECT xormode;
    ...
public:
    void setColorEffect(const COLOR_EFFECT x)
    {
        xormode = x;
    }
    ...

```

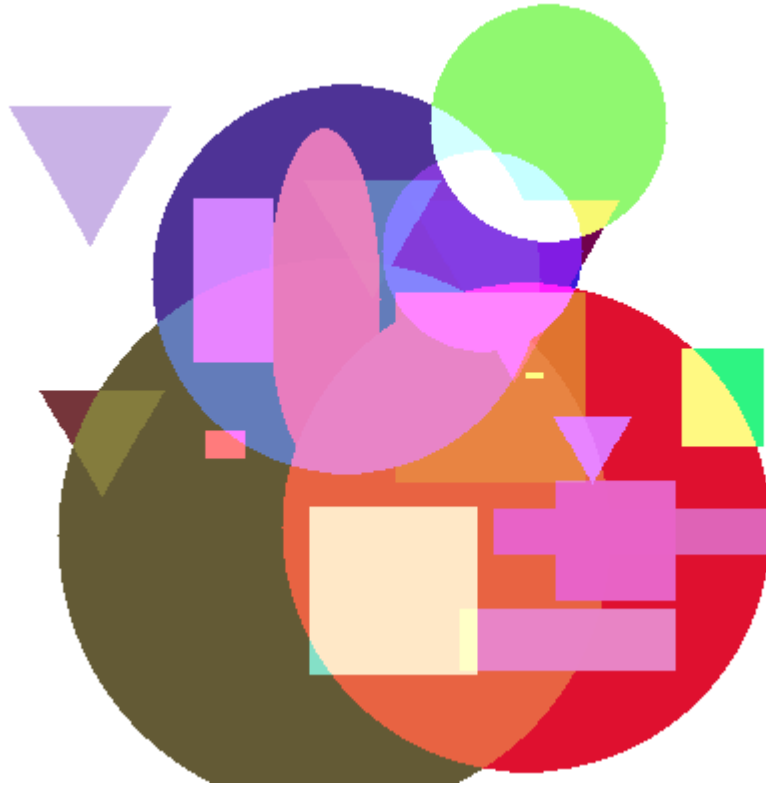
然后，我们改进 *point* 方法。

```

virtual void point(const float x1, const float y1, const uint pixel) override
{
    ...
    if(index >= 0 && index < ArraySize(data))
    {
        switch(xormode)
        {
            case COMPLEMENT:
                data[index] = (pixel ^ (1 - data[index])); // blending with complementary
                break;
            case BLENDING_XOR:
                data[index] = (pixel & 0xFF000000) | (pixel ^ data[index]); // direct mix
                break;
            case DIMMING_SUM:
                data[index] = (pixel + data[index]); // "darkening" (SUM)
                break;
            case LIGHTEN_OR:
                data[index] = (pixel & 0xFF000000) | (pixel | data[index]); // "lighteni
                break;
            case PLAIN:
            default:
                data[index] = pixel;
        }
    }
}

```

你可以尝试以不同的模式运行脚本并比较结果。不要忘记自定义背景的功能。这是一个颜色变亮效果的示例。



带有颜色变亮混合的形状图像

为了直观地看到效果的差异，你可以关闭颜色随机化和形状移动。对象的标准重叠方式对应于 `COMPLEMENT` 常量。

作为最后的实验，启用 *SaveImage* 选项。在 *OnStart* 处理程序中，当生成带有图像的文件名时，我们现在使用当前模式的名称。我们需要在文件中获取图表上图像的副本。

```
...
if(SaveImage)
{
    const string filename = EnumToString(ColorEffect) + ".bmp";
    if(ResourceSave(raster.resource(), filename))
    ...
}
```

对于我们界面中更复杂的图形构造，*Drawing* 可能不够用。因此，可以使用 MetaTrader 5 附带的或 mq5.com 代码库中可用的现成绘图类。特别是，请查看 *MQL5/Include/Canvas/Canvas.mqh* 文件。

7.1.9 字体和文本输出到图形资源

除了在图形资源的数组中渲染单个像素外，我们还可以使用内置函数来显示文本。这些函数用于更改当前字体及其特性 (*TextSetFont*)，获取给定字符串可以内切的矩形的尺寸 (*TextGetSize*)，以及直接将标题插入到生成的图像 (*TextOut*) 中。

```
bool TextSetFont(const string name, int size, uint flags, int orientation = 0)
```

该函数为后续使用 *TextOut* 函数 (见下文) 在图像缓冲区中绘制文本设置字体及其特性。*name* 参数可以包含内置 Windows 字体的名称或通过资源指令连接的 ttf 字体文件 (TrueType 字体) (如果名称以 "://" 开头)。

大小 (**size**) 可以用点 (一种印刷计量单位)或像素 (屏幕点)指定。正值表示计量单位是像素,负值以十分之一点为单位进行测量。以像素为单位的高度对于不同用户来说看起来会有所不同,具体取决于其显示器的技术能力和设置。以点为单位的高度对于每个人来说大致 (“凭肉眼判断”)相同。

印刷点是一种物理长度单位,传统上等于 1/72 英寸。因此,1 点等于 0.352778 毫米。屏幕上的像素是一种虚拟的长度度量。其物理尺寸取决于屏幕的硬件分辨率。例如,屏幕密度为 96 DPI (每英寸点数)时,1 像素将占据 0.264583 毫米或 0.75 点。然而,大多数现代显示器具有更高的 DPI 值,因此像素更小。因此,包括 Windows 在内的操作系统长期以来都有增加界面元素可见比例的设置。因此,如果你以点为单位指定大小 (负值),则文本的像素大小将取决于显示器和操作系统中的缩放设置 (例如,“标准”100%,“中等”125% 或 “大”150%)。

放大时,系统会人为地放大显示的像素。这相当于以像素为单位减小屏幕尺寸,系统会应用有效 DPI 来实现相同的物理尺寸。如果启用了缩放,则报告给程序 (包括终端,然后是 MQL 程序)的是有效 DPI。如有必要,可以从 `TERMINAL_SCREEN_DPI` 属性中找出屏幕的 DPI (请参见 [屏幕规格](#))。然而实际上,通过以点为单位设置字体大小,我们无需根据 DPI 重新计算其大小,因为系统会为我们完成此操作。

默认字体是 Arial,默认大小是 -120 (12 pt)。控件,特别是图表上的内置对象,也以点为单位操作字体大小。例如,如果在 MQL 程序中,你希望绘制与 `OBJ_LABEL` 对象中文本大小相同的文本,该对象的大小为 10 点,则应使用等于 -100 的参数 **size**。

flags 参数用于设置描述字体样式的标志组合。该组合是使用按位或运算符 (`|`) 构成的位掩码。标志分为两组:样式标志和粗细标志。

下表列出了样式标志。它们可以混合使用。

标志	说明
<code>FONT_ITALIC</code>	斜体
<code>FONT_UNDERLINE</code>	下划线
<code>FONT_STRIKEOUT</code>	删除线

粗体标志具有与其对应的相对权重 (用于比较预期效果)。

标志	说明
FW_DONTCARE	0 (将应用系统默认值)
FW_THIN	100
FW_EXTRALIGHT, FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL、FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD、FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD、FW_ULTRABOLD	800
FW_HEAVY、FW_BLACK	900

在标志组合中仅使用这些值中的一个。

orientation 参数指定文本相对于水平线的角度,单位为十分之一度。例如, `orientation = 0` 表示正常文本输出,而 `orientation = 450` 将导致 45 度倾斜 (逆时针)。

请注意,一次 **TextSetFont** 调用中所做的设置将影响所有后续的 **TextOut** 调用,直到它们被更改为止。

如果成功,函数返回 **true**; 如果出现问题 (例如,未找到字体),则返回 **false**。

在描述完所有这三个函数后,我们将考虑一个使用此函数的示例,以及另外两个函数的示例。

bool TextGetSize(const string text, uint &width, uint &height)

该函数返回当前字体设置下 (可以是默认字体或先前调用 **TextSetFont** 时指定的字体) 文本行的宽度和高度。

text 参数传递一个需要计算像素长度和宽度的字符串。维度值由函数根据 **width** 和 **height** 参数中的引用写入。

需要注意的是,调用 **TextSetFont** 时由 **orientation** 参数指定的显示文本的旋转 (倾斜) 不会以任何方式影响大小计算。换句话说,如果文本应该旋转 45 度,那么 MQL 程序本身必须计算文本可以容纳的最小正方形。**TextGetSize** 函数计算标准 (水平) 位置的文本大小。

bool TextOut(const string text, int x, int y, uint anchor, uint &data[], uint width, uint height, uint color, ENUM_COLOR_FORMAT color_format)

该函数在图形缓冲区中的指定坐标处绘制文本,同时考虑颜色、格式和先前的设置 (字体、样式和方向)。

文本在 **text** 参数中传递,并且必须是单行形式。

以像素为单位指定的 **x** 和 **y** 坐标定义了图形缓冲区中显示文本的点。生成的题字的哪个位置将位于点 (x, y) 取决于 **anchor** 参数中的绑定方法 (见下文)。

缓冲区由 *data* 数组表示，尽管该数组是一维的，但它存储了一个维度为 *width x height* 个点的二维“画布”。该数组可以从 *ResourceReadImage* 函数获取，或者由 MQL 程序分配。在完成所有编辑操作（包括文本输出）后，应该基于此缓冲区创建一个新资源或将其应用于已有的资源。在这两种情况下，都应该调用 *ResourceCreate*。

文本的颜色和颜色的处理方式由参数 *color* 和 *color_format* 设置（请参见 *ENUM_COLOR_FORMAT*）。请注意，用于颜色的类型是 *uint*，即，要传递 *color*，应使用 *ColorToARGB* 进行转换。

由 *anchor* 参数指定的锚定方法是两个文本定位标志的组合：垂直和水平。

水平文本位置标志有：

- *TA_LEFT* - 锚定到边界框的左侧
- *TA_CENTER* - 锚定到矩形左右两侧的中间
- *TA_RIGHT* - 锚定到边界框的右侧

垂直文本位置标志有：

- *TA_TOP* - 锚定到边界框的顶部
- *TA_VCENTER* - 锚定到矩形顶部和底部之间的中间
- *TA_BOTTOM* - 锚定到边界框的底部

总共有 9 种有效的标志组合来描述锚定方法。



输出文本相对于锚点的位置

在这里，图片的中心包含一个在生成的图像中坐标为 (*x*, *y*) 的刻意放大的点。根据标志，文本相对于此点出现在指定位置（文本内容对应于应用的锚定方法）。

为便于参考，所有题字均以标准水平位置制作。但请注意，也可以对其中任何一个应用角度 (*orientation*)，然后相应的题字将围绕该点旋转。在此图像中，仅旋转了在两个维度上都居中的标签。

这些标志不应与文本对齐混淆。边界框的大小始终适合文本，其相对于锚点的位置在某种意义上与标志名称相反。

我们看一些使用三个函数的示例。

首先，让我们检查设置字体粗细和样式的最简单选项。*ResourceText.mq5* 脚本允许在输入变量中选择字体名称、其大小以及背景和文本的颜色。标签将在图表上显示指定的秒数。


```

input string Font = "Arial";           // Font Name
input int    Size = -240;              // Size
input color  Color = clrBlue;         // Font Color
input color  Background = clrNONE;    // Background Color
input uint   Seconds = 10;            // Demo Time (seconds)

```

每个粗细等级的名称将显示在标签文本中，因此为了简化过程 通过使用 *EnumToString*)，声明了 `ENUM_FONT_WEIGHTS` 枚举。

```

enum ENUM_FONT_WEIGHTS
{
    _DONTCARE = FW_DONTCARE,
    _THIN = FW_THIN,
    _EXTRALIGHT = FW_EXTRALIGHT,
    _LIGHT = FW_LIGHT,
    _NORMAL = FW_NORMAL,
    _MEDIUM = FW_MEDIUM,
    _SEMIBOLD = FW_SEMIBOLD,
    _BOLD = FW_BOLD,
    _EXTRABOLD = FW_EXTRABOLD,
    _HEAVY = FW_HEAVY,
};

```

```

const int nw = 10; // number of different weights

```

题注标志收集在 *rendering* 数组中，并从中选择随机组合。

```

const uint rendering[] =
{
    FONT_ITALIC,
    FONT_UNDERLINE,
    FONT_STRIKEOUT
};
const int nr = sizeof(rendering) / sizeof(uint);

```

为了在一定范围内获得随机数，有一个辅助函数 *Random*。

```

int Random(const int limit)
{
    return rand() % limit;
}

```

在脚本的主函数中，我们找到图表的大小并创建一个跨越整个空间的 `OBJ_BITMAP_LABEL` 对象。

```

void OnStart()
{
    ...
    const string name = "FONT";
    const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
    const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);

    // object for a resource with a picture filling the whole window
    ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
    ObjectSetInteger(0, name, OBJPROP_XSIZE, w);
    ObjectSetInteger(0, name, OBJPROP_YSIZE, h);
    ...

```

接下来，我们为图像缓冲区分配内存，用指定的背景颜色填充它（或默认情况下保持透明），基于该缓冲区创建一个资源，并将其绑定到对象。

```

uint data[];
ArrayResize(data, w * h);
ArrayInitialize(data, Background == clrNONE ? 0 : ColorToARGB(Background));
ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_RAW);
ObjectSetString(0, name, OBJPROP_BMPFILE, ":@" + name);
...

```

以防万一，请注意，除非对象应该在两种状态之间切换，否则我们可以在 *ObjectSetString* 调用中设置 OBJPROP_BMPFILE 属性而无需修饰符 0 或 1)。

所有字体粗细都列在 *weights* 数组中。

```

const uint weights[] =
{
    FW_DONTCARE,
    FW_THIN,
    FW_EXTRALIGHT, // FW_ULTRALIGHT,
    FW_LIGHT,
    FW_NORMAL,     // FW_REGULAR,
    FW_MEDIUM,
    FW_SEMIBOLD,   // FW_DEMIBOLD,
    FW_BOLD,
    FW_EXTRABOLD,  // FW_ULTRABOLD,
    FW_HEAVY,      // FW_BLACK
};
const int nw = sizeof(weights) / sizeof(uint);

```

在循环中，我们依次使用 *TextSetFont* 为每行设置下一个粗细等级，预先选择一个随机样式。使用 *TextOut* 在缓冲区中绘制字体描述，包括其名称和粗细。

```

const int step = h / (nw + 2);
int cursor = 0;    // Y coordinate of the current "text line"

for(int weight = 0; weight < nw; ++weight)
{
    // apply random style
    const int r = Random(8);
    uint render = 0;
    for(int j = 0; j < 3; ++j)
    {
        if((bool)(r & (1 << j))) render |= rendering[j];
    }
    TextSetFont(Font, Size, weights[weight] | render);

    // generate font description
    const string text = Font + EnumToString((ENUM_FONT_WEIGHTS)weights[weight]);

    // draw text on a separate "line"
    cursor += step;
    TextOut(text, w / 2, cursor, TA_CENTER | TA_TOP, data, w, h,
        ColorToARGB(Color), COLOR_FORMAT_ARGB_RAW);
}
...

```

现在更新资源和图表。

```

ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_RAW);
ChartRedraw();
...

```

用户可以提前停止演示。

```

const uint timeout = GetTickCount() + Seconds * 1000;
while(!IsStopped() && GetTickCount() < timeout)
{
    Sleep(1000);
}

```

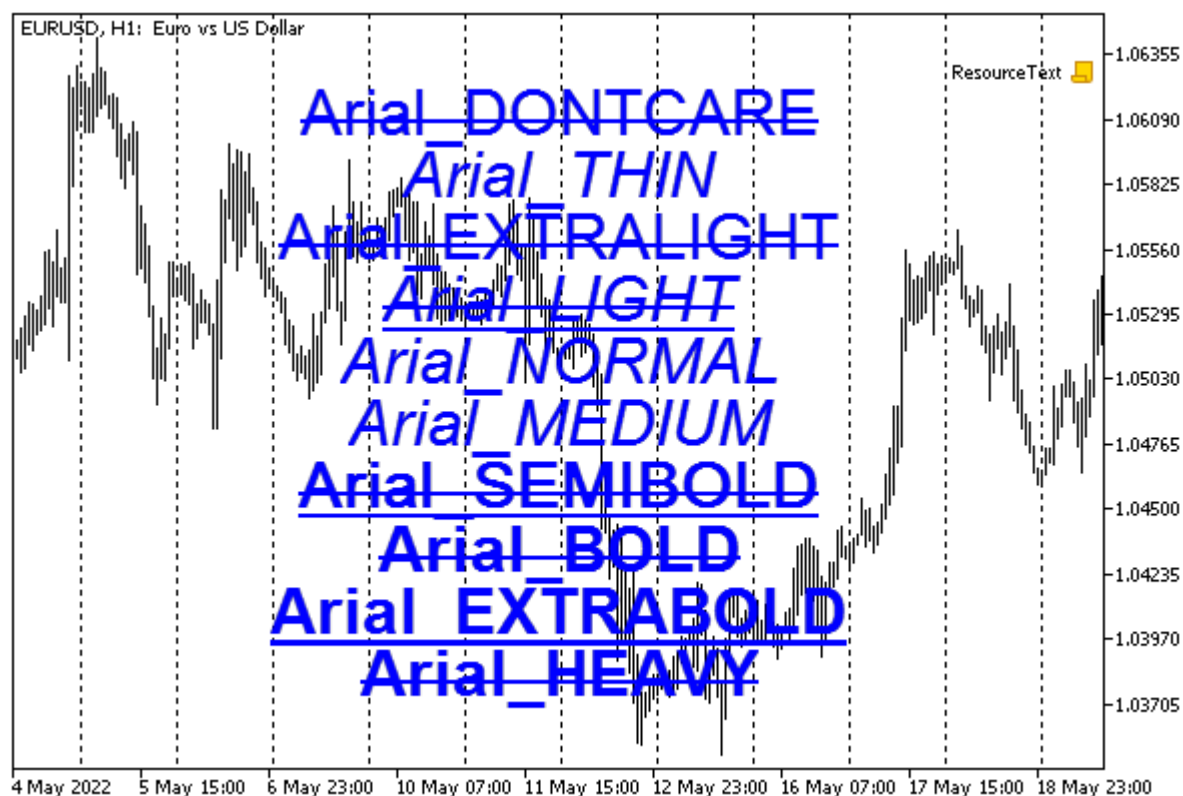
最后，脚本删除资源和对象。

```

ObjectDelete(0, name);
ResourceFree("::" + name);
}

```

脚本的结果如下图所示。



以不同粗细和样式绘制文本

在 *ResourceFont.mq5* 的第二个示例中，我们将通过包含自定义字体作为资源并使用 90 度增量的文本旋转来使任务更加困难。

字体文件位于脚本旁边。

```
#resource "a_LCDNova3DCmObl.ttf"
```

可以在输入参数中更改消息。

```
input string Message = "Hello world!"; // Message
```

这一次，OBJ_BITMAP_LABEL 不会占据整个窗口，因此在水平和垂直方向上都居中。

```
void OnStart()
{
    const string name = "FONT";
    const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
    const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);

    // object for a resource with a picture
    ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
    ObjectSetInteger(0, name, OBJPROP_XDISTANCE, w / 2);
    ObjectSetInteger(0, name, OBJPROP_YDISTANCE, h / 2);
    ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_CENTER);
    ...
}
```

首先，分配最小尺寸的缓冲区，只是为了完成资源创建。稍后我们将扩展它以适应题字的尺寸，为此保留了变量 *width* 和 *height*。

```

uint data[], width, height;
ArrayResize(data, 1);
ResourceCreate(name, data, 1, 1, 0, 0, 1, COLOR_FORMAT_ARGB_RAW);
ObjectSetString(0, name, OBJPROP_BITMAP, "::-" + name);
...

```

在测试时间倒计时的循环中，我们需要更改题字的方向，为此有 *angle* 变量（度数将在其中滚动）。方向将每秒更改一次，计数在 *remain* 变量中。

```

const uint timeout = GetTickCount() + Seconds * 1000;
int angle = 0;
int remain = 10;
...

```

在循环中，我们不断更改文本的旋转，并在文本本身中显示秒数的倒计时计数器。对于每个新的题字，使用 *TextGetSize* 计算其大小，并基于此重新分配缓冲区。

```

while(!IsStopped() && GetTickCount() < timeout)
{
    // apply new angle
    TextSetFont("::a_LCDNova3DCm0bl.ttf", -240, 0, angle * 10);

    // form the text
    const string text = Message + " (" + (string)remain-- + ")";

    // get the text size, allocate the array
    TextGetSize(text, width, height);
    ArrayResize(data, width * height);
    ArrayInitialize(data, 0);          // transparency

    // for vertical orientation, swap sizes
    if((bool)(angle / 90 & 1))
    {
        const uint t = width;
        width = height;
        height = t;
    }

    // adjust the size of the object
    ObjectSetInteger(0, name, OBJPROP_XSIZE, width);
    ObjectSetInteger(0, name, OBJPROP_YSIZE, height);

    // draw text
    TextOut(text, width / 2, height / 2, TA_CENTER | TA_VCENTER, data, width, height,
        ColorToARGB(clrBlue), COLOR_FORMAT_ARGB_RAW);

    // update resource and chart
    ResourceCreate(name, data, width, height, 0, 0, width, COLOR_FORMAT_ARGB_RAW);
    ChartRedraw();

    // change angle
    angle += 90;

    Sleep(100);
}
...

```

请注意，如果文本是垂直的，则需要交换尺寸。更一般地，对于旋转到任意角度的文本，需要更多的数学运算才能获得适合整个文本的缓冲区大小。

最后，我们还删除对象和资源。

```

ObjectDelete(0, name);
ResourceFree("::" + name);
}

```

脚本执行的某个时刻如下图所示。



带有自定义字体的题字

作为最后一个示例，我们看一下 `ResourceTextAnchOrientation.mq5` 脚本，它显示了文本的各种旋转和锚点。

该脚本使用指定的字体生成指定数量的标签 (`ExampleCount`)。

```
input string Font = "Arial";           // Font Name
input int    Size = -150;              // Size
input int    ExampleCount = 11;        // Number of examples
```

锚点和旋转是随机选择的。

为了在标签中指定锚点的名称，声明了包含所有有效选项的 `ENUM_TEXT_ANCHOR` 枚举。因此，我们可以简单地对任何随机选择的元素调用 `EnumToString`。

```
enum ENUM_TEXT_ANCHOR
{
    LEFT_TOP = TA_LEFT | TA_TOP,
    LEFT_VCENTER = TA_LEFT | TA_VCENTER,
    LEFT_BOTTOM = TA_LEFT | TA_BOTTOM,
    CENTER_TOP = TA_CENTER | TA_TOP,
    CENTER_VCENTER = TA_CENTER | TA_VCENTER,
    CENTER_BOTTOM = TA_CENTER | TA_BOTTOM,
    RIGHT_TOP = TA_RIGHT | TA_TOP,
    RIGHT_VCENTER = TA_RIGHT | TA_VCENTER,
    RIGHT_BOTTOM = TA_RIGHT | TA_BOTTOM,
};
```

这些新常量的数组在 `OnStart` 处理程序中声明。

```

void OnStart()
{
    const ENUM_TEXT_ANCHOR anchors[] =
    {
        LEFT_TOP,
        LEFT_VCENTER,
        LEFT_BOTTOM,
        CENTER_TOP,
        CENTER_VCENTER,
        CENTER_BOTTOM,
        RIGHT_TOP,
        RIGHT_VCENTER,
        RIGHT_BOTTOM,
    };
    const int na = sizeof(anchors) / sizeof(uint);
    ...

```

初始对象和资源创建类似于 *ResourceText.mq5* 中的示例，因此我们在此省略它们。最有趣的事情发生在循环中。

```

for(int i = 0; i < ExampleCount; ++i)
{
    // apply a random angle
    const int angle = Random(360);
    TextSetFont(Font, Size, 0, angle * 10);

    // take random coordinates and an anchor point
    const ENUM_TEXT_ANCHOR anchor = anchors[Random(na)];
    const int x = Random(w / 2) + w / 4;
    const int y = Random(h / 2) + h / 4;
    const color clr = ColorMix::HSVtoRGB(angle);

    // draw a circle directly in that place of the image,
    // where the anchor point goes
    TextOut(ShortToString(0x2022), x, y, TA_CENTER | TA_VCENTER, data, w, h,
        ColorToARGB(clr), COLOR_FORMAT_ARGB_NORMALIZE);

    // form the text describing the anchor type and angle
    const string text = EnumToString(anchor) +
        "(" + (string)angle + CharToString(0xB0) + ")";

    // draw text
    TextOut(text, x, y, anchor, data, w, h,
        ColorToARGB(clr), COLOR_FORMAT_ARGB_NORMALIZE);
}
...

```

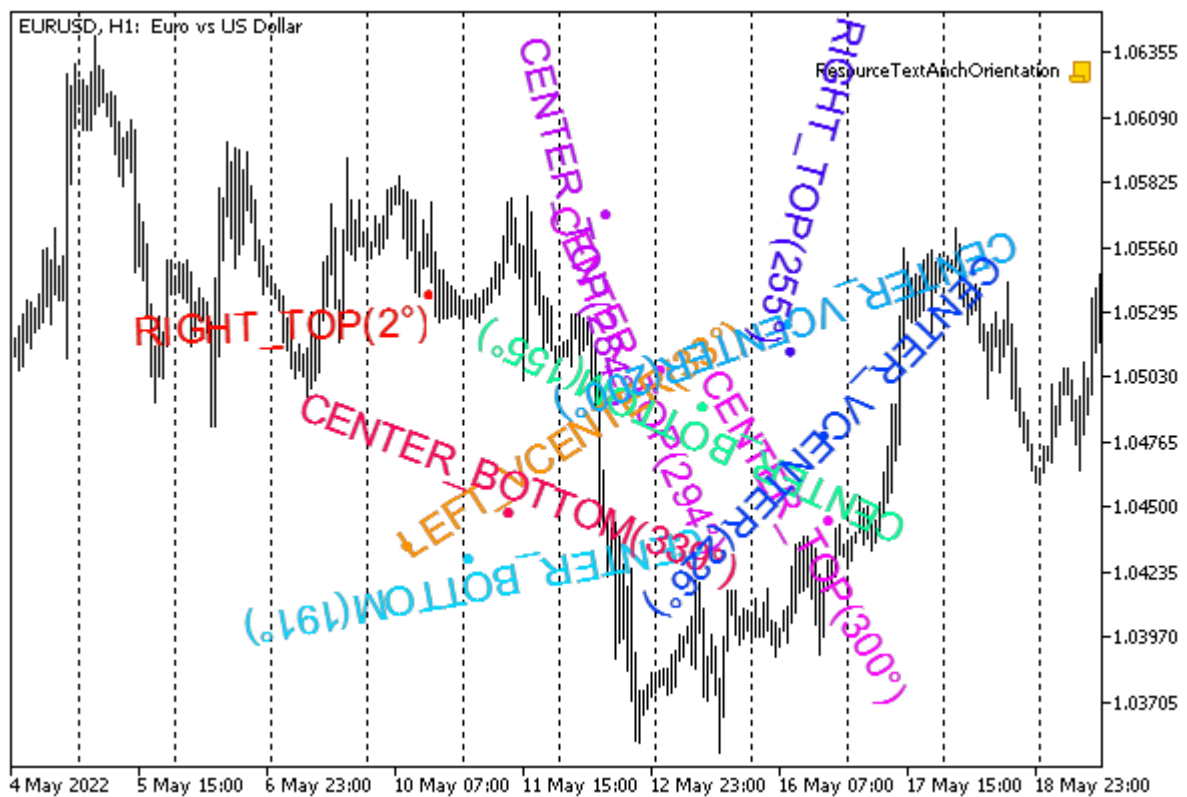
剩下的只是更新图片和图表，然后等待用户的命令并释放资源。


```
ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_NORMALIZE);
ChartRedraw();

const uint timeout = GetTickCount() + Seconds * 1000;
while(!IsStopped() && GetTickCount() < timeout)
{
    Sleep(1000);
}

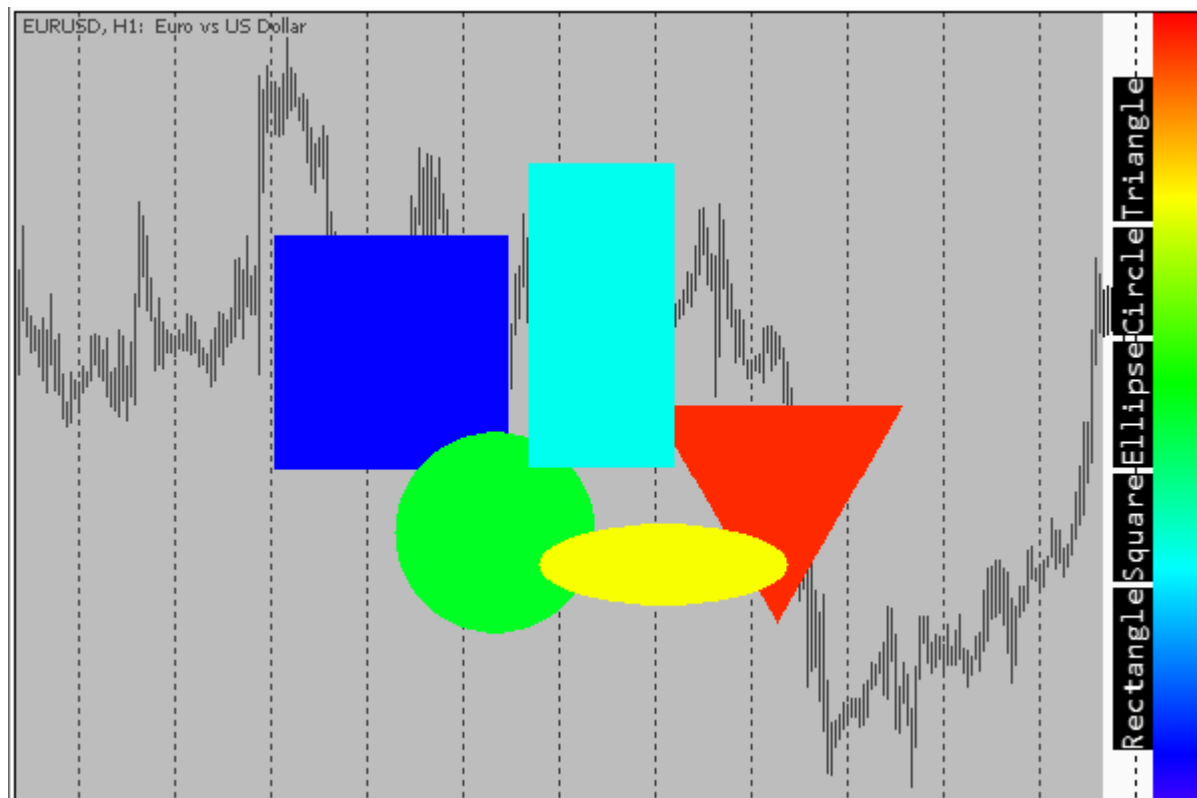
ObjectDelete(0, name);
ResourceFree("::" + name);
}
```

结果如下。



具有随机坐标、锚点和角度的文本输出

此外，为了独立学习，本书提供了一个玩具图形编辑器 `SimpleDrawing.mq5`。它被设计为一个无缓冲区的指标，并在其工作中使用先前考虑的形状类（参见 `ResourceShapesDraw.mq5`）。它们几乎没有改动地放在头文件 `ShapesDrawing.mqh` 中。以前，形状是由脚本随机生成的。现在用户可以在图表上选择并绘制它们。为此，根据注册的形状类的数量，实现了一个带有调色板和按钮栏的界面。该界面由 `SimpleDrawing` 类 (`SimpleDrawing.mqh`) 实现。



简单图形编辑器

面板和调色板可以沿图表的任何边界放置，展示了旋转标签的能力。

通过按下面板中的按钮来选择要绘制的下一个形状：按钮会“粘”在按下状态，其背景颜色表示选定的绘图颜色。要更改颜色，请点击调色板上的任意位置。

当在面板中选择其中一种形状类型时（其中一个按钮处于“活动”状态），点击绘图区域（图表的其余部分，用阴影表示）会在该位置绘制预定义大小的形状。此时，按钮“关闭”。在此状态下，当所有按钮都处于非活动状态时，你可以使用鼠标在工作区中移动形状。如果我们按住 **Ctrl**，则形状会调整大小而不是移动。“热点”位于每个形状的中心（敏感区域的大小由源代码中的宏设置，对于非常高 DPI 的显示器可能需要增加）。

请注意，编辑器在生成的资源的名称中包含绘图 ID (*ChartID*)。这允许在多个图表上并行运行编辑器。

7.1.10 在交易中应用图形资源

当然，美化并不是这些资源的主要目的。我们来看看如何基于这些资源创建一个有用的工具。我们还将消除一个疏漏：到目前为止，我们只在 `OBJ_BITMAP_LABEL` 对象中使用了资源，这些对象是以屏幕坐标定位的。然而，图形资源也可以嵌入到 `OBJ_BITMAP` 对象中，并参考报价坐标：价格和时

在本书的前面部分，我们看到了 *IndDeltaVolume.mq5* 指标，它可以计算每个柱线的 *delta* 成交量（逐笔交易明细或实际交易量）。除了这种 *delta* 成交量表示法外，还有另一种也很受用户欢迎的表示法：市场概要分析。这是在价位背景下的成交量分布。这种柱状图可以针对整个窗口、给定深度（例如一天内）或单个柱线绘制。

它是我们以新指标 *DeltaVolumeProfile.mq5* 的形式实现的最后一个选项。我们已经在上述指标的框架内考虑了逐笔交易明细历史请求的主要技术细节，因此现在我们将主要关注图形组件。

输入变量中的 *ShowSplittedDelta* 标志将控制成交量的显示方式：按买入/卖出方向分解或折叠。

```
input bool ShowSplittedDelta = true;
```

指标中没有缓冲区。它将根据用户的要求，特别是通过点击特定柱线，计算并显示该柱线的柱状图。因此，我们将使用 *OnChartEvent* 处理程序。在该处理程序中，我们获取屏幕坐标，将其重新计算为价格和时间，然后调用辅助函数 *RequestData* 开始计算。

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &str)
{
    if(id == CHARTEVENT_CLICK)
    {
        datetime time;
        double price;
        int window;
        ChartXYToTimePrice(0, (int)lparam, (int)dparam, window, time, price);
        time += PeriodSeconds() / 2;
        const int b = iBarShift(_Symbol, _Period, time, true);
        if(b != -1 && window == 0)
        {
            RequestData(b, iTime(_Symbol, _Period, b));
        }
    }
    ...
}
```

为了填充它，我们需要 *DeltaVolumeProfile* 类，该类类似于 *IndDeltaVolume.mq5* 中的 *CalcDeltaVolume* 类。

新类描述的变量考虑了成交量计算方法 (*tickType*)、构建图表基于的价格类型 (*barType*)、*ShowSplittedDelta* 输入变量的模式 (将被置于成员变量 *delta* 中)以及图表上所生成对象的前缀。

```

class DeltaVolumeProfile
{
    const COPY_TICKS tickType;
    const ENUM_SYMBOL_CHART_MODE barType;
    const bool delta;

    static const string prefix;
    ...
public:
    DeltaVolumeProfile(const COPY_TICKS type, const bool d) :
        tickType(type), delta(d),
        barType((ENUM_SYMBOL_CHART_MODE)SymbolInfoInteger(_Symbol, SYMBOL_CHART_MODE))
    {
    }

    ~DeltaVolumeProfile()
    {
        ObjectsDeleteAll(0, prefix, 0); // TODO: delete resources
    }
    ...
};

static const string DeltaVolumeProfile::prefix = "DVP";

DeltaVolumeProfile deltas(TickType, ShowSplittedDelta);

```

只有在有真实成交量的交易金融工具中，才可将 *tick type* 更改为 `TRADE_TICKS` 值。默认启用 `INFO_TICKS` 模式，该模式适用于所有金融工具。

通过 *createProfileBar* 方法请求特定柱线的逐笔交易明细。

```

int createProfileBar(const int i)
{
    MqlTick ticks[];
    const datetime time = iTime(_Symbol, _Period, i);
    // prev and next - time limits of the bar
    const datetime prev = time;
    const datetime next = prev + PeriodSeconds();
    ResetLastError();
    const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL,
        prev * 1000, next * 1000 - 1);
    if(n > -1 && _LastError == 0)
    {
        calcProfile(i, time, ticks);
    }
    else
    {
        return _LastError;
    }
    return n;
}

```

逐笔交易明细的直接分析和成交量的计算在受保护方法 *calcProfile* 中进行。在该方法中，我们首先要找出柱线的价格范围及其大小（以像素为单位）。

```

void calcProfile(const int b, const datetime time, const MqlTick &ticks[])
{
    const string name = prefix + (string)(ulong)time;
    const double high = iHigh(_Symbol, _Period, b);
    const double low = iLow(_Symbol, _Period, b);
    const double range = high - low;

    ObjectCreate(0, name, OBJ_BITMAP, 0, time, high);

    int x1, y1, x2, y2;
    ChartTimePriceToXY(0, 0, time, high, x1, y1);
    ChartTimePriceToXY(0, 0, time, low, x2, y2);

    const int h = y2 - y1 + 1;
    const int w = (int)(ChartGetInteger(0, CHART_WIDTH_IN_PIXELS)
        / ChartGetInteger(0, CHART_WIDTH_IN_BARS));
    ...
}

```

根据这些信息，我们创建一个 OBJ_BITMAP 对象，为图像分配一个数组，并创建一个资源。整幅图片的背景为空（透明）。每个对象都以其柱线 *High* 价格的中点为锚点，宽度为一个柱线。

```

uint data[];
ArrayResize(data, w * h);
ArrayInitialize(data, 0);
ResourceCreate(name + (string)ChartID(), data, w, h, 0, 0, w, COLOR_FORMAT_ARGB,

ObjectSetString(0, name, OBJPROP_BMPFILE, "::-" + name + (string)ChartID());
ObjectSetInteger(0, name, OBJPROP_XSIZE, w);
ObjectSetInteger(0, name, OBJPROP_YSIZE, h);
ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_UPPER);
...

```

然后，以传递数组的逐笔交易明细为单位计算交易量。价位数等于以像素为单位的柱线高度 (*h*)。通常情况下，它小于以点为单位的的价格范围，因此像素可以作为计算统计数据的篮子。如果在较小的时间范围内，点的范围小于以像素为单位的大小，直方图在视觉上就会显得稀疏。采购量和销售量分别累加到 *plus* 和 *minus* 数组中。

```

long plus[], minus[], max = 0;
ArrayResize(plus, h);
ArrayResize(minus, h);
ArrayInitialize(plus, 0);
ArrayInitialize(minus, 0);

const int n = ArraySize(ticks);
for(int j = 0; j < n; ++j)
{
    const double p1 = price(ticks[j]); // returns Bid or Last
    const int index = (int)((high - p1) / range * (h - 1));
    if(tickType == TRADE_TICKS)
    {
        // if real volumes are available, we can take them into account
        if((ticks[j].flags & TICK_FLAG_BUY) != 0)
        {
            plus[index] += (long)ticks[j].volume;
        }
        if((ticks[j].flags & TICK_FLAG_SELL) != 0)
        {
            minus[index] += (long)ticks[j].volume;
        }
    }
    else // tickType == INFO_TICKS or tickType == ALL_TICKS
    if(j > 0)
    {
        // if there are no real volumes,
        // price movement up/down is an estimate of the volume type
        if((ticks[j].flags & (TICK_FLAG_ASK | TICK_FLAG_BID)) != 0)
        {
            const double d = (((ticks[j].ask + ticks[j].bid)
                               - (ticks[j - 1].ask + ticks[j - 1].bid)) / _Point);
            if(d > 0) plus[index] += (long)d;
            else minus[index] -= (long)d;
        }
    }
}
...

```

为使直方图正常化，我们会求出最大值。

```

    if(delta)
    {
        if(plus[index] > max) max = plus[index];
        if(minus[index] > max) max = minus[index];
    }
    else
    {
        if(fabs(plus[index] - minus[index]) > max)
            max = fabs(plus[index] - minus[index]);
    }
}
...

```

最后，生成的统计数据将输出到图形缓冲区 *data* 并发送到资源。买入量显示为蓝色，卖出量显示为红色。如果启用了净值模式，则金额显示为绿色。

```

for(int i = 0; i < h; i++)
{
    if(delta)
    {
        const int dp = (int)(plus[i] * w / 2 / max);
        const int dm = (int)(minus[i] * w / 2 / max);
        for(int j = 0; j < dp; j++)
        {
            data[i * w + w / 2 + j] = ColorToARGB(clrBlue);
        }
        for(int j = 0; j < dm; j++)
        {
            data[i * w + w / 2 - j] = ColorToARGB(clrRed);
        }
    }
    else
    {
        const int d = (int)((plus[i] - minus[i]) * w / 2 / max);
        const int sign = d > 0 ? +1 : -1;
        for(int j = 0; j < fabs(d); j++)
        {
            data[i * w + w / 2 + j * sign] = ColorToARGB(clrGreen);
        }
    }
}
ResourceCreate(name + (string)ChartID(), data, w, h, 0, 0, w, COLOR_FORMAT_ARGB
}

```

现在我们可以返回 *RequestData* 函数：它的任务是调用 *createProfileBar* 方法并处理错误（如果有）。


```

void RequestData(const int b, const datetime time, const int count = 0)
{
    Comment("Requesting ticks for ", time);
    if(deltas.createProfileBar(b) <= 0)
    {
        Print("No data on bar ", b, ", at ", TimeToString(time),
            ". Sending event for refresh...");
        ChartSetSymbolPeriod(0, _Symbol, _Period); // request to update the chart
        EventChartCustom(0, TRY_AGAIN, b, count + 1, NULL);
    }
    Comment("");
}

```

唯一的错误处理策略是再次请求逐笔交易明细，因为它们可能还来不及加载。为此，函数会向图表发送一条自定义 TRY_AGAIN 消息，并自行处理。

```

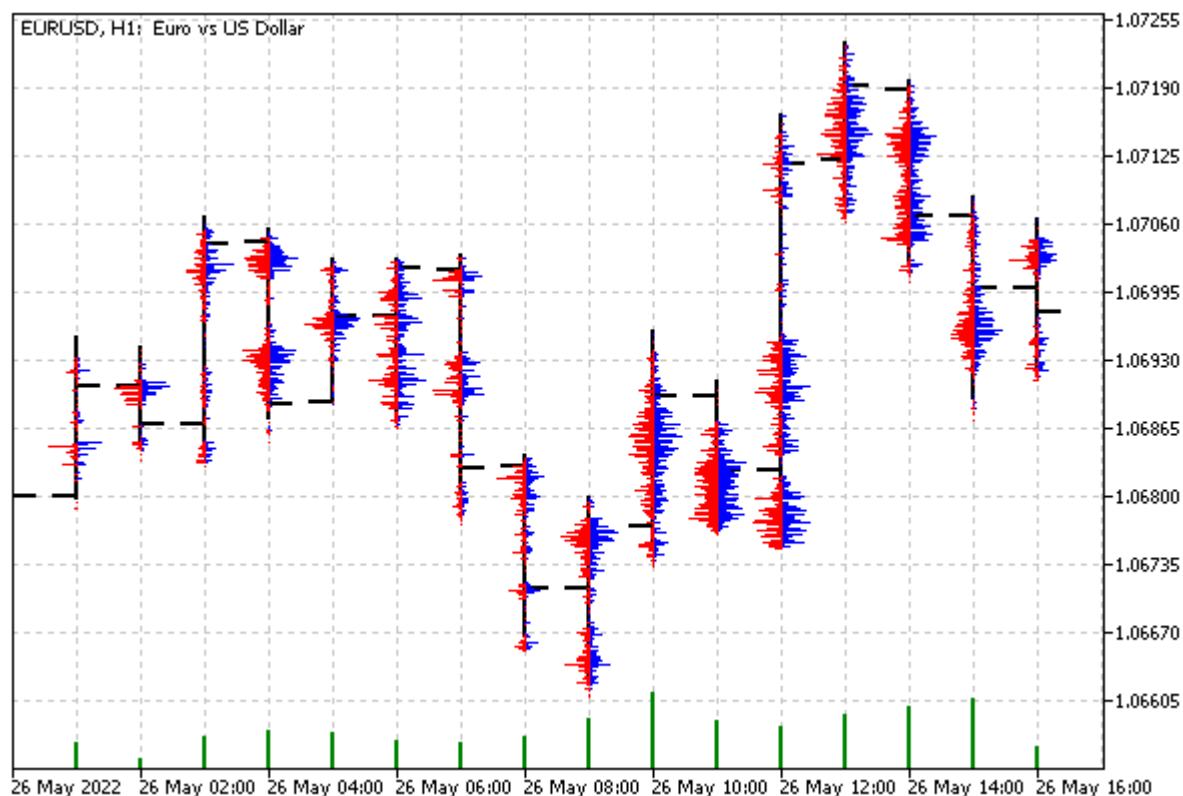
void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &str)
{
    ...
    else if(id == CHARTEVENT_CUSTOM + TRY_AGAIN)
    {
        Print("Refreshing... ", (int)dparam);
        const int b = (int)lparam;
        if((int)dparam < 5)
        {
            RequestData(b, iTime(_Symbol, _Period, b), (int)dparam);
        }
        else
        {
            Print("Give up. Check tick history manually, please, then click the bar again");
        }
    }
}

```

我们重复此过程的次数不超过 5 次，因为逐笔交易明细历史记录深度有限，无缘无故加载计算机毫无意义。

DeltaVolumeProfile 类还具有处理 CHARTEVENT_CHART_CHANGE 消息的机制，以便在改变图表大小或比例时重绘现有对象。详情请参见源代码。

该指标的结果如下图所示。



在图形资源中显示单独成交量的每柱线柱状图

请注意，绘制指标后不会立即显示柱状图：必须点击柱线才能计算其柱状图。

7.2 自定义交易品种

MetaTrader 5 的一个有趣的技术特性是支持自定义金融工具。这些交易品种不是由经纪商在服务器端定义的，而是由交易者直接在终端中定义的。

自定义交易品种可以添加到 **Market Watch** 列表中，与标准交易品种并排显示。带有这些交易品种的图表可以按常规方式使用。

创建自定义交易品种最简单的方法是在相应的特性中指定其计算公式。为此，从终端界面中，在 **Market Watch** 窗口中调用上下文菜单，执行 **Symbols** 命令，转到交易品种层级结构及其 **Custom** 分支，然后按 **Create symbol** 按钮。结果，将打开一个用于设置新交易品种特性的对话框。在同一位置，你可以从文件将外部分时报价历史 (**Ticks** 选项卡) 或行情 (**Bars** 选项卡) 导入到类似的工具中。这在 [MetaTrader 5 文档](#) 中有详细讨论。

然而，MQL5 API 提供了对自定义交易品种最完整的控制。

对于自定义交易品种，API 提供了一组处理 [金融工具](#) 和 [市场观察](#) 的函数。特别是，可以使用标准函数 (例如 **SymbolsTotal**、**SymbolName** 和 **SymbolInfo**) 从程序中列出此类交易品种。我们已经在 [自定义交易品种特性](#) 一节中简要地提到了这种可能性并提供了一个示例。自定义交易品种的一个显著特征是启用了 **SYMBOL_CUSTOM** 标志 (特性)。

使用内置函数，你可以拼接期货，生成具有指定特征的随机时间序列，模拟砖形图、等距柱线、等量图以及其他非标准类型的图表 (例如，秒级时间范围)。此外，与导入静态文件不同，软件控制的自定义交易品种可以基于来自加密货币交易所等 [Web 服务](#) 的数据实时生成。关于将 MQL 程序与 [Web](#) 集成的讨论尚未展开，但这种可能性不容忽视。

自定义交易品种可以轻松地用于在测试程序中测试策略或作为技术分析的附加方法。然而，这项技术有其局限性。

由于自定义交易品种是在终端中定义的，而不是在服务器上定义的，因此无法在线交易。特别是，如果你创建了一个砖形图，则基于它的交易策略需要以某种方式进行调整，以便交易信号和交易实际上由不同的交易品种分隔：人工用户交易品种和真实经纪商交易品种。我们将研究几个[解决该问题的方法](#)。

此外，由于平台中一个时间范围的所有柱线的持续时间都相同，因此任何具有不同周期的柱线（砖形图、等量图等）的模拟通常都基于可用的 M1 时间范围中较小的一个，并且不提供与现实的完全时间同步。换句话说，必须强制属于此类柱线的分时报价具有 60 秒内的人工时间，即使砖形图“砖块”或给定交易量的柱线实际上需要更多时间来形成。否则，如果我们将分时报价置于实时状态，它们将形成下一个 M1 柱线，从而违反砖形图或等量图的规则。此外，在某些情况下，砖形图“砖块”或其他人工柱线应以小于前一个柱线 1 分钟的时间间隔创建（例如，当波动性快速增加时）。在这种情况下，将有必要更改自定义金融工具行情中历史柱线的时间（追溯性地“将它们向左移动”或在新柱线上设置未来的时间（这是非常不可取的）。在用户定义交易品种技术的框架内，无法以通用方式解决此问题。

7.2.1 创建和删除自定义交易品种

要使用自定义交易品种，首先需要的两个函数是 *CustomSymbolCreate* 和 *CustomSymbolDelete*。

bool CustomSymbolCreate(const string name, const string path = "", const string origin = NULL)

该函数在指定的组（*path*）中创建具有指定名称（*name*）的自定义交易品种，如果需要，还可以使其具有示例交易品种的特性——示例交易品种的名称可以在 *origin* 参数中指定。

name 参数应该是一个简单的标识符，不包含层级结构。如有必要，应在参数 *path* 中指定一个或多个必需的组（子文件夹）级别，分隔符为反斜杠“\”（此处不支持正斜杠，与文件系统不同）。在文字字符串中，必须使用双反斜杠“\\”。

默认情况下，如果 *path* 字符串为空（"" 或 NULL），则交易品种直接在 *Custom* 文件夹中创建，该文件夹在交易品种的通用层级结构中为用户交易品种分配。如果已填充路径，则会在 *Custom* 文件夹按照完整深度创建该路径（如果尚不存在相应的文件夹）。

交易品种的名称以及任何级别的组名都可以包含拉丁字母和数字，不能包含标点符号、空格和特殊字符。此外，仅允许使用“.”、“_”、“&”和“#”。

无论交易品种应该在哪个组中创建，其名称在整个交易品种层级结构中都必须都是唯一的。如果已存在同名交易品种，该函数将返回 *false*，并在 *_LastError* 中设置错误代码 5300 (ERR_NOT_CUSTOM_SYMBOL) 或 5304 (ERR_CUSTOM_SYMBOL_EXIST)。

请注意，如果 *path* 字符串中层级结构的最后一个（甚至唯一一个）元素与 *name* 完全匹配（区分大小写），则它被视为路径一部分的交易品种名称，而不是文件夹。例如，如果名称和路径分别包含字符串“Example”和“MQL5Book\Example”，则交易品种“Example”将在“Custom\MQL5Book\”文件夹中创建。同时，如果我们将名称更改为“example”，我们将在“Custom\MQL5Book\Example”文件夹中得到“example”交易品种。

此特性还有另一个后果。SYMBOL_PATH 特性返回的路径末尾会包含交易品种名称。因此，如果我们将某个示例交易品种的值未经更改地传递给新创建的交易品种，我们将得到以下效果：将创建一个以旧交易品种名称命名的文件夹，其中将出现一个新的交易品种。因此，如果你想在与原始交易品种相同的组中创建自定义交易品种，则必须从 SYMBOL_PATH 特性获取的字符串中剥离原始交易品种的名称。

我们将在下一节的示例中演示复制 `SYMBOL_PATH` 特性的副作用。然而，这种效果也可以作为积极的一面来使用。特别是，通过基于一个原始交易品种创建多个其自定义交易品种，复制 `SYMBOL_PATH` 将确保所有新交易品种都放置在以原始交易品种名称命名的文件夹中，即，它将根据其原型交易品种对交易品种进行分组。

自定义交易品种的 `SYMBOL_PATH` 特性始终以 "Custom \" 文件夹开头 (会自动添加此前缀)。

名称长度限制为 31 个字符。超过限制时，`CustomSymbolCreate` 将返回 `false` 并设置错误代码 5302 (ERR_CUSTOM_SYMBOL_NAME_LONG)。

参数 `path` 的最大长度为 127 个字符，包括 "Custom \"、组分隔符 "\" 以及交易品种名称 (如果在末尾指定)。

`origin` 参数允许你选择性地指定一个交易品种的名称，将从该交易品种中复制所创建自定义交易品种的特性。创建自定义交易品种后，你可以使用适当的函数 (请参见 `CustomSymbolSet` 函数) 将其任何特性更改为所需的值。

如果将不存在的交易品种作为 `origin` 参数给出，则自定义交易品种在创建后为空白，就好像未指定 `origin` 参数一样。这将引发错误 4301 (ERR_MARKET_UNKNOWN_SYMBOL)。

在以空白方式创建的新交易品种中，所有特性都设置为其默认值。例如，合约大小为 100000，价格中的位数是 4，保证金计算根据外汇规则进行，图表基于 *Bid* 价。

当你指定 `origin` 时，仅从该交易品种向新交易品种传递设置，而不传递报价或分时报价，因为它们应单独生成。这将在后续各节中介绍。

交易品种在创建后不会自动添加到 *Market Watch*。因此，必须以明确方式 (手动或以编程方式) 添加。若没有报价，图表窗口将为空。

`bool CustomSymbolDelete(const string name)`

该函数删除具有指定名称的自定义交易品种。不仅删除设置，还删除交易品种上的所有数据 (行情和分时报价)。值得注意的是，历史记录不会立即删除，而是在一段延迟之后才删除，如果你打算重新创建同名交易品种，这可能会成为问题的根源 (我们将在 *添加、替换和删除报价* 一节的示例中讨论这一点)。

只能删除自定义交易品种。此外，你不能删除 *Market Watch* 选定的交易品种或具有打开图表的交易品种。请注意，交易品种也可以 *隐式* 选择，而无需显示在可见列表中 (在这种情况下，`SYMBOL_VISIBLE` 特性为 `false`，而 `SYMBOL_SELECT` 特性为 `true`)。在尝试删除之前，此类交易品种必须首先通过调用 `SymbolSelect("name", false)` 来“隐藏”：否则，我们会收到 `CUSTOM_SYMBOL_SELECTED` (5306) 错误。

如果删除交易品种后留下空文件夹 (或文件夹层级结构)，空文件夹也会被删除。

例如，我们创建一个简单的脚本 `CustomSymbolCreateDelete.mq5`。在输入参数中，你可以指定名称、路径和示例交易品种。

```
input string CustomSymbol = "Dummy";           // Custom Symbol Name
input string CustomPath = "MQL5Book\\Part7";   // Custom Symbol Folder
input string Origin;
```

在 *OnStart* 处理程序中，让我们检查是否已存在具有给定名称的交易品种。如果没有，则在用户确认后，我们将创建这样一个交易品种。如果交易品种已经存在并且是自定义交易品种，则在用户允许的情况下将其删除 (这将有助于在实验结束后进行清理)。

```

void OnStart()
{
    bool custom = false;
    if(!PRTF(SymbolExist(CustomSymbol, custom)))
    {
        if(IDYES == MessageBox("Create new custom symbol?", "Please, confirm", MB_YESNC
        {
            PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, Origin));
        }
    }
    else
    {
        if(custom)
        {
            if(IDYES == MessageBox("Delete existing custom symbol?", "Please, confirm",
            {
                PRTF(CustomSymbolDelete(CustomSymbol));
            }
        }
        else
        {
            Print("Can't delete non-custom symbol");
        }
    }
}
}

```

使用默认选项连续运行两次应该会生成以下日志条目。

```

SymbolExist(CustomSymbol,custom)=false / ok
Create new custom symbol?
CustomSymbolCreate(CustomSymbol,CustomPath,Origin)=true / ok

SymbolExist(CustomSymbol,custom)=true / ok
Delete existing custom symbol?
CustomSymbolDelete(CustomSymbol)=true / ok

```

在两次运行之间，你可以在终端中打开交易品种对话框，并检查相应的自定义交易品种是否已出现在交易品种层级结构中。

7.2.2 自定义交易品种特性

自定义交易品种与经纪商提供的交易品种具有相同的特性。这些特性通过 [金融工具一章](#)中讨论的标准函数读取。

自定义交易品种的特性可以通过一组特殊的 *CustomSymbolSet* 函数来设置，每种基本类型（整数、实数、字符串）对应一个函数。

```
bool CustomSymbolSetInteger(const string name, ENUM_SYMBOL_INFO_INTEGER property, long value)
```

```
bool CustomSymbolSetDouble(const string name, ENUM_SYMBOL_INFO_DOUBLE property, double value)
```

```
bool CustomSymbolSetString(const string name, ENUM_SYMBOL_INFO_STRING property, string value)
```

这些函数名为 *name* 的自定义交易品种设置 *property* 的值为 *value*。所有现有特性都分组到枚举 `ENUM_SYMBOL_INFO_INTEGER`、`ENUM_SYMBOL_INFO_DOUBLE`、`ENUM_SYMBOL_INFO_STRING` 中，这些枚举已在上述章节的各节中逐元素讨论过。

该函数返回操作状态标识：成功 (*true*) 或错误 (*false*)。导致错误的一个可能问题是，并非所有特性都允许更改。当尝试设置只读特性时，我们会收到错误

`CUSTOM_SYMBOL_PROPERTY_WRONG` (5307)。如果你尝试向特性写入无效值，则会收到 `CUSTOM_SYMBOL_PARAMETER_ERROR` (5308) 错误。

请注意，如果以下任何特性在交易品种规范中发生更改，则自定义交易品种的分钟和分时报价历史记录将被完全删除：

- `SYMBOL_CHART_MODE` -用于构建柱线的价格类型 (*Bid* 或 *Last*)
- `SYMBOL_DIGITS` -价格值中的小数位数
- `SYMBOL_POINT` -一个点的价值
- `SYMBOL_TRADE_TICK_SIZE` -一项分时报价的价值，允许的最小价格变动
- `SYMBOL_TRADE_TICK_VALUE` - 每项分时报价的价格变动成本 (另请参见 `SYMBOL_TRADE_TICK_VALUE_PROFIT`、`SYMBOL_TRADE_TICK_VALUE_LOSS`)
- `SYMBOL_FORMULA` -价格计算公式

如果自定义交易品种是按公式计算的，则在删除其历史后，终端将自动尝试使用更新的特性创建新的历史记录。但是，对于以编程方式生成的交易品种，MQL 程序本身必须负责重新计算。

对于修改先前创建的自定义交易品种 (在 [CustomSymbolCreate](#) 函数中指定第三个参数 *origin* 之后)，编辑单个特性的需求最为迫切。

在其他情况下，批量更改特性可能会导致细微的影响。关键在于特性在内部是相互关联的，更改其中一个特性可能需要其他特性处于特定状态才能成功完成操作。此外，设置某些特性会导致其他特性自动更改。

在最简单的示例中，设置 `SYMBOL_DIGITS` 特性后，你会发现 `SYMBOL_POINT` 特性也已更改。下面是一个不太明显的例子：分配 `SYMBOL_CURRENCY_MARGIN` 或 `SYMBOL_CURRENCY_PROFIT` 对外汇交易品种没有影响，因为系统假定货币名称分别占据名称的前 3 个和后 3 个字母 ("XXXXYY[suffix]"). 请注意，在创建“空白”交易品种后，默认被视为外汇交易品种，因此在未首先更改市场的情况下无法为其设置这些特性。

复制或设置交易品种特性时，请注意平台隐含了一些特殊性。特别是，`SYMBOL_TRADE_CALC_MODE` 特性的默认值为 0 (在交易品种创建后，但在设置任何特性之前)，而 `ENUM_SYMBOL_CALC_MODE` 枚举中的 0 对应于 `SYMBOL_CALC_MODE_FOREX` 成员。同时，对于形式为 XXXYYY (其中 XXX 和 YYY 是货币代码) 加上可选后缀的外汇交易品种，隐含了特殊的命名规则。因此，如果你没有预先将 `SYMBOL_TRADE_CALC_MODE` 更改为其他所需模式，则指定交易品种名称的子字符串 (第一和第二个三联符号) 将自动归入基础货币 (`SYMBOL_CURRENCY_BASE`) 和利润货币 (`SYMBOL_CURRENCY_PROFIT`) 的特性中。例如，如果你指定名称 "Dum m y"，它将被拆分为两个伪货币 "Dum" 和 "m y"。

另一个细微差别是，在设置精度为 N 位小数的 `SYMBOL_POINT` 值之前，你需要确保 `SYMBOL_DIGITS` 至少为 N。

本书随附了脚本 *CustomSymbolProperties.mq5*，它允许你试验创建当前图表交易品种的副本，并在实践中研究由此产生的效果。特别是，你可以选择交易品种的名称、其路径以及遍历（设置）所有支持特性的方向，即在语言中特性编号方面的正向或反向。该脚本使用一个特殊的类 *CustomSymbolMonitor*，它是上述内置函数的包装器：我们 [稍后](#) 将对其进行描述。

7.2.3 设置保证金率

之前，我们研究了 *SymbolInfoMarginRate* 函数，该函数返回经纪商为每个交易品种设定的保证金率。对于自定义交易品种，我们可以使用 *CustomSymbolSetMarginRate* 函数自由设置这些费率。

```
bool CustomSymbolSetMarginRate(const string name, ENUM_ORDER_TYPE orderType, double initial, double maintenance)
```

该函数根据订单的类型和方向（根据 *orderType* 值，在 *ENUM_ORDER_TYPE* 枚举中提供）设置保证金率。用于计算初始保证金和维持保证金（已开仓或现有仓位每手所需的抵押品）的费率分别在 *initial* 和 *maintenance* 参数中传递。

最终的保证金金额是根据 [保证金要求](#) 一节中描述的几个交易品种特性（*SYMBOL_TRADE_CALC_MODE*、*SYMBOL_MARGIN_INITIAL*、*SYMBOL_MARGIN_MAINTENANCE* 等）确定的，因此如果需要，也应在自定义交易品种上设置这些特性。

函数将返回成功 (*true*) 或错误 (*false*) 的指标。

借助此函数和与保证金计算相关的特性，你可以模拟由于某种原因无法使用的服务器的交易条件，并在测试程序中调试你的 MQL 程序。

7.2.4 配置报价和交易时段

两个 API 函数允许设置自定义金融工具的报价和交易时段。这两个概念已在 [交易和报价时段的时间表](#) 一节中讨论过。

```
bool CustomSymbolSetSessionQuote(const string name, ENUM_DAY_OF_WEEK dayOfWeek, uint sessionIndex, datetime from, datetime to)
```

```
bool CustomSymbolSetSessionTrade(const string name, ENUM_DAY_OF_WEEK dayOfWeek, uint sessionIndex, datetime from, datetime to)
```

CustomSymbolSetSessionQuote 为特定星期几 (*dayOfWeek*) 设置由数字 (*sessionIndex*) 指定的报价时段的开始和结束时间。*CustomSymbolSetSessionTrade* 对交易时段执行相同的操作。

时段编号从 0 开始。

时段只能按顺序添加，也就是说，只有当索引为 0 的时段已存在时，才能添加索引为 1 的时段。如果违反此规则，则不会创建新时段，并且函数将返回 *false*。

from 和 *to* 参数中的日期值以秒为单位测量，并且 *from* 应小于 *to*。范围限制为两天，从 0 00 时 00 分 00 秒到 172800（第二天 23 时 59 分 59 秒）。需要更改日期是为了能够指定在午夜之前开始并在午夜之后结束的时段。这种情况往往发生在交易所位于相对于经纪商（交易商）服务器的地球另一端时。

如果为 *sessionIndex* 时段传递了零开始和结束参数 (*from = 0* 且 *to = 0*)，则该时段将被删除，并且后续时段（如果有）的编号将向下移动。

交易时段不能超出报价时段。

例如，我们可以通过移动日内报价时间和时段表来为不同的时区创建金融工具的副本，以便在不同条件下调试机器人，就像对待任何特殊经纪商一样。

7.2.5 添加、替换和删除报价

自定义交易品种通过以下两个内置函数填充报价：*CustomRatesUpdate* 和 *CustomRatesReplace*。除了交易品种的名称外，两者都期望输入一个 M1 时间范围的 *MqlRates* 结构体数组（更高的时间范围会自动从 M1 完成）。*CustomRatesReplace* 还有一对额外的参数（*from* 和 *to*），用于定义历史记录编辑受限的时间范围。

```
int CustomRatesUpdate(const string symbol, const MqlRates &rates[], uint count = WHOLE_ARRAY)

int CustomRatesReplace(const string symbol, datetime from, datetime to, const MqlRates &rates[],
uint count = WHOLE_ARRAY)
```

CustomRatesUpdate 将缺失的柱线添加到历史记录中，并用数组中的数据替换现有的匹配柱线。

CustomRatesReplace 用数组中的数据完全替换指定时间间隔内的历史记录。

这些函数的区别在于预期应用场景的不同。下表更详细地列出了这些差异。

CustomRatesUpdate	CustomRatesReplace
将传递的 <i>MqlRates</i> 数组的元素应用于历史记录，无论其时间戳如何	仅应用传递的 <i>MqlRates</i> 数组中落在指定范围内的那些元素
保留历史记录中那些在函数调用之前已经存在且与数组中柱线时间不一致的 M1 柱线	范围之外的所有历史记录保持不变
当时间戳匹配时，用数组中的柱线替换现有的历史柱线	完全删除指定范围内的现有历史柱线
如果与旧柱线没有匹配项，则将数组中的元素作为“新”柱线插入	将数组中落在相关范围内的柱线插入到指定的历史记录范围中

rates 数组中的数据必须由有效的 OHLC 价格表示，并且柱线开盘时间不能包含秒。

from 和 *to* 之间的时间间隔是包含性的：*from* 等于要处理的第一个柱线的时间，*to* 等于最后一个柱线的时间。

下图更清楚地说明了这些规则。每个柱线的唯一时间戳都用其自己的拉丁字母表示。历史记录中可用的柱线用大写字母显示，而数组中的柱线用小写字母显示。字符 ‘*u*’ 表示相应时间的历史记录或数组中的间隔符。

History	ABC-EFGHIJKLMN-PQRST-----	B
Array	-----hijk--nopqrstuvwxyz	A
Result of CustomRatesUpdate	ABC-EFGhijkLMnopqrstuvwxyz	R
Result of CustomRatesReplace	ABC-E--hijk--nopqrstuvwxyz---	S
	^ ^ from to TIME	

可选参数 *count* 设置应使用的 *rates* 数组中元素的数量（其他元素将被忽略）。这允许你部分处理传递的数组。默认值 `WHOLE_ARRAY` 表示整个数组。

可以使用 *CustomRatesDelete* 函数完全或部分删除自定义交易品种的报价历史记录。

int CustomRatesDelete(const string symbol, datetime from, datetime to)

在这里，参数 *from* 和 *to* 也设置了被移除柱线的时间范围。要覆盖整个历史记录，请指定 `0` 和 `LONG_MAX`。

所有三个函数都返回已处理柱线的数量：更新或删除。如果发生错误，结果为 `-1`。

应该注意的是，自定义交易品种的报价不仅可以通过添加现成的柱线来形成，还可以通过分时报价数组甚至单个分时报价序列来形成。相关函数将在 [下一节中](#) 介绍。当添加分时报价时，终端将自动基于它们计算柱线。这些方法之间的区别在于，自定义分时报价历史允许你在“真实”分时报价模式下测试 MQL 程序，而柱线历史记录只会迫使你要么将自己限制在 OHLC M1 或开盘价模式，要么依赖于测试器实现的分时报价模拟。

此外，通过逐个添加分时报价，你可以在自定义交易品种的图表上模拟标准的 *OnTick* 和 *OnCalculate* 事件，这会使图表实现动画效果，类似于在线可用的工具，并在 MQL 程序绘制在图表上时启动相应的处理函数。但我们将在下一节讨论这个问题。

作为使用新函数的一个示例，我们考虑脚本 *CustomSymbolRandomRates.mq5*。它旨在按照“随机游走”或对现有报价进行“噪声处理”的原则生成随机报价。后者可用于评估 EA 交易的稳定性。

为了检查报价形成的正确性，我们还将支持一种模式，在该模式下创建原始金融工具的完整副本，并在其图表上启动脚本。

所有模式都收集在 `RANDOMIZATION` 枚举中。

```
enum RANDOMIZATION
{
    ORIGINAL,
    RANDOM_WALK,
    FUZZY_WEAK,
    FUZZY_STRONG,
};
```

我们用两种强度级别来实现报价噪声：弱和强。

在输入参数中，除了模式之外，你还可以选择交易品种层级结构中的文件夹、日期范围以及用于初始化随机生成器的数字（以便能够重现结果）。

```

input string CustomPath = "MQL5Book\\Part7";    // Custom Symbol Folder
input RANDOMIZATION RandomFactor = RANDOM_WALK;
input datetime _From;                          // From (default: 120 days ago)
input datetime _To;                            // To (default: current time)
input uint RandomSeed = 0;

```

默认情况下，如果未指定日期，脚本会生成最近 120 天的报价。*RandomSeed* 参数中的值 0 表示随机初始化。

交易品种的名称是根据当前图表的交易品种和选定的设置生成的。

```

const string CustomSymbol = _Symbol + "." + EnumToString(RandomFactor)
    + (RandomSeed ? "_" + (string)RandomSeed : "");

```

在 *OnStart* 的开头，我们将准备并检查数据。

```

datetime From;
datetime To;

void OnStart()
{
    From = _From == 0 ? TimeCurrent() - 60 * 60 * 24 * 120 : _From;
    To = _To == 0 ? TimeCurrent() / 60 * 60 : _To;
    if(From > To)
    {
        Alert("Date range must include From <= To");
        return;
    }

    if(RandomSeed != 0) MathSrand(RandomSeed);
    ...
}

```

由于脚本很可能需要运行多次，因此我们将提供删除先前创建的自定义交易品种的功能，并向用户请求二次确认。

```

bool custom = false;
if(PRTF(SymbolExist(CustomSymbol, custom)) && custom)
{
    if(IDYES == MessageBox(StringFormat("Delete custom symbol '%s'?", CustomSymbol)
        "Please, confirm", MB_YESNO))
    {
        if(CloseChartsForSymbol(CustomSymbol))
        {
            Sleep(500); // wait for the changes to take effect (opportunistically)
            PRTF(CustomRatesDelete(CustomSymbol, 0, LONG_MAX));
            PRTF(SymbolSelect(CustomSymbol, false));
            PRTF(CustomSymbolDelete(CustomSymbol));
        }
    }
}
...

```

辅助函数 *CloseChartsForSymbol* 此处未显示 (有兴趣的可以查看附加的源代码): 其目的是查看打开的图表列表并关闭那些工作交易品种是被删除的自定义交易品种的图表 (否则, 删除将无法进行)。

更重要的是, 要注意调用具有完整日期范围的 *CustomRatesDelete*。如果不这样做, 先前用户交易品种的数据将在历史数据库 (文件夹 *bases/Custom/history/<symbol-name>*) 的磁盘上保留一段时间。换句话说, 上面最后一行显示的 *CustomSymbolDelete* 调用不足以实际从终端清除自定义交易品种。

如果用户决定立即再次创建同名交易品种 (我们在下面的代码中提供了这样的机会), 那么旧的报价可能会混入新的报价中。

此外, 在用户确认后, 将启动生成报价的过程。这是由 *GenerateQuotes* 函数完成的 (见下文)。

```

if(IDYES == MessageBox(StringFormat("Create new custom symbol '%s'?", CustomSymbol
    "Please, confirm", MB_YESNO))
{
    if(PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
    {
        if(RandomFactor == RANDOM_WALK)
        {
            CustomSymbolSetInteger(CustomSymbol, SYMBOL_DIGITS, 8);

            CustomSymbolSetString(CustomSymbol, SYMBOL_DESCRIPTION, "Randomized quotes")

            const int n = GenerateQuotes();
            Print("Bars M1 generated: ", n);
            if(n > 0)
            {
                SymbolSelect(CustomSymbol, true);
                ChartOpen(CustomSymbol, PERIOD_M1);
            }
        }
    }
}

```

如果成功，新创建的交易品种将在 **Market Watch** 中被选中，并为其打开一个图表。在此过程中，演示了如何设置一对特性：SYMBOL_DIGITS 和 SYMBOL_DESCRIPTION。

在 **GenerateQuotes** 函数中，除了 RANDOM_WALK 模式外，都必须请求原始交易品种的报价。

```

int GenerateQuotes()
{
    MqlRates rates[];
    MqlRates zero = {};
    datetime start;    // time of the current bar
    double price;      // last closing price

    if(RandomFactor != RANDOM_WALK)
    {
        if(PRTF(CopyRates(_Symbol, PERIOD_M1, From, To, rates)) <= 0)
        {
            return 0; // error
        }
        if(RandomFactor == ORIGINAL)
        {
            return PRTF(CustomRatesReplace(CustomSymbol, From, To, rates));
        }
        ...
    }
}

```

务必要记住，**CopyRates** 受图表上柱线数量限制的影响，该限制在终端设置中设置。

在 ORIGINAL 模式下，我们只需将生成的数组 **rates** 转发到 **CustomRatesReplace** 函数中。对于噪声模式，我们将专门选择的 **price** 和 **start** 变量设置为第一个柱线的初始价格和时间值。

```

        price = rates[0].open;
        start = rates[0].time;
    }
    ...

```

在随机游走模式下，不需要报价，因此我们只需为将来的随机 M1 柱线分配 *rates* 数组。

```

else
{
    ArrayResize(rates, (int)((To - From) / 60) + 1);
    price = 1.0;
    start = From;
}
...

```

接下来，在遍历 *rates* 数组的循环中，将随机值添加到原始交易品种的噪声价格中或“按原样”添加。在 RANDOM_WALK 模式下，我们自己负责增加变量 *start* 中的时间。在其他模式下，时间已经包含在初始报价中。

```

const int size = ArraySize(rates);

double hlc[3]; // future High Low Close (in unknown order)
for(int i = 0; i < size; ++i)
{
    if(RandomFactor == RANDOM_WALK)
    {
        rates[i] = zero;           // zeroing the structure
        rates[i].time = start += 60; // plus a minute to the last bar
        rates[i].open = price;      // start from the last price
        hlc[0] = RandomWalk(price);
        hlc[1] = RandomWalk(price);
        hlc[2] = RandomWalk(price);
    }
    else
    {
        double delta = 0;
        if(i > 0)
        {
            delta = rates[i].open - price; // cumulative correction
        }
        rates[i].open = price;
        hlc[0] = RandomWalk(rates[i].high - delta);
        hlc[1] = RandomWalk(rates[i].low - delta);
        hlc[2] = RandomWalk(rates[i].close - delta);
    }
    ArraySort(hlc);

    rates[i].high = fmax(hlc[2], rates[i].open);
    rates[i].low = fmin(hlc[0], rates[i].open);
    rates[i].close = price = hlc[1];
    rates[i].tick_volume = 4;
}
...

```

基于最后一个柱线的收盘价，生成 3 个随机值（使用 *RandomWalk* 函数）。它们的最大值和最小值分别成为新柱线的 *High* 和 *Low* 价。平均值是 *Close* 价。

在循环结束时，我们将数组传递给 *CustomRatesReplace*。

```

return PRTF(CustomRatesReplace(CustomSymbol, From, To, rates));
}

```

在 *RandomWalk* 函数中，尝试模拟具有宽尾部分布的情况，这是真实报价的典型情况。

```
double RandomWalk(const double p)
{
    const static double factor[] = {0.0, 0.1, 0.01, 0.05};
    const static double f = factor[RandomFactor] / 100;
    const double r = (rand() - 16383.0) / 16384.0; // [-1,+1]
    const int sign = r >= 0 ? +1 : -1;
    if(r != 0)
    {
        return p + p * sign * f * sqrt(-log(sqrt(fabs(r))));
    }
    return p;
}
```

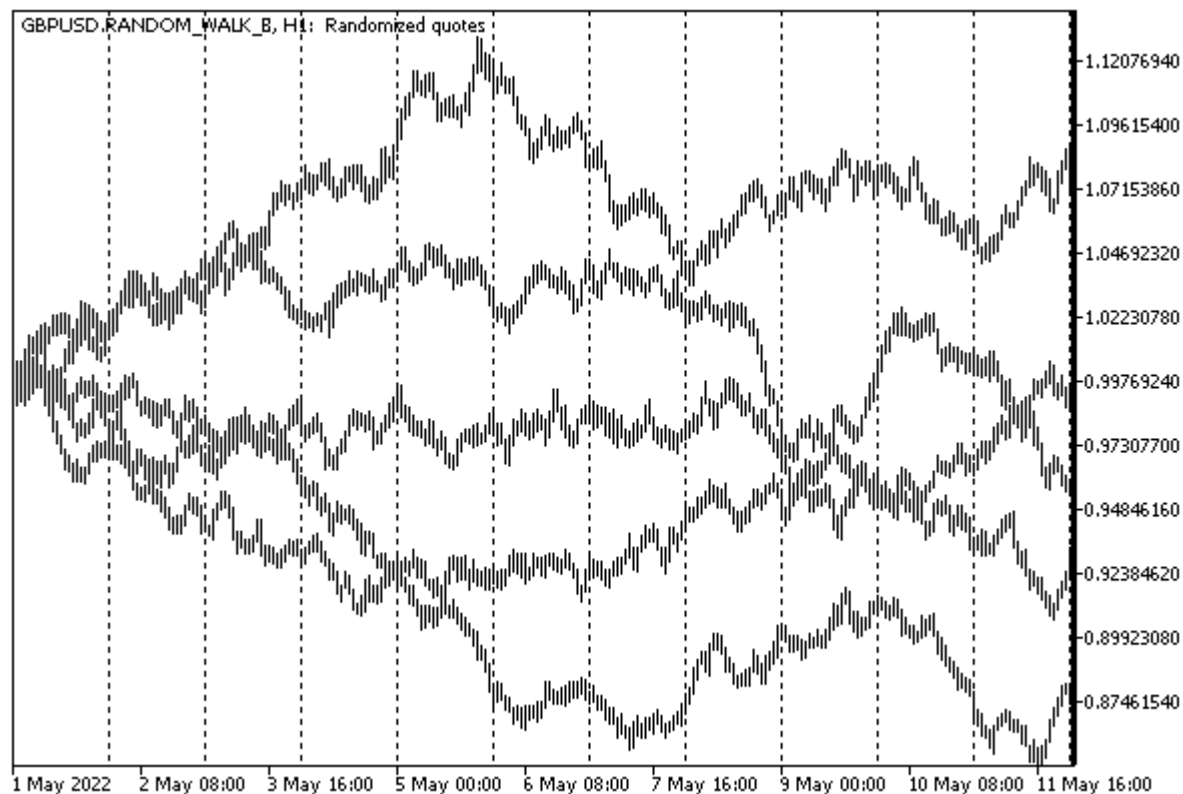
随机变量的离散系数取决于模式。例如，弱噪声增加（或减少）最多百分之一的价格，强噪声增加百分之五的价格。

运行时，脚本会输出如下所示的详细日志：

```
Create new custom symbol 'GBPUSD.RANDOM_WALK'?
CustomSymbolCreate(CustomSymbol,CustomPath,_Symbol)=true / ok
CustomRatesReplace(CustomSymbol,From,To,rates)=171416 / ok
Bars M1 generated: 171416
```

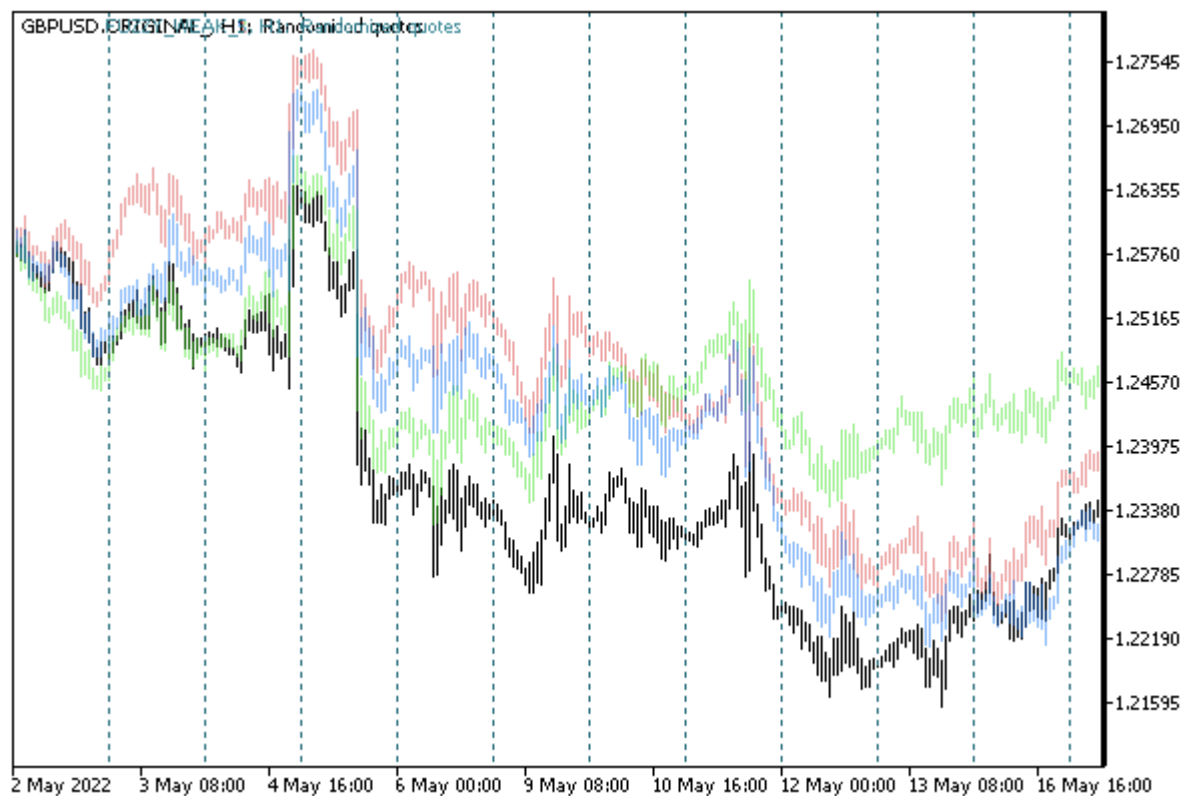
我们看看结果如何。

下图显示了随机游走的几种实现（视觉叠加是在图形编辑器中完成的，实际上，每个自定义交易品种一般都是在单独的窗口中打开）。



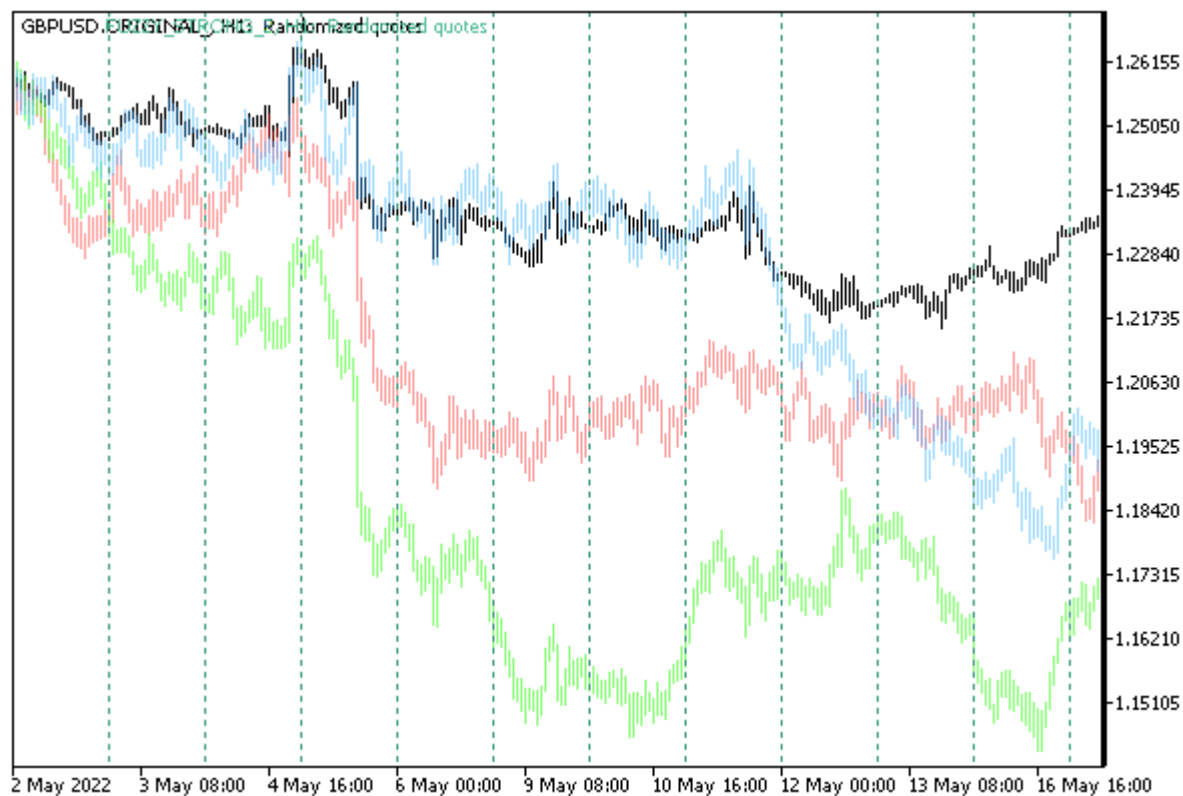
具有随机游走的自定义交易品种的报价选项

这是 GBPUSD 报价加噪后的样子（黑色为原始报价，彩色为加噪报价）。首先，是弱噪声版本。



弱噪声的 GBPUSD 报价

然后是强噪声。



强噪声的 GBPUSD 报价

尽管保留了局部特征，但明显存在较大的差异。

7.2.6 添加、替换和移除分时报价

MQL5 API 允许你不仅在柱线级别,而且在分时报价级别生成自定义交易品种的历史。因此,在测试和优化 EA 交易时可以实现更高的真实性,并模拟自定义交易品种图表的实时更新,向这些自定义交易品种广播你的分时报价。形成柱线时,系统会自动考虑传输到系统的分时报价集。换句话说,如果以分时报价的形式(即 *MqlTick* 结构体数组)提供了同一时期价格变化的更详细信息,则无需调用上一节中操作 *MqlRates* 结构体的函数。每柱线 *MqlRates* 报价的唯一优点是性能和内存效率。

有两个用于添加分时报价的函数: *CustomTicksAdd* 和 *CustomTicksReplace*。第一个函数添加到达 *Market Watch* 窗口的交互式分时报价(终端会自动将它们从该窗口传输到分时报价数据库),并在 MQL 程序中生成相应的事件。第二个函数直接将分时报价写入分时报价数据库。

int CustomTicksAdd(const string symbol, const MqlTick &ticks[], uint count = WHOLE_ARRAY)

CustomTicksAdd 函数将 *ticks* 数组中的数据添加到 *symbol* 中指定的自定义交易品种的价格历史中。默认情况下,如果 *count* 设置等于 *WHOLE_ARRAY*,则添加整个数组。如有必要,你可以指定一个较小的数字,仅下载一部分分时报价。

请注意,在调用函数时,自定义交易品种必须已在 *Market Watch* 窗口中被选中。对于未在 *Market Watch* 窗口中选择的交易品种,你需要使用 *CustomTicksReplace* 函数(见下文)。

分时报价数据数组必须按时间升序排序,即要求满足以下条件:对于所有 $i < j$, $ticks[i].time\_msc \leq ticks[j].time\_msc$

该函数返回添加的分时报价数量,如果发生错误则返回 -1。

CustomTicksAdd 函数以与从经纪商服务器传来相同的方式将分时报价广播到图表。通常,该函数应用于一个或多个分时报价。在这种情况下,它们会在 *Market Watch* 窗口中“播放”,然后从那里保存到分时报价数据库中。

但是,当一次调用中传输大量数据时,该函数会更改其行为以节省资源。如果传输超过 256 项分时报价,它们将被分成两部分。第一部分包含大量数据,立即直接写入分时报价数据库(与 *CustomTicksReplace* 的行为相同)。第二部分由最后(最新的)128 项分时报价组成,传递到 *Market Watch* 窗口,然后由终端保存到数据库中。

MqlTick 结构体有两个带时间值的字段: *time* (以秒为单位的分时报价时间)和 *time_msc* (以毫秒为单位的分时报价时间)。两个值都从 1970 年 1 月 1 日开始计时。已填充(非空)的 *time_msc* 字段优先于 *time*。请注意, *time* 是根据公式 $time\_msc / 1000$ 重新计算的结果,以秒为单位填充。如果 *time_msc* 字段为零,则使用 *time* 字段的值,而 *time_msc* 字段又从公式 $time * 1000$ 中获取以毫秒为单位的值。如果两个字段都等于零,则将当前服务器时间(精确到毫秒)放入分时报价中。

在描述交易量的两个字段中, *volume_real* 的优先级高于 *volume*。

根据特定数组元素(结构体 *MqlTick*)中填充了哪些其他字段,系统会在 *flags* 字段中为保存的分时报价设置标志:

- $ticks[i].bid \neq TICK\_FLAG\_BD$ (分时报价更改了卖价)
- $ticks[i].ask \neq TICK\_FLAG\_ASK$ (分时报价更改了买价)
- $ticks[i].last \neq TICK\_FLAG\_LAST$ (分时报价更改了最后一笔交易的价格)
- $ticks[i].volume$ 或 $ticks[i].volume\_real \neq TICK\_FLAG\_VOLUME$ (分时报价更改了交易量)

如果某个字段的值小于或等于零,则相应的标志不会写入 *flags* 字段。

TICK_FLAG_BUY 和 TICK_FLAG_SELL 标志不会添加到自定义交易品种的历史中。

CustomTicksReplace 函数用所传递数组中的数据完全替换指定时间间隔内自定义交易品种的价格历史。

```
int CustomTicksReplace(const string symbol, long from_msc, long to_msc,
    const MqlTick &ticks[], uint count = WHOLE_ARRAY)
```

时间间隔由 *from_msc* 和 *to_msc* 参数设置，单位为自 1970 年 1 月 1 日以来的毫秒数。两个值都包含在时间间隔内。

ticks 数组必须按分时报价到达的时间顺序排列，这对应于时间的增加，或者更确切地说，是非递减，因为具有相同时间的分时报价在具有毫秒精度的流中经常连续出现。

count 参数可用于处理数组的一部分。

在 *to_msc* 中指定的时间之前，或者以分时报价顺序发生错误之前，分时报价会逐日顺序替换。首先处理指定范围内的第一天，然后是下一天，依此类推。一旦检测到分时报价时间与升序（非递减）顺序不符，分时报价替换过程将在当前日期停止。在这种情况下，前几天的分时报价将成功替换，而当前日期（发生错误分时报价时）和指定时间间隔内的所有剩余日期将保持不变。该函数将返回 -1，*_LastError* 中的错误代码为 0（“无错误”）。

如果 *ticks* 数组在 *from_msc* 和 *to_msc*（含）之间的总时间间隔内没有某个时期的数据，则在执行该函数后，自定义交易品种的历史中将出现与缺失数据对应的间隙。

如果在指定的时间间隔内分时报价数据库中没有数据，**CustomTicksReplace** 将从 *ticks* 数组中向其添加分时报价。

CustomTicksDelete 函数可用于删除指定时间间隔内的所有分时报价。

```
int CustomTicksDelete(const string symbol, long from_msc, long to_msc)
```

正在编辑的自定义交易品种的名称在 *symbol* 参数中设置，要清除的时间间隔由参数 *from_msc* 和 *to_msc*（含）设置，单位为毫秒。

该函数返回移除的分时报价数量，如果发生错误则返回 -1。

注意！使用 **CustomTicksDelete** 删除分时报价会导致自动移除相应的柱线！但是，调用 **CustomRatesDelete**，即移除柱线，并不会移除分时报价！

为了在实践中掌握这些材料，我们将使用新学习的函数解决几个应用问题。

首先，我们接触一个有趣的任务，即基于真实交易品种但降低分时报价密度来创建自定义交易品种。与基于真实分时报价的模式相比，这将加快测试和优化速度，并减少资源消耗（主要是 RAM），同时保持可接受的、接近理想的过程质量。

加快测试和优化速度

交易者经常寻求加快 EA 交易优化和测试过程的方法。在可能的解决方案中，有些是显而易见的，只需更改设置即可（如果允许），还有一些更耗时，需要调整 EA 交易或测试环境。

第一类解决方案包括：

- 通过消除某些参数或减少其步长来缩小优化空间；
- 缩短优化周期；
- 切换到质量较低的分时报价模拟模式（例如，从真实分时报价到 OHLC M1）；
- 启用以点而不是资金计算利润；

- 升级计算机；
- 使用 MQL Cloud 或其他本地网络计算机。

第二类与开发相关的解决方案包括：

- 代码概要分析，在此基础上可以消除代码中的“瓶颈”；
- 如果可能，使用资源高效的指标计算，即不使用 `#property tester_everytick_calculate` 指令；
- 将指标算法（如果使用）直接移植到 EA 交易代码中：指标调用会产生一定的开销成本；
- 消除图形和对象；
- 缓存计算（如果可能）；
- 减少同时未平仓仓位和平仓订单的数量（大量订单时，每项分时报价的计算可能会变得明显）；
- 结算、订单、交易和仓位的完全虚拟化：内置的记账机制由于其通用性、多币种支持和其他特性，有其自身的开销，可以通过在 MQL5 代码中执行类似的操作来消除（尽管此选项最耗时）。

分时报价密度降低属于一种中间类型的解决方案：它需要以编程方式创建自定义交易品种，但不会影响 EA 交易的源代码。

具有更少分时报价的自定义交易品种将由脚本 *CustomSymbolFilterTicks.mq5* 生成。初始金融工具将是启动脚本的图表的工作交易品种。在输入参数中，你可以指定自定义交易品种的文件夹和历史处理的开始日期。默认情况下，如果未指定日期，则计算最近 120 天。

```
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input datetime _Start; // Start (default: 120 days back)
```

交易品种的名称由源金融工具的名称和“.TckFltr”后缀组成。稍后我们将向其添加分时报价减少方法的标记。

```
string CustomSymbol = _Symbol + ".TckFltr";
const uint DailySeconds = 60 * 60 * 24;
datetime Start = _Start == 0 ? TimeCurrent() - DailySeconds * 120 : _Start;
```

为方便起见，在 *OnStart* 处理程序中，如果先前副本已存在，则可以删除它。

```

void OnStart()
{
    bool custom = false;
    if(PRTF(SymbolExist(CustomSymbol, custom)) && custom)
    {
        if(IDYES == MessageBox(StringFormat("Delete existing custom symbol '%s'?", Cust
            "Please, confirm", MB_YESNO))
        {
            SymbolSelect(CustomSymbol, false);
            CustomRatesDelete(CustomSymbol, 0, LONG_MAX);
            CustomTicksDelete(CustomSymbol, 0, LONG_MAX);
            CustomSymbolDelete(CustomSymbol);
        }
        else
        {
            return;
        }
    }
}

```

接下来，在用户同意后，将创建一个交易品种。历史由辅助函数 *GenerateTickData* 中的分时报价数据填充。如果成功，脚本会将新交易品种添加到 *Market Watch* 并打开图表。

```

if(IDYES == MessageBox(StringFormat("Create new custom symbol '%s'?", CustomSymbol
    "Please, confirm", MB_YESNO))
{
    if(PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
    {
        CustomSymbolSetString(CustomSymbol, SYMBOL_DESCRIPTION, "Prunned ticks by "
            if(GenerateTickData())
            {
                SymbolSelect(CustomSymbol, true);
                ChartOpen(CustomSymbol, PERIOD_H1);
            }
        }
    }
}
}

```

GenerateTickData 函数以每天为单位，分批循环处理分时报价。每天的分时报价通过调用 *CopyTicksRange* 来请求。然后需要以某种方式减少它们，这由我们将在下面展示的 *TickFilter* 类实现。最后，使用 *CustomTicksReplace* 将分时报价数组添加到自定义交易品种历史中。

```

bool GenerateTickData()
{
    bool result = true;
    datetime from = Start / DailySeconds * DailySeconds; // round up to the beginning
    ulong read = 0, written = 0;
    uint day = 0;
    const uint total = (uint)((TimeCurrent() - from) / DailySeconds + 1);
    MqlTick array[];

    while(!IsStopped() && from < TimeCurrent())
    {
        Comment(TimeToString(from, TIME_DATE), " ", day++, "/", total);

        const int r = CopyTicksRange(_Symbol, array, COPY_TICKS_ALL,
            from * 1000L, (from + DailySeconds) * 1000L - 1);
        if(r < 0)
        {
            Alert("Error reading ticks at ", TimeToString(from, TIME_DATE));
            result = false;
            break;
        }
        read += r;

        if(r > 0)
        {
            const int t = TickFilter::filter(Mode, array);
            const int w = CustomTicksReplace(CustomSymbol,
                from * 1000L, (from + DailySeconds) * 1000L - 1, array);
            if(w <= 0)
            {
                Alert("Error writing custom ticks at ", TimeToString(from, TIME_DATE));
                result = false;
                break;
            }
            written += w;
        }
        from += DailySeconds;
    }

    if(read > 0)
    {
        PrintFormat("Done ticks - read: %lld, written: %lld, ratio: %.1f%%",
            read, written, written * 100.0 / read);
    }
    Comment("");
    return result;
}

```

在所有阶段都实现了错误控制和已处理分时报价的计数。最后，我们将初始和剩余分时报价的数量以及“压缩”因子输出到日志中。

现在，我们直接转向分时报价缩减技术。显然，可以采用很多方法，每种方法都或多或少地适用于特定的交易策略。我们将提供 3 个基本版本，组合在 *TickFilter* 类 (*TickFilter.mqh*) 中。此外，为了完整起见，还支持不减少分时报价的复制模式。

因此，该类中实现了以下模式：

- 无缩减
- 跳过价格单调变化而没有反转的分时报价序列（类似于“之字形”）
- 跳过点差内的价格波动
- 仅当 *Bid* 或 *Ask* 在两个相邻分时报价之间表示极值时，才记录具有分形配置的分时报价

这些模式被描述为 `FILTER_MODE` 枚举的元素。

```
class TickFilter
{
public:
    enum FILTER_MODE
    {
        NONE,
        SEQUENCE,
        FLUTTER,
        FRACTALS,
    };
    ...
}
```

每种模式都由一个单独的静态方法实现，该方法接受一个需要精简的分时报价数组作为输入。数组编辑在原地执行（不分配新的输出数组）。

```
static int filterBySequences(MqlTick &data[]);
static int filterBySpreadFlutter(MqlTick &data[]);
static int filterByFractals(MqlTick &data[]);
```

所有方法都返回剩余的分时报价数量（缩减后的数组大小）。

为了统一不同模式下过程的执行，提供了 *filter* 方法。对于 `NONE` 模式，*data* 数组保持不变。

```
static int filter(FILTER_MODE mode, MqlTick &data[])
{
    switch(mode)
    {
        case SEQUENCE: return filterBySequences(data);
        case FLUTTER: return filterBySpreadFlutter(data);
        case FRACTALS: return filterByFractals(data);
    }
    return ArraySize(data);
}
```

例如，以下展示了如何在 *filterBySequences* 方法中通过单调分时报价序列进行筛选。

```

static int filterBySequences(MqlTick &data[])
{
    const int size = ArraySize(data);
    if(size < 3) return size;

    int index = 2;
    bool dirUp = data[1].bid - data[0].bid + data[1].ask - data[0].ask > 0;

    for(int i = 2; i < size; i++)
    {
        if(dirUp)
        {
            if(data[i].bid - data[i - 1].bid + data[i].ask - data[i - 1].ask < 0)
            {
                dirUp = false;
                data[index++] = data[i];
            }
        }
        else
        {
            if(data[i].bid - data[i - 1].bid + data[i].ask - data[i - 1].ask > 0)
            {
                dirUp = true;
                data[index++] = data[i];
            }
        }
    }
    return ArrayResize(data, index);
}

```

这是分形精简呈现的效果。

```

static int filterByFractals(MqlTick &data[])
{
    int index = 1;
    const int size = ArraySize(data);
    if(size < 3) return size;

    for(int i = 1; i < size - 2; i++)
    {
        if((data[i].bid < data[i - 1].bid && data[i].bid < data[i + 1].bid)
            || (data[i].ask > data[i - 1].ask && data[i].ask > data[i + 1].ask))
        {
            data[index++] = data[i];
        }
    }

    return ArrayResize(data, index);
}

```

我们依次为 EURUSD 创建几种分时报价密度降低模式的自定义交易品种，并比较它们的性能，即“压缩”程度、测试速度以及EA交易交易性能的变化。

例如，精简分时报价序列会产生以下结果（在 MQ Demo 上一年半的历史）。

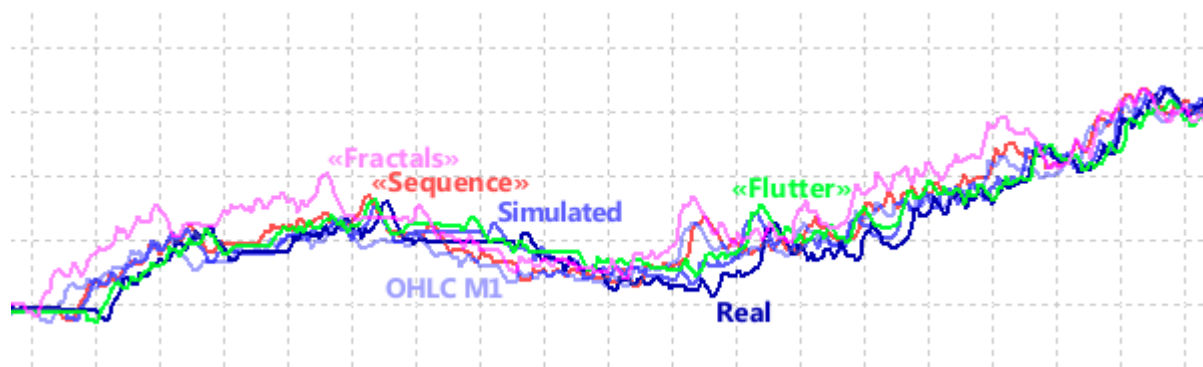
```
Create new custom symbol 'EURUSD.TckFltr-SE'?
Fixing SYMBOL_TRADE_TICK_VALUE: 0.0 <<< 1.0
true SYMBOL_TRADE_TICK_VALUE 1.0 -> SUCCESS (0)
Fixing SYMBOL_TRADE_TICK_SIZE: 0.0 <<< 1e-05
true SYMBOL_TRADE_TICK_SIZE 1e-05 -> SUCCESS (0)
Number of found discrepancies: 2
Fixed
Done ticks - read: 31553509, written: 16927376, ratio: 53.6%
```

对于平滑波动和分形模式，指标是不同的：

```
EURUSD.TckFltr-FL will be updated
Done ticks - read: 31568782, written: 22205879, ratio: 70.3%
...
Create new custom symbol 'EURUSD.TckFltr-FR'?
...
Done ticks - read: 31569519, written: 12732777, ratio: 40.3%
```

对于基于压缩分时报价的实际交易实验，我们需要 EA 交易。我们采用 *BandOsMATicks.mq5* 的改编版本，与 原始版本相比，其中启用了每项分时报价的交易（在 *SimpleStrategy::trade* 方法中禁用了 *if(lastBar == iTime(_Symbol, _Period, 0)) return false;* 行），并且信号指标的值取自柱线 0 和 1（以前只有已完成的柱线 1 和 2）。

我们使用从 2021 年初到 2022 年 6 月 1 日的日期范围运行 EA 交易。设置附加在 *MQ5/Presets/MQL5Book/BandOsMATicks.set* 文件中。所有模式下余额曲线的总体行为都非常相似。



不同模式下按分时报价划分的测试余额组合图表

不同曲线等效极值的水平偏移是由于标准报告图表使用交易数量而不是时间作为水平坐标，而交易数量由于不同分时报价基础上交易信号触发的准确性而有所不同。

性能指标的差异如下表所示 N - 交易数量，\$ - 利润，PF - 盈利因子，RF - 恢复因子，DD - 回撤）：

模式	分时报价	时间 mm:ss.msec	内存	N	\$	PF	RF	DD
真	31002919	02:45.251	835 Mb	962	166.24	1.32	2.88	54.99
模拟	25808139	01:58.131	687 Mb	928	171.94	1.34	3.44	47.64
OHLC M1	2084820	00:11.094	224 Mb	856	193.52	1.39	3.97	46.55
序列	16310236	01:24.784	559 Mb	860	168.95	1.34	2.92	55.16
快速震荡	21362616	01:52.172	623 Mb	920	179.75	1.37	3.60	47.28
分形	12270854	01:04.756	430 Mb	866	142.19	1.27	2.47	54.80

我们将认为基于真实分时报价的测试是最可靠的，并通过其与此测试的接近程度来评估其余测试。显然，由于准确性的显著损失（未考虑开盘价模式），OHLC M1 模式显示出最高的速度和较低的资源成本。它表现出过于乐观的财务结果。

在三种人为压缩分时报价的模式中，“序列”模式在指标集方面最接近真实模式。“序列”模式在时间上比真实模式快 2 倍，在内存消耗方面效率高 1.5 倍。“快速震荡”模式似乎更好地保留了原始交易数量。速度最快且内存需求最低的分形模式，当然比 OHLC M1 模式花费更多的时间和资源，但它不会高估交易得分。

请记住，对于不同的交易策略、金融工具，甚至特定经纪商的分时报价历史，分时报价缩减算法可能会有不同的工作方式，或者相反，会产生较差的结果。请在你的工作环境中使用 EA 交易进行研究。

作为使用自定义交易品种的第二个示例，我们考虑一下使用 *CustomTicksAdd* 进行分时报价转换所提供的一个有趣功能。

许多交易者使用交易面板，即带有交互式控件的程序，用于手动执行任意交易操作。你必须主要在线练习使用它们，因为测试程序会施加一些限制。首先，测试程序不支持图表事件和对象。这会导致控件停止工作。此外，在测试程序中，你不能对图形标记应用任意对象。

我们尝试解决这些问题。

我们可以基于历史分时报价以慢动作生成自定义交易品种。然后，此类交易品种的图表将成为可视化测试程序的类似物。

这种方法有几个优点：

- 所有图表事件的标准行为
- 指标的交互式应用和设置
- 对象的交互式应用和调整
- 动态切换时间范围
- 测试历史数据直至当前时间，包括今天（标准测试器不允许测试今天）

关于最后一点，我们注意到 MetaTrader 5 的开发人员故意禁止检查最后（当前）一天的交易，尽管有时需要快速查找错误（在代码中或交易策略中）。

动态修改价格（例如增加点差）也具有潜在的趣味性。

基于此类自定义交易品种的图表，我们稍后可以基于历史数据实现手动交易模拟器。

交易品种生成器将是非交易性 EA 交易 *CustomTester.mq5*。在其输入参数中，我们将提供新自定义交易品种在交易品种层级结构中的位置指示、用于分时报价转换（和构建自定义交易品种行情）的过去开始日期，以及图表的时间周期，该图表将自动打开以进行可视化测试。

```
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input datetime _Start; // Start (120-day indent by default)
input ENUM_TIMEFRAMES Timeframe = PERIOD_H1;
```

新交易品种的名称由当前图表的交易品种名称和 ".Tester" 后缀构成。

```
string CustomSymbol = _Symbol + ".Tester";
```

如果参数中未指定开始日期，EA 交易将从当前日期向后缩进 120 天。

```
const uint DailySeconds = 60 * 60 * 24;
datetime Start = _Start == 0 ? TimeCurrent() - DailySeconds * 120 : _Start;
```

分时报价将从工作交易品种的真实分时报价历史中一次性读取一整天的数据。正在读取的日期的指针存储在 *Cursor* 变量中。

```
bool FirstCopy = true;
// additionally 1 day ago, because otherwise, the chart will not update immediately
datetime Cursor = (Start / DailySeconds - 1) * DailySeconds; // round off at the border c
```

要重现的全天分时报价将在 *Ticks* 数组中请求，然后从那里以 *step* 大小小批量转换到自定义交易品种的图表。

```
MqlTick Ticks[]; // ticks for the "current" day in the past
int Index = 0; // position in ticks within a day
int Step = 32; // fast forward 32 ticks at a time (default)
int StepRestore = 0; // remember the speed for the duration of the pause
long Chart = 0; // created custom symbol chart
bool InitDone = false; // sign of completed initialization
```

为了以恒定速率播放分时报价，我们在 *OnInit* 中启动计时器。

```
void OnInit()
{
    EventSetMillisecondTimer(100);
}

void OnTimer()
{
    if(!GenerateData())
    {
        EventKillTimer();
    }
}
```

分时报价将由 *GenerateData* 函数生成。启动后，当 *InitDone* 标志被重置时，我们将尝试创建一个新交易品种，或者如果自定义交易品种已存在，则清除旧的行情和分时报价。

```

bool GenerateData()
{
    if(!InitDone)
    {
        bool custom = false;
        if(PRTF(SymbolExist(CustomSymbol, custom)) && custom)
        {
            if(IDYES == MessageBox(StringFormat("Clean up existing custom symbol '%s'?",
                CustomSymbol), "Please, confirm", MB_YESNO))
            {
                PRTF(CustomRatesDelete(CustomSymbol, 0, LONG_MAX));
                PRTF(CustomTicksDelete(CustomSymbol, 0, LONG_MAX));
                Sleep(1000);
                MqlRates rates[1];
                MqlTick tcks[];
                if(PRTF(CopyRates(CustomSymbol, PERIOD_M1, 0, 1, rates)) == 1
                    || PRTF(CopyTicks(CustomSymbol, tcks) > 0))
                {
                    Alert("Can't delete rates and Ticks, internal error");
                    ExpertRemove();
                }
            }
            else
            {
                return false;
            }
        }
        else
        {
            if(!PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
            {
                return false;
            }
        }
        ... // (A)
    }
}

```

此时，我们将在 (A) 处省略一些内容，稍后再回到这一点。

创建交易品种后，我们在 *Market Watch* 窗口中将其选中并为其打开一个图表。

```

SymbolSelect(CustomSymbol, true);
Chart = ChartOpen(CustomSymbol, Timeframe);
... // (B)
ChartSetString(Chart, CHART_COMMENT, "Custom Tester");
ChartSetInteger(Chart, CHART_SHOW_OBJECT_DESCR, true);
ChartRedraw(Chart);
InitDone = true;
}
...

```

这里也缺少几行 (B); 它们与未来的改进有关，但目前还不是必需的。

如果交易品种已经创建，我们开始以 *Step* 个分时报价为一批次广播分时报价，但不能超过 256 个。此限制与 *CustomTicksAdd* 函数的某些细节有关。

```

else
{
    for(int i = 0; i <= (Step - 1) / 256; ++i)
        if(Step > 0 && !GenerateTicks())
        {
            return false;
        }
}
return true;
}

```

辅助函数 *GenerateTicks* 以 *Step* 个 (但不能超过 256 个) 分时报价为一批次广播分时报价, 通过偏移量 *Index* 从每日数组 *Ticks* 中读取它们。当数组为空或我们已将其读到末尾时, 我们通过调用 *FillTickBuffer* 请求第二天的分时报价。

```

bool GenerateTicks()
{
    if(Index >= ArraySize(Ticks)) // daily array is empty or read to the end
    {
        if(!FillTickBuffer()) return false; // fill the array with ticks per day
    }

    const int m = ArraySize(Ticks);
    MqlTick array[];
    const int n = ArrayCopy(array, Ticks, 0, Index, fmin(fmin(Step, 256), m));
    if(n <= 0) return false;

    ResetLastError();
    if(CustomTicksAdd(CustomSymbol, array) != ArraySize(array) || _LastError != 0)
    {
        Print(_LastError); // in case of ERR_CUSTOM_TICKS_WRONG_ORDER (5310)
        ExpertRemove();
    }
    Comment("Speed: ", (string)Step, " / ", STR_TIME_MSC(array[n - 1].time_msc));
    Index += Step; // move forward by 'Step' ticks
    return true;
}

```

FillTickBuffer 函数使用 *CopyTicksRange* 进行操作。

```

bool FillTickBuffer()
{
    int r;
    ArrayResize(Ticks, 0);
    do
    {
        r = PRTF(CopyTicksRange(_Symbol, Ticks, COPY_TICKS_ALL, Cursor * 1000L,
            (Cursor + DailySeconds) * 1000L - 1));
        if(r > 0 && FirstCopy)
        {
            // NB: this pre-call is only needed to display the chart
            // from "Waiting for update" state
            PRTF(CustomTicksReplace(CustomSymbol, Cursor * 1000L,
                (Cursor + DailySeconds) * 1000L - 1, Ticks));
            FirstCopy = false;
            r = 0;
        }
        Cursor += DailySeconds;
    }
    while(r == 0 && Cursor < TimeCurrent()); // skip non-trading days
    Index = 0;
    return r > 0;
}

```

当 EA 交易停止时，我们还将关闭相关的图表（以便在下次启动时不会重复）。

```

void OnDeinit(const int)
{
    if(Char != 0)
    {
        ChartClose(Char);
    }
    Comment("");
}

```

至此，EA 交易可以被认为是完整的了，但是存在一个问题。问题在于，由于某种原因，自定义交易品种的特性不会从原始工作交易品种中“按原样”复制，至少在 MQL5 API 的当前实现中是这样。这甚至适用于非常重要的特性，例如 `SYMBOL_TRADE_TICK_VALUE`、`SYMBOL_TRADE_TICK_SIZE`。如果在调用 `CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)` 后立即打印这些特性的值，我们将在那里看到零。

为了组织特性的检查、比较以及必要时更正，我们编写了一个特殊的类 `CustomSymbolMonitor` (`CustomSymbolMonitor.mqh`)，它派生自 `SymbolMonitor`。你可以自行研究其内部结构，而这里我们仅介绍公共接口。

构造函数允许你创建一个自定义交易品种监视器，指定一个示例工作交易品种（通过字符串中的名称，或从 `SymbolMonitor` 对象），该交易品种用作设置的来源。

```

class CustomSymbolMonitor: public SymbolMonitor
{
public:
    CustomSymbolMonitor(); // sample - _Symbol
    CustomSymbolMonitor(const string s, const SymbolMonitor *m = NULL);
    CustomSymbolMonitor(const string s, const string other);

    //set/replace sample symbol
    void inherit(const SymbolMonitor &m);

    // copy all properties from the sample symbol in forward or reverse order
    bool setAll(const bool reverseOrder = true, const int limit = UCHAR_MAX);

    // check all properties against the sample, return the number of corrections
    int verifyAll(const int limit = UCHAR_MAX);

    // check the specified properties with the sample, return the number of correction
    int verify(const int &properties[]);

    // copy the given properties from the sample, return true if they all applied
    bool set(const int &properties[]);

    // copy the specific property from the sample, return true if applied
    template<typename E>
    bool set(const E e);

    bool set(const ENUM_SYMBOL_INFO_INTEGER property, const long value) const
    {
        return CustomSymbolSetInteger(name, property, value);
    }

    bool set(const ENUM_SYMBOL_INFO_DOUBLE property, const double value) const
    {
        return CustomSymbolSetDouble(name, property, value);
    }

    bool set(const ENUM_SYMBOL_INFO_STRING property, const string value) const
    {
        return CustomSymbolSetString(name, property, value);
    }
};

```

由于自定义交易品种与标准交易品种不同，允许你设置自己的特性，因此向该类添加了三个 **set** 方法。特别是，它们用于批量传输示例的特性并在其他类方法中检查这些操作的成功性。

我们现在可以回到自定义交易品种生成器及其源代码片段，如前面注释 (A) 所示。

```

// (A) check important properties and set them in "manual" mode
SymbolMonitor sm; // _Symbol
CustomSymbolMonitor csm(CustomSymbol, &sm);
int props[] = {SYMBOL_TRADE_TICK_VALUE, SYMBOL_TRADE_TICK_SIZE};
const int d1 = csm.verify(props); // check and try to fix
if(d1)
{
    Print("Number of found discrepancies: ", d1); // number of edits
    if(csm.verify(props)) // check again
    {
        Alert("Custom symbol can not be created, internal error!");
        return false; // symbol cannot be used without successful edits
    }
    Print("Fixed");
}

```

现在你可以运行 *CustomTester.mq5* EA 交易，并观察报价如何在自动打开的图表中动态形成，以及分时报价如何从历史转发到 *Market Watch* 窗口。

但是，这是以每 0.1 秒 32 个分时报价的恒定速率完成的。理想情况下，应根据用户的请求动态更改播放速度，既可以加快也可以减慢。例如，可以通过键盘来实现这种控制。

因此，你需要添加 *OnChartEvent* 处理程序。我们知道，对于 *CHARTEVENT_KEYDOWN* 事件，程序在 *lparam* 参数中接收按下键的代码，我们将其传递给 *CheckKeys* 函数（见下文）。与 (B) 密切相关的片段 (C) 不得不暂时推迟，我们稍后会回到它。

```

void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &str)
{
    ... // (C)
    if(id == CHARTEVENT_KEYDOWN) // these events only arrive while the chart is active
    {
        CheckKeys(lparam);
    }
}

```

在 *CheckKeys* 函数中，我们正在处理“向上箭头”和“向下箭头”键以增加和减少播放速度。此外，“暂停”键允许你完全暂停“测试”（分时报价传输）的过程。再次按下“暂停”键将以相同的速度恢复工作。

```

void CheckKeys(const long key)
{
    if(key == VK_DOWN)
    {
        Step /= 2;
        if(Step > 0)
        {
            Print("Slow down: ", Step);
            ChartSetString(Chart, CHART_COMMENT, "Speed: " + (string)Step);
        }
        else
        {
            Print("Paused");
            ChartSetString(Chart, CHART_COMMENT, "Paused");
            ChartRedraw(Chart);
        }
    }
    else if(key == VK_UP)
    {
        if(Step == 0)
        {
            Step = 1;
            Print("Resumed");
            ChartSetString(Chart, CHART_COMMENT, "Resumed");
        }
        else
        {
            Step *= 2;
            Print("Speed up: ", Step);
            ChartSetString(Chart, CHART_COMMENT, "Speed: " + (string)Step);
        }
    }
    else if(key == VK_PAUSE)
    {
        if(Step > 0)
        {
            StepRestore = Step;
            Step = 0;
            Print("Paused");
            ChartSetString(Chart, CHART_COMMENT, "Paused");
            ChartRedraw(Chart);
        }
        else
        {
            Step = StepRestore;
            Print("Resumed");
            ChartSetString(Chart, CHART_COMMENT, "Speed: " + (string)Step);
        }
    }
}

```


在首先确保 EA 交易工作的图表处于活动状态后，可以实际测试新代码。回想一下，键盘事件仅发送到活动窗口。这是我们测试程序的另一个问题。

由于用户必须在自定义交易品种图表上执行交易操作，因此生成器窗口几乎总是位于后台。要想切换到生成器窗口以暂时停止分时报价流然后恢复，这种做法不切实际。因此，需要以某种方式直接从自定义交易品种窗口通过键盘组织交互式控制。

为此，适合使用一个特殊的指标，我们可以自动将其添加到打开的自定义交易品种窗口中。该指标将在其自己的窗口（带有自定义交易品种的窗口）中拦截键盘事件，并将这些事件发送到生成器窗口。

指标的源代码附在 *KeyboardSpy.mq5* 文件中。当然，该指标没有图表。一对输入参数专用于获取图表 ID *HostID*（应向其发送消息）和自定义事件代码 *EventID*（交互式事件将打包到其中）。

```
#property indicator_chart_window
#property indicator_plots 0

input long HostID;
input ushort EventID;
```

主要工作在 *OnChartEvent* 处理程序中完成。

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &sparam)
{
    if(id == CHARTEVENT_KEYDOWN)
    {
        EventChartCustom(HostID, EventID, lparam,
            // this is always 0 when inside iCustom
            (double)(ushort)TerminalInfoInteger(TERMINAL_KEYSTATE_CONTROL),
            sparam);
    }
}
```

请注意，我们选择的所有“热键”都是简单的，也就是说，它们不使用带有键盘状态键（例如 *Ctrl* 或 *Shift*）的快捷方式。这是强制执行的，因为在以编程方式创建的指标（具体而言是通过 *iCustom*）内部，无法读取键盘状态。换句话说，调用 *TerminalInfoInteger(TERMINAL_KEYSTATE_XYZ)* 总是返回 0。在上面的处理程序中，我们添加它只是出于演示目的，这样的话，如果你愿意，可以通过在“接收端”显示传入参数来验证此限制。

但是，单个箭头和暂停点击将正常传输到父图表，这对我们来说已经足够了。剩下的唯一事情就是将指标与 EA 交易集成。

在前面跳过的片段（B）中，在生成器初始化期间，我们将创建一个指标并将其添加到自定义交易品种图表中。

```
#define EVENT_KEY 0xDEDED // custom event
...
// (B)
const int handle = iCustom(CustomSymbol, Timeframe, "MQL5Book/p7/KeyboardSpy",
    ChartID(), EVENT_KEY);
ChartIndicatorAdd(Chart, 0, handle);
```

接下来，在片段（C）中，我们将确保从指标接收用户消息并将其传输到已知的 *CheckKeys* 函数。

```

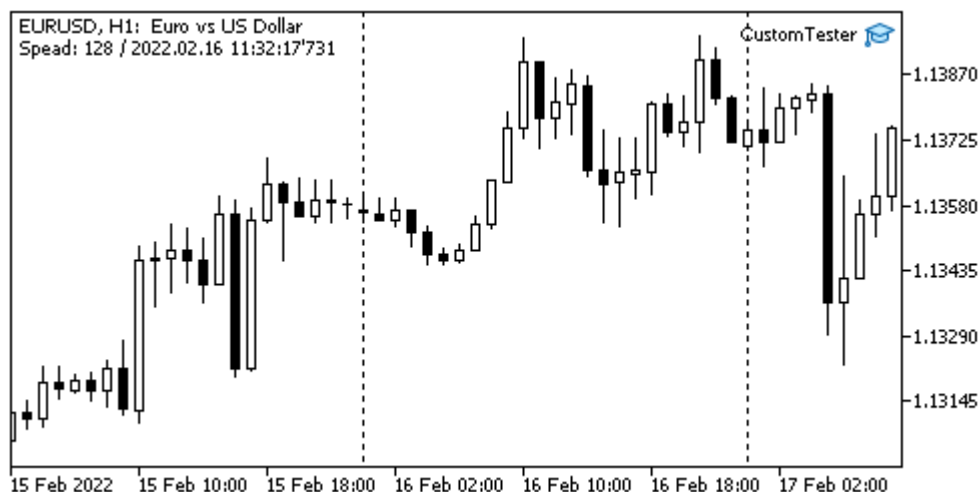
void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &str)
{
    // (C)
    if(id == CHARTEVENT_CUSTOM + EVENT_KEY) // notifications from the dependent chart
    {
        CheckKeys(lparam); // "remote" processing of key presses
    }
    else if(id == CHARTEVENT_KEYDOWN) // these events are only fired while the chart is active
    {
        CheckKeys(lparam); // standard processing
    }
}

```

因此，现在可以在带有 EA 交易的图表和由其生成的自定义交易品种的图表上控制播放速度。

使用新的工具包，你可以尝试与“过去活动”的图表进行交互式工作。图表上会显示带有当前播放速度或暂停标记的注释。

在带有 EA 交易的图表上，注释中会显示“当前”广播分时报价的时间。



一个重现真实交易品种分时报价 (和行情) 历史的 EA 交易

用户在此窗口中基本没有可以执行的操作 (除非删除 EA 交易并停止自定义交易品种生成)。分时报价转换过程本身在此处不可见。此外，由于 EA 交易会打开一个自定义交易品种图表 (其中历史行情会更新)，因此正是这个图表变为活动状态。为了获得上面的屏幕截图，我们特别需要短暂切换到原始图表。

因此，我们回到自定义交易品种的图表。它在过去平稳渐进地更新的方式已经很棒了，但是无法在其上进行交易实验。例如，如果你在其上运行常用的交易面板，其控件虽然在形式上可以工作，但不会执行交易，因为自定义交易品种在服务器上不存在，因此你会收到错误。在任何未专门针对自定义交易品种进行调整的程序中都会观察到此特性。我们展示一个如何虚拟化自定义交易品种交易的示例。

为了简化示例 (但不失一般性)，我们将以最简单的 EA 交易 *CustomOrderSend.mq5* 为基础，而不是交易面板，该 EA 交易可以通过按键执行多种交易操作：

- 'B' - 市价买入
- 'S' - 市价卖出
- 'U' - 下达限价买单

- 'L' -下达限价卖单
- 'C' -全部平仓
- 'D' -删除所有订单
- 'R' -将交易报告输出到日志

在 EA 交易输入参数中,我们将设置单笔交易的交易量 (默认为最小手数)以及止损和止盈水平的点数距离。

```
input double Volume;           // Volume (0 = minimal lot)
input int Distance2SLTP = 0;    // Distance to SL/TP in points (0 = no)

const double Lot = Volume == 0 ? SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volum
```

如果 *Distance2SLTP* 等于零,则市价单中不设置保护水平,也不形成挂单。当 *Distance2SLTP* 具有非零值时,它用作下挂单时与当前价格的距离 (向上或向下,取决于命令)。

考虑到先前从 *MqlTradeSync.mqh*中介绍的类,上述逻辑转换为以下源代码。

```

#include <MQL5Book/MqlTradeSync.mqh>

#define KEY_B 66
#define KEY_C 67
#define KEY_D 68
#define KEY_L 76
#define KEY_R 82
#define KEY_S 83
#define KEY_U 85

void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &text)
{
    if(id == CHARTEVENT_KEYDOWN)
    {
        MqlTradeRequestSync request;
        const double ask = SymbolInfoDouble(_Symbol, SYMBOL_ASK);
        const double bid = SymbolInfoDouble(_Symbol, SYMBOL_BID);
        const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);

        switch((int)lparam)
        {
            case KEY_B:
                request.buy(Lot, 0,
                    Distance2SLTP ? ask - point * Distance2SLTP : Distance2SLTP,
                    Distance2SLTP ? ask + point * Distance2SLTP : Distance2SLTP);
                break;
            case KEY_S:
                request.sell(Lot, 0,
                    Distance2SLTP ? bid + point * Distance2SLTP : Distance2SLTP,
                    Distance2SLTP ? bid - point * Distance2SLTP : Distance2SLTP);
                break;
            case KEY_U:
                if(Distance2SLTP)
                {
                    request.buyLimit(Lot, ask - point * Distance2SLTP);
                }
                break;
            case KEY_L:
                if(Distance2SLTP)
                {
                    request.sellLimit(Lot, bid + point * Distance2SLTP);
                }
                break;
            case KEY_C:
                for(int i = PositionsTotal() - 1; i >= 0; i--)
                {
                    request.close(PositionGetTicket(i));
                }
                break;
            case KEY_D:
                for(int i = OrdersTotal() - 1; i >= 0; i--)

```

```

        {
            request.remove(OrderGetTicket(i));
        }
        break;
        case KEY_R:
            // there should be something here...
            break;
    }
}
}

```

正如我们所见，这里同时使用了标准的交易 API 函数和 *MqlTradeRequestSync* 方法。后者间接地最终也会调用许多内置函数。我们需要让这个 EA 交易与自定义交易品种进行交易。

最简单但耗时的想法是用它们自己的类似功能替换所有标准函数，这些类似功能会在某些结构中计算订单、交易、仓位和财务统计数据。当然，必须拥有需要调整的 EA 交易源代码才能实现。

该方法的实验性实现在附加文件 *CustomTrade.mqh* 中进行了演示。你可以自行熟悉完整的代码，因为在本书的框架内，我们仅列出要点。

首先，我们注意到许多计算是以简化形式进行的，不支持许多模式，并且未对数据的正确性进行完整检查。请将源代码用作你自己开发的起点。

整个代码都包含在 *CustomTrade* 命名空间中，以避免冲突。

订单、交易和仓位实体被形式化为相应的类 *CustomOrder*、*CustomDeal* 和 *CustomPosition*。它们都是 *MonitorInterface<I,D,S> ::TradeState* 类的继承者。回想一下，这个类已经自动支持为每种类型的对象及其特定的枚举三元组形成整数、实数和字符串特性的数组。例如，*CustomOrder* 看起来是这样的：

```

class CustomOrder: public MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,
    ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>::TradeState
{
    static long ticket; // order counter and ticket provider
    static int done;    // counter of executed (historical) orders
public:
    CustomOrder(const ENUM_ORDER_TYPE type, const double volume, const string symbol)
    {
        _set(ORDER_TYPE, type);
        _set(ORDER_TICKET, ++ticket);
        _set(ORDER_TIME_SETUP, SymbolInfoInteger(symbol, SYMBOL_TIME));
        _set(ORDER_TIME_SETUP_MSC, SymbolInfoInteger(symbol, SYMBOL_TIME_MSC));
        if(type <= ORDER_TYPE_SELL)
        {
            // TODO: no deferred execution yet
            setDone(ORDER_STATE_FILLED);
        }
        else
        {
            _set(ORDER_STATE, ORDER_STATE_PLACED);
        }

        _set(ORDER_VOLUME_INITIAL, volume);
        _set(ORDER_VOLUME_CURRENT, volume);

        _set(ORDER_SYMBOL, symbol);
    }

    void setDone(const ENUM_ORDER_STATE state)
    {
        const string symbol = _get<string>(ORDER_SYMBOL);
        _set(ORDER_TIME_DONE, SymbolInfoInteger(symbol, SYMBOL_TIME));
        _set(ORDER_TIME_DONE_MSC, SymbolInfoInteger(symbol, SYMBOL_TIME_MSC));
        _set(ORDER_STATE, state);
        ++done;
    }

    bool isActive() const
    {
        return _get<long>(ORDER_TIME_DONE) == 0;
    }

    static int getDoneCount()
    {
        return done;
    }
};

```

请注意，在旧的“当前”时间的虚拟环境中，你不能使用 *TimeCurrent* 函数，而是使用自定义交易品种的最后已知时间 *SymbolInfoInteger(symbol, SYMBOL_TIME)*。

在虚拟交易期间，当前对象及其历史会累积在相应类的数组中。

```

AutoPtr<CustomOrder> orders[];
CustomOrder *selectedOrders[];
CustomOrder *selectedOrder = NULL;
AutoPtr<CustomDeal> deals[];
CustomDeal *selectedDeals[];
CustomDeal *selectedDeal = NULL;
AutoPtr<CustomPosition> positions[];
CustomPosition *selectedPosition = NULL;

```

选择订单、交易和仓位的隐喻是为了在内置函数中模拟类似的方法。对于它们，*CustomTrade* 命名空间中有重复项，这些重复项使用宏替换指令替换了原始项。

```

#define HistorySelect CustomTrade::MT5HistorySelect
#define HistorySelectByPosition CustomTrade::MT5HistorySelectByPosition
#define PositionGetInteger CustomTrade::MT5PositionGetInteger
#define PositionGetDouble CustomTrade::MT5PositionGetDouble
#define PositionGetString CustomTrade::MT5PositionGetString
#define PositionSelect CustomTrade::MT5PositionSelect
#define PositionSelectByTicket CustomTrade::MT5PositionSelectByTicket
#define PositionsTotal CustomTrade::MT5PositionsTotal
#define OrdersTotal CustomTrade::MT5OrdersTotal
#define PositionGetSymbol CustomTrade::MT5PositionGetSymbol
#define PositionGetTicket CustomTrade::MT5PositionGetTicket
#define HistoryDealsTotal CustomTrade::MT5HistoryDealsTotal
#define HistoryOrdersTotal CustomTrade::MT5HistoryOrdersTotal
#define HistoryDealGetTicket CustomTrade::MT5HistoryDealGetTicket
#define HistoryOrderGetTicket CustomTrade::MT5HistoryOrderGetTicket
#define HistoryDealGetInteger CustomTrade::MT5HistoryDealGetInteger
#define HistoryDealGetDouble CustomTrade::MT5HistoryDealGetDouble
#define HistoryDealGetString CustomTrade::MT5HistoryDealGetString
#define HistoryOrderGetDouble CustomTrade::MT5HistoryOrderGetDouble
#define HistoryOrderGetInteger CustomTrade::MT5HistoryOrderGetInteger
#define HistoryOrderGetString CustomTrade::MT5HistoryOrderGetString
#define OrderSend CustomTrade::MT5OrderSend
#define OrderSelect CustomTrade::MT5OrderSelect
#define HistoryOrderSelect CustomTrade::MT5HistoryOrderSelect
#define HistoryDealSelect CustomTrade::MT5HistoryDealSelect

```

例如，这是 *MT5HistorySelectByPosition* 函数的实现方式。

```

bool MT5HistorySelectByPosition(long id)
{
    ArrayResize(selectedOrders, 0);
    ArrayResize(selectedDeals, 0);

    for(int i = 0; i < ArraySize(orders); i++)
    {
        CustomOrder *ptr = orders[i][];
        if(!ptr.isActive())
        {
            if(ptr._get<long>(ORDER_POSITION_ID) == id)
            {
                PUSH(selectedOrders, ptr);
            }
        }
    }

    for(int i = 0; i < ArraySize(deals); i++)
    {
        CustomDeal *ptr = deals[i][];
        if(ptr._get<long>(DEAL_POSITION_ID) == id)
        {
            PUSH(selectedDeals, ptr);
        }
    }
    return true;
}

```

可以看到，该组的所有函数都带有 MT5 前缀，因此它们的双重用途立即清晰可见，并且很容易将它们与第二组函数区分开来。

CustomTrade 命名空间中的第二组函数执行实用操作：检查和更新订单、交易和仓位的状态，根据情况创建新对象并删除旧对象。具体来说，它们包括 **CheckPositions** 和 **CheckOrders** 函数，这些函数可以在计时器上调用或响应用户操作。但是，如果你使用其他几个旨在显示虚拟交易账户当前和历史状态的函数，则可以不这样做：

- **string ReportTradeState()** 返回一个多行文本，其中包含未平仓仓位和已下订单的列表
- **void PrintTradeHistory()** 在日志中显示订单和交易的历史

这些函数独立调用 **CheckPositions** 和 **CheckOrders** 以向你提供最新信息。

此外，还有一个函数用于以对象形式在图表上可视化仓位和有效订单：**DisplayTrades**。

头文件 **CustomTrade.mqh** 应包含在 EA 交易中，位于其他头文件之前，以便宏替换对所有后续的源代码行都有效。

```

#include <MQL5Book/CustomTrade.mqh>
#include <MQL5Book/MqlTradeSync.mqh>

```

现在，上述算法 **CustomOrderSend.mq5** 可以在基于当前自定义交易品种的虚拟环境中开始“交易”（不需要服务器或标准测试器），而无需任何额外更改。

为了快速显示状态，我们将启动第二个计时器并定期更改注释，以及显示图形对象。


```

int OnInit()
{
    EventSetTimer(1);
    return INIT_SUCCEEDED;
}

void OnTimer()
{
    Comment(CustomTrade::ReportTradeState());
    CustomTrade::DisplayTrades();
}

```

为了通过按 **R** 构建报告，我们添加 *OnChartEvent* 处理程序。

```

void OnChartEvent(const int id, const long &lparam, const double &dparam, const string &str)
{
    if(id == CHARTEVENT_KEYDOWN)
    {
        switch((int)lparam)
        {
            ...
            case KEY_R:
                CustomTrade::PrintTradeHistory();
                break;
        }
    }
}

```

最后，一切准备就绪，可以实际测试新的软件包了。

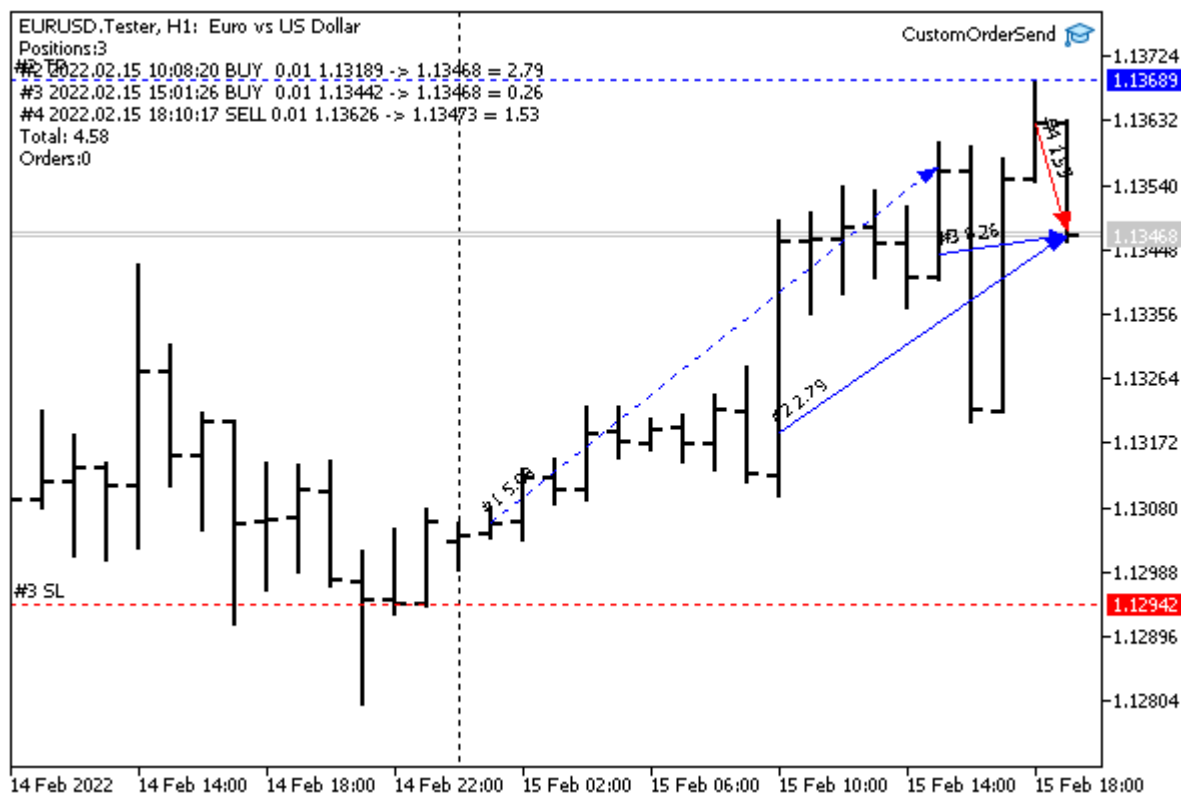
在 EURUSD 上运行自定义交易品种生成器 *CustomTester.mq5*。在打开的 “EURUSD.Tester” 图表上，运行 *CustomOrderSend.mq5* 并开始交易。以下是测试过程的图片。



自定义交易品种图表上的虚拟交易

在这里你可以看到两个未平仓的多头仓位 (带有保护水平) 和一个挂起的卖出限价订单。

一段时间后, 其中一个仓位被平仓 (下面用带箭头的蓝色虚线表示), 一个挂起的卖出订单被触发 (带箭头的红色实线), 从而产生以下图片。



自定义交易品种图表上的虚拟交易

在平掉所有仓位（一些通过止盈，其余通过用户命令）后，通过按 `R` 发出生成报告的命令。

```
History Orders:
(1) #1 ORDER_TYPE_BUY 2022.02.15 01:20:50 -> 2022.02.15 01:20:50 L=0.01 @ 1.1306
(4) #2 ORDER_TYPE_SELL_LIMIT 2022.02.15 02:34:29 -> 2022.02.15 18:10:17 L=0.01 @ 1.13
(2) #3 ORDER_TYPE_BUY 2022.02.15 10:08:20 -> 2022.02.15 10:08:20 L=0.01 @ 1.13189
(3) #4 ORDER_TYPE_BUY 2022.02.15 15:01:26 -> 2022.02.15 15:01:26 L=0.01 @ 1.13442
(1) #5 ORDER_TYPE_SELL 2022.02.15 15:35:43 -> 2022.02.15 15:35:43 L=0.01 @ 1.13568
(2) #6 ORDER_TYPE_SELL 2022.02.16 09:39:17 -> 2022.02.16 09:39:17 L=0.01 @ 1.13724
(4) #7 ORDER_TYPE_BUY 2022.02.16 23:31:15 -> 2022.02.16 23:31:15 L=0.01 @ 1.13748
(3) #8 ORDER_TYPE_SELL 2022.02.16 23:31:15 -> 2022.02.16 23:31:15 L=0.01 @ 1.13742
Deals:
(1) #1 [#1] DEAL_TYPE_BUY DEAL_ENTRY_IN 2022.02.15 01:20:50 L=0.01 @ 1.1306 = 0.00
(2) #2 [#3] DEAL_TYPE_BUY DEAL_ENTRY_IN 2022.02.15 10:08:20 L=0.01 @ 1.13189 = 0.00
(3) #3 [#4] DEAL_TYPE_BUY DEAL_ENTRY_IN 2022.02.15 15:01:26 L=0.01 @ 1.13442 = 0.00
(1) #4 [#5] DEAL_TYPE_SELL DEAL_ENTRY_OUT 2022.02.15 15:35:43 L=0.01 @ 1.13568 = 5.08
(4) #5 [#2] DEAL_TYPE_SELL DEAL_ENTRY_IN 2022.02.15 18:10:17 L=0.01 @ 1.13626 = 0.00
(2) #6 [#6] DEAL_TYPE_SELL DEAL_ENTRY_OUT 2022.02.16 09:39:17 L=0.01 @ 1.13724 = 5.35
(4) #7 [#7] DEAL_TYPE_BUY DEAL_ENTRY_OUT 2022.02.16 23:31:15 L=0.01 @ 1.13748 = -1.22
(3) #8 [#8] DEAL_TYPE_SELL DEAL_ENTRY_OUT 2022.02.16 23:31:15 L=0.01 @ 1.13742 = 3.00
Total: 12.21, Trades: 4
```

圆括号表示仓位标识符，方括号表示相应交易的订单凭证（两种类型的订单号前都有一个井号'#'）。

这里不考虑掉期和佣金。可以添加对掉期和佣金的计算。

我们将在 [自定义交易品种交易细节](#) 一节中探讨另一个示例，该示例使用自定义交易品种分时报价。我们将讨论如何创建等量图。

7.2.7 转换订单簿变更

如有必要，MQL 程序可以使用 `CustomBookAdd` 函数为自定义交易品种生成订单簿。这尤其适用于来自外部交易所的金融工具，例如加密货币。

```
int CustomBookAdd(const string symbol, const MqlBookInfo &books[], uint count = WHOLE_ARRAY)
```

该函数使用 `books` 数组中的数据，将订单簿的状态广播到使用自定义 `symbol` 的已签名 MQL 程序。该数组描述了订单簿的完整状态，即所有买入和卖出订单。转换后的状态完全取代先前的状态，并可通过 `MarketBookGet` 函数获取。

使用 `count` 参数，可以指定要传递给函数的 `books` 数组的元素数量。默认情况下使用整个数组。

该函数返回操作状态标识：成功 (`true`) 或错误 (`false`)。

要获取由 `CustomBookAdd` 函数生成的订单簿，需要这些订单簿的 MQL 程序必须像往常一样使用 `MarketBookAdd` 来订阅事件。

订单簿的更新不会更新金融工具的 `Bid` 和 `Ask`。要更新所需的价格，请使用 `CustomTicksAdd` 添加分时报价。

系统会检查传输数据的正确性：价格和交易量必须大于零，并且对于每个元素，都必须指定其类型、价格和交易量（字段 *volume* 和/或 *volume_real*）。如果只要订单簿中有一个元素描述不正确，该函数就会返回错误。

系统也会检查自定义金融工具的订单簿深度参数（*SYMBOL_TICKS_BOOKDEPTH*）。如果转换后的订单簿中的卖出或买入级别数量超过此值，则多余的级别将被丢弃。

具有更高精度的交易量 *volume_real* 优先于普通 *volume*。如果为订单簿元素同时指定了这两个值，则将使用 *volume_real*。

注意！在当前实现中，*CustomBookAdd* 会自动锁定自定义交易品种，就好像它是由 *MarketBookAdd* 订阅的一样，但同时，*OnBookEvent* 事件不会到达（理论上，生成订单簿的程序可以通过显式调用 *MarketBookAdd* 来订阅它们，并控制其他程序接收的内容）。你可以通过调用 *MarketBookRelease* 来移除此锁定。

这可能是必需的，因为对于已订阅订单簿的交易品种，无论如何都无法将其从 *Market Watch* 中隐藏（直到所有显式或隐式订阅从程序中取消，并且订单簿窗口关闭）。因此，此类交易品种无法删除。

例如，我们创建一个非交易性 EA 交易 *PseudoMarketBook.mq5*，它将从最近的分时报价历史中生成订单簿的伪状态。这对于那些不转换订单簿的交易品种（尤其是外汇）可能很有用。如果你愿意，可以使用此类自定义交易品种对你自己的使用订单簿的交易算法进行形式化调试。

在输入参数中，我们指示订单簿的最大深度。

```
input uint CustomBookDepth = 20;
```

自定义交易品种的名称将通过在当前图表交易品种的名称后添加后缀 ".Pseudo" 来形成。

```
string CustomSymbol = _Symbol + ".Pseudo";
```

在 *OnInit* 处理程序中，我们创建一个自定义交易品种，并将其公式设置为原始交易品种的名称。因此，我们将获得一个由终端自动更新的原始交易品种的副本，并且我们无需费心复制报价或分时报价。

```
int OnInit()
{
    bool custom = false;
    if(!PRTF(SymbolExist(CustomSymbol, custom)))
    {
        if(PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
        {
            CustomSymbolSetString(CustomSymbol, SYMBOL_DESCRIPTION, "Pseudo book generated");
            CustomSymbolSetString(CustomSymbol, SYMBOL_FORMULA, "\"\" + _Symbol + "\"");
        }
    }
    ...
}
```

如果自定义交易品种已存在，EA 交易可以向用户提议删除它并在那里完成工作（用户应首先关闭所有带有此交易品种的图表）。

```

else
{
    if(IDYES == MessageBox(StringFormat("Delete existing custom symbol '%s'?",
        CustomSymbol), "Please, confirm", MB_YESNO))
    {
        PRTF(MarketBookRelease(CustomSymbol));
        PRTF(SymbolSelect(CustomSymbol, false));
        PRTF(CustomRatesDelete(CustomSymbol, 0, LONG_MAX));
        PRTF(CustomTicksDelete(CustomSymbol, 0, LONG_MAX));
        if(!PRTF(CustomSymbolDelete(CustomSymbol)))
        {
            Alert("Can't delete ", CustomSymbol, ", please, check up and delete manually");
        }
        return INIT_PARAMETERS_INCORRECT;
    }
}
...

```

此交易品种的一个特殊功能是设置 `SYMBOL_TICKS_BOOKDEPTH` 特性，以及读取合约大小 `SYMBOL_TRADE_CONTRACT_SIZE`，这在生成交易量时是必需的。

```

if(SymbolInfoInteger(_Symbol, SYMBOL_TICKS_BOOKDEPTH) != CustomBookDepth
&& SymbolInfoInteger(CustomSymbol, SYMBOL_TICKS_BOOKDEPTH) != CustomBookDepth)
{
    Print("Adjusting custom market book depth");
    CustomSymbolSetInteger(CustomSymbol, SYMBOL_TICKS_BOOKDEPTH, CustomBookDepth);
}

depth = (int)PRTF(SymbolInfoInteger(CustomSymbol, SYMBOL_TICKS_BOOKDEPTH));
contract = PRTF(SymbolInfoDouble(CustomSymbol, SYMBOL_TRADE_CONTRACT_SIZE));

return INIT_SUCCEEDED;
}

```

该算法在 *OnTick* 处理程序中启动。在这里，我们调用了 *GenerateMarketBook* 函数，但这个函数尚未编写。它将填充通过引用传递的 *MqlBookInfo* 结构体数组，然后我们将使用 *CustomBookAdd* 将其发送到自定义交易品种。

```

void OnTick()
{
    MqlBookInfo book[];
    if(GenerateMarketBook(2000, book))
    {
        ResetLastError();
        if(!CustomBookAdd(CustomSymbol, book))
        {
            Print("Can't add market books, ", E2S(_LastError));
            ExpertRemove();
        }
    }
}

```

GenerateMarketBook 函数会分析最新的 *count* 个分时报价，并基于它们模拟订单簿的可能状态，遵循以下假设：

- 已买入的可能会被卖出
- 已卖出的可能会被买入

在一般情况下 (在没有交易所标志的情况下)，将分时报价划分为对应于买入和卖出的部分，可以通过价格本身的变动来估计：

- **Ask**向上变动被视为买入
- **Bid**向下变动被视为卖出

结果，我们得到以下算法。

```

bool GenerateMarketBook(const int count, MqlBookInfo &book[])
{
    MqlTick tick; // order book centre
    if(!SymbolInfoTick(_Symbol, tick)) return false;

    double buys[]; // buy volumes by price levels
    double sells[]; // sell volumes by price levels

    MqlTick ticks[];
    CopyTicks(_Symbol, ticks, COPY_TICKS_ALL, 0, count); // request tick history
    for(int i = 1; i < ArraySize(ticks); ++i)
    {
        // we believe that ask was pushed up by buys
        int k = (int)MathRound((tick.ask - ticks[i].ask) / _Point);
        if(ticks[i].ask > ticks[i - 1].ask)
        {
            // already bought, probably will take profit by selling
            if(k <= 0)
            {
                Place(sells, -k, contract / sqrt(sqrt(ArraySize(ticks) - i)));
            }
        }

        // believe that the bid was pushed down by sells
        k = (int)MathRound((tick.bid - ticks[i].bid) / _Point);
        if(ticks[i].bid < ticks[i - 1].bid)
        {
            // already sold, probably will take profit by buying
            if(k >= 0)
            {
                Place(buys, k, contract / sqrt(sqrt(ArraySize(ticks) - i)));
            }
        }
    }
    ...
}

```

辅助函数 *Place* 会填充 *buys* 和 *sells* 数组，按价格水平在其中累积交易量。我们将在下面展示这一点。数组中的索引定义为与当前最优价格 (*Bid*或*Ask*)的点数距离。交易量的大小与分时报价的“时效性”成反比，即越久远的分时报价，影响力就较小。

填充数组后，系统基于它们形成一个 *MqlBookInfo* 结构体数组。

```

for(int i = 0, k = 0; i < ArraySize(sells) && k < depth; ++i) // top half of the c
{
    if(sells[i] > 0)
    {
        MqlBookInfo info = {};
        info.type = BOOK_TYPE_SELL;
        info.price = tick.ask + i * _Point;
        info.volume = (long)sells[i];
        info.volume_real = (double)(long)sells[i];
        PUSH(book, info);
        ++k;
    }
}

for(int i = 0, k = 0; i < ArraySize(buys) && k < depth; ++i) // bottom half of the
{
    if(buys[i] > 0)
    {
        MqlBookInfo info = {};
        info.type = BOOK_TYPE_BUY;
        info.price = tick.bid - i * _Point;
        info.volume = (long)buys[i];
        info.volume_real = (double)(long)buys[i];
        PUSH(book, info);
        ++k;
    }
}

return ArraySize(book) > 0;
}

```

Place 函数很简单。

```

void Place(double &array[], const int index, const double value = 1)
{
    const int size = ArraySize(array);
    if(index >= size)
    {
        ArrayResize(array, index + 1);
        for(int i = size; i <= index; ++i)
        {
            array[i] = 0;
        }
    }
    array[index] += value;
}

```

以下屏幕截图显示了运行 *PseudoMarketBook.mq5* EA 交易的 EURUSD 图表，以及生成的订单簿版本。



基于 EURUSD 的自定义交易品种的综合订单簿

7.2.8 自定义交易品种交易细节

自定义交易品种仅为客户端终端所知，在交易服务器上不可用。因此，如果自定义交易品种是基于某个真实交易品种构建的，那么放置在此类自定义交易品种图表上的任何 EA 交易都应视为原始交易品种生成交易订单。

作为此问题的最简单解决方案，你可以将 EA 交易放置在原始交易品种的图表上，但从自定义交易品种接收信号（例如，从指标）。另一种显而易见的方法是在执行交易操作时替换交易品种的名称。为了测试这两种方法，我们需要一个自定义交易品种和一个 EA 交易。

作为自定义交易品种的一个有趣的实际示例，我们采用几种不同的等量图。

等量图是一种基于其中包含的交易量相等的原则构建的柱线图。在常规图表上，每个新柱线都以指定的频率形成，与时间范围大小一致。在等量图上，当分时报价或实际交易量的总和达到预设值时，每个柱线被视为已形成。此时，程序开始计算下一个柱线的数量。当然，在计算交易量的过程中，会控制价格变动，我们在图表上得到通常的价格集：*Open*、*High*、*Low*和*Close*。

等距柱线的构建方式类似：当价格向任一方向移动给定点数时，会打开一个新的柱线。

因此，*EqualVolumeBars.mq5* EA 交易将支持三种模式，即三种图表类型：

- *EqualTickVolumes* -按分时报价计数的等量柱线
- *EqualRealVolumes* -按实际交易量计数的等量柱线（如果它们被广播）
- *RangeBars* -等距柱线

它们通过输入参数 *WorkMode* 进行选择。

柱线大小和用于计算的历史深度在 *TicksInBar* 和 *StartDate* 参数中指定。

```
input int TicksInBar = 1000;
input datetime StartDate = 0;
```

根据模式，自定义交易品种将分别获得后缀 “_Eqv”、“_Qrv” 或 “_Rng”，并附加柱线大小。

尽管等量/等距图上的水平轴仍然表示时间顺序，但每个柱线的时间戳是任意的，并且取决于每个时间范围内的波动性（交易笔数或交易规模）。在这方面，选择自定义交易品种图表的时间范围时应使其等于最小的 M1。

平台的限制是所有柱线都具有相同的名义持续时间，但在我们的“人工”图表的情况下，应该记住每个柱线的实际持续时间并不相同，并且可能远远超过 1 分钟，或者相反，远远少于 1 分钟。因此，如果一个柱线的给定交易量足够小，则可能会出现如下情况：新柱线的形成频率远高于每分钟一次，此时自定义交易品种柱线的虚拟时间将领先于现实时间，进入未来。为防止这种情况发生，你应该增加柱线的交易量（*TicksInBar* 参数）或将旧柱线向左移动。

用于管理自定义交易品种的初始化和其他辅助任务（特别是重置现有历史记录以及打开带有新交易品种的图表）的执行方式与其他示例类似，我们将在此省略它们。我们转向具有应用性质的细节。

我们将使用内置函数 *CopyTicks/CopyTicksRange* 读取实际分时报价的历史记录：第一个函数用于以 10,000 个分时报价为一批次交换历史记录，第二个函数用于请求自上次处理以来的新分时报价。所有这些功能都打包在 *TicksBuffer* 类中（随附了完整的源代码）。

```
class TicksBuffer
{
private:
    MqlTick array[]; // internal array of ticks
    int tick;        // incremental index of the next tick for reading
public:
    bool fill(ulong &cursor, const bool history = false);
    bool read(MqlTick &t);
};
```

公共方法 *fill* 用于从 *cursor* 时间（以毫秒为单位）开始，用下一部分分时报价填充内部数组。同时，每次调用时 *cursor* 中的时间都会根据读入缓冲区的最后一个分时报价的时间向前移动（请注意，该参数是通过引用传递的）。

参数 *history* 确定是使用 *CopyTicks* 还是 *CopyTicksRange*。通常，如果是在线，我们将从 *OnTick* 处理程序中读取一个或多个新的分时报价。

read 方法从内部数组中返回一个分时报价，并将内部指针（*tick*）移至下一个分时报价。如果在读取时到达数组末尾，该方法将返回 *false*，这意味着是时候调用 *fill* 方法了。

使用这些方法，分时报价历史旁路算法的实现如下（此代码间接从 *OnInit* 通过计时器调用）。

```

ulong cursor = StartDate * 1000;
TicksBuffer tb;

while(tb.fill(cursor, true) && !IsStopped())
{
    MqlTick t;
    while(tb.read(t))
    {
        HandleTick(t, true);
    }
}

```

在 *HandleTick* 函数中，在某些用于控制分时报价数量、总交易量（实际交易量，如果有）以及价格移动距离的全局变量中，必须要考虑分时报价 *t* 的特性。根据操作模式，应以不同方式分析这些变量以确定新柱线形成的条件。因此，如果在等量模式下，分时报价数量超过 *TicksInBar*，我们应该通过将计数器重置为 1 来开始一个新的柱线。在这种情况下，新柱线的时间计为四舍五入到最接近分钟的分时报价时间。

这组全局变量用于存储自定义交易品种上最后（“当前”）柱线的虚拟时间（*now_time*）、其 OHLC 价格和交易量。

```

datetime now_time;
double now_close, now_open, now_low, now_high;
long now_volume, now_real;

```

无论是在历史记录读取期间，还是稍后当 EA 交易开始实时处理在线分时报价时（我们稍后会回到这一点），这些变量都会不断更新。

以某种简化的形式，*HandleTick* 内部的算法如下所示：

```

void HandleTick(const MqlTick &t, const bool history = false)
{
    now_volume++;           // count the number of ticks
    now_real += (long)t.volume; // sum up all real volumes

    if(!IsNewBar()) // continue the current bar
    {
        if(t.bid < now_low) now_low = t.bid; // monitor price fluctuations downward
        if(t.bid > now_high) now_high = t.bid; // and upwards
        now_close = t.bid; // update the closing price

        if(!history)
        {
            // update the current bar if we are not in the history
            WriteToChart(now_time, now_open, now_low, now_high, now_close,
                now_volume - !history, now_real);
        }
    }
    else // new bar
    {
        do
        {
            // save the closed bar with all attributes
            WriteToChart(now_time, now_open, now_low, now_high, now_close,
                WorkMode == EqualTickVolumes ? TicksInBar : now_volume,
                WorkMode == EqualRealVolumes ? TicksInBar : now_real);

            // round up the time to the minute for the new bar
            datetime time = t.time / 60 * 60;

            // prevent bars with old or same time
            // if gone to the "future", we should just take the next count M1
            if(time <= now_time) time = now_time + 60;

            // start a new bar from the current price
            now_time = time;
            now_open = t.bid;
            now_low = t.bid;
            now_high = t.bid;
            now_close = t.bid;
            now_volume = 1; // first tick in the new bar
            if(WorkMode == EqualRealVolumes) now_real -= TicksInBar;
            now_real += (long)t.volume; // initial real volume in the new bar

            // save new bar 0
            WriteToChart(now_time, now_open, now_low, now_high, now_close,
                now_volume - !history, now_real);
        }
        while(IsNewBar() && WorkMode == EqualRealVolumes);
    }
}

```

history 参数确定计算是基于历史记录还是已经实时进行（基于传入的在线分时报价）。如果基于历史记录，则每个柱线只需形成一次即可，而在线情况下，当前柱线会随着每个分时报价而更新。这可以加快历史记录的处理速度。

当满足根据模式关闭下一个柱线的条件时，辅助函数 *IsNewBar* 返回 *true*。

```
bool IsNewBar()
{
    if(WorkMode == EqualTickVolumes)
    {
        if(now_volume > TicksInBar) return true;
    }
    else if(WorkMode == EqualRealVolumes)
    {
        if(now_real > TicksInBar) return true;
    }
    else if(WorkMode == RangeBars)
    {
        if((now_high - now_low) / _Point > TicksInBar) return true;
    }

    return false;
}
```

WriteToChart 函数通过调用 *CustomRatesUpdate* 创建具有给定特征的柱线。

```
void WriteToChart(datetime t, double o, double l, double h, double c, long v, long m)
{
    MqlRates r[1];

    r[0].time = t;
    r[0].open = o;
    r[0].low = l;
    r[0].high = h;
    r[0].close = c;
    r[0].tick_volume = v;
    r[0].spread = 0;
    r[0].real_volume = m;

    if(CustomRatesUpdate(SymbolName, r) < 1)
    {
        Print("CustomRatesUpdate failed: ", _LastError);
    }
}
```

上述读取和处理分时报价的循环是在初始访问历史记录期间，在创建或完成重新计算已存在的用户交易品种之后执行的。当涉及到新的分时报价时，*OnTick* 函数使用类似的代码，但没有“历史性”标志。

```

void OnTick()
{
    static ulong cursor = 0;
    MqlTick t;

    if(cursor == 0)
    {
        if(SymbolInfoTick(_Symbol, t))
        {
            HandleTick(t);
            cursor = t.time_msc + 1;
        }
    }
    else
    {
        TicksBuffer tb;
        while(tb.fill(cursor))
        {
            while(tb.read(t))
            {
                HandleTick(t);
            }
        }
    }

    RefreshWindow(now_time);
}

```

RefreshWindow 函数在 **Market Watch** 窗口中添加一个自定义交易品种的分时报价。

请注意，分时报价转发会使柱线中的分时报价计数器增加 1，因此，当将分时报价计数器写入第 0 个柱线时，我们之前减去了 1（参见调用 **WriteToChart** 时的表达式 **now_volume - !history**）。

生成分时报价很重要，因为它会触发自定义金融工具图表上的 **OnTick** 事件，这可能允许放置在此类图表上的 EA 交易进行交易。然而，这项技术需要一些额外的技巧，我们稍后会探讨。

```

void RefreshWindow(const datetime t)
{
    MqlTick ta[1];
    SymbolInfoTick(_Symbol, ta[0]);
    ta[0].time = t;
    ta[0].time_msc = t * 1000;
    if(CustomTicksAdd(SymbolName, ta) == -1)
    {
        Print("CustomTicksAdd failed:", _LastError, " ", (long) ta[0].time);
        ArrayPrint(ta);
    }
}

```

我们需要强调的是，生成的自定义分时报价的时间始终设置为当前柱线的标签，因为我们不能保留真实的分时报价时间：如果它已经提前超过 1 分钟，并且我们将这样的分时报价发送到 **Market Watch**，终端将创建下一个 M1 柱线，这将违反我们的“等量”结构，因为我们的柱线不是按时间形成的，而是按交易量填充形成的（并且我们自己控制这个过程）。

理论上，我们可以为每个分时报价增加一毫秒，但我们无法保证柱线不需要存储超过 60,000 个分时报价（例如，如果用户订购一个具有特定价格范围的图表，而在该特定价格范围内，我们无法预测此类波动具体需要多少个分时报价）。

在按交易量计算的模式中，理论上可以使用线性公式对分时报价时间的秒和毫秒分量进行插值：

- `EqualTickVolumes - (now_volume - 1) * 60000 / TicksInBar;`
- `EqualRealVolumes - (now_real - 1) * 60000 / TicksInBar;`

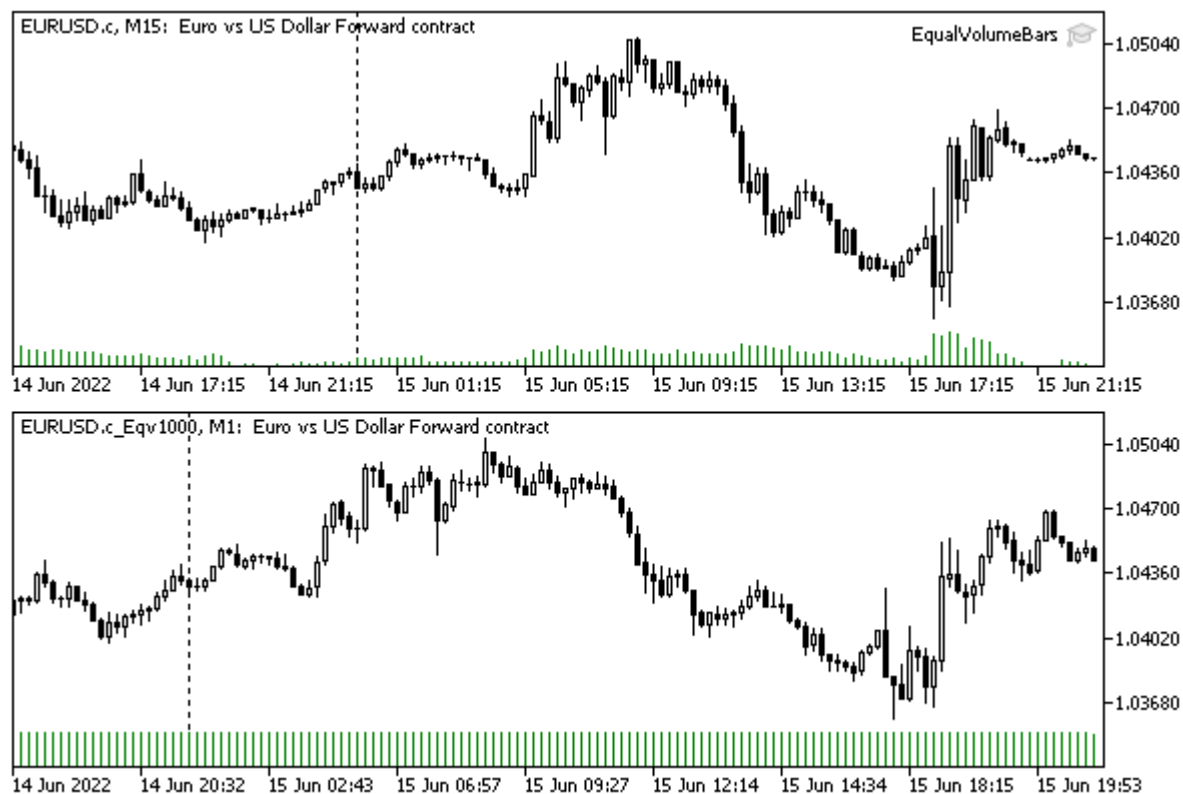
然而，这只不过是识别分时报价的一种手段，而不是试图使“人工”分时报价的时间更接近真实分时报价的时间。这不仅仅是真实分时报价流不均匀性的损失，这本身就会导致原始交易品种与据其生成的自定义交易品种之间的价格差异。

主要问题是需要在 M1 柱线的边界处对分时报价时间进行取整，并将其“打包”在一分钟内（请参见关于特殊类型图表的补充说明）。例如，下一个实际时间为 12:37:05'123 的分时报价成为第 1001 个分时报价，并且应该形成一个新的等量柱线。但是，M1 柱线只能标记到分钟，比如 12:37。结果，该金融工具在 12:37 的实际价格将与为等量柱线 12:37 提供 *Open* 的分时报价中的价格不匹配。此外，如果接下来的 1000 个分时报价持续几分钟，我们还是必须得压缩这些分时报价的时间，以免达到 12:38 标记。

当通过标准 M1 时间周期图表模拟特殊图表时，由于时间量化，该问题具有系统性。在这样的图表上无法完全解决此问题。但是，当以连续时间生成带有分时报价的自定义交易品种时（例如，使用合成行情或基于来自外部服务的流数据），则不会出现此问题。

务必要注意，分时报价转发仅在此版本的生成器中在线完成，而在历史记录中不会生成自定义分时报价！这样做是为了加快行情的创建速度。如果你需要在较慢的过程中生成分时报价历史记录，则应调整 EA 交易 *EqualVolumeBars.mq5*：排除 *WriteToChart* 函数，并使用 *CustomTicksReplace/CustomTicksAdd* 执行整个生成过程。同时，应该记住，应将分时报价的原始时间替换为另一个时间，在一分钟柱线内，以免扰乱形成的等量图的结构。

我们来看看 *EqualVolumeBars.mq5* 是如何运作的。这是 EURUSD M15 的工作图表，其中运行着 EA 交易。它具有等量图，其中每个柱线分配了 1000 个分时报价。



由 EqualVolumeBars EA 交易生成的等量 EURUSD 图表，每柱线 1000 个分时报价

请注意，除最后一个仍在形成（分时报价计数仍在继续）的柱线外，所有柱线上的分时报价量均相等。

统计信息显示在日志中。

```
Creating "EURUSD.c_Eqv1000"
Processing tick history...
End of CopyTicks at 2022.06.15 12:47:51
Bar 0: 2022.06.15 12:40:00 866 0
2119 bars written in 10 sec
Open "EURUSD.c_Eqv1000" chart to view results
```

我们检查另一种操作模式：等距。下图是一个图表，其中每个柱线的距离是 250 点。



由 EqualVolumeBars 生成的每柱线 250 点的 EURUSD 等距图表

对于交易所金融工具，EA 交易允许使用实际交易量模式，例如，如下所示：



每柱线实际交易量为 10000 的以太坊原始图表和等量图表

放置 EA 交易生成器时工作交易品种的时间范围并不重要，因为计算始终使用分时报价历史记录。

同时，自定义交易品种图表的时间范围必须等于 M1 (终端中可用的最小时间范围)。因此，柱线的时间通常尽可能地 (在可能的范围内) 匹配其形成时刻。然而，在市场剧烈波动期间，当分时报价数量或交易量大小每分钟形成几个柱线时，柱线的时间将早于实际时间。当市场波动幅度较小时，等量柱线的时间标记情况将恢复正常。这不会影响在线价格的流动，因此可能并不特别重要，因为使用等量或等距柱线的全部意义在于与绝对时间脱钩。

不幸的是，原始交易品种的名称和据其上创建的自定义交易品种的名称无法通过平台本身以任何方式联系起来。如果在自定义交易品种的特性中有一个字符串字段 "origin" (来源)，我们可以在其中写入真实工作工具的名称，那将会很方便。该字段默认为空，但如果填写，平台可以在所有交易订单和历史请求中替换交易品种，并且对用户来说是自动且透明的。理论上，在用户定义交易品种的特性中，有一个 SYMBOL_BASIS 字段在含义上是合适的，但由于我们无法保证用户定义交易品种的任意生成器 (任何 MQL 程序) 会正确填写它或完全用于此目的，因此我们不能依赖它的使用。

由于平台中没有这种机制，我们需要自己实现它。由于必须使用参数来设置源交易品种和用户交易品种名称之间的对应关系，

为了解决这个问题，我们开发了 *CustomOrder* 类 (请参见附加文件 *CustomOrder.mqh*)。它包含所有与发送交易订单和请求历史相关的 MQL API 函数的包装器方法，这些函数都有一个带有交易品种名称的字符串参数。在这些方法中，自定义交易品种被替换为当前工作交易品种，反之亦然。其他 API 函数不需要 "挂钩"。下面是一个片段。

```

class CustomOrder
{
private:
    static string workSymbol;

    static void replaceRequest(MqlTradeRequest &request)
    {
        if(request.symbol == _Symbol && workSymbol != NULL)
        {
            request.symbol = workSymbol;
            if(MQLInfoInteger(MQL_TESTER)
                && (request.type == ORDER_TYPE_BUY
                    || request.type == ORDER_TYPE_SELL))
            {
                if(TU::Equal(request.price, SymbolInfoDouble(_Symbol, SYMBOL_ASK)))
                    request.price = SymbolInfoDouble(workSymbol, SYMBOL_ASK);
                if(TU::Equal(request.price, SymbolInfoDouble(_Symbol, SYMBOL_BID)))
                    request.price = SymbolInfoDouble(workSymbol, SYMBOL_BID);
            }
        }
    }

public:
    static void setReplacementSymbol(const string replacementSymbol)
    {
        workSymbol = replacementSymbol;
    }

    static bool OrderSend(MqlTradeRequest &request, MqlTradeResult &result)
    {
        replaceRequest(request);
        return ::OrderSend(request, result);
    }
    ...
}

```

请注意，主要工作方法 *replaceRequest* 不仅会替换交易品种，还会替换当前的 *Ask* 和 *Bid*。这是因为许多自定义工具，例如我们的等量图，其虚拟时间与真实原型交易品种的时间不同。因此，由测试程序模拟的自定义金融工具的价格与真实金融工具的相应价格不同步。

这种伪影现象仅在测试程序中出现。在线交易时，自定义交易品种图表将与真实图表同步更新（价格上），尽管柱线标签会有所不同（一个“人工”M1 柱线的实际持续时间或多或少于一分钟，并且其倒计时时间不是一分钟的倍数）。因此，这种价格转换更多的是一种预防措施，以避免在测试程序中出现重新报价。但是，在测试程序中，我们通常不需要进行交易品种替换，因为测试程序可以与自定义交易品种进行交易（与经纪商的服务器不同）。此外，完全是出于兴趣，我们将比较在有和没有字符替换的情况下运行的测试结果。

为了最大限度地减少对客户端源代码的编辑，提供了以下形式的全局函数和宏（适用于所有 *CustomOrder* 方法）：

```

bool CustomOrderSend(const MqlTradeRequest &request, MqlTradeResult &result)
{
    return CustomOrder::OrderSend((MqlTradeRequest)request, result);
}

#define OrderSend CustomOrderSend

```

它们允许将所有标准 API 函数调用自动重定向到 *CustomOrder* 类方法。为此，只需将 *CustomOrder.mqh* 纳入 EA 交易中，并设置工作交易品种，例如，在 *WorkSymbol* 参数中：

```

#include <CustomOrder.mqh>
#include <Expert/Expert.mqh>
...
input string WorkSymbol = "";

int OnInit()
{
    if(WorkSymbol != "")
    {
        CustomOrder::setReplacementSymbol(WorkSymbol);

        // initiate the opening of the chart tab of the working symbol (in the visual m
        MqlRates rates[1];
        CopyRates(WorkSymbol, PERIOD_CURRENT, 0, 1, rates);
    }
    ...
}

```

重要的是，指令 `#include<CustomOrder.mqh>` 是第一个，在其他指令之前。因此，它会影响所有源代码，包括 MetaTrader 5 发行版中的标准库。如果未指定替换交易品种，则连接的 *CustomOrder.mqh* 对 EA 交易没有影响，并“透明地”将控制权转移给标准 API 函数。

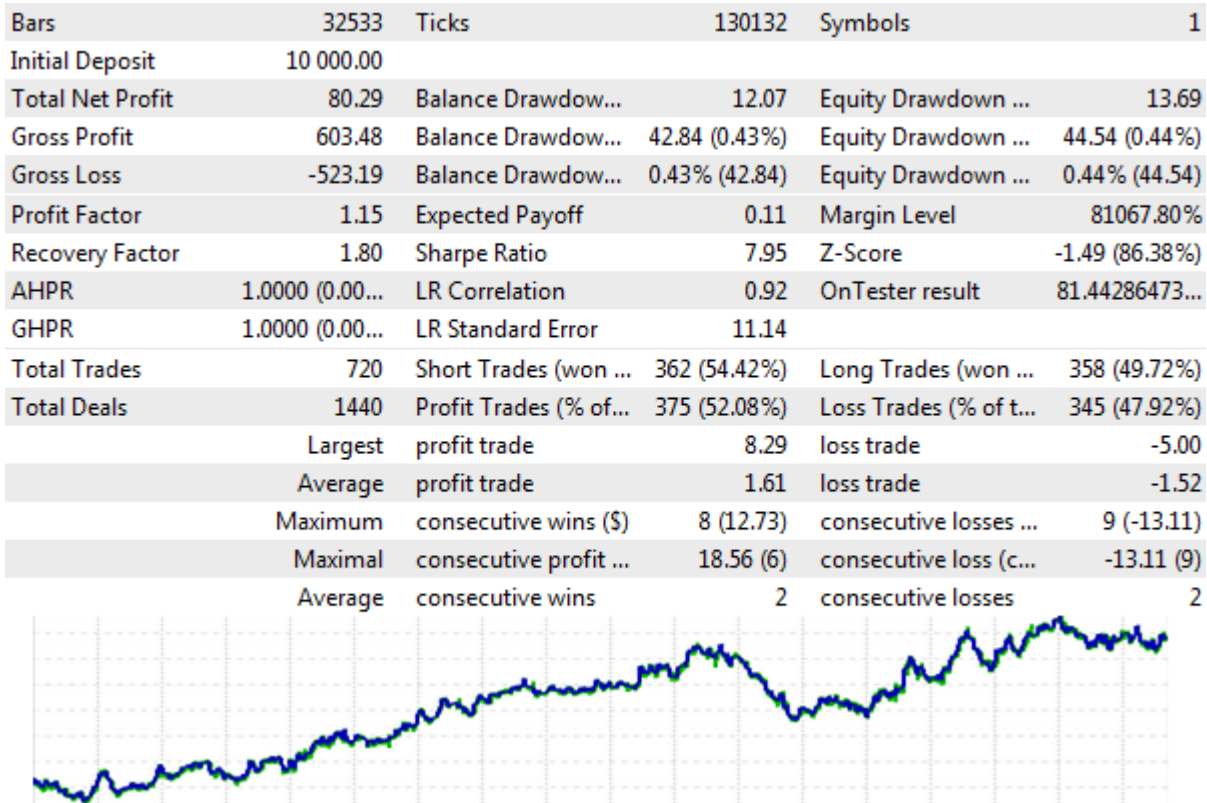
现在我们已经准备好测试在自定义交易品种上进行交易的想法，包括自定义交易品种本身。

应用上述技术，我们修改了已经熟悉的 EA 交易 *BandOsMaPro*，将其重命名为 *BandOsMaCustom.mq5*。我们在 EURUSD 等量图上对其进行测试，该图表的柱线大小为 1000 个分时报价，使用 *EqualVolumeBars.mq5* 获得。

优化或测试模式设置为 OHLC M1 价格（更准确的方法没有意义，因为我们没有生成分时报价，还因此版本以形成的柱线价格进行交易）。日期范围是整个 2021 年和 2022 年上半年。附加了带有 *BandOsMACustom.set* 设置的文件。

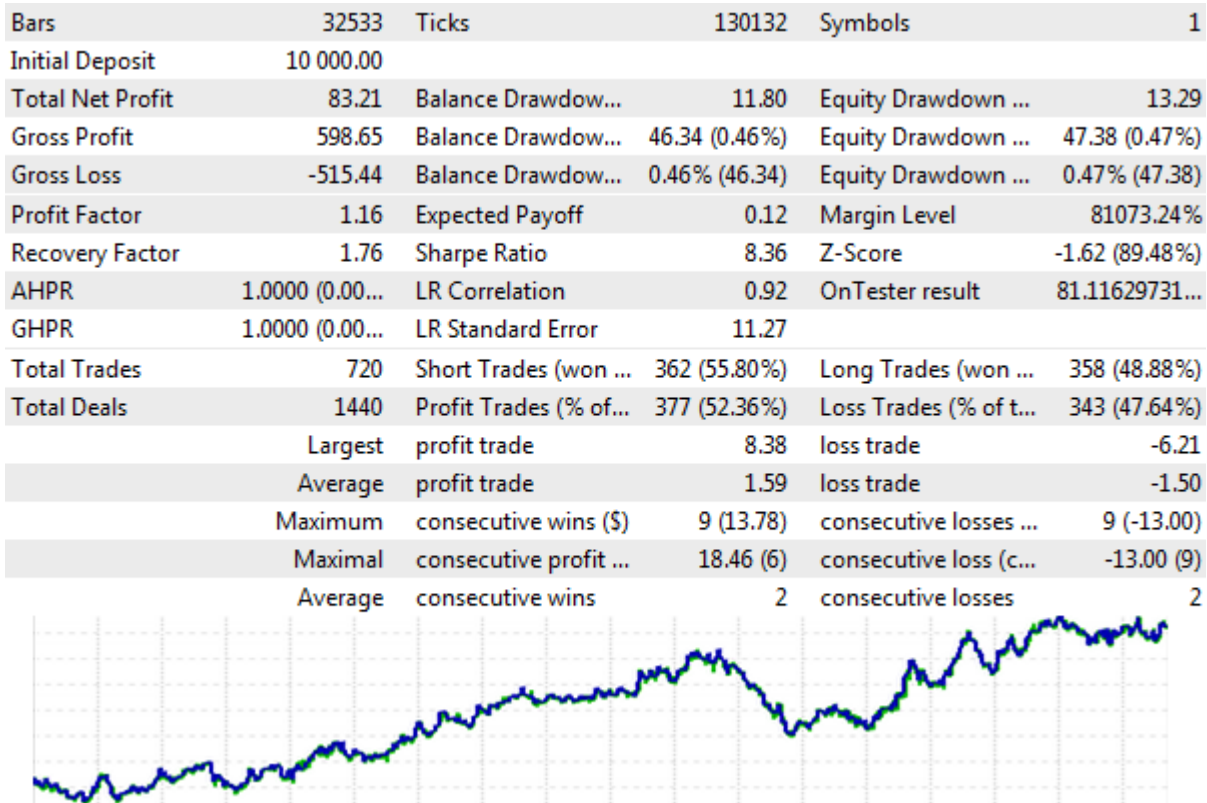
在测试程序设置中，别忘了选择自定义交易品种 EURUSD_Eqv1000 和 M1 时间范围，因为等量柱线是基于该交易品种模拟的。

当 *WorkSymbol* 参数为空时，EA 交易可交易自定义交易品种。结果如下：



在 EURUSD_Eqv1000 等量图上进行交易时的测试程序报告

如果 *WorkSymbol* 参数等于 EURUSD，则 EA 交易交易 EURUSD 货币对，尽管它在 EURUSD_Eqv1000 图表上工作。结果有所不同，但差异不大。



从 EURUSD_Eqv1000 等量图交易 EURUSD 时的测试程序报告

然而，正如本节开头已经提到的，对于依赖指标信号进行交易的 EA 交易来说，有一种更简单的方法来支持自定义交易品种。为此，只需在自定义交易品种上创建指标，并将 EA 交易放置在工作交易品种的图表上即可。

我们可以轻松实现此选项。我们将其称为 *BandOsMACustomSignal.mq5*。

不再需要头文件 *CustomOrder.mqh*。我们添加了两个新的输入参数来代替 *WorkSymbol* 输入参数：

```
input string SignalSymbol = "";
input ENUM_TIMEFRAMES SignalTimeframe = PERIOD_M1;
```

应将它们传递给管理指标的 *BandOsMaSignal* 类的构造函数。以前，到处都使用 *_Symbol* 和 *_Period*。

```
interface TradingSignal
{
    virtual int signal(void);
    virtual string symbol();
    virtual ENUM_TIMEFRAMES timeframe();
};

class BandOsMaSignal: public TradingSignal
{
    int hOsMA, hBands, hMA;
    int direction;
    const string _symbol;
    const ENUM_TIMEFRAMES _timeframe;
public:
    BandOsMaSignal(const string s, const ENUM_TIMEFRAMES tf,
        const int fast, const int slow, const int signal, const ENUM_APPLIED_PRICE pric,
        const int bands, const int shift, const double deviation,
        const int period, const int x, ENUM_MA_METHOD method): _symbol(s), _timeframe(tf)
    {
        hOsMA = iOsMA(s, tf, fast, slow, signal, price);
        hBands = iBands(s, tf, bands, shift, deviation, hOsMA);
        hMA = iMA(s, tf, period, x, method, hOsMA);
        direction = 0;
    }
    ...
    virtual string symbol() override
    {
        return _symbol;
    }

    virtual ENUM_TIMEFRAMES timeframe() override
    {
        return _timeframe;
    }
}
```

由于信号的交易品种和时间周期现在可以与图表的交易品种和周期不同，因此我们通过添加读取方法扩展了 *TradingSignal* 接口。实际值在 *OnInit* 中传递给构造函数。

```
int OnInit()
{
    ...
    strategy = new SimpleStrategy(
        new BandOsMaSignal(SignalSymbol != "" ? SignalSymbol : _Symbol,
            SignalSymbol != "" ? SignalTimeframe : _Period,
            p.fast, p.slow, SignalOsMA, PriceOsMA,
            BandsMA, BandsShift, BandsDeviation,
            PeriodMA, ShiftMA, MethodMA),
        Magic, StopLoss, Lots);
    return INIT_SUCCEEDED;
}
```

在 *SimpleStrategy* 类中，*trade* 方法现在根据信号的特性而不是当前图表来检查新柱线的出现。

```
virtual bool trade() override
{
    // looking for a signal once at the opening of the bar of the desired symbol an
    if(lastBar == iTime(command[].symbol(), command[].timeframe(), 0)) return false

    int s = command[].signal(); // get signal
    ...
}
```

对于具有相同设置的比较实验，应在 EURUSD 上启动 EA 交易 *BandOsMACustomSignal.mq5* (你可以使用 M1 或其他时间范围)，并且应在 *SignalSymbol* 参数中指定 EURUSD_Eqv1000。*SignalTimeframe* 默认应等于 PERIOD_M1。结果，我们将得到一个类似的报告。

Bars	545621	Ticks	2159010	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	77.36	Balance Drawdown...	12.97	Equity Drawdown ...	14.60
Gross Profit	601.63	Balance Drawdown...	45.33 (0.45%)	Equity Drawdown ...	47.19 (0.47%)
Gross Loss	-524.27	Balance Drawdown...	0.45% (45.33)	Equity Drawdown ...	0.47% (47.19)
Profit Factor	1.15	Expected Payoff	0.11	Margin Level	81068.86%
Recovery Factor	1.64	Sharpe Ratio	1.85	Z-Score	-1.40 (83.85%)
AHPR	1.0000 (0.00...	LR Correlation	0.90	OnTester result	77.49928252...
GHPR	1.0000 (0.00...	LR Standard Error	12.35		
Total Trades	726	Short Trades (won ...	364 (53.57%)	Long Trades (won ...	362 (50.55%)
Total Deals	1452	Profit Trades (% of...	378 (52.07%)	Loss Trades (% of t...	348 (47.93%)
	Largest	profit trade	9.14	loss trade	-5.00
	Average	profit trade	1.59	loss trade	-1.51
	Maximum	consecutive wins (\$)	8 (12.73)	consecutive losses ...	6 (-12.48)
	Maximal	consecutive profit ...	19.25 (6)	consecutive loss (c...	-12.48 (6)
	Average	consecutive wins	2	consecutive losses	2



在 EURUSD 图表上基于来自 EURUSD_Eqv1000 等量交易品种的信号进行交易时的测试程序报告

此处的柱线和分时报价数量不同，因为选择 `EURUSD` 作为测试金融工具，而不是自定义的 `EURUSD_Eqv1000`。

所有三个测试结果都略有不同。这是由于将行情“打包”到分钟柱线中以及原始金融工具和自定义金融工具价格变动的轻微不同步造成的。哪个结果更准确？这很可能取决于具体的交易系统及其实现特性。对于我们的带有柱线开盘控制的 `EA` 交易 *BandOsMa*，直接在 `EURUSD_Eqv1000` 上进行交易的版本应该具有最真实的结果。理论上，经验法则指出，在几个备选检查中，最可靠的是利润最低的那个，这几乎总是得到满足。

因此，我们分析了几种用于调整 `EA` 交易以在自定义交易品种上进行交易的技术，这些自定义交易品种在经纪商的工作交易品种中有原型。然而，这种情况并非强制性的。在许多情况下，自定义交易品种是基于来自外部系统（例如加密货币交易所）的数据生成的。对自定义交易品种进行交易必须将其公共 API 用于 `MT5 网络函数`。

使用自定义交易品种模拟特殊类型的图表

许多交易者使用特殊类型的图表，其中不考虑连续的实时时间。这不仅包括等量和等距柱线，还包括砖形图（Renko）、点数图（Point-And-Figure, PAF）、卡吉图（Kagi）等。自定义交易品种允许在 MetaTrader 5 中使用 M1 时间范围图表模拟这些类型的图表，但如果不是进行技术分析，而是在测试交易系统，则应谨慎对待。

对于特殊类型的图表，实际的柱线开盘时间（精确到毫秒）几乎总是不与 M1 柱线标记的分钟完全一致。因此，自定义柱线的开盘价不同于标准交易品种 M1 柱线的开盘价。

此外，其他 OHLC 价格也会有所不同，因为特殊图表上 M1 柱线形成的实际持续时间不等于一分钟。例如，等量图的 1000 个分时报价可能会累积超过 5 分钟。

自定义柱线的收盘价也不对应于实际的收盘时间，因为自定义柱线在技术上是一个 M1 柱线，即其名义持续时间为 1 分钟。

在使用诸如经典砖形图或点数图之类的图表类型时应特别小心。事实是，它们的反转柱线的开盘价与前一个柱线的收盘价之间存在缺口。因此，开盘价成为未来价格变动的预测指标。

对此类图表的分析应该根据已形成的柱线进行，也就是说，其特征价格是收盘价，然而，当按柱线工作时，测试程序仅为当前（最后）柱线提供开盘价（没有按收盘价的模式）。即使我们从已关闭的柱线（通常是第一个柱线）获取指标信号，交易仍会在第 0 个柱线的当前价格进行。即使我们转向分时报价模式，测试程序也总是根据一般规则生成分时报价，以基于每根柱线配置的参考点为指导。测试程序不考虑我们试图用 M1 柱线进行可视化模拟的特殊图表的结构和行为。

在测试程序中使用此类交易品种以任何模式（按开盘价、M1 OHLC 或按分时报价）进行交易都会影响结果的准确性：它们过于乐观，可能导致过高的期望。在这方面，至关重要的是不要在单独的砖形图或点数图上检查交易系统，而应结合在真实交易品种上执行订单进行检查。

自定义交易品种也可用于秒级时间范围或分时报价图表。在这种情况下，也会为柱线和分时报价生成与真实时间脱钩的虚拟时间。因此，此类图表非常适合操作分析，但在开发和测试交易策略（尤其是多交易品种策略）时需要额外注意。

对于任何自定义交易品种，另一种选择是在 `EA` 交易或指标内部独立计算柱线和分时报价数组。然而，调试和可视化此类结构需要额外的工作量。

7.3 经济日历

在制定交易策略时，最好考虑影响市场的基本面因素。MetaTrader 5 内置了经济日历，它在程序界面中作为工具栏中的一个单独选项卡提供，并且可以选择性地直接在图表上显示标签。可以通过终端设置对话框中“社区”选项卡上的一个单独标志来启用日历（登录社区不是必需的）。

由于 MetaTrader 5 支持算法交易，因此也可以从 MQL5 API 以编程方式访问经济日历事件。在本章中，我们将介绍能够读取、筛选和监控经济事件变化的函数和数据结构。

经济日历包含许多国家宏观经济指标的描述、发布时间表和历史数据值。对于每个事件，计划发布的准确时间、重要程度、对特定货币的影响、预测值以及其他特性都是已知的。宏观经济指标的实际值在发布时立即到达 MetaTrader 5。

日历的可用性允许你自动分析传入事件并在 EA 交易中以各种方式对其做出反应，例如，作为突破策略的一部分进行交易或在通道内进行波动性交易。另一方面，预判市场即将发生的波动可以帮助你确定低风险时段，并暂时关闭那些由于可能造成损失而对剧烈价格波动敏感的机器人。

所有处理经济日历的函数和结构体使用的 `datetime` 类型的值都等于交易服务器时间 (*TimeTradeServer*)，包括其时区和夏令时 (DST) 设置。换句话说，为了正确测试基于新闻交易的 EA 交易，其开发者必须独立更改那些 DST 模式与当前模式不同的时期（每年大约半年）的历史新闻时间。

日历函数不能在 [测试程序中使用](#)：当尝试调用任何这些函数时，我们会得到 `FUNCTION_NOT_ALLOWED (4014)` 错误。在这方面，测试基于日历的策略需要遵循特定流程：在线图表上运行 MQL 程序时，首先将日历条目保存到外部存储（例如，文件中），然后在测试程序中运行的 MQL 程序中加载和读取这些条目。

7.3.1 日历的基本概念

在处理经济日历时，我们将运用若干概念，MQL5 为这些概念的规范描述定义了特殊类型的结构体。

首先，日历事件与特定国家/地区相关联，使用 *MqlCalendarCountry* 结构体来描述每个国家/地区。

```
struct MqlCalendarCountry
{
    ulong id;           //country identifier according to ISO 3166-1
    string name;         // text name of the country (in the current terminal encod
    string code;         // two-letter country designation according to ISO 3166-1
    string currency;     // international country currency code
    string currency_symbol; // symbol/sign of the country's currency
    string url_name;     // country name used in the URL on the mql5.com website
};
```

我们将在下一节中了解如何以 *MqlCalendarCountry* 结构体数组的形式获取日历中可用的国家/地区列表及其特性。

目前，我们只需关注其 *id* 字段。该字段之所以重要，是因为它是确定日历事件是否属于特定国家/地区的关键。在每个国家/地区（或注册的国家/地区联盟，如欧盟），都存在一个国际公认的、会影响市场并因此被纳入日历的经济指标类型和信息事件类型列表。

每种事件类型均由 *MqlCalendarEvent* 结构体定义，其中 *country_id* 字段将事件唯一地关联到对应国家/地区。我们稍后会探讨所使用的枚举类型。

```
struct MqlCalendarEvent
{
    ulong          id;           // event ID
    ENUM_CALENDAR_EVENT_TYPE type; // event type
    ENUM_CALENDAR_EVENT_SECTOR sector; // sector to which the event belongs
    ENUM_CALENDAR_EVENT_FREQUENCY frequency; // frequency (periodicity) of the event
    ENUM_CALENDAR_EVENT_TIMEMODE time_mode; // event time mode
    ulong          country_id; // country identifier
    ENUM_CALENDAR_EVENT_UNIT unit; // indicator unit
    ENUM_CALENDAR_EVENT_IMPORTANCE importance; // importance of the event
    ENUM_CALENDAR_EVENT_MULTIPLIER multiplier; // indicator multiplier
    uint           digits;      // number of decimal places
    string         source_url; // URL of the event publication source
    string         event_code; // event code
    string         name;       // text name of the event in the termin
};
```

重要的是要理解，*MqlCalendarEvent* 结构体描述的正是事件的类型（例如，消费者价格指数 CPI 的发布），而不是可能每季度、每月或按其他时间表发生的某个具体事件。它包含了事件的通用特性，包括重要性、频率、与经济板块的关系、计量单位、名称以及信息来源。至于实际指标和预测指标，这些将在针对此类特定事件的日历条目中提供：这些条目以 *MqlCalendarValue* 结构体的形式存储，我们稍后会讨论。用于查询支持的事件类型的函数将在后续章节介绍。

type 字段中的事件类型被指定为 `ENUM_CALENDAR_EVENT_TYPE` 枚举值之一。

标识符	说明
CALENDAR_TYPE_EVENT	事件（会议、演讲等）
CALENDAR_TYPE_INDICATOR	经济指标
CALENDAR_TYPE_HOLIDAY	假日（周末）

事件所属的经济板块则从 `ENUM_CALENDAR_EVENT_SECTOR` 枚举中选择。

标识符	说明
CALENDAR_SECTOR_NONE	未设置部门
CALENDAR_SECTOR_MARKET	市场、交易所
CALENDAR_SECTOR_GDP	国内生产总值（GDP）
CALENDAR_SECTOR_JOBS	劳动力市场
CALENDAR_SECTOR_PRICES	价格
CALENDAR_SECTOR_MONEY	资金
CALENDAR_SECTOR_TRADE	交易

标识符	说明
CALENDAR_SECTOR_GOVERNMENT	政府
CALENDAR_SECTOR_BUSINESS	企业
CALENDAR_SECTOR_CONSUMER	消费
CALENDAR_SECTOR_HOUSING	住房
CALENDAR_SECTOR_TAXES	税收
CALENDAR_SECTOR_HOLIDAYS	假日

事件的频率通过 ENUM_CALENDAR_EVENT_FREQUENCY 枚举在 *frequency* 字段中显示。

标识符	说明
CALENDAR_FREQUENCY_NONE	未设置发布频率
CALENDAR_FREQUENCY_WEEK	每周
CALENDAR_FREQUENCY_MONTH	每月
CALENDAR_FREQUENCY_QUARTER	每季度
CALENDAR_FREQUENCY_YEAR	每年
CALENDAR_FREQUENCY_DAY	每日

事件持续时间 (*time_mode*) 可以用 ENUM_CALENDAR_EVENT_TMEMODE 枚举中的一个元素来描述。

标识符	说明
CALENDAR_TMEMODE_DATETIME	事件的具体时间是已知的
CALENDAR_TMEMODE_DATE	事件持续一整天
CALENDAR_TMEMODE_NOTIME	时间未发布
CALENDAR_TMEMODE_TENTATIVE	只有日期是事先知道的, 但事件的确切时间不知道 (时间是事后指定的)。

事件的重要性通过 ENUM_CALENDAR_EVENT_IMPORTANCE 枚举在 *importance* 字段中指定。

标识符	说明
CALENDAR_IMPORTANCE_NONE	未设置
CALENDAR_IMPORTANCE_LOW	低
CALENDAR_IMPORTANCE_MODERATE	中
CALENDAR_IMPORTANCE_HIGH	高

作为 `ENUM_CALENDAR_EVENT_UNIT` 枚举的一个成员，事件值所使用的度量单位在 *unit* 字段中定义。

标识符	说明
CALENDAR_UNIT_NONE	未设置单位
CALENDAR_UNIT_PERCENT	利息 (%)
CALENDAR_UNIT_CURRENCY	本国货币
CALENDAR_UNIT_HOUR	小时数
CALENDAR_UNIT_JOB	工作场所数量
CALENDAR_UNIT_RIG	钻机
CALENDAR_UNIT_USD	美元
CALENDAR_UNIT_PEOPLE	人数
CALENDAR_UNIT_MORTGAGE	按揭贷款数量
CALENDAR_UNIT_VOTE	投票数
CALENDAR_UNIT_BARREL	桶数
CALENDAR_UNIT_CUBICFEET	体积 (立方英尺)
CALENDAR_UNIT_POSITION	合约投机仓位净额
CALENDAR_UNIT_BUILDING	建筑物数量

在某些情况下，经济指标的数值需要根据 `ENUM_CALENDAR_EVENT_MULTIPLIER` 枚举中的某一元素使用 *multiplier*。

标识符	说明
CALENDAR_MULTIPLIER_NONE	未设置乘数
CALENDAR_MULTIPLIER_THOUSANDS	千倍
CALENDAR_MULTIPLIER_MILLIONS	百万倍
CALENDAR_MULTIPLIER_BILLIONS	十亿倍
CALENDAR_MULTIPLIER_TRILLIONS	万亿倍

至此，我们已了解所有用于描述 *MqlCalendarEvent* 结构体中事件类型的特殊数据类型。

每个独立的日历条目则由 *MqlCalendarValue* 结构体表示。其详细描述如下，但首先请注意一个关键点：*MqlCalendarValue* 包含一个 *event_id* 字段，该字段指向事件类型的标识符，即 *MqlCalendarEvent* 结构体中某个 *id* 的值。

如前所述，*MqlCalendarEvent* 结构体又通过 *country_id* 字段与 *MqlCalendarCountry* 关联。因此，一旦特定国家/地区或事件类型的信息录入日历数据库，便可为其注册任意数量的同类事件。当然，数据库的填充工作由信息提供方负责，而非开发者。

简而言之，系统分别维护着三个内部数据表：

- 🕒 *MqlCalendarCountry* 结构体表：描述国家/地区信息
- 🕒 *MqlCalendarEvent* 结构体表：描述事件类型
- 🕒 *MqlCalendarValue* 结构体表：存储各类具体事件的指标

通过引用事件类型标识符，可以有效避免在具体事件记录中重复信息。例如，月度发布的 CPI 值都只引用同一个包含该事件类型通用特性的 *MqlCalendarEvent* 结构体。如果不是采用这种分表方式，那么就必须在每个 CPI 日历条目中重复相同的特性。这种使用标识符字段在数据表之间建立关系的方法被称为 *relational*，我们将在关于 *SQLite* 的章节中再次讨论。所有这些都在下图中有说明。



结构体间通过标识符字段链接的示意图

所有这些表都存储在内部日历数据库中，该数据库在终端连接到服务器期间会持续保持最新状态。

日历条目（即具体事件）是 **MqlCalendarValue** 结构体。它们也通过其 **id** 字段中的唯一编号进行标识（这三个核心表各自都有自己的 **id** 字段）。

```

struct MqlCalendarValue
{
    ulong      id;                // entry ID
    ulong      event_id;          // event type ID
    datetime   time;              // time and date of the event
    datetime   period;            // reporting period of the event
    int        revision;          // revision of the published indicator in relation
    long       actual_value;       // actual value in ppm or LONG_MIN
    long       prev_value;        // previous value in ppm or LONG_MIN
    long       revised_prev_value; // revised previous value in ppm or LONG_MIN
    long       forecast_value;     // forecast value in ppm or LONG_MIN
    ENUM_CALENDAR_EVENT_IMPACT impact_type; // potential impact on the exchange rate

    // functions for checking values
    bool HasActualValue(void) const; // true if the actual_value field is filled
    bool HasPreviousValue(void) const; // true if the prev_value field is filled
    bool HasRevisedValue(void) const; // true if the revised_prev_value field is fi
    bool HasForecastValue(void) const; // true if the forecast_value field is filled

    // functions for getting values
    double GetActualValue(void) const; // actual_value or nan if value is not set
    double GetPreviousValue(void) const; // prev_value or nan if value is not set
    double GetRevisedValue(void) const; // revised_prev_value or nan if value is not
    double GetForecastValue(void) const; // forecast_value or nan if value is not set
};

```

对于每个事件，除了其发布时间 (*time*) 外，还存储了以下四个值：

- ⌚ 实际值 (*actual_value*)，在新闻发布后即刻知晓
- ⌚ 前次值 (*prev_value*)，在上次同类新闻发布时知晓
- ⌚ 前次指标的修正值 (*revised_prev_value*) (如果自上次发布后有所修改)
- ⌚ 预测值 (*forecast_value*)

显而易见，并非所有字段都必须填写。因此，对于未来的事件，当前值是缺失的 (尚未知晓)，而过去值的修正也并非一定发生。此外，这四个字段仅对量化指标有意义，而日历中还反映了监管机构的演讲、会议和假日。

空字段 (无值) 由常量 `LONG_MIN` (-9223372036854775808) 表示。如果字段中的值已指定 (不等于 `LONG_MIN`)，则它对应于指标真实值乘以一百万后的结果，也就是说，要获得常规 (真实) 形式的指标，需要将字段值除以 1,000,000。

为了方便程序员，该结构体定义了 4 个 *Has* 方法用于检查字段是否已填充，以及 4 个 *Get* 方法用于返回已转换为实数的相应字段值，如果字段未填充，则方法将返回 `NaN` (Not A Number, 非数字)。

有时，为了获取绝对值 (如果算法需要)，额外分析 *MqlCalendarEvent* 结构体中的 *multiplier* 特性非常重要，因为某些值是根据 `ENUM_CALENDAR_EVENT_MULTIPLIER` 枚举以倍数单位指定的。此外，*MqlCalendarEvent* 拥有 *digits* 字段，它指定了接收值的有效小数位数，用于后续的正确格式化 (例如，在调用 *NormalizeDouble* 时)。

报告期 (即已发布指标的计算周期) 在 *period* 字段中设置为该周期的第一天。例如，如果指标是按月计算的，那么日期 '2022.05.01 00:00:00' 表示五月份。报告期的时长 (例如，月、季度、年) 则

在关联的 *MqlCalendarEvent* 结构体的 *frequency* 字段中定义：该字段的类型是前面与其他枚举一起描述过的特殊枚举 `ENUM_CALENDAR_EVENT_FREQUENCY`。

特别值得关注的是 *impact_type* 字段，在新闻发布后，该字段会通过比较当前值和预测值，自动设置相应货币对汇率的影响方向。这种影响可能是积极的（货币预期升值），也可能是消极的（货币预期贬值）。例如，销售额降幅大于预期将被标记为具有消极影响，而失业率降幅大于预期则为积极影响。但并非所有事件的这种特性都能被明确解读（某些经济指标被认为是矛盾的），此外，还应注意变化的相对数值。

事件对本国货币汇率的潜在影响使用 `ENUM_CALENDAR_EVENT_IMPACT` 枚举表示。

标识符	说明
<code>CALENDAR_IMPACT_NA</code>	未说明影响
<code>CALENDAR_IMPACT_POSITIVE</code>	积极影响
<code>CALENDAR_IMPACT_NEGATIVE</code>	消极影响

经济日历的另一个重要概念是其“变更”这一事实。遗憾的是，并没有一个专门的结构体来描述变更。变更唯一的特性是其独特的 ID，这是一个整数，在内部日历库每次发生更改时由系统分配。

我们知道，信息提供者会不断修改日历：新的即将发生的事件会被添加进来，而已发布的指标和预测也可能被修正。因此，跟踪任何编辑的发生至关重要，通过检测周期性递增的变更编号可以实现这一点。

具有特定标识符的编辑时间及其具体内容在 `MQL5` 中是不可获取的。如果需要，`MQL` 程序应自行实现定期的日历状态查询和记录分析。

一系列 `MQL5` 函数允许获取关于国家/地区、事件类型、具体日历条目及其变更的信息。我们将在后续章节中对此进行探讨。

注意！首次访问日历时（如果之前未在终端工具栏中打开过“日历”选项卡），内部日历数据库与服务器之间的同步可能需要几秒钟的时间。

7.3.2 获取可用国家/地区的列表和描述

你可以使用 *CalendarCountries* 函数获取日历中播报事件的完整国家/地区列表。

`int CalendarCountries(MqlCalendarCountry &countries[])`

该函数会使用 *MqlCalendarCountry* 结构体来填充通过引用传递的 *countries* 数组。该数组可以是动态的，也可以是具有足够大小的固定数组。

成功时，该函数返回从服务器接收到的国家/地区描述数量，如果出错则返回 0。在 *_LastError* 中可能的错误代码中，我们尤其可能发现 5401 `ERR_CALENDAR_TIMEOUT`，表示请求超时）或 5400 `ERR_CALENDAR_MORE_DATA`，表示固定数组的大小不足以获取所有国家/地区的描述）。在后一种情况下，系统将仅复制适合的部分。

我们编写一个简单的脚本 *CalendarCountries.mq5*，它获取完整的国家/地区列表并将其输出到日志中。


```

void OnStart()
{
    MqlCalendarCountry countries[];
    PRTF(CalendarCountries(countries));
    ArrayPrint(countries);
}

```

以下是一个示例结果。

```

CalendarCountries(countries)=23 / ok

```

	[id]	[name]	[code]	[currency]	[currency_symbol]	[url_name]	[rese
[0]	554	"New Zealand"	"NZ"	"NZD"	"\$"	"new-zealand"	
[1]	999	"European Union"	"EU"	"EUR"	"€"	"european-union"	
[2]	392	"Japan"	"JP"	"JPY"	"¥"	"japan"	
[3]	124	"Canada"	"CA"	"CAD"	"\$"	"canada"	
[4]	36	"Australia"	"AU"	"AUD"	"\$"	"australia"	
[5]	156	"China"	"CN"	"CNY"	"¥"	"china"	
[6]	380	"Italy"	"IT"	"EUR"	"€"	"italy"	
[7]	702	"Singapore"	"SG"	"SGD"	"R\$"	"singapore"	
[8]	276	"Germany"	"DE"	"EUR"	"€"	"germany"	
[9]	250	"France"	"FR"	"EUR"	"€"	"france"	
[10]	76	"Brazil"	"BR"	"BRL"	"R\$"	"brazil"	
[11]	484	"Mexico"	"MX"	"MXN"	"Mex\$"	"mexico"	
[12]	710	"South Africa"	"ZA"	"ZAR"	"R"	"south-africa"	
[13]	344	"Hong Kong"	"HK"	"HKD"	"HK\$"	"hong-kong"	
[14]	356	"India"	"IN"	"INR"	"₹"	"india"	
[15]	578	"Norway"	"NO"	"NOK"	"Kr"	"norway"	
[16]	0	"Worldwide"	"WW"	"ALL"	"	"worldwide"	
[17]	840	"United States"	"US"	"USD"	"\$"	"united-states"	
[18]	826	"United Kingdom"	"GB"	"GBP"	"£"	"united-kingdom"	
[19]	756	"Switzerland"	"CH"	"CHF"	"F"	"switzerland"	
[20]	410	"South Korea"	"KR"	"KRW"	"₩"	"south-korea"	
[21]	724	"Spain"	"ES"	"EUR"	"€"	"spain"	
[22]	752	"Sweden"	"SE"	"SEK"	"Kr"	"sweden"	

值得注意的是，标识符 0（代码“WW”和伪货币“ALL”）对应于全球性事件（涉及多国，例如 G7、G20 会议等多），而货币“EUR”与日历中可用的几个欧盟国家相关联（可以看到，并没有显示整个欧元区）。此外，欧盟本身有一个通用标识符 999。

如果你对特定国家/地区感兴趣，可以根据 ISO 3166-1 标准通过其数字代码检查其可用性。特别地，在上面的日志中，这些代码显示在**第一列（id 字段）**。

要通过其 ID（在 *id* 参数中指定）获取单个国家/地区的描述，你可以使用 *CalendarCountryById* 函数。

bool CalendarCountryById(const long id, MqlCalendarCountry &country)

如果成功，该函数将返回 *true* 并填充 *country* 结构体的字段。

如果未找到该国家/地区，我们将得到 *false*，并且在 *_LastError* 中将得到错误代码 5402 (ERR_CALENDAR_NO_DATA)。

有关使用此函数的示例，请参见 [按国家/地区或货币获取事件记录](#)。

7.3.3 按国家/地区和货币查询事件类型

财经事件和节假日的日历在每个国家/地区都有其自身的特殊性。MQL 程序可以查询特定国家/地区内的事件类型，以及与特定货币相关的事件类型。后者在多个国家/地区使用相同货币的情况下非常重要，例如，大多数欧盟成员国。

int CalendarEventByCountry(const string country, MqlCalendarEvent &events[])

CalendarEventByCountry 函数使用日历中针对指定国家/地区 (根据 ISO 3166-1 alpha-2 标准的双字母国家/地区代码) 的所有可用事件类型的描述来填充通过引用传递的 **MqlCalendarEvent** 结构体数组。我们在上一节的日志中看到了此类代码的示例: EU 代表欧盟, US 代表美国, DE 代表德国, CN 代表中国, 依此类推。

接收数组可以是动态的, 也可以是具有足够大小的固定数组。

该函数返回接收到的描述数量, 如果出错则返回 0。特别地, 如果固定数组无法容纳所有事件, 该函数将用可用数据的适合部分填充固定数组, 并将 **_LastError** 代码设置为等于 CALENDAR_MORE_DATA (5400)。也可能发生内存分配错误 (4004, ERR_NOT_ENOUGH_MEMORY) 或来自服务器的日历请求超时 (5401, ERR_CALENDAR_TIMEOUT)。

如果具有给定代码的国家/地区不存在, 则会发生 INTERNAL_ERROR (4001)。

通过指定 NULL 或空字符串 "" 来代替 **country**, 你可以获取所有国家/地区的完整事件列表。

我们使用简单的脚本 **CalendarEventKindsByCountry.mq5** 来测试该函数的性能。它只有一个输入参数, 即我们感兴趣的国家/地区的代码。

```
input string CountryCode = "HK";
```

接下来, 通过调用 **CalendarEventByCountry** 发出事件类型请求, 如果成功, 则将产生的数组记录到日志中。

```
void OnStart()
{
    MqlCalendarEvent events[];
    if(PRTF(CalendarEventByCountry(CountryCode, events)))
    {
        Print("Event kinds for country: ", CountryCode);
        ArrayPrint(events);
    }
}
```

以下是结果示例 (由于代码行很长, 为了在书中发布, 它们被人为地分成了 2 个块: 第一个块包含 **MqlCalendarEvent** 结构体的数字字段, 第二个块包含字符串字段)。

```

CalendarEventByCountry(CountryCode,events)=26 / ok
Event kinds for country: HK
[id] [type] [sector] [frequency] [time_mode] [country_id] [unit] [importance] [multip
[ 0] 344010001 1 5 2 0 344 6 1 3 1 »
[ 1] 344010002 1 5 2 0 344 1 1 0 1 »
[ 2] 344020001 1 4 2 0 344 1 1 0 1 »
[ 3] 344020002 1 2 3 0 344 1 3 0 1 »
[ 4] 344020003 1 2 3 0 344 1 2 0 1 »
[ 5] 344020004 1 6 2 0 344 1 1 0 1 »
[ 6] 344020005 1 6 2 0 344 1 1 0 1 »
[ 7] 344020006 1 6 2 0 344 2 2 3 3 »
[ 8] 344020007 1 9 2 0 344 1 1 0 1 »
[ 9] 344020008 1 3 2 0 344 1 2 0 1 »
[10] 344030001 2 12 0 1 344 0 0 0 0 »
[11] 344030002 2 12 0 1 344 0 0 0 0 »
[12] 344030003 2 12 0 1 344 0 0 0 0 »
[13] 344030004 2 12 0 1 344 0 0 0 0 »
[14] 344030005 2 12 0 1 344 0 0 0 0 »
[15] 344030006 2 12 0 1 344 0 0 0 0 »
[16] 344030007 2 12 0 1 344 0 0 0 0 »
[17] 344030008 2 12 0 1 344 0 0 0 0 »
[18] 344030009 2 12 0 1 344 0 0 0 0 »
[19] 344030010 2 12 0 1 344 0 0 0 0 »
[20] 344030011 2 12 0 1 344 0 0 0 0 »
[21] 344030012 2 12 0 1 344 0 0 0 0 »
[22] 344030013 2 12 0 1 344 0 0 0 0 »
[23] 344030014 2 12 0 1 344 0 0 0 0 »
[24] 344030015 2 12 0 1 344 0 0 0 0 »
[25] 344500001 1 8 2 0 344 0 1 0 1 »

```

日志的后续内容 (右侧片段)。

```

» [source_url] [event_code] [name]
[ 0]» "https://www.hkma.gov.hk/eng/" "foreign-exchange-reserves" "Foreign Exchange Re
[ 1]» "https://www.hkma.gov.hk/eng/" "hkma-m3-money-supply-yy" "HKMA M3 Money Supply
[ 2]» "https://www.censtatd.gov.hk/en/" "cpi-yy" "CPI y/y"
[ 3]» "https://www.censtatd.gov.hk/en/" "gdp-qq" "GDP q/q"
[ 4]» "https://www.censtatd.gov.hk/en/" "gdp-yy" "GDP y/y"
[ 5]» "https://www.censtatd.gov.hk/en/" "exports-mm" "Exports y/y"
[ 6]» "https://www.censtatd.gov.hk/en/" "imports-mm" "Imports y/y"
[ 7]» "https://www.censtatd.gov.hk/en/" "trade-balance" "Trade Balance"
[ 8]» "https://www.censtatd.gov.hk/en/" "retail-sales-yy" "Retail Sales y/y"
[ 9]» "https://www.censtatd.gov.hk/en/" "unemployment-rate-3-months" "Unemployment Ra
[10]» "https://publicholidays.hk/" "new-years-day" "New Year's Day"
[11]» "https://publicholidays.hk/" "lunar-new-year" "Lunar New Year"
[12]» "https://publicholidays.hk/" "ching-ming-festival" "Ching Ming Festival"
[13]» "https://publicholidays.hk/" "good-friday" "Good Friday"
[14]» "https://publicholidays.hk/" "easter-monday" "Easter Monday"
[15]» "https://publicholidays.hk/" "birthday-of-buddha" "The Birthday of the Buddha"
[16]» "https://publicholidays.hk/" "labor-day" "Labor Day"
[17]» "https://publicholidays.hk/" "tuen-ng-festival" "Tuen Ng Festival"
[18]» "https://publicholidays.hk/" "hksar-establishment-day" "HKSAR Establishment Day
[19]» "https://publicholidays.hk/" "day-following-mid-autumn-festival" "The Day Follo
[20]» "https://publicholidays.hk/" "national-day" "National Day"
[21]» "https://publicholidays.hk/" "chung-yeung-festival" "Chung Yeung Festival"
[22]» "https://publicholidays.hk/" "christmas-day" "Christmas Day"
[23]» "https://publicholidays.hk/" "first-weekday-after-christmas-day" "The First Wee
[24]» "https://publicholidays.hk/" "day-following-good-friday" "The Day Following Goo
[25]» "https://www.markiteconomics.com" "nikkei-pmi" "S&P Global PMI"

```

int CalendarEventByCurrency(const string currency, MqlCalendarEvent &events[])

CalendarEventByCurrency 函数在传入的 *events* 数组中填充日历中与指定 *currency* 相关的各种事件的描述。三位字母代码是所有外汇交易商中通用的。

如果指定的货币代码无效，函数将返回 0 (无错误) 和一个空数组。

如果指定 NULL 或空字符串 "" 来代替 *currency*，则可以获得完整的日历事件列表。

我们使用 *CalendarEventKindsByCurrency.mq5* 脚本测试该函数。输入参数指定了货币代码。

```
input string Currency = "CNY";
```

在 *OnStart* 处理程序中，我们请求事件并将其输出到日志中。

```

void OnStart()
{
    MqlCalendarEvent events[];
    if(PRTF(CalendarEventByCurrency(Currency, events)))
    {
        Print("Event kinds for currency: ", Currency);
        ArrayPrint(events);
    }
}

```

以下是结果示例 (用缩写给出)。

```

CalendarEventByCurrency(Currency,events)=40 / ok
Event kinds for currency: CNY
[id] [type] [sector] [frequency] [time_mode] [country_id] [unit] [importance] [multip
[ 0] 156010001 1 4 2 0 156 1 2 0 1 »
[ 1] 156010002 1 4 2 0 156 1 1 0 1 »
[ 2] 156010003 1 4 2 0 156 1 1 0 1 »
[ 3] 156010004 1 2 3 0 156 1 3 0 1 »
[ 4] 156010005 1 2 3 0 156 1 2 0 1 »
[ 5] 156010006 1 9 2 0 156 1 2 0 1 »
[ 6] 156010007 1 8 2 0 156 1 2 0 1 »
[ 7] 156010008 1 8 2 0 156 0 3 0 1 »
[ 8] 156010009 1 8 2 0 156 0 3 0 1 »
[ 9] 156010010 1 8 2 0 156 1 2 0 1 »
[10] 156010011 0 5 0 0 156 0 2 0 0 »
[11] 156010012 1 3 2 0 156 1 2 0 1 »
[12] 156010013 1 8 2 0 156 1 1 0 1 »
[13] 156010014 1 8 2 0 156 1 1 0 1 »
[14] 156010015 1 8 2 0 156 0 3 0 1 »
[15] 156010016 1 8 2 0 156 1 2 0 1 »
[16] 156010017 1 9 2 0 156 1 2 0 1 »
[17] 156010018 1 2 3 0 156 1 2 0 1 »
[18] 156020001 1 6 2 3 156 6 2 3 2 »
[19] 156020002 1 6 2 3 156 1 1 0 1 »
[20] 156020003 1 6 2 3 156 1 1 0 1 »
[21] 156020004 1 6 2 3 156 2 2 3 2 »
[22] 156020005 1 6 2 3 156 1 1 0 1 »
[23] 156020006 1 6 2 3 156 1 1 0 1 »
...

```

右侧片段。

```

» [source_url] [event_code] [name]
[ 0]» "http://www.stats.gov.cn/english/" "cpi-mm" "CPI m/m"
[ 1]» "http://www.stats.gov.cn/english/" "cpi-yy" "CPI y/y"
[ 2]» "http://www.stats.gov.cn/english/" "ppi-yy" "PPI y/y"
[ 3]» "http://www.stats.gov.cn/english/" "gdp-qq" "GDP q/q"
[ 4]» "http://www.stats.gov.cn/english/" "gdp-yy" "GDP y/y"
[ 5]» "http://www.stats.gov.cn/english/" "retail-sales-yy" "Retail Sales y/y"
[ 6]» "http://www.stats.gov.cn/english/" "industrial-production-yy" "Industrial Produ
[ 7]» "http://www.stats.gov.cn/english/" "manufacturing-pmi" "Manufacturing PMI"
[ 8]» "http://www.stats.gov.cn/english/" "non-manufacturing-pmi" "Non-Manufacturing P
[ 9]» "http://www.stats.gov.cn/english/" "fixed-asset-investment-yy" "Fixed Asset Inv
[10]» "http://www.stats.gov.cn/english/" "nbs-press-conference-on-economic-situation"
[11]» "http://www.stats.gov.cn/english/" "unemployment-rate" "Unemployment Rate"
[12]» "http://www.stats.gov.cn/english/" "industrial-profit-yy" "Industrial Profit y/
[13]» "http://www.stats.gov.cn/english/" "industrial-profit-ytd-yy" "Industrial Profi
[14]» "http://www.stats.gov.cn/english/" "composite-pmi" "Composite PMI"
[15]» "http://www.stats.gov.cn/english/" "industrial-production-ytd-yy" "Industrial P
[16]» "http://www.stats.gov.cn/english/" "retail-sales-ytd-yy" "Retail Sales YTD y/y"
[17]» "http://www.stats.gov.cn/english/" "gdp-ytd-yy" "GDP YTD y/y"
[18]» "http://english.customs.gov.cn/" "trade-balance-usd" "Trade Balance USD"
[19]» "http://english.customs.gov.cn/" "imports-usd-yy" "Imports USD y/y"
[20]» "http://english.customs.gov.cn/" "exports-usd-yy" "Exports USD y/y"
[21]» "http://english.customs.gov.cn/" "trade-balance" "Trade Balance"
[22]» "http://english.customs.gov.cn/" "imports-yy" "Imports y/y"
[23]» "http://english.customs.gov.cn/" "exports-yy" "Exports y/y"
...

```

细心的读者会注意到，事件类型标识符包含了国家/地区代码、新闻来源编号以及来源内部的序列号（编号从 1 开始）。因此，事件类型标识符的通用格式是：CCCSSNNNN，其中 CCC 是国家/地区代码，SS 是来源，NNNN 是编号。例如，156020001 是中国第二个来源的第一条新闻，而 344030010 是香港第三个来源的第十条新闻。唯一的例外是全球性新闻，其“国家/地区”代码不是 000 而是 1000。

7.3.4 按 ID 获取事件说明

真实的 MQL 程序通常会按时间范围、国家、货币或其他标准筛选来请求当前或即将发生的日历事件。用于此目的的 API 函数（我们稍后会讨论）返回的是 *MqlCalendarValue* 结构体，这种结构体只存储事件标识符，不存储其描述。因此，如果你需要提取完整信息，*CalendarEventById* 函数会很有用。

bool CalendarEventById(ulong id, MqlCalendarEvent &event)

CalendarEventById 函数按事件 ID 获取该事件的描述。该函数返回成功或错误指示。

下一节将给出如何使用此函数的示例。

7.3.5 按国家/地区或货币获取事件记录

在日历中，针对给定的日期范围查询各种类型的特定事件，并按国家/地区或货币进行筛选。

```
int CalendarValueHistory(MqlCalendarValue &values[], datetime from, datetime to = 0,
    const string country = NULL, const string currency = NULL)
```

CalendarValueHistory 函数用 *from* 和 *to* 之间时间范围内的日历条目填充通过引用传递的 *values* 数组。这两个参数都可以包含日期和时间。值 *from* 包含在区间内，但值 *to* 未包含在区间内。换句话说，该函数选择 *time* 特性满足以下复合条件的日历条目 (**MqlCalendarValue** 结构体): *from* $\leq$ *time* < *to*。

必须指定起始时间 *from*，而结束时间 *to* 是可选的：如果省略或等于 0，则所有未来的事件都将复制到数组中。

时间 *to* 应该大于 *from*，除非前者等于 0。查询所有可用事件（过去和未来）的特殊组合是 *from* 和 *to* 都为 0。

如果接收数组是动态的，将自动为其分配内存。如果数组是固定大小的，则复制的条目数量不会超过数组的大小。

country 和 *currency* 参数允许你按国家/地区或货币设置额外的记录筛选。*country* 参数接受双字母 ISO 3166-1 alpha-2 国家/地区代码（例如，“DE”、“FR”、“EU”），而 *currency* 参数接受三字母货币名称（例如，“EUR”、“CNY”）。

任何参数中的默认值 NULL 或空字符串 "" 等效于没有相应的筛选器。

如果同时指定了两个筛选器，则仅选择那些同时满足“国家/地区”和“货币”这两个条件的事件的值。如果日历中包含具有多种货币的国家/地区，并且每种货币也在多个国家/地区流通，这可能会派上用场。目前日历中没有此类事件。要获取欧元区国家的事件，只需指定特定国家的代码或“EU”，并假定货币为“EUR”。

该函数返回复制的元素数量，并可能设置错误代码。特别是，如果来自服务器的请求超时，在 *_LastError* 中，我们会得到错误 5401 (ERR_CALENDAR_TIMEOUT)。如果固定数组容纳不下所有记录，代码将等于 5400 (ERR_CALENDAR_MORE_DATA)，但数组将被填充。为动态数组分配内存时，可能会出现错误 4004 (ERR_NOT_ENOUGH_MEMORY)。

注意！数组中元素的顺序可能与时间顺序不同。你必须按时间对记录进行排序。

使用 **CalendarValueHistory** 函数，我们可以查询即将发生的事件：

```
MqlCalendarValue values[];
if(CalendarValueHistory(values, TimeCurrent()))
{
    ArrayPrint(values);
}
```

然而，使用此代码，我们将得到一个信息不足的表格，其中事件名称、重要性和货币代码将隐藏在 **MqlCalendarValue::event_id** 字段的事件 ID 之后，并间接隐藏在 **MqlCalendarEvent::country_id** 字段的国家/地区标识符之后。为了使信息输出更友好，你应该通过事件代码请求事件的描述，从该描述中获取国家/地区代码，然后获取其特性。我们在示例脚本 **CalendarForDates.mq5** 中展示它。

在输入参数中，我们将提供输入国家代码和货币进行筛选的功能。默认情况下，请求欧盟的事件。

```
input string CountryCode = "EU";
input string Currency = "";
```

事件的日期范围将自动计算为向前和向后一段时间。这个“一段时间”也将留给用户从三个选项中选择：一天、一周或一个月。

```
#define DAY_LONG    60 * 60 * 24
#define WEEK_LONG   DAY_LONG * 7
#define MONTH_LONG  DAY_LONG * 30
#define YEAR_LONG   MONTH_LONG * 12
```

```
enum ENUM_CALENDAR_SCOPE
{
    SCOPE_DAY = DAY_LONG,
    SCOPE_WEEK = WEEK_LONG,
    SCOPE_MONTH = MONTH_LONG,
    SCOPE_YEAR = YEAR_LONG,
};
```

```
input ENUM_CALENDAR_SCOPE Scope = SCOPE_DAY;
```

我们定义自己的结构体 *MqlCalendarRecord*，它是 *MqlCalendarValue* 的派生类，并向其添加字段以便于显示那些将通过依赖结构体中的链接（标识符）填充的特性。

```
struct MqlCalendarRecord: public MqlCalendarValue
{
    static const string importances[];

    string importance;
    string name;
    string currency;
    string code;
    double actual, previous, revised, forecast;
    ...
};
```

```
static const string MqlCalendarRecord::importances[] = {"None", "Low", "Medium", "Hig
```

添加的字段中有包含重要性（静态数组 *importances* 的值之一）、事件名称、国家/地区和货币的行，以及四个 *double* 格式的值。这实际上意味着为了打印时的视觉呈现而重复信息。稍后我们将为日历准备一个更高级的“包装器”。

要填充该对象，我们将需要一个参数化构造函数，它接受原始的 *MqlCalendarValue* 结构体。在所有继承的字段通过运算符 '=' 隐式复制到新对象后，我们调用专门准备的 *extend* 方法。


```

MqlCalendarRecord() { }

MqlCalendarRecord(const MqlCalendarValue &value)
{
    this = value;
    extend();
}

```

在 **extend** 方法中，我们通过其标识符获取事件的描述。然后，基于事件描述中的国家/地区标识符，我们得到一个包含国家/地区特性的结构体。之后，我们可以从接收到的 **MqlCalendarEvent** 和 **MqlCalendarCountry** 结构体中填充添加字段的前半部分。

```

void extend()
{
    MqlCalendarEvent event;
    CalendarEventById(event_id, event);

    MqlCalendarCountry country;
    CalendarCountryById(event.country_id, country);

    importance = importances[event.importance];
    name = event.name;
    currency = country.currency;
    code = country.code;

    MqlCalendarValue value = this;

    actual = value.GetActualValue();
    previous = value.GetPreviousValue();
    revised = value.GetRevisedValue();
    forecast = value.GetForecastValue();
}

```

接下来，我们调用内置的 **Get** 方法来填充四个 **double** 类型的字段（包含财经指标）。

现在我们可以主 **OnStart** 处理程序中使用这个新结构体了。

```

void OnStart()
{
    MqlCalendarValue values[];
    MqlCalendarRecord records[];
    datetime from = TimeCurrent() - Scope;
    datetime to = TimeCurrent() + Scope;
    if(PRTF(CalendarValueHistory(values, from, to, CountryCode, Currency)))
    {
        for(int i = 0; i < ArraySize(values); ++i)
        {
            PUSH(records, MqlCalendarRecord(values[i]));
        }
        Print("Near past and future calendar records (extended): ");
        ArrayPrint(records);
    }
}

```

这里，通过为输入参数中设置的当前条件调用 *CalendarValueHistory* 来填充标准 *MqlCalendarValue* 结构体数组。接下来，所有元素都转移到 *MqlCalendarRecord* 数组中。此外，在创建对象时，它们会用额外的信息进行扩展。最后，事件数组输出到日志中。

日志条目会很长。首先，我们显示左半部分，这正是如果我们打印标准 *MqlCalendarValue* 结构体数组时会看到的内容。

```

CalendarValueHistory(values,from,to,CountryCode,Currency)=6 / ok
Near past and future calendar records (extended):
[id] [event_id] [time] [period] [revision] [actual_value] [prev_value] [revised_prev_
[0] 162723 999020003 2022.06.23 03:00:00 1970.01.01 00:00:00 0 -9223372036854775808 -
[1] 162724 999020003 2022.06.24 03:00:00 1970.01.01 00:00:00 0 -9223372036854775808 -
[2] 168518 999010034 2022.06.24 11:00:00 1970.01.01 00:00:00 0 -9223372036854775808 -
[3] 168515 999010031 2022.06.24 13:10:00 1970.01.01 00:00:00 0 -9223372036854775808 -
[4] 168509 999010014 2022.06.24 14:30:00 1970.01.01 00:00:00 0 -9223372036854775808 -
[5] 161014 999520001 2022.06.24 22:30:00 2022.06.21 00:00:00 0 -9223372036854775808 -

```

下半部分是对名称、重要性和含义的“解码”。

```

CalendarValueHistory(values,from,to,CountryCode,Currency)=6 / ok
Near past and future calendar records (extended):
[importance] [name] [currency] [code] [actual] [previous] [revised] [forecast]
[0] "High" "EU Leaders Summit" "EUR" "EU" nan nan nan nan
[1] "High" "EU Leaders Summit" "EUR" "EU" nan nan nan nan
[2] "Medium" "ECB Supervisory Board Member McCaul Speech" "EUR" "EU" nan nan nan nan
[3] "Medium" "ECB Supervisory Board Member Fernandez-Bollo Speech" "EUR" "EU" nan nan
[4] "Medium" "ECB Vice President de Guindos Speech" "EUR" "EU" nan nan nan nan
[5] "Low" "CFTC EUR Non-Commercial Net Positions" "EUR" "EU" nan -6.000000 nan nan

```

7.3.6 获取特定类型的事件记录

如有必要，MQL 程序能够请求特定类型的事件：为此，只需提前知道事件标识符即可，例如，使用在 [按国家/地区和货币查询事件类型](#) 一节中介绍的 *CalendarEventByCountry* 或 *CalendarEventByCurrency* 函数。

```
int CalendarValueHistoryByEvent(ulong id, MqlCalendarValue &values[], datetime from, datetime to = 0)
```

CalendarValueHistoryByEvent 函数用由 *id* 标识符指示的特定类型事件的记录来填充通过引用传递的数组。参数 *from* 和 *to* 允许你限制搜索事件的日期范围。

如果未指定可选参数 *to*，则从 *from* 时间开始以及未来的所有日历条目都将放入数组中。要查询所有过去的事件，请将 *from* 设置为 0。如果 *from* 和 *to* 参数都为 0，则将返回所有历史事件和计划事件。在所有其他情况下，当 *to* 不等于 0 时，它必须大于 *from*。

values 数组可以是动态的（然后函数将根据数据量自动扩展或缩小它）或固定大小的（然后只有适合的部分将被复制到数组中）。

该函数返回复制的元素数量。

作为示例，请考虑 *CalendarStatsByEvent.mq5* 脚本，它计算给定国家/地区或货币在给定时间范围内不同类型事件的统计数据（出现频率）。

分析条件在输入变量中指定。

```
input string CountryOrCurrency = "EU";
input ENUM_CALENDAR_SCOPE Scope = SCOPE_YEAR;
```

根据 *CountryOrCurrency* 字符串的长度，该字符串可能会被解释为国家/地区代码（2 个字符）或货币代码（3 个字符）。

为了收集统计数据，我们将声明一个结构体；其字段将存储事件类型的标识符和名称、其重要性以及此类事件的计数器。

```
struct CalendarEventStats
{
    static const string importances[];
    ulong id;
    string name;
    string importance;
    int count;
};
```

```
static const string CalendarEventStats::importances[] = {"None", "Low", "Medium", "Hi
```

在 *OnStart* 函数中，我们首先使用 *CalendarEventByCountry* 或 *CalendarEventByCurrency* 函数请求指定历史深度和未来的所有类型的事件，然后，在遍历 *events* 数组中接收到的事件描述的循环中，我们为每个事件 ID 调用 *CalendarValueHistoryByEvent*。在这个应用中，我们对 *values* 数组的内容不感兴趣，因为我们只需要知道它们的数量。

```

void OnStart()
{
    MqlCalendarEvent events[];
    MqlCalendarValue values[];
    CalendarEventStats stats[];

    const datetime from = TimeCurrent() - Scope;
    const datetime to = TimeCurrent() + Scope;

    if(StringLen(CountryOrCurrency) == 2)
    {
        PRTF(CalendarEventByCountry(CountryOrCurrency, events));
    }
    else
    {
        PRTF(CalendarEventByCurrency(CountryOrCurrency, events));
    }

    for(int i = 0; i < ArraySize(events); ++i)
    {
        if(CalendarValueHistoryByEvent(events[i].id, values, from, to))
        {
            CalendarEventStats event = {events[i].id, events[i].name,
                CalendarEventStats::importances[events[i].importance], ArraySize(values)}
            PUSH(stats, event);
        }
    }

    SORT_STRUCT(CalendarEventStats, stats, count);
    ArrayReverse(stats);
    ArrayPrint(stats);
}

```

成功调用函数后，我们填充 *CalendarEventStats* 结构体并将其添加到结构体数组 *stats* 中。接下来，我们按照既定方式对结构体进行排序（SORT_STRUCT 宏在 [数组中的比较、排序和搜索](#) 一节中描述）。

使用默认设置运行脚本会在日志中生成类似这样的内容（已缩写）。

```
CalendarEventByCountry(CountryOrCurrency,events)=82 / ok
      [id]                                     [name] [importance] [cc]
[ 0] 999520001 "CFTC EUR Non-Commercial Net Positions"      "Low"
[ 1] 999010029 "ECB President Lagarde Speech"                 "High"
[ 2] 999010035 "ECB Executive Board Member Elderson Speech"  "Medium"
[ 3] 999030027 "Core CPI"                                     "Low"
[ 4] 999030026 "CPI"                                          "Low"
[ 5] 999030025 "CPI excl. Energy and Unprocessed Food y/y"  "Low"
[ 6] 999030024 "CPI excl. Energy and Unprocessed Food m/m"  "Low"
[ 7] 999030010 "Core CPI m/m"                                 "Medium"
[ 8] 999030013 "CPI y/y"                                      "Low"
[ 9] 999030012 "Core CPI y/y"                                 "Low"
[10] 999040006 "Consumer Confidence Index"                   "Low"
[11] 999030011 "CPI m/m"                                     "Medium"
...
[65] 999010008 "ECB Economic Bulletin"                       "Medium"
[66] 999030023 "Wage Costs y/y"                              "Medium"
[67] 999030009 "Labour Cost Index"                            "Low"
[68] 999010025 "ECB Bank Lending Survey"                     "Low"
[69] 999010030 "ECB Supervisory Board Member af Jochnick Speech" "Medium"
[70] 999010022 "ECB Supervisory Board Member Hakkarainen Speech" "Medium"
[71] 999010028 "ECB Financial Stability Review"              "Medium"
[72] 999010009 "ECB Targeted LTRO"                           "Medium"
[73] 999010036 "ECB Supervisory Board Member Tuominen Speech" "Medium"
```

请注意，总共接收到 82 种类型的事件，然而，在统计数组中，我们只有 74 种。这是因为，如果在指定的日期范围内没有任何类型的事件，*CalendarValueHistoryByEvent* 函数会返回 *false* (失败)并在 *_LastError* 中返回零错误代码。在上面的测试中，有 8 个这样的条目理论上存在，但在一年内从未遇到过。

7.3.7 按 ID 读取事件记录

了解近期事件的时间表后，交易者可以相应地调整他们的机器人。日历 API 中没有用于自动跟踪新闻发布的函数或事件 (这里的“事件”是指像 *OnCalendar* 那样处理新财经信息的功能，类似于 *OnTick*)。算法必须以任何选定的频率自行执行此操作。特别是，你可以使用先前讨论的函数之一 (例如，*CalendarValueHistoryByEvent*、*CalendarValueHistory*)找出所需事件的标识符，然后调用 *CalendarValueById* 来获取 *MqlCalendarValue* 结构体中字段的当前状态。

```
bool CalendarValueById(ulong id, MqlCalendarValue &value)
```

该函数用关于特定事件的当前信息填充通过引用传递的结构体。

函数的返回值表示成功 (*true*) 或错误 (*false*) 的标志。

我们创建一个简单的无缓冲区指标 *CalendarRecordById.mq5*，它将在未来找到类型为“财经指标”(即数值指标)的最近事件，并按计时器轮询其状态。当新闻发布时，数据将发生变化 (指标的“实际”值将变得已知)，并且指标将显示警报。

轮询日历的频率在输入变量中设置。

```
input uint TimerSeconds = 5;
```

我们在 *OnInit* 中运行计时器。

```
void OnInit()
{
    EventSetTimer(TimerSeconds);
}
```

为便于将事件描述输出到日志，我们使用 *MqlCalendarRecord* 结构体（我们已经从 *CalendarForDates.mq5* 脚本示例中了解该结构体）。

为了存储新闻信息的初始状态，我们描述了 *track* 结构体。

```
MqlCalendarValue track;
```

当该结构体为空（并且 *id* 字段中包含“0”）时，程序必须查询即将发生的事件，并在其中找到类型为 *CALENDAR_TYPE_INDICATOR* 且当前值尚未知的最近事件。

```
void OnTimer()
{
    if(!track.id)
    {
        MqlCalendarValue values[];
        if(PRTF(CalendarValueHistory(values, TimeCurrent(), TimeCurrent() + DAY_LONG *
        {
            for(int i = 0; i < ArraySize(values); ++i)
            {
                MqlCalendarEvent event;
                CalendarEventById(values[i].event_id, event);
                if(event.type == CALENDAR_TYPE_INDICATOR && !values[i].HasActualValue())
                {
                    track = values[i];
                    PrintFormat("Started monitoring %lld", track.id);
                    StructPrint(MqlCalendarRecord(track), ARRAYPRINT_HEADER);
                    return;
                }
            }
        }
    }
    ...
}
```

找到的事件被复制到 *track* 并输出到日志。之后，每次调用 *OnTimer* 都归结为将事件的更新信息获取到 *update* 结构体中，该结构体通过 *track.id* 标识符传递给 *CalendarValueById*。接下来，使用辅助函数 *StructCompare*（基于 *StructToCharArray* 和 *ArrayCompare*，请参见完整的源代码）比较原始结构体和新结构体。任何差异都会导致打印新状态（预测可能已更改），如果出现当前值，则计时器停止。要开始等待下一条新闻，需要重新初始化此指标：这只是一个演示，为了根据新闻列表控制情况，我们稍后将开发一个更实用的筛选器类。

```

else
{
    MqlCalendarValue update;
    if(CalendarValueById(track.id, update))
    {
        if(fabs(StructCompare(track, update)) == 1)
        {
            Alert(StringFormat("News %lld changed", track.id));
            PrintFormat("New state of %lld", track.id);
            StructPrint(MqlCalendarRecord(update), ARRAYPRINT_HEADER);
            if(update.HasActualValue())
            {
                Print("Timer stopped");
                EventKillTimer();
            }
            else
            {
                track = update;
            }
        }
    }

    if(TimeCurrent() <= track.time)
    {
        Comment("Forthcoming event time: ", track.time,
            ", remaining: ", Timing::stringify((uint)(track.time - TimeCurrent())));
    }
    else
    {
        Comment("Forthcoming event time: ", track.time,
            ", late for: ", Timing::stringify((uint)(TimeCurrent() - track.time)));
    }
}
}

```

在等待事件期间，指标会显示一条注释，其中包含新闻发布的预期时间以及距离发布还有多少时间（或延迟了多少时间）。



关于等待或错过下一条新闻的评论

需要注意的是，新闻可能会比预定日期稍早或稍晚发布。这在历史上测试新闻策略时会产生一些问题，因为终端和通过 MQL5 API 更新日历条目的时间并未提供。我们将在下一节中尝试部分解决这个问题。

以下是指标产生的日志输出片段，中间有间隔：

```
CalendarValueHistory(values,TimeCurrent(),TimeCurrent()+(60*60*24)*3)=186 / ok
Started monitoring 156045
[id] [event_id] [time] [period] [revision] »
156045 840020013 2022.06.27 15:30:00 2022.05.01 00:00:00 0 »
» [actual_value] [prev_value] [revised_prev_value] [forecast_value] [impact_type] »
» -9223372036854775808 400000 -9223372036854775808 0 0 »
» [importance] [name] [currency] [code] [actual] [previous] [revised] [forecast]
» "Medium" "Durable Goods Orders m/m" "USD" "US" nan 0.40000 nan 0.00000
...
Alert: News 156045 changed
New state of 156045
[id] [event_id] [time] [period] [revision] »
156045 840020013 2022.06.27 15:30:00 2022.05.01 00:00:00 0 »
» [actual_value] [prev_value] [revised_prev_value] [forecast_value] [impact_type] »
» 700000 400000 -9223372036854775808 0 1 »
» [importance] [name] [currency] [code] [actual] [previous] [revised] [forecast]
» "Medium" "Durable Goods Orders m/m" "USD" "US" 0.70000 0.40000 nan 0.00000
Timer stopped
```

更新后的新闻具有 *actual_value* 值。

为了在测试期间避免等待过久，建议在主要市场的工作时间内运行此指标，此时新闻发布的密度较高。

CalendarValueById 函数并非唯一，也可能不是最灵活的监控日历变化的方法。我们将在后续章节中探讨其他几种方法。

7.3.8 按国家/地区或货币跟踪事件变化

如 [日历的基本概念](#) 一节所述，平台通过某些内部方式注册所有事件变更。每个状态都由一个变更标识符 (*change_id*) 来表征。在 MQL5 函数中，有两个函数允许你找到此标识符 (在任意时间点)，然后请求在此时间点之后发生变更的日历条目。其中一个函数是 *CalendarValueLast*，本节将对其进行讨论。第二个函数 *CalendarValueLastByEvent* 将在下一节讨论。

```
int CalendarValueLast(ulong &change_id, MqlCalendarValue &values[],
    const string country = NULL, const string currency = NULL)
```

CalendarValueLast 函数设计用于两个目的：获取最后已知的日历变更标识符 *change_id*，并用自上次 (由相同 *change_id* 中传入的 ID 指定) 修改以来已修改的记录填充 *values* 数组。换句话说，*change_id* 参数既作为输入也作为输出。因此，它是一个引用，并且需要指定一个变量。

如果我们将等于 0 的 *change_id* 输入到函数中，则函数将用当前标识符填充该变量，但不会填充数组。

(可选) 使用参数 *country* 和 *currency*，你可以按国家/地区和货币设置记录筛选。

该函数返回复制的日历项数量。由于在第一种操作模式下 (*change_id* = 0) 数组不会被填充，因此返回 0 不是错误。如果自指定的变更以来日历未被修改，我们也可能得到 0。因此，要检查错误，你应该分析 *_LastError*。

因此，使用该函数的通常方法是循环检查日历的变更。

```
ulong change = 0;
MqlCalendarValue values[];
while(!IsStopped())
{
    // pass the last identifier known to us and get a new one if it appeared
    if(CalendarValueLast(change, values))
    {
        // analysis of added and changed records
        ArrayPrint(values);
        ...
    }
    Sleep(1000);
}
```

这可以在循环中、按计时器或按其他事件进行。

标识符不断增加，但它们可能会乱序，也就是说，可能会跳过几个值。

需要注意的是，每个日历条目始终仅以其最后一个状态可用：MQL5 中不提供变更历史。通常，这不是问题，因为每个新闻的生命周期都是标准的：提前足够长的时间添加到数据库，并在事件发生时补充相关数据。然而，在实践中，可能会发生各种偏差：编辑预测、调整时间或修正数值。无法通过 MQL5 API 从日历历史中准确查明记录中何时以及何处发生了更改。因此，那些根据即时情况做出决策的交易系统将需要独立保存变更历史并将其集成到 EA 交易中，以便在测试程序中运行。

使用 *CalendarValueLast* 函数，我们可以创建一个有用的服务 *CalendarChangeSaver.mq5*，它将按指定的时间间隔检查日历的变更，如果存在变更，则将变更标识符与当前服务器时间一起保存到文件中。这将允许后续使用文件信息对 EA 交易在日历历史上的测试进行更真实的模拟。当然，这将需要规划整个日历数据库的导出/导入，我们将逐步处理这个问题。

我们提供输入变量来指定文件名和轮询之间的时间段（以毫秒为单位）。

```
input string Filename = "calendar.chn";
input int PeriodMsc = 1000;
```

在 *OnStart* 处理程序的开头，我们打开二进制文件进行写入，或者更确切地说是追加（如果文件已存在）。此处未检查现有文件的格式，因此在嵌入到实际应用程序中时应添加保护。

```
void OnStart()
{
    ulong change = 0, last = 0;
    int count = 0;
    int handle = FileOpen(Filename,
        FILE_WRITE | FILE_READ | FILE_SHARE_WRITE | FILE_SHARE_READ | FILE_BIN);
    if(handle == INVALID_HANDLE)
    {
        PrintFormat("Can't open file '%s' for writing", Filename);
        return;
    }

    const ulong p = FileSize(handle);
    if(p > 0)
    {
        PrintFormat("Resuming file %lld bytes", p);
        FileSeek(handle, 0, SEEK_END);
    }

    Print("Requesting start ID...");
    ...
}
```

这里我们插入一个题外话。

每次日历更改时，至少必须将一对整数 8 字节数字写入文件：当前时间（*datetime*）和新闻 ID（*ulong*），但同时更改的记录可能不止一条。因此，除了日期之外，已更改记录的数量也打包到第一个数字中。这考虑到了日期可存储在 0x7FFFFFFF 范围内，因此高位的 3 个字节未使用。而服务将在对应时间戳之后写入的标识符数量，就存储在这两个最高有效字节中（位于左移 48 位的偏移位置）。PACK_DATETIME_COUNTER 宏创建一个“扩展”日期，而另外两个宏 DATETIME 和 COUNTER，我们稍后在读取变更存档时（由另一个程序）会需要。

```
#define PACK_DATETIME_COUNTER(D,C) (D | (((ulong)(C)) << 48))
#define DATETIME(A) ((datetime)((A) & 0x7FFFFFFF))
#define COUNTER(A) ((ushort)((A) >> 48))
```

现在我们回到主要的服务代码。在一个每 *PeriodMsc* 毫秒激活一次的循环中，我们使用 *CalendarValueLast* 请求变更。如果存在变更，我们将当前服务器时间和接收到的标识符数组写入文件。

```

while(!IsStopped())
{
    if(!TerminalInfoInteger(TERMINAL_CONNECTED))
    {
        Print("Waiting for connection...");
        Sleep(PeriodMsc);
        continue;
    }

    MqlCalendarValue values[];
    const int n = CalendarValueLast(change, values);
    if(n > 0)
    {
        string records = "[" + Description(values[0]);
        for(int i = 1; i < n; ++i)
        {
            records += "," + Description(values[i]);
        }
        records += "]";
        Print("New change ID: ", change, " ",
            TimeToString(TimeTradeServer(), TIME_DATE | TIME_SECONDS), "\n", records)
        FileWriteLong(handle, PACK_DATETIME_COUNTER(TimeTradeServer(), n));
        for(int i = 0; i < n; ++i)
        {
            FileWriteLong(handle, values[i].id);
        }
        FileFlush(handle);
        ++count;
    }
    else if(_LastError == 0)
    {
        if(!last && change)
        {
            Print("Start change ID obtained: ", change);
        }
    }

    last = change;
    Sleep(PeriodMsc);
}
PrintFormat("%d records added", count);
FileClose(handle);
}

```

为了方便地显示每个新闻事件的信息，我们编写了一个辅助函数 *Description*。

```

string Description(const MqlCalendarValue &value)
{
    MqlCalendarEvent event;
    MqlCalendarCountry country;
    CalendarEventById(value.event_id, event);
    CalendarCountryById(event.country_id, country);
    return StringFormat("%lld (%s/%s @ %s)",
        value.id, country.code, event.name, TimeToString(value.time));
}

```

因此，日志不仅会显示标识符，还会显示新闻的国家/地区代码、标题和预定时间。

假设该服务应该运行相当长的时间，以便收集足够测试期间（天、周、月）的信息。不过遗憾的是，就像订单簿一样，平台不提供现成的订单簿或日历编辑历史，因此这些内容的收集工作就完全留给了 MQL 程序的开发者。

我们来看看该服务的实际应用。在日志的下一段片段中（时间段为 2022.06.28 15:30 - 16:00），一些新闻事件与遥远的未来相关（它们包含 `prev_value` 字段的值，该字段也是同名当前事件的 ***actual_value*** 字段）。然而，更重要的是：新闻发布的实际时间有时可能与计划时间相差很大，有时甚至相差几分钟。

```

Requesting start ID...
Start change ID obtained: 86358784
New change ID: 86359040 2022.06.28 15:30:42
[155955 (US/Wholesale Inventories m/m @ 2022.06.28 15:30)]
New change ID: 86359296 2022.06.28 15:30:45
[155956 (US/Wholesale Inventories m/m @ 2022.07.08 17:00)]
New change ID: 86359552 2022.06.28 15:30:48
[156117 (US/Goods Trade Balance @ 2022.06.28 15:30)]
New change ID: 86359808 2022.06.28 15:30:51
[156118 (US/Goods Trade Balance @ 2022.07.27 15:30)]
New change ID: 86360064 2022.06.28 15:30:54
[156231 (US/Retail Inventories m/m @ 2022.06.28 15:30)]
New change ID: 86360320 2022.06.28 15:30:57
[156232 (US/Retail Inventories m/m @ 2022.07.15 17:00)]
New change ID: 86360576 2022.06.28 15:31:00
[156255 (US/Retail Inventories excl. Autos m/m @ 2022.06.28 15:30)]
New change ID: 86360832 2022.06.28 15:31:03
[156256 (US/Retail Inventories excl. Autos m/m @ 2022.07.15 17:00)]
New change ID: 86361088 2022.06.28 15:31:07
[155956 (US/Wholesale Inventories m/m @ 2022.07.08 17:00)]
New change ID: 86361344 2022.06.28 15:31:10
[156118 (US/Goods Trade Balance @ 2022.07.27 15:30)]
New change ID: 86361600 2022.06.28 15:31:13
[156232 (US/Retail Inventories m/m @ 2022.07.15 17:00)]
New change ID: 86362368 2022.06.28 15:36:47
[158534 (US/Challenger Job Cuts y/y @ 2022.07.07 14:30)]
New change ID: 86362624 2022.06.28 15:51:23
...
New change ID: 86364160 2022.06.28 16:01:39
[154531 (US/HPI m/m @ 2022.06.28 16:00)]
New change ID: 86364416 2022.06.28 16:01:42
[154532 (US/HPI m/m @ 2022.07.26 16:00)]
New change ID: 86364672 2022.06.28 16:01:46
[154543 (US/HPI y/y @ 2022.06.28 16:00)]
New change ID: 86364928 2022.06.28 16:01:49
[154544 (US/HPI y/y @ 2022.07.26 16:00)]
New change ID: 86365184 2022.06.28 16:01:54
[154561 (US/HPI @ 2022.06.28 16:00)]
New change ID: 86365440 2022.06.28 16:01:58
[154571 (US/HPI @ 2022.07.26 16:00)]
New change ID: 86365696 2022.06.28 16:02:01
[154532 (US/HPI m/m @ 2022.07.26 16:00)]
New change ID: 86365952 2022.06.28 16:02:05
[154544 (US/HPI y/y @ 2022.07.26 16:00)]
New change ID: 86366208 2022.06.28 16:02:09
[154571 (US/HPI @ 2022.07.26 16:00)]

```

当然，这种差异并非对所有类别的交易策略都重要，只对那些在市场上快速交易的策略重要。对于它们来说，创建的日历编辑存档可以为新闻 EA 交易提供更准确的测试。我们将在未来讨论如何将日历“连接”到测试程序，但现在，我们将展示如何读取接收到的文件。

我们将使用 *CalendarChangeReader.mq5* 脚本来演示所讨论的功能。在实践中，给定的源代码应该放在 EA 交易中。

输入变量允许你设置要读取的文件名和扫描的开始日期。如果服务继续工作（写入文件），你需要将文件复制为不同名称或复制到另一个文件夹中（在示例脚本中，文件被重命名）。如果 *Start* 参数为空，则新闻变更的读取将从当天的开始处开始。

```
input string Filename = "calendar2.chn";
input datetime Start;
```

ChangeState 结构体用于存储有关单个编辑的信息。

```
struct ChangeState
{
    datetime dt;
    ulong ids[];

    ChangeState(): dt(LONG_MAX) {}
    ChangeState(const datetime at, ulong &_ids[])
    {
        dt = at;
        ArraySwap(ids, _ids);
    }

    void operator=(const ChangeState &other)
    {
        dt = other.dt;
        ArrayCopy(ids, other.ids);
    }
};
```

它用在 *ChangeFileReader* 类中，该类完成了读取文件并将适合特定时间点的变更提供给调用者的大部分工作。

文件句柄作为参数传递给构造函数，测试的开始时间也是如此。读取文件并为一个日历编辑填充 *ChangeState* 结构体，这两个操作是在 *readState* 方法中执行的。

```

class ChangeFileReader
{
    const int handle;
    ChangeState current;
    const ChangeState zero;

public:
    ChangeFileReader(const int h, const datetime start = 0): handle(h)
    {
        if(readState())
        {
            if(start)
            {
                ulong dummy[];
                check(start, dummy, true); // find the first edit after start
            }
        }
    }

    bool readState()
    {
        if(FileIsEnding(handle)) return false;
        ResetLastError();
        const ulong v = FileReadLong(handle);
        current.dt = DATETIME(v);
        ArrayFree(current.ids);
        const int n = COUNTER(v);
        for(int i = 0; i < n; ++i)
        {
            PUSH(current.ids, FileReadLong(handle));
        }
        return _LastError == 0;
    }
    ...

```

check 方法读取文件，直到将来出现下一个编辑。在这种情况下，自上次方法调用以来的所有先前（按时间戳）编辑都放在输出数组 *records* 中。

```
bool check(datetime now, ulong &records[], const bool fastforward = false)
{
    if(current.dt > now) return false;

    ArrayFree(records);

    if(!fastforward)
    {
        ArrayCopy(records, current.ids);
        current = zero;
    }

    while(readState() && current.dt <= now)
    {
        if(!fastforward) ArrayInsert(records, current.ids, ArraySize(records));
    }

    return true;
}
};
```

下文介绍了如何在 *OnStart* 中使用该类。


```

void OnStart()
{
    const long day = 60 * 60 * 24;
    datetime now = Start ? Start : (datetime)(TimeCurrent() / day * day);

    int handle = FileOpen(Filename,
        FILE_READ | FILE_SHARE_WRITE | FILE_SHARE_READ | FILE_BIN);
    if(handle == INVALID_HANDLE)
    {
        PrintFormat("Can't open file '%s' for reading", Filename);
        return;
    }

    ChangeFileReader reader(handle, now);

    // reading step by step, time now artificially increased in this demo
    while(!FileIsEnding(handle))
    {
        // in a real application, a call to reader.check can be made on every tick
        ulong records[];
        if(reader.check(now, records))
        {
            Print(now);           // output time
            ArrayPrint(records); // array of IDs of changed news
        }
        now += 60; // add 1 minute at a time, can be per second
    }

    FileClose(handle);
}

```

以下是针对与前一个日志片段上下文中服务保存的相同日历变更的脚本结果。

```

2022.06.28 15:31:00
155955 155956 156117 156118 156231 156232 156255
2022.06.28 15:32:00
156256 155956 156118 156232
2022.06.28 15:37:00
158534
...
2022.06.28 16:02:00
154531 154532 154543 154544 154561 154571
2022.06.28 16:03:00
154532 154544 154571

```

相同的标识符在虚拟时间中以与在线时相同的延迟再现，尽管在这里你可以看到四舍五入到 1 分钟，这是因为我们在循环中设置了如此大小的人为步长。理论上，出于效率考虑，我们可以将检查推迟到存储在 *ChangeState current* 结构体中的时间。附件的源代码定义了 *getState* 方法来获取这个时间。

7.3.9 按类型跟踪事件变化

借助 MQL5 API, 你不仅可以请求整个日历或按国家/地区或货币的近期变更, 还可以在更窄的范围内, 或者更确切地说, 针对特定类型事件发出请求。

理论上, 我们可以说内置函数根据几个基本条件 (时间、国家、货币或事件类型) 提供事件筛选。对于其他特性, 例如重要性或经济板块, 你需要实现自己的筛选, 我们稍后会处理这个问题。现在, 我们介绍 *CalendarValueLastByEvent* 函数。

```
int CalendarValueLastByEvent(ulong id, ulong &change_id, MqlCalendarValue &values[])
```

该函数用自 *change_id* 以来发生的、具有 *id* 标识符的特定类型事件的记录来填充通过引用传递的 *values* 数组。这个 *change_id* 参数既是输入也是输出: 调用代码在其中传递日历过去状态的标签 (在此之后请求变更), 在交回控制权后, 函数会将日历数据库状态的当前标签写入 *change_id*。下次调用该函数时应使用这个参数。

如果在 *change_id* 中传递 *null*, 则函数不填充数组, 而只是通过参数 *change_id* 发送数据库的当前状态。

该数组可以是动态的 (在这种情况下, 将根据数据量自动调整) 或固定大小的 (如果其大小不足, 则仅复制适合的数据)。

函数的输出值等于复制到 *values* 数组中的元素数量。如果没有变更或指定了 *change_id = 0*, 则函数将返回 0。

要检查错误, 请分析内置的 *_LastError* 变量。以下列出了一些可能的错误代码:

- ⌚ 4004 -ERR_NOT_ENOUGH_MEMORY (内存不足, 无法完成请求),
- ⌚ 5401 -ERR_CALENDAR_TIMEOUT (请求超时),
- ⌚ 5400 -ERR_CALENDAR_MORE_DATA (固定数组的大小不足以获取所有值)。

我们不会为 *CalendarValueLastByEvent* 提供单独的示例。相反, 让我们转向一个更复杂但需求量更大的任务: 根据新闻特性的任意条件查询和筛选日历条目, 其中将涉及所有 “日历” API 函数。这将是下一节的主题。

7.3.10 多条件筛选事件

正如本章前面章节所述, MQL5 API 允许你基于若干条件请求经济日历事件:

- 按国家/地区 (*CalendarValueHistory*、*CalendarValueLast*)
- 按频率 (*CalendarValueHistory*、*CalendarValueLast*)
- 按事件类型 ID (*CalendarValueHistoryByEvent*、*CalendarValueLastByEvent*)
- 按时间范围 (*CalendarValueHistory*、*CalendarValueHistoryByEvent*)
- 按自上次日历轮询以来的变更 (*CalendarValueLast*、*CalendarValueLastByEvent*)
- 按特定新闻的 ID (*CalendarValueById*)

这可以总结为以下函数表 (在所有 *CalendarValue* 函数中, 此处仅缺少用于获取单个特定值的 *CalendarValueById*)。

条 件	时 间 范 围	最 新 变 更
国 家 / 地 区	CalendarValueHistory	CalendarValueLast
货 币	CalendarValueHistory	CalendarValueLast
事 件	CalendarValueHistoryByEvent	CalendarValueLastByEvent

这样的工具集涵盖了主要 (但并非全部)流行的日历分析场景。因此,在实践中,常常需要在 MQL5 中实现自定义的筛选机制,尤其包括按以下条件请求事件:

- 多个国家/地区
- 多种货币
- 多种事件类型
- 事件的任意特性值 (重要性、经济板块、报告期、类型、是否有预测、对汇率的估计影响、事件名称中的子字符串等)

为了解决这些问题,我们创建了 *CalendarFilter* 类 (*CalendarFilter.mqh*)。

由于内置 API 函数的特性,某些新闻特性被赋予了比其他特性更高的优先级。这包括国家/地区、货币和日期范围。它们可以在类的构造函数中指定,之后相应的特性便不能在筛选条件中动态更改。

这是因为该筛选器类后续将扩展新闻缓存功能,以便能够在测试程序中读取,而构造函数的初始条件实际上定义了可以进行进一步筛选的缓存上下文。例如,如果在创建对象时指定了国家代码 "EU",那么通过它请求美国或巴西的新闻显然没有意义。日期范围也类似:在构造函数中指定日期范围导致无法接收该日期范围之外的新闻。

我们也可以创建一个没有初始条件的对象 (因为所有构造函数参数都是可选的),这样它就能够缓存和筛选整个日历数据库中的新闻 (截至保存那一刻)。

此外,由于国家和货币现在几乎是唯一对应的 (欧盟和欧元除外),它们通过单个参数 *context* 传递给构造函数:如果指定一个长度为 2 个字符的字符串,则表示国家代码 (或国家组合);如果长度为 3 个字符,则表示货币代码。对于代码 "EU" 和 "EUR",欧元区是 "EU" (在有正式条约的国家内)的一个子集。在特殊情况下,如果关注非欧元区的欧盟国家,它们也可以用 "EU" 上下文来描述。如有必要,针对这些国家/地区货币 (BGN、HUF、DKK、ISK、PLN、RON、HRK、CZK、SEK) 新闻的更窄条件可以使用我们稍后将介绍的方法动态添加到筛选器中。然而,由于这些属于特殊情况,无法保证此类新闻一定会进入日历。

我们来开始研究这个类。

```

class CalendarFilter
{
protected:
    // initial (optional) conditions set in the constructor, invariants
    string context;    // country and currency
    datetime from, to; // date range
    bool fixedDates;   // if 'from'/'to' are passed in the constructor, they cannot be

    // dedicated selectors (countries/currencies/event type identifiers)
    string country[], currency[];
    ulong ids[];

    MqlCalendarValue values[]; // filtered results

    virtual void init()
    {
        fixedDates = from != 0 || to != 0;
        if(StringLen(context) == 3)
        {
            PUSH(currency, context);
        }
        else
        {
            // even if context is NULL, we take it to poll the entire calendar base
            PUSH(country, context);
        }
    }
    ...
public:
    CalendarFilter(const string _context = NULL,
        const datetime _from = 0, const datetime _to = 0):
        context(_context), from(_from), to(_to)
    {
        init();
    }
    ...

```

为国家/地区和货币分配了两个数组：*country* 和 *currency*。如果在对象创建期间它们没有从 *context* 中填充，那么 MQL 程序将能够为多个国家/地区或货币添加条件，以便对它们执行组合新闻查询。

为了存储所有其他新闻特性的条件，在 *CalendarFilter* 对象中描述了 *selectors* 数组，其第二维等于 3。我们可以说这是一种表，其中每行有 3 列。

```
long selectors[][3]; // [0] - property, [1] - value, [2] - condition
```

在第 0 个索引处，将存放新闻特性的标识符。由于这些特性分布在三个基础表 (*MqlCalendarCountry*、*MqlCalendarEvent*、*MqlCalendarValue*) 中，它们使用广义枚举 *ENUM_CALENDAR_PROPERTY* (*CalendarDefines.mqh*) 的元素进行描述。

```

enum ENUM_CALENDAR_PROPERTY
{
    CALENDAR_PROPERTY_COUNTRY_ID,           // +/- means support for field filtering
    CALENDAR_PROPERTY_COUNTRY_NAME,         // -ulong
    CALENDAR_PROPERTY_COUNTRY_CODE,         // -string
    CALENDAR_PROPERTY_COUNTRY_CURRENCY,     // +string (2 characters)
    CALENDAR_PROPERTY_COUNTRY_GLYPH,        // +string (3 characters)
    CALENDAR_PROPERTY_COUNTRY_URL,          // -string (1 characters)
    CALENDAR_PROPERTY_COUNTRY_URL,          // -string

    CALENDAR_PROPERTY_EVENT_ID,             // +ulong (event type ID)
    CALENDAR_PROPERTY_EVENT_TYPE,           // +ENUM_CALENDAR_EVENT_TYPE
    CALENDAR_PROPERTY_EVENT_SECTOR,         // +ENUM_CALENDAR_EVENT_SECTOR
    CALENDAR_PROPERTY_EVENT_FREQUENCY,      // +ENUM_CALENDAR_EVENT_FREQUENCY
    CALENDAR_PROPERTY_EVENT_TIMEMODE,       // +ENUM_CALENDAR_EVENT_TIMEMODE
    CALENDAR_PROPERTY_EVENT_UNIT,           // +ENUM_CALENDAR_EVENT_UNIT
    CALENDAR_PROPERTY_EVENT_IMPORTANCE,     // +ENUM_CALENDAR_EVENT_IMPORTANCE
    CALENDAR_PROPERTY_EVENT_MULTIPLIER,     // +ENUM_CALENDAR_EVENT_MULTIPLIER
    CALENDAR_PROPERTY_EVENT_DIGITS,         // -uint
    CALENDAR_PROPERTY_EVENT_SOURCE,         // +string ("http[s]://")
    CALENDAR_PROPERTY_EVENT_CODE,           // -string
    CALENDAR_PROPERTY_EVENT_NAME,           // +string (4+ characters or wildcard '*')

    CALENDAR_PROPERTY_RECORD_ID,            // -ulong
    CALENDAR_PROPERTY_RECORD_TIME,          // +datetime
    CALENDAR_PROPERTY_RECORD_PERIOD,        // +datetime (like long)
    CALENDAR_PROPERTY_RECORD_REVISION,      // +int
    CALENDAR_PROPERTY_RECORD_ACTUAL,        // +long
    CALENDAR_PROPERTY_RECORD_PREVIOUS,      // +long
    CALENDAR_PROPERTY_RECORD_REVISED,      // +long
    CALENDAR_PROPERTY_RECORD_FORECAST,      // +long
    CALENDAR_PROPERTY_RECORD_IMPACT,        // +ENUM_CALENDAR_EVENT_IMPACT

    CALENDAR_PROPERTY_RECORD_PREVISED,     // +non-standard (previous or revised if any)

    CALENDAR_PROPERTY_CHANGE_ID,            // -ulong (reserved)
};

```

索引 1 将存储用于在选择新闻记录的条件中与它们进行比较的值。例如，如果要按经济板块设置筛选器，则在 `selectors[i][0]` 中写入 `CALENDAR_PROPERTY_EVENT_SECTOR`，并在 `selectors[i][1]` 中写入标准枚举 `ENUM_CALENDAR_EVENT_SECTOR` 的一个值。

最后，最后一列（第 2 个索引之下）保留用于比较选择器值与新闻中特性值的操作：所有支持的操作都总结在 IS 枚举中。

```
enum IS
{
    EQUAL,
    NOT_EQUAL,
    GREATER,
    LESS,
    OR_EQUAL,
    ...
};
```

我们在 *TradeFilter.mqh* 中见过类似的方法。因此，我们将不仅能安排值的相等条件，还能安排不等或大于/小于关系的条件。例如，很容易想象一个针对 `CALENDAR_PROPERTY_EVENT_IMPORTANCE` 字段的筛选器，该字段应 `GREATER`（大于）`CALENDAR_IMPORTANCE_LOW`（这是标准 `ENUM_CALENDAR_EVENT_IMPORTANCE` 枚举的一个元素），这意味着选择重要性为“中等”和“高”的新闻。

下一个专为日历定义的枚举是 `ENUM_CALENDAR_SCOPE`。由于日历筛选通常与时间跨度相关，这里列出了最常用的时间跨度。

```
#define DAY_LONG      (60 * 60 * 24)
#define WEEK_LONG     (DAY_LONG * 7)
#define MONTH_LONG    (DAY_LONG * 30)
#define QUARTER_LONG  (MONTH_LONG * 3)
#define YEAR_LONG     (MONTH_LONG * 12)

enum ENUM_CALENDAR_SCOPE
{
    SCOPE_DAY = DAY_LONG,          // Day
    SCOPE_WEEK = WEEK_LONG,        // Week
    SCOPE_MONTH = MONTH_LONG,      // Month
    SCOPE_QUARTER = QUARTER_LONG,  // Quarter
    SCOPE_YEAR = YEAR_LONG,        // Year
};
```

所有枚举都放在一个单独的头文件 *CalendarDefines.mqh* 中。

我们回到 *CalendarFilter* 类。*selectors* 数组的类型是 *long*，这适合存储几乎所有相关类型的值：枚举、日期和时间、标识符、整数，甚至经济指标值，因为它们在日历中以 *long* 数字的形式存储（以实际值的百万分之一为单位）。然而，如何处理字符串特性呢？

这个问题通过使用字符串数组 *stringCache* 来解决，所有在筛选条件中提及的行都将添加到此数组中。

```
class CalendarFilter
{
protected:
    ...
    string stringCache[]; // cache of all rows in 'selectors'
    ...
};
```

然后，我们可以在 *selectors[i][1]* 中轻松保存 *stringCache* 数组中元素的索引，而不是字符串值本身。

为了用筛选条件填充 *selectors* 数组，提供了几个 *let* 方法，特别是针对枚举：

```

class CalendarFilter
{
...
public:
    // all fields of enum types are processed here
    template<typename E>
    CalendarFilter *let(const E e, const IS c = EQUAL)
    {
        const int n = EXPAND(selectors);
        selectors[n][0] = resolve(e); // by type E, returning the element ENUM_CALENDAR
        selectors[n][1] = e;
        selectors[n][2] = c;
        return &this;
    }
    ...
}

```

对于指标的实际值：

```

// the following fields are processed here:
// CALENDAR_PROPERTY_RECORD_ACTUAL, CALENDAR_PROPERTY_RECORD_PREVIOUS,
// CALENDAR_PROPERTY_RECORD_REVISIED, CALENDAR_PROPERTY_RECORD_FORECAST,
// and CALENDAR_PROPERTY_RECORD_PERIOD (as long)
CalendarFilter *let(const long value, const ENUM_CALENDAR_PROPERTY property, const
{
    const int n = EXPAND(selectors);
    selectors[n][0] = property;
    selectors[n][1] = value;
    selectors[n][2] = c;
    return &this;
}
...

```

以及对于字符串：

```

// conditions for all string properties can be found here (abbreviated)
CalendarFilter *let(const string find, const IS c = EQUAL)
{
    const int wildcard = (StringFind(find, "*") + 1) * 10;
    switch(StringLen(find) + wildcard)
    {
    case 2:
        // if the initial context is different from the country, we can supplement i
        // otherwise the filter is ignored
        if(StringLen(context) != 2)
        {
            if(ArraySize(country) == 1 && StringLen(country[0]) == 0)
            {
                country[0] = find; // narrow down "all countries" to one (may add more
            }
            else
            {
                PUSH(country, find);
            }
        }
        break;
    case 3:
        // we can set a filter for a currency only if it was not in the initial cont
        if(StringLen(context) != 3)
        {
            PUSH(currency, find);
        }
        break;
    default:
        {
            const int n = EXPAND(selectors);
            PUSH(stringCache, find);
            if(StringFind(find, "http://") == 0 || StringFind(find, "https://") == 0)
            {
                selectors[n][0] = CALENDAR_PROPERTY_EVENT_SOURCE;
            }
            else
            {
                selectors[n][0] = CALENDAR_PROPERTY_EVENT_NAME;
            }
            selectors[n][1] = ArraySize(stringCache) - 1;
            selectors[n][2] = c;
            break;
        }
    }

    return &this;
}

```

在针对字符串的方法重载中, 请注意, 长度为 2 或 3 个字符的字符串 (如果它们没有模板星号 *, 该星号是任意字符序列的替代品) 分别进入国家/地区和符号 (货币) 数组, 而所有其他字符串则被视作名称或新闻来源的片段, 这两个字段都涉及 *stringCache* 和 *selectors*。

该类还以特殊方式支持按事件类型 (标识符) 进行筛选。

```
protected:
    ulong ids[];           // filtered event types
    ...
public:
    CalendarFilter *let(const ulong event)
    {
        PUSH(ids, event);
        return &this;
    }
    ...
```

因此, 优先筛选器 (在 `selectors` 数组之外处理) 的数量不仅包括国家/地区、货币和日期范围, 还包括事件类型标识符。这样的构造决策是由于这些参数可以作为输入传递给某些日历 API 函数。我们以结构体数组 (*MqlCalendarValue*、*MqlCalendarEvent*、*MqlCalendarCountry*) 中输出字段值的形式获取所有其他新闻特性。我们将根据 `selectors` 数组中的规则, 基于这些结构体执行二次筛选。

所有 *let* 方法都返回一个指向对象的指针, 这允许它们的调用可以链式进行。例如, 像这样:

```
CalendarFilter f;
f.let(CALENDAR_IMPORTANCE_LOW, GREATER) // important and moderately important news
  .let(CALENDAR_TIMEMODE_DATETIME) // only events with exact time
  .let("DE").let("FR") // a couple of countries, or, to choose from...
  .let("USD").let("GBP") // ...a couple of currencies (but both conditions won't work
  .let(TimeCurrent() - MONTH_LONG, TimeCurrent() + WEEK_LONG) // date range "around"
  .let(LONG_MIN, CALENDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL) // there is a forecast
  .let("farm"); // full text search by news titles
```

理论上, 国家/地区和货币条件可以组合使用。但是, 请注意, 只能为国家/地区或货币之一 (而非两者) 设置多个值。在当前实现中, 这两个上下文方面中的一个 (两者任选其一) 仅支持一个或零个值 (即不对其进行筛选)。例如, 如果选择了货币 `EUR`, 则可以将新闻的搜索上下文缩小到仅德国和法国 (国家代码 `"DE"` 和 `"FR"`)。结果, 欧洲央行和欧盟统计局的新闻将被丢弃, 具体来说, 意大利和西班牙的新闻也将被丢弃。然而, 在这种情况下, 指定 `EUR` 是多余的, 因为德国和法国没有其他货币。

由于该类使用内置函数, 其中 *country* 和 *currency* 参数使用逻辑 “与” 运算应用于新闻, 因此请检查筛选条件的一致性。

在调用代码设置了筛选条件之后, 有必要根据这些条件选择新闻。这就是公共方法 *select* 的作用 (为简化起见, 此处给出简化版)。

```

public:
    bool select(MqlCalendarValue &result[])
    {
        int count = 0;
        ArrayFree(result);
        if(ArraySize(ids)) // identifiers of event types
        {
            for(int i = 0; i < ArraySize(ids); ++i)
            {
                MqlCalendarValue temp[];
                if(PRTF(CalendarValueHistoryByEvent(ids[i], temp, from, to)))
                {
                    ArrayCopy(result, temp, ArraySize(result));
                    ++count;
                }
            }
        }
        else
        {
            // several countries or currencies, choose whichever is more as a basis,
            // only the first element from the smaller array is used
            if(ArraySize(country) > ArraySize(currency))
            {
                const string c = ArraySize(currency) > 0 ? currency[0] : NULL;
                for(int i = 0; i < ArraySize(country); ++i)
                {
                    MqlCalendarValue temp[];
                    if(PRTF(CalendarValueHistory(temp, from, to, country[i], c)))
                    {
                        ArrayCopy(result, temp, ArraySize(result));
                        ++count;
                    }
                }
            }
            else
            {
                const string c = ArraySize(country) > 0 ? country[0] : NULL;
                for(int i = 0; i < ArraySize(currency); ++i)
                {
                    MqlCalendarValue temp[];
                    if(PRTF(CalendarValueHistory(temp, from, to, c, currency[i])))
                    {
                        ArrayCopy(result, temp, ArraySize(result));
                        ++count;
                    }
                }
            }
        }

        if(ArraySize(result) > 0)
        {

```

```

        filter(result);
    }

    if(count > 1 && ArraySize(result) > 1)
    {
        SORT_STRUCT(MqlCalendarValue, result, time);
    }

    return ArraySize(result) > 0;
}

```

根据已填充的优先属性数组中的哪一个，该方法调用不同的 API 函数来轮询日历：

- 如果 *ids* 数组已填充，则对所有标识符循环调用 *CalendarValueHistoryByEvent*
- 如果 *country* 数组已填充且其大小大于货币数组，则调用 *CalendarValueHistory* 并遍历国家/地区
- 如果 *currency* 数组已填充且其大小大于或等于国家/地区数组的大小，则调用 *CalendarValueHistory* 并遍历货币

每个函数调用都会填充一个临时的结构体数组 *MqlCalendarValue temp[]*，该数组会顺序累积到 *result* 参数数组中。在根据主要条件（日期、国家/地区、货币、标识符）将所有相关新闻写入其中（如果存在主要条件）之后，一个辅助方法 *filter* 开始发挥作用，它根据 *selectors* 中的条件筛选该数组。在 *select* 方法的末尾，新闻条目按时间顺序排序，因为组合多个“日历”函数查询的结果可能会打乱顺序。排序是使用 *SORT_STRUCT* 宏实现的，这在 [数组的比较、排序和搜索](#) 一节中已经讨论过。

对于新闻数组中的每个元素，*filter* 方法会调用工作方法 *match*，该方法返回一个布尔指示，表明该新闻是否符合筛选条件。如果不符合，则从数组中移除该元素。

```

protected:
void filter(MqlCalendarValue &result[])
{
    for(int i = ArraySize(result) - 1; i >= 0; --i)
    {
        if(!match(result[i]))
        {
            ArrayRemove(result, i, 1);
        }
    }
}
...

```

最后，*match* 方法会分析我们的 *selectors* 数组，并将其与传入的 *MqlCalendarValue* 结构体的字段进行比较。此处的代码以缩略形式提供。

```

bool match(const MqlCalendarValue &v)
{
    MqlCalendarEvent event;
    if(!CalendarEventById(v.event_id, event)) return false;

    // loop through all filter conditions, except for countries, currencies, dates,
    // which have already been previously used when calling Calendar functions
    for(int j = 0; j < ArrayRange(selectors, 0); ++j)
    {
        long field = 0;
        string text = NULL;

        // get the field value from the news or its description
        switch((int)selectors[j][0])
        {
            case CALENDAR_PROPERTY_EVENT_TYPE:
                field = event.type;
                break;
            case CALENDAR_PROPERTY_EVENT_SECTOR:
                field = event.sector;
                break;
            case CALENDAR_PROPERTY_EVENT_TIMEMODE:
                field = event.time_mode;
                break;
            case CALENDAR_PROPERTY_EVENT_IMPORTANCE:
                field = event.importance;
                break;
            case CALENDAR_PROPERTY_EVENT_SOURCE:
                text = event.source_url;
                break;
            case CALENDAR_PROPERTY_EVENT_NAME:
                text = event.name;
                break;
            case CALENDAR_PROPERTY_RECORD_IMPACT:
                field = v.impact_type;
                break;
            case CALENDAR_PROPERTY_RECORD_ACTUAL:
                field = v.actual_value;
                break;
            case CALENDAR_PROPERTY_RECORD_PREVIOUS:
                field = v.prev_value;
                break;
            case CALENDAR_PROPERTY_RECORD_REVISIED:
                field = v.revised_prev_value;
                break;
            case CALENDAR_PROPERTY_RECORD_PREVISIED: // previous or revised (if any)
                field = v.revised_prev_value != LONG_MIN ? v.revised_prev_value : v.prev_
                break;
            case CALENDAR_PROPERTY_RECORD_FORECAST:
                field = v.forecast_value;
                break;
        }
    }
}

```

```

...
}

// compare value with filter condition
if(text == NULL) // numeric fields
{
    switch((IS)selectors[j][2])
    {
        case EQUAL:
            if(!equal(field, selectors[j][1])) return false;
            break;
        case NOT_EQUAL:
            if(equal(field, selectors[j][1])) return false;
            break;
        case GREATER:
            if(!greater(field, selectors[j][1])) return false;
            break;
        case LESS:
            if(greater(field, selectors[j][1])) return false;
            break;
    }
}
else // string fields
{
    const string find = stringCache[(int)selectors[j][1]];
    switch((IS)selectors[j][2])
    {
        case EQUAL:
            if(!equal(text, find)) return false;
            break;
        case NOT_EQUAL:
            if(equal(text, find)) return false;
            break;
        case GREATER:
            if(!greater(text, find)) return false;
            break;
        case LESS:
            if(greater(text, find)) return false;
            break;
    }
}
}

return true;
}

```

equal 和 *greater* 方法几乎完全复制了我们之前使用筛选器类开发时所用的方法。

至此，筛选问题基本得到解决，也就是说，MQL 程序可以按以下方式使用 *CalendarFilter* 对象：

```
CalendarFilter f;
f.let()... // a series of calls to the let method to set filtering conditions
MqlCalendarValue records[];
if(f.select(records))
{
    ArrayPrint(records);
}
```

实际上，**select** 方法还有另一项重要功能，我们将其留作独立研究课题。

首先，在最终的新闻列表中，最好能在过去和未来之间以某种方式插入一个分隔符 (**delimiter**)，以便于快速定位。理论上，这个功能对于日历来说极其重要，但出于某种原因，它在 MetaTrader 5 用户界面和 mq5.com 网站上都不可用。我们的实现能够在过去和未来之间插入一个空结构体，我们应该在视觉上将其显示出来（我们将在下面处理这个问题）。

其次，结果数组的大小可能会相当大（尤其是在选择设置的初始阶段），因此 **select** 方法还额外提供了限制数组大小 (**limit**) 的功能。这是通过移除距当前时间最远的元素来实现的。

因此，完整的方法原型如下所示：

```
bool select(MqlCalendarValue &result[],
            const bool delimiter = false, const int limit = -1);
```

默认情况下，不插入定界符，数组也不会被截断。

前几段我们提到了筛选的一个附加子任务，即结果数组的可视化。**CalendarFilter** 类有一个特殊的方法 **format**，它将传入的 **MqlCalendarValue &data[]** 结构体数组转换为一个人类可读的字符串数组 **string &result[]**。该方法的代码可以在随附的文件 **CalendarFilter.mqh** 中找到。

```
bool format(const MqlCalendarValue &data[],
            const ENUM_CALENDAR_PROPERTY &props[], string &result[],
            const bool padding = false, const bool header = false);
```

我们想要显示的 **MqlCalendarValue** 字段是在 **props** 数组中指定的。回想一下，**ENUM_CALENDAR_PROPERTY** 枚举包含了所有三个相关的日历结构体中的字段，因此 MQL 程序不仅可以自动显示特定事件记录中的经济指标，还可以显示其名称、特性、国家/地区或货币代码。所有这些都由 **format** 方法实现。

输出的 **result** 数组中的每一行都包含一个字段值（数字、描述、枚举元素）的文本表示。**result** 数组的大小等于输入结构体数量（在 **data** 中）与所显示字段数量（在 **props** 中）的乘积。可选参数 **header** 允许你在输出数组的开头添加一个包含字段（例）名称的行。**padding** 参数控制文本中额外空格的生成，以便于在等宽字体中显示表格（例如，在日志中）。

CalendarFilter 类还有另一个重要的公共方法：**update**。

```
bool update(MqlCalendarValue &result[]);
```

其结构几乎完全重复了 **select**。然而，该方法调用的是 **CalendarValueHistoryByEvent** 和 **CalendarValueHistory** 函数，而不是 **CalendarValueLastByEvent** 和 **CalendarValueLast**。该方法的目的显而易见：查询日历中符合筛选条件的最近变更。但其运行需要一个变更 ID。类中确实定义了这样一个字段：它在 **select** 方法内部首次被填充。

```

class CalendarFilter
{
protected:
    ...
    ulong change;
    ...
public:
    bool select(MqlCalendarValue &result[],
        const bool delimiter = false, const int limit = -1)
    {
        ...
        change = 0;
        MqlCalendarValue dummy[];
        CalendarValueLast(change, dummy);
        ...
    }
}

```

CalendarFilter 类的一些细微之处仍然“处于幕后”，但我们将在后续章节中逐步讨论。

我们来实际测试一下这个筛选器：首先在一个简单的脚本 *CalendarFilterPrint.mq5* 中，然后在一个更实用的指标 *CalendarMonitor.mq5* 中。

在脚本的输入参数中，你可以设置上下文（国家/地区代码或货币）、时间范围以及用于按事件名称进行全文搜索的字符串，还可以限制结果新闻表的大小。

```

input string Context; // Context (country - 2 characters, currency - 3 characters, en
input ENUM_CALEDAR_SCOPE Scope = SCOPE_MONTH;
input string Text = "farm";
input int Limit = -1;

```

根据这些参数，会创建一个全局筛选器对象。

```
CalendarFilter f(Context, TimeCurrent() - Scope, TimeCurrent() + Scope);
```

然后，在 *OnStart* 中，我们配置了几个额外的常量条件（事件的重要性为“中等”和“高”）以及是否存在预测（字段不等于 `LONG_MIN`），并将一个搜索字符串传递给对象。

```

void OnStart()
{
    f.let(CALENDAR_IMPORTANCE_LOW, GREATER)
        .let(LONG_MIN, CALEDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL)
        .let(Text); // with '*' replacement support
    // NB: strings with the character length of 2 or 3 without '*' will be treated
    // as a country or currency code, respectively
}

```

接下来，调用 *select* 方法，并使用 *format* 方法将得到的 *MqlCalendarValue* 结构体数组格式化为一个包含 9 列的表格。

```

MqlCalendarValue records[];
// apply the filter conditions and get the result
if(f.select(records, true, Limit))
{
    static const ENUM_CALEDAR_PROPERTY props[] =
    {
        CALEDAR_PROPERTY_RECORD_TIME,
        CALEDAR_PROPERTY_COUNTRY_CURRENCY,
        CALEDAR_PROPERTY_EVENT_NAME,
        CALEDAR_PROPERTY_EVENT_IMPORTANCE,
        CALEDAR_PROPERTY_RECORD_ACTUAL,
        CALEDAR_PROPERTY_RECORD_FORECAST,
        CALEDAR_PROPERTY_RECORD_PREVISED,
        CALEDAR_PROPERTY_RECORD_IMPACT,
        CALEDAR_PROPERTY_EVENT_SECTOR,
    };
    static const int p = ArraySize(props);

    // output the formatted result
    string result[];
    if(f.format(records, props, result, true, true))
    {
        for(int i = 0; i < ArraySize(result) / p; ++i)
        {
            Print(SubArrayCombine(result, " | ", i * p, p));
        }
    }
}
}

```

表格的单元格连接成行并输出到日志中。

使用默认设置 (即, 针对所有国家和货币, 事件名称中包含 "fam" 部分且重要性为 "中等"和 "高"), 你可以得到类似这样的时间表。


```

Selecting calendar records...
country[i]= / ok
calendarValueHistory(temp,from,to,country[i],c)=2372 / ok
Filtering 2372 records
Got 9 records
TIME | CUR | NAME | IMPORTAN | ACTU | FORE | PREV | IMPACT | SECT
2022.06.02 15:15 | USD | ADP Nonfarm Employment Change | HIGH | +128 | -225 | +202 |
2022.06.02 15:30 | USD | Nonfarm Productivity q/q | MODERATE | -7.3 | -7.5 | -7.5 | P
2022.06.03 15:30 | USD | Nonfarm Payrolls | HIGH | +390 | -19 | +436 | POSITIVE | JOB
2022.06.03 15:30 | USD | Private Nonfarm Payrolls | MODERATE | +333 | +8 | +405 | POS
2022.06.09 08:30 | EUR | Nonfarm Payrolls q/q | MODERATE | +0.3 | +0.3 | +0.3 | NA |
- | - | - | - | - | - | - | - | - | -
2022.07.07 15:15 | USD | ADP Nonfarm Employment Change | HIGH | +nan | -263 | +128 |
2022.07.08 15:30 | USD | Nonfarm Payrolls | HIGH | +nan | -229 | +390 | NA | JOBS
2022.07.08 15:30 | USD | Private Nonfarm Payrolls | MODERATE | +nan | +51 | +333 | NA

```

现在我们看一下 *CalendarMonitor.mqh* 指标。它的目的是根据指定的筛选条件，在图表上向用户显示当前选定的事件。为了可视化这个表格，我们将使用已经熟悉的计分板类 *Tableau.mqh*，参见 [期货订单的保证金计算](#) 一节）。该指标没有缓冲区和图表。

输入参数允许你设置时间窗口的范围 (*scope*)，以及 *CalendarFilter* 对象的全局上下文，上下文可以是 *Context* 中的货币或国家/地区代码 (默认为空，即无限制)，或者通过布尔标志 *UseChartCurrencies* 来设置。该标志默认启用，建议使用它以便自动接收构成图表工作交易品种的那些货币的新闻。

```

input string Context; // Context (country - 2 chars, currency - 3 chars, empty - all)
input ENUM_CALENDAR_SCOPE Scope = SCOPE_WEEK;
input bool UseChartCurrencies = true;

```

可以为事件类型、板块和重要性应用额外的筛选器。

```

input ENUM_CALENDAR_EVENT_TYPE_EXT Type = TYPE_ANY;
input ENUM_CALENDAR_EVENT_SECTOR_EXT Sector = SECTOR_ANY;
input ENUM_CALENDAR_EVENT_IMPORTANCE_EXT Importance = IMPORTANCE_MODERATE; // Importa

```

Importance (重要性) 设置的是选择的下限，而不是精确匹配。因此，默认值 IMPORTANCE_MODERATE 不仅会捕获中等重要性的事件，还会捕获高重要性的事件。

细心的读者会注意到这里使用了一些未知的枚举：ENUM_CALENDAR_EVENT_TYPE_EXT、ENUM_CALENDAR_EVENT_SECTOR_EXT、ENUM_CALENDAR_EVENT_IMPORTANCE_EXT。它们位于前面提到的 *CalendarDefines.mqh* 文件中，并且 (几乎一对一地) 与类似的内置枚举相吻合。唯一的区别是它们增加了一个表示“任何”值的元素。我们需要描述这样的枚举是为了简化条件的输入：现在每个字段的筛选器都使用一个下拉列表进行配置，你可以从中选择一个值或关闭筛选器。如果不是因为增加了枚举元素，我们就必须为每个字段在界面中输入一个逻辑上的“开/关”标志。

此外，输入参数允许你根据事件中是否存在实际值、预测值和前次指标来查询事件，以及通过搜索文本字符串 (*Text*) 来查询。

```

input string Text;
input ENUM_CALEDAR_HAS_VALUE HasActual = HAS_ANY;
input ENUM_CALEDAR_HAS_VALUE HasForecast = HAS_ANY;
input ENUM_CALEDAR_HAS_VALUE HasPrevious = HAS_ANY;
input ENUM_CALEDAR_HAS_VALUE HasRevised = HAS_ANY;
input int Limit = 30;

```

CalendarFilter 和 *tableau* 对象在全局级别进行描述。

```

CalendarFilter f(Context);
AutoPtr<Tableau> t;

```

请注意，筛选器只创建一次，而表格则由一个自动选择器表示，并将根据接收数据的大小动态重新创建。

筛选器设置是在 *OnInit* 中根据输入参数通过连续调用 *let* 方法完成的。

```

int OnInit()
{
    if(!f.isLoaded()) return INIT_FAILED;

    if(UseChartCurrencies)
    {
        const string base = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_BASE);
        const string profit = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_PROFIT);
        f.let(base);
        if(base != profit)
        {
            f.let(profit);
        }
    }

    if(Type != TYPE_ANY)
    {
        f.let((ENUM_CALENDAR_EVENT_TYPE)Type);
    }

    if(Sector != SECTOR_ANY)
    {
        f.let((ENUM_CALENDAR_EVENT_SECTOR)Sector);
    }

    if(Importance != IMPORTANCE_ANY)
    {
        f.let((ENUM_CALENDAR_EVENT_IMPORTANCE)(Importance - 1), GREATER);
    }

    if(StringLen(Text))
    {
        f.let(Text);
    }

    if(HasActual != HAS_ANY)
    {
        f.let(LONG_MIN, CALENDAR_PROPERTY_RECORD_ACTUAL,
            HasActual == HAS_SET ? NOT_EQUAL : EQUAL);
    }
    ...

    EventSetTimer(1);

    return INIT_SUCCEEDED;
}

```

最后，会启动一个辅助计时器。所有工作都在 *OnTimer* 中实现。

```

void OnTimer()
{
    static const ENUM_CALEDAR_PROPERTY props[] = // table columns
    {
        CALEDAR_PROPERTY_RECORD_TIME,
        CALEDAR_PROPERTY_COUNTRY_CURRENCY,
        CALEDAR_PROPERTY_EVENT_NAME,
        CALEDAR_PROPERTY_EVENT_IMPORTANCE,
        CALEDAR_PROPERTY_RECORD_ACTUAL,
        CALEDAR_PROPERTY_RECORD_FORECAST,
        CALEDAR_PROPERTY_RECORD_PREVISED,
        CALEDAR_PROPERTY_RECORD_IMPACT,
        CALEDAR_PROPERTY_EVENT_SECTOR,
    };
    static const int p = ArraySize(props);

    MqlCalendarValue records[];

almost one to one    f.let(TimeCurrent() - Scope, TimeCurrent() + Scope); // shift the

    const ulong trackID = f.getChangeID();
    if(trackID) // if the state has already been removed, check for changes
    {
        if(f.update(records)) // request changes by filters
        {
            // if there are changes, notify the user
            string result[];
            f.format(records, props, result);
            for(int i = 0; i < ArraySize(result) / p; ++i)
            {
                Alert(SubArrayCombine(result, " | ", i * p, p));
            }
            // "fall through" further to update the table
        }
        else if(trackID == f.getChangeID())
        {
            return; // calendar without changes
        }
    }

    // request a complete set of news by filters
    f.select(records, true, Limit);

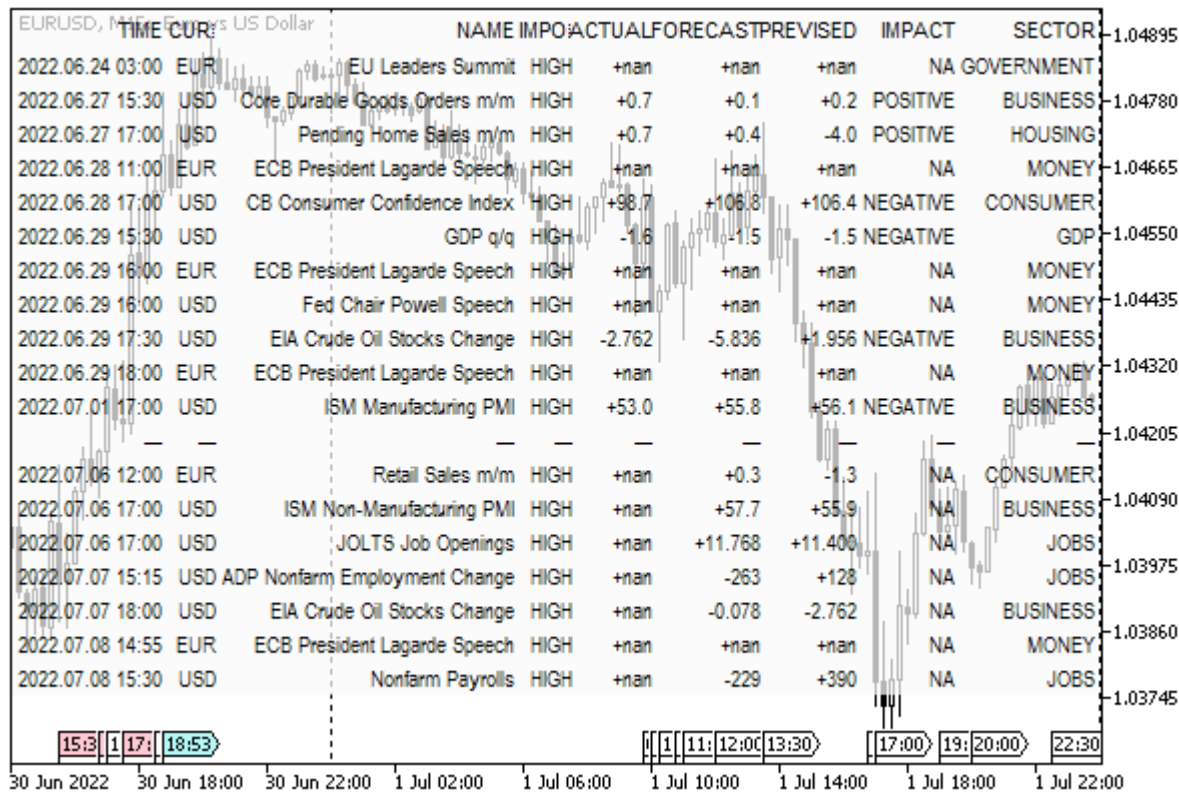
    // display the news table on the chart
    string result[];
    f.format(records, props, result, true, true);

    if(t[] == NULL || t[].getRows() != ArraySize(records) + 1)
    {
        t = new Tableau("CALT", ArraySize(records) + 1, p,
            TBL_CELL_HEIGHT_AUTO, TBL_CELL_WIDTH_AUTO,

```

```
Corner, Margins, FontSize, FontName, FontName + " Bold",
TBL_FLAG_ROW_0_HEADER,
BackgroundColor, BackgroundTransparency);
}
const string hints[] = {};
t[].fill(result, hints);
}
```

如果我们在 EURUSD 图表上使用默认设置运行该指标，我们可以得到如下图所示的画面。



图表上已筛选和格式化的新闻集

7.3.11 将日历数据库传输到测试程序

日历仅在线时可供 MQL 程序使用，因此测试新闻交易策略会带来一些困难。一种解决方案是独立创建日历的某个镜像，即缓存，然后在测试程序内部使用它。缓存存储技术可以不同，例如文件或嵌入式 SQLite 数据库。在本节中，我们将展示一个使用文件的实现。

无论如何，在使用日历缓存时，请记住它对应于特定的时间点 X。在 X 之前发生的所有“旧”事件（财经报告）中，实际值已经设定，而在之后的事件中（相对于 X 的“未来”），没有实际值，并且在出现更新、更近期的缓存副本之前也不会有。换句话说，测试 X 右侧的指标和 EA 交易没有意义。至于 X 左侧的那些，你应该避免“前瞻性偏差”，也就是说，在每个特定新闻发布时间之前不要读取当前指标。

注意！在终端中请求日历数据时，所有事件的时间都会报告为考虑到服务器当前时区的时间，包括可能对“夏令时”进行的修正（通常这意味着时间戳增加 1 小时）。这样可以使新闻发布与在线报价时间同步。然而，过去的时钟变化（半年前、一年前或更早）仅显示在报价中，而不显示在日历事件中。整个日历数据库通过 MQL5 根据服务器的当前时区读取。因此，任何创建的日历存档都将包含那些在存储时与当时活动的 DST 模式（开启或关闭）相同

的事件的正确时间戳。对于“相反”半年的事件，在读取存档后需要独立地进行一小时的调整。在下面的示例中，这种情况被忽略了。

我们将缓存类称为 *CalendarCache* 并将其放在名为 *CalendarCache.mqh* 的文件中。我们将需要在文件中保存日历库的所有 3 个表 (*MqlCalendarCountry*、*MqlCalendarEvent*、*MqlCalendarValue*)。MQL5 提供了函数 *FileWriteArray* 和 *FileReadArray* (参见 [读写数组](#))，它们可以直接将简单结构体数组写入文件和从文件读取。然而，在我们的案例中，3 个结构体中有 2 个没那么简单，因为它们有字符串字段。因此，我们需要一种单独存储字符串的机制，类似于我们已经在 *CalendarFilter* 类中使用过的那种 (那里有一个字符串数组 *stringCache*，并且在筛选器中指明了该数组中所需字符串的索引)。

为了避免在同一个“字典”中遗漏来自不同“日历”结构体的字符串，我们将准备一个模板类 *StringRef*：类型参数 *T* 将是任何 *MqlCalendar* 结构体。这将为我们的国家/地区的独立字符串缓存，以及一个用于事件类型的独立字符串缓存。

```

template<typename T>
struct StringRef
{
    static string cache[];
    int index;
    StringRef(): index(-1) { }

    void operator=(const string s)
    {
        if(index == -1)
        {
            PUSH(cache, s);
            index = ArraySize(cache) - 1;
        }
        else
        {
            cache[index] = s;
        }
    }

    string operator[](int x = 0) const
    {
        if(index != -1)
        {
            return cache[index];
        }
        return NULL;
    }

    static bool save(const int handle)
    {
        FileWriteInteger(handle, ArraySize(cache));
        for(int i = 0; i < ArraySize(cache); ++i)
        {
            FileWriteInteger(handle, StringLen(cache[i]));
            FileWriteString(handle, cache[i]);
        }
        return true;
    }

    static bool load(const int handle)
    {
        const int n = FileReadInteger(handle);
        for(int i = 0; i < n; ++i)
        {
            PUSH(cache, FileReadString(handle, FileReadInteger(handle)));
        }
        return true;
    }
};

```

```
template<typename T>
static string StringRef::cache[];
```

字符串通过使用 `operator=` 存储在 `cache` 数组中，并使用 `operator[]` 从中提取（带有一个总是被忽略的虚拟索引）。每个对象只存储字符串在数组中的索引。`cache` 数组被声明为静态，因此它将累积一个 `T` 结构体的所有字符串字段。可根据需要更改缓存方法，使结构体的每个字段都有其自己的数组，但这对于我们来说并不重要。

将数组写入文件和从文件读取由一对静态方法 `save` 和 `load` 执行：两者都以文件句柄作为参数。

考虑到 `StringRef` 类，我们描述一些结构体，它们复制了标准的日历结构体，但使用 `StringRef` 对象代替字符串字段。例如，对于 `MqlCalendarCountry`，我们得到 `MqlCalendarCountryRef`。标准结构体和修改后的结构体通过重载的运算符 `'='` 和 `'[]'` 以类似的方式相互复制。

```
struct MqlCalendarCountryRef
{
    ulong id;
    StringRef<MqlCalendarCountry> name;
    StringRef<MqlCalendarCountry> code;
    StringRef<MqlCalendarCountry> currency;
    StringRef<MqlCalendarCountry> currency_symbol;
    StringRef<MqlCalendarCountry> url_name;

    void operator=(const MqlCalendarCountry &c)
    {
        id = c.id;
        name = c.name;
        code = c.code;
        currency = c.currency;
        currency_symbol = c.currency_symbol;
        url_name = c.url_name;
    }

    MqlCalendarCountry operator[](int x = 0) const
    {
        MqlCalendarCountry r;
        r.id = id;
        r.name = name[];
        r.code = code[];
        r.currency = currency[];
        r.currency_symbol = currency_symbol[];
        r.url_name = url_name[];
        return r;
    }
};
```

请注意，第一种方法的赋值运算符具有来自 `StringRef` 的重载 `'='`，因此所有行都进入数组 `StringRef<MqlCalendarCountry>::cache`。在第二种方法中，`'[]'` 运算符调用可隐式获取字符串的地址，并直接从 `StringRef` 返回存储在该地址的 `cache` 数组中的字符串。

MqlCalendarEventRef 结构体以类似的方式定义，但其中只有 3 个字段 (*source_url*、*event_code* 和 *name*) 需要将 *string* 类型替换为 *StringRef<MqlCalendarEvent>*。*MqlCalendarValue* 结构体不需要这样的转换，因为它没有字符串字段。

至此，准备阶段结束，你可以继续学习主要的缓存类 *CalendarCache*。

基于总体考量，以及为了与已经开发的 *CalendarFilter* 类兼容，我们在缓存中描述指定上下文 (国家/地区或货币)、存储事件的日期范围以及缓存生成时刻 (时间 X, 变量 t) 的字段。

```
class CalendarCache
{
    string context;
    datetime from, to;
    datetime t;
    ...

public:
    CalendarCache(const string _context = NULL,
                  const datetime _from = 0, const datetime _to = 0):
        context(_context), from(_from), to(_to), t(0)
    {
        ...
    }
}
```

实际上，创建日历缓存时设置限制并没有太大意义。一个完整的缓存可能更实用，因为缓存大小并没有那么重要，到 2022 年中期，缓存大小也只有几十兆字节 (这包括从 2007 年开始的历史数据以及计划到 2024 年的事件)。不过，功能被人为限制的演示程序可以适当采取限制措施。

很明显，缓存中应提供日历结构体数组来存储所有数据。

```
MqlCalendarValue values[];
MqlCalendarEvent events[];
MqlCalendarCountry countries[];
...
```

最初，由 *update* 方法根据日历数据库来填充这些数组。

```

bool update()
{
    string country = NULL, currency = NULL;
    if(StringLen(context) == 3)
    {
        currency = context;
    }
    else if(StringLen(context) == 2)
    {
        country = context;
    }

    Print("Reading online calendar base...");

    if(!PRTF(CalendarValueHistory(values, from, to, country, currency))
        || (currency != NULL ?
            !PRTF(CalendarEventByCurrency(currency, events)) :
            !PRTF(CalendarEventByCountry(country, events)))
        || !PRTF(CalendarCountries(countries)))
    {
        // object is not ready, t = 0
    }
    else
    {
        t = TimeTradeServer();
    }
    return (bool)t;
}

```

t 字段是缓存健康状况的标志，记录了填充数组的时间。

可以使用 `save` 方法，将填充的缓存对象写入文件。文件开头有一个标头 `CALENDAR_CACHE_HEADER` -这是字符串 `"MQL5 Calendar Cache\r\nv.1.0\r\n"`，它允许你在读取时确保格式正确。接下来，该方法“按原样”保存 *context*、*from*、*to* 和 *t* 变量以及 *values* 数组。在数组本身之前，我们写入其大小，以便在读取时恢复它。

```

bool save(string filename = NULL)
{
    if(!t) return false;

    MqlDateTime mdt;
    TimeToStruct(t, mdt);
    if(filename == NULL) filename = "calendar-" +
        StringFormat("%04d-%02d-%02d-%02d-%02d.cal",
            mdt.year, mdt.mon, mdt.day, mdt.hour, mdt.min);
    int handle = PRTF(FileOpen(filename, FILE_WRITE | FILE_BIN));
    if(handle == INVALID_HANDLE) return false;

    FileWriteString(handle, CALENDAR_CACHE_HEADER);
    FileWriteString(handle, context, 4);
    FileWriteLong(handle, from);
    FileWriteLong(handle, to);
    FileWriteLong(handle, t);
    FileWriteInteger(handle, ArraySize(values));
    FileWriteArray(handle, values);
    ...
}

```

对于 *events* 和 *countries* 数组,则使用我们带有 “Ref” 后缀的包装结构体。辅助方法 *store* 将 *events* 数组转换为一个简单结构体数组 *erefs*, 其中字符串被替换为字符串字典 *StringRef<MqlCalendarEvent>* 中的数字。这样的简单结构体已经可以按常规方式写入文件,但为了后续读取它们,还需要保存字典的所有行 调用 *StringRef<MqlCalendarEvent> ::save(handle)*。Country 结构体以相同的方式转换并保存到文件。

```

MqlCalendarEventRef erefs[];
store(erefs, events);
FileWriteInteger(handle, ArraySize(erefs));
FileWriteArray(handle, erefs);
StringRef<MqlCalendarEvent>::save(handle);

MqlCalendarCountryRef crefs[];
store(crefs, countries);
FileWriteInteger(handle, ArraySize(crefs));
FileWriteArray(handle, crefs);
StringRef<MqlCalendarCountry>::save(handle);

FileClose(handle);
return true;
}

```

前面提到的 *store* 方法非常简单:在该方法中,通过对元素的循环,在 *MqlCalendarEventRef* 或 *MqlCalendarCountryRef* 结构体中执行重载的赋值运算符。

```

template<typename T1,typename T2>
void static store(T1 &array[], T2 &origin[])
{
    ArrayResize(array, ArraySize(origin));
    for(int i = 0; i < ArraySize(origin); ++i)
    {
        array[i] = origin[i];
    }
}

```

为了将接收到的文件加载到缓存对象中，编写了一个镜像方法 *load*。该方法按相同的顺序将数据从文件读入变量和数组中，同时对事件类型和国家/地区的字符串字段执行反向转换。

```

bool load(const string filename)
{
    Print("Loading calendar cache ", filename);
    t = 0;
    int handle = PRTF(FileOpen(filename, FILE_READ | FILE_BIN));
    if(handle == INVALID_HANDLE) return false;

    const string header = FileReadString(handle, StringLen(CALENDAR_CACHE_HEADER));
    if(header != CALENDAR_CACHE_HEADER) return false; // not our format

    context = FileReadString(handle, 4);
    if(!StringLen(context)) context = NULL;
    from = (datetime)FileReadLong(handle);
    to = (datetime)FileReadLong(handle);
    t = (datetime)FileReadLong(handle);
    Print("Calendar cache interval: ", from, "-", to);
    Print("Calendar cache saved at: ", t);
    int n = FileReadInteger(handle);
    FileReadArray(handle, values, 0, n);

    MqlCalendarEventRef erefs[];
    n = FileReadInteger(handle);
    FileReadArray(handle, erefs, 0, n);
    StringRef<MqlCalendarEvent>::load(handle);
    restore(events, erefs);

    MqlCalendarCountryRef crefs[];
    n = FileReadInteger(handle);
    FileReadArray(handle, crefs, 0, n);
    StringRef<MqlCalendarCountry>::load(handle);
    restore(countries, crefs);

    FileClose(handle);
    ... // something else will be here
}

```

辅助方法 *restore* 在对 *MqlCalendarEventRef* 或 *MqlCalendarCountryRef* 结构体中的元素进行循环时使用 ‘[]’ 运算符的重载，以通过行号获取行本身并将其分配给标准的 *MqlCalendarEvent* 或 *MqlCalendarCountry* 结构体。

```

template<typename T1,typename T2>
void static restore(T1 &array[], T2 &origin[])
{
    ArrayResize(array, ArraySize(origin));
    for(int i = 0; i < ArraySize(origin); ++i)
    {
        array[i] = origin[i][];
    }
}

```

在这个阶段，我们已经可以编写一个基于 *CalendarCache* 类的简单测试指标，对在线图表运行这个指标，并将该指标与日历缓存一起保存到文件中。然后可以在测试程序中，通过指标副本加载该文件，并接收完整的事件集。然而，这对于实际开发来说还不够。

事实是，为了快速访问数据，需要提供 *indexing*，这是编程中一个众所周知的概念，我们稍后将在关于数据库的章节中讨论它。理论上，我们可以使用内置的 SQLite 引擎来存储缓存，然后我们将“免费”获得索引，但我们稍后再讨论细节。

如果我们想象如何在我们的缓存中有效地实现标准日历函数的类似功能，那么索引的意义就很容易理解了。例如，在 *CalendarValueById* 函数中传递事件 ID。直接枚举 *values* 数组中的记录将非常耗时。因此，需要用某种“数据结构”来补充该数组，以便优化搜索。我们之所以将“数据结构”用引号括起来，因为它不是指编程语言的含义 (*struct*)，而是泛指数据构建的体系结构。我们所谓的“数据结构”可以由不同的部分组成，并基于不同的组织原则。当然，额外的数据会需要内存，但用内存换取速度是编程中常见的做法。

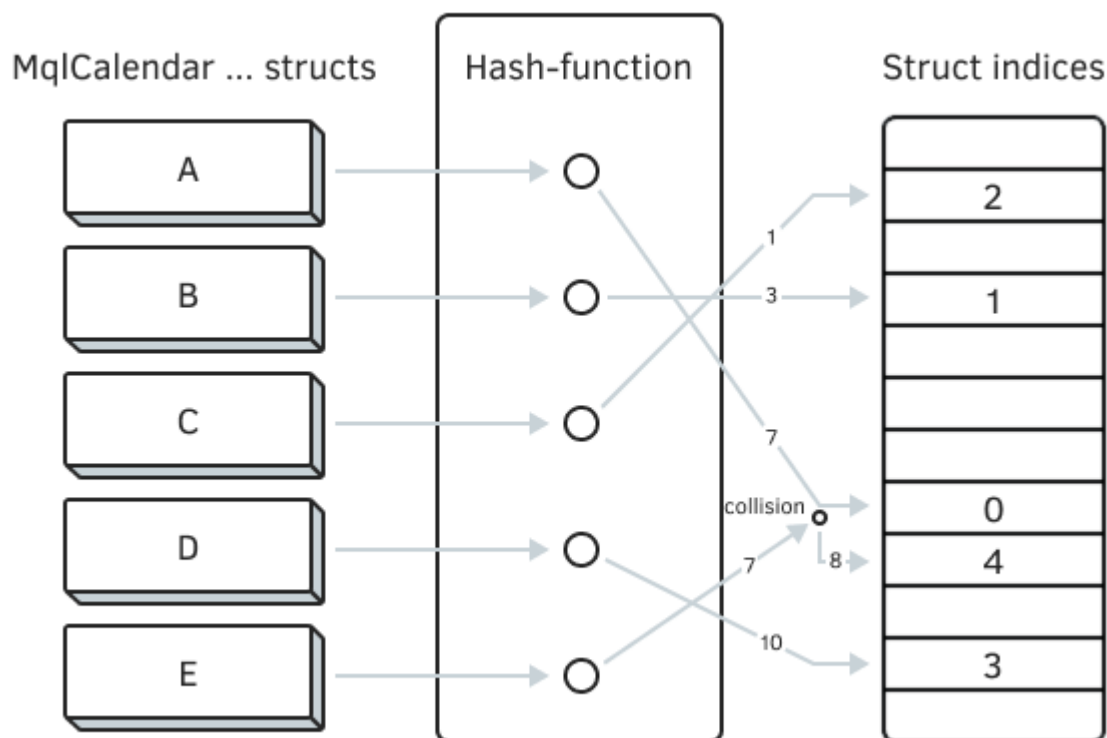
建立索引最简单的解决方案是一个单独的二维数组，按升序排序，以便可以使用 *ArrayBsearch* 函数快速搜索。该二维数组的第二维仅需两个元素：索引 *[i][0]* 处存储排序所用的标识符，索引 *[i][1]* 处则存储该结构体在原数组中的序号。

另一个常用的概念是 *hashing*，它是将初始值转换为某些键（哈希值，整数）的过程，其方式是提供最小数量的冲突（不同初始数据的键匹配）。键的基本特性是其值的接近均匀的随机分布，因此它们可以用作预分配数组中的索引。为原始数据的单个元素计算哈希函数是一个快速的过程，实际上可以得到元素本身的地址。例如，众所周知的哈希映射数据结构就遵循这个原则。

如果两个原始值确实得到了相同的哈希值（尽管这种情况很少见），它们会为它们的键排成一个列表，并在列表内执行顺序搜索。然而，由于选择的哈希函数使得匹配数量很小，通常在计算出哈希值后搜索就能命中目标。

为了演示，我们将在 *CalendarCache* 类中同时使用这两种方法：哈希和二分搜索。

MetaTrader 5 软件包包含一组用于创建哈希映射的类 (MQL5/Include/Generic/HashMap.mqh)，但我们将使用我们自己更简单的实现，其中只保留了使用哈希函数的原理。



通过哈希进行数据索引的方案

在我们的案例中，只需对日历对象的标识符进行哈希就足够了。我们选择的哈希函数必须将标识符转换为特殊数组内的索引：日历结构体数组中标识符的位置将存储在该索引的单元格中。对于国家/地区、事件类型和特定新闻，都分配了其各自的数组。

```
int id4country[];
int id4event[];
int id4value[];
```

它们的元素将条目的序号存储在相关数组 (*countries*, *events*, *values*) 中。

对于每个“重定向”数组，需要分配的结构体数量必须至少是数据库（和缓存）中相应结构体数量的 3 倍。由于这种冗余，我们最大限度地减少了哈希冲突的数量。据信，选择等于素数的大小可以实现最高效率。因此，该类有一个静态方法 *size2prime*，它根据源数据中的元素数量返回哈希“桶”数组 (*id4* 数组之一) 的推荐大小。

```

static int size2prime(const int size)
{
    static int primes[] =
    {
        17, 53, 97, 193, 389,
        769, 1543, 3079, 6151,
        12289, 24593, 49157, 98317,
        196613, 393241, 786433, 1572869,
        3145739, 6291469, 12582917, 25165843,
        50331653, 100663319, 201326611, 402653189,
        805306457, 1610612741
    };

    const int pmax = ArraySize(primes);
    for(int p = 0; p < pmax; ++p)
    {
        if(primes[p] >= 2 * size)
        {
            return primes[p];
        }
    }
    return size;
}

```

整个日历哈希过程在 *hash* 方法中描述。我们以 *countries* 结构体数组为例来看看它的开头，其他两个数组的处理方式类似。

因此，我们通过调用 *size2prime*，根据 *countries* 数组的大小计算出推荐的“素数”索引大小 *id4country*。最初，索引数组用值 -1 填充，也就是说，其所有元素都是空闲的。接下来，在遍历国家/地区的循环中，需要为每个下一个国家/地区标识符计算哈希值，并用于在 *id4country* 数组中找到一个空闲索引。这是辅助方法 *place* 的工作。

```

bool hash()
{
    Print("Hashing calendar...");
    ...
    const int c = PRTF(ArraySize(countries));
    PRTF(ArrayResize(id4country, size2prime(c)));
    ArrayInitialize(id4country, -1);

    for(int i = 0; i < c; ++i)
    {
        if(place(countries[i].id, i, id4country) == -1)
        {
            return false; // failure
        }
    }
    ...
    return true; // success
}

```

place 内部的哈希函数是表达式 $(\text{MathSwap}(\text{id}) \wedge 0\text{xEFCDA}8967452301) \% n$, 其中 *id* 是我们的标识符, *n* 是索引数组的大小。因此, 计算结果总是被缩减到 *array[]* 内的有效索引。选择哈希函数的原理是一个独立的主题, 超出了本书的范围。

```

int place(const ulong id, const int index, int &array[])
{
    const int n = ArraySize(array);
    int p = (int)((MathSwap(id) ^ 0xEFCDA8967452301) % n); // hash function
    int attempt = 0;
    while(array[p] != -1)
    {
        if(++attempt > n / 10) // number of collisions - no more than 1/10 of the nu
        {
            return -1; // error writing to index array
        }
        p = (p + attempt) % n;
    }
    array[p] = index;
    return p;
}

```

如果索引数组中 *p* 位置的单元格未被占用 (等于 -1), 我们立即将日历结构体的位置地址写入元素 [*p*]。如果单元格已被占用, 我们尝试使用公式 $p = (p + \text{attempt}) \% n$ 来选择下一个, 其中 *attempt* 是尝试次数的计数器 (这是我们伪装的具有匹配哈希值的元素列表版本)。如果失败尝试次数达到原始数据的十分之一, 索引将失败, 但鉴于我们的索引数组的超大规模和哈希数据的已知性质 (唯一标识符), 这种情况实际上是不可能发生的。

对结构体数组进行哈希处理后, 我们得到一个已填充的索引数组 (其中有空闲空间, 但这是有意设计的), 通过它可以根据日历元素的标识符在结构体数组中找到相应结构体的位置。这是通过与 *place* 含义相反的 *find* 方法完成的。


```

template<typename S>
int find(const ulong id, const int &array[], const S &structs[])
{
    const int n = ArraySize(array);
    if(!n) return false;
    int p = (int)((MathSwap(id) ^ 0xEFCDAB8967452301) % n); // hash function
    int attempt = 0;
    while(structs[array[p]].id != id)
    {
        if(++attempt > n / 10)
        {
            return -1; // error extracting from index array
        }
        p = (p + attempt) % n;
    }
    return array[p];
}

```

我们展示一下它的实际运用。标准日历函数包括 *CalendarCountryById* 和 *CalendarEventById*。当你在测试程序中测试 MQL 程序时，程序将无法直接访问这两个函数，但能够将日历缓存加载到 *CalendarCache* 对象中，因此它应该有类似的方法。

```

bool calendarCountryById(ulong country_id, MqlCalendarCountry &cnt)
{
    const int index = find(country_id, id4country, countries);
    if(index == -1) return false;

    cnt = countries[index];
    return true;
}

bool calendarEventById(ulong event_id, MqlCalendarEvent &event)
{
    const int index = find(event_id, id4event, events);
    if(index == -1) return false;

    event = events[index];
    return true;
}

```

它们使用 *find* 方法以及索引数组 *id4country* 和 *id4event*。

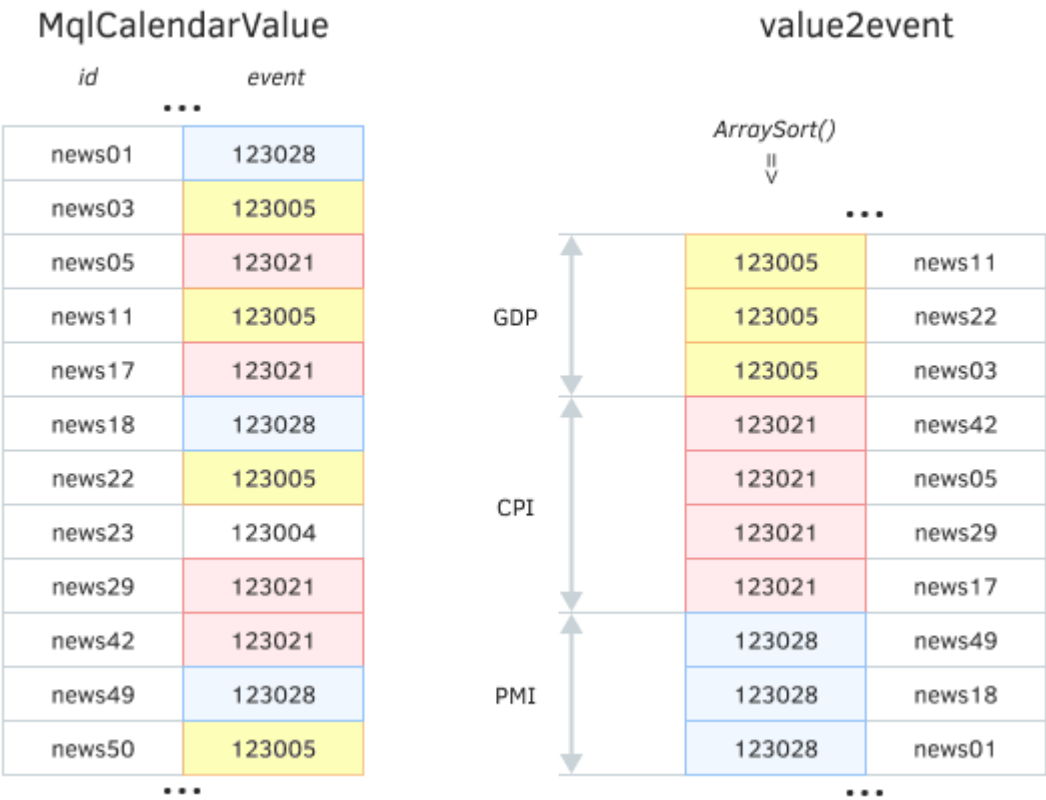
但这些并不是日历最需要的功能。更常见的情况是，包含新闻策略的 MQL 程序需要的是 *CalendarValueHistory*、*CalendarValueHistoryByEvent*、*CalendarValueLast* 或 *CalendarValueLastByEvent* 函数。它们提供了按时间、国家/地区或货币快速访问日历条目的功能。

因此，*CalendarCache* 类也应该提供类似的方法。这里我们将使用第二种“索引”方法—通过在已排序数组中进行二分搜索。

为了实现上述方法，我们向类中再添加 4 个二维数组，以建立新闻与事件类型、新闻与国家/地区、新闻与货币以及新闻与其发布时间之间的对应关系。

```
ulong value2event[][2];    // [0] - event_id, [1] - value_id
ulong value2country[][2]; // [0] - country_id, [1] - value_id
ulong value2currency[][2]; // [0] - currency ushort[4]<->long, [1] - value_id
ulong value2time[][2];    // [0] - time, [1] - value_id
```

在每行的第一个元素中，即索引 `[i][0]` 下，将分别记录事件 ID、国家/地区、货币或时间。在该系列的第二个元素中，即索引 `[i][1]` 下，将放置特定新闻的 ID。一次性填充所有数组后，将使用 `ArraySort` 按 `[i][0]` 的值对数组进行排序。然后，我们可以按 ID（例如，按 `event_id`）在 `value2event` 数组中搜索所有此类新闻：`ArrayBsearch` 函数将返回第一个匹配元素的编号，随后是具有相同 `event_id` 的其他元素，直到遇到不同的标识符。第二个“列”中的顺序未定义（可以是任意的）。



基于排序快速搜索相关结构体

这种不同类型结构体相互绑定的操作在 `bind` 方法中执行。每个“绑定”数组的大小与新闻数组的大小相同。通过循环遍历所有新闻，我们使用现成的索引数组和 `find` 方法进行快速寻址。

```

bool bind()
{
    Print("Binding calendar tables...");
    const int n = ArraySize(values);
    ArrayResize(value2event, n);
    ArrayResize(value2country, n);
    ArrayResize(value2currency, n);
    ArrayResize(value2time, n);
    for(int i = 0; i < n; ++i)
    {
        value2event[i][0] = values[i].event_id;
        value2event[i][1] = values[i].id;

        const int e = find(values[i].event_id, id4event, events);
        if(e == -1) return false;

        value2country[i][0] = events[e].country_id;
        value2country[i][1] = values[i].id;

        const int c = find(events[e].country_id, id4country, countries);
        if(c == -1) return false;

        value2currency[i][0] = currencyId(countries[c].currency);
        value2currency[i][1] = values[i].id;

        value2time[i][0] = values[i].time;
        value2time[i][1] = values[i].id;
    }
    ArraySort(value2event);
    ArraySort(value2country);
    ArraySort(value2currency);
    ArraySort(value2time);
    return true;
}

```

就货币而言，使用 `currencyId` 函数从字符串获得的特殊数字会被作为标识符。

```

static ulong currencyId(const string s)
{
    union CRNC4
    {
        ushort word[4];
        ulong ul;
    } v;
    StringToShortArray(s, v.word);
    return v.ul;
}

```

现在我们终于可以展示 `CalendarCache` 类的整个构造函数了。

```

CalendarCache(const string _context = NULL,
               const datetime _from = 0, const datetime _to = 0):
    context(_context), from(_from), to(_to), t(0), eventId(0)
{
    if(from > to) // label that context is a filename
    {
        load(_context);
    }
    else
    {
        if(!update() || !hash() || !bind())
        {
            t = 0;
        }
    }
}

```

在在线图表上启动时，使用默认参数创建的对象将收集所有日历信息 (*update*)，对其建立索引 (*hash*)，并链接这些表 (*bind*)。如果在任何阶段出现问题，变量 *t* 中的错误标志将为 0。如果成功，*TimeTradeServer* 函数的值将保留在那里（记住，该值是在 *update* 内部设置的）。这种随时可用的对象可以使用前面描述的 *save* 方法导出到文件。

在测试程序中启动时，应使用特殊的 *from* 和 *to* 参数组合 (*from > to*) 创建对象——在这种情况下，程序会将 *context* 字符串视为文件名，并从中加载日历状态。最简单的方法是这样的：

```
CalendarCache calca("filename.cal", true);
```

在 *load* 方法内部，我们还将调用 *hash* 和 *bind* 以使对象进入工作状态。

```

bool load(const string filename)
{
    ... // reading the file was shown earlier
    const bool result = hash() && bind();
    if(!result) t = 0;
    return result;
}

```

以 *CalendarValueLast* 函数为例，我们展示了 *calendarValueLast* 方法（具有完全相同的原型）的等效实现。由于没有开放的软件 API 来读取在线日历变更表，缓存将使用当前的“服务器”时间作为变更标识符。假设我们可以使用 *CalendarChangeSaver.mq5* 服务保存的变更 ID 信息，但这种方法需要在开始测试之前进行长期的统计数据收集。因此，测试程序生成的“服务器”时间被认为是相当充分的替代方案。

当 MQL 程序首次以空标识符请求变更时，我们直接返回 *TimeTradeServer* 提供的值。

```

int calendarValueLast(ulong &change, MqlCalendarValue &result[],
    const string code = NULL, const string currency = NULL)
{
    if(!change)
    {
        change = TimeTradeServer();
        return 0;
    }
    ...

```

如果变更标识符已经不为零，我们继续算法的主要分支。

根据 *code* 和 *currency* 参数的内容，我们找到国家/地区和货币的标识符。默认为 0，这意味着搜索所有变更。

```

ulong country_id = 0;
ulong currency_id = currency != NULL ? currencyId(currency) : 0;

if(code != NULL)
{
    for(int i = 0; i < ArraySize(countries); ++i)
    {
        if(countries[i].code == code)
        {
            country_id = countries[i].id;
            break;
        }
    }
}
...

```

接下来，使用传递的时间计数 *change* 作为搜索的开始，我们在 *value2time* 中找到所有新闻，直到新的、当前值 *TimeTradeServer*。在循环内部，我们使用 *find* 方法在 *values* 数组中查找相应 *MqlCalendarValue* 结构体的索引，并在必要时将关联事件类型的国家/地区和货币与期望值进行比较。所有符合标准的新闻项都写入 *result* 输出数组。

```

const ulong past = change;
const int index = ArrayBsearch(value2time, past);
if(index < 0 || index >= ArrayRange(value2time, 0)) return 0;

int i = index;
while(value2time[i][0] <= (ulong)past && i < ArrayRange(value2time, 0)) ++i;

if(i >= ArrayRange(value2time, 0)) return 0;

for(int j = i; j < ArrayRange(value2time, 0)
    && value2time[j][0] <= (ulong)TimeTradeServer(); ++j)
{
    const int p = find(value2time[j][1], id4value, values);
    if(p != -1)
    {
        change = TimeTradeServer();
        if(country_id != 0 || currency_id != 0)
        {
            const int q = find(values[p].event_id, id4event, events);
            if(country_id != 0 && country_id != events[q].country_id) continue;
            if(currency_id != 0)
            {
                const int m = find(events[q].country_id, id4country, countries);
                if(countries[m].currency != currency) continue;
            }
        }

        PUSH(result, values[p]);
    }
}

return ArraySize(result);
}

```

calendarValueHistory、*calendarValueHistoryByEvent* 和 *calendarValueLastByEvent* 方法的实现原理类似（后者实际上将所有工作委托给前面讨论的 *calendarValueLast* 方法）。完整的源代码可以在附件文件 *CalendarCache.mqh* 中找到。

基于缓存类，创建一个派生类 *CalendarFilter* 是合乎逻辑的，该类在处理请求时将访问缓存而不是日历。

完成的解决方案在文件 *CalendarFilterCached.mqh* 中。由于缓存 API 是在标准 API 的基础上设计的，集成仅简化为将筛选器调用转发到缓存对象（自动指针 *cache*）。

```

class CalendarFilterCached: public CalendarFilter
{
protected:
    AutoPtr<CalendarCache> cache;

    virtual bool calendarCountryById(ulong country_id, MqlCalendarCountry &cnt) override
    {
        return cache[].calendarCountryById(country_id, cnt);
    }

    virtual bool calendarEventById(ulong event_id, MqlCalendarEvent &event) override
    {
        return cache[].calendarEventById(event_id, event);
    }

    virtual int calendarValueHistoryByEvent(ulong event_id, MqlCalendarValue &temp[],
        datetime _from, datetime _to = 0) override
    {
        return cache[].calendarValueHistoryByEvent(event_id, temp, _from, _to);
    }

    virtual int calendarValueHistory(MqlCalendarValue &temp[],
        datetime _from, datetime _to = 0,
        const string _code = NULL, const string _coin = NULL) override
    {
        return cache[].calendarValueHistory(temp, _from, _to, _code, _coin);
    }

    virtual int calendarValueLast(ulong &_change, MqlCalendarValue &result[],
        const string _code = NULL, const string _coin = NULL) override
    {
        return cache[].calendarValueLast(_change, result, _code, _coin);
    }

    virtual int calendarValueLastByEvent(ulong event_id, ulong &_change,
        MqlCalendarValue &result[]) override
    {
        return cache[].calendarValueLastByEvent(event_id, _change, result);
    }

public:
    CalendarFilterCached(CalendarCache *_cache): cache(_cache),
        CalendarFilter(_cache.getContext(), _cache.getFrom(), _cache.getTo())
    {
    }

    virtual bool isLoading() const override
    {
        // readiness is determined by the cache
        return cache[].isLoading();
    }
}

```

```
};
```

为了在测试程序中测试日历，我们创建一个新版本的指标 *CalendarMonitorCached.mq5*。

CalendarMonitor.mq5 —

主要区别如下。

我们假设某个缓存文件将以名称 “xyz.cal” (在文件夹 *MTL5/Files* 中) 创建或已经创建，因此我们使用指令 *tester_file* 将其连接到 MQL 程序。

```
#property tester_file "xyz.cal"
```

该指令确保将缓存传输到任何代理，包括分布式代理 (然而，这对于 EA 交易而言比指标更相关)。可以使用新的输入变量 *CalendarCacheFile* 创建具有此名称 (或其他名称) 的缓存文件。如果用户将默认名称更改为其他名称，那么要在测试程序中工作，你将需要更正该指令 (需要重新编译!)，或者将文件传输到终端的共享文件夹 (缓存类支持此功能，但 “留作幕后”)，然而，这样的文件不再可供远程代理使用。

```
input string CalendarCacheFile = "xyz.cal";
```

CalendarFilter 对象现在被描述为一个自动指针，因为根据指标运行的位置，它可以使用原始类 *CalendarFilter* 以及派生类 *CalendarFilterCached*。

```
AutoPtr<CalendarFilter> fptr;  
AutoPtr<CalendarCache> cache;
```

在 *OnInit* 的开头，有一个新的片段负责生成缓存和读取它。


```

int OnInit()
{
    cache = new CalendarCache(CalendarCacheFile, true);
    if(cache[].isLoading())
    {
        fptr = new CalendarFilterCached(cache[]);
    }
    else
    {
        if(MQLInfoInteger(MQL_TESTER))
        {
            Print("Can't run in the tester without calendar cache file");
            return INIT_FAILED;
        }
        else
        if(StringLen(CalendarCacheFile))
        {
            Alert("Calendar cache not found, trying to create '" + CalendarCacheFile + "");
            cache = new CalendarCache();
            if(cache[].save(CalendarCacheFile))
            {
                Alert("File saved. Re-run indicator in online chart or in the tester");
            }
            else
            {
                Alert("Error: ", _LastError);
            }
            ChartIndicatorDelete(0, 0, MQLInfoString(MQL_PROGRAM_NAME));
            return INIT_PARAMETERS_INCORRECT;
        }
        Alert("Currently working in online mode (no cache)");
        fptr = new CalendarFilter(Context);
    }
    CalendarFilter *f = fptr[];
    ... // continued without changes
}

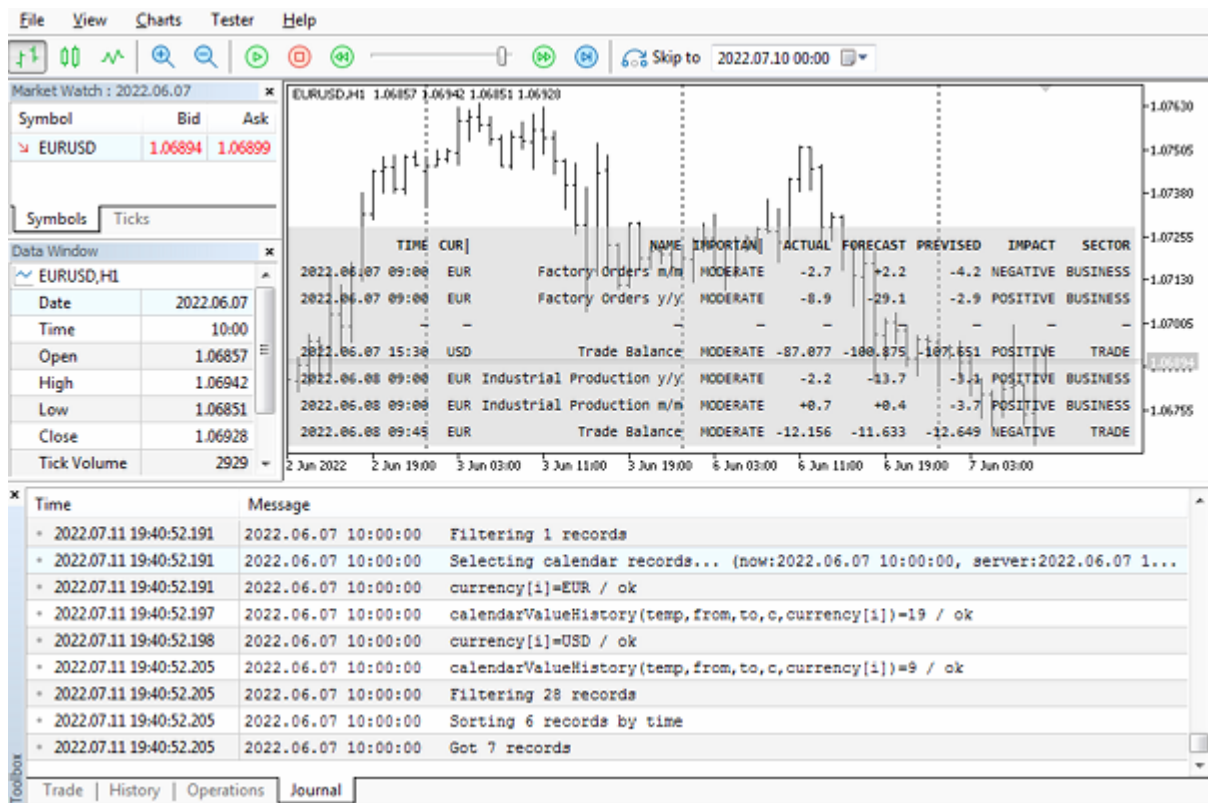
```

如果已读取缓存文件，我们将获得完成的 *CalendarCache* 对象，该对象传递给 *CalendarFilterCached* 构造函数。否则，程序会检查它是在测试程序中运行还是在线运行。测试程序中缺少缓存是致命的情况。在常规图表上，程序会基于内置日历数据创建一个新对象，并将其保存在指定名称的缓存中。但如果文件名为空，指标将完全像原始指标一样工作，即直接与日历交互。

我们在 EURUSD 图表上运行该指标。用户将收到警告，提示未找到指定文件，并已尝试保存它。前提是终端设置中已启用日历，我们应该在日志中得到大致如下的行。下面是带有详细诊断信息的版本。可以通过在源代码中注释掉指令 *#define LOGGING* 来禁用详细信息。

```
Loading calendar cache xyz.cal
FileOpen(filename,FILE_READ|FILE_BIN|flags)=-1 / CANNOT_OPEN_FILE(5004)
Alert: Calendar cache not found, trying to create 'xyz.cal'
Reading online calendar base...
CalendarValueHistory(values,from,to,country,currency)=157173 / ok
CalendarEventByCountry(country,events)=1493 / ok
CalendarCountries(countries)=23 / ok
Hashing calendar...
ArraySize(countries)=23 / ok
ArrayResize(id4country,size2prime(c))=53 / ok
Total collisions: 9, worse:3, average: 2.25 in 4
ArraySize(events)=1493 / ok
ArrayResize(id4event,size2prime(e))=3079 / ok
Total collisions: 495, worse:7, average: 1.43478 in 345
ArraySize(values)=157173 / ok
ArrayResize(id4value,size2prime(v))=393241 / ok
Total collisions: 3511, worse:1, average: 1.0 in 3511
Binding calendar tables...
FileOpen(filename,FILE_WRITE|FILE_BIN|flags)=1 / ok
Alert: File saved. Re-run indicator in online chart or in the tester
```

现在，我们可以在测试程序中选择指标 *CalendarMonitorCached.mq5*，并根据历史数据动态查看新闻表的变化情况。



测试程序中使用日历缓存的新闻指标
日历缓存的存在允许你根据新闻测试交易策略。我们将在下一节中讨论。

7.3.12 日历交易

新闻交易策略多种多样：有使用市价单或挂单的，有分析财经指标（价格运动方向）的，也有不分析指标（捕捉波动性）的。此外，在许多其他交易系统中加入一个反新闻筛选器也很有用。由于 MQL5 日历在测试程序中不可用，因此优化和调试所有此类程序都很困难。然而，借助上一节中开发的缓存，我们可以纠正这种情况。

我们尝试创建一个 EA 交易，它将在新闻发布时，根据对其价格影响的评估进入市场。刚刚使用指标 *CalendarMonitorCached.mq5* 创建了 “xyz.ca1” 缓存文件。

回想一下，缓存中的日历镜像始终对应于保存的时刻，并在读取时需要谨慎：对于较晚的事件，实际指标是未知的，而更久远的事件可能根本不存在。在下次优化或测试之前，你应该定期更新日历缓存文件。

如有必要，还应考虑一年中的 DST 时间设置：如果事件的 DST 模式与保存日历存档时的 DST 不同，你将需要将时间向前或向后调整 1 小时。通过选择没有 DST 的经纪人或在 H1 以上的时间范围上构建策略，可以避免这些困难。

EA 交易 *CalendarTrading.mq5* 将仅交易满足以下条件的新闻事件：

- 与图表的工作交易品种相关
- 类型为财经指标（即量化指标）
- 重要性高
- 刚刚收到指标的当前值

最后一点很重要，因为对于具有预测值和实际值的指标，系统会相应地设置字段 *impact_type* 的值：它将作为交易信号（指示进入市场的方向）。

新闻发布的准确时间通常与 *MqlCalendarValue::time* 字段中输入的计划时间不一致。日历不会记录这个时间，并且在缓存中也不可用。在这方面，新闻策略测试的准确性可能会受到影响。如果你想使分析和决策更接近在线过程，请使用诸如 *CalendarChangeSaver.mq5* 之类的服务积累新闻发布统计数据，并将其嵌入缓存中。

默认情况下，交易以最小手数进行，止盈和止损水平设置在指定的点数距离。所有这些都反映在输入参数中。

```
input double Volume;           // Volume (0 = minimal lot)
input int Distance2SLTP = 500; // Distance to SL/TP in points (0 = no)
input uint MultiplePositions = 25;
```

对于对冲账户，我们允许同时存在多个仓位，默认为 25 个。这是推荐的测试环境，因为它允许你独立评估不同类型新闻并行交易的盈利能力（每个仓位都是独立创建的，不会因其他新闻而导致平仓操作）。另一方面，仅维持一个仓位会自动抵消不同新闻的冲突信号。

可选地，EA 交易支持两种类型的筛选器：新闻类型标识符，按标题搜索文本。

```
sinput ulong EventID;
sinput string Text;
```

这对于未来针对特定新闻的研究可能很有用。

在全局级别，通过对新闻的分析处理和仓位跟踪来描述对象指针。

```

AutoPtr<CalendarFilter> fptr;
AutoPtr<CalendarCache> cache;
AutoPtr<TrailingStop> trailing[];

```

当前工作交易品种的操作模式和货币对存储在相应的变量中。为简化示例，假定其用于外汇（在其他市场上，你将进行单一货币交易，即交易代码的报价币种）。

```

const bool Hedging =
    AccountInfoInteger(ACCOUNT_MARGIN_MODE) == ACCOUNT_MARGIN_MODE_RETAIL_HEDGING;
const string Base = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_BASE);
const string Profit = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_PROFIT);

```

在 *OnInit* 处理程序中，我们加载日历缓存并按照如上所述配置筛选器。在线图表中允许不启用缓存：然后 EA 交易以实战模式工作，直接与日历交互。在测试程序中，缺少缓存文件将阻止 EA 交易启动。

```

int OnInit()
{
    cache = new CalendarCache("xyz.cal", true);
    if(cache[].isLoading())
    {
        fptr = new CalendarFilterCached(cache[]);
    }
    else
    {
        if(!MQLInfoInteger(MQL_TESTER))
        {
            Print("Calendar cache file not found, fall back to online mode");
            fptr = new CalendarFilter();
        }
        else
        {
            Print("Can't proceed in the tester without calendar cache file");
            return INIT_FAILED;
        }
    }
    CalendarFilter *f = fptr[];

    if(!f.isLoading()) return INIT_FAILED;

    // if a specific type of event is set, we look only at it
    if(EventID > 0) f.let(EventID);
    else
    {
        // otherwise follow the news on the currencies of the current symbol
        f.let(Base);
        if(Base != Profit)
        {
            f.let(Profit);
        }

        // financial indicators, high importance, actual value
        f.let(CALENDAR_TYPE_INDICATOR);
        f.let(LONG_MIN, CALENDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL);
        f.let(CALENDAR_IMPORTANCE_HIGH);

        if(StringLen(Text)) f.let(Text);
    }

    f.describe();

    if(Distance2SLTP)
    {
        ArrayResize(trailing, Hedging && MultiplePositions ? MultiplePositions : 1);
    }
    // check the news filter and start trading on it by a second timer
    EventSetTimer(1);
}

```

```

    return INIT_SUCCEEDED;
}

```

在 *OnTimer* 处理程序中，我们根据配置的筛选器请求新闻的变更。

```

void OnTimer()
{
    CalendarFilter *f = fptr[];
    MqlCalendarValue records[];

    f.let(TimeTradeServer() - SCOPE_DAY, TimeTradeServer() + SCOPE_DAY);

    if(f.update(records)) // find changes that undergo filtering
    {
        // output properties of changed news to the log
        static const ENUM_CALENDAR_PROPERTY props[] =
        {
            CALENDAR_PROPERTY_RECORD_TIME,
            CALENDAR_PROPERTY_COUNTRY_CURRENCY,
            CALENDAR_PROPERTY_COUNTRY_CODE,
            CALENDAR_PROPERTY_EVENT_NAME,
            CALENDAR_PROPERTY_EVENT_IMPORTANCE,
            CALENDAR_PROPERTY_RECORD_ACTUAL,
            CALENDAR_PROPERTY_RECORD_FORECAST,
            CALENDAR_PROPERTY_RECORD_PREVISED,
            CALENDAR_PROPERTY_RECORD_IMPACT,
        };
        static const int p = ArraySize(props);
        string result[];
        f.format(records, props, result);
        for(int i = 0; i < ArraySize(result) / p; ++i)
        {
            Print(SubArrayCombine(result, " | ", i * p, p));
        }
        ...
    }
}

```

当检测到合适的变更时，它们将按以下方式记录（下方是真实日志的片段），指明时间、货币、国家/地区、名称、当前值和预测值、前次值以及信号的理论解释：

```

...
Filtering 5 records
2021.02.16 13:00 | EUR | EU | Employment Change q/q | HIGH | +0.3 | -0.4 | +1.0 | POS
2021.02.16 13:00 | EUR | EU | GDP q/q | HIGH | -0.6 | -0.7 | -0.7 | POSITIVE
instant buy 0.01 EURUSD at 1.21638 sl: 1.21138 tp: 1.22138 (1.21637 / 1.21638 / 1.216
deal #64 buy 0.01 EURUSD at 1.21638 done (based on order #64)
...
Filtering 3 records
2021.07.06 12:05 | EUR | DE | ZEW Economic Sentiment Indicator | HIGH | +63.3 | +84.1
instant sell 0.01 EURUSD at 1.18473 sl: 1.18973 tp: 1.17973 (1.18473 / 1.18474 / 1.18
deal #265 sell 0.01 EURUSD at 1.18473 done (based on order #265)
...

```

新闻对价格的潜在影响应基于字段内评估 *impact_type* 计算。这里需要注意的是，我们有两种货币：基础货币和报价货币。当新闻对基础货币产生积极影响时，汇率预计会上涨；如果是消极

影响，汇率会下跌。对于报价货币，情况正好相反：积极影响应导致货币对中第二种货币的价格上涨，这意味着汇率下降，而消极影响则导致汇率上涨。这种标准化的价格运动方向在以下片段中使用 *sign* 变量计算。

```
static const int impacts[3] = {0, +1, -1};
int impact = 0;
string about = "";
ulong lasteventid = 0;
for(int i = 0; i < ArraySize(records); ++i)
{
    int sign = result[i * p + 1] == Profit ? -1 : +1;
    impact += sign * impacts[records[i].impact_type];
    about += StringFormat("%+lld ", sign * (long)records[i].event_id);
    lasteventid = records[i].event_id;
}

if(impact == 0) return; // no signal
...
```

通常会有几条新闻同时发布，因此有必要累积所有这些新闻的评级。这一操作在 *impact* 变量中完成。由于我们的策略只筛选单一的、最高重要性的新闻，因此来自它们的所有单一信号都简单地相加，没有权重系数。*about* 字符串变量用于准备即将进行的交易的注释文本：将在该变量中引用引发交易的事件标识符。

如果机器人在净额结算账户上启动，或者已达到允许的最大仓位数量，我们将平掉一个仓位。

```
PositionFilter positions;
ulong tickets[];
positions.let(POSITION_SYMBOL, _Symbol).select(tickets);
const int n = ArraySize(tickets);

if(n >= (int)(Hedging ? MultiplePositions : 1))
{
    MqlTradeRequestSync position;
    position.close(_Symbol) && position.completed();
}
...
```

现在你可以根据信号开设新仓位。事件标识符被设置为“魔术”数字，这将使我们以后能够分析不同类型新闻背景下交易的财务表现。

```

MqlTradeRequestSync request;
request.magic = lasteventid;
request.comment = about;
const double ask = SymbolInfoDouble(_Symbol, SYMBOL_ASK);
const double bid = SymbolInfoDouble(_Symbol, SYMBOL_BID);
const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
ulong ticket = 0;

if(impact > 0)
{
    ticket = request.buy(Lot, 0,
        Distance2SLTP ? ask - point * Distance2SLTP : 0,
        Distance2SLTP ? ask + point * Distance2SLTP : 0);
}
else if(impact < 0)
{
    ticket = request.sell(Lot, 0,
        Distance2SLTP ? bid + point * Distance2SLTP : 0,
        Distance2SLTP ? bid - point * Distance2SLTP : 0);
}

if(ticket && request.completed() && Distance2SLTP)
{
    for(int i = 0; i < ArraySize(trailing); ++i)
    {
        if(trailing[i][] == NULL) // looking for a free slot for the position tra
        {
            trailing[i] = new TrailingStop(ticket, Distance2SLTP, Distance2SLTP /
                break;
        }
    }
}
}
}

```

我们在每个分时报价到来时移动所有仓位的止损。


```

void OnTick()
{
    for(int i = 0; i < ArraySize(trailing); ++i)
    {
        if(trailing[i][])
        {
            if(!trailing[i][].trail()) // position was closed
            {
                trailing[i] = NULL; // release object and slot
            }
        }
    }
}

```

现在到了最有趣的部分。多亏了测试程序，不仅可以总体分析新闻策略的成功与否，还可以按特定新闻进行细分分析。相应的功能块在我们的 *OnTester* 处理程序中实现。数据收集是使用交易筛选器执行的。通过交易筛选器可以获得元组数组 *trades*，其中报告了每笔交易的利润、隔夜利息、佣金和魔术数字，此后，我们在三个 *MapArray* 对象中累积结果：它们分别为每个 *magic* 数字单独计算利润、亏损和交易数量。

```

double OnTester()
{
    Print("Trade profits by calendar events:");
    HistorySelect(0, LONG_MAX);
    DealFilter filter;
    int props[] = {DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_MAGIC};
    filter.let(DEAL_TYPE, (1 << DEAL_TYPE_BUY) | (1 << DEAL_TYPE_SELL), IS::OR_BITWISE
        .let(DEAL_ENTRY, (1 << DEAL_ENTRY_OUT) | (1 << DEAL_ENTRY_INOUT) | (1 << DEAL_E
            IS::OR_BITWISE);
    Tuple4<double, double, double, ulong> trades[];
    MapArray<ulong, double> profits;
    MapArray<ulong, double> losses;
    MapArray<ulong, int> counts;
    if(filter.select(props, trades))
    {
        for(int i = 0; i < ArraySize(trades); ++i)
        {
            counts.inc((ulong)trades[i]._4);
            const double payout = trades[i]._1 + trades[i]._2 + trades[i]._3;
            if(payout >= 0)
            {
                profits.inc((ulong)trades[i]._4, payout);
                losses.inc((ulong)trades[i]._4, 0);
            }
            else
            {
                profits.inc((ulong)trades[i]._4, 0);
                losses.inc((ulong)trades[i]._4, payout);
            }
        }
    }
    ...
}

```

结果，我们得到一个表格，逐行显示每种事件类型的统计数据：其标识符、国家/地区、货币、总利润或亏损、交易数量（新闻数量）、盈利因子和事件名称。

```

for(int i = 0; i < profits.getSize(); ++i)
{
    MqlCalendarEvent event;
    MqlCalendarCountry country;
    const ulong keyId = profits.getKey(i);
    if(cache[].calendarEventById(keyId, event)
        && cache[].calendarCountryById(event.country_id, country))
    {
        PrintFormat("%lld %s %s %+.2f [%d] (PF:%.2f) %s",
            event.id, country.code, country.currency,
            profits[keyId] + losses[keyId], counts[keyId],
            profits[keyId] / (losses[keyId] != 0 ? -losses[keyId] : DBL_MIN),
            event.name);
    }
    else
    {
        Print("undefined ", DoubleToString(profits.getValue(i), 2));
    }
}
}
return 0;
}

```

为了测试这个想法，我们在 EURUSD 货币对上运行 EA 交易，时间段从 2021 年初到 2022 年中。以下是从 *OnTester* 输出的日志片段。

```

Trade profits by calendar events:
840040001 US USD -21.81 [17] (PF:0.53) ISM Manufacturing PMI
840190001 US USD -10.95 [17] (PF:0.69) ADP Nonfarm Employment Change
840200001 US USD -67.09 [78] (PF:0.60) EIA Crude Oil Stocks Change
999030003 EU EUR +14.13 [19] (PF:1.46) Retail Sales m/m
840040003 US USD -17.12 [18] (PF:0.59) ISM Non-Manufacturing PMI
840030016 US USD -1.20 [19] (PF:0.97) Nonfarm Payrolls
840030021 US USD +5.25 [14] (PF:1.21) JOLTS Job Openings
840020010 US USD -14.63 [17] (PF:0.63) Retail Sales m/m
276070001 DE EUR -22.71 [17] (PF:0.47) ZEW Economic Sentiment Indicator
840020005 US USD +10.76 [18] (PF:1.37) Building Permits
840120001 US USD -20.78 [17] (PF:0.49) Existing Home Sales
276030003 DE EUR +18.57 [17] (PF:1.87) Ifo Business Climate
840180002 US USD -3.22 [14] (PF:0.89) CB Consumer Confidence Index
840020014 US USD -8.74 [16] (PF:0.74) Core Durable Goods Orders m/m
840020008 US USD -14.54 [16] (PF:0.63) New Home Sales
250010005 FR EUR +0.66 [10] (PF:1.03) GDP q/q
840010007 US USD +0.99 [15] (PF:1.04) GDP q/q
840120003 US USD +4.53 [18] (PF:1.15) Pending Home Sales m/m
276010008 DE EUR -0.72 [10] (PF:0.97) GDP q/q
999030016 EU EUR -14.04 [14] (PF:0.59) GDP q/q
999030001 EU EUR +1.30 [2] (PF:1.35) Employment Change q/q

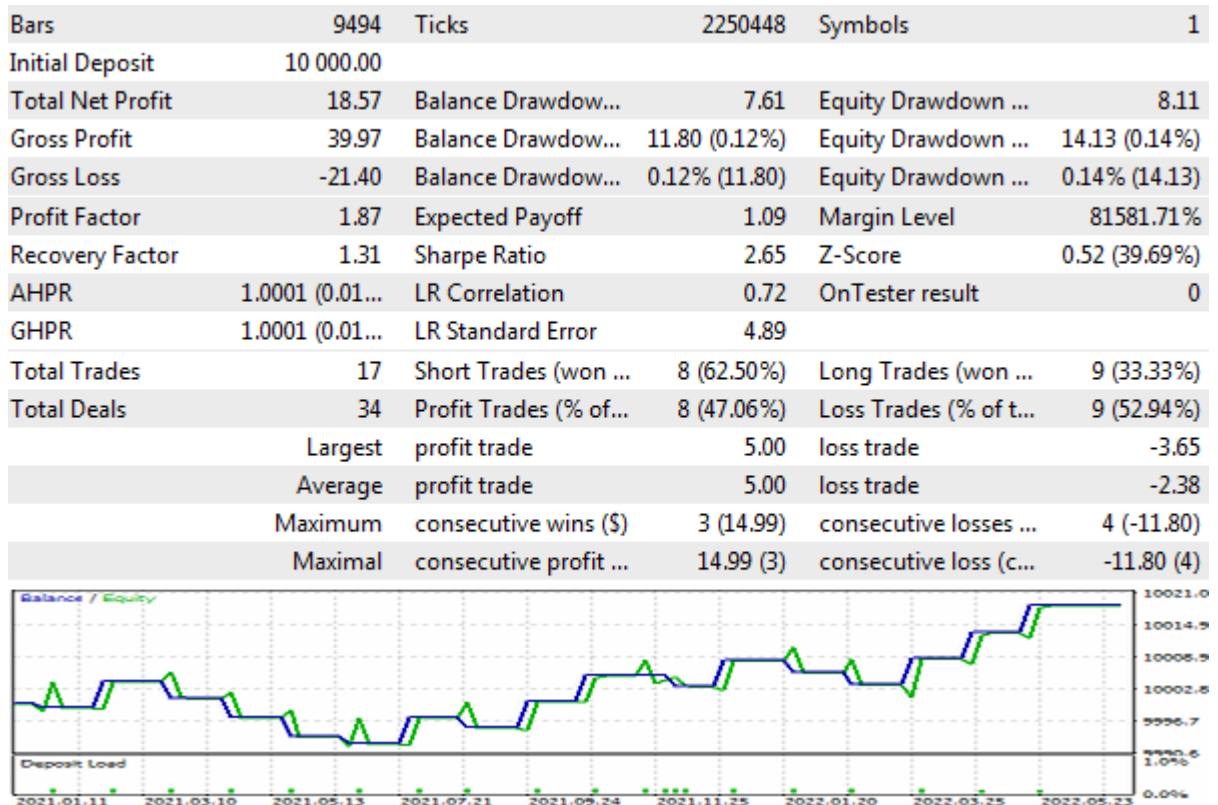
```

结果不尽如人意。尽管如此，新闻交易充满了主观性。首先，新闻实际价值对走势影响的理论评估可能与市场群体的情绪预期背离，也可能与未纳入经济日历的其他信息背景（无法量化）

不符。其次，我们已经提到了实际价值发布时间的不准确性。第三，我们的策略以最简单的形式实现，没有分析初步的价格运动（可能新闻已经泄露并且市场已经提前消化了该新闻）。

总的来说，这次测试发现，交易者最喜欢的非农就业报告或 GDP 报告并不能保证成功，至少在我们的默认设置难以确保获利。接下来，需要以常规方式分析单个交易，找出问题所在，选择参数并改进算法，特别是添加一个用于服务器时区夏令时切换的时间调整模块。

同时，这种技术本身运作良好，我们可以先尝试选择最成功的新闻。例如，我们选择新闻 276030003 (Ifo 商业景气指数)。通过将其设置到 *EventID* 中，我们将收到以下报告，与我们计算的指标相吻合。



基于 Ifo 商业景气指数新闻在测试程序中交易的报告

你还可以尝试针对一组类似的事件进行交易。特别是，要仅对 GDP 新闻（不同国家的）做出反应，请在 *Text* 变量中输入字符串 “*GDP”。添加星号是因为，如果没有星号，一个 3 字符的字符串将被筛选器类视作货币。任何长度不是 2（国家代码）或 3（货币代码）的字符串都可以按原样指定，例如，“fam”、“Nonfam”、“Sales”，筛选器将它们作为名称的子字符串进行搜索，区分大小写。

7.4 密码学

算法交易出现在交易所交易和信息技术的交叉点，一方面允许连接越来越多的新市场进行工作，另一方面也扩展了交易平台的功能。密码学，或者更广泛地说，信息安全，是已经渗透到包括交易者工具箱在内的大多数活动领域的一项技术趋势。

MQ5 提供了用于加密、压缩数据和对数据进行哈希计算的函数：*CryptEncode* 和 *CryptDecode*。我们已经在本书的一些示例中使用过它们：在 *EnvSignature.mq5* 脚本（将程序绑定到运行时属性）和 *ServiceAccount.mq5* 服务（服务）。

在本章中，我们将更详细地讨论这些函数。然而，在直接描述它们之前，我们回顾一下信息转换方法：这个编程方向非常广泛，而 MQL5 仅支持标准的一部分。这个列表将来可能会扩展，但目前，如果你在帮助中找不到所需的加密方法，请尝试在 mq5.com 网站上（在文章部分或源代码数据库中）查找现成的实现。

7.4.1 可用信息转换方法概述

信息保护可以为不同的目的而实施，因此使用不同的方法。特别地，可能需要向外部观察者完全隐藏信息的本质，或者确保其传输，同时保证其状态不变，但信息本身仍然可用。第一种情况，我们谈论的是加密；第二种情况指的是数字指纹（哈希）。因此，加密和哈希都归结为使用原始数据（如果需要，还包括附加参数）处理成新的表示形式。

加密和哈希都有许多变种。

最普遍的分类将加密分为公钥（非对称）加密和私钥（对称）加密。

非对称方案意味着数据交换的每个参与者都有一对密钥—公钥和私钥。公钥和私钥对是使用特殊算法预先生成的。每个私钥只有其所有者知道。每个人的公钥则为众人所知。在传输加密数据之前，需要以某种方式交换公钥。接下来，数据提供者使用只有他们自己知道的私钥，并结合一个或多个数据接收者的公钥。而接收者则使用他们自己的私钥和发送者的公钥来解密。

对称加密方案对加密和解密使用相同的秘密（私钥）密钥。

MQL5 支持开箱即用的私钥功能（对称加密）。MQL5 内置工具不提供使用非对称加密的电子签名功能。

在加密方法中，还有一类比较特别的是无密钥的原始算法。借助它们，用户可以实现有条件地隐藏信息或转换信息类型。这些包括，例如，ROT13（将其字母数字代码移位 13 来替换字符，特别用于 Windows 注册表）或 Base64（将二进制文件转换为文本，反之亦然，通常用于 Web 项目）。另一个流行的数据转换任务是数据压缩。从某种意义上说，这也可以被认为是加密，因为数据变得对人或应用程序不可读。

哈希方法也有很多。最著名且最简单的可能是 CRC（循环冗余校验）。与允许从加密消息中恢复原始消息的加密不同，哈希仅基于原始信息创建一个指纹（一组特征字节），这样，在后续重新计算时其不变状态（高概率地）保证了原始信息的不变性。当然，这假设信息对相应软件系统的所有参与者/用户都是可用的。无法通过哈希恢复信息。通常，哈希的大小（其中的字节数）对于每种方法都是有限且标准化的，因此对于一个 80 个字符长的字符串和一个 1 MB 的文件，我们将得到相同大小的哈希。大多数用户会发现哈希的一个真正有用的应用是网站和程序对密码进行哈希处理，即后者以哈希形式存储在家中，并在登录时与密码哈希进行验证，而不是原始形式的密码。

应该指出的是，我们在前一章中已经遇到过“哈希”这个术语：我们使用了一个哈希函数来索引经济日历结构。那个简单的哈希保护程度非常弱，特别表现在碰撞（不同数据的结果相同）的概率很高，我们在算法中对此进行了特殊处理。它适用于将数据均匀伪随机分布到有限数量的“桶”中的问题。相比之下，工业标准的哈希专门用于验证信息的完整性，并使用复杂得多的计算方法。但在这种情况下，哈希的长度是几十个字节，而不是一个单独的数字。

MQL 程序可用的信息加密和哈希方法收集在 `ENUM_CRYPT_METHOD` 枚举中。

常量	说明
CRYPT_BASE64	Base64 重编码
CRYPT_DES	使用 56 位 (7 字节) 密钥的 DES 加密
CRYPT_AES128	使用 128 位 (16 字节) 密钥的 AES 加密
CRYPT_AES256	使用 256 位 (32 字节) 密钥的 AES 加密
CRYPT_HASH_MD5	MD5 哈希计算 (16 字节)
CRYPT_HASH_SHA1	SHA1 哈希计算 (20 字节)
CRYPT_HASH_SHA256	SHA256 哈希计算 (32 字节)
CRYPT_ARCH_ZIP	使用 “deflate” 方法进行压缩

指定的枚举用于两个加密 API 函数中—*CryptEncode* (加密/哈希) 和 *CryptDecode* (解密)。这些内容将在后续各节中介绍。

AES 和 DES 加密方法除了数据外，还需要加密密钥——一个预定义长度的字节数组 (在表格的括号中指出)。如前所述，密钥必须保密，并且只有程序的开发者或信息的所有者知道。加密的密码强度，即攻击者计算机破解密钥的难度，直接取决于密钥的大小：密钥越大，保护越可靠。因此，DES 被认为是过时的，并在金融领域被其改进版本 Triple DES 所取代：它意味着使用三个不同的密钥连续应用 DES 三次，这在 MQL5 中很容易实现。Triple DES 有一个流行的版本，在第二次迭代中执行解密而不是使用密钥 2 进行加密，也就是说，在最终的第三轮 DES 之前，它将数据恢复到一个中间的、故意不正确的表示。但是 Triple DES 也计划在 2024 年后从行业标准中移除。

同时，密码强度应与秘密 (密钥和信息) 的生命周期相称。如果需要快速的安全消息流，定期更新的较短密钥将提供更好的性能。

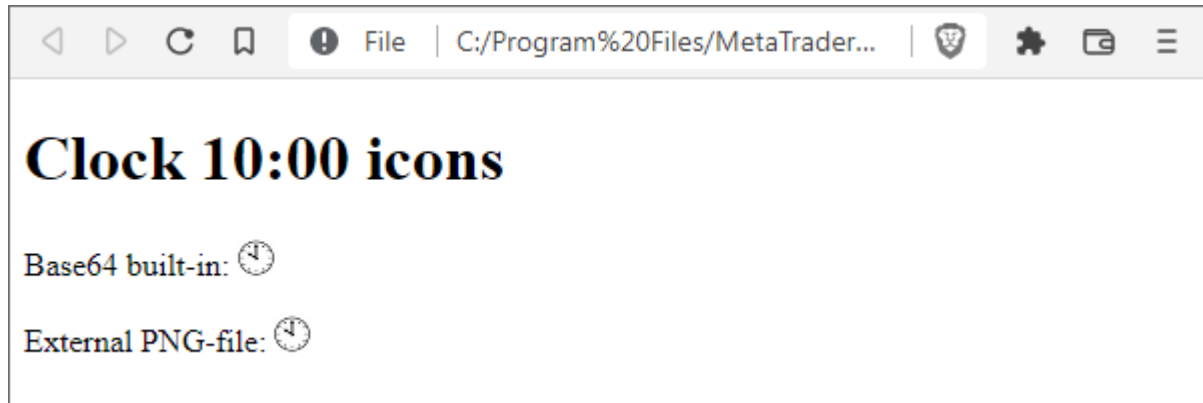
在哈希方法中，最现代的是 SHA256 (SHA-2 标准的一个子集)。SHA1 和 MD5 方法被认为不安全，但为了与现有服务兼容仍被广泛使用。对于哈希方法，括号中指出了包含数据数字指纹的结果字节数组的大小。哈希不需要密钥，但在许多应用中，会将“盐”(salt) 附加到哈希数据上——这是一种秘密组件，使攻击者难以重现所需的哈希 (例如，在猜测密码时)。

CRYPT_ARCH_ZIP 元素提供 ZIP 归档以及互联网上的数据请求传输/接收 (参见 [WebRequest](#))。

尽管该方法的名称中包含 ZIP，但压缩数据不等同于普通的 ZIP 存档，后者除了 “deflate” 容器外，总是包含元数据：特殊的头部、文件列表及其属性。在 [mq5.com](#) 网站上，文章和源代码库中，你可以找到将文件压缩到 ZIP 存档并从中提取的现成实现。压缩和提取由 *CryptEncode/CryptDecode* 函数执行，所有其他必要的 ZIP 格式结构都在 MQL5 代码中描述和填充。

Base64 方法旨在将二进制数据转换为文本，反之亦然。二进制数据通常包含许多不可打印的字符，并且不受编辑和输入工具 (例如 MQL 程序属性对话框中的输入变量) 的支持。例如，在处理流行的 JSON 对象数据交换文本格式时，Base64 可能很有用。

每 3 个原始字节在 Base64 中编码为 4 个字符，导致数据大小增加三分之一。本书附带了我们将在以下示例中进行实验的测试文件，特别是网页 [MQL5/Files/MQL5Book/clock10.htm](#) 和其中使用的时钟图像文件 [MQL5/Files/MQL5Book/clock10.png](#)。在这个介绍阶段，你已经可以清楚地看到二进制数据和 Base64 文本在内部表示上的可能性和差异，同时保持外观相同。



嵌入二进制图像和 Base64 格式的网页

相同的钟面图像作为外部文件 *clock10.png* 插入到页面中，同时其 Base64 编码也包含在 *img* 标签中（在其 *src* 属性中：这就是“数据 URL”）。直接在网页本身的文本中，它看起来是这样的（没有必要将长的 Base64 字符串按 76 个字符的宽度换行，但标准允许这样做，并且这里为了出版方便就这样做了）：

```

```

很快我们将使用 *CryptEncode* 函数重现这个字符序列，但现在，只需注意使用类似的技术，我们可以从 MQL5 生成带有嵌入式图形的 HTML 报告。

7.4.2 加密、哈希和数据打包：CryptEncode

MQL5 中负责数据加密、哈希和压缩的函数是 *CryptEncode*。它通过指定的方法将所传递源数组 *data* 的数据转换到目标数组 *result* 中。

```
int CryptEncode(ENUM_CRYPT_METHOD method, const uchar &data[], const uchar &key[], uchar
&result[])
```

加密方法还需要传递一个带有私有（秘密）密钥的字节数组 *key*：其长度取决于具体方法，并在上一节的 ENUM_CRYPT_METHOD 方法表格中指定。如果 *key* 数组的大小更大，则仍将仅使用所需数量的第一个字节作为密钥。

哈希或压缩不需要密钥，但对于 CRYPT_ARCH_ZIP 有一个注意事项。事实是，终端内置的“deflate”算法的实现在结果数据中添加了几个字节以控制完整性：最初的 2 个字节包含“deflate”算法的设置，末尾的 4 个字节包含 Adler32 校验和。由于此特性，生成的打包容器与 ZIP 存档为存档的每个单独元素生成的容器不同（ZIP 标准在其头部存储含义相似的 CRC32）。因此，为了能够基于 *CryptEncode* 函数打包的数据创建和读取兼容的 ZIP 存档，MQL5 允许你使用 *key* 数组中的特殊值来禁用其自身的完整性检查和额外字节的生成。

```
uchar key[] = {1, 0, 0, 0};
CryptEncode(CRYPT_ARCH_ZIP, data, key, result);
```

可以使用任何长度至少为 4 字节的密钥。获得的 *result* 数组可以根据标准 ZIP 格式添加标题 (这个问题超出了本书的范围), 以创建其他程序可以访问的存档。

函数返回放置在目标数组中的字节数, 如果出错则返回 0。错误代码将照常存储在 *_LastError* 中。

我们使用 *CryptEncode.mq5* 脚本来检查函数性能。它允许用户输入文本 (*Text*) 或指定一个文件 (*File*) 进行处理。要使用文件, 你需要清除 *Text* 字段。

你可以选择一个特定的 *Method*, 或者一次性遍历所有方法, 以直观地查看和比较不同的结果。对于这样的审查循环, 请在 *Method* 参数中保留默认值 *_CRYPT_ALL*。

顺便说一句, 为了引入这样的功能, 我们再次需要扩展标准枚举 (这次是 *ENUM_CRYPT_METHOD*), 但由于 MQL5 中的枚举不能像类那样被继承, 这里实际上声明了一个新的枚举 *ENUM_CRYPT_METHOD_EXT*。这样做的一个额外好处是, 我们为元素添加了更友好的名称 (在注释中, 带有将在设置对话框中显示的提示)。

```
enum ENUM_CRYPT_METHOD_EXT
{
    _CRYPT_ALL = 0xFF,                // Try All in a Loop
    _CRYPT_DES = CRYPT_DES,           // DES    (key required, 7 bytes)
    _CRYPT_AES128 = CRYPT_AES128,     // AES128 (key required, 16 bytes)
    _CRYPT_AES256 = CRYPT_AES256,     // AES256 (key required, 32 bytes)
    _CRYPT_HASH_MD5 = CRYPT_HASH_MD5, // MD5
    _CRYPT_HASH_SHA1 = CRYPT_HASH_SHA1, // SHA1
    _CRYPT_HASH_SHA256 = CRYPT_HASH_SHA256, // SHA256
    _CRYPT_ARCH_ZIP = CRYPT_ARCH_ZIP, // ZIP
    _CRYPT_BASE64 = CRYPT_BASE64,     // BASE64
};

input string Text = "Let's encrypt this message"; // Text (empty to process File)
input string File = "MQL5Book/clock10.htm"; // File (used only if Text is empty)
input ENUM_CRYPT_METHOD_EXT Method = _CRYPT_ALL;
```

默认情况下, *Text* 参数填充了要加密的消息。你可以将其替换为自己的内容。如果我们清除 *Text*, 程序将处理文件。至少有一个参数 (*Text* 或 *File*) 应包含信息。

由于加密需要密钥, 其他两个选项允许你直接以文本形式输入它 (尽管密钥不必是文本, 并且可以包含任何二进制数据, 但输入中不支持这些数据), 或根据加密方法生成所需的长度。


```
enum DUMMY_KEY_LENGTH
{
    DUMMY_KEY_0 = 0,    // 0 bytes (no key)
    DUMMY_KEY_7 = 7,    // 7 bytes (sufficient for DES)
    DUMMY_KEY_16 = 16,  // 16 bytes (sufficient for AES128)
    DUMMY_KEY_32 = 32,  // 32 bytes (sufficient for AES256)
    DUMMY_KEY_CUSTOM,   // use CustomKey
};

input DUMMY_KEY_LENGTH GenerateKey = DUMMY_KEY_CUSTOM; // GenerateKey (length, or frc
input string CustomKey = "My top secret key is very strong";
```

最后，有一个选项 *DisableCRCinZIP* 用于启用 ZIP 兼容模式，这仅影响 *CRYPT_ARCH_ZIP* 方法。

```
input bool DisableCRCinZIP = false;
```

为了简化检查方法是否需要加密密钥或是否计算哈希（不可逆的单向转换），定义了 2 个宏。

```
#define KEY_REQUIRED(C) ((C) == CRYPT_DES || (C) == CRYPT_AES128 || (C) == CRYPT_AES256)
#define IS_HASH(C) ((C) == CRYPT_HASH_MD5 || (C) == CRYPT_HASH_SHA1 || (C) == CRYPT_HASH_SH
```

OnStart 的开头部分包含了所需变量和数组的描述。

```
void OnStart()
{
    ENUM_CRYPT_METHOD method = 0;
    int methods[];           // here we will collect all the elements of ENUM_CRYPT_ME
    uchar key[] = {};        // empty by default: suitable for hashing, zip, base64
    uchar zip[], opt[] = {1, 0, 0, 0}; // "options" for zip
    uchar data[], result[];  // initial data and result
```

根据 *GenerateKey* 设置，我们从 *CustomKey* 字段获取密钥，或者仅仅用单调递增的整数值填充 *key* 数组。实际上，密钥应该是一个秘密的、非平凡的、任意选择的值块。

```
if(GenerateKey == DUMMY_KEY_CUSTOM)
{
    if(StringLen(CustomKey))
    {
        PRTF(CustomKey);
        StringToCharArray(CustomKey, key, 0, -1, CP_UTF8);
        ArrayResize(key, ArraySize(key) - 1);
    }
}
else if(GenerateKey != DUMMY_KEY_0)
{
    ArrayResize(key, GenerateKey);
    for(int i = 0; i < GenerateKey; ++i) key[i] = (uchar)i;
}
```

此处及下文，请注意在 *StringToCharArray* 之后使用 *ArrayResize*。务必将数组大小减少 1 个元素，因为如果 *StringToCharArray* 函数将字符串转换为字节数组时包含了末尾的 0，这可能会破坏程序的预期执行。特别是在这种情况下，我们的秘密密钥中会有一个额外的零字节，如果在接收端没有使用带有类似人为处理的程序，那么它将无法解密消息。这种额外的零也可能影响与数据交换协议的兼容性（如果 MQL 程序与“外部世界”进行某种集成）。

接下来，我们以十六进制格式记录下最终密钥的原始表示：这是由 `ByteArrayPrint` 函数完成的，该函数曾在 [以简化模式读写文件](#) 一节中使用过。

```
if(ArraySize(key))
{
    Print("Key (bytes):");
    ByteArrayPrint(key);
}
else
{
    Print("Key is not provided");
}
```

根据 `Text` 或 `File` 是否可用，我们用文本字符或文件内容填充 `data` 数组。

```
if(StringLen(Text))
{
    PRTF(Text);
    PRTF(StringToCharArray(Text, data, 0, -1, CP_UTF8));
    ArrayResize(data, ArraySize(data) - 1);
}
else if(StringLen(File))
{
    PRTF(File);
    if(PRTF(FileLoad(File, data)) <= 0)
    {
        return; // error
    }
}
```

最后，我们遍历所有方法或使用特定方法执行一次转换。

```

const int n = (Method == _CRYPT_ALL) ?
    EnumToArray(method, methods, 0, UCHAR_MAX) : 1;
ResetLastError();
for(int i = 0; i < n; ++i)
{
    method = (ENUM_CRYPT_METHOD)((Method == _CRYPT_ALL) ? methods[i] : Method);
    Print("- ", i, " ", EnumToString(method), ", key required: ",
        KEY_REQUIRED(method));

    if(method == CRYPT_ARCH_ZIP)
    {
        if(DisableCRCinZIP)
        {
            ArrayCopy(zip, opt); // array with additional option dynamic for ArraySwa
        }
        ArraySwap(key, zip); // change key to empty or option
    }

    if(PRTF(CryptEncode(method, data, key, result)))
    {
        if(StringLen(Text))
        {
            // code page Latin (Western) to unify the display for all users
            Print(CharArrayToString(result, 0, WHOLE_ARRAY, 1252));
            ByteArrayPrint(result);
            if(method != CRYPT_BASE64)
            {
                const uchar dummy[] = {};
                uchar readable[];
                if(PRTF(CryptEncode(CRYPT_BASE64, result, dummy, readable)))
                {
                    PrintFormat("Try to decode this with CryptDecode.mq5 (%s):",
                        EnumToString(method));
                    // to receive encoded data back for decoding
                    // via string input, apply Base64 over binary result
                    Print("base64:" + CharArrayToString(readable, 0, WHOLE_ARRAY, 1252)
                }
            }
        }
    }
    else
    {
        string parts[];
        const string filename = File + "." +
            parts[StringSplit(EnumToString(method), '_', parts) - 1];
        if(PRTF(FileSave(filename, result)))
        {
            Print("File saved: ", filename);
            if(IS_HASH(method))
            {
                ByteArrayPrint(result, 1000, "");
            }
        }
    }
}

```

```

    }
  }
}
}
}

```

当我们转换文本时，我们会记录结果，但由于除了 CRYPT_BASE64 方法外，结果几乎总是二进制数据，它们的显示将完全是乱码（说实话，二进制数据不应该被记录，但我们这样做是为了清晰起见）。不可打印的符号和代码大于 128 的符号在不同语言的计算机上显示不同。因此，为了统一所有读者的示例显示，在 **CharArrayToString** 中形成行时，我们使用显式代码页（252，西欧语言）。诚然，出版书籍时使用的字体很可能会影响某些字符的显示方式（字体中的字形集可能有限）。

务必要注意，我们仅在显示方法中控制代码页的选择，而 **result** 数组中的字节不会因此改变（当然，以这种方式获得的字符串不应再发送到任何地方；它仅用于可视化，数据交换应使用 result 本身的字节）。

然而，我们仍然希望为用户提供一些保存加密结果以便以后解码的机会。最简单的方法是使用 CRYPT_BASE64 方法重新转换二进制数据。

关于文件编码，我们只需将结果保存在一个新文件中，其名称是在原始文件名上添加方法名称中最后一个单词的扩展名。例如，将 CRYPT_HASH_MD5 应用于文件 **Example.txt**，我们将得到输出文件 **Example.txt.MD5**，其中包含源文件的 MD5 哈希。请注意，对于 CRYPT_ARCH_ZIP 方法，我们将得到一个带有 ZIP 扩展名的文件，但它不是标准的 ZIP 存档（由于缺少包含元信息的头部和目录表）。

我们使用默认设置运行该脚本：它们对应于在循环中检查文本 “Let’s encrypt this message” 的所有方法。

```

CustomKey=My top secret key is very strong / ok
Key (bytes):
[00] 4D | 79 | 20 | 74 | 6F | 70 | 20 | 73 | 65 | 63 | 72 | 65 | 74 | 20 | 6B | 65 |
[16] 79 | 20 | 69 | 73 | 20 | 76 | 65 | 72 | 79 | 20 | 73 | 74 | 72 | 6F | 6E | 67 |
Text=Let's encrypt this message / ok
StringToCharArray(Text,data,0,-1,CP_UTF8)=26 / ok
- 0 CRYPT_BASE64, key required: false
CryptEncode(method,data,key,result)=36 / ok
TGV0J3MgZW5jcnlwdCB0aGlzIG1lc3NhZ2U=
[00] 54 | 47 | 56 | 30 | 4A | 33 | 4D | 67 | 5A | 57 | 35 | 6A | 63 | 6E | 6C | 77 |
[16] 64 | 43 | 42 | 30 | 61 | 47 | 6C | 7A | 49 | 47 | 31 | 6C | 63 | 33 | 4E | 68 |
[32] 5A | 32 | 55 | 3D |
- 1 CRYPT_AES128, key required: true
CryptEncode(method,data,key,result)=32 / ok
`T* Ě[3hß Ã/-C }-ŠÑØN""*Ê† ‡Ñ
[00] 01 | 0B | AF | 54 | 2A | 12 | CB | 5B | 33 | 68 | DF | 0E | C3 | 2F | 2D | 43 |
[16] 19 | 7D | AC | 8A | D1 | 8F | D8 | 4E | A8 | AE | CA | 81 | 86 | 06 | 87 | D1 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_AES128):
base64:'AQuvVCoSy1szaN8Owy8tQxl9rIrRj9hOqK7KgYYGh9E='
- 2 CRYPT_AES256, key required: true
CryptEncode(method,data,key,result)=32 / ok
ø'UL»ĖšëDC%ô ñ.K)ĖýÁ Lá, +< !Dï
[00] F8 | 91 | 55 | 4C | BB | C9 | 73 | EB | 44 | 43 | 89 | F4 | 06 | 13 | AC | 2E |
[16] 4B | 29 | 8C | FD | C1 | 11 | 4C | E1 | B8 | 05 | 2B | 3C | 14 | 21 | 44 | EF |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_AES256):
base64:'+JFVTLvJc+tEQ4n0Bh0sLkspjP3BEUzhuAUrPBQhR08='
- 3 CRYPT_DES, key required: true
CryptEncode(method,data,key,result)=32 / ok
μ b &"#ÇĀ+ý°'¥ B8f;r0-Pè<6âî,ĖĖ
[00] B5 | 06 | 9D | 62 | 11 | 26 | 93 | 23 | C7 | C5 | 2B | FD | BA | 27 | A5 | 10 |
[16] 42 | 38 | 66 | A1 | 72 | D8 | 2D | 50 | E8 | 3C | 36 | E2 | EC | 82 | CB | A3 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_DES):
base64:'tQadYhEmkyPHxSv9uielEEI4ZqFy2C1Q6Dw24uyCy6M='
- 4 CRYPT_HASH_SHA1, key required: false
CryptEncode(method,data,key,result)=20 / ok
$ßö*©°ø
€|)bĖbzÇí Ů€
[00] A7 | DF | F6 | 2A | A9 | BA | F8 | 0A | 80 | 7C | 29 | 62 | CB | 62 | 7A | C7 |
[16] CD | 0E | DB | 80 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=28 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_HASH_SHA1):
base64:'p9/2Kqm6+AqAfCliy2J6x80024A='
- 5 CRYPT_HASH_SHA256, key required: false
CryptEncode(method,data,key,result)=32 / ok
ÚZ2šĖ»"¾7 €... ñ-ĀĀ'~|"ome2r@¾ô®³"
[00] DA | 5A | 32 | 9A | 80 | BB | 94 | BE | 37 | 0C | 80 | 85 | 07 | F1 | 96 | C4 |
[16] C1 | B4 | 98 | A6 | 93 | 6F | 6D | 65 | 32 | 72 | 40 | BE | F4 | AE | B3 | 94 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_HASH_SHA256):
base64:'2loymoC7LL43DICFB/GWxMG0mKaTb21lMnJAvvSus5Q='
- 6 CRYPT_HASH_MD5, key required: false

```

```

CryptEncode(method,data,key,result)=16 / ok
zIGT...  Fû;-3pèå
[00] 7A | 49 | 47 | 54 | 85 | 1B | 7F | 11 | 46 | FB | 3B | 97 | 33 | FE | E8 | E5 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=24 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_HASH_MD5):
base64:'eklHVIUbfxFG+zuXM/7o5Q=='
- 7 CRYPT_ARCH_ZIP, key required: false
CryptEncode(method,data,key,result)=34 / ok
x^óI-Q/VHÍK.ª,(Q(ÉÈ,VÈM-.NLO
[00] 78 | 5E | F3 | 49 | 2D | 51 | 2F | 56 | 48 | CD | 4B | 2E | AA | 2C | 28 | 51 |
[16] 28 | C9 | C8 | 2C | 56 | C8 | 4D | 2D | 2E | 4E | 4C | 4F | 05 | 00 | 80 | 07 |
[32] 09 | C2 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=48 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_ARCH_ZIP):
base64:'eF7zSS1RL1ZIZUsuqiwoUSjJyCxWyE0tLk5MTwUAgAcJwg=='

```

在这种情况下，该密钥的长度足以满足所有三种加密方法的需求，而其他不需要密钥的方法则直接忽略它。因此，所有函数调用均已成功完成。

在下一节中，我们将学习如何解码加密内容，并可以检查 *CryptDecode* 函数是否返回原始消息。请注意这部分日志。

启用的 *DisableCRCinZIP* 选项将使 CRYPT_ARCH_ZIP 方法的结果减少几个开销字节。

```

- 7 CRYPT_ARCH_ZIP, key required: false
CryptEncode(method,data,key,result)=28 / ok
óI-Q/VHÍK.ª,(Q(ÉÈ,VÈM-.NLO
[00] F3 | 49 | 2D | 51 | 2F | 56 | 48 | CD | 4B | 2E | AA | 2C | 28 | 51 | 28 | C9 |
[16] C8 | 2C | 56 | C8 | 4D | 2D | 2E | 4E | 4C | 4F | 05 | 00 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=40 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_ARCH_ZIP):
base64:'80ktUS9WSM1LLqosKFEoycgsVshNLS50TE8FAA=='

```

现在，我们将编码实验转移到文件上。为此，再次运行脚本并清除 *Text* 字段中的文本。结果，程序将多次处理 *MQL5Book/clock10.htm* 文件，并将创建几个具有不同扩展名的派生文件。

```

File=MQL5Book/clock10.htm / ok
FileLoad(File,data)=988 / ok
- 0 CRYPT_BASE64, key required: false
CryptEncode(method,data,key,result)=1320 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.BASE64
- 1 CRYPT_AES128, key required: true
CryptEncode(method,data,key,result)=992 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.AES128
- 2 CRYPT_AES256, key required: true
CryptEncode(method,data,key,result)=992 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.AES256
- 3 CRYPT_DES, key required: true
CryptEncode(method,data,key,result)=992 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.DES
- 4 CRYPT_HASH_SHA1, key required: false
CryptEncode(method,data,key,result)=20 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.SHA1
[00] 486ADFDD071CD23AB28E820B164D813A310B213F
- 5 CRYPT_HASH_SHA256, key required: false
CryptEncode(method,data,key,result)=32 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.SHA256
[00] 8990BBAC9C23B1F987952564EBCEF2078232D8C9D6F2CCC2A50784E8CDE044D0
- 6 CRYPT_HASH_MD5, key required: false
CryptEncode(method,data,key,result)=16 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.MD5
[00] 0CC4FBC899554BE0C0DBF5C18748C773
- 7 CRYPT_ARCH_ZIP, key required: false
CryptEncode(method,data,key,result)=687 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.ZIP

```

你可以从文件管理器中查看所有文件的内部，并确保与原始内容已无共同之处。许多文件管理器都有计算哈希和的命令或插件，以便可以将它们与打印到日志中的 MD5、SHA1 和 SHA256 十六进制值进行比较。

如果我们尝试编码文本或文件而未提供正确长度的密钥，我们将收到 `INVALID_ARRAY(4006)` 错误。例如，对于默认的文本消息，我们在 *method* 参数中选择 AES256（需要 32 字节的密钥）。使用 *GenerateKey* 参数，我们预订一个长度为 16 字节的密钥（或者你可以部分或完全删除 *CustomKey* 字段中的文本，保留 *GenerateKey* 为默认值）。

Key (bytes):

```
[00] 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
Text=Let's encrypt this message / ok
StringToCharArray(Text,data,0,-1,CP_UTF8)=26 / ok
- 0 CRYPT_AES256, key required: true
CryptEncode(method,data,key,result)=0 / INVALID_ARRAY(4006)
```

你还可以使用 `CRYPTARCH_ZIP` 方法或使用常规压缩软件压缩同一个文件 (就像我们对 *clock10.htm* 所做的那样)。如果你稍后使用二进制查看器实用程序 (通常内置于文件管理器中) 查看, 那么两个结果都将显示一个共同的打包块, 差异仅在于围绕它的元数据。

[illegible]

使用 CRYPT ARCH ZIP 方法压缩的文件 (左) 与包含它的标准 ZIP 存档 (右) 的比较

它显示存档的中间和主要部分是一个字节序列 (以深色突出显示), 与 *CryptEncode* 函数产生的字节序列相同。

最后，我们将展示图形文件 `clock10.png` 的 **Base64** 文本表示是如何生成的。为此，清除字段 **Text** 并在 **File** 参数中写入 `MQL5Book/clock10.png`。在下拉列表 **Method** 中选择 **Base64**。

```
File=MQL5Book/clock10.png / ok
FileLoad(File,data)=457 / ok
- 0 CRYPT_BASE64, key required: false
CryptEncode(method,data,key,result)=612 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.png.BASE64
```

结果创建了 `clock10.png.BASE64` 文件。在其中，我们将看到插入到网页代码中 `img` 标签里的那一行文本。

顺便说一句，“deflate”压缩方法是 PNG 图形格式的基础，因此我们可以使用 **CryptEncode** 将资源位图保存为 PNG 文件。随书附带了头文件 **PNG.mqh**，其中包含描述图像所需内部结构的最小支持；建议独立试验其源代码。使用 **PNG.mqh**，我们编写了一个简单的脚本 **CryptPNG.mq5**，它将终端附带的“euro.bmp”文件中的资源转换为“my.png”文件。未实现 PNG 文件的加载。

```
#resource "\\Images\\euro.bmp"

#include <MQL5Book/PNG.mqh>

void OnStart()
{
    uchar null[];      // empty key for CRYPT_ARCH_ZIP
    uchar result[];    // receiving array
    uint data[];       // original pixels
    uchar bytes[];     // original bytes
    int width, height;
    PRTF(ResourceReadImage("::Images\\euro.bmp", data, width, height));

    ArrayResize(bytes, ArraySize(data) * 3 + width); // *3 for PNG_CTYPE_TRUECOLOR (RGB)
    ArrayInitialize(bytes, 0);
    int j = 0;
    for(int i = 0; i < ArraySize(data); ++i)
    {
        if(i % width == 0) bytes[j++] = 0; // each line is prepended with a filter mode
        const uint c = data[i];
        // bytes[j++] = (uchar)((c >> 24) & 0xFF); // alpha, for PNG_CTYPE_TRUECOLORALF
        bytes[j++] = (uchar)((c >> 16) & 0xFF);
        bytes[j++] = (uchar)((c >> 8) & 0xFF);
        bytes[j++] = (uchar)(c & 0xFF);
    }

    PRTF(CryptEncode(CRYPT_ARCH_ZIP, bytes, null, result));

    int h = PRTF(FileOpen("my.png", FILE_BIN | FILE_WRITE));

    PNG::Image image(width, height, result); // default PNG_CTYPE_TRUECOLOR (RGB)
    image.write(h);

    FileClose(h);
}
```

7.4.3 数据解密和解压缩: CryptDecode

为了执行数据解密和解压缩操作, MQL5 提供了 *CryptDecode* 函数。

CryptDecode 函数使用指定的方法将 *data* 数组的数据进行逆向转换到接收的 *result* 数组中。

```
int CryptDecode(ENUM_CRYPT_METHOD method, const uchar &data[], const uchar &key[], uchar &result[])
```

请注意, 由 *CryptEncode* 函数等执行的哈希和计算是单向转换: 无法从哈希中恢复原始数据。

函数返回放置在目标数组中的字节数, 如果出错则返回 0。错误代码将添加到 *_LastError* 中。例如, 如果我们尝试解码哈希 (*method* 等于 *CRYPT_HASH* 常量之一), 则可能是 *INVALID_PARAMETER* (4003); 如果解密密钥长度不足或缺失, 则可能是 *INVALID_ARRAY* (4006)。

如果密钥不正确 (与加密时使用的密钥不同), 我们将得到乱码而不是编码的源数据, 但错误代码为零。这是函数的正常行为。

我们使用相同的脚本 *CryptDecode.mq5* 来检查 *CryptDecode* 的工作。

在输入参数中, 你可以指定要转换的文本或文件。文本始终隐含为 *Base64* 编码, 因为所有编码后的数据都是二进制格式, 并且 *input* 参数中不支持。转换方法从 *Method* 列表中选择。

```
input string Text; // Text (base64, or empty to process File)
input string File = "MQL5Book/clock10.htm.BASE64";
input ENUM_CRYPT_METHOD_EXT Method = _CRYPT_BASE64;
```

如果 *GenerateKey* 包含 *DUMMY_KEY_CUSTOM* 选项, 则加密方法需要一个密钥, 该密钥可以在 *CustomKey* 字段中指定为字符串。你还可以从 *DUMMY_KEY_LENGTH* 枚举 (与 *CryptEncode.mq5* 脚本中的相同) 生成所需长度的演示密钥。

```
input DUMMY_KEY_LENGTH GenerateKey = DUMMY_KEY_CUSTOM; // GenerateKey (length, or fr
input string CustomKey = "My top secret key is very strong";
input bool DisableCRCinZIP = false;
```

在 *GenerateKey* 和 *CustomKey* 中, 你应选择与启动 *CryptEncode.mq5* 时相同的值。

OnStart 中的算法首先描述所需的数组, 并从字符串获取密钥或通过简单生成 (仅用于演示, 应使用专用软件或算法生成可工作的抗加密密钥)。

```

void OnStart()
{
    ENUM_CRYPT_METHOD method = 0;
    int methods[];
    uchar key[] = {};          // default empty key suitable for zip and base64
    uchar data[], result[];
    uchar zip[], opt[] = {1, 0, 0, 0};

    if(GenerateKey == DUMMY_KEY_CUSTOM)
    {
        if(StringLen(CustomKey))
        {
            PRTF(CustomKey);
            StringToCharArray(CustomKey, key, 0, -1, CP_UTF8);
            ArrayResize(key, ArraySize(key) - 1);
        }
    }
    else if(GenerateKey != DUMMY_KEY_0)
    {
        ArrayResize(key, GenerateKey);
        for(int i = 0; i < GenerateKey; ++i) key[i] = (uchar)i;
    }

    if(ArraySize(key))
    {
        Print("Key (bytes):");
        ByteArrayPrint(key);
    }
    else
    {
        Print("Key is not provided");
    }
}

```

接下来,我们读取文件内容或从 *Text* 字段解码 *Base64* (取决于填充了哪个字段)以获取要处理的数据。

```

method = (ENUM_CRYPT_METHOD)Method;
Print("- ", EnumToString(method), ", key required: ", KEY_REQUIRED(method));
if(StringLen(Text))
{
    if(method != CRYPT_BASE64)
    {
        // since all methods except Base64 produce binary results,
        // they are additionally converted to CryptEncode.mq5 using Base64 to text,
        // so here we want to recover binary data from text input
        // before decryption
        uchar base64[];
        const uchar dummy[] = {};
        PRTF(Text);
        PRTF(StringToCharArray(Text, base64, 0, -1, CP_UTF8));
        ArrayResize(base64, ArraySize(base64) - 1);
        Print("Text (bytes):");
        ByteArrayPrint(base64);
        if(!PRTF(CryptDecode(CRYPT_BASE64, base64, dummy, data)))
        {
            return; // error
        }

        Print("Raw data to decipher (after de-base64):");
        ByteArrayPrint(data);
    }
    else
    {
        PRTF(StringToCharArray(Text, data, 0, StringLen(Text), CP_UTF8));
        ArrayResize(data, ArraySize(data) - 1);
    }
}
else if(StringLen(File))
{
    PRTF(File);
    if(PRTF(FileLoad(File, data)) <= 0)
    {
        return; // error
    }
}

```

如果用户尝试从哈希中恢复数据，我们将显示警告。

```

if(IS_HASH(method))
{
    Print("WARNING: hashes can not be used to restore data! CryptDecode will fail."
}

```

最后，我们直接执行解密或解压缩（解包）。对于文本，结果只被记录下来。对于文件，我们在名称后添加扩展名“.dec”并写入一个新文件：可以将其与原始文件（使用 *CryptEncode.mq5* 脚本处理的文件）进行比较。

```

ResetLastError();
if(PRTF(CryptDecode(method, data, key, result)))
{
    if(StringLen(Text))
    {
        Print("Text restored:");
        Print(CharArrayToString(result, 0, WHOLE_ARRAY, CP_UTF8));
    }
    else // File
    {
        const string filename = File + ".dec";
        if(PRTF(FileSave(filename, result)))
        {
            Print("File saved: ", filename);
        }
    }
}

```

如果你使用默认设置运行该脚本，它将尝试解码文件 *MQL5Book/clock10.htm.BASE64*。假设这是在上一节的实验中创建的，因此该过程应该会成功。

```

- CRYPT_BASE64, key required: false
File=MQL5Book/clock10.htm.BASE64 / ok
FileLoad(File,data)=1320 / ok
CryptDecode(method,data,key,result)=988 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.BASE64.dec

```

获得的 *clock10.htm.BASE64.dec* 文件与原始的 *clock10.htm* 完全相同。如果你解密扩展名为 AES128、AES256 或 DES 的文件，并且指定了与加密时使用的相同密钥，则应该发生同样的情况。

为清晰起见，我们检查文本的解密。以前，使用 AES128 方法加密一个已知短语会产生一个二进制文件，为方便起见，该二进制文件被转换为以下 **Base64** 字符串。

```
AQuvVCoSy1szaN80wy8tQxl9rIrRj9h0qK7KgYYGh9E=
```

我们将其输入到 **Text** 字段，并在 **Method** 下拉列表中选择 AES128。我们将看到以下日志。

```

CustomKey=My top secret key is very strong / ok
Key (bytes):
[00] 4D | 79 | 20 | 74 | 6F | 70 | 20 | 73 | 65 | 63 | 72 | 65 | 74 | 20 | 6B | 65 |
[16] 79 | 20 | 69 | 73 | 20 | 76 | 65 | 72 | 79 | 20 | 73 | 74 | 72 | 6F | 6E | 67 |
- CRYPT_AES128, key required: true
Text=AQuvVCoSy1szaN80wy8tQxl9rIrRj9h0qK7KgYYGh9E= / ok
StringToCharArray(Text,base64,0,-1,CP_UTF8)=44 / ok
Text (bytes):
[00] 41 | 51 | 75 | 76 | 56 | 43 | 6F | 53 | 79 | 31 | 73 | 7A | 61 | 4E | 38 | 4F |
[16] 77 | 79 | 38 | 74 | 51 | 78 | 6C | 39 | 72 | 49 | 72 | 52 | 6A | 39 | 68 | 4F |
[32] 71 | 4B | 37 | 4B | 67 | 59 | 59 | 47 | 68 | 39 | 45 | 3D |
CryptDecode(CRYPT_BASE64,base64,dummy,data)=32 / ok
Raw data to decipher (after de-base64):
[00] 01 | 0B | AF | 54 | 2A | 12 | CB | 5B | 33 | 68 | DF | 0E | C3 | 2F | 2D | 43 |
[16] 19 | 7D | AC | 8A | D1 | 8F | D8 | 4E | A8 | AE | CA | 81 | 86 | 06 | 87 | D1 |
CryptDecode(method,data,key,result)=32 / ok
Text restored:
Let's encrypt this message

```

信息已成功解密。

如果使用相同的输入文本，选择生成任意密钥（尽管长度足够），得到的将是胡言乱语，而不是信息。

```

Key (bytes):
[00] 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
- CRYPT_AES128, key required: true
Text=AQuvVCoSy1szaN80wy8tQxl9rIrRj9h0qK7KgYYGh9E= / ok
StringToCharArray(Text,base64,0,-1,CP_UTF8)=44 / ok
Text (bytes):
[00] 41 | 51 | 75 | 76 | 56 | 43 | 6F | 53 | 79 | 31 | 73 | 7A | 61 | 4E | 38 | 4F |
[16] 77 | 79 | 38 | 74 | 51 | 78 | 6C | 39 | 72 | 49 | 72 | 52 | 6A | 39 | 68 | 4F |
[32] 71 | 4B | 37 | 4B | 67 | 59 | 59 | 47 | 68 | 39 | 45 | 3D |
CryptDecode(CRYPT_BASE64,base64,dummy,data)=32 / ok
Raw data to decipher (after de-base64):
[00] 01 | 0B | AF | 54 | 2A | 12 | CB | 5B | 33 | 68 | DF | 0E | C3 | 2F | 2D | 43 |
[16] 19 | 7D | AC | 8A | D1 | 8F | D8 | 4E | A8 | AE | CA | 81 | 86 | 06 | 87 | D1 |
CryptDecode(method,data,key,result)=32 / ok
Text restored:
??? ?L?? ??J Q+?]v?9?????n?N?Ů

```

如果你混淆了加密方法，程序的行为将类似。

选择“反哈希”方法是没有意义的：INVALID_PARAMETER (4003)。

```

- CRYPT_HASH_MD5, key required: false
File=MQL5Book/clock10.htm.MD5 / ok
FileLoad(File,data)=16 / ok
WARNING: hashes can not be used to restore data! CryptDecode will fail.
CryptDecode(method,data,key,result)=0 / INVALID_PARAMETER(4003)

```

尝试解包（CRYPT_ARCH_ZIP）并非压缩的“deflate”数据块的内容将导致 INTERNAL_ERROR (4001)。如果你为一个没有 CRC 的“存档”启用了跳过 CRC 选项，或者相反，在没有该选项的情况下解压缩数据，尽管打包时使用了该选项，也可能得到同样的错误。

7.5 网络函数

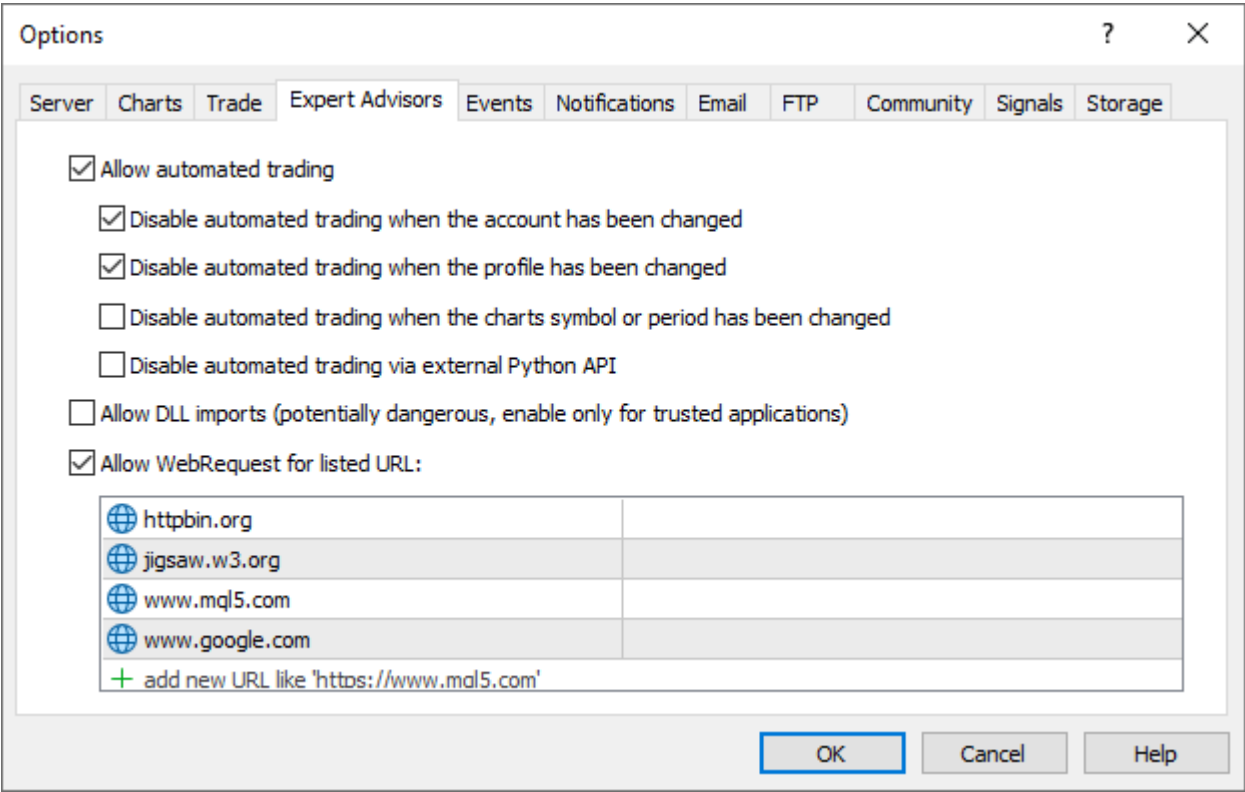
MQL 程序可以使用各种协议与分布式网络或互联网服务器上的其他计算机通信。这些函数支持与网站和服务 (HTTP/HTTPS)、文件传输 (FTP)、电子邮件发送 (SMTP) 以及推送通知相关的操作。

网络函数可以分为三组：

- ① **SendFTP**、**SendMail** 和 **SendNotification** 是用于发送文件、电子邮件和移动通知的最基本函数。
- ① **WebRequest** 函数旨在处理 Web 资源，并允许你轻松发送 HTTP 请求 (包括 GET 和 POST)。
- ① **Socket** 函数集允许你通过系统套接字与远程主机创建 TCP 连接 (包括安全的 TLS 连接)。

各组列出的顺序对应于从提供客户端和服务端之间现成交互机制的高级函数，到允许根据特定公共服务 (例如，加密货币交易所或交易信号服务) 的要求实现任意应用协议的低级函数的过渡。当然，这样的实现需要付出大量的努力。

为了最终用户的安全，MQL 程序可以使用 **Socket** 函数和 **WebRequest** 连接的允许 Web 地址列表，必须在设置对话框的 **Expert Advisors** 选项卡中明确指定。在这里，你可以指定域名、网页的完整路径 (不仅是站点，还包括 URL 的其他片段，如文件夹或端口号) 或 IP 地址。以下是本章示例中某些域设置的屏幕截图。



终端设置中访问网络资源的权限

你无法以编程方式编辑此列表。如果你尝试访问不在此列表中的网络资源，MQL 程序将收到错误并且请求将被拒绝。

一定要注意，所有网络函数仅提供到特定服务器的客户端连接，也就是说，无法使用 MQL5 组织一个服务器来等待和处理传入的请求。要这样做，需要将终端与外部程序或互联网服务 (例如，云服务) 集成。

7.5.1 发送推送通知

我们知道，终端允许你将来自 MetaQuotes 服务、终端本身以及 MQL 程序的推送通知发送到 **iOS** 或 **Android** 操作系统的移动设备。该技术使用 MetaQuotes ID，这是一个用户的唯一标识符（见注释）。MetaQuotes ID 是在用户设备上安装移动版终端时分配的，之后必须在终端设置的“通知”选项卡中指定该 ID（可以指定多个标识符，用逗号分隔）。此后，MQL 程序便可使用发送推送通知的功能。

事实上，MetaQuotes ID 并非标识用户，而是标识移动终端的特定安装实例；一个用户可以有多个安装实例。默认情况下，该 ID 与 mql5.com 社区的注册无关，尽管可以在网站上指定这种绑定。不要将社区中的用户注册与 MetaQuotes ID 混淆。要使用通知功能，终端用户无需登录社区。

bool SendNotification(const string text)

SendNotification 函数将带有指定文本的推送通知发送到所有具有终端设置中 MetaQuotes ID 的移动终端。消息长度不超过 255 个字符。

如果通知成功从终端发送，函数返回 **true**，如果发生错误则返回 **false**。**_LastError** 中可能的错误代码包括：

- 4515 -ERR_NOTIFICATION_SEND_FAILED -通信问题
- 4516 -ERR_NOTIFICATION_WRONG_PARAMETER -无效参数，例如空字符串
- 4517 -ERR_NOTIFICATION_WRONG_SETTINGS -MetaQuotes ID 配置不正确或缺失
- 4518 -ERR_NOTIFICATION_TOO_FREQUENT -函数调用过于频繁

如果与服务器存在连接，消息会立即发送。如果用户的设备在线，消息应该会到达收件人，但在一般情况下无法保证送达。程序不会收到关于消息送达情况的回执通知。服务器上不保存用于延迟发送的推送消息历史记录。

该函数的使用频率有限制：每秒不超过 2 次调用，每分钟不超过 10 次。

SendNotification 函数不在策略测试程序中执行。

本书包含一个简单的脚本 **NetNotification.mq5**，设置正确时，它会发送一个测试通知。

```
void OnStart()
{
    const string message = MQLInfoString(MQL_PROGRAM_NAME)
        + " runs on " + AccountInfoString(ACCOUNT_SERVER)
        + " " + (string)AccountInfoInteger(ACCOUNT_LOGIN);
    Print("Sending notification: " + message);
    PRTF(SendNotification(NULL)); // INVALID_PARAMETER(4003)
    PRTF(SendNotification(message)); // NOTIFICATION_WRONG_SETTINGS(4517) or 0 (success)
}
```

7.5.2 发送电子邮件通知

终端允许你将电子邮件发送到设置对话框中“邮件”选项卡上指定的电子邮件地址。为此，MQL5 提供了 **SendMail** 函数。

bool SendMail(const string subject, const string text)

该函数参数设置标题和文本（邮件正文）。

如果邮件已在邮件服务器上排队等待发送,则函数返回 *true*; 否则返回 *false*。如果在设置中禁用了邮件功能,或者邮件数据 (SMTP 服务器、端口、登录名、密码) 包含错误或未指定,则可能发生错误。

SendMail 函数不在策略测试程序中执行。

MQL5 不支持检查传入邮件和阅读邮件 (即 POP、MAP 协议)。

本书包含 *NetMail.mq5* 脚本,该脚本会尝试发送一封测试邮件。

```
void OnStart()
{
    const string message = "Hello from "
        + AccountInfoString(ACCOUNT_SERVER)
        + " " + (string)AccountInfoInteger(ACCOUNT_LOGIN);
    Print("Sending email: " + message);
    PRTF(SendMail(MQLInfoString(MQL_PROGRAM_NAME),
        message)); // MAIL_SEND_FAILED(4510) or 0 (success)
}
```

7.5.3 将文件发送到 FTP 服务器

MetaTrader 5 支持向 FTP 服务器发送文件。要使此功能正常工作,你必须在设置对话框的“FTP”选项卡中输入必要的 FTP 详细信息:FTP 服务器地址、登录名、密码,以及用于在服务器上放置文件的路径 (可选)。如果你的计算机位于 ISP 网络中,而 ISP 未为你分配公共 IP 地址,那么你可能需要开启被动模式。

MQL 程序直接发送文件的功能依赖于 *SendFTP* 函数。

bool SendFTP(const string filename, const string path = NULL)

该函数将具有指定名称的文件从终端设置发送到 FTP 服务器。如果需要,你可以指定一个与预先配置的路径不同的路径。如果未指定 *path* 参数,则使用设置中描述的目录。

上传的文件必须位于 *MQL5/Files* 文件夹或其子文件夹中。

该函数返回操作状态标识:成功 (*true*) 或错误 (*false*)。_LastError 中潜在的错误包括:

- 4514 -ERR_FTP_SEND_FAILED -通过 FTP 发送文件失败
- 4519 -ERR_FTP_NO_SERVER -未指定 FTP 服务器
- 4520 -ERR_FTP_NO_LOGIN -未指定 FTP 登录名
- 4521 -ERR_FTP_FILE_ERROR -在 MQL5/Files 目录中未找到指定文件
- 4522 -ERR_FTP_CONNECT_FAILED -连接到 FTP 服务器时发生错误
- 4523 -ERR_FTP_CHANGEDIR -在 FTP 服务器上未找到用于上传文件的目录
- 4524 -ERR_FTP_CLOSED -与 FTP 服务器的连接已关闭

该函数会阻塞 MQL 程序的执行,直到操作完成。因此,不允许在指标中使用该函数。

此外, *SendFTP* 函数不在策略测试程序中执行。

终端仅支持向 FTP 服务器发送单个文件。MQL5 不提供所有其他 FTP 命令。

NetFtp.mq5 示例脚本会截取当前图表的屏幕截图,并尝试通过 FTP 发送。

```

void OnStart()
{
    const string filename = _Symbol + "-" + PeriodToString() + "-"
        + (string)(ulong)TimeTradeServer() + ".png";
    PRTF(ChartScreenShot(0, filename, 300, 200));
    Print("Sending file: " + filename);
    PRTF(SendFTP(filename, "/upload")); // 0 (success) or FTP_CONNECT_FAILED(4522), FT
}

```

7.5.4 通过 HTTP/HTTPS 与 Web 服务器进行数据交换

MQL5 允许你将程序与 Web 服务集成并从互联网请求数据。可以使用 **WebRequest** 函数通过 HTTP/HTTPS 协议发送和接收数据，该函数有两个版本：分别用于与 Web 服务器的简化交互和高级交互。

```

int WebRequest(const string method, const string url, const string cookie, const string referer,
    int timeout, const char &data[], int size, char &result[], string &response)
int WebRequest(const string method, const string url, const string headers, int timeout,
    const char &data[], char &result[], string &response)

```

这两个函数的主要区别在于，简化版本只允许你在请求中指定两种类型的标头：**cookie** 和 **referer**，即进行跳转的来源地址（这里没有拼写错误—历史上“referrer”这个词在 HTTP 标头中是通过一个 ‘r’ 书写的）。扩展版本接受一个通用的 **headers** 参数来发送任意一组头文件。请求标头的格式为“名称: 值”，如果多于一个，则通过换行符“\r\n”连接。

如果我们假设 **cookie** 字符串必须包含“name1=value1; name2=value2”，并且 **referer** 链接等于“google.com”，那么要调用第二个版本的函数并达到与第一个版本相同的效果，我们需要在 **headers** 参数中添加以下内容：“Cookie: name1=value1; name2=value2\r\nReferer: google.com”。

method 参数指定协议方法之一：“HEAD”、“GET”或“POST”。请求的资源或服务的地址在 **url** 参数中传递。根据 HTTP 规范，网络资源标识符的长度限制为 2048 字节，但在撰写本书时，MQL5 的限制为 1024 字节。

请求的最大持续时间由 **timeout**（毫秒）确定。

两个版本的函数都将数据从 **data** 数组传输到服务器。第一个选项另外需要指定此数组的大小（以字节为单位，即 **size**）。

要发送带有几个变量值的简单请求，你可以将它们组合成像“name1=value1&name2=value2&...”这样的字符串，并将其添加到 GET 请求地址中的分隔符‘?’之后，或者使用“Content-Type: application/x-www-form-urlencoded”标头将其放入 POST 请求的 **data** 数组中。对于更复杂的情况，例如上传文件，请使用 POST 请求和“Content-Type: multipart/form-data”。

接收的 **result** 数组获取服务器响应主体（如果有）。服务器响应标头放置在 **response** 字符串中。

函数返回服务器的 HTTP 响应代码，如果发生系统错误（例如，通信问题或参数错误），则返回 -1。**_LastError** 中可能出现的潜在错误代码包括：

- 5200 -ERR_W EBREQUEST_INVALID_ADDRESS -无效 URL
- 5201 -ERR_W EBREQUEST_CONNECT_FAILED -连接到指定 URL 失败
- 5202 -ERR_W EBREQUEST_TIMEOUT -从服务器接收响应的超时已过
- 5203 -ERR_W EBREQUEST_REQUEST_FAILED -请求导致的任何其他错误

回想一下，即使请求在 MQL5 级别没有错误地执行，应用程序错误也可能包含在服务器的 HTTP 响应代码中（例如，需要授权、无效的数据格式、页面未找到等）。在这种情况下，`result` 将为空，解决问题的方法通常需要在分析接收到的 *response* 标头之后才能明确。

要使用 *WebRequest* 函数，应将服务器地址添加到终端设置中 *Expert Advisors* 选项卡下的允许 URL 列表中。终端会根据指定的协议自动选择服务器端口：“http://”为 80，“https://”为 443。

WebRequest 函数是同步的，即它会在等待服务器响应时暂停程序执行。因此，不允许从指标中调用该函数，因为指标在每个品种的公共流中工作。一个指标执行的延迟将停止更新该交易品种的所有图表。

在策略测试程序中工作时，不执行 *WebRequest* 函数。

我们从一个执行单个请求的简单脚本 *WebRequestTest.mq5* 开始。在输入参数中，我们将提供方法选择（默认为“GET”）、测试网页的地址、附加标头（可选）以及超时。

```
input string Method = "GET"; // Method (GET,POST)
input string Address = "https://httpbin.org/headers";
input string Headers;
input int Timeout = 5000;
```

地址的输入方式与浏览器地址栏中的一样：所有被 HTTP 规范禁止直接在地址中使用的字符（包括本地字母数字），在发送前都会被 *WebRequest* 函数根据 *urlencode* 算法自动进行“转义”处理（浏览器也会执行完全相同的操作，但我们通常看不到，因为这种转义后的形式是为网络基础设施传输而设计的，并非直接供人阅读）。

我们还将添加 *DumpDataToFiles* 选项：当它等于 *true* 时，脚本会将服务器的响应保存到一个单独的文件中，因为它可能相当大。值 *false* 指示直接将数据输出到日志。

```
input bool DumpDataToFiles = true;
```

我们必须马上说明，测试此类脚本需要一个服务器。若有兴趣，可以安装一个本地 Web 服务器，例如 `node.js`，但这需要自行准备或安装服务器端脚本（在这种情况下，需要连接 JavaScript 模块）。一个更简单的方法是使用互联网上可用的公共测试 Web 服务器。例如，你可以使用 `httpbin.org`、`httpbingo.org`、`webhook.site`、`putsreq.com`、`www.mockable.io` 或 `reqbin.com`。它们提供不同的功能集合。选择或找到适合你的（方便易懂，或尽可能灵活）。

在 *Address* 参数中，默认是服务器 API `httpbin.org` 的 *endpoint* 地址。这个动态的“网页”将其请求的 HTTP 标头（以 JSON 格式）返回给客户端。因此，我们能够在程序中看到究竟是什么从终端到达了 Web 服务器。

不要忘记在终端设置中将“httpbin.org”域添加到允许列表中。

JSON 文本格式是 Web 服务的既成事实标准。在 `mql5.com` 网站上可以找到用于解析 JSON 的现成类实现，但现在，我们只“按原样”显示 JSON。

在 *OnStart* 处理程序中，我们使用给定的参数调用 *WebRequest*，并在错误代码为非负时处理结果。服务器响应标头 (*response*) 总是被记录下来。

```

void OnStart()
{
    uchar data[], result[];
    string response;

    int code = PRTF(WebRequest(Method, Address, Headers, Timeout, data, result, response));
    if(code > -1)
    {
        Print(response);
        if(ArraySize(result) > 0)
        {
            PrintFormat("Got data: %d bytes", ArraySize(result));
            if(DumpDataToFiles)
            {
                string parts[];
                URL::parse(Address, parts);

                const string filename = parts[URL_HOST] +
                    (StringLen(parts[URL_PATH]) > 1 ? parts[URL_PATH] : "/_index.htm");
                Print("Saving ", filename);
                PRTF(FileSave(filename, result));
            }
            else
            {
                Print(CharArrayToString(result, 0, 80, CP_UTF8));
            }
        }
    }
}

```

为了形成文件名，我们使用了头文件 `URL.mqh` 中的 `URL` 辅助类（此处将不会完整描述）。`URL::parse` 方法根据规范将传递的字符串解析为 `URL` 组件，因为 `URL` 的一般形式总是“protocol://domain.com:port/path?query#hash”；请注意，许多片段是可选的。结果放置在接收数组中，其中的索引对应于 `URL` 的特定部分，并在 `URL_PARTS` 枚举中描述：

```

enum URL_PARTS
{
    URL_COMPLETE,    // full address
    URL_SCHEME,      // protocol
    URL_USER,        // username/password (deprecated, not supported)
    URL_HOST,        // server
    URL_PORT,        // port number
    URL_PATH,        // path/directories
    URL_QUERY,       // query string after '?'
    URL_FRAGMENT,    // fragment after '#' (not highlighted)
    URL_ENUM_LENGTH
};

```

因此，当接收到的数据应写入文件时，脚本会将其创建在以服务器命名的文件夹中（`parts[URL_HOST]`），以此类推，继续处理其他部分，并保留 `URL` 中的路径层次结构（`parts[URL_PATH]`）：在最简单的情况下，这仅仅是“端点”的名称。当请求站点的主页时，路径仅包含斜杠 '/'，文件被命名为“_index.htm”。

我们尝试使用默认参数运行该脚本，首先记住在终端设置中允许此服务器。在日志中，我们将看到以下几行（服务器响应的 HTTP 标头和有关成功保存文件的消息）：

```
WebRequest(Method,Address,Headers,Timeout,data,result,response)=200 / ok
Date: Fri, 22 Jul 2022 08:45:03 GMT
Content-Type: application/json
Content-Length: 291
Connection: keep-alive
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

Got data: 291 bytes
Saving httpbin.org/headers
FileSave(filename,result)=true / ok
```

`httpbin.org/headers` 文件包含服务器看到的我们请求的标头（服务器在应答我们时自己添加了 JSON 格式）。

```
{
  "headers":
  {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Accept-Language": "ru,en",
    "Host": "httpbin.org",
    "User-Agent": "MetaTrader 5 Terminal/5.3333 (Windows NT 10.0; Win64; x64)",
    "X-Amzn-Trace-Id": "Root=1-62da638f-2554..." // <- this is added by the reverse p
  }
}
```

因此，终端报告它准备好接受任何类型的数据，支持特定方法的压缩以及首选语言列表。此外，它在 `User-Agent` 字段中显示为 `MetaTrader 5`。在与某些专门针对浏览器优化的站点工作时，后者可能是不可取的。然后我们可以在 `headers` 输入参数中指定一个虚构的名称，例如，“`User-Agent: Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.0.0 Safari/537.36`”。

上面列出的一些测试站点允许你在服务器上组织一个具有随机名称的临时测试环境，用于你的个人实验：为此，你需要从浏览器访问该站点并获取一个通常在 24 小时内有效的唯一链接。然后，你将能够使用此链接作为来自 MQL5 的请求地址，并直接从浏览器监控请求的行为。还可以在浏览器中配置服务器响应，特别是尝试提交表单。

我们把这个例子稍微复杂化一点。服务器可能需要客户端执行额外的操作来完成请求，特别是授权、执行“重定向”（转到不同地址）、降低请求频率等。所有这些“信号”都由 `WebRequest` 函数返回的特殊 HTTP 代码表示。例如，代码 301 和 302 表示因不同原因重定向，而 `WebRequest` 会在内部自动执行它，重新请求服务器指定的地址上的页面（因此，重定向代码永远不会出现在 MQL 程序代码中）。401 代码要求客户端提供用户名和密码，这里的全部责任在于我们。发送这些数据有很多方法。一个新的脚本 `WebRequestAuth.mq5` 演示了处理服务器使用 HTTP 响应标头请求的两种授权选项：“`WWW-Authenticate: Basic`”或“`WWW-Authenticate: Digest`”。在标头中它可能看起来像这样：

```
WWW-Authenticate:Basic realm="DemoBasicAuth"
```

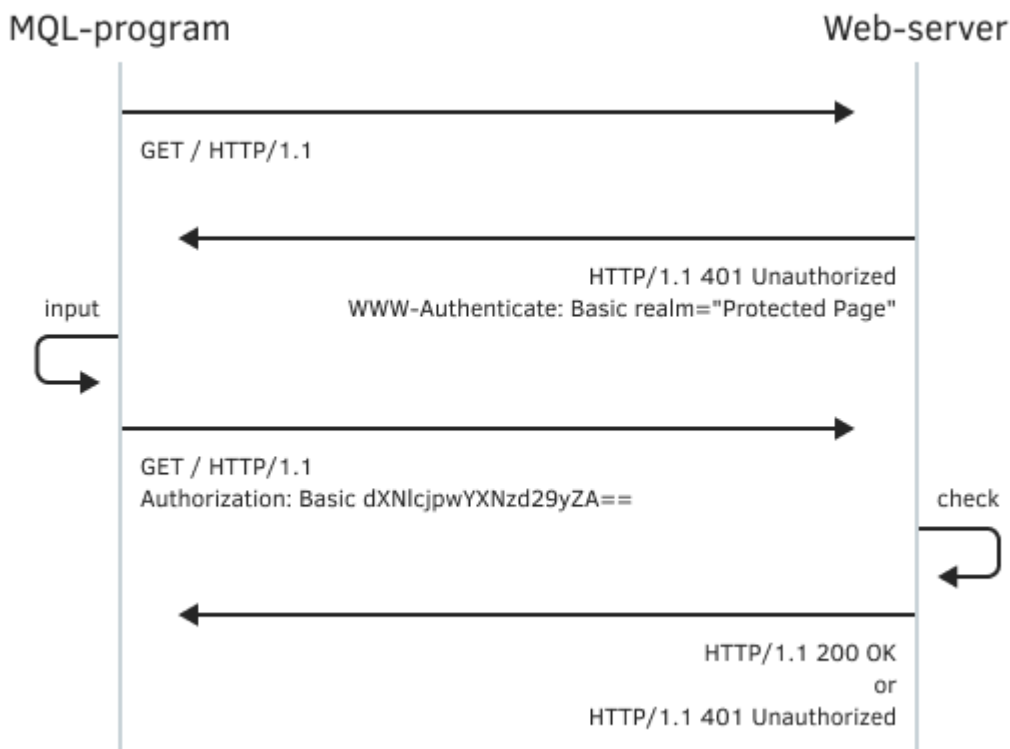
或者像这样：

```
WWW-Authenticate:Digest realm="DemoDigestAuth",qop="auth", »
» nonce="cuFAuHbb5UDvtFGkZEB2mNxjqEG/DjDr",opaque="fyNjGC4x8Zgt830PpzbXRvoqExsZeQSDZ
```

其中第一个最简单，但也最不安全，因此几乎不使用：在书中给出只是因为第一阶段学习它很容易。其工作要点是在响应服务器请求时生成以下 HTTP 请求，方法是添加一个特殊的标头：

```
Authorization: Basic dXNlcjpwYXNzd29yZA==
```

在这里，“Basic”关键字后面是 Base64 编码的字符串 “user:password”，其中包含实际的用户名和密码，而 ‘:’ 字符在此后 “按原样”作为连接块插入。更清楚地说，交互过程如图所示。



Web 服务器上的简单授权方案

Digest 授权方案被认为更高级。在这种情况下，服务器在其响应中提供一些附加信息：

- *realms* -生成条目的站点名称（站点区域）
- *qop* -Digest 方法的变体（我们只考虑 “auth”）
- *nonce* -将用于生成授权数据的随机字符串
- *opaque* -我们将在标头中 “按原样”传回的随机字符串
- *algorithm* -可选的哈希算法名称，默认为 MD5

要进行授权，你需要执行以下步骤：

1. 生成你自己的随机字符串 *cnonce*
2. 初始化或递增你的请求计数器 *nc*
3. 计算 *hash1* = MD5(*user:realm:password*)
4. 计算 *hash2* = MD5(*method:uri*), 这里 *uri* 是页面的路径和名称
5. 计算 *response* = MD5(*hash1:nonce:nc:cnonce:qop:hash2*)

之后，客户端便可再次向服务器发送请求，并在其请求标头添加如下内容：

```
Authorization: Digest username="user",realm="realm",nonce="...", »
» uri="/path/to/page",qop=auth,nc=00000001,cnonce="...",response="...",opaque="..."
```

由于服务器和客户端拥有相同的信息，服务器将能够重复这些计算步骤并验证哈希值是否匹配。

我们在脚本参数中添加用于输入用户名和密码的变量。默认情况下，**Address** 参数包含了 **digest-auth** 端点的地址，该端点可以通过参数 **qop** (“auth”)、登录名 (“test”) 和密码 (“pass”) 来请求授权。这些在端点路径中都是可选的（你可以测试其他方法和用户凭据，例如：“https://httpbin.org/digest-auth/auth-int/mqclient/mqpassword”）。

```
const string Method = "GET";
input string Address = "https://httpbin.org/digest-auth/auth/test/pass";
input string Headers = "User-Agent: noname";
input int Timeout = 5000;
input string User = "test";
input string Password = "pass";
input bool DumpDataToFiles = true;
```

我们在 **Headers** 参数中指定了一个虚拟的浏览器名称，用以演示此功能。

在 **OnStart** 函数中，我们增加了对 HTTP 401 错误代码的处理。如果未提供用户名和密码，程序将无法继续执行。

```
void OnStart()
{
    string parts[];
    URL::parse(Address, parts);
    uchar data[], result[];
    string response;
    int code = PRTF(WebRequest(Method, Address, Headers, Timeout, data, result, response));
    Print(response);
    if(code == 401)
    {
        if(StringLen(User) == 0 || StringLen>Password) == 0)
        {
            Print("Credentials required");
            return;
        }
    }
    ...
}
```

下一步是分析从服务器接收到的标头信息。为方便起见，我们编写了 **HTTPHeader** 类 (**HTTPHeader.mqh**)。完整的标头文本、元素分隔符（本例中为换行符 ‘\n’）以及每个元素内部名称与值之间使用的字符（本例中为冒号 ‘:’）都会传递给它构造函数。在创建过程中，该对象会“解析”文本，之后便可通过重载的 **[]** 操作符（其参数类型为字符串）来访问各个标头元素。因此，我们可以通过检查是否存在名为 “WWW-Authenticate” 的标头来判断服务器是否要求授权。如果文本中存在此元素且其值等于 “Basic”，我们就使用 **Base64** 编码的登录名和密码来构建 “Authorization: Basic” 响应标头。

```

code = -1;
HTTPHeader header(response, '\n', ':');
const string auth = header["WWW-Authenticate"];
if(StringFind(auth, "Basic ") == 0)
{
    string Header = Headers;
    if(StringLen(Header) > 0) Header += "\r\n";
    Header += "Authorization: Basic ";
    Header += HttpHeader::hash(User + ":" + Password, CRYPT_BASE64);
    PRTF(Header);
    code = PRTF(WebRequest(Method, Address, Header, Timeout, data, result, respc
    Print(response);
}
...

```

对于 Digest 授权, 过程则要复杂一些, 需遵循前述算法。


```

else if(StringFind(auth, "Digest ") == 0)
{
    HttpHeader params(StringSubstr(auth, 7), ',', '=');
    string realm = HttpHeader::unquote(params["realm"]);
    if(realm != NULL)
    {
        string qop = HttpHeader::unquote(params["qop"]);
        if(qop == "auth")
        {
            string h1 = HttpHeader::hash(User + ":" + realm + ":" + Password);
            string h2 = HttpHeader::hash(Method + ":" + parts[URL_PATH]);
            string nonce = HttpHeader::unquote(params["nonce"]);
            string counter = StringFormat("%08x", 1);
            string cnonce = StringFormat("%08x", MathRand());
            string h3 = HttpHeader::hash(h1 + ":" + nonce + ":" + counter + ":" +
                cnonce + ":" + qop + ":" + h2);

            string Header = Headers;
            if(StringLen(Header) > 0) Header += "\r\n";
            Header += "Authorization: Digest ";
            Header += "username=\"" + User + "\", ";
            Header += "realm=\"" + realm + "\", ";
            Header += "nonce=\"" + nonce + "\", ";
            Header += "uri=\"" + parts[URL_PATH] + "\", ";
            Header += "qop=" + qop + ", ";
            Header += "nc=" + counter + ", ";
            Header += "cnonce=\"" + cnonce + "\", ";
            Header += "response=\"" + h3 + "\", ";
            Header += "opaque=" + params["opaque"] + "";
            PRTF(Header);
            code = PRTF(WebRequest(Method, Address, Header, Timeout, data, result,
                Print(response));
        }
    }
}

```

静态方法 *HttpHeader::hash* 会为所有必需的组合字符串生成一个十六进制的哈希表示 (默认为 MD5)。基于这些数据, 将为下一次 *WebRequest* 调用构建相应的标头。静态方法 *HttpHeader::unquote* 用于移除值两侧的引号。

脚本的其余部分保持不变。重复发送的 HTTP 请求可能会成功, 届时我们将获得受保护页面的内容; 或者授权被拒绝, 服务器则会返回类似“访问被拒绝”的信息。

由于默认参数包含了正确的值 (“/digest-auth/auth/test/pass” 对应用户 “test” 和密码 “pass”), 运行脚本后我们应得到如下结果 (所有主要步骤和数据均已记录):

```

WebRequest(Method,Address,Headers,Timeout,data,result,response)=401 / ok
Date: Fri, 22 Jul 2022 10:45:56 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 0
Connection: keep-alive
Server: gunicorn/19.9.0
WWW-Authenticate: Digest realm="me@kennethreitz.com" »
» nonce="87d28b529a7a8797f6c3b81845400370", qop="auth",
» opaque="4cb97ad7ea915a6d24cf1ccbf6feeaba", algorithm=MD5, stale=FALSE
...

```

第一次 *WebRequest* 调用以代码 401 结束，响应标头中包含了授权请求（“W W W -Authenticate”）及必要的参数。基于这些参数，我们计算出正确的应答，并为新的请求准备好了标头。

```

Header=User-Agent: noname
Authorization: Digest username="test",realm="me@kennethreitz.com" »
» nonce="87d28b529a7a8797f6c3b81845400370",uri="/digest-auth/auth/test/pass",
» qop=auth,nc=00000001,cnonce="00001c74",
» response="c09e52bca9cc90caf9a707d046b567b2",opaque="4cb97ad7ea915a6d24cf1ccbf6feeaba"
...

```

第二次请求返回代码 200 以及有效负载，我们将此负载写入文件。

```

WebRequest(Method,Address,Header,Timeout,data,result,response)=200 / ok
Date: Fri, 22 Jul 2022 10:45:56 GMT
Content-Type: application/json
Content-Length: 47
Connection: keep-alive
Server: gunicorn/19.9.0
...
Got data: 47 bytes
Saving httpbin.org/digest-auth/auth/test/pass
FileSave(filename,result)=true / ok

```

在 *MQL5/Files/httpbin.org/digest-auth/auth/test/pass* 文件中，你可以找到该“网页”，或者更确切地说，是 JSON 格式的成功授权状态。

```

{
  "authenticated": true,
  "user": "test"
}

```

如果在运行脚本时指定了错误的密码，我们将从服务器收到一个空响应，并且不会写入文件。

使用 *WebRequest*，我们便自动进入了分布式软件系统的领域。在这类系统中，程序的正确运行不仅取决于我们客户端的 MQL 代码，还依赖于服务器（更不用说像代理服务器这样的中间环节了）。因此，你需要为别人可能出现的错误做好准备。例如，在撰写本书时，*httpbin.org* 上的 *digest-auth* 端点实现就存在一个问题：请求中输入的用户名并未参与授权校验，因此只要密码正确，任何登录名都能成功授权。尽管如此，为了检验我们的脚本，你还是可以尝试其他服务，比如 *httpbingo.org/digest-auth/auth/test/pass*。你也可以将脚本配置为访问 *jigsaw.w3.org/HTTP/Digest/* 地址，该地址期望的登录名/密码是“guest”/“guest”。

在实践中，大多数网站通过直接嵌入网页的表单来实现授权：在 HTML 代码内部，这些表单本质上是带有若干输入字段的 *form* 容器标签，用户填写这些字段后，通过 POST 方法发送给服务

器。有鉴于此,分析提交表单的示例颇有意义。然而,在深入探讨此问题之前,最好先强调另一种技术。

关键在于,客户端与服务器之间的交互通常伴随着双方状态的改变。以授权为例,这一点最容易理解:授权前,用户对系统而言是未知的;授权后,系统便知晓了用户的登录名,并能应用该用户的偏好设置(例如语言、颜色主题、论坛显示方式等),同时还允许用户访问那些未授权访客无法进入的页面(服务器会通过返回 HTTP 403 Forbidden 状态码来阻止此类尝试)。

分布式 Web 应用程序中客户端与服务器部分状态的一致性支持与同步,是通过 Cookie 机制实现的,该机制指的是 HTTP 标头中的命名变量及其值。“Cookie”这个词源于“幸运饼干”,因为 **cookies** 也包含着用户不可见的小段信息。

服务器和客户端任何一方都可以向 HTTP 标头添加 **cookie**。服务器通过类似下面这样的一行来实现:

```
Set-Cookie: name=value; [Domain=domain; Path=path; Expires=date; Max-Age=number_of_se
```

其中,名称和值是必需的,其余属性(如 **Domain**、**Path**、**Expires** 和 **Max age** 等)则是可选的,实际应用中属性可能更多。

收到这样的标头(或多个标头)后,客户端必须记住变量的名称和值,并在后续所有访问该 **Domain** 及域内对应 **Path** 的请求中(在过期日期 **Expires** 或 **Max-Age** 之前)将它们回传给服务器。

在客户端发出的 HTTP 请求中, **cookie** 以下列字符串形式传递:

```
Cookie: name(N)=value(N) [; name(i)=value(i) ...]opt
```

这里,所有由服务器设置、客户端已知、与当前请求的域和路径匹配且未过期的 `name=value` 对,都通过分号和空格分隔列出。

服务器和客户端在每次 HTTP 请求时都会交换所有必要的 Cookie,这种分布式系统的架构风格也因此被称为 REST (**Representational State Transfer**, 表述性状态转移)。例如,用户成功登录服务器后,服务器会通过“Set-Cookie:”标头)设置一个包含用户标识符的特殊“cookie”,之后 Web 浏览器(在我们的例子中是运行 MQL 程序的终端)会在后续请求中发送这个 Cookie(通过向“Cookie:”标头添加相应的行)。

WebRequest 函数会自动为我们处理所有这些工作:从传入的标头中收集 Cookie,并向传出的 HTTP 请求中添加相应的 Cookie。

Cookie 会由终端根据其设置在不同会话间进行存储。要验证这一点,只需从一个使用 Cookie 的网站请求两次网页即可。

注意:Cookie 是与特定网站相关联存储的,因此所有使用 **WebRequest** 访问同一网站的 MQL 程序,其传出标头中的 Cookie 都会被自动替换,这一点用户可能不易察觉。

为了简化连续请求的操作,一种合理做法是将常用行为封装到一个专门的 **HttpRequest** 类 (**HttpRequest.mqh**)。我们将在这个类中存储所有请求都可能需要的通用 HTTP 标头(例如,支持的语言、代理服务器指令等)。此外,像超时这样的设置也是通用的。这两个设置都通过对象的构造函数传入。

```

class HTTPRequest: public HttpCookie
{
protected:
    string common_headers;
    int timeout;

public:
    HTTPRequest(const string h, const int t = 5000):
        common_headers(h), timeout(t) { }
    ...

```

默认情况下, 超时设置为 5 秒。该类的核心方法, 在某种意义上也是通用的方法, 是 *request*。

```

int request(const string method, const string address,
            string headers, const uchar &data[], uchar &result[], string &response)
{
    if(headers == NULL) headers = common_headers;

    ArrayResize(result, 0);
    response = NULL;
    Print(">>> Request:\n", method + " " + address + "\n" + headers);

    const int code = PRTF(WebRequest(method, address, headers, timeout, data, result));
    Print("<<< Response:\n", response);
    return code;
}
};

```

我们再介绍几个针对特定类型查询的方法。

GET 请求仅使用标头, 而文档的主体 (通常称为 *payload*, 即有效负载) 为空。

```

int GET(const string address, uchar &result[], string &response,
        const string custom_headers = NULL)
{
    uchar nodata[];
    return request("GET", address, custom_headers, nodata, result, response);
}

```

POST 请求中通常包含有效负载。

```

int POST(const string address, const uchar &payload[],
         uchar &result[], string &response, const string custom_headers = NULL)
{
    return request("POST", address, custom_headers, payload, result, response);
}

```

表单可以以不同格式发送。最简单的是 "application/x-www-form-urlencoded"。它意味着有效负载将是一个字符串 (可能非常长, 因为规范没有限制, 完全取决于 Web 服务器的设置)。对于这类表单, 我们将提供一个带有 *payload* 字符串参数的更易用 POST 方法重载。

```

int POST(const string address, const string payload,
    uchar &result[], string &response, const string custom_headers = NULL)
{
    uchar bytes[];
    const int n = StringToCharArray(payload, bytes, 0, -1, CP_UTF8);
    ArrayResize(bytes, n - 1); // remove terminal zero
    return request("POST", address, custom_headers, bytes, result, response);
}

```

我们编写一个简单的脚本 *WebRequestCookie.mq5* 来测试我们的客户端 Web 引擎。它的任务是请求同一个网页两次：第一次服务器很可能会提供设置其 Cookie，然后这些 Cookie 将在第二次请求中被自动代入。在输入参数中，指定用于测试的页面地址：就设为 *mq15.com* 网站吧。我们还将通过修正后的 “User-Agent” 字符串来模拟默认的请求标头。

```

input string Address = "https://www.mql5.com";
input string Headers = "User-Agent: Mozilla/5.0 (Windows NT 10.0) Chrome/103.0.0.0";

```

在脚本的主函数中，我们描述 *HttpRequest* 对象，并在一个循环中执行两次 GET 请求。

注意！此测试基于一个假设：即 MQL 程序此前尚未访问过 *www.mql5.com* 网站，也未从该网站接收过 Cookie。一旦脚本运行过一次，Cookie 将保留在终端的缓存中，届时将无法重现此示例的初次效果：在循环的两次迭代中，我们都将得到相同的日志条目。

不要忘记在终端设置中将 “*www.mql5.com*” 域添加到允许列表中。

```

void OnStart()
{
    uchar result[];
    string response;
    HttpRequest http(Headers);

    for(int i = 0; i < 2; ++i)
    {
        if(http.GET(Address, result, response) > -1)
        {
            if(ArraySize(result) > 0)
            {
                PrintFormat("Got data: %d bytes", ArraySize(result));
                if(i == 0) // show the beginning of the document only the first time
                {
                    const string s = CharArrayToString(result, 0, 160, CP_UTF8);
                    int j = -1, k = -1;
                    while((j = StringFind(s, "\r\n", j + 1)) != -1) k = j;
                    Print(StringSubstr(s, 0, k));
                }
            }
        }
    }
}

```

循环的第一次迭代将生成以下日志条目（有删节）：

```

>>> Request:
GET https://www.mql5.com
User-Agent: Mozilla/5.0 (Windows NT 10.0) Chrome/103.0.0.0
WebRequest(method,address,headers,timeout,data,result,response)=200 / ok
<<< Response:
Server: nginx
Date: Sun, 24 Jul 2022 19:04:35 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache,no-store
Content-Encoding: gzip
Expires: -1
Pragma: no-cache
Set-Cookie: sid=CfDJ802AwC...Ne2yP5QXpPKA2; domain=.mql5.com; path=/; samesite=lax; h
Vary: Accept-Encoding
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Content-Security-Policy: default-src 'self'; script-src 'self' ...
Generate-Time: 2823
Agent-Type: desktop-ru-en
X-Cache-Status: MISS
Got data: 184396 bytes

<!DOCTYPE html>
<html lang="ru">
<head>
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />

```

我们收到了一个名为 **sid** 的新 Cookie。要验证其有效性，你可以查看日志的第二部分，即循环第二次迭代的记录。

```
>>> Request:
GET https://www.mql5.com
User-Agent: Mozilla/5.0 (Windows NT 10.0) Chrome/103.0.0.0
WebRequest(method,address,headers,timeout,data,result,response)=200 / ok
<<< Response:
Server: nginx
Date: Sun, 24 Jul 2022 19:04:36 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache, no-store, must-revalidate, no-transform
Content-Encoding: gzip
Expires: -1
Pragma: no-cache
Vary: Accept-Encoding
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Content-Security-Policy: default-src 'self'; script-src 'self' ...
Generate-Time: 2950
Agent-Type: desktop-ru-en
X-Cache-Status: MISS
```

遗憾的是，这里我们无法看到 **WebRequest** 内部形成的完整传出标头，但 **Cookie** 已通过“Cookie:”标头发送给服务器这一事实，可以从服务器在第二次响应中不再要求设置该 **Cookie** 得到证明。

理论上，这个 **Cookie** 仅仅是识别访问者（大多数网站都这样做），并不代表他们已获得授权。因此，我们回到以通用方式提交表单的练习，这实际上也为将来处理输入登录名和密码这类具体任务打下基础。

回想一下，要提交表单，我们可以使用带有字符串参数 **payload** 的 **POST** 方法。根据“**x-www-form-urlencoded**”标准准备数据的原则是，将命名的变量及其值写在一行连续的字符串中（有点类似于 **Cookie**）。

```
name('N')=value('N')[&name('i')=value('i')...]°P°t
```

名称和值用 '=' 符号连接，各个名称值对则使用 '&' 字符连接。值可以为空。例如，

```
Name=John&Age=33&Education=&Address=
```

务必注意，从技术角度看，这个字符串在发送前必须按照 **urlencode** 算法进行转换（该格式名称的由来即在于此），不过，**WebRequest** 函数会自动为我们完成这个转换。

变量名由 **Web** 表单（网页中 **form** 标签的内容）或 **Web** 应用程序的逻辑确定，但无论如何，**Web** 服务器必须能够理解这些名称和值。因此，为了熟悉这项技术，我们需要一个带有表单的测试服务器。

测试表单位于 <https://httpbin.org/forms/post>。这是一个用于订购披萨的对话框。

Customer name:

Telephone:

E-mail address:

Pizza Size

☐ Small

☐ Medium

☐ Large

Pizza Toppings

☐ Bacon

☐ Extra Cheese

☐ Onion

☐ Mushroom

Preferred delivery time:

Delivery instructions:

测试 Web 表单

其内部结构和行为由以下 HTML 代码描述。其中，我们主要关注的是 *input* 标签，它们设置了服务器期望接收的变量。此外，还应注意 *form* 标签中的 *action* 属性，因为它定义了 POST 请求应发送到的地址，在本例中是 `/post`，与域名结合后得到 `httpbin.org/post`。这正是我们将在 MQL 程序中使用的地址。


```

<!DOCTYPE html>
<html>
  <body>
    <form method="post" action="/post">
      <p><label>Customer name: <input name="custname"></label></p>
      <p><label>Telephone: <input type=tel name="custtel"></label></p>
      <p><label>E-mail address: <input type=email name="custemail"></label></p>
      <fieldset>
        <legend> Pizza Size </legend>
        <p><label> <input type=radio name=size value="small"> Small </label></p>
        <p><label> <input type=radio name=size value="medium"> Medium </label></p>
        <p><label> <input type=radio name=size value="large"> Large </label></p>
      </fieldset>
      <fieldset>
        <legend> Pizza Toppings </legend>
        <p><label> <input type=checkbox name="topping" value="bacon"> Bacon </label></p>
        <p><label> <input type=checkbox name="topping" value="cheese"> Extra Cheese </l
        <p><label> <input type=checkbox name="topping" value="onion"> Onion </label></p>
        <p><label> <input type=checkbox name="topping" value="mushroom"> Mushroom </lab
      </fieldset>
      <p><label>Preferred delivery time: <input type=time min="11:00" max="21:00" step=
      <p><label>Delivery instructions: <textarea name="comments"></textarea></label></p>
      <p><button>Submit order</button></p>
    </form>
  </body>
</html>

```

在 *WebRequestForm.mq5* 脚本中，我们准备了类似的输入变量，供用户在发送到服务器之前指定。

```

input string Address = "https://httpbin.org/post";

input string Customer = "custname=Vincent Silver";
input string Telephone = "custtel=123-123-123";
input string Email = "custemail=email@address.org";
input string PizzaSize = "size=small"; // PizzaSize (small,medium,large)
input string PizzaTopping = "topping=bacon"; // PizzaTopping (bacon,cheese,onion,mush
input string DeliveryTime = "delivery=";
input string Comments = "comments=";

```

预设的字符串仅为方便一键测试而显示：你可以将它们替换为你自己的内容，但请注意，在每个字符串内部，只应编辑 '=' 右侧的值，而应保留 '=' 左侧的名称（未知的名称将被服务器忽略）。

在 *OnStart* 函数中，我们描述了 HTTP 标头 "Content-Type:"，并准备了一个包含所有变量的连接字符串。

```

void OnStart()
{
    uchar result[];
    string response;
    string header = "Content-Type: application/x-www-form-urlencoded";
    string form_fields;
    StringConcatenate(form_fields,
        Customer, "&",
        Telephone, "&",
        Email, "&",
        PizzaSize, "&",
        PizzaTopping, "&",
        DeliveryTime, "&",
        Comments);
    HTTPRequest http;
    if(http.POST(Address, form_fields, result, response) > -1)
    {
        if(ArraySize(result) > 0)
        {
            PrintFormat("Got data: %d bytes", ArraySize(result));
            // NB: UTF-8 is implied for many content-types,
            // but some may be different, analyze the response headers
            Print(CharArrayToString(result, 0, WHOLE_ARRAY, CP_UTF8));
        }
    }
}

```

然后，我们执行 POST 方法并记录服务器的响应。以下是一个示例结果。

```

>>> Request:
POST https://httpbin.org/post
Content-Type: application/x-www-form-urlencoded
WebRequest(method,address,headers,timeout,data,result,response)=200 / ok
<<< Response:
Date: Mon, 25 Jul 2022 08:41:41 GMT
Content-Type: application/json
Content-Length: 780
Connection: keep-alive
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

Got data: 721 bytes
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "comments": "",
    "custemail": "email@address.org",
    "custname": "Vincent Silver",
    "custtel": "123-123-123",
    "delivery": "",
    "size": "small",
    "topping": "bacon"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Accept-Language": "ru,en",
    "Content-Length": "127",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "MetaTrader 5 Terminal/5.3333 (Windows NT 10.0; x64)",
    "X-Amzn-Trace-Id": "Root=1-62de5745-25bd1d823a9609f01cff04ad"
  },
  "json": null,
  "url": "https://httpbin.org/post"
}

```

测试服务器以 JSON 副本的形式确认收到了数据。在实际应用中，服务器当然不会返回数据本身，而只是报告成功状态，并可能重定向到受数据影响的另一个网页（例如，显示订单号）。

借助这类 POST 请求（通常规模较小），授权操作也常常得以执行。但说实话，大多数 Web 服务为了安全目的，会刻意将这个过程复杂化，并要求你首先根据用户详细信息计算几个哈希值。一般来说，专门开发的公共 API 会在其文档中包含所有必要算法的描述。但也不是绝对。例如，我们将无法使用 *WebRequest* 在 *mq15.com* 上登录，因为该站点没有开放的编程接口。

向 Web 服务发送请求时，请始终遵守关于不超过请求频率的规则：通常，每个服务都会指定自己的限制，违反这些限制将导致你的客户端程序、账户或 IP 地址随后被封锁。

7.5.5 建立和断开网络套接字连接

在前面的章节中，我们熟悉了 MQL5 的高级网络函数：它们各自为特定的应用协议提供支持。例如，SMTP 用于发送电子邮件 (*SendMail*)，FTP 用于文件传输 (*SendFTP*)，而 HTTP 允许接收 Web 文档 (*WebRequest*)。所有提及的标准都基于更底层的传输层 TCP (Transmission Control Protocol, 传输控制协议)。它在层级结构中并非最底层，还有更低层次的协议，但我们在此不作讨论。

应用协议的标准实现隐藏了许多内部的技术细节，使程序员无需花费数小时例行公事般地遵循规范。然而，标准实现缺乏灵活性，并且没有考虑到标准中嵌入的高级特性。因此，有时需要在 TCP 层面，即套接字层面进行网络通信编程。

可以将套接字比作磁盘上的文件：套接字也由一个整数描述符来描述，通过该描述符可以读取或写入数据，但这发生在分布式的网络基础设施中。与文件不同，计算机上的套接字数量是有限的，因此在将套接字描述符与网络资源 (地址、URL) 关联之前，必须预先向系统请求它。我们还要预先说明，通过套接字访问信息是流式的，也就是说，不可能像在文件中那样将某个“指针”回退到开头。

写入和读取线程互不相交，但可能会影响未来的读取或写入数据，因为传输的信息通常被服务器和客户端程序解释为控制命令。协议标准定义了流中包含的是命令还是数据。

SocketCreate 函数允许在 MQL5 中创建一个“空的”套接字描述符。

int SocketCreate(uint flags = 0)

其唯一的参数保留给将来用于指定决定套接字模式的标志位模式，但目前仅支持一个存根标志：SOCKET_DEFAULT 对应当前模式，可以省略。在系统层面，这相当于一个阻塞模式的套接字 (网络程序员可能会对此感兴趣)。

如果成功，函数返回套接字句柄。否则，返回 INVALID_HANDLE。

一个 MQL 程序最多可以创建 128 个套接字。当超出限制时，错误 5271 (ERR_NETSOCKET_TOO_MANY_OPENED) 会被记录到 *_LastError* 中。

打开套接字后，应将其与网络地址关联。

bool SocketConnect(int socket, const string server, uint port, uint timeout)

SocketConnect 函数使套接字连接到位于指定地址和端口的服务器 (例如，Web 服务器通常分别在 HTTP 和 HTTPS 的 80 或 443 端口上运行，SMTP 在 25 端口上运行)。地址可以是域名或 IP 地址。

timeout 参数允许你设置等待服务器响应的超时时间 (以毫秒为单位)。

函数返回成功连接的标志 (*true*) 或错误 (*false*)。错误代码写入 *_LastError*，例如 5272 (ERR_NETSOCKET_CANNOT_CONNECT)。

请注意，连接地址必须添加到终端设置中允许的地址列表中 (对话框 *Service -> Settings -> Advisors*)。

完成网络工作后，应使用 *SocketClose* 释放套接字。

bool SocketClose(const int socket)

SocketClose 函数通过其句柄关闭先前使用 *SocketCreate* 函数打开的套接字。如果套接字先前已通过 *SocketConnect* 连接，则连接将被断开。

函数还会返回成功 (*true*) 或错误 (*false*) 的指示。特别是, 向 *_LastError* 传递无效句柄时, 将在日志中记录错误 5270 (ERR_NETSOCKET_INVALIDHANDLE)。

需要注意的是, 本节及后续章节的所有函数都禁止在指标中使用: 在指标中尝试使用套接字将导致出现错误 4014 (ERR_FUNCTION_NOT_ALLOWED “系统函数不允许被调用”)。

来看一个入门示例: *SocketConnect.mql5* 脚本。在输入参数中, 你可以指定服务器的地址和端口。我们应该从像 *mql5.com* 这样的常规 Web 服务器开始测试。

```
input string Server = "www.mql5.com";
input uint Port = 443;
```

在 *OnStart* 函数中, 我们只是创建一个套接字并将其绑定到网络资源。

```
void OnStart()
{
    PRTF(Server);
    PRTF(Port);
    const int socket = PRTF(SocketCreate());
    if(PRTF(SocketConnect(socket, Server, Port, 5000)))
    {
        PRTF(SocketClose(socket));
    }
}
```

如果终端中的所有设置都正确并且已连接到互联网, 我们将得到以下“报告”。

```
Server=www.mql5.com / ok
Port=443 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,5000)=true / ok
SocketClose(socket)=true / ok
```

7.5.6 检查套接字状态

使用套接字时, 有必要检查其状态, 因为分布式网络不像文件系统那样可靠。特别是, 连接可能会因某种原因丢失。 *SocketIsConnected* 函数允许你查明这一点。

bool SocketIsConnected(const int socket)

该函数会检查具有指定句柄 (从 *SocketCreate* 获得) 的套接字是否已连接到其网络资源 (在 *Socket Connect* 中指定), 并在成功时返回 *true*。

另一个函数 *SocketIsReadable* 让你知道与套接字关联的系统缓冲区中是否有任何可读取的数据。这意味着我们通过网络地址连接到的计算机已向我们发送 (并可能继续发送) 数据。

uint SocketIsReadable(const int socket)

该函数返回可以从套接字读取的字节数。如果出错, 返回 0。

熟悉 Windows/Linux 套接字系统 API 的程序员知道, 当套接字内部缓冲区中没有传入数据时, 值 0 也可能是一种正常状态。然而, 该函数在 MQL5 中的行为有所不同。当系统套接字缓冲区为空时, 它会推测性地返回 1, 将实际的数据可用性检查推迟到下一次调用某个读取函数时。特别是, 这种返回虚拟 1 字节结果的情况, 通常发生在首次对套接字调用该函数且接收内部缓冲区仍为空时。

执行此函数时可能会发生错误，这意味着通过 `SocketConnect` 建立的连接已断开（在 `_LastError` 中我们将得到代码 5273, `ERR_NETSOCKET_IO_ERROR`）。

`SocketIsReadable` 函数在设计用于使用 `SocketRead` 进行“非阻塞”数据读取的程序中很有用。关键在于，当接收缓冲区中没有数据时，`SocketRead` 函数会等待数据到达，从而暂停程序的执行（暂停时间取决于指定的超时值）。

另一方面，阻塞读取在某种意义上更可靠，因为一旦新数据到达，你的程序就会“唤醒”，但是需要根据其他一些事件（通常是定时器或在循环中）定期使用 `SocketIsReadable` 检查数据是否存在。

在 TLS 安全模式下使用 `SocketIsReadable` 函数时应特别小心。该函数返回“原始”数据的量，在 TLS 模式下，这是一个加密块。如果“原始”数据尚未累积到解密块的大小，则后续调用读取函数 `SocketTlsRead` 将阻塞程序执行，等待缺失的片段。如果“原始”数据已包含准备好解密的块，则读取函数将返回比“原始”字节数少的已解密字节。因此，启用 TLS 时，建议始终将 `SocketIsReadable` 函数与 `SocketTlsReadAvailable` 结合使用。否则，程序的行为将与预期不同。遗憾的是，MQL5 没有提供与 TLS 模式兼容且不强加所述约定的 `SocketTlsIsReadable` 函数。

类似的 `SocketIsWritable` 函数会检查给定套接字当前是否可写入。

`bool SocketIsWritable(const int socket)`

函数返回成功 (`true`) 或错误 (`false`) 的指示。在后一种情况下，通过 `SocketConnect` 建立的连接将被断开。

这是一个简单的脚本 `SocketIsConnected.mq5`，用于测试这些函数。在输入参数中，我们将提供输入地址和端口的机会。

```
input string Server = "www.mql5.com";
input uint Port = 443;
```

在 `OnStart` 处理程序中，我们创建一个套接字，连接到站点，并开始循环中检查套接字的状态。在第二次迭代之后，我们强制关闭套接字，这应该会导致退出循环。

```

void OnStart()
{
    PRTF(Server);
    PRTF(Port);
    const int socket = PRTF(SocketCreate());
    if(PRTF(SocketConnect(socket, Server, Port, 5000)))
    {
        int i = 0;
        while(PRTF(SocketIsConnected(socket)) && !IsStopped())
        {
            PRTF(SocketIsReadable(socket));
            PRTF(SocketIsWritable(socket));
            Sleep(1000);
            if(++i >= 2)
            {
                PRTF(SocketClose(socket));
            }
        }
    }
}

```

以下条目显示在日志中。

```

Server=www.mql5.com / ok
Port=443 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,5000)=true / ok
SocketIsConnected(socket)=true / ok
SocketIsReadable(socket)=0 / ok
SocketIsWritable(socket)=true / ok
SocketIsConnected(socket)=true / ok
SocketIsReadable(socket)=0 / ok
SocketIsWritable(socket)=true / ok
SocketClose(socket)=true / ok
SocketIsConnected(socket)=false / NETSOCKET_INVALIDHANDLE(5270)

```

7.5.7 设置套接字数据发送和接收超时

由于网络连接不可靠，**Socket** 函数的所有操作都支持集中的超时设置。如果在指定时间内数据读取或发送未成功完成，函数将停止尝试执行相应的操作。

你可以使用 **SocketTimeouts** 函数设置接收和发送数据的超时。

bool SocketTimeouts(int socket, uint timeout_send, uint timeout_receive)

两个超时都以毫秒为单位给出，并在系统级别影响指定套接字上的所有函数。

其中 **SocketRead** 函数有其自己的 **timeout** 参数，你可以通过该参数在 **SocketRead** 函数的特定调用期间额外控制超时。

SocketTimeouts 如果成功则返回 **true**，否则返回 **false**。

默认情况下，没有超时，这意味着无限期等待所有数据被接收或发送。

7.5.8 通过非安全套接字连接读写数据

历史上,套接字默认通过简单连接提供数据传输。以开放形式传输数据支持以技术手段分析所有流量。近年来,安全问题越来越受到重视,因此几乎所有地方都采用了 TLS (传输层安全)技术;TLS 可对发送方和接收方之间的所有数据进行即时加密。特别地,对于互联网连接,区别在于 HTTP (简单连接)和 HTTPS (安全)协议。

MySQL 为处理简单连接和安全连接提供了不同组的 **Socket** 函数。在本节中,我们将熟悉简单模式,稍后我们将转向受保护的模式。

要从套接字读取数据,可使用 **SocketRead** 函数。

int SocketRead(int socket, uchar &buffer[], uint maxlen, uint timeout)

套接字描述符从 **SocketCreate** 获取,并使用 **Socket Connect**连接到网络资源。

buffer 参数是对数据将读入的数组的引用。如果数组是动态的,其大小会按读取的字节数增加,但不能超过 `INT_MAX` (2147483647)。你可以在 **maxlen** 参数中限制读取字节的数量。无法容纳的数据将保留在套接字的内部缓冲区中:它可以通过后续调用 **SocketRead** 来获取。**maxlen** 的值必须在 1 和 `INT_MAX` (2147483647) 之间。

timeout 参数指定等待读取完成的时间 (以毫秒为单位)。如果在此时间内未收到数据,则尝试终止,函数以结果 -1 退出。

发生错误时也返回 -1,而 **_LastError** 中的错误代码,例如 5273 (`ERR_NETSOCKET_IO_ERROR`),表示通过 **SocketConnect** 建立的连接现已断开。

如果成功,函数返回读取的字节数。

将读取超时设置为 0 时,使用默认值 120000 (2 分钟)。

要向套接字写入数据,请使用 **SocketSend** 函数。

遗憾的是,函数名 **SocketRead** 和 **SocketSend** 并不“对称”:“read”的反向操作是“write”,“send”的反向操作是“receive”。在其他平台上有网络 API 开发经验的开发者可能对此很陌生。

int SocketSend(int socket, const uchar &buffer[], uint maxlen)

第一个参数是先前创建和打开的套接字的句柄。传递无效句柄时,**_LastError** 会收到错误 5270 (`ERR_NETSOCKET_INVALIDHANDLE`)。**buffer** 数组包含要发送的数据,数据大小在 **maxlen** 参数中指定 (引入该参数是为了方便从固定数组发送部分数据)。

函数成功时返回写入套接字的字节数,错误时返回 -1。

系统级错误 (5273, `ERR_NETSOCKET_IO_ERROR`) 表示断开连接。

SocketReadWriteHTTP.mq5 脚本演示了如何使用套接字通过 HTTP 协议来执行工作,即从 Web 服务器请求有关页面的信息。这是 **WebRequest** 函数在后台为我们执行的一部分工作。

我们在输入参数中保留默认地址:站点“`www.mql5.com`”。选择端口号 80,因为这是非安全 HTTP 连接的默认值 (尽管某些服务器可能使用不同的端口:81、8080 等)。此示例尚不支持为安全连接保留的端口 (特别是最常用的 443)。此外,在 **Server** 参数中,重要的是输入域名而不是特定页面,因为该脚本只能请求主页面,即根路径“/”。


```
input string Server = "www.mql5.com";
input uint Port = 80;
```

在脚本的主函数中，我们将创建一个套接字并使用指定的参数在其上打开一个连接（超时为 5 秒）。

```
void OnStart()
{
    PRTF(Server);
    PRTF(Port);
    const int socket = PRTF(SocketCreate());
    if(PRTF(SocketConnect(socket, Server, Port, 5000)))
    {
        ...
    }
}
```

我们看一下 HTTP 协议是如何工作的。客户端以特殊设计的标头（具有预定义名称和值的字符串）的形式发送请求，其中特别包括网页地址，而服务器则以整个网页或操作状态作为响应，同样也使用特殊的标头。客户端可以使用 GET 请求网页，使用 POST 请求发送一些数据，或者使用精简的 HEAD 请求检查网页状态。理论上，还有更多 HTTP 方法—你可以在 HTTP 协议规范中详细了解。

因此，脚本必须通过套接字连接生成并发送一个 HTTP 标头。在最简单的形式中，下面的 HEAD 请求允许你获取页面的元信息（我们可以用 GET 代替 HEAD 来请求整个页面，但会有一些复杂的问题，我们稍后再讨论）。

```
HEAD / HTTP/1.1
Host: _server_
User-Agent: MetaTrader 5

// <- two newlines in a row \r\n\r\n
```

“HEAD”（或其他方法）之后的正斜杠是任何服务器上到根目录的最短可能路径，这通常会导致显示主页。如果我们想要一个特定的网页，我们可以写入类似“GET /en/forum / HTTP/1.1”的内容，并从 *mql5.com* 获取英文论坛的目录。请用实际域名替换“_server_”字符串。

尽管“User-Agent:”标头是可选的，但这个标头允许程序向服务器进行自我介绍，若没有它，某些服务器可能会拒绝请求。

注意两个空行：它们标记了标头的结束。在我们的脚本中，用以下表达式形成标题很方便：

```
StringFormat("HEAD / HTTP/1.1\r\nHost: %s\r\n\r\n", Server)
```

现在，我们只需将其发送到服务器。为此，我们编写了一个简单的函数 *HTTPSend*。它接收一个套接字描述符和一个标头行。

```
bool HTTPSend(int socket, const string request)
{
    char req[];
    int len = StringToCharArray(request, req, 0, WHOLE_ARRAY, CP_UTF8) - 1;
    if(len < 0) return false;
    return SocketSend(socket, req, len) == len;
}
```

在内部，我们将字符串转换为字节数组并调用 *SocketSend*。

接下来，我们需要接收服务器的响应，为此我们编写了 `HTTPRecv` 函数。它也需要一个套接字描述符和对一个应放置数据的字符串的引用，但更为复杂。

```
bool HTTPRecv(int socket, string &result, const uint timeout)
{
    char response[];
    int len;          // signed integer needed for error flag -1
    uint start = GetTickCount();
    result = "";

    do
    {
        ResetLastError();
        if(!(len = (int)SocketIsReadable(socket)))
        {
            Sleep(10); // wait for data or timeout
        }
        else          // read the data in the available volume
        if((len = SocketRead(socket, response, len, timeout)) > 0)
        {
            result += CharArrayToString(response, 0, len); // NB: without CP_UTF8 only '
            const int p = StringFind(result, "\r\n\r\n");
            if(p > 0)
            {
                // HTTP header ends with a double newline, use this
                // to make sure the entire header is received
                Print("HTTP-header found");
                StringSetLength(result, p); // cut off the body of the document (in case
                return true;
            }
        }
    }
    while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);

    if(_LastError) PRTF(_LastError);

    return StringLen(result) > 0;
}
```

在这里，我们在循环中检查指定超时内出现的数据，并将其读入 `response` 缓冲区。发生错误会终止循环。

缓冲区字节立即转换成字符串并连接到 `result` 变量中的完整响应。务必注意，对于 HTTP 标头，我们只能使用具有默认编码的 `CharArrayToString` 函数，因为它只允许使用拉丁字母和 ANSI 中的一些特殊字符。

完整的 Web 文档通常采用 UTF-8 编码（但可能采用其他非拉丁编码，这恰好在 HTTP 标头中指明），要接收完整的 Web 文档，将需要更复杂的处理：首先，你需要将所有发送的块收集到一个公共缓冲区中，然后指明 CP_UTF8 将整个内容转换为字符串（否则，任何以双字节编码的字符在发送时都可能被“切割”，并分别到达不同的块中；这就是为什么我们不能期望在单个片段中获得正确的 UTF-8 字节流）。我们将在以下部分改进此示例。

有了 `HTTPSend` 和 `HTTPRecv` 函数，我们就完成了 `OnStart` 代码。

```

void OnStart()
{
    ...
    if(PRTF(HTTPSend(socket, StringFormat("HEAD / HTTP/1.1\r\nHost: %s \r\n"
        "User-Agent: MetaTrader 5\r\n\r\n", Server))))
    {
        string response;
        if(PRTF(HTTPRecv(socket, response, 5000)))
        {
            Print(response);
        }
    }
    ...
}

```

在从服务器接收到的 HTTP 标头中，我们可能会对以下几行感兴趣：

- 'Content-Length:'-文档总长度，以字节为单位
- 'Content-Language:'-文档语言（例如，“de-DE, ru”）
- 'Content-Type:'-文档编码（例如，“text/html; charset=UTF-8”）
- 'Last-Modified:'-文档的最后修改时间，这样就不会下载已经存在的内容（原则上，我们可以在 HTTP 请求中添加 'If-Modified-Since:' 标头）

我们将更详细地讨论如何确定文档长度（数据大小），因为几乎所有标头都是可选的，也就是说，它们是由服务器随意报告的，在它们缺失的情况下，会使用替代机制。知道大小对于何时关闭连接非常重要，即确保所有数据都已接收。

使用默认参数运行该脚本会产生以下结果。

```

Server=www.mql5.com / ok
Port=80 / ok
SocketCreate()==1 / ok
SocketConnect(socket,Server,Port,5000)=true / ok
HTTPSend(socket,StringFormat(HEAD / HTTP/1.1
Host: %s
,Server))=true / ok
HTTP-header found
HTTPRecv(socket,response,5000)=true / ok
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Sun, 31 Jul 2022 10:24:00 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: https://www.mql5.com/
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
X-Frame-Options: SAMEORIGIN

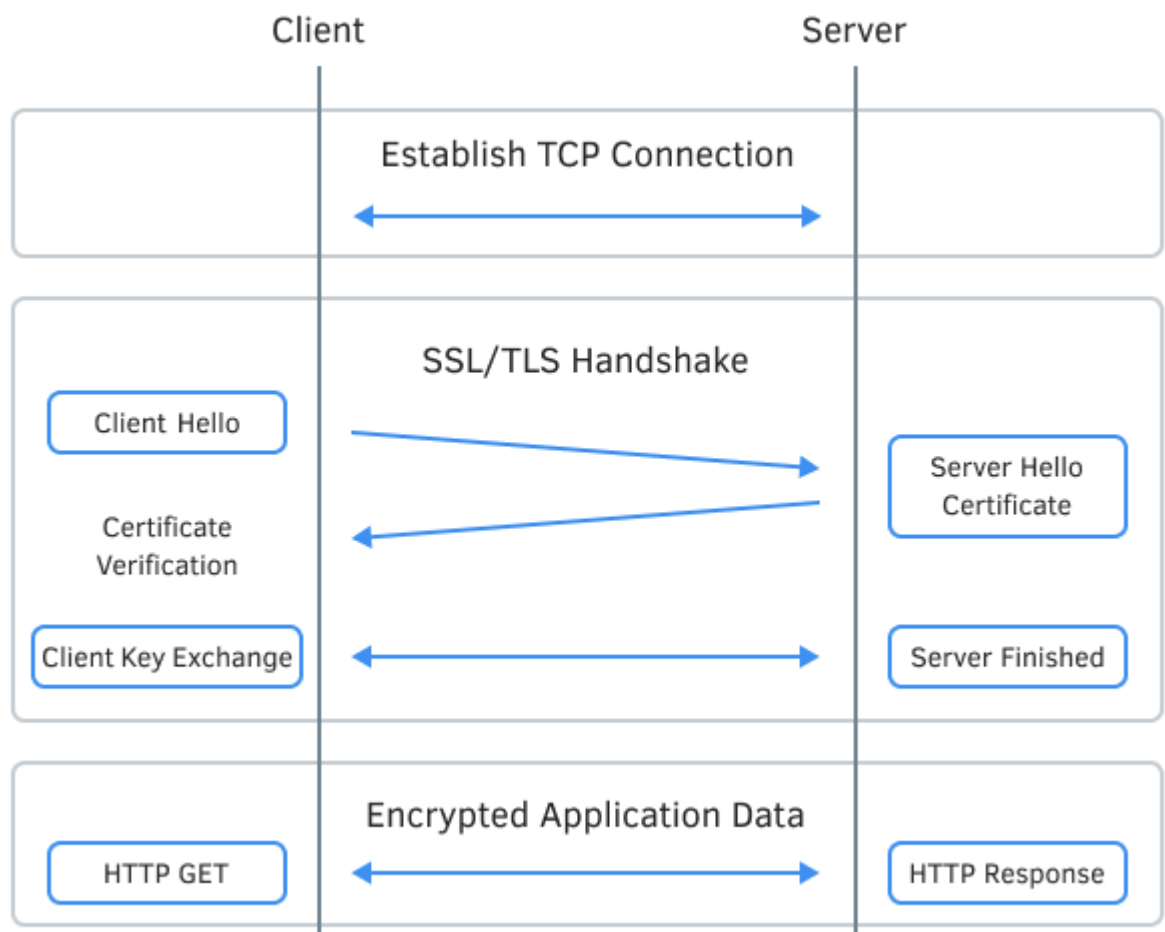
```

请注意，这个站点，像今天大多数站点一样，会将我们的请求重定向到一个安全连接：这是通过状态码 “301 Moved Permanently” 和新地址 “Location: https://www.mql5.com/” 实现的（这里的 “https” 协议很重要）。要重试一个启用了 TLS 的请求，必须使用其他一些函数，我们稍后会讨论它们。

7.5.9 准备安全套接字连接

为了将套接字连接转换为受保护状态并进行检查，MQL6 提供了以下函数：*SocketTlsHandshake* 和 *SocketTlsCertificate*。通常情况下，如果连接是建立在 443 端口上的，我们不需要通过调用 *SocketTlsHandshake* 来“手动”启用保护。事实是，这对于 HTTPS (TLS) 来说是标准的。

保护基于客户端和服务端之间数据流的加密，最初使用一对非对称密钥：公钥和私钥。我们已经在 [可用信息转换方法概述](#) 一节中讨论过这个主题。每个正规网站都会从网络社区信任的某个证书颁发机构 (CA) 获取数字证书。该证书包含网站的公钥，并由该中心数字签名。浏览器和其他客户端应用程序会存储 (也可以导入)CA 的公钥，因此可以验证特定证书的质量。



建立安全的 TLS 连接
(图片来自互联网)

此外，在准备安全连接时，浏览器或应用程序会生成某个“秘密”，用站点的公钥加密这个秘密并将其发送给站点，站点则用只有自己知道的私钥解密秘密。这个阶段在实践中看起来更复杂，但最终将确保客户端和服务端都拥有当前会话 (连接) 的加密密钥。通信的双方都使用此密钥在一端加密后续的请求和响应，并在另一端解密它们。

SocketTlsHandshake 函数使用 TLS 握手协议启动与指定主机的安全 TLS 连接。在这种情况下，客户端和服务端就连接参数达成一致：所用协议的版本和数据加密方法。

`bool SocketTlsHandshake(int socket, const string host)`

函数参数中传递了套接字句柄和建立连接的服务器地址 (实际上，这与在 *SocketConnect* 中指定的名称相同)。

在建立安全连接之前，程序必须首先使用 `SocketConnect` 与主机建立常规的 TCP 连接。

函数成功时返回 `true`；否则返回 `false`。如果发生错误，代码 5274 (`ERR_NETSOCKET_HANDSHAKE_FAILED`) 会被写入 `_LastError`。

`SocketTlsCertificate` 函数获取用于保护网络连接的证书信息。

```
int SocketTlsCertificate(int socket, string &subject, string &issuer, string &serial, string
&thumbprint, datetime &expiration)
```

如果已为该套接字建立了安全连接（无论是在显式且成功的 `SocketTlsHandshake` 调用之后，还是在通过 443 端口连接之后），此函数会通过套接字描述符将相应信息填充到所有其他引用变量中：证书所有者名称 (`subject`)、证书颁发者名称 (`issuer`)、序列号 (`serial`)、数字指纹 (`thumbprint`) 和证书有效期 (`expiration`)。

函数在成功收到证书相关信息时返回 `true`，如果出错则返回 `false`。错误代码是 5275 (`ERR_NETSOCKET_NO_CERTIFICATE`)。这可以用来确定通过 `SocketConnect` 打开的连接是否立即处于受保护模式。我们将在下一节的示例中运用这一点。

7.5.10 通过安全套接字连接读写数据

安全连接自身有一套用于在客户端和服务端之间交换数据的函数。这些函数的名称和操作概念与先前讨论的函数 `SocketRead` 和 `SocketSend` 几乎一致。

```
int SocketTlsRead(int socket, uchar &buffer[], uint maxlen)
```

`SocketTlsRead` 函数从指定套接字上打开的安全 TLS 连接中读取数据。数据进入通过引用传递的 `buffer` 数组。如果它是动态的，其大小将根据数据量增加，但不会超过 `INT_MAX` (2147483647) 字节。

`maxlen` 参数指定要接收的已解密字节数（它们的数量总是小于进入套接字内部缓冲区的“原始”加密数据量）。数组无法容纳的数据保留在套接字中，并可以通过下一次 `SocketTlsRead` 调用接收。

该函数将一直执行，直到接收到指定数量的数据或达到 `SocketTimeouts` 中指定的超时。

成功时，函数返回读取的字节数；错误时，返回 -1，同时 `_LastError` 中写入代码 5273 (`ERR_NETSOCKET_IO_ERROR`)。出现错误表示连接已终止。

```
int SocketTlsReadAvailable(int socket, uchar &buffer[], const uint maxlen)
```

`SocketTlsReadAvailable` 函数从安全 TLS 连接中读取所有可用的已解密数据，但最多不超过 `maxlen` 字节。与 `SocketTlsRead` 不同，`SocketTlsReadAvailable` 不会等待必须存在给定数量的数据，而是立即返回当前存在的数据。因此，如果套接字的内部缓冲区是“空的”（尚未从服务器接收到任何内容，或者已经读取或尚未形成准备解密的块），函数将返回 0，并且不会在接收数组 `buffer` 中记录任何内容。这是正常情况。

`maxlen` 的值必须在 1 和 `INT_MAX` (2147483647) 之间。

```
int SocketTlsSend(int socket, const uchar &buffer[], uint bufferlen)
```

`SocketTlsSend` 函数通过在指定套接字上打开的安全连接从 `buffer` 数组发送数据。其操作原理与先前描述的函数 `SocketSend` 相同，唯一的区别在于连接类型。

我们基于先前讨论的 *SocketReadWriteHTTP.mq5* 创建一个新脚本 *SocketReadWriteHTTPS.mq5*, 并在选择 HTTP 方法 (默认为 GET, 而非 HEAD)、设置超时和支持安全连接方面增加灵活性。默认端口是 443。

```
input string Method = "GET"; // Method (HEAD,GET)
input string Server = "www.google.com";
input uint Port = 443;
input uint Timeout = 5000;
```

默认服务器是 www.google.com。不要忘记将其 (以及你输入的任何其他服务器) 添加到终端设置中允许的列表中。

为了确定连接是否安全, 我们将使用 *SocketTlsCertificate* 函数: 如果该函数成功, 则表明服务器已提供证书并且 TLS 模式已激活。如果该函数返回 *false* 并抛出错误代码 NETSOCKET_NO_CERTIFICATE(5275), 这意味着我们正在使用普通连接, 但由于我们对非安全连接感到满意, 因此可以忽略并重置该错误。

```
void OnStart()
{
    PRTF(Server);
    PRTF(Port);
    const int socket = PRTF(SocketCreate());
    if(socket == INVALID_HANDLE) return;
    SocketTimeouts(socket, Timeout, Timeout);
    if(PRTF(SocketConnect(socket, Server, Port, Timeout)))
    {
        string subject, issuer, serial, thumbprint;
        datetime expiration;
        bool TLS = false;
        if(PRTF(SocketTlsCertificate(socket, subject, issuer, serial, thumbprint, expiration)))
        {
            PRTF(subject);
            PRTF(issuer);
            PRTF(serial);
            PRTF(thumbprint);
            PRTF(expiration);
            TLS = true;
        }
        ...
    }
}
```

OnStart 函数的其余部分按照先前的计划实现: 使用 *HTTPSend* 函数发送请求, 并使用 *HTTPRecv* 函数接收应答。但这一次, 我们额外将 TLS 标志传递给这些函数, 并且它们的实现必须略有不同。

```

if(PRTF(HTTPSend(socket, StringFormat("%s / HTTP/1.1\r\nHost: %s\r\n"
    "User-Agent: MetaTrader 5\r\n\r\n", Method, Server), TLS)))
{
    string response;
    if(PRTF(HTTPRecv(socket, response, Timeout, TLS)))
    {
        Print("Got ", StringLen(response), " bytes");
        // for large documents, we will save to a file
        if(StringLen(response) > 1000)
        {
            int h = FileOpen(Server + ".htm", FILE_WRITE | FILE_TXT | FILE_ANSI, 0);
            FileWriteString(h, response);
            FileClose(h);
        }
        else
        {
            Print(response);
        }
    }
}

```

从 *HTTPSend* 的示例中可以看出，根据 TLS 标志，我们使用 *SocketTlsSend* 或 *SocketSend*。

```

bool HTTPSend(int socket, const string request, const bool TLS)
{
    char req[];
    int len = StringToCharArray(request, req, 0, WHOLE_ARRAY, CP_UTF8) - 1;
    if(len < 0) return false;
    return (TLS ? SocketTlsSend(socket, req, len) : SocketSend(socket, req, len)) == 1
}

```

HTTPRecv 的情况要复杂一些。由于我们提供了下载整个页面（不仅仅是标头）的功能，我们需要某种方法来知道是否已接收到所有数据。即使在整个文档传输完毕后，套接字通常也会保持打开状态，以优化未来预期的请求。但我们的程序将不知道传输是正常停止，还是网络基础设施中某处可能出现了暂时的“拥塞”（有时可以在浏览器中观察到这种宽松的、间歇性的页面加载）。或者相反，在连接失败的情况下，我们可能错误地认为我们已经接收了整个文档。

事实是，套接字本身仅作为程序之间的通信手段，并处理抽象的数据块：它们不知道数据的类型、含义及其逻辑结束。所有这些问题都由像 HTTP 这样的应用层协议处理。因此，我们需要深入研究规范并自己实现检查。

```

bool HTTPRecv(int socket, string &result, const uint timeout, const bool TLS)
{
    uchar response[]; // accumulate the data as a whole (headers + body of the web doc
    uchar block[];     // separate read block
    int len;           // current block size (signed integer for error flag -1)
    int lastLF = -1;   // position of the last line feed found LF(Line-Feed)
    int body = 0;      // offset where document body starts
    int size = 0;      // document size according to title
    result = "";       // set an empty result at the beginning
    int chunk_size = 0, chunk_start = 0, chunk_n = 1;
    const static string content_length = "Content-Length:";
    const static string crlf = "\r\n";
    const static int crlf_length = 2;
    ...
}

```

确定接收数据大小的最简单方法是基于对“Content-Length:”标头的分析。这里我们需要三个变量：*lastLF*、*size* 和 *content_length*。不过，这个标头并不一定会存在，我们还需要处理“块”，也就是为此引入了变量 *chunk_size*、*chunk_start*、*crlf* 和 *crlf_length* 来检测块。

为了演示各种接收数据的技术，我们在此示例中使用了一个“非阻塞”函数

SocketTlsReadAvailable。但是，对于非安全连接，没有类似的函数，因此我们必须自己编写它（稍后会介绍）。算法的总体方案很简单：它是一个循环，尝试接收大小为 1024（或更少）字节的新数据块。如果我们成功读取到一些内容，就将其累积到 *response* 数组中。如果套接字的输入缓冲区为空，函数将返回 0，我们会暂停一小会儿。最后，如果发生错误或超时，循环将中断。

```

uint start = GetTickCount();
do
{
    ResetLastError();
    if((len = (TLS ? SocketTlsReadAvailable(socket, block, 1024) :
        SocketReadAvailable(socket, block, 1024))) > 0)
    {
        const int n = ArraySize(response);
        ArrayCopy(response, block, n); // put all the blocks together
        ...
        // main operation here
    }
    else
    {
        if(len == 0) Sleep(10); // wait a bit for the arrival of a portion of data
    }
}
while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);
...

```

首先，你需要等待输入数据流中 HTTP 标头的完成。正如我们从前面的示例中已经看到的，标头通过双换行符（即字符序列 `\r\n\r\n`）与文档主体分隔。通过定位两个相继出现的 `'\n'` (LF) 符号很容易检测到它。

搜索的结果将是数据开始到标头结束、文档开始处的字节偏移量。我们将把它存储在变量 *body* 中。


```

if(body == 0) // look for the completion of the headers until we find it
{
    for(int i = n; i < ArraySize(response); ++i)
    {
        if(response[i] == '\n') // LF
        {
            if(lastLF == i - crlf_length) // found sequence "\r\n\r\n"
            {
                body = i + 1;
                string headers = CharArrayToString(response, 0, i);
                Print("* HTTP-header found, header size: ", body);
                Print(headers);
                const int p = StringFind(headers, content_length);
                if(p > -1)
                {
                    size = (int)StringToInteger(StringSubstr(headers,
                        p + StringLen(content_length)));
                    Print("* ", content_length, size);
                }
                ...
                break; // header/body boundary found
            }
            lastLF = i;
        }
    }
}

if(size == ArraySize(response) - body) // entire document
{
    Print("* Complete document");
    break;
}
...

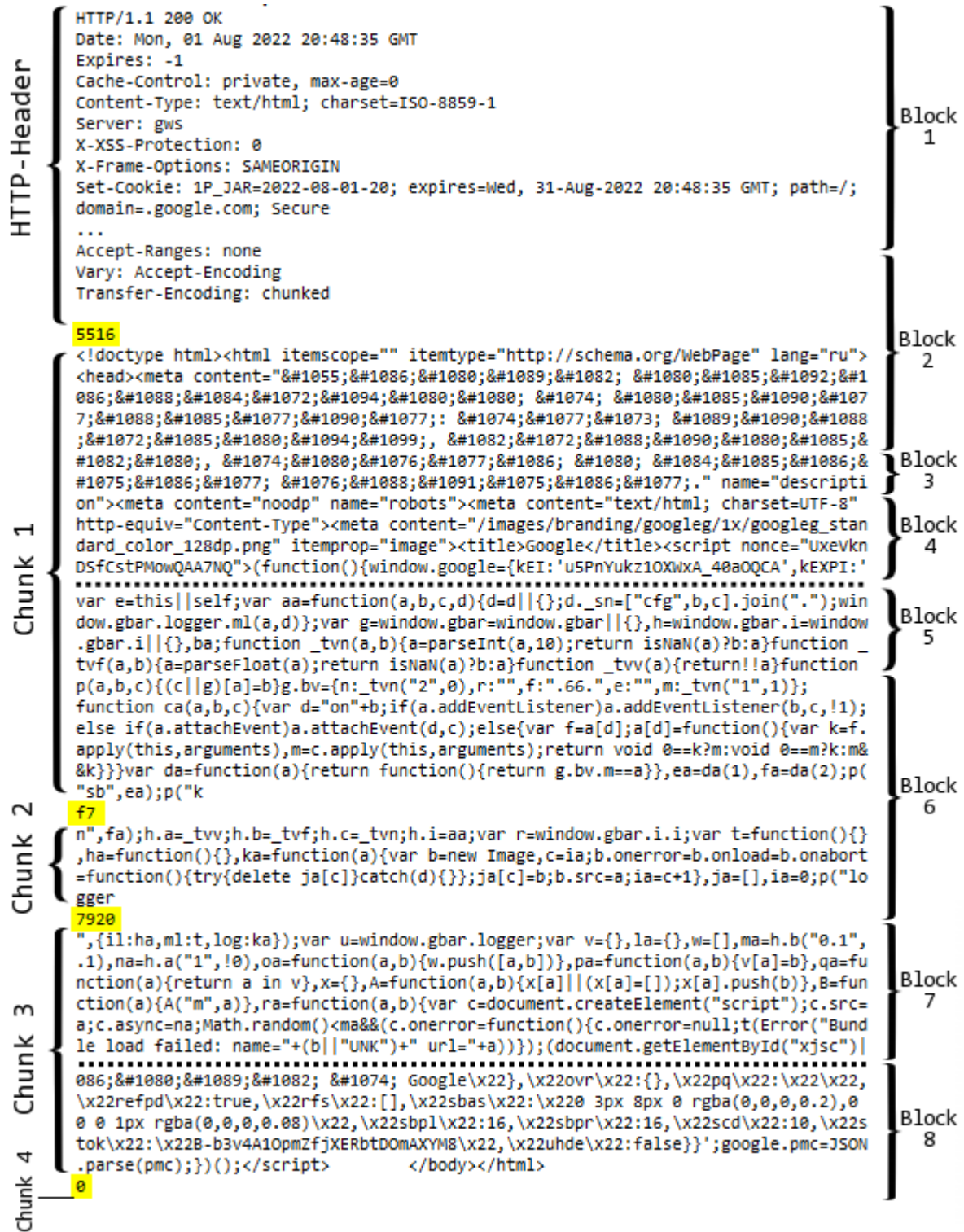
```

这会立即搜索 “Content-Length:” 标头并从中提取大小。填充的 **size** 变量使得可以在接收到整个文档时编写一个额外的条件语句来退出数据接收循环。

一些服务器以称为“块”的部分形式提供内容。在这种情况下，HTTP 标头中缺少 “Transfer-Encoding: chunked” 行，而 “Content-Length:” 行。每个块以一个十六进制数开始，指示块的大小，后跟一个换行符和指定数量的数据字节。块以另一个换行符结束。标记文档结束的最后一个块的大小为零。

请注意，这种分段是由服务器自主执行，依据是服务器自身的当前的优化发送“偏好”，与在套接字级别为通过网络传输而将信息划分成的数据块（包）无关。换句话说，块往往会被任意分段，网络包之间的边界甚至可能出现在块大小的数字之间。

具体描述如下（左边是文档的块，右边是来自套接字缓冲区的数据块）。



网页文档在 HTTP 和 TCP 层面传输过程中的分段方式

在我们的算法中，数据包在每次迭代时进入 *block* 数组，但逐个分析它们没有意义，所有主要工作都是针对共同的 *response* 数组进行的。

因此，如果 HTTP 标头已完全接收但其中未找到字符串 “Content-Length:”，我们会改为采用了 “Transfer-Encoding: chunked” 模式的算法分支。通过 *response* 数组中 *body* 的当前位置（紧随 HTTP 标头完成之后），选定字符串片段并假设为十六进制格式转换为数字：这是由辅助函数 *HexStringToInteger* 完成的（参见随附的源代码）。如果确实是一个数字，我们将其写入

`chunk_size`, 在 `chunk_start` 中将该位置标记为“块”的开始, 并从 `response` 中移除带有数字和框架换行符的字节。

```
...
if(lastLF == i - crlf_length) // found sequence "\r\n\r\n"
{
    body = i + 1;
    ...
    const int p = StringFind(headers, content_length);
    if(p > -1)
    {
        size = (int)StringToInteger(StringSubstr(headers,
            p + StringLen(content_length)));
        Print("* ", content_length, size);
    }
    else
    {
        size = -1; // server did not provide document length
        // try to find chunks and the size of the first one
        if(StringFind(headers, "Transfer-Encoding: chunked") > 0)
        {
            // chunk syntax:
            // <hex-size>\r\n<content>\r\n...
            const string preview = CharArrayToString(response, body, 2);
            chunk_size = HexStringToInteger(preview);
            if(chunk_size > 0)
            {
                const int d = StringFind(preview, crlf) + crlf_length;
                chunk_start = body;
                Print("Chunk: ", chunk_size, " start at ", chunk_start,
                    ArrayRemove(response, body, d));
            }
        }
    }
    break; // header/body boundary found
}
lastLF = i;
...
```

现在, 要检查文档的完整性, 你不仅需要分析 `size` 变量 (正如我们所见, 在没有 “Content-Length:” 的情况下, 可以通过赋 -1 来实际禁用它), 还需要分析用于块的新变量: `chunk_start` 和 `chunk_size`。操作方案与 HTTP 标头之后相同: 通过 `response` 数组中的偏移量 (前一个块结束的地方), 我们分离出下一个“块”的大小。我们继续这个过程, 直到找到大小为零的块。

```

...
if(size == ArraySize(response) - body) // entire document
{
    Print("* Complete document");
    break;
}
else if(chunk_size > 0 && ArraySize(response) - chunk_start >= chunk_size)
{
    Print("* ", chunk_n, " chunk done: ", chunk_size, " total: ", ArraySize(r
const int p = chunk_start + chunk_size;
const string preview = CharArrayToString(response, p, 20);
if(StringLen(preview) > crlf_length // there is '\r\n...\r\n'
    && StringFind(preview, crlf, crlf_length) > crlf_length)
{
    chunk_size = HexStringToInteger(preview, crlf_length);
    if(chunk_size > 0)
    {
        // twice '\r\n': before and after chunk
        int d = StringFind(preview, crlf, crlf_length) + crlf_length;
        chunk_start = p;
        Print("Chunk: ", chunk_size, " start at ", chunk_start, " -", d);
        ArrayRemove(response, chunk_start, d);
        ++chunk_n;
    }
    else
    {
        Print("* Final chunk");
        ArrayRemove(response, p, 5); // "\r\n0\r\n"
        break;
    }
} // otherwise wait for more data
}

```

因此，我们通过分析传入流的结果，以两种不同的方式提供了退出循环的条件（此外还有通过超时和错误退出）。在循环正常结束时，我们将数组中从 *body* 位置开始并包含整个文档的那部分转换到 *response* 字符串中。否则，我们简单地返回我们设法获取的所有内容，包括标头，以供“分析”。

```

bool HTTPRecv(int socket, string &result, const uint timeout, const bool TLS)
{
    ...
    do
    {
        ResetLastError();
        if((len = (TLS ? SocketTlsReadAvailable(socket, block, 1024) :
            SocketReadAvailable(socket, block, 1024))) > 0)
        {
            ... // main operation here - discussed above
        }
        else
        {
            if(len == 0) Sleep(10); // wait a bit for the arrival of a portion of data
        }
    }
    while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);

    if(_LastError) PRTF(_LastError);

    if(ArraySize(response) > 0)
    {
        if(body != 0)
        {
            // TODO: Desirable to check 'Content-Type:' for 'charset=UTF-8'
            result = CharArrayToString(response, body, WHOLE_ARRAY, CP_UTF8);
        }
        else
        {
            // to analyze wrong cases, return incomplete headers as is
            result = CharArrayToString(response);
        }
    }

    return StringLen(result) > 0;
}

```

唯一剩下的函数是 *SocketReadAvailable*，它类似于 *SocketTlsReadAvailable*，用于非安全连接。

```

int SocketReadAvailable(int socket, uchar &block[], const uint maxlen = INT_MAX)
{
    ArrayResize(block, 0);
    const uint len = SocketIsReadable(socket);
    if(len > 0)
        return SocketRead(socket, block, fmin(len, maxlen), 10);
    return 0;
}

```

该脚本已准备就绪。

使用套接字实现一个简单的网页请求花费了我们相当大的力气。这展示了在底层支持网络协议通常隐藏了大量繁琐的工作。当然，对于 HTTP，我们使用内置的 *WebRequest* 实现更容易也更正确，但它并不包含 HTTP 的所有功能（此外，我们顺便提到了 HTTP 1.1，但还有 HTTP/2），

而且还有其它应用层协议，数量极为庞大。因此，需要在 MetaTrader 5 中使用 *Socket* 函数来集成它们。

我们使用默认设置运行 *SocketReadWriteHTTPS.mq5*。

```

Server=www.google.com / ok
Port=443 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,Timeout)=true / ok
SocketTlsCertificate(socket,subject,issuer,serial,thumbprint,expiration)=true / ok
subject=CN=www.google.com / ok
issuer=C=US, O=Google Trust Services LLC, CN=GTS CA 1C3 / ok
serial=00c9c57583d70aa05d12161cde9ee32578 / ok
thumbprint=1EEE9A574CC92773EF948B50E79703F1B55556BF / ok
expiration=2022.10.03 08:25:10 / ok
HTTPSend(socket,StringFormat(%s / HTTP/1.1
Host: %s
,Method,Server),TLS)=true / ok
* HTTP-header found, header size: 1080
HTTP/1.1 200 OK
Date: Mon, 01 Aug 2022 20:48:35 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2022-08-01-20; expires=Wed, 31-Aug-2022 20:48:35 GMT;
    path=/; domain=.google.com; Secure
...
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked
Chunk: 22172 start at 1080 -6
* 1 chunk done: 22172 total: 24081
Chunk: 30824 start at 23252 -8
* 2 chunk done: 30824 total: 54083
* Final chunk
HTTPRecv(socket,response,Timeout,TLS)=true / ok
Got 52998 bytes

```

正如我们所见，文档是以块的形式传输的，并已保存到一个临时文件中（你可以在 *MQL5/Files/www.mql5.com.htm* 中找到它）。

现在，我们为站点 “*www.mql5.com*” 和端口 80 运行该脚本。从上一节我们知道，在这种情况下，该站点会发出一个到其受保护版本的重定向，但这个“重定向”并非空的：它有一个存根文档，现在我们可以完整地获取它。对我们而言，在这种情况下正确使用了 “Content-Length:” 标头事关重大。

```

Server=www.mql5.com / ok
Port=80 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,Timeout)=true / ok
HTTPSend(socket,StringFormat(%s / HTTP/1.1
Host: %s
,Method,Server),TLS)=true / NETSOCKET_NO_CERTIFICATE(5275)
* HTTP-header found, header size: 291
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Sun, 31 Jul 2022 19:28:57 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: https://www.mql5.com/
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
X-Frame-Options: SAMEORIGIN
* Content-Length:162
* Complete document
HTTPRecv(socket,response,Timeout,TLS)=true / ok
<html>
<head><title>301 Moved Permanently</title></head>
<body>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx</center>
</body>
</html>

```

另一个在实践中使用套接字的更大规模示例，我们将在 [项目一章](#) 中讨论。

7.6 SQLite 数据库

MetaTrader 5 原生支持 SQLite 数据库。它是一款轻量级但功能齐全的数据库管理系统 (DBMS)。传统上，此类系统专注于处理数据表，数据表存储着具有一组共同属性的同类型记录，并且不同类型 (即不同表) 的记录之间可以建立各种对应关系 (链接或关联)，因此这类数据库也被称为关系型数据库。我们已经探讨过 [经济日历](#) 结构之间此类连接的示例，但日历数据库存储在终端内部，而本节的函数将允许你从 MQL 程序创建任意数据库。

DBMS 在这些数据结构上的专业化使其能够优化 (加速和简化) 许多常用操作，例如对大量数据进行排序、搜索、筛选、汇总或计算其他聚合函数。

然而，这也有另一面：DBMS 编程需要其自身的 SQL (结构化查询语言)，仅掌握纯 MQL5 是不够的。与 MQL5 (属于 *imperative* 语言，即使用运算符指示做什么、如何做、按什么顺序做) 不同，SQL 是 *declarative*，也就是说，它描述初始数据和期望的结果，而不指定如何以及按什么顺序执行计算。SQL 中算法的含义以 SQL 查询的形式描述。查询类似于一个单独的 MQL5 运算符，是使用特殊语法形成的字符串。

我们无需编写复杂的循环和比较，只需通过向 SQLite 函数 (例如 [DatabaseExecute](#) 或 [DatabasePrepare](#)) 传递 SQL 查询即可调用它们。要将查询结果获取到现成的 MQL5 结构体中，你可以使用 [DatabaseReadBind](#) 函数。这样，你可以在一次调用中读取记录 (结构体) 的所有字段。

借助数据库函数，可以轻松创建表、向其中添加记录、进行修改，并根据复杂条件进行选择，例如，用于以下任务：

- 获取交易历史和报价
- 保存优化和测试结果
- 准备数据并与其他分析软件包交换
- 分析经济日历数据
- 存储 MQL5 程序的设置和状态

此外，SQL 查询中还可以使用广泛的通用、统计和数学函数。而且，即使不创建表，也可以计算包含它们的表达式。

SQLite 不需要单独的应用程序、配置和管理，对资源要求不高，并支持流行的 SQL92 标准的大多数命令。另一个便利之处在于，整个数据库驻留在用户计算机硬盘上的单个文件中，并且可以轻松传输或备份。然而，为了加快读、写和修改操作，也可以使用 `DATABASE_OPEN_MEMORY` 标志在 RAM 中打开/创建数据库，但在这种情况下，这样的数据库将仅对此特定程序可用，不能用于多个程序的协同工作。

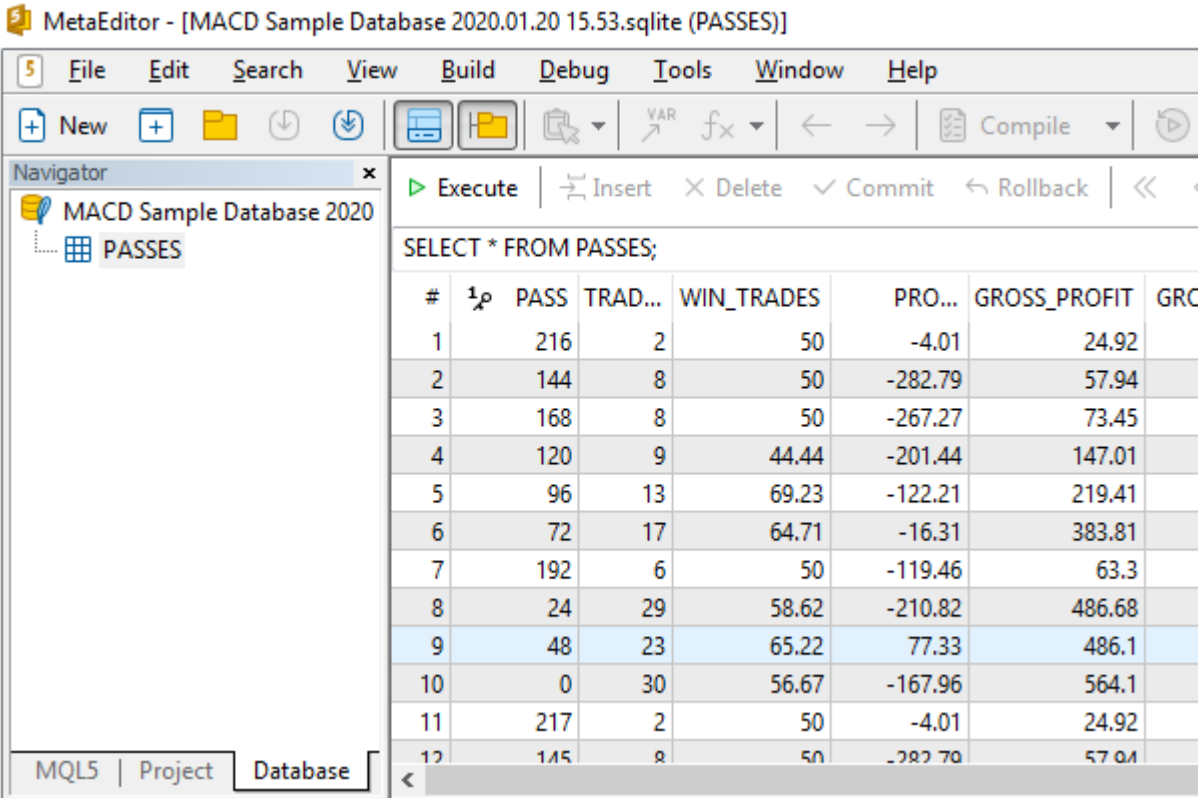
务必注意，与功能齐全的 DBMS 相比，SQLite 的相对简单性也带来了一些限制。特别是，SQLite 没有一个专门的进程（系统服务或应用程序）来提供对数据库和表管理 API 的集中访问，这就是为什么不能保证不同进程对同一数据库（文件）的并行、共享访问。因此，如果需要从执行相同 EA 交易实例的优化代理同时对数据库进行读写，将需要在其中编写代码来同步访问（否则，正在写入和读取的数据将处于不一致状态：毕竟，并发非同步进程的写入、修改、删除和读取顺序是随机的）。此外，同时更改数据库的尝试可能会导致 MQL 程序收到“数据库繁忙”的错误（并且不会执行请求的操作）。唯一不需要同步 SQLite 并行操作的场景是仅涉及读取操作时。

我们将仅介绍开始应用 SQL 所必需的基础知识。SQL 语法和工作原理的完整描述超出了本书的范围。请查阅 SQLite 站点的文档。然而，请注意 MQL5 和 MetaEditor 仅支持命令和 [SQL 语法结构](#) 的有限子集。

MQL Wizard MetaEditor 中的向导有一个用于创建数据库的嵌入式选项，它会立即提议通过定义字段列表来创建第一个表。此外，**Navigator** 提供了一个用于处理数据库的单独选项卡。

使用 **Wizard** 或 **Navigator** 的上下文菜单，你可以创建一个包含受支持格式（*.db、*.sql、*.sqlite 等）的空数据库（磁盘上文件，默认情况下放置在 **MQL5/Files** 目录中）。此外，在上下文菜单中，你可以从 sql 文件导入整个数据库，或从 csv 文件导入单个表。

现有或创建的数据库可以通过同一菜单轻松打开。之后，其表将出现在 **Navigator** 中，窗口的右侧主区域将显示一个包含用于调试 SQL 查询的工具和结果表的面板。例如，双击表名会执行对所有记录字段的快速查询，这对应于顶部输入字段中出现的“SELECT *FROM 'table'”语句。



在 MetaEditor 中查看 SQLite 数据库

你可以编辑请求并单击 **Execute** 按钮以激活它。可能的 SQL 语法错误会输出到日志中。

有关 **Wizard**、数据库的导入/导出以及与它们进行交互式工作的更多详细信息，请参阅 [MetaEditor 文档](#)。

7.6.0 MQL5 中的数据库操作原理

数据库以表的形式存储信息。获取、修改和向其中添加新数据都是使用 SQL 语言查询完成的。我们将在以下章节中描述其具体细节。与此同时，我们借助与交易无关的 **DatabaseRead.mq5** 脚本，了解如何创建一个简单的数据库并从中获取信息。这里提到的所有函数稍后都会详细描述。现在一定要想象一下总体原则。

使用内置的 **DatabaseOpen/DatabaseClose** 函数创建和关闭数据库，类似于处理文件，因为我们也为数据库创建一个描述符，检查数据库，并在最后将其关闭。

```

void OnStart()
{
    string filename = "company.sqlite";
    // create or open a database
    int db = DatabaseOpen(filename, DATABASE_OPEN_READWRITE | DATABASE_OPEN_CREATE);
    if(db == INVALID_HANDLE)
    {
        Print("DB: ", filename, " open failed with code ", _LastError);
        return;
    }
    ...// further work with the database
    // close the database
    DatabaseClose(db);
}

```

打开数据库后，我们将确保其中没有以我们所需名称命名的表。如果表已存在，那么当尝试像我们的示例中那样向其插入相同数据时，将会发生错误，因此我们使用 *DatabaseTableExists* 函数。

删除和创建表是通过查询完成的，这些查询通过两次调用 *DatabaseExecute* 函数发送到数据库，并伴有错误检查。

```

...
// if the table COMPANY exists, then delete it
if(DatabaseTableExists(db, "COMPANY"))
{
    if(!DatabaseExecute(db, "DROP TABLE COMPANY"))
    {
        Print("Failed to drop table COMPANY with code ", _LastError);
        DatabaseClose(db);
        return;
    }
}
// creating table COMPANY
if(!DatabaseExecute(db, "CREATE TABLE COMPANY("
    "ID      INT      PRIMARY KEY NOT NULL,"
    "NAME    TEXT     NOT NULL,"
    "AGE     INT      NOT NULL,"
    "ADDRESS CHAR(50),"
    "SALARY  REAL );"))
{
    Print("DB: ", filename, " create table failed with code ", _LastError);
    DatabaseClose(db);
    return;
}
...

```

我们来阐述一下 SQL 查询的本质。在 COMPANY 表中，我们只有 5 个字段：记录 ID、姓名、年龄、地址和薪水。这里的 ID 字段是一个键，即唯一索引。索引能够唯一标识每条记录，并可跨表用于将它们链接在一起。这类似于仓位 ID 链接属于特定仓位的所有交易和订单。

现在你需要用数据填充表，这是通过“INSERT”查询完成的：

```
// insert data into table
if(!DatabaseExecute(db,
    "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (1,'Paul',32,'Californ
    "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (2,'Allen',25,'Texas',
    "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (3,'Teddy',23,'Norway'
    "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (4,'Mark',25,'Rich-Mon
{
    Print("DB: ", filename, " insert failed with code ", _LastError);
    DatabaseClose(db);
    return;
}
...
```

在这里，向 COMPANY 表添加了 4 条记录，每条记录都有一个字段列表，并指明了将写入这些字段的值。记录是通过单独的“INSERT...”查询插入的，这些查询通过特殊的分隔符‘;’组合成一行，但我们也可以通过单独的 *DatabaseExecute* 调用将每条记录插入表中。

由于脚本末尾数据库将保存到“company.sqlite”文件中，下次运行时，我们会尝试使用相同的 ID 将相同的数据写入 COMPANY 表。这将导致出现错误，因此我们之前删除了表，以便每次运行脚本时都从头开始。

现在，我们从 COMPANY 表中获取 *SALARY > 15000* 字段的所有记录。这是通过 *DatabasePrepare* 函数完成的，该函数会“编译”请求文本并返回其句柄，以供后续在 *DatabaseRead* 或 *DatabaseReadBind* 函数的字符串参数传递给 SQLite 执行引擎。

```
// prepare a request with a descriptor
int request = DatabasePrepare(db, "SELECT * FROM COMPANY WHERE SALARY>15000");
if(request == INVALID_HANDLE)
{
    Print("DB: ", filename, " request failed with code ", _LastError);
    DatabaseClose(db);
    return;
}
...
```

成功创建请求后，我们需要获取其执行结果。这可以使用 *DatabaseRead* 函数完成，该函数在第一次调用时将执行查询并跳转到结果中的第一条记录。在每次后续调用时，它将读取下一条记录，直到到达末尾。在这种情况下，它将返回 *false*，表示“没有更多记录了”。

```

// printing all records with salary over 15000
int id, age;
string name, address;
double salary;
Print("Persons with salary > 15000:");
for(int i = 0; DatabaseRead(request); i++)
{
    // read the values of each field from the received record by its number
    if(DatabaseColumnInteger(request, 0, id) && DatabaseColumnText(request, 1, name)
        DatabaseColumnInteger(request, 2, age) && DatabaseColumnText(request, 3, address)
        DatabaseColumnDouble(request, 4, salary))
        Print(i, ": ", id, " ", name, " ", age, " ", address, " ", salary);
    else
    {
        Print(i, ": DatabaseRead() failed with code ", _LastError);
        DatabaseFinalize(request);
        DatabaseClose(db);
        return;
    }
}
// deleting handle after use
DatabaseFinalize(request);

```

执行结果将是：

```

Persons with salary > 15000:
0:  1 Paul 32 California 25000.0
1:  3 Teddy 23 Norway 20000.0
2:  4 Mark 25 Rich-Mond  65000.0

```

DatabaseRead 函数允许遍历查询结果中的所有记录，然后通过 *DatabaseColumn* 函数获取结果表中每一列的完整信息。这些函数旨在以通用方式处理任何查询的结果，但代价是代码冗余。

如果预先知道查询结果的结构，最好使用 *DatabaseReadBind* 函数，它允许一次将整个记录读入一个结构体中。我们可以用这种方式重制前面的示例，并以新名称 *DatabaseReadBind.mq5* 呈现它。首先，我们声明 *Person* 结构体：

```

struct Person
{
    int    id;
    string name;
    int    age;
    string address;
    double salary;
};

```

然后，只要函数返回 *true*，我们就在循环中用 *DatabaseReadBind(request, person)* 从查询结果中减去（读取）每条记录：

```

Person person;
Print("Persons with salary > 15000:");
for(int i = 0; DatabaseReadBind(request, person); i++)
    Print(i, ": ", person.id, " ", person.name, " ", person.age,
          " ", person.address, " ", person.salary);
DatabaseFinalize(request);

```

因此，我们立即从当前记录中获取所有字段的值，而无需单独读取它们。

这个入门示例摘录自文章 [SQLite: 在 MQL5 中本地处理 SQL 数据库](#)，其中除了这个示例外，还考虑了交易者应用数据库的几种方案。具体来说，你可以在那里找到从交易中恢复仓位历史、从策略、交易品种或最优选交易时段等方面分析交易报告，以及处理优化结果的技术。

掌握这些材料可能需要一些 SQL 基础知识，因此我们将在后续章节中简要介绍。

7.6.1 SQL 基础知识

在 SQLite 中执行的所有任务都假定存在一个或多个工作数据库，因此创建和打开数据库（类似于文件）是建立必要编程环境的强制性框架操作。SQLite 中没有以编程方式删除数据库的功能，因为它假定你只需从磁盘删除数据库文件。

在打开的数据库上下文中可用的操作可以大致分为以下主要组：

- 创建和删除表，以及修改其模式，即列描述，包括类型、名称和限制的标识。
- 在表中创建（添加）、读取、编辑和删除记录；这些操作通常用通用缩写 CRUD（Create, Read, Update, Delete）表示。
- 根据复杂条件构建查询以从一个或多个表的组合中选择记录。
- 可通过下列方式来优化算法：在选定列上构建索引、使用视图（view）、将批量操作包装在事务中、声明事件处理触发器以及其他高级工具

在 SQL 数据库中，所有这些操作都是使用保留的 SQL 命令（或语句）执行的。由于与 MQL5 集成的特殊性，某些操作由内置的 MQL5 函数执行。例如，打开、应用或取消事务是由 [DatabaseTransaction](#) 函数执行的，尽管 SQL 标准（以及 SQLite 的公共实现）具有相应的 SQL 命令（BEGIN TRANSACTION、COMMIT 和 ROLLBACK）。

大多数 SQL 命令在 MQL 程序中也可用：它们作为 [DatabaseExecute](#) 或 [DatabasePrepare](#) 函数的字符串参数传递给 SQLite 执行引擎。这两种选项之间的区别在于几个细微之处。

[DatabasePrepare](#) 允许准备一个查询，以便后续在每次迭代中使用不同的参数值进行大规模循环执行（参数本身，即它们在查询中的名称，是相同的）。此外，这些准备好的查询提供了使用 [DatabaseRead](#) 和 [DatabaseReadBind](#) 读取结果的机制。因此，可以将它们用于处理一组选定记录的操作。

相比之下，[DatabaseExecute](#) 函数单方面执行传递的单个查询：命令进入 SQLite 引擎内部，对数据执行某些操作，但不返回任何内容。这通常用于创建表或批量修改数据。

将来，我们将经常需要使用几个基本概念。我们一一介绍它们：

表 - 结构化的数据集，由行和列组成。每一行都是一条单独的数据记录，其字段（特性）通过相应列的名称和类型进行描述。所有数据库表都以物理方式存储在数据库文件中，并且可供读写（如果在打开数据库时未限制权限）。

视图 一种虚拟表，由 SQLite 引擎根据给定的 SQL 查询、其他表或视图计算得出。视图是只读的。不同于任何表（包括 SQL 允许在程序会话期间在内存中创建的临时表），每次访问视图时，都会动态重新计算视图。

索引 一种服务数据结构（平衡树，B-树），用于通过预定义字段（特性）的值或其组合快速搜索记录。

触发器 一个或多个 SQL 语句的子程序，被指定为在特定表中添加、更改或删除记录的事件（之前或之后）发生时自动运行。

以下是最常用的 SQL 语句及其执行的操作的简短列表：

- CREATE 创建数据库对象（表、视图、索引、触发器）；
- ALTER 更改对象（表）；
- DROP 删除对象（表、视图、索引、触发器）；
- SELECT 选择满足给定条件的记录或计算值；
- INSERT 添加新数据（一条或一组记录）；
- UPDATE 更改现有记录；
- DELETE 从表中删除记录；

该列表仅显示了启动相应 SQL 语言结构的关键字。更详细的语法将在下面显示。它们的实际应用将在以下示例中展示。

每个语句可以跨越多行（换行符和多余的空格会被忽略）。如果需要，可以一次向 SQLite 发送多个命令。在这种情况下，在每个命令之后，应该使用命令终止符 ';'（分号）。

系统分析命令中的文本时不区分大小写，但在 SQL 中习惯于用大写字母书写关键字。

创建表时，我们必须指定其名称，以及列的列表，列用括号括起来并以逗号分隔。每一列都被赋予一个名称、一个类型以及可选的约束。最简单的形式：

```
CREATE TABLE table_name
( column_name type [ constraints ] [, column_name type [ constraints ...] ...]);
```

我们将在 [下一节](#) 中了解 SQL 中的限制。与此同时，我们来看一个清晰的示例（包含不同类型和选项）：

```
CREATE TABLE IF NOT EXISTS example_table
(id INTEGER PRIMARY KEY,
 name TEXT,
 timestamp INTEGER DEFAULT CURRENT_TIMESTAMP,
 income REAL,
 data BLOB);
```

创建索引的语法是：

```
CREATE [ UNIQUE ] INDEX index_name
ON table_name( column_name [, column_name ...]);
```

现有索引会在对相应列具有筛选条件的查询中自动使用。没有索引，处理过程会更慢。

删除表（如果已向其中写入数据，则连同数据一起删除）非常简单：

```
DROP TABLE table_name;
```

可以如下所示向表中插入数据：

```
INSERT INTO table_name [ ( column_name [, column_name ...] ) ]
VALUES( value [, value ...]);
```

括号中的第一个列表包含列名，是可选的（见下文解释）。第一个列表必须与第二个包含这些列值的列表相匹配。例如，

```
INSERT INTO example_table (name, income) VALUES ('Morning Flat Breakout', 1000);
```

请注意，在 SQL 中，字符串字面量用单引号括起来。

如果 INSERT 语句中省略了列名，则假定 VALUES 关键字后跟的是表中所有列的值，并且严格按照它们在表中描述的顺序排列。

该运算符还有更复杂的形式，特别允许从其他表或查询结果中插入记录。

按条件选择记录，并可选地限制返回字段（列）的列表，由 SELECT 命令执行。

```
SELECT column_name [, column_name ...] FROM table_name [WHERE condition ];
```

如果你想完整地返回每个匹配的记录（所有列），请使用星号表示法：

```
SELECT *FROM table_name [WHERE condition ];
```

当条件不存在时，系统返回表中的所有记录。

作为条件，可以代入一个逻辑表达式，该表达式包含列名和各种比较运算符，以及内置的 SQL 函数和嵌套 SELECT 查询的结果（此类查询写在括号中）。比较运算符包括：

- 逻辑与
- 逻辑或
- IN，列表中的某个值
- NOT IN，列表外的某个值
- BETWEEN，范围内的某个值
- LIKE -与带有特殊通配符（%、'_'）的模式在拼写上相似
- EXISTS -检查嵌套查询结果的非空性

例如，选择收入至少为 1000 且年龄不超过一年（预先四舍五入到最近的月份）的记录名称：

```
SELECT name FROM example_table
WHERE income >= 1000 AND timestamp > datetime('now', 'start of month', '-1 year');
```

此外，选择结果可以按升序或降序排序（ORDER BY），按特征分组（GROUP BY），并按组筛选（HAVING）。我们还可以限制其中的记录数量（LIMIT、OFFSET）。对于每个组，可以返回任何聚合函数的值，特别是 COUNT、SUM、MIN、MAX 和 AVG，这些函数是在所有组成员记录上计算的。

```

SELECT [ DISTINCT ] column_name [, column_name...](i) FROM table_name
[ WHERE condition ]
[ ORDER BY column_name [ ASC | DESC ]
  [ LIMIT quantity OFFSET start_offset ] ]
[ GROUP BY column_name [ HAVING condition ] ];

```

可选关键字 `DISTINCT` 可用于移除重复项 (如果根据当前选择标准在结果中找到它们)。该关键字仅在没有分组的情况下才有意义。

只有在存在排序的情况下, `LIMIT` 才能给出可重现的结果。

如果需要, `SELECT` 选择可以不是从一个表而是从多个表进行, 根据所需的字段组合将它们结合起来。为此使用关键字 `JOIN`。

```

SELECT [...] FROM table name_1
  [ INNER | OUTER | CROSS ] JOIN table_name_2
  ON boolean_condition

```

或

```

SELECT [...] FROM table name_1
  [ INNER | OUTER | CROSS ] JOIN table_name_2
  USING ( common_column_name [, common_column_name ...] )

```

SQLite 支持三种 `JOIN`: `INNER JOIN`、`OUTER JOIN` 和 `CROSS JOIN`。本书通过示例对其进行了概要介绍, 你可以自行进一步探索细节。

例如, 使用 `JOIN`, 你可以构建一个表中的记录与另一个表中的记录的所有组合, 或者根据匹配仓位标识符的原则比较交易表 (我们称之为 “deals”) 中的交易与同一表中的交易, 但要使交易的方向 (入市/出市) 相反, 从而产生一个虚拟的交易表。

```

SELECT // list the columns of the results table with aliases (after 'as')
  d1.time as time_in, d1.position_id as position, d1.type as type, // table d1
  d1.volume as volume, d1.symbol as symbol, d1.price as price_in,
  d2.time as time_out, d2.price as price_out,                      // table d2
  d2.swap as swap, d2.profit as profit,
  d1.commission + d2.commission as commission                    // combination
FROM deals d1 INNER JOIN deals d2 // d1 and d2 - aliases of one table "deals"
ON d1.position_id = d2.position_id // merge condition by position
WHERE d1.entry = 0 AND d2.entry = 1 // selection condition "entry/exit"

```

这是 MQL5 帮助中的一个 SQL 查询, 在 *DatabaseExecute* 和 *DatabasePrepare* 函数的描述中可以找到 `JOIN` 的示例。

`SELECT` 的一个基本特性是它总是向调用程序返回结果, 这与其他查询 (如 `CREATE`、`INSERT` 等) 不同。然而, 从 SQLite 3.35 开始, `INSERT`、`UPDATE` 和 `DELETE` 语句也可视需要使用附加的 `RETURNING` 关键字返回值。例如,

```

INSERT INTO example_table (name, income) VALUES ('Morning Flat Breakout', 1000)
RETURNING id;

```

在任何情况下, MQL5 中的查询结果都是通过 *DatabaseColumn* 函数、*DatabaseRead* 和 *DatabaseReadBind* 访问的。

此外, `SELECT` 允许对表达式的结果求值, 并按原样返回它们, 或将它们与表中的结果组合起来。表达式可以包含我们从 *MQL5 表达式* 中学到的大多数运算符, 以及内置的 SQL 函数。请参

阅 SQLite 文档以获取完整列表。例如，下面是如何在终端和编辑器实例中查找 SQLite 当前的构建版本，这对于了解哪些选项可用可能很重要。

```
SELECT sqlite_version();
```

这里，整个表达式由对 `sqlite_version` 函数的单个调用组成。与从表中选择多个列类似，你可以评估用逗号分隔的多个表达式。

还提供了一些常用的 [统计](#) 和 [数学](#) 函数。

应使用 UPDATE 语句编辑记录。

```
UPDATE table_name SET column_name = value [, column_name = value ...]
WHERE condition;
```

删除命令的语法如下：

```
DELETE FROM table_name WHERE condition;
```

7.6.2 表的结构：数据类型和限制

在描述表字段时，你需要为它们指定数据类型，但 SQLite 中的数据类型概念与 MySQL 大相径庭。

MySQL 是一种强类型语言：每个变量或结构体字段始终保持其声明时的数据类型。而 SQL 是一种弱类型语言：我们在表描述中指定的类型仅仅是一种建议。程序可以将任意类型的值写入任何“单元格”（成为记录中的一个字段），并且该“单元格”会改变其类型，这一点尤其可以被内置的 MySQL 函数 `DatabaseColumnType`检测到。

当然，在实践中，大多数用户倾向于“遵循”列类型。

SQL 类型机制的第二个显著区别是存在大量描述类型的关键字，但所有这些字词最终都归结为五个存储类别。作为 SQL 的简化版本，SQLite 在大多数情况下不区分同一组的关键字（例如，在描述长度限制为 VARCHAR(80) 的字符串时，此限制不受控制，该描述等同于 TEXT 存储类），因此通过组名描述类型更为逻辑。保留某些特定类型只是为了与其他 DBMS 兼容（但这对于我们来说不重要）。

下表列出了 MySQL 类型及其对应的“近似类型”（指 SQL 类型的概括性特征）。

MySQL 类型	通用 SQL 类型
NULL（在 MySQL 中不是一种类型）	NULL（无值）
bool、char、short、int、long、uchar、ushort、uint、ulong、datetime、color、enum	INTEGER
float、double	REAL
MySQL 中无对应类型，指固定精度实数）	NUMERIC
string	TEXT
（任意“原始”数据，类似于 uchar[] 数组或其他）	BLOB（二进制大对象），NONE

当向 SQL 数据库写入值时，它会根据几条规则确定其类型：

- 没有引号、小数点或指数，则为 INTEGER
- 存在小数点和/或指数，则为 REAL
- 用单引号或双引号框起来，则表示 TEXT 类型
- 不带引号的 NULL 值对应于 NULL 类
- 带有二进制数据的字面量 (常量) 作为以 'x' 为前缀的十六进制字符串写入

使用特殊的 SQL 函数 **typeof**，你可以检查值的类型。例如，以下查询可以在 MetaEditor 中运行。

```
SELECT typeof(100), typeof(10.0), typeof('100'), typeof(x'1000'), typeof(NULL);
```

它将在结果表中输出：

integer		real		text		blob		null

不能通过比较 '=' 来检查 NULL 值 (因为结果也会给出 NULL)，而应该使用特殊的 NOT NULL 运算符。

SQLite 对存储数据施加了一些限制：其中一些限制很难达到 (因此在本文中省略)，但其他一些在设计程序时需要加以考虑。表中列的最大数量是 2000，而一行、BLOB 以及通常一条记录的大小不能超过一百万字节。SQL 查询长度限制也选择了相同的值。

就日期和时间而言，SQL 理论上可以以三种格式存储它们，但只有第一种与 MySQL 中的 **datetime** 匹配：

- INTEGER - 自 1970.01.01 以来的秒数 (也称为“Unix 纪元时间”)
- REAL - 从公元前 4714 年 11 月 24 日起的天数 (带小数部分)
- TEXT - 格式为 “YYYY-MM-DD HH:mm:ss.sss”、精确到毫秒的日期和时间，可选时区，为此添加后缀 “[±HH:mm]” 表示与 UTC 的偏移量

真实的日期存储类型 (也称为儒略日，有内置的 SQL 函数 **Julianday**) 的有趣之处在于，它允许存储精确到毫秒的时间。理论上，这也可以通过 “YYYY-MM-DDTHH:mm:ss.sssZ” 格式的字符串来实现，但这种存储非常不经济。从我们熟悉的日期 1970.01.01 00:00:00 开始，将“天数”转换为带小数部分的秒数，是根据以下公式进行的：**julianday('now') - 2440587.5) * 86400.0**。此处的 'now' 表示当前 UTC 时间，但可以更改为 SQLite 文档中描述的其他值。常量 2440587.5 精确等于指定“零”日期 (“Unix 纪元时间”起点) 的“日历”天数。

除了类型之外，每个字段还可以有一个或多个约束，这些约束在类型之后用特殊的关键字写入。约束描述了字段可以取什么值，甚至允许根据字段的预定义用途自动完成。

我们考虑主要的约束。

```
... DEFAULT expression
```

添加新记录时，如果未指定该字段的值，系统将自动输入此处指定的值 (常量) 或计算表达式 (函数)。

```
... CHECK ( boolean_expression )
```

当添加新记录时，系统将检查表达式 (可以包含字段名作为变量) 是否为真。如果表达式为假，则不会插入记录，系统将返回错误。

... UNIQUE

系统检查表中所有记录在此字段上具有不同的值。尝试添加具有已存在值的条目将导致错误，并且不会进行添加。

为了跟踪唯一性，系统会为指定字段隐式创建一个索引。

... PRIMARY KEY

系统使用标记有此属性的字段来标识表中的记录，并从其他表链接到它们（关系型数据库（如 SQLite）的名称由此而来，即关系是如何形成的）。显然，此功能还包括唯一索引。

如果表没有带 PRIMARY KEY 属性的 INTEGER 类型字段，系统会自动隐式创建这样一个类别，且名为 *rowid*。如果表有一个声明为主键的整数字段，那么它也可以使用别名 *rowid*。

如果向表中添加一条省略了 *rowid* 或 *rowid* 为 NULL 的记录，SQLite 将自动为其分配下一个整数 64 位，对应 MQL5 中的 *long*），该整数比表中最大 *rowid* 大 1。初始值为 1。

通常计数器每次只递增 1，但如果曾经插入到一个表中的记录数量（然后可能被删除）超过 *long* 的最大值，计数器将跳回到开头，系统将尝试查找空闲的数字。但这不太可能发生。例如，如果以平均每毫秒 1 条分时报价的速度向表中写入数据，那么 2.92 亿年后才会溢出。

只能有一个主键，但它可以由多个列组成，这是通过直接在表描述中使用不同于约束的语法来完成的。

```
CREATE TABLE table_name (
    column_name type [ restrictions ]
    [, column_name type [ restrictions ] ...]
    , PRIMARY KEY ( column_name [, column_name ...] ) );
```

我们回到约束。

... AUTOINCREMENT

此约束只能作为 PRIMARY KEY 的补充指定，确保标识符始终递增。这意味着任何以前的 ID，即使是用在已删除条目上的 ID，也不会被重新选择。然而，与简单的 PRIMARY KEY 相比，SQLite 中此机制的实现在计算资源方面效率较低，因此不建议使用。

... NOT NULL

此约束禁止向表中添加此字段未填充的记录。默认情况下，当没有约束时，任何非唯一字段都可以在添加的记录中省略，并将设置为 NULL。

```
... CURRENT_TIME
... CURRENT_DATE
... CURRENT_TIMESTAMP
```

使用这些指令，你可以在插入记录时自动用时间（不含日期）、日期（不含时间）或完整的 UTC 时间填充字段（前提是 INSERT SQL 语句没有显式地向此字段写入任何内容，甚至 NULL 也不行）。SQLite 不知道如何以类似方式自动检测记录更改的时间，为此你将必须编写一个触发器（这不在本书的讨论范围）。

遗憾的是，CURRENT_TIMESTAMP 组限制在 SQLite 中的实现有一个疏漏：如果字段为 NULL，则不应用时间戳。这使得 SQLite 与其他 SQL 引擎以及 SQLite 本身处理主键字段中 NULL 的方式有所不同。事实证明，对于自动标记，不能将整个对象写入数据库，而需要显式指定除日期和时间字段之外的所有字段。为了解决这个问题，我们需要一个替代方案，即在编译的查询中为相应的列替换 SQL 函数 STRFTIME('%s')。

7.6.3 OOP (MQL5) 和 SQL 集成 ORM 概念

在 MQL 程序中使用数据库意味着算法被分为两部分：控制部分用 MQL5 编写，执行部分用 SQL 编写。这就导致源代码可能开始看起来像是东拼西凑的，并且需要注意保持一致性。为避免这种情况，面向对象的语言发展了对象关系映射（ORM）的概念，即将对象映射到关系表记录，反之亦然。

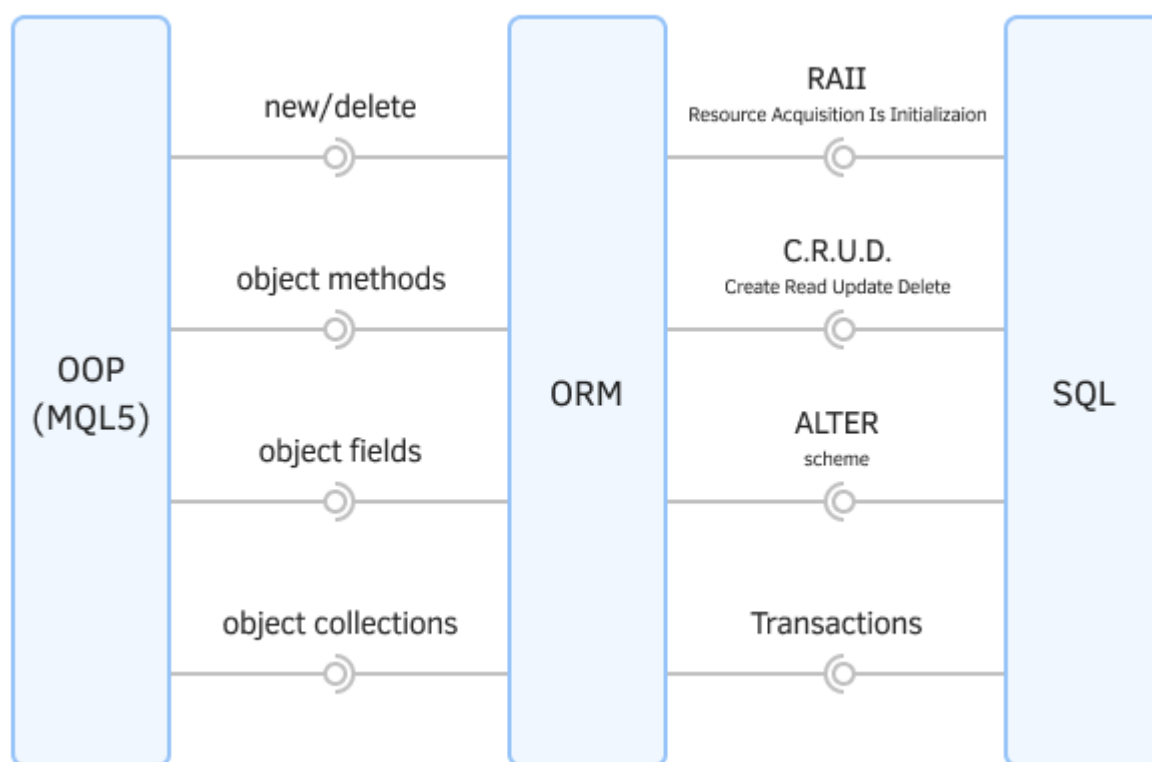
该方法的本质是将 SQL 语言中的所有操作封装在一个特殊层的类/结构体中。因此，程序的应用部分可以用纯粹的 OOP 语言（例如 MQL5）编写，而不会被 SQL 的细微差别分散注意力。

如果存在完整 ORM 实现（以包含所有命令的“黑匣子”形式），应用程序开发人员通常无需学习 SQL。

此外，ORM 允许在必要时“不知不觉地”更改 DBMS 的“引擎”。这对于 MQL5 来说并非特别重要，因为它只内置了 SQLite 数据库，但一些开发人员更喜欢使用功能齐全的 DBMS，并通过导入 DLL 将它们连接到 MetaTrader 5。

当我们需自动获取和释放资源时，使用带有构造函数和析构函数的对象非常有用。我们已经在文件描述符管理一节中介绍了这个概念（RAII，资源获取即初始化），然而，正如我们稍后将看到的，使用数据库也是基于不同类型描述符的分配和释放。

下图示意性地描绘了以 ORM 形式集成 OOP 和 SQL 时不同软件层的交互。



ORM，对象关系映射

还有一个优势，对象“包装器”（不仅仅是特定于数据库的 ORM）会自动完成数据准备和转换，并且会检查正确性以防止某些错误。

在以下章节中，当我们逐步了解用于操作数据库的内置函数时，我们将实现这些示例，逐渐构建我们自己的简单 ORM 层。由于 MQL5 的某些特性，我们的类将无法提供 100% 覆盖任务的通用性，但对许多项目将非常有用。

7.6.4 创建、打开和关闭数据库

DatabaseOpen 和 *DatabaseClose* 函数能够创建和打开数据库。

int DatabaseOpen(const string filename, uint flags)

该函数在名为 *filename* 的文件中打开或创建数据库。该参数不仅可以包含名称，还可以包含相对于 *MySQL5/Files* (特定终端实例的或共享文件夹中，见下文标志)的子文件夹路径。可以省略扩展名，这会为默认名称添加 ".sqlite"。

如果在 *filename* 参数中指定 NULL 或空字符串 ""，则数据库在临时文件中创建，该文件将在数据库关闭后自动删除。

如果在 *filename* 参数中指定 ":memory:" 字符串，则数据库将在内存中创建。这样的临时数据库将在关闭后自动删除。

flags 参数包含一个标志组合，这些标志描述了从 ENUM_DATABASE_OPEN_FLAGS 枚举创建或打开数据库的附加条件。

标识符	说明
DATABASE_OPEN_READONLY	只读打开
DATABASE_OPEN_READWRITE	读写打开
DATABASE_OPEN_CREATE	如果磁盘上不存在文件，则创建一个
DATABASE_OPEN_MEMORY	创建内存数据库
DATABASE_OPEN_COMMON	文件位于所有终端的共享文件夹中

如果在 *flags* 参数中未指定 DATABASE_OPEN_READONLY 或 DATABASE_OPEN_READWRITE 标志，则 will 使用 DATABASE_OPEN_READWRITE 标志。

成功时，函数返回数据库的句柄，该句柄随后用作一个参数，以供其他函数访问该数据库。否则，返回 INVALID_HANDLE，并且可以在 *_LastError* 中找到错误代码。

void DatabaseClose(int database)

DatabaseClose 函数通过其句柄关闭数据库，该句柄是先前从 *DatabaseOpen* 函数接收的。

调用 *DatabaseClose* 后，我们将在以下章节中学习为打开的数据库创建的所有查询句柄都将自动移除并失效。

该函数不返回任何内容。但是，如果向其传递了不正确的句柄，它会将 *_LastError* 设置为 ERR_DATABASE_INVALID_HANDLE。

我们在 *DBSQLite.mqh* 文件中开始开发数据库的面向对象包装器。

DBSQLite 类将确保数据库的创建、打开和关闭。我们稍后会对其进行扩展。

```

class DBSQLite
{
protected:
    const string path;
    const int handle;
    const uint flags;

public:
    DBSQLite(const string file, const uint opts =
        DATABASE_OPEN_CREATE | DATABASE_OPEN_READWRITE):
        path(file), flags(opts), handle(DatabaseOpen(file, opts))
    {
    }

    ~DBSQLite(void)
    {
        if(handle != INVALID_HANDLE)
        {
            DatabaseClose(handle);
        }
    }

    int getHandle() const
    {
        return handle;
    }

    bool isOpen() const
    {
        return handle != INVALID_HANDLE;
    }
};

```

请注意，数据库在创建对象时自动创建或打开，并在销毁对象时关闭。

使用这个类，我们编写一个简单的脚本 *DBinit.mq5*，它将创建或打开指定的数据库。

```

input string Database = "MQL5Book/DB/Example1";

void OnStart()
{
    DBSQLite db(Database); // create or open the base in the constru
    PRTF(db.getHandle()); // 65537 / ok
    PRTF(FileIsExist(Database + ".sqlite")); // true / ok
} // the base is closed in the destructor

```

以默认设置首次运行后，我们应该得到一个新文件 *MQL5/Files/MQL5Book/DB/Example1.sqlite*。要确认这一点，只需在代码中检查该文件是否存在。在后续以相同名称运行时，脚本只是打开数据库并记录当前的描述符（一个整数）。

7.6.5 执行没有 MQL5 数据绑定的查询

某些 SQL 查询是命令，你只需按原样将它们发送到引擎即可。它们既不需要变量输入也不需要结果。例如，如果我们的 MQL 程序需要在数据库中创建具有特定结构和名称的表、索引或视图，我们可以将其编写为带有“CREATE ...”语句的常量字符串。此外，使用此类查询进行记录的批量处理或其组合（合并、计算聚合指标和同类型修改）也很方便。也就是说，通过一个查询，可以转换整个表数据或基于它填充其他表。这些结果可以在后续查询中进行分析。

在所有这些情况下，最重要的是获得操作成功的确认。此类请求使用 `DatabaseExecute` 函数执行。

`bool DatabaseExecute(int database, const string sql)`

该函数在由 `database` 描述符指定的数据库中执行查询。请求本身作为准备好的 `sql` 字符串发送。

该函数返回操作状态标识：成功 (`true`) 或错误 (`false`)。

例如，我们可以用这个方法补充我们的 `DBSQLite` 类（描述符已经在对象内部）。

```
class DBSQLite
{
    ...
    bool execute(const string sql)
    {
        return DatabaseExecute(handle, sql);
    }
};
```

然后，创建新表（如果需要，事先创建数据库本身）的脚本可能如下所示（`DBcreateTable.mq5`）。

```
input string Database = "MQL5Book/DB/Example1";
input string Table = "table1";

void OnStart()
{
    DBSQLite db(Database);
    if(db.isOpen())
    {
        PRTF(db.execute(StringFormat("CREATE TABLE %s (msg text)", Table))); // true
    }
}
```

执行脚本后，尝试在 MetaEditor 中打开指定的数据库，并确保它包含一个带单个“msg”文本字段的空表。但这也可以通过编程方式完成（参见 [下一节](#)）。

如果我们使用相同的参数再次运行该脚本，将会遇到一个错误（尽管并非严重到强制程序关闭）。

```
database error, table table1 already exists
db.execute(StringFormat(CREATE TABLE %s (msg text),Table))=false / DATABASE_ERROR(560
```

这是因为不能重复创建已存在的表。但是，SQL 允许我们抑制这个错误，并且仅在表尚不存在时才创建它，否则几乎不执行任何操作并返回成功指示。为此，只需在查询中的表名前添加“IF NOT EXISTS”即可。

```
db.execute(StringFormat("CREATE TABLE IF NOT EXISTS %s (msg text)", Table));
```

在实践中，表用于存储关于应用领域中对象（如报价、交易、交易信号）的信息。因此，最好能根据 MQL5 中对象的描述来自动化表的创建过程。正如我们稍后将看到的，SQLite 函数能够将查询结果绑定到 MQL5 结构体（而非类）。有鉴于此，在 ORM 包装器的框架内，我们将开发一种机制，用于根据 MQL5 中特定类型的 *struct* 描述来生成“CREATE TABLE”SQL 查询。

这需要在编译时以某种方式在通用列表中注册结构体字段的名称和类型，然后在程序执行阶段，就可以从这个列表中生成 SQL 查询。

MQL5 的编译阶段会解析几类实体，这些实体可用于识别类型和名称：

- 宏
- 继承
- 模板

首先需要明确的是，收集到的字段描述是与特定结构体的上下文相关的，不应混淆，因为程序中可能包含许多具有潜在相同名称和类型的不同结构体。换句话说，最好是为每种类型的结构体在单独的列表中累积信息。模板类型非常适合此目的，其模板参数（S）将是应用程序结构体。我们将这个模板称为 *DBEntity*。

```
template<typename S>
struct DBEntity
{
    static string prototype[][3]; // 0 - type, 1 - name, 2 - constraints
    ...
};

template<typename T>
static string DBEntity::prototype[][3];
```

在该模板内部，有一个多维数组 *prototype*，我们将在其中写入字段的描述。为了捕获所应用字段的类型和名称，需要在 *DBEntity* 内部声明另一个模板结构体 *DBField*：这次其参数 T 是字段本身的类型。在其构造函数中，我们拥有关于此类型的信息（*typename(T)*），并且还通过参数获取字段的名称（以及可选的约束）。

```
template<typename S>
struct DBEntity
{
    ...
    template<typename T>
    struct DBField
    {
        T f;
        DBField(const string name, const string constraints = "")
        {
            const int n = EXPAND(prototype);
            prototype[n][0] = typename(T);
            prototype[n][1] = name;
            prototype[n][2] = constraints;
        }
    };
};
```

字段 *f* 虽未使用，但却是必需的，因为结构体不能为空。

假设我们有一个应用程序结构体 *Data* (*DBmetaProgramming.mq5*)。

```
struct Data
{
    long id;
    string name;
    datetime timestamp;
    double income;
};
```

我们可以创建其模拟结构体，该结构体继承自 *DBEntity<DataDB>*，但其字段基于 *DBField* 进行了替换，与原始集合相同。

```
struct DataDB: public DBEntity<DataDB>
{
    DB_FIELD(long, id);
    DB_FIELD(string, name);
    DB_FIELD(datetime, timestamp);
    DB_FIELD(double, income);
} proto;
```

通过将结构体的名称代入父模板参数，该结构体向程序提供了关于其自身特性的信息。

请注意与结构体声明一起进行的一次性 *proto* 变量定义。这是必要的，因为在模板中，只有在源代码中至少创建了一个此类型的对象时，每个特定的参数化类型才会被编译。重要的是，*proto* 对象的创建发生在程序启动的最开始，即全局变量初始化之时。

DB_FIELD 标识符下隐藏了一个宏：

```
#define DB_FIELD(T,N) struct T##_##N: DBField<T> { T##_##N() : DBField<T>(#N) { } } \
    _##T##_##N;
```

以下是它针对单个字段展开的方式：

```
struct Type_Name: DBField<Type>
{
    Type_Name() : DBField<Type>(Name) { }
} _Type_Name;
```

这里，该结构体不仅被定义，而且被立即创建：实际上，它替换了原始字段。

由于 *DBField* 结构体包含一个所需类型的单个变量 *f*，因此 *Data* 和 *DataDB* 的维度及内部二进制表示是相同的。通过运行 *DBmetaProgramming.mq5* 脚本，可以轻松验证这一点。

```
void OnStart()
{
    PRTF(sizeof(Data));
    PRTF(sizeof(DataDB));
    ArrayPrint(DataDB::prototype);
}
```

它会向日志输出：

```

DBEntity<Data>::DBField<long>::DBField<long>(const string,const string)
long id
DBEntity<Data>::DBField<string>::DBField<string>(const string,const string)
string name
DBEntity<Data>::DBField<datetime>::DBField<datetime>(const string,const string)
datetime timestamp
DBEntity<Data>::DBField<double>::DBField<double>(const string,const string)
double income
sizeof(Data)=36 / ok
sizeof(DataDB)=36 / ok
          [,0]          [,1]          [,2]
[0,] "long"          "id"          ""
[1,] "string"        "name"        ""
[2,] "datetime"      "timestamp"    ""
[3,] "double"        "income"      ""

```

然而，要访问这些字段，你需要编写一些不方便的代码：*data._long_id.f*、*data._string_name.f*、*data._datetime_timestamp.f*、*data._double_income.f*。

我们不会这样做，不仅（或者说不主要）因为不方便，而且因为这种构建元结构的方式与将数据绑定到 SQL 查询的原则不兼容。在接下来的章节中，我们将探讨允许在 MQL5 结构体中获取表记录和 SQL 查询结果的 *database* 函数。然而，只允许使用没有继承且不包含对象类型静态成员的简单结构体。因此，需要稍微改变揭示元信息的方式。

我们将必须保持结构体的原始类型不变，并实际上重复数据库的描述，确保没有差异（拼写错误）。这不是很方便，但目前没有其他方法。

我们将把 *DBEntity* 和 *DBField* 实例的声明移到应用程序结构体之外。在这种情况下，*DB_FIELD* 宏将接收一个额外的参数 (S)，在该参数中需要传递应用程序结构体的类型（以前它是通过在结构体内部声明来隐式获取的）。

```

#define DB_FIELD(S,T,N) \
    struct S##_##T##_##N: DBEntity<S>::DBField<T> \
    { \
        S##_##T##_##N() : DBEntity<S>::DBField<T>(#N) {} \
    }; \
    const S##_##T##_##N _##S##_##T##_##N;

```

由于表列可以有约束，因此如果需要，也需要将它们传递给 *DBField* 构造函数。为此，我们添加几个带有相应参数的宏（理论上，一列可以有多个约束，但通常不超过两个）。

```

#define DB_FIELD_C1(S,T,N,C1) \
    struct S##_##T##_##N: DBEntity<S>::DBField<T> \
    { \
        S##_##T##_##N() : DBEntity<S>::DBField<T>(#N, C1) {} \
    }; \
    const S##_##T##_##N _##S##_##T##_##N;

#define DB_FIELD_C2(S,T,N,C1,C2) \
    struct S##_##T##_##N: DBEntity<S>::DBField<T> \
    { \
        S##_##T##_##N() : DBEntity<S>::DBField<T>(#N, C1 + " " + C2) {} \
    }; \
    const S##_##T##_##N _##S##_##T##_##N;

```

所有这三个宏以及后续的开发都添加到了头文件 *DBSQLite.mqh* 中。

务必注意，这种“自制”的对象到表的绑定方式仅用于向数据库输入数据，因为从表读取数据到对象的功能在 MQL5 中是使用 *DatabaseReadBind* 函数实现的。

我们还改进了 *DBField* 的实现。MQL5 类型与 SQL 存储类不完全对应，因此在填充 *prototype[n][0]* 元素时需要进行转换。这种转换是由静态方法 *affinity* 完成的。

```

template<typename T>
struct DBField
{
    T f;
    DBField(const string name, const string constraints = "")
    {
        const int n = EXPAND(prototype);
        prototype[n][0] = affinity(typename(T));
        ...
    }

    static string affinity(const string type)
    {
        const static string ints[] =
        {
            "bool", "char", "short", "int", "long",
            "uchar", "ushort", "uint", "ulong", "datetime",
            "color", "enum"
        };
        for(int i = 0; i < ArraySize(ints); ++i)
        {
            if(type == ints[i]) return DB_TYPE::INTEGER;
        }

        if(type == "float" || type == "double") return DB_TYPE::REAL;
        if(type == "string") return DB_TYPE::TEXT;
        return DB_TYPE::BLOB;
    }
};

```

此处使用的 SQL 通用类型的文本常量放置在一个单独的命名空间中：它们可能在 MQL 程序的某些不同地方需要用到，并且必须确保没有名称冲突。

```

namespace DB_TYPE
{
    const string INTEGER = "INTEGER";
    const string REAL = "REAL";
    const string TEXT = "TEXT";
    const string BLOB = "BLOB";
    const string NONE = "NONE";
    const string _NULL = "NULL";
}

```

为了方便，可能的约束预设也在其组中进行了描述（作为提示）。

```

namespace DB_CONSTRAINT
{
    const string PRIMARY_KEY = "PRIMARY KEY";
    const string UNIQUE = "UNIQUE";
    const string NOT_NULL = "NOT NULL";
    const string CHECK = "CHECK (%s)"; // requires an expression
    const string CURRENT_TIME = "CURRENT_TIME";
    const string CURRENT_DATE = "CURRENT_DATE";
    const string CURRENT_TIMESTAMP = "CURRENT_TIMESTAMP";
    const string AUTOINCREMENT = "AUTOINCREMENT";
    const string DEFAULT = "DEFAULT (%s)"; // requires an expression (constants, funct
}

```

由于某些约束需要参数（其位置用通常的 `%s` 格式修饰符标记），我们添加一项约束存在性检查。这是 `DBField` 构造函数的最终形式。

```

template<typename T>
struct DBField
{
    T f;
    DBField(const string name, const string constraints = "")
    {
        const int n = EXPAND(prototype);
        prototype[n][0] = affinity(typename(T));
        prototype[n][1] = name;
        if(StringLen(constraints) > 0 // avoiding error STRING_SMALL_LEN(5035)
            && StringFind(constraints, "%") >= 0)
        {
            Print("Constraint requires an expression (skipped): ", constraints);
        }
        else
        {
            prototype[n][2] = constraints;
        }
    }
}

```

由于宏和辅助对象 `DBEntity<S>` 和 `DBField<T>` 的组合填充了一个原型数组，因此在 `DBSQLite` 类内部，可以实现自动生成用于创建结构体表的 SQL 查询。

`createTable` 方法使用应用程序结构体类型进行模板化，并包含一个查询存根（“CREATE TABLE %s %s (%s);”）。其第一个参数是可选指令“IF NOT EXISTS”。第二个参数是表名，默认情况下取自模板参数的类型名称 `typename(S)`，但如果需要，可以通过输入参数 `name`（如果不为 NULL）替换为其他名称。最后，括号中的第三个参数是表列的列表：它由辅助方法 `columns` 基于数组 `DBEntity<S>::prototype` 形成。

```

class DBSQLite
{
    ...
    template<typename S>
    bool createTable(const string name = NULL,
        const bool not_exist = false, const string table_constraints = "") const
    {
        const static string query = "CREATE TABLE %s %s (%s)";
        const string fields = columns<S>(table_constraints);
        if(fields == NULL)
        {
            Print("Structure '", typename(S), "' with table fields is not initialized");
            SetUserError(4);
            return false;
        }
        // attempt to create an already existing table will give an error,
        // if not using IF NOT EXISTS
        const string sql = StringFormat(query,
            (not_exist ? "IF NOT EXISTS" : ""),
            StringLen(name) ? name : typename(S), fields);
        PRTF(sql);
        return DatabaseExecute(handle, sql);
    }

    template<typename S>
    string columns(const string table_constraints = "") const
    {
        static const string continuation = ",\n";
        string result = "";
        const int n = ArrayRange(DBEntity<S>::prototype, 0);
        if(!n) return NULL;
        for(int i = 0; i < n; ++i)
        {
            result += StringFormat("%s%s %s %s",
                i > 0 ? continuation : "",
                DBEntity<S>::prototype[i][1], DBEntity<S>::prototype[i][0],
                DBEntity<S>::prototype[i][2]);
        }
        if(StringLen(table_constraints))
        {
            result += continuation + table_constraints;
        }
        return result;
    }
};

```

对于每一列，描述都包含名称、类型和可选的约束。此外，还可以传递一个针对表的通用约束 (*table_constraints*)。

在将生成的 SQL 查询发送给 *DatabaseExecute* 函数之前，*createTable* 方法会向日志输出查询文本的调试信息（ORM 类中所有此类输出都可以通过替换 PRTF 宏来集中禁用）。

现在一切就绪，可以编写测试脚本 *DBcreateTableFromStruct.mq5* 了，该脚本将通过结构体声明在 SQLite 中创建相应的表。在输入参数中，我们只设置数据库的名称，程序将根据结构体的类型自行选择表名。

```
#include <MQL5Book/DBSQLite.mqh>

input string Database = "MQL5Book/DB/Example1";

struct Struct
{
    long id;
    string name;
    double income;
    datetime time;
};

DB_FIELD_C1(Struct, long, id, DB_CONSTRAINT::PRIMARY_KEY);
DB_FIELD(Struct, string, name);
DB_FIELD(Struct, double, income);
DB_FIELD(Struct, string, time);
```

在主函数 *OnStart* 中，我们通过调用具有默认设置的 *createTable* 来创建表。如果我们不希望在下次尝试创建它时收到错误标志，则需要将 *true* 作为第一个参数 (*db.createTable<Struct>(true)*) 传递。

```
void OnStart()
{
    DBSQLite db(Database);
    if(db.isOpen())
    {
        PRTF(db.createTable<Struct>());
        PRTF(db.hasTable(typename(Struct)));
    }
}
```

hasTable 方法通过表名检查数据库中是否存在该表。我们将在 [下一节](#) 中探讨此方法的实现。现在，我们运行该脚本。首次运行后，表成功创建，你可以在日志中看到 SQL 查询（它显示时有换行，与我们在代码中构建时的方式一致）。

```
sql=CREATE TABLE Struct (id INTEGER PRIMARY KEY,
name TEXT ,
income REAL ,
time TEXT ); / ok
db.createTable<Struct>()=true / ok
db.hasTable(typename(Struct))=true / ok
```

第二次运行将从 *DatabaseExecute* 调用中返回一个错误，因为该表已存在，*hasTable* 的结果也进一步证明了这一点。

```

sql=CREATE TABLE Struct (id INTEGER PRIMARY KEY,
name TEXT ,
income REAL ,
time TEXT ); / ok
database error, table Struct already exists
db.createTable<Struct>()=false / DATABASE_ERROR(5601)
db.hasTable(typename(Struct))=true / ok

```

7.6.6 检查数据库中是否存在表

内置的 *DatabaseTableExists* 函数允许通过其名称检查表是否存在。

bool DatabaseTableExists(int database, const string table)

数据库描述符和表名在参数中指定。如果表存在，函数调用的结果为 *true*。

我们通过添加 *hasTable* 方法来扩展 *DBSQLite* 类。

```

class DBSQLite
{
    ...
    bool hasTable(const string table) const
    {
        return DatabaseTableExists(handle, table);
    }
}

```

DBcreateTable.mq5 脚本将检查表是否已出现。

```

void OnStart()
{
    DBSQLite db(Database);
    if(db.isOpen())
    {
        PRTF(db.execute(StringFormat("CREATE TABLE %s (msg text)", Table)));
        PRTF(db.hasTable(Table));
    }
}

```

再次强调，不必担心在尝试重新创建时可能收到错误。这丝毫不影响表的存在性。

```

database error, table table1 already exists
db.execute(StringFormat(CREATE TABLE %s (msg text),Table))=false / DATABASE_ERROR(560
db.hasTable(Table)=true / ok

```

由于我们正在编写一个通用的辅助类 *DBSQLite*，我们将在其中提供删除表的机制。SQL 为此目的提供了 DROP 命令。


```

class DBSQLite
{
    ...
    bool deleteTable(const string name) const
    {
        const static string query = "DROP TABLE '%s'";
        if(!DatabaseTableExists(handle, name)) return true;
        if(!DatabaseExecute(handle, StringFormat(query, name))) return false;
        return !DatabaseTableExists(handle, name)
            && ResetLastErrorOnCondition(_LastError == DATABASE_NO_MORE_DATA);
    }

    static bool ResetLastErrorOnCondition(const bool cond)
    {
        if(cond)
        {
            ResetLastError();
            return true;
        }
        return false;
    }
}

```

在执行查询之前，我们检查表是否存在，如果不存在则立即退出。

执行查询后，我们通过再次调用 *DatabaseTableExists* 来额外检查表是否已被删除。由于表不存在将用 DATABASE_NO_MORE_DATA 错误代码进行标记，这是此方法的预期结果，因此我们使用 *ResetLastErrorOnCondition* 清除该错误代码。

使用 SQL 的功能来排除删除不存在表的尝试可能更有效：只需在查询中添加短语 “IF EXISTS” 即可。因此，*deleteTable* 方法的最终版本得以简化：

```

bool deleteTable(const string name) const
{
    const static string query = "DROP TABLE IF EXISTS '%s'";
    return DatabaseExecute(handle, StringFormat(query, name));
}

```

你可以尝试编写一个用于删除表的测试脚本，但小心不要错误地删除了工作表。表会连同所有数据立即删除，不会提示确认，也无法恢复。对于重要项目，请保留数据库备份。

7.6.7 准备绑定查询：DatabasePrepare

在许多情况下，需要在 SQL 查询中嵌入参数。由于 SQL 查询“原本”是一个符合特殊语法的字符串，因此可以通过简单的 *StringFormat* 调用或通过连接，并在正确的位置添加参数值来构建。我们已经在创建表的查询 (“CREATE TABLE %s '%s' (%s);”) 中使用过这种技术，但这里只有部分参数包含数据（值列表替换了括号内的 %s），其余部分则表示一个选项和表名。在本节中，我们将专门关注向查询中替换数据。以原生的 SQL 方式执行此操作非常重要，原因有几个。

首先，SQL 查询仅作为字符串传递给 SQLite 引擎，查询在引擎中被解析为组件，检查正确性，并以某种方式“编译”（当然，不是由 MQL5 编译器进行编译）。然后，编译后的查询由数据库执行。这就是为什么我们把“原本”这个词放在引号里。

当需要使用不同参数执行相同的查询时（例如，向表中插入许多记录；我们正在逐步接近这个任务），为每条记录单独编译和检查查询效率相当低下。更正确的做法是编译查询一次，然后批量执行它，只需替换不同的值即可。

这种编译操作称为查询准备，由 *DatabasePrepare* 函数执行。

预置查询还有另一个目的：借助它们，SQLite 引擎将查询执行结果返回给 MQL5 代码（你将在[执行预置查询](#)和[单独读取查询结果记录字段](#)等章节中找到更多相关内容）。

与参数化查询相关的最后但同样重要的一点是，它们能保护你的程序免受潜在的黑客攻击，即所谓的 SQL 注入。首先，这对于公共站点的数据库至关重要，在这些站点中，用户输入的信息会嵌入到 SQL 查询中而记录到数据库：如果在这种情况下使用简单的格式替换（如 %s'），用户将能够输入一个包含额外 SQL 命令的长字符串来代替预期的数据，这将成为原始 SQL 查询的一部分，从而扭曲其含义。但是，如果 SQL 查询已经编译完成，它就不能被输入数据更改：输入数据始终被视作数据处理。

尽管 MQL 程序不是服务器程序，但仍然可以将从用户那里接收到的信息存储在数据库中。

int DatabasePrepare(int database, const string sql, ...)

DatabasePrepare 函数在指定的数据库中为 *sql* 字符串中的查询创建一个句柄。*database* 必须事先通过 *DatabaseOpen* 函数打开。

查询参数的位置在 *sql* 字符串中使用片段 '?1'、'?2'、'?3' 等来指定。该编号表示将来在 *DatabaseBind* 函数中为其分配输入值时使用的参数索引。*sql* 字符串中的数字不需要按顺序排列，如果同一参数需要插入到查询的不同位置，也可以重复使用。

注意！替换片段 '?n' 中的索引从 1 开始，而在 *DatabaseBind* 函数中从 0 开始。例如，查询主体中的 '?1' 参数将在调用索引为 0 的 *DatabaseBind* 时获取值，'?2' 参数在索引为 1 时获取值，依此类推。即使 '?n' 参数的编号存在间隙（无论是意外还是有意为之），这个固定的偏移量 1 也会保持不变。

如果你计划严格按顺序绑定所有参数，可以使用缩写表示法：在每个参数的位置，只需指明符号 '?' 而不带数字即可：在这种情况下，参数会自动编号。任何不带数字的参数 '?' 都会获得一个比其左侧已读取参数（带有显式数字或根据相同原则计算得出）的最大编号大 1 的数字，而第一个参数将获得编号 1，即 '?1'。

因此，请求

```
SELECT * FROM table WHERE risk > ?1 AND signal = ?2
```

等同于：

```
SELECT * FROM table WHERE risk > ? AND signal = ?
```

如果某些参数是常量，或者查询是为了获取结果而一次性准备执行的，则参数值可以作为逗号分隔的列表传递给 *DatabasePrepare* 函数，而不是省略号（与 *Print* 或 *Comment* 函数相同）。

查询参数只能用于设置表列中的值（在写入、更改或筛选条件时）。表名、列名、选项和 SQL 关键字不能通过 '?/' 参数传递。

DatabasePrepare 函数本身并不执行查询。从它返回的句柄必须随后传递给 *DatabaseRead* 或 *DatabaseReadBind* 函数调用。这些函数会执行查询并使结果可供读取（它可以是一条记录或多条记录）。当然，如果查询中存在参数占位符（'?' 或 '?n'），并且在 *DatabasePrepare* 中没有为它们指定值，那么在执行查询之前，你需要使用适当的 *DatabaseBind* 函数来绑定参数和数据。

如果没有为参数分配值，则在查询执行期间将为其替换为 NULL。

如果发生错误，*DatabasePrepare* 函数将返回 `INVALID_HANDLE`。

DatabasePrepare 的使用示例将在后续章节中探讨了与预置查询相关的其他特性之后介绍。

7.6.8 删除和重置预置查询

由于预置查询可以多次执行，在针对不同参数值的循环中，需要在每次迭代时将查询重置到初始状态。这是由 *DatabaseReset* 函数完成的。但如果预置查询只执行一次，则调用它没有意义。

`bool DatabaseReset(int request)`

该函数将内部编译的查询结构重置到初始状态，类似于调用 *DatabasePrepare*。但是，*DatabaseReset* 不会重新编译查询，因此速度非常快。

同样重要的是，如果已进行任何数据绑定，该函数不会重置查询中已建立的数据绑定。因此，如果需要，你可以只更改一个或少量参数的值。然后，在调用 *DatabaseReset* 之后，你只需为已更改的参数调用 *DatabaseBind* 函数即可。

在撰写本书时，MQL5 API 没有提供重置数据绑定的函数，即标准 SQLite 发行版中类似于 *sqlite_clear_bindings* 函数的函数。

在 *request* 参数中，指定先前从 *DatabasePrepare* 获取的查询的有效句柄。如果你传递先前已用 *DatabaseFinalize* (见下文) 移除的查询句柄，将返回错误。

该函数返回操作状态标识：成功 (*true*) 或错误 (*false*)。

处理重复查询的一般原则如下面的伪代码所示。*DatabaseBind* 和 *DatabaseRead* 函数将在后续章节中描述，并将被“封装”到 ORM 类中。

```
struct Data // structure example
{
    long count;
    double value;
    string comment;
};
Data data[];
... // getting data array
int r =
    DatabasePrepare(db, "INSERT... (?, ?, ?)"); // compile query with parameters
for(int i = 0; i < ArraySize(data); ++i) // data loop
{
    DatabaseBind(r, 0, data[i].count); // make data binding to parameters
    DatabaseBind(r, 1, data[i].value);
    DatabaseBind(r, 2, data[i].comment);
    DatabaseRead(r); // execute request
    ... // analyze or save results
    DatabaseReset(r); // initial state at each iteration
}
DatabaseFinalize(r);
```

当不再需要预置查询后，你应该使用 *DatabaseFinalize* 释放它所占用的计算机资源。

`void DatabaseFinalize(int request)`

该函数删除在 *DatabasePrepare* 中创建的、具有指定句柄的查询。

如果传递了不正确的描述符，该函数会将 `ERR_DATABASE_INVALID_HANDLE` 记录到 `_LastError`。

使用 `DatabaseClose` 关闭数据库时，为其创建的所有查询句柄都会自动移除并失效。

我们通过一个新的 `DBQuery` 类来补充我们的 ORM 层 (`DBSQLite.mqh`)，用于处理预置查询。目前，它只包含 RAII 概念固有的初始化和反初始化功能，但我们很快会对其进行扩展。

```
class DBQuery
{
protected:
    const string sql; // query
    const int db;      // database handle (constructor argument)
    const int handle;  // prepared request handle

public:
    DBQuery(const int owner, const string s): db(owner), sql(s),
        handle(PRTF(DatabasePrepare(db, sql)))
    {
    }

    ~DBQuery()
    {
        DatabaseFinalize(handle);
    }

    bool isValid() const
    {
        return handle != INVALID_HANDLE;
    }

    virtual bool reset()
    {
        return DatabaseReset(handle);
    }
    ...
};
```

在 `DBSQLite` 类中，我们通过创建 `DBQuery` 的实例，在 `prepare` 方法中初始化请求的准备工作。所有查询对象都将以自动指针的形式存储在内部数组 `queries` 中，这使得调用代码无需关注它们的显式删除。

```
class DBSQLite
{
    ...
protected:
    AutoPtr<DBQuery> queries[];
public:
    DBQuery *prepare(const string sql)
    {
        return PUSH(queries, new DBQuery(handle, sql));
    }
    ...
};
```

7.6.9 将数据绑定到查询参数: DatabaseBind/Array

在 SQL 查询已通过 *DatabasePrepare* 函数编译后,你可以使用接收到的查询句柄将数据绑定到查询参数,这正是 *DatabaseBind* 和 *DatabaseBindArray* 函数的用途。这两个函数不仅可以在 *DatabasePrepare* 中创建查询后立即调用,也可以在通过 *DatabaseReset* 将请求重置到其初始状态后调用(如果请求在循环中多次执行)。

数据绑定步骤并非总是必需的,因为预置查询可能没有参数。一般来说,如果查询从 SQL 向 MQL5 返回数据,并因此需要查询描述符,便会发生这种情况:如何通过其句柄读取查询结果在关于 *DatabaseRead/DatabaseReadBind* 和 *DatabaseColumn*函数的章节中介绍。

bool DatabaseBind(int request, int index, T value)

DatabaseBind 函数为具有 *request* 句柄的查询设置 *index* 参数的值。默认情况下,如果查询中的参数用替换符号 '?'(不带数字)标记,则编号从 0 开始。然而,参数也可以在查询字符串中通过数字指定 @1、'5'、?21):在这种情况下,传递给该函数的实际索引必须比字符串中相应数字小 1。这是因为查询字符串中的编号从 1 开始。

例如,以下查询需要一个参数(索引 0):

```
int r = DatabasePrepare(db, "SELECT * FROM table WHERE id=?");
DatabaseBind(r, 0, 1234);
```

如果在查询字符串中使用了 "... id=?10" 替换,则需要调用索引为 9 的 *DatabaseBind*。

DatabaseBind 原型中的 *value* 可以是任何 **简单类型** 或字符串。如果参数需要映射复合类型数据(结构体)或可以表示为字节数组的任意二进制数据,请使用 *DatabaseBindArray* 函数。

函数成功时返回 *true*。否则,返回 *false*。

bool DatabaseBindArray(int request, int index, T &array[])

DatabaseBindArray 函数为具有 *request* 句柄的查询,将 *index* 参数的值设置为一个简单类型或简单结构体(包括字符串)的数组。此函数可用于向数据库写入 **BLOB** 和 **NULL**(表示缺少值,在 SQL 中被认为是单独类型且不等于 0)。

现在,我们回到 *DBSQLite.mqh* 文件中的 *DBQuery* 类,并添加数据绑定支持。

```

class DBQuery
{
    ...
public:
    template<typename T>
    bool bind(const int index, const T value)
    {
        return PRTF(DatabaseBind(handle, index, value));
    }
    template<typename T>
    bool bindBlob(const int index, const T &value[])
    {
        return PRTF(DatabaseBindArray(handle, index, value));
    }

    bool bindNull(const int index)
    {
        static const uchar null[] = {};
        return bindBlob(index, null);
    }
    ...
};

```

BLOB 适用于将任何文件原封不动地传输到数据库，例如，如果你首先使用 *FileLoad* 函数将其读入字节数组。

显式绑定空值的必要性不那么明显。向数据库插入新记录时，调用程序通常只传递其已知的字段，而所有缺失的字段（如果它们没有用 NOT NULL 约束标记，或者在表描述中没有不同的 DEFAULT 值）将由引擎自动保留为 NULL。然而，在使用 ORM 方法时，将整个对象写入数据库很方便，包括带有唯一主键（PRIMARY KEY）的字段。新对象尚不具有此标识符，因为数据库本身在首次写入对象时会添加它，因此将新对象中的此字段绑定到 NULL 值非常重要。

7.6.10 执行预置查询 DatabaseRead/Bind

预置查询是使用 *DatabaseRead* 和 *DatabaseReadBind* 函数执行的。第一个函数会从数据库中提取结果，以便稍后可以从响应中依次接收到的每条记录中读取单个字段；第二个函数则以结构体的形式完整地提取每个匹配的记录。

bool DatabaseRead(int request)

在 *Database Prepare* 或 *DatabaseReset* 之后首次调用时，*DatabaseRead* 函数会执行查询并将内部查询结果指针设置为检索到的第一条记录（如果查询期望返回记录）。而 *DatabaseColumn* 函数能够读取记录字段的值，即查询中指定的列。

在后续调用中，*DatabaseRead* 函数会跳转到查询结果中的下一条记录，直到到达末尾。

函数成功完成时返回 *true*。*false* 值用作错误指示（例如，数据库可能被锁定或繁忙），以及正常到达结果末尾时，因此应该分析 *_LastError* 中的代码。特别是，ERR_DATABASE_NO_MORE_DATA (5126) 值表示结果已结束。

注意！如果使用 *DatabaseRead* 执行不返回数据的查询（如 INSERT、UPDATE 等），并且请求成功，则该函数立即返回 *false* 并将错误代码设置为 ERR_DATABASE_NO_MORE_DATA。

该函数的通常使用模式通过以下伪代码说明（不同类型的 *DatabaseColumn* 函数将在 下一节中介绍）。

```
int r = DatabasePrepare(db, "SELECT... WHERE...?",
    param)); //compiling the query(optional with parameters)
while(DatabaseRead(r)) // query execution (on the first iteration)
{ // and loop through result records
    int count;
    DatabaseColumnInteger(r, 0, count); // read one field from the current record
    double number;
    DatabaseColumnDouble(r, 1, number); // read another field from the current record
    ... // column types and numbers in record are determined
    // process the received values of count, number
} // loop is interrupted when the end of the result is reached
DatabaseFinalize(r);
```

请注意，由于查询（读取条件数据）实际上只执行一次（在第一次迭代时），因此无需像记录变化数据时那样调用 *DatabaseReset*。但是，如果我们想再次运行查询并“遍历”新的结果，则需要调用 *DatabaseReset*。

bool DatabaseReadBind(int request, void &object)

DatabaseReadBind 函数的运作方式与 *DatabaseRead* 类似：第一次调用执行 SQL 查询，并在成功（结果中有合适的数据）的情况下，用第一条记录的字段填充通过引用传递的 *object* 结构体；后续调用继续在查询结果中移动内部指针，用下一条记录的数据填充该结构体。

该结构体必须仅包含数字类型和/或字符串作为成员（不允许数组），它不能继承自对象类型或包含对象类型的静态成员。

object 结构体中的字段数量不应超过查询结果中的列数；否则，我们将收到错误。可以使用 *DatabaseColumnsCount* 函数动态获取列数，但是，调用者通常需要根据原始请求预先“知道”预期的数据配置。

如果结构体中的字段数量少于记录中的字段数量，则将执行部分读取。其余数据可以使用相应的 *DatabaseColumn* 函数来获取。

假定结构体的字段类型与结果列中的数据类型匹配。否则，将执行自动的隐式转换，这可能导致意外后果（例如，读入数字字段的字符串将得到 0）。

在最简单的情况下，当我们为数据库记录计算某个总值时，例如，通过调用像 *SUM(column)*、*COUNT(column)* 或 *AVERAGE(column)* 这样的聚合函数，查询的结果将是具有单个字段的单个记录。

```
SELECT SUM(swap) FROM trades;
```

由于读取结果与 *DatabaseColumn* 函数相关，我们将把示例的开发推迟到下一节，下一节中介绍这些函数。

7.6.11 单独读取字段：DatabaseColumn 函数

通过 *DatabaseRead* 或 *DatabaseReadBind* 函数执行查询后，程序便有机会滚动浏览根据指定条件选择的记录。在每次迭代中，SQLite 引擎的内部结构中会分配一个特定的记录，该记录的字段（例）可通过 *DatabaseColumn* 函数组访问。

int DatabaseColumnsCount(int request)

根据查询描述符，该函数返回查询结果中的字段（列）数。如果出错，则返回 -1。

你甚至可以找出在 DatabasePrepare 中创建的查询的字段数量（在调用 DatabaseRead 函数之前）。对于其他 DatabaseColumn 函数，你通常应首先调用 DatabaseRead（至少一次）。

使用查询结果中字段的原始编号，程序可以找出字段名称（DatabaseColumnName）、类型（DatabaseColumnType）、大小（DatabaseColumnSize）以及相应类型的值（每种类型都有其对应的函数）。

bool DatabaseColumnName(int request, int column, string &name)

该函数用查询结果（request）中由编号（column）指定的列的名称填充通过引用传递的字符串参数（name）。

字段编号从 0 开始，不能超过 DatabaseColumnsCount() - 1 的值。这不仅适用于此函数，也适用于本节所有其他函数。

该函数成功时返回 true，如果出错则返回 false。

ENUM_DATABASE_FIELD_TYPE DatabaseColumnType(int request, int column)

DatabaseColumnType 函数返回查询结果当前记录中指定列的值的类型。可能的类型收集在 ENUM_DATABASE_FIELD_TYPE 枚举中。

标识符	说明
DATABASE_FIELD_TYPE_INVALID	获取类型错误，错误代码在 _LastError
DATABASE_FIELD_TYPE_INTEGER	整数
DATABASE_FIELD_TYPE_FLOAT	实数
DATABASE_FIELD_TYPE_TEXT	字符串
DATABASE_FIELD_TYPE_BLOB	二进制数据
DATABASE_FIELD_TYPE_NULL	空值（特殊类型 NULL）

关于 SQL 类型及其与 MQL5 类型的对应关系的更多细节已在 表结构（模式）：数据类型和限制一节中介绍过。

int DatabaseColumnSize(int request, int column)

该函数返回 request 查询结果当前记录中具有 column 索引的字段值的字节大小。例如，整数值可以用不同数量的字节表示（我们从 MQL5 类型中知道这一点，特别是 short/int/long）。

下一组函数允许你从记录的相应字段中获取特定类型的值。要从下一条记录中读取值，你需要再次调用 DatabaseRead。


```

bool DatabaseColumnText(int request, int column, string &value)
bool DatabaseColumnInteger(int request, int column, int &value)
bool DatabaseColumnLong(int request, int column, long &value)
bool DatabaseColumnDouble(int request, int column, double &value)
bool DatabaseColumnBlob(int request, int column, void &data[])

```

所有函数成功时都返回 `true`，并将字段值放入 `value` 接收变量中。唯一的特例是 **DatabaseColumnBlob** 函数，它传递一个任意简单类型或简单结构体的数组作为输出变量。通过指定 `uchar[]` 数组作为最通用的选项，你可以读取任何值的字节表示（包括使用 `DATABASE_FIELD_TYPE_BLOB` 类型标记的二进制文件）。

SQLite 引擎不会检查是否为列调用了与其类型相对应的函数。如果类型无意或有意不同，系统将自动隐式地将字段值转换为接收变量的类型。

现在，在熟悉了大多数 **Database** 函数之后，我们可以完成 **DBSQLite.mqh** 文件中一组 SQL 类的开发，并继续演示实际示例。

7.6.12 通过 ORM 对象在 SQLite 中执行 CRUD 操作的示例

我们已经学习了在数据库中实现信息完整生命周期所需的所有函数，即 CRUD（Create, Read, Update, Delete）。但在进行实践之前，我们需要完成 ORM 层。

从前面几节中，已经很清楚与数据库工作的单位是一条记录：它可以是数据库表中的一条记录，也可以是查询结果中的一个元素。为了在 ORM 层面读取单条记录，我们引入 **DBRow** 类。每条记录都由一个 SQL 查询生成，因此其句柄会传递给构造函数。

正如我们所知，一条记录可以由多个列组成，其数量和类型可让我们找到 **DatabaseColumn** 函数。为了使用 **DBRow** 向 MQL 程序公开这些信息，我们预留了相关变量：`columns` 以及一个 **DBRowColumn** 结构体数组（后者包含三个字段，用于存储列的名称、类型和大小）。

此外，**DBRow** 对象在必要时可以在自身内部缓存从数据库获取的值。为此，使用了 **MqlParam** 类型的 `<code>data</code> 数组。由于我们预先不知道特定列中将是什么类型的值，因此使用 MqlParam 作为一种在其他编程环境中可用的通用 Variant 类型。`

```

class DBRow
{
protected:
    const int query;
    int columns;
    DBRowColumn info[];
    MqlParam data[];
    const bool cache;
    int cursor;
    ...
public:
    DBRow(const int q, const bool c = false):
        query(q), cache(c), columns(0), cursor(-1)
    {
    }

    int length() const
    {
        return columns;
    }
    ...
};

```

cursor 变量跟踪查询结果中的当前记录号。在请求完成之前，*cursor* 等于 -1。

虚方法 *DBread* 负责执行查询；它调用 *DatabaseRead*。

```

protected:
    virtual bool DBread()
    {
        return PRTF(DatabaseRead(query));
    }

```

我们稍后会看到为什么需要一个虚方法。使用 *DBread* 的公共方法 *next* 提供了对结果记录的“滚动”浏览，看起来像这样。

```

public:
    virtual bool next()
    {
        ...
        const bool success = DBread();
        if(success)
        {
            if(cursor == -1)
            {
                columns = DatabaseColumnsCount(query);
                ArrayResize(info, columns);
                if(cache) ArrayResize(data, columns);
                for(int i = 0; i < columns; ++i)
                {
                    DatabaseColumnName(query, i, info[i].name);
                    info[i].type = DatabaseColumnType(query, i);
                    info[i].size = DatabaseColumnSize(query, i);
                    if(cache) data[i] = this[i]; // overload operator[](int)
                }
            }
            ++cursor;
        }
        return success;
    }
}

```

如果查询是第一次被访问，我们分配内存并读取列信息。如果请求了缓存，我们还会填充 *data* 数组。为此，会为每一列调用重载的运算符 `[]`。在其中，根据值的类型，我们调用适当的 *DatabaseColumn* 函数，并将结果值放入 *MqlParam* 结构体的一个或另一个字段中。

```

virtual MqlParam operator[](const int i = 0) const
{
    MqlParam param = {};
    if(i < 0 || i >= columns) return param;
    if(ArraySize(data) > 0 && cursor != -1) // if there is a cache, return from it
    {
        return data[i];
    }
    switch(info[i].type)
    {
    case DATABASE_FIELD_TYPE_INTEGER:
        switch(info[i].size)
        {
        case 1:
            param.type = TYPE_CHAR;
            break;
        case 2:
            param.type = TYPE_SHORT;
            break;
        case 4:
            param.type = TYPE_INT;
            break;
        case 8:
        default:
            param.type = TYPE_LONG;
            break;
        }
        DatabaseColumnLong(query, i, param.integer_value);
        break;
    case DATABASE_FIELD_TYPE_FLOAT:
        param.type = info[i].size == 4 ? TYPE_FLOAT : TYPE_DOUBLE;
        DatabaseColumnDouble(query, i, param.double_value);
        break;
    case DATABASE_FIELD_TYPE_TEXT:
        param.type = TYPE_STRING;
        DatabaseColumnText(query, i, param.string_value);
        break;
    case DATABASE_FIELD_TYPE_BLOB: // return base64 only for information we can't
    {
        // return binary data in MqlParam - exact
        uchar blob[]; // representation of binary fields is given by g
        DatabaseColumnBlob(query, i, blob);
        uchar key[], text[];
        if(CryptEncode(CRYPT_BASE64, blob, key, text))
        {
            param.string_value = CharArrayToString(text);
        }
    }
    param.type = TYPE_BLOB;
    break;
    case DATABASE_FIELD_TYPE_NULL:
        param.type = TYPE_NULL;

```

```

        break;
    }
    return param;
}

```

提供了 *getBlob* 方法以从 BLOB 字段中完整读取二进制数据（如果没有关于内容格式的更具体信息，请使用 *uchar* 类型作为 S 以获取字节数组）。

```

template<typename S>
int getBlob(const int i, S &object[])
{
    ...
    return DatabaseColumnBlob(query, i, object);
}

```

对于所描述的方法，执行查询并读取其结果的过程可以用以下伪代码表示（它将现有的 *DBSQLite* 和 *DBQuery* 类置于后台，但我们很快会将它们整合在一起）：

```

int query = ...
DBRow *row = new DBRow(query);
while(row.next())
{
    for(int i = 0; i < row.length(); ++i)
    {
        StructPrint(row[i]); // print the i-th column as an MqlParam structure
    }
}

```

每次都显式地编写列循环并不优雅，因此该类提供了一种获取记录所有字段值的方法。

```

void readAll(MqlParam &params[]) const
{
    ArrayResize(params, columns);
    for(int i = 0; i < columns; ++i)
    {
        params[i] = this[i];
    }
}

```

此外，为了方便起见，该类还接收了运算符 '[]' 和 *getBlob* 方法的重载，用于通过字段名称而不是索引来读取字段。例如，

```

class DBRow
{
    ...
public:
    int name2index(const string name) const
    {
        for(int i = 0; i < columns; ++i)
        {
            if(name == info[i].name) return i;
        }
        Print("Wrong column name: ", name);
        SetUserError(3);
        return -1;
    }

    MqlParam operator[](const string name) const
    {
        const int i = name2index(name);
        if(i != -1) return this[i]; // operator()[int] overload
        static MqlParam param = {};
        return param;
    }
    ...
};

```

这样你就可以访问选定的列。

```

int query = ...
DBRow *row = new DBRow(query);
for(int i = 1; row.next(); )
{
    Print(i++, " ", row["trades"], " ", row["profit"], " ", row["drawdown"]);
}

```

但是，仍然像 *MqlParam* 数组那样单独获取记录的元素，不能称为真正的 OOP 方法。最好是将整个数据库表记录读入一个对象，一个应用程序结构体中。回想一下，MQL5 API 提供了一个合适的函数：*DatabaseReadBind*。这样，我们就获得一种优势，能够描述派生类 *DBRow* 并覆盖其虚方法 *DBRead*。

这个 *DBRowStruct* 类是一个模板，并期望参数 *S* 是允许在 *DatabaseReadBind* 中绑定的简单结构体之一。

```

template<typename S>
class DBRowStruct: public DBRow
{
protected:
    S object;

    virtual bool DBread() override
    {
        // NB: inherited structures and nested structures are not allowed;
        // count of structure fields should not exceed count of columns in table/query
        return PRTF(DatabaseReadBind(query, object));
    }

public:
    DBRowStruct(const int q, const bool c = false): DBRow(q, c)
    {
    }

    S get() const
    {
        return object;
    }
};

```

有了派生类, 我们几乎可以无缝地从基类获取对象。

```

int query = ...
DBRowStruct<MyStruct> *row = new DBRowStruct<MyStruct>(query);
MyStruct structs[];
while(row.next())
{
    PUSH(structs, row.get());
}

```

现在就可以将 *DBRow/DBRowStruct* 与 *DBQuery* 关联起来, 将伪代码转换为工作代码了。在 *DBQuery* 中, 我们向 *DBRow* 对象添加一个自动指针, 该对象将包含来自查询结果的当前记录的数据 (如果已执行)。使用自动指针使调用代码无需担心释放 *DBRow* 对象: 它们要么随 *DBQuery* 一起删除, 要么在由于查询重新启动 (如果需要) 而重新创建时删除。*DBRow* 或 *DBRowStruct* 对象的初始化由模板方法 *start* 完成。

```

class DBQuery
{
protected:
    ...
    AutoPtr<DBRow> row;    // current entry
public:
    DBQuery(const int owner, const string s): db(owner), sql(s),
        handle(PRTF(DatabasePrepare(db, sql)))
    {
        row = NULL;
    }

    template<typename S>
    DBRow *start()
    {
        DatabaseReset(handle);
        row = typename(S) == "DBValue" ? new DBRow(handle) : new DBRowStruct<S>(handle)
        return row[];
    }
}

```

DBValue 类型是一个虚拟结构体, 仅用于指示程序创建底层的 **DBRow** 对象, 而不违反带有 **DatabaseReadBind** 调用的代码行的可编译性。

通过 **start** 并对请求进行如下准备之后, 此前展示的所有伪代码片段便可以正常运行了:

```

DBSQLite db("MQL5Book/DB/Example1"); // open base
DBQuery *query = db.prepare("PRAGMA table_xinfo('Struct')"); // prepare the request
DBRowStruct<DBTableColumn> *row = query.start<DBTableColumn>(); // get object cursor
DBTableColumn columns[]; // receiving array of
while(row.next()) // loop while there are records in the query result
{
    PUSH(columns, row.get()); // getting an object from the current record
}
ArrayPrint(columns);

```

此示例从数据库中读取有关特定表配置的元信息 (我们在 [执行没有 MQL5 数据绑定的查询](#) 一节的 **DBTableColumn** 示例中创建了它): 每一列都由一条包含多个字段的记录描述 (SQLite 标准), 这在结构体 **DBTableColumn** 中形式化了。

```

struct DBTableColumn
{
    int cid; // identifier (serial number)
    string name; // name
    string type; // type
    bool not_null; // attribute NOT NULL (yes/no)
    string default_value; // default value
    bool primary_key; // PRIMARY KEY sign (yes/no)
};

```

为了让用户不必每次都编写将结果记录转换为结构体对象的循环, **DBQuery** 类提供了一个模板方法 **readAll**, 该方法用查询结果中的信息填充一个引用的结构体数组。类似的 **readAll** 方法会填充一个指向 **DBRow** 对象的指针数组 (这更适合接收包含来自不同表的列的综合查询结果)。

在四重操作中, `DBRowStruct::get` CRUD 方法负责字母 R (Read)。为了使对象的读取功能更完整, 我们将支持按标识符从数据库中进行对象的点恢复。

SQLite 数据库中的绝大多数表都有一个主键 `rowid` (除非开发者出于某种原因在描述中使用了 “WITHOUT ROW ID” 选项), 因此新的 `read` 方法将接受一个键值作为参数。默认情况下, 表名假定等于接收结构体的类型, 但可以通过 `table` 参数更改为替代类型。考虑到这样的请求是一次性的并且应该返回一条记录, 将 `read` 方法直接放在 `DBSQLite` 类中并在内部管理短暂存活的 `DBQuery` 和 `DBRowStruct<S>` 对象是有意义的。

```
class DBSQLite
{
    ...
public:
    template<typename S>
    bool read(const long rowid, S &s, const string table = NULL,
              const string column = "rowid")
    {
        const static string query = "SELECT * FROM '%s' WHERE %s=%ld;";
        const string sql = StringFormat(query,
            StringLen(table) ? table : typename(S), column, rowid);
        PRTF(sql);
        DBQuery q(handle, sql);
        if(!q.isValid()) return false;
        DBRowStruct<S> *r = q.start<S>();
        if(r.next())
        {
            s = r.get();
            return true;
        }
        return false;
    }
};
```

主要工作由 SQL 查询 “SELECT *FROM ‘%s’ WHERE %s=%ld;” 完成, 该查询通过匹配 `rowid` 键从指定表中返回包含所有字段的记录。

现在你可以像这样从数据库创建一个特定对象 (假定我们感兴趣的标识符必须存储在某处)。

```
DBSQLite db("MQL5Book/DB/Example1");
long rowid = ... // id in the identifier
Struct s;
if(db.read(rowid, s))
    StructPrint(s);
```

最后, 在一些需要最大查询灵活性的复杂情况下 (例如, 多个表的组合, 通常是带有 JOIN 的 SELECT, 或嵌套查询), 我们仍然必须允许使用显式的 SQL 命令来获取选择结果, 尽管这违反了 ORM 原则。这种可能性由 `DBSQLite::prepare` 方法开启, 我们已经在 [管理预置查询](#) 一节中介绍过。

我们已经考虑了所有主要的读取方式。

然而, 我们还没有任何东西可以从数据库中读取, 因为我们跳过了添加记录的步骤。

我们尝试实现对象的创建 (C)。回想一下, 在我们的对象概念中, 结构体类型半自动地定义数据库表 (使用 `DB_FIELD` 宏)。例如, **Struct** 结构体允许在数据库中创建一个名为 “Struct” 的表, 其列集与该结构体的字段相对应。我们通过 **DBSQLite** 类中的模板方法 `createTable` 提供了此功能。现在, 同样需要编写一个模板方法 `insert`, 它将向此表中添加一条记录。

一个结构体对象被传递给该方法, 对于其类型, 必须存在已填充的 `DBEntity<S>::prototype<S>` 数组 (它由宏填充)。借助这个数组, 我们可以形成一个参数列表 (更准确地说, 是它们的替代符 ‘?’n’): 这是由静态方法 `qlist` 完成的。然而, 查询的准备工作仍然只完成了一半。在下面的代码中, 我们将需要根据对象的特性绑定输入数据。

“INSERT” 命令中添加了 “RETURNING row id” 语句, 因此当查询成功时, 我们期望得到一个包含单个值 (新 `rowid`) 的结果行。

```
class DBSQLite
{
    ...
public:
    template<typename S>
    long insert(S &object, const string table = NULL)
    {
        const static string query = "INSERT INTO '%s' VALUES(%s) RETURNING rowid;";
        const int n = ArrayRange(DBEntity<S>::prototype, 0);
        const string sql = StringFormat(query,
            StringLen(table) ? table : typename(S), qlist(n));
        PRTF(sql);
        DBQuery q(handle, sql);
        if(!q.isValid()) return 0;
        DBRow *r = q.start<DBValue>();
        if(object.bindAll(q))
        {
            if(r.next()) // the result should be one record with one new rowid value
            {
                return object.rowid(r[0].integer_value);
            }
        }
        return 0;
    }

    static string qlist(const int n)
    {
        string result = "?1";
        for(int i = 1; i < n; ++i)
        {
            result += StringFormat(",?%d", (i + 1));
        }
        return result;
    }
};
```

insert 方法的源代码中有一点需要特别注意。为了将值绑定到查询参数, 我们调用 `object.bindAll(q)` 方法。这意味着在你想要与数据库集成的应用程序结构体中, 你需要实现这样一个方法, 该方法为引擎提供所有成员变量。

此外，为了识别对象，假定存在一个带有主键的字段，并且只有该对象“知道”这个字段是什么。因此，该结构体具有 *rowid* 方法，它具有双重作用：首先，它将在数据库中分配的记录标识符传输给对象；其次，如果之前已经分配过，它允许从对象中找出这个标识符。

用于更改记录的 *DBSQLite::update* (U) 方法在许多方面与 *insert* 类似，因此建议你自行熟悉它。其基础是 SQL 查询 “UPDATE %s SET (%s)=(%s) WHERE rowid=%d;”，该查询应该传递结构体的所有字段 (*bindAll()* 对象) 和键 (*rowid()* 对象)。

最后，我们提到通过对象进行的记录的点删除 (D) 是在 *DBSQLite::remove* 方法中实现的 (单词 *delete* 是 MQL5 的一个运算符)。

我们在一个示例脚本 *DBfillTableFromStructArray.mq5* 中展示所有方法，其中定义了新 *Struct* 结构体。

我们将把几种常用类型的值作为该结构体的字段。

```
struct Struct
{
    long id;
    string name;
    double number;
    datetime timestamp;
    string image;
    ...
};
```

在字符串字段 *image* 中，调用代码将指定图形资源的名称或文件名，在绑定到数据库时，相应的二进制数据将作为 BLOB 复制。随后，当我们从数据库中将数据读入 *Struct* 对象时，二进制数据最终会进入 *image* 字符串，但当然会有失真 (因为字符串会在第一个空字节处中断)。要从数据库中准确提取 BLOB，你需要调用 *DBRow::getBlob* 方法 (基于 *DatabaseColumnBlob*)。

创建关于 *Struct* 结构体字段的元信息会提供以下宏。MQL 程序可以基于它们自动在数据库中为 *Struct* 对象创建一个表，并根据对象的特性初始化传递给查询的数据的绑定 (这种绑定不应与用于获取查询结果的反向绑定，即 *DatabaseReadBind*，相混淆)。

```
DB_FIELD_C1(Struct, long, id, DB_CONSTRAINT::PRIMARY_KEY);
DB_FIELD(Struct, string, name);
DB_FIELD(Struct, double, number);
DB_FIELD_C1(Struct, datetime, timestamp, DB_CONSTRAINT::CURRENT_TIMESTAMP);
DB_FIELD(Struct, blob, image);
```

为了填充一个小的测试结构体数组，该脚本具有输入变量：它们指定了三种货币，其报价将进入 *number* 字段。我们还在脚本中嵌入了两个标准图像，以测试 BLOB 的处理：它们将进入 *image* 字段。*timestamp* 字段将由我们的 ORM 类自动填充为记录的当前插入或修改时间戳。*id* 字段中的主键将必须由 SQLite 本身填充。

```

#resource "\\Images\\euro.bmp"
#resource "\\Images\\dollar.bmp"

input string Database = "MQL5Book/DB/Example2";
input string EURUSD = "EURUSD";
input string USDCNH = "USDCNH";
input string USDJPY = "USDJPY";

```

由于输入查询变量 (那些相同的 `m`) 的值最终是使用 *DatabaseBind* 或 *DatabaseBindArray* 函数按编号绑定的, 方法中的 *bindAll* 结构体应建立编号与其字段之间的对应关系: 假定按声明顺序进行简单编号。

```

struct Struct
{
    ...
    bool bindAll(DBQuery &q) const
    {
        uint pixels[] = {};
        uint w, h;
        if(StringLen(image)) // load binary data
        {
            if(StringFind(image, "::") == 0) // this is a resource
            {
                ResourceReadImage(image, pixels, w, h);
                // debug/test example (not BMP, no header)
                FileSave(StringSubstr(image, 2) + ".raw", pixels);
            }
            else // it's a file
            {
                const string res = "::" + image;
                ResourceCreate(res, image);
                ResourceReadImage(res, pixels, w, h);
                ResourceFree(res);
            }
        }
        // when id = NULL, the base will assign a new rowid
        return (id == 0 ? q.bindNull(0) : q.bind(0, id))
            && q.bind(1, name)
            && q.bind(2, number)
            // && q.bind(3, timestamp) // this field will be autofilled CURRENT_TIMESTAMP
            && q.bindBlob(4, pixels);
    }
    ...
};

```

rowid 方法非常简单。

```

struct Struct
{
    ...
    long rowid(const long setter = 0)
    {
        if(setter) id = setter;
        return id;
    }
};

```

定义结构体后，我们描述一个包含 4 个元素的测试数组。其中只有 2 个附加了图像。所有对象都具有零标识符，因为它们尚未在数据库中。

```

Struct demo[] =
{
    {0, "dollar", 1.0, 0, "::Images\\dollar.bmp"},
    {0, "euro", SymbolInfoDouble(EURUSD, SYMBOL_ASK), 0, "::Images\\euro.bmp"},
    {0, "yuan", 1.0 / SymbolInfoDouble(USDCNH, SYMBOL_BID), 0, NULL},
    {0, "yen", 1.0 / SymbolInfoDouble(USDJPY, SYMBOL_BID), 0, NULL},
};

```

在主函数 *OnStart* 中，我们创建或打开一个数据库（默认为 *MQ5Book/DB/Example2.sqlite*）。为以防万一，我们尝试删除 “Struct” 表，以确保重复执行脚本时的结果可重现性和调试，然后我们将为 *Struct* 结构体创建一个表。

```

void OnStart()
{
    DBSQLite db(Database);
    if(!PRTF(db.isOpen())) return;
    PRTF(db.deleteTable(typename(Struct)));
    if(!PRTF(db.createTable<Struct>(true))) return;
    ...
}

```

我们使用循环，而不是逐个添加对象：

```

// -> this option (set aside)
for(int i = 0; i < ArraySize(demo); ++i)
{
    PRTF(db.insert(demo[i])); // get a new rowid on each call
}

```

在这个循环中，我们将使用 *insert* 方法的另一种实现，它一次性接收一个对象数组作为输入并将其在单个请求中处理，这样效率更高（但该方法的总体思路是前面考虑的单个对象的 *insert* 方法）。

```

db.insert(demo); // new rowids are placed in objects
ArrayPrint(demo);
...

```

现在，我们尝试根据某些条件从数据库中选择记录，例如，那些没有分配图像的记录。为此，让我们准备一个包装在 *DBQuery* 对象中的 SQL 查询，然后我们以两种方式获取其结果：通过绑定到 *Struct* 结构体或通过通用类 *DBRow* 的实例。

```

DBQuery *query = db.prepare(StringFormat("SELECT * FROM %s WHERE image IS NULL",
    typename(Struct)));

// approach 1: application type of the Struct structure
Struct result[];
PRTF(query.readAll(result));
ArrayPrint(result);

query.reset(); // reset the query to try again

// approach 2: generic DBRow record container with MqlParam values
DBRow *rows[];
query.readAll(rows); // get DBRow objects with cached values
for(int i = 0; i < ArraySize(rows); ++i)
{
    Print(i);
    MqlParam fields[];
    rows[i].readAll(fields);
    ArrayPrint(fields);
}
...

```

两种选项都应给出相同的结果，尽管呈现方式不同（见下面的日志）。

接下来，我们的脚本暂停 1 秒，以便我们可以注意到我们将要更改的后续条目时间戳的变化。

```

Print("Pause...");
Sleep(1000);
...

```

对于 **result[]** 数组中的对象，我们分配位于脚本旁边文件夹中的 “yuan.bmp” 图像。然后，我们更新数据库中的对象。

```

for(int i = 0; i < ArraySize(result); ++i)
{
    result[i].image = "yuan.bmp";
    db.update(result[i]);
}
...

```

运行脚本后，你可以在 MetaEditor 内置的数据库导航器中确保所有四条记录在数据库中都有 BLOB，以及前两条记录和后两条记录的时间戳差异。

我们演示二进制数据的提取。我们将首先看到 BLOB 如何映射到 **image** 字符串字段（二进制数据不用于日志，我们这样做仅为演示目的）。

```

const long id1 = 1;
Struct s;
if(db.read(id1, s))
{
    Print("Length of string with Blob: ", StringLen(s.image));
    Print(s.image);
}
...

```

然后，我们用 `getBlob` 读取完整数据（总长度大于上面的行）。

```

DBRow *r;
if(db.read(id1, r, "Struct"))
{
    uchar bytes[];
    Print("Actual size of Blob: ", r.getBlob("image", bytes));
    FileSave("temp.bmp.raw", bytes); // not BMP, no header
}

```

我们需要得到 `temp.bmp.raw` 文件，它与 `MQL5/Files/Images/dollar.bmp.raw` 相同，后者是在 `Struct::bindAll` 方法中为调试目的创建的。因此，很容易验证写入和读取的二进制数据的精确对应关系。

请注意，由于我们将资源的二进制内容存储在数据库中，它不是 BMP 源文件：资源会进行颜色归一化并存储一个无头像素数组以及关于图像的元信息。

运行该脚本会生成详细的日志。特别是，数据库和表的创建由以下几行标记。

```

db.isOpen()=true / ok
db.deleteTable(typename(Struct))=true / ok
sql=CREATE TABLE IF NOT EXISTS Struct (id INTEGER PRIMARY KEY,
name TEXT ,
number REAL ,
timestamp INTEGER CURRENT_TIMESTAMP,
image BLOB ); / ok
db.createTable<Struct>(true)=true / ok

```

用于插入对象数组的 SQL 查询准备一次，然后使用预先绑定的不同数据多次执行（此处仅显示一次迭代）。`DatabaseBind` 函数调用的数量与查询中的 ‘?’ 变量匹配（‘?’ 被我们的类自动替换为 `SQL STRFTIME(‘%s’)` 函数调用，以获取当前 UTC 时间戳）。

```

sql=INSERT INTO 'Struct' VALUES(?1,?2,?3,STRFTIME('%s'),?5) RETURNING rowid; / ok
DatabasePrepare(db,sql)=131073 / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseRead(query)=true / ok
...

```

接下来，一个已经分配了主键 `rowid` 的结构体数组输出到日志的第一列。

```

      [id]   [name] [number]           [timestamp]           [image]
[0]    1 "dollar"  1.00000 1970.01.01 00:00:00 "::Images\dollar.bmp"
[1]    2 "euro"   1.00402 1970.01.01 00:00:00 "::Images\euro.bmp"
[2]    3 "yuan"   0.14635 1970.01.01 00:00:00 null
[3]    4 "yen"    0.00731 1970.01.01 00:00:00 null

```

选择没有图像的记录会得到以下结果 我们用不同方法执行此查询两次:第一次我们填充 *Struct* 结构体数组,第二次是 *DBRow* 数组,从中为每个字段获取 *MqlParam* 形式的“值”)。

```

DatabasePrepare(db,sql)=196609 / ok
DatabaseReadBind(query,object)=true / ok
DatabaseReadBind(query,object)=true / ok
DatabaseReadBind(query,object)=false / DATABASE_NO_MORE_DATA(5126)
query.readAll(result)=true / ok
      [id] [name] [number]           [timestamp] [image]
[0]    3 "yuan"  0.14635 2022.08.20 13:14:38 null
[1]    4 "yen"   0.00731 2022.08.20 13:14:38 null
DatabaseRead(query)=true / ok
DatabaseRead(query)=true / ok
DatabaseRead(query)=false / DATABASE_NO_MORE_DATA(5126)
0
      [type] [integer_value] [double_value] [string_value]
[0]    4           3           0.00000 null
[1]   14           0           0.00000 "yuan"
[2]   13           0           0.14635 null
[3]   10      1661001278       0.00000 null
[4]    0           0           0.00000 null
1
      [type] [integer_value] [double_value] [string_value]
[0]    4           4           0.00000 null
[1]   14           0           0.00000 "yen"
[2]   13           0           0.00731 null
[3]   10      1661001278       0.00000 null
[4]    0           0           0.00000 null
...

```

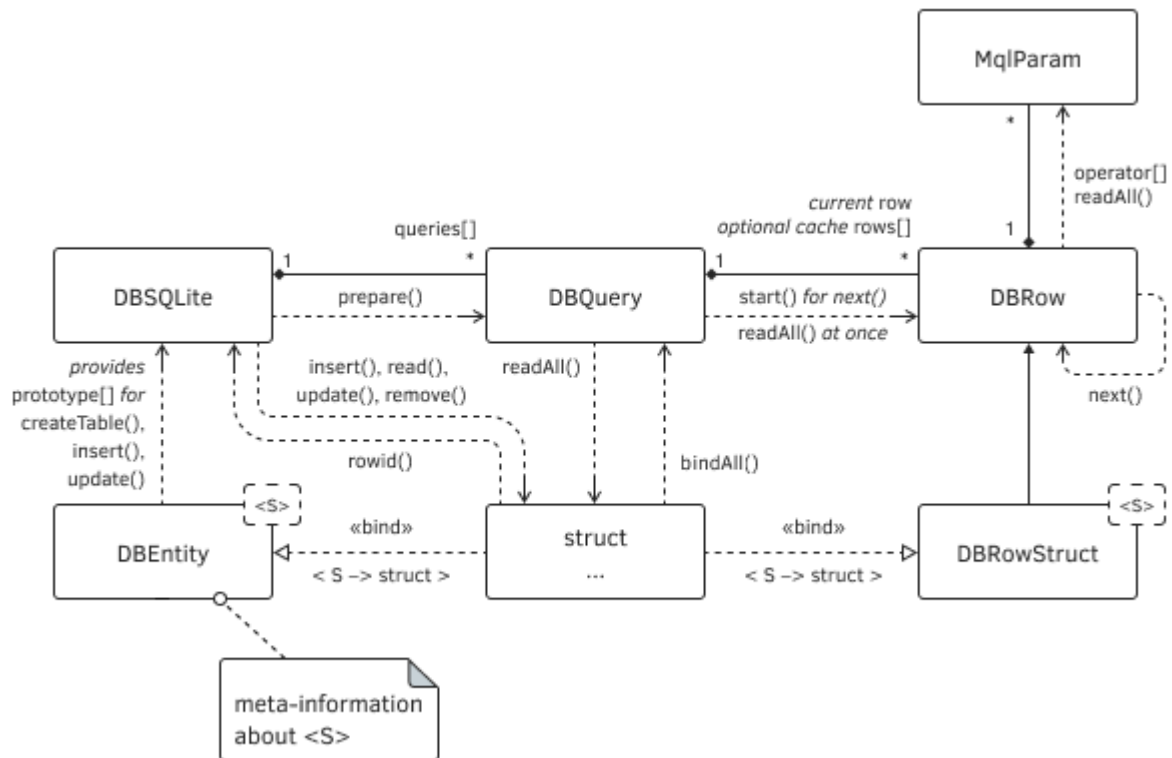
脚本的第二部分更新了几个找到的没有图像的记录,并向它们添加了 BLOB。


```
Pause...
sql=UPDATE 'Struct' SET (id,name,number,timestamp,image)=
    (?1,?2,?3,STRFTIME('%s'),?5) WHERE rowid=3; / ok
DatabasePrepare(db,sql)=262145 / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseRead(handle)=false / DATABASE_NO_MORE_DATA(5126)
sql=UPDATE 'Struct' SET (id,name,number,timestamp,image)=
    (?1,?2,?3,STRFTIME('%s'),?5) WHERE rowid=4; / ok
DatabasePrepare(db,sql)=327681 / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseRead(handle)=false / DATABASE_NO_MORE_DATA(5126)
...
```

最后, 可通过以下两种方式获取二进制数据: 不兼容方式和兼容方式, 前者通过 `image` 字符串字段作为读取整个 `DatabaseReadBind` 对象的结果 (这样做只是为了在日志中可视化字节序列); 后者通过 `DatabaseRead` 和 `DatabaseColumnBlob`, 这两种方式给出的结果不同: 当然, 第二种方法是正确的: 4096 字节的 BLOB 的长度和内容都得到了恢复。

```
sql=SELECT * FROM 'Struct' WHERE rowid=1; / ok  
DatabasePrepare(db,sql)=393217 / ok  
DatabaseReadBind(query,object)=true / ok  
Length of string with Blob: 922  
 ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?  
sql=SELECT * FROM 'Struct' WHERE rowid=1; / ok  
DatabasePrepare(db,sql)=458753 / ok  
DatabaseRead(query)=true / ok  
Actual size of Blob: 4096
```

总结我们自己开发的 ORM 包装器的中间结果,我们呈现了其类的概括性方案。

ORM 类图 (MQL5 $\leftrightarrow$ SQL)

7.6.13 事务

SQLite 支持 *transactions*，即逻辑上相关的一组操作，这些操作要么全部执行，要么完全不执行，这确保了数据库中数据的一致性。

*transaction*的概念在数据库上下文中具有新的含义,不同于我们过去用来描述 交易事务的含义。交易事务是指对交易账户实体 (包括订单、成交和仓位) 的单独操作。

事务提供了数据库更改的 4 个主要特性:

- 原子性 (不可分割) - 事务成功完成后, 其中包含的所有更改都将进入数据库; 如果发生错误, 则任何更改都不会进入数据库。
- 一致性 - 数据库当前的正确状态只能更改为另一种正确状态 (根据应用程序逻辑, 排除中间状态)。
- 隔离性 - 当前连接的事务中的更改在该事务结束之前对同一数据库的其他连接不可见, 反之亦然, 在当前连接存在未完成事务时, 其他连接的更改在当前连接中也不可见。
- 持久性 - 成功事务的更改保证存储在数据库中。

这些特性的术语 (原子性、一致性、隔离性和持久性) 构成了数据库理论中众所周知的缩写 ACID。

即使程序的正常运行由于系统故障而中断,数据库也将保持其工作状态。

最常来说明事务的例子是银行系统，其中资金从一个客户的账户转移到另一个客户的账户。这应该影响两条客户余额记录：一条记录中的余额按转账金额减少，另一条则增加。如果这些更改中只有一个生效，将会打乱银行账户的平衡：根据哪个操作失败，转账金额可能会消失，或者相反，无中生有。

可以举一个更接近交易实践的例子，但基于“相反”的原则。事实是 MetaTrader 5 中记录订单、成交和仓位的系统并非事务性的。

特别是，正如我们从 [创建 EA 交易](#) 一章中了解到的，一个已触发的订单（市价或挂单），在活动订单列表中消失后，可能不会立即显示在仓位列表中。因此，为了分析实际结果，有必要在 MQL 程序中实现对交易环境更新（实际化）的等待。如果记账系统基于事务，那么订单的执行、历史记录中交易的注册以及仓位的出现都将被包含在一个事务中并相互协调。终端开发者选择了不同的方法：尽可能快速和异步地返回交易环境的任何修改，其完整性必须由 MQL 程序监控。

任何更改数据库的 SQL 命令（实际上，除了 SELECT 之外的所有命令）如果事先没有显式地进行处理，都将自动被包装在一个事务中。

MQL5 API 提供了 3 个用于管理事务的函数：*DatabaseTransactionBegin*、*DatabaseTransactionCommit* 和 *DatabaseTransactionRollback*。所有函数成功时返回 *true*，如果出错则返回 *false*。

bool DatabaseTransactionBegin(int database)

DatabaseTransactionBegin 函数在具有从 [DatabaseOpen](#) 获取的指定描述符的数据库中开始执行事务。

对数据库进行的所有后续更改都累积在内部事务缓存中，并且在调用 *DatabaseTransactionCommit* 函数之前不会进入数据库。

MQL5 中的事务不能嵌套：如果事务已经启动，则重新调用 *DatabaseTransactionBegin* 将返回错误标志并在日志中输出消息。

```
database error, cannot start a transaction within a transaction
DatabaseTransactionBegin(db)=false / DATABASE_ERROR(5601)
```

相应地，你不能多次尝试并完成事务。

bool DatabaseTransactionCommit(int database)

DatabaseTransactionCommit 函数结束先前在具有指定句柄的数据库中启动的事务，并应用所有累积的更改（保存它们）。如果 MQL 程序启动一个事务但在关闭数据库之前未应用它，则所有更改都将丢失。

如果需要，程序可以撤销事务，从而撤销自事务开始以来的所有更改。

bool DatabaseTransactionRollback(int database)

DatabaseTransactionRollback 函数对包含 *database* 句柄的数据库先前启动的事务中所含的所有操作执行“回滚”。

我们来完成 *DBSQLite* 类中用于处理事务的方法，同时考虑到它们嵌套的限制，我们将在 *transaction* 变量中计算这个限制。如果为 0，则 *begin* 方法通过调用 *DatabaseTransactionBegin* 启动一个事务。所有后续启动事务的尝试只会增加计数器。在 *commit* 方法中，我们递减计数器，当计数器达到 0 时，我们调用 *DatabaseTransactionCommit*。

```

class DBSQLite
{
protected:
    int transaction;
    ...
public:
    bool begin()
    {
        if(transaction > 0)    // already in transaction
        {
            transaction++;    // keep track of the nesting level
            return true;
        }
        return (bool)(transaction = PRTF(DatabaseTransactionBegin(handle)));
    }

    bool commit()
    {
        if(transaction > 0)
        {
            if(--transaction == 0) // outermost transaction
                return PRTF(DatabaseTransactionCommit(handle));
        }
        return false;
    }

    bool rollback()
    {
        if(transaction > 0)
        {
            if(--transaction == 0)
                return PRTF(DatabaseTransactionRollback(handle));
        }
        return false;
    }
};

```

此外，我们创建 *DBTransaction* 类，它将允许在块（例如，函数）内部描述对象，确保在程序退出块时自动启动事务并随后应用（或取消）它。

```

class DBTransaction
{
    DBSQLite *db;
    const bool autocommit;
public:
    DBTransaction(DBSQLite &owner, const bool c = false): db(&owner), autocommit(c)
    {
        if(CheckPointer(db) != POINTER_INVALID)
        {
            db.begin();
        }
    }

    ~DBTransaction()
    {
        if(CheckPointer(db) != POINTER_INVALID)
        {
            autocommit ? db.commit() : db.rollback();
        }
    }

    bool commit()
    {
        if(CheckPointer(db) != POINTER_INVALID)
        {
            const bool done = db.commit();
            db = NULL;
            return done;
        }
        return false;
    }
};

```

使用此类对象的策略无需处理块 (函数) 各种退出选项。

```

void DataFunction(DBSQLite &db)
{
    DBTransaction tr(db);
    DBQuery *query = db.prepare("UPDATE..."); // batch changes
    ... // base modification
    if(... /* error1 */) return;                // automatic rollback
    ... // base modification
    if(... /* error2 */) return;                // automatic rollback
    tr.commit();
}

```

要使对象在任何阶段自动应用更改, 请在其构造函数的第二个参数中传递 *true*。

```

void DataFunction(DBSQLite &db)
{
    DBTransaction tr(db, true);
    DBQuery *query = db.prepare("UPDATE..."); // batch changes
    ... // base modification
    if(... /* condition1 */) return;           // automatic commit
    ... // base modification
    if(... /* condition2 */) return;           // automatic commit
    ...
} // automatic commit

```

你可以在循环内部描述 *DBTransaction* 对象，然后在每次迭代时，将启动并关闭一个单独的事务。

事务的演示将在 [使用 SQLite 搜索交易策略的示例](#)一节中提供。

7.6.14 数据库表的导入和导出

MySQL 允许将单个数据库表导出到 CSV 文件或从 CSV 文件导入。不提供将整个数据库作为带有 SQL 命令的文件进行导出/导入的功能。

long DatabasImport(int database, const string table, const string filename, uint flags, const string separator, ulong skip_rows, const string comment_chars)

DatabasImport 函数将指定文件中的数据导入到表中。前两个参数给出了打开的数据库描述符和表名。

如果名为 *table* 的表不存在，它将自动创建。表中字段的名称和类型将根据文件中包含的数据自动识别。

导入的文件不仅可以是现成的 CSV 文件，还可以是包含 CSV 文件的 ZIP 存档。filename 可能包含路径。将相对于 *MySQL/Files* 目录搜索文件。

ENUM_DATABASE_IMPORT_FLAGS 枚举中描述了可以按位组合的有效标志：

- DATABASE_IMPORT_HEADER -第一行包含表字段的名称
- DATABASE_IMPORT_CRLF -对于换行符，使用 CRLF 字符序列
- DATABASE_IMPORT_APPEND -将数据添加到现有表中
- DATABASE_IMPORT_QUOTED_STRINGS -字符串值在双引号内
- DATABASE_IMPORT_COMMON_FOLDER -终端的公共文件夹

separator 参数设置 CSV 文件中的分隔符字符。

skip_rows 参数可跳过文件中指定数量的前导行。

comment_chars 参数包含文件中用作注释标志的字符。以这些字符中任何一个开头的行都将被视作注释并且不会被导入。

函数返回导入的行数，如果出错则返回 -1。

long DatabaseExport(int database, const string table_or_sql, const string filename, uint flags, const string separator)

DatabaseExport 函数将表或 SQL 查询的结果导出到 CSV 文件。数据库句柄以及表名或查询文本在前两个参数中指定。

如果导出查询结果，则 SQL 查询必须以 “SELECT” 或 “select” 开头。换句话说，SQL 查询不能更改数据库状态；否则，**DatabaseExport** 将以错误结束。

文件名 **filename** 可能包含当前终端实例 **MQL5/Files** 目录内的路径，或终端的共享文件夹，具体取决于标志。

flags 参数用于指定控制文件格式和位置的标志组合。

- DATABASE_EXPORT_HEADER -输出带有字段名称的字符串
- DATABASE_EXPORT_INDEX -显示行号
- DATABASE_EXPORT_NO_BOM -不在文件开头插入标签 **BOM**（默认情况下会插入 BOM）
- DATABASE_EXPORT_CRLF -使用 CRLF 换行（默认为 LF）
- DATABASE_EXPORT_APPEND -将数据附加到现有文件的末尾（默认情况下，文件会被覆盖），如果文件不存在，则创建
- DATABASE_EXPORT_QUOTED_STRINGS -以双引号输出字符串值
- DATABASE_EXPORT_COMMON_FOLDER - CSV 文件将创建在所有终端的公共文件夹中 **MetaQuotes/ Terminal/ Common/ File**

separator 参数指定列分隔符字符。如果为 NULL，则将使用制表符 ‘\t’ 作为分隔符。空字符串 “” 被认为是有效的分隔符，但生成的 CSV 文件无法作为表读取，它将是一组行。

数据库中的文本字段可能包含换行符（\r 或 \r\n）以及在 **separator** 参数中指定的分隔符字符。在这种情况下，必须在 **flags** 参数中使用 DATABASE_EXPORT_QUOTED_STRINGS 标志。如果存在此标志，所有输出字符串都将用双引号括起来，并且如果字符串包含双引号，它将被替换为两个双引号。

该函数返回导出的记录数，如果出错则返回负值。

7.6.15 将表和 SQL 查询打印到日志

如果需要，MQL 程序可以使用 **DatabasePrint** 函数将表的内容或 SQL 查询的结果输出到日志。

long DatabasePrint(int database, const string table_or_sql, uint flags)

第一个参数传递数据库句柄，后跟表名或查询文本 (**table_or_sql**)。SQL 查询必须以 “SELECT” 或 “select” 开头，也就是说不能更改数据库的状态。否则，**DatabasePrint** 函数将以错误结束。

flags 参数指定了决定输出格式的标志组合。

- ⌚ DATABASE_PRINT_NO_HEADER -不显示表列名（字段名）
- ⌚ DATABASE_PRINT_NO_INDEX -不显示行号
- ⌚ DATABASE_PRINT_NO_FRAME -不显示分隔标头和数据的框架
- ⌚ DATABASE_PRINT_STRINGS_RIGHT -字符串进行右对齐

如果 **flags = 0**，则显示列和行，标头和数据由框架分隔，并且行进行左对齐。

函数返回显示的记录数，如果出错则返回 -1。

我们将在下一节中使用该函数。

遗憾的是，该函数不允许输出带有参数的 [预置查询](#)。如果存在参数，则需要在 MQL5 级别将它们嵌入到查询文本中。

7.6.16 使用 SQLite 搜索交易策略的示例

我们尝试使用 SQLite 来解决实际问题。我们将把带有报价历史的结构体导入 *MqlRates* 数据库，并对其进行分析，以识别模式并搜索潜在的交易策略。当然，任何选定的逻辑也可以在 MQL5 中实现，但 SQL 允许你以不同的方式来实现，在许多情况下更有效，并且可以使用许多有趣的内置 SQL 函数。本书的主题旨在学习 MQL5，不会深入研究这项技术，但我们提及它是值得算法交易者关注的。

用于将报价历史转换为数据库格式的脚本名为 *DBquotesImport.mq5*。在输入参数中，你可以设置数据库名称的前缀和事务的大小（一个事务中的记录数）。

```
input string Database = "MQL5Book/DB/Quotes";
input int TransactionSize = 1000;
```

为了使用我们的 ORM 层将 *MqlRates* 结构体添加到数据库，该脚本定义了一个辅助的 *MqlRatesDB* 结构体，它提供了将结构体字段绑定到数据库列的规则。由于我们的脚本只向数据库写入数据而不从数据库中读取数据，因此它不需要使用 *DatabaseReadBind* 函数进行绑定，这将对结构体的“简单性”施加一定的限制。如果没有这个限制，则可以从 *MqlRates* 派生 *MqlRatesDB* 结构体（并且不重复字段的描述）。


```

struct MqlRatesDB: public MqlRates
{
    /* for reference:

        datetime time;
        double    open;
        double    high;
        double    low;
        double    close;
        long      tick_volume;
        int       spread;
        long      real_volume;
    */

    bool bindAll(DBQuery &q) const
    {
        return q.bind(0, time)
            && q.bind(1, open)
            && q.bind(2, high)
            && q.bind(3, low)
            && q.bind(4, close)
            && q.bind(5, tick_volume)
            && q.bind(6, spread)
            && q.bind(7, real_volume);
    }

    long rowid(const long setter = 0)
    {
        // rowid is set by us according to the bar time
        return time;
    }
};

DB_FIELD_C1(MqlRatesDB, datetime, time, DB_CONSTRAINT::PRIMARY_KEY);
DB_FIELD(MqlRatesDB, double, open);
DB_FIELD(MqlRatesDB, double, high);
DB_FIELD(MqlRatesDB, double, low);
DB_FIELD(MqlRatesDB, double, close);
DB_FIELD(MqlRatesDB, long, tick_volume);
DB_FIELD(MqlRatesDB, int, spread);
DB_FIELD(MqlRatesDB, long, real_volume);

```

数据库名称由前缀 *Database*、当前运行脚本的图表的名称和时间周期构成。数据库中创建了单个名为 “MqlRatesDB” 的表，其字段配置由 DB_FIELD 宏指定。请注意，主键不会由数据库根据 *time* 字段（柱线开盘时间）生成，而是直接从柱线中获取。

```

void OnStart()
{
    Print("");
    DBSQLite db(Database + _Symbol + PeriodToString());
    if(!PRTF(db.isOpen())) return;

    PRTF(db.deleteTable(typename(MqlRatesDB)));

    if(!PRTF(db.createTable<MqlRatesDB>(true))) return;
    ...
}

```

接下来，我们使用 *TransactionSize* 个柱线的数据包，从历史记录中请求柱线并将它们添加到表中。这是由辅助函数 *ReadChunk* 完成的，只要有数据（函数返回 *true*）或用户不手动停止脚本，该函数就会一直被循环调用。函数代码如下所示。

```

int offset = 0;
while(ReadChunk(db, offset, TransactionSize) && !IsStopped())
{
    offset += TransactionSize;
}

```

过程完成后，我们向数据库查询表中生成的记录数并将其输出到日志。

```

DBRow *rows[];
if(db.prepare(StringFormat("SELECT COUNT(*) FROM %s",
    typename(MqlRatesDB))).readAll(rows))
{
    Print("Records added: ", rows[0][0].integer_value);
}
}

```

ReadChunk 函数如下所示。

```

bool ReadChunk(DBSQLite &db, const int offset, const int size)
{
    MqlRates rates[];
    MqlRatesDB ratesDB[];
    const int n = CopyRates(_Symbol, PERIOD_CURRENT, offset, size, rates);
    if(n > 0)
    {
        DBTransaction tr(db, true);
        Print(rates[0].time);
        ArrayResize(ratesDB, n);
        for(int i = 0; i < n; ++i)
        {
            ratesDB[i] = rates[i];
        }

        return db.insert(ratesDB);
    }
    else
    {
        Print("CopyRates failed: ", _LastError, " ", E2S(_LastError));
    }
    return false;
}

```

它通过内置的 *CopyRates* 函数调用来填充 *rates* 柱线数组。然后将柱线传输到 *ratesDB* 数组，以便仅使用一条语句 *db.insert(ratesDB)* 就可以将信息写入数据库（我们已经在 *MqlRatesDB* 中规范了如何正确执行此操作）。

块内部存在 *DBTransaction* 对象（启用了自动“提交”选项）意味着对数组的所有操作都被事务“覆盖”。为了指示进度，在处理每个柱线块期间，第一根柱线的标签会显示在日志中。

当 *CopyRates* 函数返回数据并且它们成功插入数据库时，*OnStart* 中的循环继续，并将复制的柱线编号向历史深处移动。当达到可用历史的末尾或终端设置中设置的柱线限制时，*CopyRates* 将返回错误 4401 (HISTORY_NOT_FOUND)，脚本将退出。

我们在 EURUSD, H1 图表上运行该脚本。日志应该显示类似这样的内容。

```

db.isOpen()=true / ok
db.deleteTable(typename(MqlRatesDB))=true / ok
db.createTable<MqlRatesDB>(true)=true / ok
2022.06.29 20:00:00
2022.05.03 04:00:00
2022.03.04 10:00:00
...
CopyRates failed: 4401 HISTORY_NOT_FOUND
Records added: 100000

```

我们现在可以基于 *QuotesEURUSDH1.sqlite* 进行实验，以测试各种交易假设。可以在 MetaEditor 中打开它以确保数据正确传输。

我们检查一个基于历史规律的最简单策略之一。我们将按日内时间和星期几细分，查找连续两根同向柱线的统计数据。如果对于某个时间和星期几的组合存在明显的优势，将来可以将其视为沿第一根柱线方向入市的信号。

首先, 让我们设计一个 SQL 查询, 该查询请求某个时期的报价并计算每个柱线上的价格变动, 即相邻开盘价之间的差额。

由于柱线的时间存储为秒数 (按照 MQL5 中 *datetime* 的标准, 并且同时也是 SQL 的 “Unix 纪元时间”), 因此最好将其显示转换为字符串以便于阅读, 所以我们从基于 `DATETIME` 函数的 *datetime* 字段开始 SELECT 查询:

```
SELECT
    DATETIME(time, 'unixepoch') as datetime, open, ...
```

该字段不参与分析, 此处仅供用户参考。之后, 显示价格以供参考, 以便我们可以通过调试打印来检查价格增量的计算。

由于我们将从整个文件中选择某个时期, 因此条件将需要 “纯粹形式” 的 *time* 字段, 并且也应将其添加到请求中。此外, 根据计划的报价分析, 我们需要从柱线标签中分离出其日内时间以及星期几 (它们的编号与 MQL5 中采用的编号一致, 0 是星期日)。我们分别调用查询的最后两列 *intraday* 和 *day*, 并使用 `TIME` 和 `STRFTIME` 函数来获取它们。

```
SELECT
    DATETIME(time, 'unixepoch') as datetime, open,
    time,
    TIME(time, 'unixepoch') AS intraday,
    STRFTIME('%w', time, 'unixepoch') AS day, ...
```

要在 SQL 中计算价格增量, 可以使用 `LAG` 函数。它返回指定列的值, 并带有指定行数的偏移量。例如, `LAG(X, 1)` 表示获取前一条记录中 *X* 的值, 第二个参数 1 表示偏移量默认为 1, 即可以省略以获得等效的条目 `LAG(X)`。要获取下一条记录的值, 请调用 `LAG(X, -1)`。在任何情况下, 使用 `LAG` 时, 都需要一个额外的语法结构来指定记录的排序顺序, 最简单的形式是 `OVER(ORDER BY column)`。

因此, 要获取两个相邻柱线开盘价之间的价格增量, 我们可以编写:

```
...
    (LAG(open, -1) OVER (ORDER BY time) - open) AS delta, ...
```

这一列具有预测性, 因为它展望了未来。

我们可以通过将增量与它们相乘来揭示两根柱线是否同向形成: 正值表示持续上涨或下跌:

```
...
    (LAG(open, -1) OVER (ORDER BY time) - open) * (open - LAG(open) OVER (ORDER BY time))
    AS product, ...
```

这个指标被选为计算中使用起来最简单的: 对于真实的交易系统, 你可以选择更复杂的标准。

为了评估系统在回测中产生的利润, 你需要将前一根柱线的方向 (它作为未来运动的指标) 乘以下一根柱线上的价格增量。方向在 *direction* 列中计算 (使用 `SIGN` 函数), 仅供参考。*estimate* 列中的利润估算是前一个运动 *direction* 与下一个柱线 (*delta*) 的乘积: 如果方向保持不变, 我们将获得正结果 (以点为单位)。

```
...
SIGN(open - LAG(open) OVER (ORDER BY time)) AS direction,
(LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORDER BY tim
AS estimate ...
```

在 SQL 命令的表达式中,你不能使用在同一命令中定义的 AS 别名。因此,我们不能将 *estimate* 定义为 *delta * direction*,而必须显式重复乘积的计算。然而,我们回想一下,*delta* 和 *direction* 列并非程序化分析所必需,此处添加它们仅为向用户可视化表格。

在 SQL 命令的末尾,我们指定从中进行选择的表,以及回溯日期范围的筛选条件:两个参数 "from" 和 "to"。

```
...
FROM MqlRatesDB
WHERE (time >= ?1 AND time < ?2)
```

(可选)我们可以添加一个约束 *LIMIT?*3 (并输入一些较小的值,例如 10),这样在首次对查询结果进行可视化验证时,就不必费力查看数万条记录。

你可以使用 *DatabasePrint* 函数来检查 SQL 命令的运行情况,然而,遗憾的是,该函数不允许处理带有参数的预置查询。因此,我们将不得不使用 *StringFormat* 将 SQL 参数占位符 'n' 替换为查询字符串格式化,并在那里替换参数值。或者,也可以完全避免使用 *DatabasePrint*,并独立地逐行将结果输出到日志 (通过 *DBRow* 数组)。

因此,请求的最终片段将变为:

```
...
WHERE (time >= %ld AND time < %ld)
ORDER BY time LIMIT %d;
```

应该注意的是,此查询中的 *datetime* 值将以“机器”格式从 MQL5 传入,即自 1970 年以来的秒数。如果想在 MetaEditor 中调试相同的 SQL 查询,那么使用日期字面量 (字符串)来编写日期范围条件会更方便,如下所示:

```
WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-01')
```

再次强调,这里我们需要使用 *STRFTIME* 函数 SQL 中的 %s 修饰符将指定的日期字符串转换为“Unix 纪元时间”标签;%s 类似于 MQL5 格式字符串纯属巧合)。

将设计好的 SQL 查询保存到一个单独的文本文件 *DBQuotesIntradayLag.sql*,并将其作为资源连接到同名的测试脚本 *DBQuotesIntradayLag.mq5*。

```
#resource "DBQuotesIntradayLag.sql" as string sql1
```

脚本的第一个参数允许你在数据库名称中设置一个前缀,在图表上以相同交易品种和时间周期运行 *DBQuotesImport.mq5* 后,该数据库应该已经存在。后续的输入用于日期范围和日志中调试打印的长度限制。

```
input string Database = "MQL5Book/DB/Quotes";
input datetime SubsetStart = D'2022.01.01';
input datetime SubsetStop = D'2023.01.01';
input int Limit = 10;
```

报价表是预先知道的,来自上一个脚本。

```
const string Table = "MqlRatesDB";
```

在 *OnStart* 函数中,我们打开数据库并确保报价表可用。

```

void OnStart()
{
    Print("");
    DBSQLite db(Database + _Symbol + PeriodToString());
    if(!PRTF(db.isOpen())) return;
    if(!PRTF(db.hasTable(Table))) return;
    ...
}

```

接下来，我们在 SQL 查询字符串中替换参数。我们不仅要注意将 SQL 参数 'n' 替换为格式序列，还要首先将百分号 % 加倍，否则 **StringFormat** 函数会将其视为自己的命令，而不会将它们传递到 SQL 中。

```

string sqlrep = sql1;
StringReplace(sqlrep, "%", "%%");
StringReplace(sqlrep, "?1", "%ld");
StringReplace(sqlrep, "?2", "%ld");
StringReplace(sqlrep, "?3", "%d");

const string sqlfmt = StringFormat(sqlrep, SubsetStart, SubsetStop, Limit);
Print(sqlfmt);

```

所有这些操作都只是为了在 **DatabasePrint** 函数的上下文中执行请求所必需的。在分析脚本的工作版本中，我们会读取查询结果并以编程方式分析它们，从而绕过格式化和调用 **DatabasePrint**。

最后，我们执行 SQL 查询并将结果表输出到日志中。

```

DatabasePrint(db.getHandle(), sqlfmt, 0);
}

```

以下是我们在 2022 年初对 10 条 EURUSD,H1 柱线所看到的内容。

```

db.isOpen()=true / ok
db.hasTable(Table)=true / ok
SELECT
DATETIME(time, 'unixepoch') as datetime,
open,
time,
TIME(time, 'unixepoch') AS intraday,
STRFTIME('%w', time, 'unixepoch') AS day,
(LAG(open,-1) OVER (ORDER BY time) - open) AS delta,
SIGN(open - LAG(open) OVER (ORDER BY time)) AS direction,
(LAG(open,-1) OVER (ORDER BY time) - open) * (open - LAG(open) OVER (ORDER BY time))
AS product,
(LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORDER BY time))
AS estimate
FROM MqlRatesDB
WHERE (time >= 1640995200 AND time < 1672531200)
ORDER BY time LIMIT 10;
#| datetime open time intraday day delta dir product estimate
--+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1| 2022-01-03 00:00:00 1.13693 1641168000 00:00:00 1 0.0003200098
2| 2022-01-03 01:00:00 1.13725 1641171600 01:00:00 1 2.999999e-05 1 9.5999478e-09 2.9
3| 2022-01-03 02:00:00 1.13728 1641175200 02:00:00 1 -0.001060006 1 -3.1799748e-08 -0
4| 2022-01-03 03:00:00 1.13622 1641178800 03:00:00 1 -0.0003400007 -1 3.6040028e-07 0
5| 2022-01-03 4:00:00 1.13588 1641182400 4:00:00 1 -0.001579991 -1 5.3719982e-07 0.00
6| 2022-01-03 05:00:00 1.1343 1641186000 05:00:00 1 0.0005299919 -1 -8.3739827e-07 -0
7| 2022-01-03 06:00:00 1.13483 1641189600 06:00:00 1 -0.0007699937 1 -4.0809905e-07 -
8| 2022-01-03 7:00:00 1.13406 1641193200 7:00:00 1 -0.0002600149 -1 2.0020098e-07 0.0
9| 2022-01-03 08:00:00 1.1338 1641196800 08:00:00 1 0.000510001 -1 -1.3260079e-07 -0.
10| 2022-01-03 9:00:00 1.13431 1641200400 9:00:00 1 0.0004800036 1 2.4480023e-07 0.00
...

```

很容易就能确保柱线的盘中时间以及星期 - 1 (对应周一) 分配正确。还可以检查 *delta* 增量。第一行的 *product* 和 *estimate* 值是空的, 因为它们需要计算缺失的前一行。

我们将 SQL 查询复杂化, 将具有相同时间组合 (*intraday*) 和星期组合 (*day*) 的记录分组, 并计算某个目标指标, 该指标可以说明每个组合的交易成功率。我们把平均单元大小 *product* 除以相同乘积的标准差作为这样一个指标。相邻柱线价格增量的平均乘积越大, 预期利润就越高, 而这些乘积的点差越小, 预测就越稳定。SQL 查询中的指标名称为 *objective*。

除目标指标外, 我们还将计算利润估计值 (*backtest_profit*) 和利润因子 (*backtest_PF*)。我们将以盘中时间和星期为背景, 将利润估算为所有柱线的价格增量总和 (*estimate*) (作为价格增量的开盘柱线的大小是以每个柱线的点数为单位的未来利润的模拟值)。利润系数传统上是正增量和负增量之商。

```

SELECT
AVG(product) / STDDEV(product) AS objective,
SUM(estimate) AS backtest_profit,
SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS backtest_PF,
intraday, day
FROM
(
SELECT
time,
TIME(time, 'unixepoch') AS intraday,
STRFTIME('%w', time, 'unixepoch') AS day,
(LAG(open,-1) OVER (ORDER BY time) - open) AS delta,
SIGN(open - LAG(open) OVER (ORDER BY time)) AS direction,
(LAG(open,-1) OVER (ORDER BY time) - open) * (open - LAG(open) OVER (ORDER BY time))
AS product,
(LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORDER BY time))
AS estimate
FROM MqlRatesDB
WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-01'))
)
GROUP BY intraday, day
ORDER BY objective DESC

```

第一个 SQL 查询已变为嵌套查询，我们现在从这个嵌套查询中通过外部 SQL 查询累积数据。通过 **GROUP BY intraday, day**，我们可以按日内时间和星期几的所有组合进行分组，并对每个组合进行聚合统计。此外，我们还添加了按目标指标排序 (**ORDER BY objective DESC**)，以便最佳选项位于表的顶部。

在嵌套查询中，我们移除了 **LIMIT** 参数，因为分组后的数量已变得可接受，远少于分析的柱线数量。因此，对于 H1，我们得到 120 个选项 (24*5)。

扩展后的查询放置在 **DBQuotesIntradayLagGroup.sql** 文本文件中，该文件又作为资源连接到同名的测试脚本 **DBQuotesIntradayLagGroup.mq5**。其源代码与前一个几乎没有区别，因此我们将直接显示其针对默认日期范围 (从 2015 年初到 2021 年初，不包括 2021 年和 2022 年) 的启动结果。


```

db.isOpen()=true / ok
db.hasTable(Table)=true / ok
SELECT
AVG(product) / STDDEV(product) AS objective,
SUM(estimate) AS backtest_profit,
SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS backtest_PF,
intraday, day
FROM
(
SELECT
...
FROM MqlRatesDB
WHERE (time >= 1420070400 AND time < 1609459200)
)
GROUP BY intraday, day
ORDER BY objective DESC
#| objective backtest_profit backtest_PF intraday day
-----+-----
1| 0.16713214428916 0.073200000000001 1.46040631486258 16:00:00 5
2| 0.118128291843983 0.0433099999999995 1.33678071539657 20:00:00 3
3| 0.103701251751617 0.009299999999999853 1.14148790506616 5:00:00 2
4| 0.102930330078208 0.01643999999999973 1.1932071923845 8:00:00 4
5| 0.089531492651001 0.00643000000000006 1.10167615433271 7:00:00 2
6| 0.0827628326995007 -8.99999999970369e-05 0.999601152226913 17:00:00 4
7| 0.0823433025146974 0.01597000000000012 1.21665988332657 21:00:00 1
8| 0.0767938336191962 0.005229999999999874 1.04226945769012 13:00:00 1
9| 0.0657741522256548 0.01622999999999986 1.09699976093712 15:00:00 2
10| 0.0635243373432768 0.009320000000000044 1.08294766820933 22:00:00 3
...
110| -0.0814131025461459 -0.01891000000000015 0.820605255668329 21:00:00 5
111| -0.0899571263478305 -0.03219000000000028 0.721250432975386 22:00:00 4
112| -0.0909772560603298 -0.02261000000000016 0.851161872161138 19:00:00 4
113| -0.0961794181717023 -0.008469999999999931 0.936377976414036 12:00:00 5
114| -0.108868074018582 -0.02460999999999998 0.634920634920637 0:00:00 5
115| -0.109368419185336 -0.02507000000000013 0.744496534855268 8:00:00 2
116| -0.121893581607986 -0.02345999999999998 0.610945273631843 0:00:00 3
117| -0.135416609546408 -0.08988999999999971 0.343437294573087 0:00:00 1
118| -0.142128458003631 -0.02552000000000018 0.681835182645536 6:00:00 4
119| -0.142196924506816 -0.02057000000000004 0.629769618430515 0:00:00 2
120| -0.15200009633513 -0.03014999999999988 0.708864426419475 2:00:00 1

```

因此，从分析结果可以得知，周五的 16 点 H1 柱线是基于前一根柱线延续趋势的最佳候选。其次优选的是周三的 20 点柱线。以此类推。

然而，最好是在远期（样本外）时段检验找到的这些设置。

为此，我们可以不仅在“过去”的日期范围（在我们的测试中直到 2021 年）执行当前的 SQL 查询，而且在“未来”（从 2021 年初开始）再次执行。两个查询的结果应按我们的分组（*intraday*, *day*）进行连接（JOIN）。然后，在保持按目标指标排序的同时，我们将在相邻的列中看到相同时间和星期几组合的盈利和盈利因子，以及它们下降了多少。

以下是最终的 SQL 查询（缩写版）：

```

SELECT * FROM
(
SELECT
AVG(product) / STDDEV(product) AS objective,
SUM(estimate) AS backtest_profit,
SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS backtest_PF,
intraday, day
FROM
(
SELECT ...
FROM MqlRatesDB
WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-01'))
)
GROUP BY intraday, day
) backtest
JOIN
(
SELECT
SUM(estimate) AS forward_profit,
SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS forward_PF,
intraday, day
FROM
(
SELECT ...
FROM MqlRatesDB
WHERE (time >= STRFTIME('%s', '2021-01-01'))
)
GROUP BY intraday, day
) forward
USING(intraday, day)
ORDER BY objective DESC

```

请求的完整文本在 *DBQuotesIntradayBackAndForward.sql* 文件中提供。它在 *DBQuotesIntradayBackAndForward.mq5* 脚本中作为资源连接。

以默认设置运行脚本，我们可以得到以下指标（缩写）：

```

#| objective backtest_profit backtest_PF intraday day forward_profit forward_PF
--+-----
1| 0.16713214428916 0.073200000001 1.46040631486 16:00:00 5 0.004920000048 1.12852664
2| 0.118128291843983 0.0433099999995 1.33678071539 20:00:00 3 0.0078800000055 1.277856
3| 0.103701251751617 0.00929999999853 1.14148790506 5:00:00 2 0.0022100000082 1.121495
4| 0.102930330078208 0.01643999999973 1.1932071923 8:00:00 4 0.0014099999969 1.07253086
5| 0.089531492651001 0.00643000000006 1.10167615433 7:00:00 2 -0.009119999869 0.561749
6| 0.0827628326995007 -8.99999999970e-05 0.999601152226 17:00:00 4 0.009070000091 1.1
7| 0.0823433025146974 0.01597000000012 1.21665988332 21:00:00 1 0.00250999999 1.121314
8| 0.0767938336191962 0.00522999999874 1.04226945769 13:00:00 1 -0.0084900000055 0.753
9| 0.0657741522256548 0.0162299999986 1.09699976093 15:00:00 2 0.01423999997 1.349791
10| 0.0635243373432768 0.00932000000044 1.08294766820 22:00:00 3 -0.00456999993 0.828
...

```

因此，具有最佳交易时间安排的交易系统在“未来”时期继续显示盈利，尽管不如回测时那么大。

当然，所考虑的示例仅仅是交易系统的一个特例。例如，我们可以找到在一周中的某个时间和日期组合，此时反转策略在相邻柱线上有效，或者完全基于其他原则（分析分时报价、日历、交易信号组合等）。

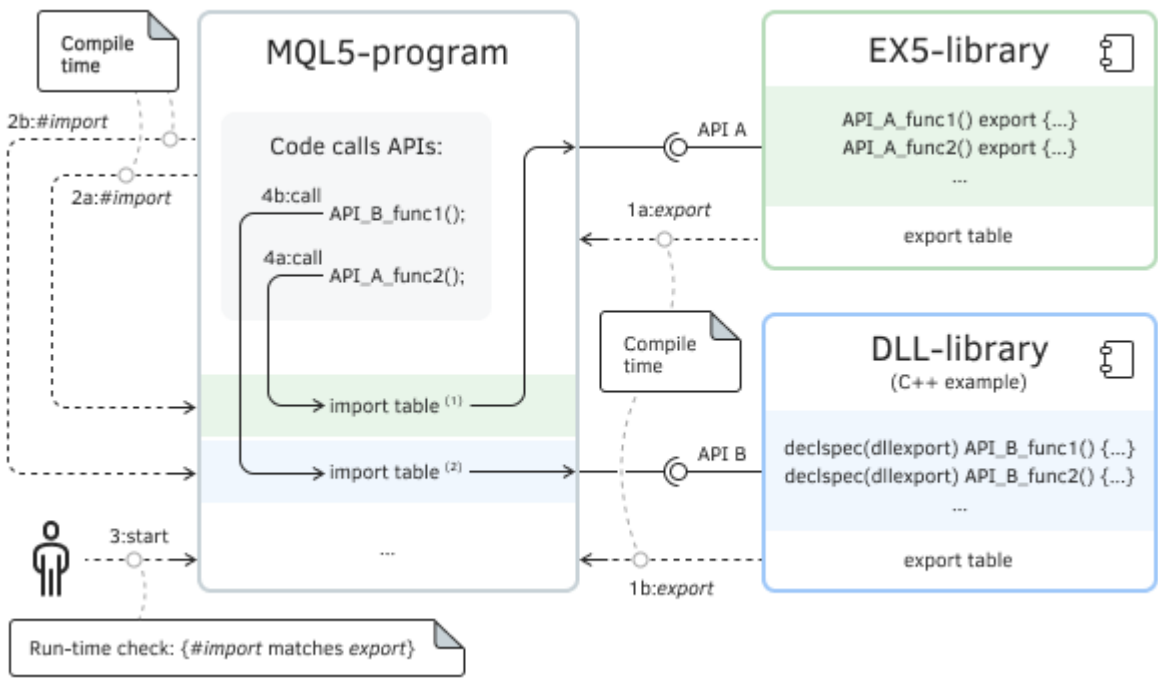
最重要的是，SQLite 引擎提供了许多方便的工具，否则这些工具在 MQL5 中需要由我们自行实现。说实话，学习 SQL 需要时间。该平台允许你选择两种技术的最佳组合以进行高效编程。

7.7 二进制格式库的开发和连接

除了特殊类型的 MQL 程序 -EA 交易、指标、脚本和服务之外 -MetaTrader5 平台允许你创建和连接具有任意功能的独立二进制模块，这些模块被编译为符合 Windows 标准的 ex5 文件或常用的 dll (动态链接库)。它们可以是分析算法、图形可视化、与 web 服务的网络交互、外部程序的控制或操作系统本身。在任何情况下，这类库在终端中不是作为独立的 MQL 程序工作，而是与上述 4 种类型中的任何一种协同工作。

集成函数库和主 (父) 程序的思路是，库导出某些函数，即声明它们可供外部使用，然后程序导入它们的原型。正是原型的描述 - 名称集、参数列表和返回值 - 这样您就可以在代码中调用这些函数，而不需要实现它们。

然后，在 MQL 程序启动期间，执行早期动态链接。这意味着在主程序之后加载库，并在库中导入的原型和导出函数之间建立对应关系。通过名称、参数列表和返回类型建立一对一的对应关系是成功加载的先决条件。如果没有为至少一个函数的导入说明找到相应的导出实现，MQL 程序的执行将被取消 (程序在启动阶段报错并结束)。



MQL 程序与库通信组件图

启动 MQL 程序时，不能选择包含的库。此链接是由开发人员在库导入时编译主程序期间设置的。但是，用户可以在程序启动之前用另一个 ex5/dll 文件手动替换原先的文件（假设实现的导出函数的原型在库中匹配）。例如，如果库包含带标签的字符串资源，这可以用于切换用户界面语言。然而，库通常作为一种商业产品提供，因此作者不准备以开放头文件的形式发布。

对那些从其他环境转到 MQL5 并且已经熟悉 DLL 技术的程序员，我们想补充一则关于后期动态链接的说明，也是 DLL 的一个优点。在执行过程中，一个 MQL 程序 (或 DLL 模块) 不可能完全动态链接到另一个 MQL 程序。唯一类似的操作是 MQL5 允许你在此期间通过 *iCustom* 或者 *IndicatorCreate* 链接 EA 交易和指标，其中指标充当动态链接库 (但是，与之进行的编程交互必须通过指标 API 来完成，这意味着与直接函数调用 (通过 *export/import*) 相比，*CopyBuffer* 的开销会增加)。

注意，在正常情况下，当从源代码编译 MQL 程序而不导入外部函数时，使用静态链接，即生成的二进制代码直接引用被调用的函数，因为这些函数在编译时是已知的。

严格地说，一个库也可以依赖其他库，即它可以导入一些函数。从理论上讲，这种依赖关系链甚至可以更长：例如，MQL 程序包含库 A，库 A 使用库 B，而库 B 又使用库 C。但是，不建议使用这种依赖链，因为这会导致产品的分发和安装变得复杂，并且使潜在启动问题原因的确认变得更加困难。因此，库通常直接连接到父 MQL 程序。

在本章中，我们将描述在 MQL5 中创建库、导出和导入函数 (包括对函数中使用的数据类型的限制) 以及连接外部 (现成) DLL 的过程。本书未涵盖 DLL 开发方面的内容。

7.7.1 创建 ex5 库；函数 *export*

要说明一个库，请将 *#property library* 指令添加到主模块 (已编译) 的源代码中 (通常在文件的开头)。

```
#property library
```

在编译过程中，通过 *#include* 在任何其他文件中指定该指令将无效。

library 特性通知编译器给定的 ex5 文件是一个库：在 ex5 文件的头文件中存储了一个标记。

在 MetaTrader 5 中为库保留了一个单独的文件夹 *MQL5/Libraries*。可以在其中组织嵌套文件夹的层级，就像 MQL5 中对其他类型程序所做的操作一样。

库不直接参与事件处理，因此编译器不要求代码中存在任何标准处理程序。但是，您可以从该库所连接的 MQL 程序的事件处理程序中调用该库的导出函数。

要从库中导出一个函数，只需用一个特殊的 *export* 关键字。该修饰符必须放在函数头文件的最末端。

```
result_type function_id ( [ parameter_type parameter_id
                          [ = default_value] ... ] ) export
{
    ...
}
```

参数必须是简单类型或字符串、具有这种类型的字段的结构或它们的数组。MQL5 对象类型允许指针和引用 (有关导入 DLL 的限制，参见 [相关章节](#))。

让我们来分析一些例子。参数是一个质数：

```
double Algebraic2(const double x) export
{
    return x / sqrt(1 + x * x);
}
```

这些参数是指向一个对象的指针和指向一个指针的引用 (允许您在函数内部分配一个指针)。

```

class X
{
public:
    X() { Print(__FUNCSIG__); }
};
void setObject(const X *obj) export { ... }
void getObject(X *&obj) export { obj = new X(); }

```

参数是一个结构体：

```

struct Data
{
    int value;
    double data[];
    Data(): value(0) { }
    Data(const int i): value(i) { ArrayResize(data, i); }
};

void getRefStruct(const int i, Data &data) export { ... }

```

只能导出函数，而不能导出整个类或结构。在指针和引用的帮助下，可以避免这些限制，我们将在后面详细讨论有关内容。

不能使用 **export** 关键字和 **#import** 指令声明函数模板。

export 修饰符指示编译器将该函数包含在给定的 ex5 可执行文件的导出函数表中。借助这个功能，这类函数可以从其他 MQL 程序中获得（“可见”），使用特殊指令 **#import** 导入之后，它们就可以使用了。

所有要导出的函数都必须用 **export** 修饰符标记。主程序不需要导入所有这些函数，因为它只能导入必要的函数。

如果您忘记导出一个函数，但将它包含在主 MQL 程序的导入指令中，那么当后者启动时，将会出现一个错误：

```

cannot find 'function' in 'library.ex5'
unresolved import function call

```

如果导出函数和它的导入原型的描述存在差异，也会出现类似问题。例如，如果您在更改编程接口（通常在单独的头文件中说明）后忘记重新编译库或主程序，就会发生这种情况。

无法对库进行调试，所以如果有必要，应该有一个帮助脚本或另一个 MQL 程序，它是在调试器模式下从库的源代码构建的，并且可以通过断点或分步执行。当然，这将需要使用一些真实或人工数据来模拟对导出函数的调用。

对于 DLL，导出函数以不同方式描述，具体取决于创建它们的编程语言。详细情况，请查询您选择的开发环境的文档。

我们以一个简单库 **MQL5/Libraries/MQL5Book/LibRand.mq5** 为例进行分析，从这个库中导出了几个函数，它们具有不同类型的参数和结果。该库设计用于生成随机数据：

- 具有伪正态分布的数值数据
- 具有来自给定集的随机字符的字符串（可能适用于密码）

特别是，可以使用 **PseudoNormalValue** 函数获得一个随机数，其中期望值和方差被设置为参数。

```
double PseudoNormalValue(const double mean = 0.0, const double sigma = 1.0,
    const bool rooted = false) export
{
    // use ready-made sqrt for mass generation in a cycle in PseudoNormalArray
    const double s = !rooted ? sqrt(sigma) : sigma;
    const double r = (rand() - 16383.5) / 16384.0; // [-1,+1] excluding borders
    const double x = -(log(1 / ((r + 1) / 2) - 1) * s) / M_PI * M_E + mean;
    return x;
}
```

PseudoNormalArray 函数用给定数量 (*n*) 的随机值填充数组, 并具有所需的分布。

```
bool PseudoNormalArray(double &array[], const int n,
    const double mean = 0.0, const double sigma = 1.0) export
{
    bool success = true;
    const double s = sqrt(fabs(sigma)); // passing ready sqrt when calling PseudoNormalArray
    ArrayResize(array, n);
    for(int i = 0; i < n; ++i)
    {
        array[i] = PseudoNormalValue(mean, s, true);
        success = success && MathIsValidNumber(array[i]);
    }
    return success;
}
```

为了生成一个随机字符串, 我们编写了 *RandomString* 函数, 它从提供的字符集 (*pattern*) 中选择给定数量 (*length*) 的任意字符。当 *pattern* 参数为空 (默认) 时, 假定为一组完整的字母和数字。辅助函数 *StringPatternAlpha* 和 *StringPatternDigit* 用于获取这些字母和数字; 这些函数也是可导出函数 (本书未列出, 请参见源代码)。

```
string RandomString(const int length, string pattern = NULL) export
{
    if(StringLen(pattern) == 0)
    {
        pattern = StringPatternAlpha() + StringPatternDigit();
    }
    const int size = StringLen(pattern);
    string result = "";
    for(int i = 0; i < length; ++i)
    {
        result += ShortToString(pattern[rand() % size]);
    }
    return result;
}
```

一般来说, 要使用一个库, 必须发布一个头文件, 说明可以从外部获得的所有内容 (内部实现的细节可以而且应该被隐藏)。在我们的例子中, 该文件被称为 *MQL5Book/LibRand.mqh*。特别是, 它说明了用户定义的类型 (在我们的例子中是 *STRING_PATTERN* 枚举) 和函数原型。

虽然我们还不知道 *#import* 块的确切语法, 但这不应该影响它内部声明的清晰性: 这里重复了导出函数的头文件, 但没有使用 *export* 关键字。

```

enum STRING_PATTERN
{
    STRING_PATTERN_LOWERCASE = 1, // lowercase letters only
    STRING_PATTERN_UPPERCASE = 2, // capital letters only
    STRING_PATTERN_MIXEDCASE = 3  // both registers
};

#import "MQL5Book/LibRand.ex5"
string StringPatternAlpha(const STRING_PATTERN _case = STRING_PATTERN_MIXEDCASE);
string StringPatternDigit();
string RandomString(const int length, string pattern = NULL);
void RandomStrings(string &array[], const int n, const int minlength,
    const int maxlength, string pattern = NULL);
void PseudoNormalDefaultMean(const double mean = 0.0);
void PseudoNormalDefaultSigma(const double sigma = 1.0);
double PseudoNormalDefaultValue();
double PseudoNormalValue(const double mean = 0.0, const double sigma = 1.0,
    const bool rooted = false);
bool PseudoNormalArray(double &array[], const int n,
    const double mean = 0.0, const double sigma = 1.0);
#import

```

探讨了 `#import` 指令之后，我们将在下一节编写一个使用该库的测试脚本。

7.7.2 包括库和函数的 `#import`

函数从已编译的 MQL5 模块（`.ex5` 文件）和 Windows 动态库模块（`.dll` 文件）中导入。模块名称在 `#import` 指令中指定，后面是导入函数原型说明。该代码块必须以另一个 `#import` 指令结束，此外，它可以没有名称，只需关闭代码块本身，或者可以在指令中指定另一个库的名称，同时下一个导入块启动。一串导入块应该始终以不带库名的指令结尾。

该指令最简单的形式如下：

```

#import "[path] module_name [.extension]"
    function_type function_name([parameter_list]);
    [function_type function_name([parameter_list]);]
    ...
#import

```

可以指定不带扩展名的库文件的名称：默认情况下采用 DLL。扩展名 `ex5` 是必须的。

该名称前面可以是库地址路径。默认情况下，如果没有路径，将在 `MQL5/Libraries` 文件夹或库所连接的 MQL 程序旁的文件夹中搜索库。否则，根据类型是 DLL 还是 EX5，可以应用不同的规则来搜索库。这些规则在 [单独章节](#)中介绍。

以下是来自两个库的顺序导入块的例子：

```
#import "user32.dll"
    int      MessageBoxW(int hWnd, string szText, string szCaption, int nType);
    int      SendMessageW(int hWnd, int Msg, int wParam, int lParam);
#import "lib.ex5"
    double   round(double value);
#import
```

使用这样的指令，可以从源代码中调用导入的函数，就像在 MQL 程序中直接定义函数一样。所有关于加载库和重定向调用到第三方模块的技术事项都由 MQL 程序执行环境处理。

为了让编译器正确地发出对导入函数的调用并组织参数的传递，需要一个完整的说明：包括结果类型、所有参数、修饰符和默认值（如果它们出现在源代码中的话）。

由于导入的函数在已编译模块的外部，因此编译器无法检查传递的参数和返回值是否正确。预期数据和接收数据的格式之间的任何差异都将导致程序执行过程出错，这可能表现为程序的紧急停止或非预期行为。

如果无法加载库或者无法找到调用的导入函数，MQL 程序将终止，并在日志中显示相应的消息。在问题解决之前，程序将无法运行，例如通过以下方式来解决：修改和重新编译，将所需的库放在搜索路径上的某个地址，或者允许使用 DLL（仅适用于 DLL）。

当共享多个库时（不管是 DLL 还是 EX5），记住它们必须有不同的名称，无论它们的地址目录是什么。所有导入的函数都有一个与库文件名称匹配的作用域，即它是一种 **命名空间**，会为每个包含的库隐式分配该命名空间。

导入的函数可以有任何名称，包括与内置函数名称匹配的名称（尽管不建议这样做）。此外，可以从不同的模块中同时导入同名的函数。在这种情况下，应使用 **上下文权限** 来确定应调用哪个函数。

例如：


```

import "kernel32.dll"
    int GetLastError();
import "lib.ex5"
    int GetLastError();
import

class Foo
{
public:
    int GetLastError() { return(12345); }
    void func()
    {
        Print(GetLastError());           // call a class method
        Print(::GetLastError());         // calling the built-in (global) MQL5 function
        Print(kernel32::GetLastError()); // function call from kernel32.d
        Print(lib::GetLastError());      // function call from lib.ex5
    }
};

void OnStart()
{
    Foo foo;
    foo.func();
}

```

让我们来分析一个简单的 *LibRandTest.mqh* 脚本的例子，它使用了上一节中创建的 EX5 库中的函数。

```
#include <MQL5Book/LibRand.mqh>
```

在输入参数中，您可以选择数字数组中的元素数量、分布参数以及直方图的步长，我们将计算直方图以确保分布近似符合正态分布。

```

input int N = 10000;
input double Mean = 0.0;
input double Sigma = 1.0;
input double HistogramStep = 0.5;
input int RandomSeed = 0;

```

MQL 5 中内置的随机数生成器的初始化由 *RandomSeed* 的值执行，或者，此处为 0，*GetTickCount* 被选取（每次启动时都是新的）。

为了构建直方图，我们使用 *MapArray* 和 *QuickSortStructT*（我们已经在 *多货币指标* 以及有关 *数组排序* 的章节中使用过它们）。该映射将累加直方图单元中命中随机数的计数器，步长为 *HistogramStep*。

```

#include <MQL5Book/MapArray.mqh>
#include <MQL5Book/QuickSortStructT.mqh>

```

要显示基于地图的直方图，你需要能够以键值顺序对映射进行排序。为此，我们必须定义一个派生类。

```

#define COMMA ,

template<typename K,typename V>
class MyMapArray: public MapArray<K,V>
{
public:
    void sort()
    {
        SORT_STRUCT(Pair<K COMMA V>, array, key);
    }
};

```

请注意, COMMA 宏成为逗号字符, 的替代表示形式, 并在调用另一个 SORT_STRUCT 宏时使用。如果不进行这种替换, 预处理器会将 <K,V> 对中的逗号解释为普通的宏参数分隔符, 其结果是 SORT_STRUCT 会收到 4 个参数输入, 而不是预期的 3 个, 这将导致编译错误。预处理器对 MQL5 语法一无所知。

在 **OnStart** 开头, 在初始化生成器之后, 我们检查单个随机字符串和不同长度的字符串数组的接收。

```

void OnStart()
{
    const uint seed = RandomSeed ? RandomSeed : GetTickCount();
    Print("Random seed: ", seed);
    MathSrand(seed);

    // call two library functions: StringPatternDigit and RandomString
    Print("Random HEX-string: ", RandomString(30, StringPatternDigit() + "ABCDEF"));
    Print("Random strings:");
    string text[];
    RandomStrings(text, 5, 10, 20);           // 5 lines from 10 to 20 characters long
    ArrayPrint(text);
    ...
}

```

接下来, 我们测试正态分布的随机数。

```

// call another library function: PseudoNormalArray
double x[];
PseudoNormalArray(x, N, Mean, Sigma); // filled array x

Print("Random pseudo-gaussian histogram: ");

// take 'long' as key type, because 'int' has already been used for index access
MyMapArray<long,int> map;

for(int i = 0; i < N; ++i)
{
// value x[i] determines the cell of the histogram, where we increase the statistics
  map.inc((long)MathRound(x[i] / HistogramStep));
}
map.sort(); // sort by key (i.e. by value)

int max = 0; // searching for maximum for normalization
for(int i = 0; i < map.getSize(); ++i)
{
  max = fmax(max, map.getValue(i));
}

const double scale = fmax(max / 80, 1); // the histogram has a maximum of 80 symbo

for(int i = 0; i < map.getSize(); ++i) // print the histogram
{
  const int p = (int)MathRound(map.getValue(i) / scale);
  string filler;
  StringInit(filler, p, '*');
  Print(StringFormat("%.2f (%4d)",
    map.getKey(i) * HistogramStep, map.getValue(i)), " ", filler);
}

```

以下是使用默认设置运行的结果（计时器随机化 - 每次运行将选择一个 **seed**）。

随机种子:8859858

随机十六进制字符串:E58B125BCCDA67ABAB2F1C6D6EC677

随机字符串:

"K4Z0pdIy5yxq4ble2" "NxTrVRl6q5j3Hr2FY" "6qxRdDzjp3WNA8xV" "UlOPYinnGd36" "60Cmde6rvE

随机伪高斯直方图:

```
-9.50 ( 2)
-8.50 ( 1)
-8.00 ( 1)
-7.00 ( 1)
-6.50 ( 5)
-6.00 ( 10) *
-5.50 ( 10) *
-5.00 ( 24) *
-4.50 ( 28) **
-4.00 ( 50) ***
-3.50 ( 100) *****
-3.00 ( 195) *****
-2.50 ( 272) *****
-2.00 ( 510) *****
-1.50 ( 751) *****
-1.00 (1029) *****
-0.50 (1288) *****
+0.00 (1457) *****
+0.50 (1263) *****
+1.00 (1060) *****
+1.50 ( 772) *****
+2.00 ( 480) *****
+2.50 ( 280) *****
+3.00 ( 172) *****
+3.50 ( 112) *****
+4.00 ( 52) ***
+4.50 ( 43) **
+5.00 ( 10) *
+5.50 ( 8)
+6.00 ( 8)
+6.50 ( 2)
+7.00 ( 3)
+7.50 ( 1)
```

在此库中，只有内置类型的导入和导出的函数。然而，从实用的角度来看，具有结构、类和模板的对象接口更有趣，也更受欢迎。我们将在 [单独的章节](#) 中讨论在库中使用它们的细微差别。

在测试程序中测试 EA 交易和指标时，应该记住与库相关的重要一点。主测试 MQL 程序所需的库是由 `#import` 指令自动确定的。但是，如果从主程序调用自定义指标，而某个库连接到该主程序，则有必要在程序特性中明确指出它间接依赖于某个特定的库。这是通过以下指令完成的：

```
#property tester_library"path_library_name.extension"
```

7.7.3 库文件搜索顺序

如果指定的库名称不含路径或相对路径，则根据库类型的不同，搜索将根据不同的规则执行。

系统库 (DLL) 是根据操作系统的规则加载的。如果库已经被加载 (例如, 通过另一个 EA 交易, 或者甚至从并行启动的另一个客户端终端), 那么调用将转到已经加载的库。否则, 搜索将按以下顺序进行:

1. 从中导入 DLL 的已编译 EX5 程序的文件夹已启动。
2. **MQ5/Libraries** 文件夹。
3. 正在运行的 MetaTrader 5 终端所在的文件夹。
4. 系统文件夹 (通常在 Windows 中)。
5. Windows 目录。
6. 终端进程的当前工作文件夹 (可能与终端的位置文件夹不同)。
7. PATH 系统变量中列出的文件夹。

在 **#import** 指令中, 不建议使用格式为 **Drive:/Directory/FileName.dll** 的完全限定可加载模块名。

如果 DLL 在工作中使用了另一个 DLL, 那么在没有第二个 DLL 的情况下, 第一个 DLL 将无法加载。

对导入的 EX5 库的搜索按以下顺序执行:

1. 用于启动导入 EX5 程序的文件夹。
2. 特定终端实例的 **MQ5/Libraries** 文件夹。
3. 所有 MetaTrader 5 终端 (**Common/MQ5/Libraries**) 的公共文件夹中的 **MQ5/Libraries** 文件夹。

在加载一个 MQL 程序之前, 会构建一个所有 EX5 库模块的总列表, 其中支持的模块既可以从程序本身使用, 也可以从该列表中的库使用。它被称为依赖项列表, 可以成为一个分支很多的“树”。

对于 EX5 库, 终端还提供一次性下载的可重用模块。

无论是什么类型的库, 它的每个实例都使用与调用 EA 交易、脚本、服务或指标的上下文相关的本身的数据。库不是共享访问 MQL5 变量或数组的工具。

EX5 库和 DLL 运行在调用模块的线程上。

并没有通用方法在库代码中找出它加载的位置。

7.7.4 DLL 连接细节

下列实体不能作为参数传递给从 DLL 导入的函数:

- 类 (对象和指向它们的指针)
- 包含动态数组、字符串、类和其他复杂结构的结构
- 字符串数组或上述复杂对象

所有简单类型参数都是通过值传递的, 除非明确声明它们是通过引用传递的。传递一个字符串时, 传递被复制字符串的缓冲区地址; 如果字符串是通过引用传递的, 那么这个特定字符串的缓冲区地址将被传递给从 DLL 导入的函数, 而不进行复制。

将数组传递给 DLL 时, 始终传递数据缓冲区开头的地址 (不考虑 **AS_SERIES** 标志)。DLL 内部的函数对 **AS_SERIES** 标志一无所知, 传递的数组是一个长度未知的数组, 因此需要一个附加参数来指定它的大小。

在描述导入函数的原型时，可以使用带有默认值的参数。

导入 DLL 时，应在特定 MQL 程序的特性中或在终端的常规设置中授予使用它们的权限。对此，在 [权限](#) 一节中，我们演示了 *EnvPermissions.mqh*，它具有一个使用系统 DLL 读取 Windows 系统剪贴板内容的函数。这个函数是可选的：它的调用被注释掉了，因为当时我们不知道如何使用库。现在，我们将它转移到一个单独的脚本 *LibClipboard.mqh*。

运行该脚本可能会提示用户进行确认（出于安全原因，默认情况下 DLL 是禁用的）。如有必要，可在对话框中的“依赖项”选项卡上启用该选项。

头文件 *MQL5/Include/WinApi* 目录中提供，该目录还包括 *#import* 指令，用于非常需要的系统功能，如剪贴板管理（*OpenClipboard*、*GetClipboardData* 和 *CloseClipboard*）、内存管理（*GlobalLock* 和 *GlobalUnlock*）、Windows 窗口以及许多其他函数。我们将只包括两个文件：*winuser.mqh* 和 *winbase.mqh*。它们包含必需的导入指令，并通过连接到 *windef.mqh*，间接包含 Windows 术语宏（HANDLE 和 PVOID）：

```
#define HANDLE long
#define PVOID long

#import "user32.dll"
...
int OpenClipboard(HANDLE wnd_new_owner);
HANDLE GetClipboardData(uint format);
int CloseClipboard(void);
...
#import

#import "kernel32.dll"
...
PVOID GlobalLock(HANDLE mem);
int GlobalUnlock(HANDLE mem);
...
#import
```

此外，我们还从 *kernel32.dll* 库导入 *lstrcatW* 函数，因为我们不满意 *winbase.mqh* 默认提供的说明：这给了函数第二个原型，适用于在第一个参数中传递 PVOID 值。

```
#include <WinApi/winuser.mqh>
#include <WinApi/winbase.mqh>

#define CF_UNICODETEXT 13 // one of the standard exchange formats - Unicode text
#import "kernel32.dll"
string lstrcatW(PVOID string1, const string string2);
#import
```

使用剪贴板的本质是使用 *OpenClipboard* 捕获对剪贴板的访问权限，之后你应获得一个数据句柄（*GetClipboardData*），将其转换为内存地址（*GlobalLock*），最后将数据从系统内存复制到你的变量（*lstrcatW*）。然后，以相反的顺序释放被占用的资源（*GlobalUnlock* 和 *CloseClipboard*）。

```

void ReadClipboard()
{
    if(OpenClipboard(NULL))
    {
        HANDLE h = GetClipboardData(CF_UNICODETEXT);
        PVOID p = GlobalLock(h);
        if(p != 0)
        {
            const string text = lstrcatW(p, "");
            Print("Clipboard: ", text);
            GlobalUnlock(h);
        }
        CloseClipboard();
    }
}

```

尝试将文本复制到剪贴板，然后运行脚本：剪贴板的内容应该被记录。如果缓冲区包含无文本表示的图像或其他数据，结果将为空。

从 DLL 导入的函数须遵循 Windows API 函数的二进制可执行链接约定。为了确保这种约定，编译器特定的关键字被用在程序的源文本中，例如 C 或 C++ 中的 `__stdcall`。这些链接规则意味着：

- 调用函数（在我们的例子中，是 MQL 程序）必须看到被调用（从 DLL 导入）函数的原型，以便正确地将参数堆叠到栈上。
- 调用函数（在我们的例子中，是 MQL 程序）以相反的顺序从右到左堆叠参数 - 导入函数就是按这个顺序读取传递给它的参数。
- 参数通过值传递，除了那些通过引用显式传递的参数（在我们的例子中是字符串）。
- 导入的函数读取传递给它的参数并清除堆栈。

下面是另一个使用 DLL 的脚本示例 `LibWindowTree.mq5`。它的任务是遍历所有终端窗口的树，并获得它们的类名（根据使用 WinAPI 在系统中的注册）和标题。这里的窗口指的是 Windows 界面的标准元素，也包括控件。这个过程对于自动化使用终端非常有用：模拟窗口中的按钮按压，通过 MQL5 切换不可用的模式，等等。

为了导入所需的系统函数，我们包含使用 `user32.dll` 的头文件 `WinUser.mqh`。

```
#include <WinAPI/WinUser.mqh>
```

你可以使用 `GetClassNameW` 和 `GetWindowTextW` 函数获得窗口类的名称，在 `GetWindowData` 函数中调用这两个函数。

```

void GetWindowData(HANDLE w, string &clazz, string &title)
{
    static ushort receiver[MAX_PATH];
    if(GetWindowTextW(w, receiver, MAX_PATH))
    {
        title = ShortArrayToString(receiver);
    }
    if(GetClassNameW(w, receiver, MAX_PATH))
    {
        clazz = ShortArrayToString(receiver);
    }
}

```

函数名中的 W 后缀意味着它们适用于 Unicode 格式的字符串 (每个字符 2 个字节), 这是当今最常用的格式 (ANSI 字符串中的 A 后缀仅在向后兼容旧库时才有意义)。

给定一个 Windows 窗口的初始句柄, 遍历其父窗口的层级由 *TraverseUp* 函数提供: 它的操作基于 *GetParent* 系统函数。对于每个找到的窗口, *TraverseUp* 调用 *GetWindowData* 并将结果类名和标题输出到日志中。

```

HANDLE TraverseUp(HANDLE w)
{
    HANDLE p = 0;
    while(w != 0)
    {
        p = w;
        string clazz, title;
        GetWindowData(w, clazz, title);
        Print("", clazz, " ", title, "");
        w = GetParent(w);
    }
    return p;
}

```

深入到层级中的遍历是由 *TraverseDown* 函数执行的: *FindWindowExW* 系统函数用于枚举子窗口。


```

HANDLE TraverseDown(const HANDLE w, const int level = 0)
{
    // request first child window (if any)
    HANDLE child = FindWindowExW(w, NULL, NULL, NULL);
    while(child)           // oop while there are child windows
    {
        string clazz, title;
        GetWindowData(child, clazz, title);
        Print(StringFormat("%*s", level * 2, ""), "'", clazz, "' '", title, "'");
        TraverseDown(child, level + 1);
        // requesting next child window
        child = FindWindowExW(w, child, NULL, NULL);
    }
    return child;
}

```

在 *OnStart* 函数中，我们通过从运行脚本的当前图表的句柄向上遍历窗口来找到主终端窗口。然后我们构建整个终端窗口树。

```

void OnStart()
{
    HANDLE h = TraverseUp(ChartGetInteger(0, CHART_WINDOW_HANDLE));
    Print("Main window handle: ", h);
    TraverseDown(h, 1);
}

```

我们还可以通过类名和/或标题搜索所需的窗口，因此可以通过调用 *FindWindowW* 立即获得主窗口，因为它的属性已知。

```
h = FindWindowW("MetaQuotes::MetaTrader::5.00", NULL);
```

以下是日志示例 (片段):

```

'AfxFrameOrView140su' ''
'Afx:0000000013F110000:b:00000000000010003:0000000000000006:000000000000306BA' 'EURUSD,H
'MDIClient' ''
'MetaQuotes::MetaTrader::5.00' '12345678 - MetaQuotes-Demo: Demo Account - Hedge - ..
Main window handle: 263576
'mscctl_statusbar32' 'For Help, press F1'
'AfxControlBar140su' 'Standard'
'ToolbarWindow32' 'Timeframes'
'ToolbarWindow32' 'Line Studies'
'ToolbarWindow32' 'Standard'
'AfxControlBar140su' 'Toolbox'
'Afx:0000000013F110000:b:00000000000010003:0000000000000006:0000000000000000' 'Toolbox'
'AfxWnd140su' ''
'ToolbarWindow32' ''
...
'MDIClient' ''
'Afx:0000000013F110000:b:00000000000010003:0000000000000006:000000000000306BA' 'EURUSD,H
'AfxFrameOrView140su' ''
'Edit' '0.00'
'Afx:0000000013F110000:b:00000000000010003:0000000000000006:000000000000306BA' 'XAUUSD,D
'AfxFrameOrView140su' ''
'Edit' '0.00'
'Afx:0000000013F110000:b:00000000000010003:0000000000000006:000000000000306BA' 'EURUSD,M
'AfxFrameOrView140su' ''
'Edit' '0.00'

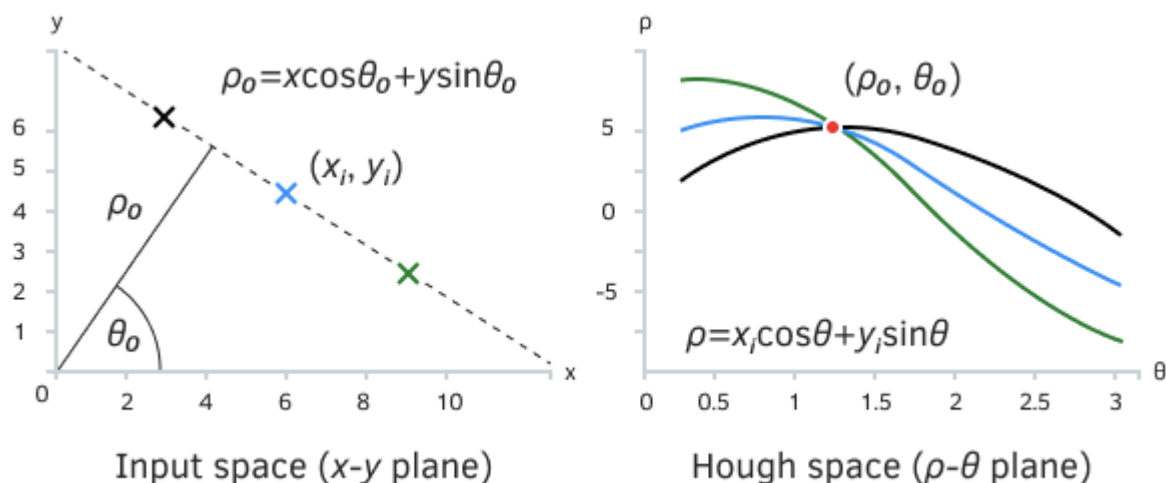
```

7.7.5 MQL5 库中的类和模板

尽管通常禁止对类和模板进行导出和导入，但开发人员可以通过将抽象基接口的说明移动到库头文件中并传递指针来绕过这些限制。我们用一个库的例子（执行图像霍夫变换）来说明这个概念。

霍夫变换是一种提取图像特征的算法，该算法通过将图像与由一组参数说明的某种形式模型（公式）进行比较。

最简单的霍夫变换是通过将图像上的直线转换成极坐标来选择直线。通过这种处理，排列成一行的“填充”像素序列在极坐标空间中，在直线倾斜的特定角度（“theta”）和其相对于坐标中心的偏移（“ro”）的交点处形成峰值。



直线的霍夫变换

左边 (原始) 图像上的三个彩色点都在极坐标空间 (右边) 中留下了轨迹, 因为可以从不同角度以及垂直于中心的角度通过一个点绘制无限多条直线。除了红色标记外, 每个轨迹片段只被“标记”一次: 在这一点上, 所有三个轨迹相交并给出最大响应 (3)。事实上, 正如我们在原始图像中看到的, 一条直线穿过所有三个点。因此, 线的两个参数由极坐标中的最大值给出。

我们可以在价格图上使用霍夫变换来突出替代的支撑线和阻力线。如果在单个极值处画出这些线, 并且执行异常值的分析, 那么霍夫变换线可以分析所有 **High** 或者 **Low** 价格, 甚至是柱线内分时报价的分布。根据这些你可以得到一个更合理的水平估计。

我们从 `LibHoughTransform.mqh` 头文件开始探讨。由于一些抽象图分析提供了初始数据, 那么让我们来定义 `HoughImage` 接口模板。

```
template<typename T>
interface HoughImage
{
    virtual int getWidth() const;
    virtual int getHeight() const;
    virtual T get(int x, int y) const;
};
```

在处理图像时, 您需要知道图像的尺寸和每个像素的内容, 基于通用性, 用参数类型 `T` 表示。很明显, 在最简单的情况下, 它可以是 `int` 或者 `double` 类型。

调用分析图像处理稍微复杂一点。在库中, 我们需要描述类, 将它的对象从一个特殊的工厂函数中返回 (以指针的形式)。应该从库中导出的就是这个函数。假设如下:

```

template<typename T>
class HoughTransformDraft
{
public:
    virtual int transform(const HoughImage<T> &image, double &result[],
        const int elements = 8) = 0;
};

HoughTransformDraft<?> *createHoughTransform() export { ... } // Problem - template!

```

但是，不能导出模板类型和模板函数。因此，我们将制作一个中间的非模板类 *HoughTransform*，在这个类中，我们将为图像参数添加一个模板方法。遗憾的是，模板方法不能是虚的，因此我们将在方法内部手动调用（使用 *dynamic_cast*），用虚方法将处理重定向到派生类。

```

class HoughTransform
{
public:
    template<typename T>
    int transform(const HoughImage<T> &image, double &result[],
        const int elements = 8)
    {
        HoughTransformConcrete<T> *ptr = dynamic_cast<HoughTransformConcrete<T> *>(&this);
        if(ptr) return ptr.extract(image, result, elements);
        return 0;
    }
};

template<typename T>
class HoughTransformConcrete: public HoughTransform
{
public:
    virtual int extract(const HoughImage<T> &image, double &result[],
        const int elements = 8) = 0;
};

```

HoughTransformConcrete

类的内部实现将被写入

MQL5/Libraries/MQL5Book/LibHoughTransform.mq5 库文件中。

```

#property library

#include <MQL5Book/LibHoughTransform.mqh>

template<typename T>
class LinearHoughTransform: public HoughTransformConcrete<T>
{
protected:
    int size;

public:
    LinearHoughTransform(const int quants): size(quants) { }
    ...

```

由于我们将重新计算图像点到空间中新的极坐标中，因此应该为该任务分配一定的大小。这里我们讨论的是离散霍夫变换，因为我们认为原始图像是一组离散的点（像素），我们将在单元（量子）中累加垂直的角度值。为简单起见，我们将关注正方形空间的变体，其中角度和到中心的距离的读数数量相等。此参数被传递给类构造函数。

```

template<typename T>
class LinearHoughTransform: public HoughTransformConcrete<T>
{
protected:
    int size;
    Plain2DArray<T> data;
    Plain2DArray<double> trigonometric;

    void init()
    {
        data.allocate(size, size);
        trigonometric.allocate(2, size);
        double t, d = M_PI / size;
        int i;
        for(i = 0, t = 0; i < size; i++, t += d)
        {
            trigonometric.set(0, i, MathCos(t));
            trigonometric.set(1, i, MathSin(t));
        }
    }

public:
    LinearHoughTransform(const int quants): size(quants)
    {
        init();
    }
    ...

```

为了计算“填充的”像素在变换后的尺寸空间（大小 *size* *size*）中留下的“足迹”统计，我们描述了数据数组 *data*。辅助模板类 *Plain2DArray*（使用类型参数 *T*）允许模拟任意大小的二维数组。使用 *double* 类型参数的相同类适用于 *trigonometric* 表（包含预计算的角度正余弦值）。我们需要用该表来快速映射像素到一个新的空间。

检测最突出直线参数的方法称为 **extract**。它将图像作为输入，并且必须用找到的直线参数对填充输出 **result** 数组。在下面的等式中：

$$y = a * x + b$$

a 参数 (斜率, “theta”) 将被写入 **result** 数组的偶数位, 同时 **b** 参数 (偏移, “fo”) 将被写入数组的奇数位。例如, 该方法完成后第一条最明显的直线由以下表达式描述:

$$y = \text{result}[0] * x + \text{result}[1];$$

对于第二条直线, 索引将分别增加到 2 和 3, 依此类推, 直至所要求的最大线数 (**lines**)。 **result** 数组大小等于线数的两倍。

```
template<typename T>
class LinearHoughTransform: public HoughTransformConcrete<T>
{
    ...
    virtual int extract(const HoughImage<T> &image, double &result[],
        const int lines = 8) override
    {
        ArrayResize(result, lines * 2);
        ArrayInitialize(result, 0);
        data.zero();

        const int w = image.getWidth();
        const int h = image.getHeight();
        const double d = M_PI / size; // 180 / 36 = 5 degrees, for example
        const double rstep = MathSqrt(w * w + h * h) / size;
        ...
    }
};
```

在直线搜索块中组织图像像素上的嵌套循环。对于每个“填充的”(非零)点, 执行倾斜循环, 并且在变换的空间中标记对应的极坐标对。在这种情况下, 我们只需调用方法, 根据像素返回的值增加单元的内容: **data.inc((int)r, i, v)**, 但是根据应用程序和类型 **T**, 它可能需要更复杂的处理。

```

double r, t;
int i;
for(int x = 0; x < w; x++)
{
    for(int y = 0; y < h; y++)
    {
        T v = image.get(x, y);
        if(v == (T)0) continue;

        for(i = 0, t = 0; i < size; i++, t += d) // t < Math.PI
        {
            r = (x * trigonometric.get(0, i) + y * trigonometric.get(1, i));
            r = MathRound(r / rstep); // range [-range, +range]
            r += size; // [0, +2size]
            r /= 2;

            if((int)r < 0) r = 0;
            if((int)r >= size) r = size - 1;
            if(i < 0) i = 0;
            if(i >= size) i = size - 1;

            data.inc((int)r, i, v);
        }
    }
}
...

```

在该方法的第二部分中，在新空间中搜索最大值，并填充输出数组 *result*。

```

for(i = 0; i < lines; i++)
{
    int x, y;
    if(!findMax(x, y))
    {
        return i;
    }

    double a = 0, b = 0;
    if(MathSin(y * d) != 0)
    {
        a = -1.0 * MathCos(y * d) / MathSin(y * d);
        b = (x * 2 - size) * rstep / MathSin(y * d);
    }
    if(fabs(a) < DBL_EPSILON && fabs(b) < DBL_EPSILON)
    {
        i--;
        continue;
    }
    result[i * 2 + 0] = a;
    result[i * 2 + 1] = b;
}

return i;
}

```

findMax 辅助方法 (参见源代码) 将新空间中最大值的坐标写入 *x* 和 *y* 变量, 额外覆盖该位置的邻域, 以免反复查找。

LinearHoughTransform 类已经准备好了, 我们可以编写一个可导出的工厂函数来生成对象。

```

HoughTransform *createHoughTransform(const int quants,
    const ENUM_DATATYPE type = TYPE_INT) export
{
    switch(type)
    {
        case TYPE_INT:
            return new LinearHoughTransform<int>(quants);
        case TYPE_DOUBLE:
            return new LinearHoughTransform<double>(quants);
        ...
    }
    return NULL;
}

```

因为不允许导出模板, 所以我们在第二个参数中使用 *ENUM_DATATYPE* 枚举来改变转换期间和原始图像表示中的数据类型。

为了测试结构体的导出/导入, 我们还说明了一个带有元信息的结构体, 该元信息关于库的给定版本中的变换, 并导出了一个返回此结构体的函数。


```

struct HoughInfo
{
    const int dimension; // number of parameters in the model formula
    const string about; // verbal description
    HoughInfo(const int n, const string s): dimension(n), about(s) { }
    HoughInfo(const HoughInfo &other): dimension(other.dimension), about(other.about)
};

HoughInfo getHoughInfo() export
{
    return HoughInfo(2, "Line: y = a * x + b; a = p[0]; b = p[1];");
}

```

霍夫变换的各种修改不仅可以揭示直线，还可以揭示对应于给定解析公式的其他结构（例如，圆）。这种修改将揭示不同数量的参数并具有不同的含义。使用自文档化函数更容易集成库（尤其是当有很多库的时候；注意，我们的头文件只包含与实现这个霍夫变换接口的任何库相关的一般信息，而不仅仅是直线信息）。

当然，用公共方法导出类的例子有点片面，因为有可能直接导出变换函数。然而，在实践中，类往往包含更多的功能。特别是，我们很容易在类中添加算法灵敏度的调整、来自线条的示例模式的存储（用于检测在历史数据上验证过的信号）等等。

我们在一个指标中使用库，这个指标根据给定数量柱线的 *High* 和 *Low* 价格来计算支撑线和阻力线。借助霍夫变换和编程接口，该库可以显示数条最重要的支撑线和阻力线。

指标的源代码在 `MQL5/Indicators/MQL5Book/p7/LibHoughChannel.mq5` 文件中。它还包括 `LibHoughTransform.mqh` 头文件，我们在其中添加了导入指令。

```

#import "MQL5Book/LibHoughTransform.ex5"
HoughTransform *createHoughTransform(const int quants,
    const ENUM_DATATYPE type = TYPE_INT);
HoughInfo getHoughInfo();
#import

```

在分析的图像中，我们用像素表示报价中特定价格类型（OHLC）的位置。为了实现图像，我们需要描述从 `Hough Image<int>` 衍生的 `HoughQuotes` 类。

我们将以几种方式“绘出”像素：在烛体内，在蜡烛的全范围内，以及直接在高点和低点处。所有这些都在 `PRICE_LINE` 枚举中形式化了。目前，指标只使用 *HighHigh* 和 *LowLow*，但这可以在设置中取消。

```

class HoughQuotes: public HoughImage<int>
{
public:
    enum PRICE_LINE
    {
        HighLow = 0,    // Bar Range |High..Low|
        OpenClose = 1,  // Bar Body |Open..Close|
        LowLow = 2,      // Bar Lows
        HighHigh = 3,    // Bar Highs
    };
    ...

```

在构造函数参数和内部变量中，我们指定了用于分析的柱线的范围。柱线数量 *size* 决定图像的水平尺寸。为简单起见，我们将使用相同数量的垂直读数。因此，价格离散化步骤 (*step*) 等于 *size* 柱线的实际价格范围 (*pp*) 除以 *size*。对于变量 *base*，我们计算价格的下限，这些价格需要在所示的柱线中考虑。需要使用该变量来绑定线的构造 (基于霍夫变换找到的参数)。

```

protected:
    int size;
    int offset;
    int step;
    double base;
    PRICE_LINE type;

public:
    HoughQuotes(int startbar, int barcount, PRICE_LINE price)
    {
        offset = startbar;
        size = barcount;
        type = price;
        int hh = iHighest(NULL, 0, MODE_HIGH, size, startbar);
        int ll = iLowest(NULL, 0, MODE_LOW, size, startbar);
        int pp = (int)((iHigh(NULL, 0, hh) - iLow(NULL, 0, ll)) / _Point);
        step = pp / size;
        base = iLow(NULL, 0, ll);
    }
    ...

```

回顾一下，*HoughImage* 接口需要实现 3 种方法：*getWidth*、*getHeight* 和 *get*。前两种方法很简单。

```
virtual int getWidth() const override
{
    return size;
}

virtual int getHeight() const override
{
    return size;
}
```

根据从 PRICE_LINE 中选择的计算方法，如果指定的点落在柱线或单元格范围内，则 **get** 方法基于引号获取“像素”返回 1。否则，返回 0。可以通过评估分形、持续增加的极值或具有更高权重（像素加粗）的“舍入”价格，显著改进该方法。

```

virtual int get(int x, int y) const override
{
    if(offset + x >= iBars(NULL, 0)) return 0;

    const double price = convert(y);
    if(type == HighLow)
    {
        if(price >= iLow(NULL, 0, offset + x) && price <= iHigh(NULL, 0, offset + x)
        {
            return 1;
        }
    }
    else if(type == OpenClose)
    {
        if(price >= fmin(iOpen(NULL, 0, offset + x), iClose(NULL, 0, offset + x))
        && price <= fmax(iOpen(NULL, 0, offset + x), iClose(NULL, 0, offset + x)))
        {
            return 1;
        }
    }
    else if(type == LowLow)
    {
        if(iLow(NULL, 0, offset + x) >= price - step * _Point / 2
        && iLow(NULL, 0, offset + x) <= price + step * _Point / 2)
        {
            return 1;
        }
    }
    else if(type == HighHigh)
    {
        if(iHigh(NULL, 0, offset + x) >= price - step * _Point / 2
        && iHigh(NULL, 0, offset + x) <= price + step * _Point / 2)
        {
            return 1;
        }
    }
    return 0;
}

```

convert 帮助方法提供了从像素 *y* 坐标到价格值的重新计算。

```

double convert(const double y) const
{
    return base + y * step * _Point;
}
};

```

现在我们已经为编写指标的技术部分做好了准备。首先，我们声明三个输入变量来选择要分析的片段，以及线数。所有线将由一个共同的前缀来标识。

```

input int BarOffset = 0;
input int BarCount = 21;
input int MaxLines = 3;

const string Prefix = "HoughChannel-";

```

提供变换服务的对象被描述为全局的：这里工厂函数 *createHoughTransform* 从库中调用。

```
HoughTransform *ht = createHoughTransform(BarCount);
```

在 *OnInit* 函数中，我们使用第二个导入函数 *getHoughInfo* 记录库的说明。

```

int OnInit()
{
    HoughInfo info = getHoughInfo();
    Print(info.dimension, " per ", info.about);
    return INIT_SUCCEEDED;
}

```

在柱线开端，我们将在 *OnCalculate* 中进行一次计算。

```

int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
    static datetime now = 0;
    if(now != iTime(NULL, 0, 0))
    {
        ... // see the next block
        now = iTime(NULL, 0, 0);
    }
    return rates_total;
}

```

在不同价格类型形成的一对图像上 (*highs* 和 *lows*)，运行两次变化计算。在这种情况下，由同一对象 *ht* 顺序执行工作。如果成功检测到直线，我们使用 *DrawLine* 函数将它们显示在图表上。因为这些线在结果数组是按重要性降序排列的，所以这些线的权重是递减的。

```

HoughQuotes highs(BarOffset, BarCount, HoughQuotes::HighHigh);
HoughQuotes lows(BarOffset, BarCount, HoughQuotes::LowLow);
static double result[];
int n;
n = ht.transform(highs, result, fmin(MaxLines, 5));
if(n)
{
    for(int i = 0; i < n; ++i)
    {
        DrawLine(highs, Prefix + "Highs-" + (string)i,
            result[i * 2 + 0], result[i * 2 + 1], clrBlue, 5 - i);
    }
}
n = ht.transform(lows, result, fmin(MaxLines, 5));
if(n)
{
    for(int i = 0; i < n; ++i)
    {
        DrawLine(lows, Prefix + "Lows-" + (string)i,
            result[i * 2 + 0], result[i * 2 + 1], clrRed, 5 - i);
    }
}

```

DrawLine 函数基于趋势图形对象 OBJ_TREND, 参见源代码)。

当取消指标的初始化时, 我们删除线和分析对象。

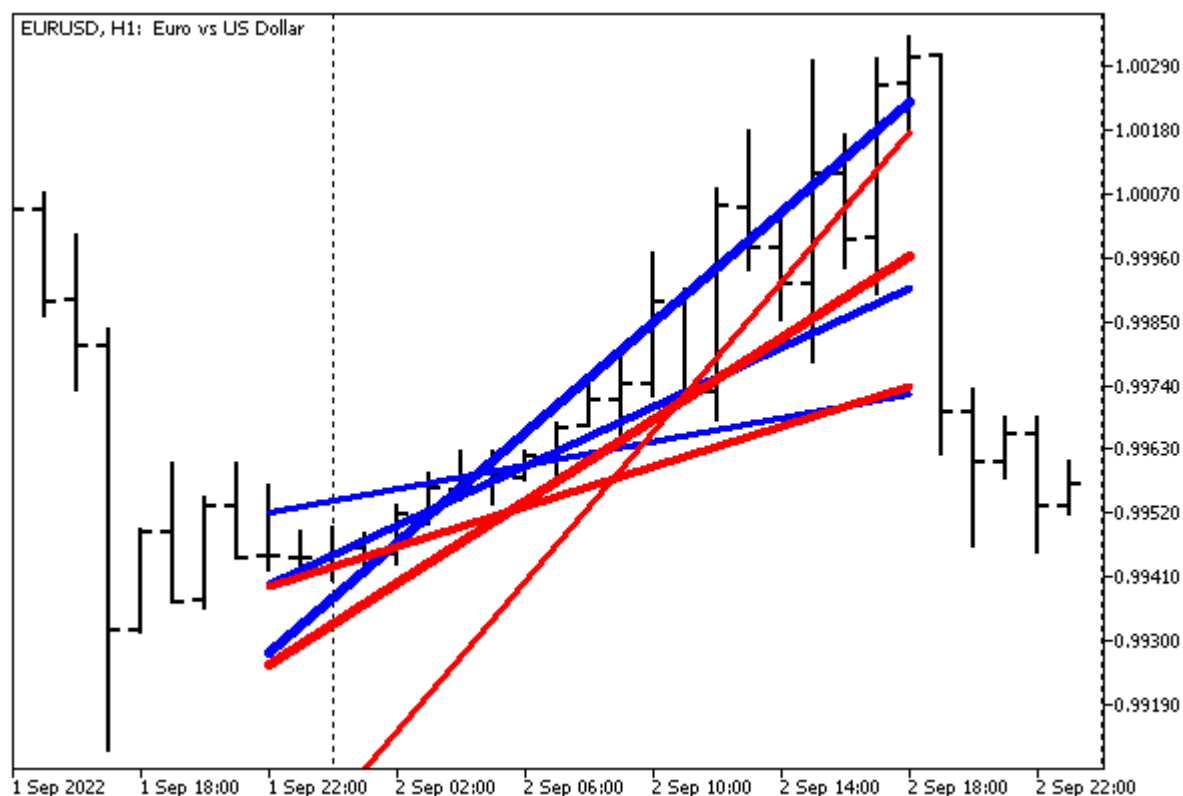
```

void OnDeinit(const int)
{
    AutoPtr<HoughTransform> destructor(ht);
    ObjectsDeleteAll(0, Prefix);
}

```

在测试新的开发之前, 千万别忘了编译库和指标。

使用默认设置运行指标会得到类似这样的结果。



基于霍夫变换库的具有高/低价主线的指标

在我们的例子中，测试是成功的。但是如果需要调试库该怎么办？对此没有内置的工具，那么可以使用下面的技巧。库源代码测试被有条件地编译到产品的调试版本中，并且根据构建的库来测试产品。我们来分析指标的例子。

我们提供 `LIB_HOUGH_MPL_DEBUG` 宏以支持将库源代码直接集成到指标中。宏应该放在包含头文件之前。

```
#define LIB_HOUGH_IMPL_DEBUG
#include <MQL5Book/LibHoughTransform.mqh>
```

在头文件本身中，我们将用预处理程序条件编译指令覆盖来自库的二进制独立副本的导入块。启用宏时，将使用 `#include` 语句运行另一个分支。

```
#ifdef LIB_HOUGH_IMPL_DEBUG
#include "../Libraries/MQL5Book/LibHoughTransform.mq5"
#else
#import "MQL5Book/LibHoughTransform.ex5"
HoughTransform *createHoughTransform(const int quants,
    const ENUM_DATATYPE type = TYPE_INT);
HoughInfo getHoughInfo();
#import
#endif
```

在库源文件 `LibHoughTransform.mq5` 中，在 `getHoughInfo` 函数内部，根据宏是启用还是禁用，我们将输出添加到关于编译方法的信息日志中。

```
HoughInfo getHoughInfo() export
{
#ifdef LIB_HOUGH_IMPL_DEBUG
    Print("inline library (debug)");
#else
    Print("standalone library (production)");
#endif
    return HoughInfo(2, "Line: y = a * x + b; a = p[0]; b = p[1];");
}
```

在 *LibHoughChannel.mq5* 文件中的指标代码中, 如果取消注释 `#define LIB_HOUGH_IMPL_DEBUG` 指令, 则可以测试逐步图像分析。

7.7.6 从 .NET 库导入函数

为使用 .NET 库函数, MQL5 提供了一种特殊服务: 可以简单地导入 DLL 本身, 而无需指定某些函数。MetaEditor 会自动导入可供你使用的所有函数:

- 普通旧式数据 (POD) - 只包含简单数据类型的结构;
 - 公共静态函数 - 其参数仅使用简单 POD 类型和结构或其数组。
- 遗憾的是, 由于这些函数可被 MetaEditor 识别, 目前无法看到函数原型。

例如, 在 *TestLib.dll* 库中, 有以下 *TestClass* 类的 *Inc* 函数的 C# 代码:

```
public class TestClass
{
    public static void Inc(ref int x)
    {
        x++;
    }
}
```

那么, 要导入并调用它, 只需编写:

```
#import "TestLib.dll"

void OnStart()
{
    int x = 1;
    TestClass::Inc(x);
    Print(x);
}
```

执行后, 脚本将返回值 2。

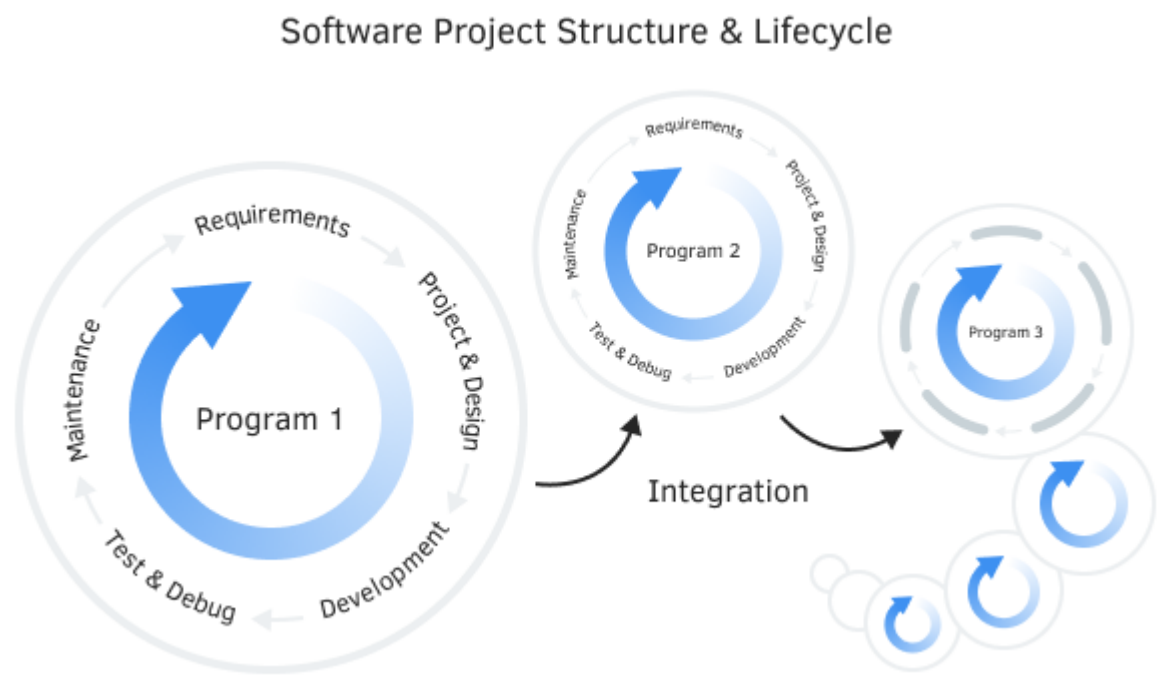
7.8 项目

通常, 软件产品是在标准生命周期内开发的:

- ① 收集和增加需求
- ① 设计
- ① 开发

- ⌚ 测试
- ⌚ 利用

由于功能的不断改进和扩展，通常有必要将源文件、资源和第三方库（这里我们不仅指二进制格式的库，更广义地说，是指任何一组文件，例如头文件）系统化。甚至单个程序被集成到一个体现应用理念的公共产品中。



项目的结构和生命周期

例如，在开发交易机器人时，通常需要连接现成的或定制指标，使用外部机器学习算法意味着编写用于导出报价数据的脚本和用于重新导入训练好的模型的脚本，与通过互联网进行数据交换相关的程序（例如，交易信号）可能需要 Web 服务器及其他编程语言的设置（至少调试和测试需要），即便并非为了部署公共服务。

几个相互关联的产品的整体复合体，连同它们的“依赖项”（这是指所使用的资源和库，它们被独立编写或取自第三方资源），形成了一个软件项目。

当一个程序超过一定规模时，如果没有专门的项目管理工具，那么就很难对其进行方便有效的开发。这一点也完全适用于基于 MQL5 的程序，因为许多交易者使用复杂的交易系统。

MetaEditor 支持类似于其他软件包的项目概念。目前，这一功能还处于开发的初期，到本书发行时，它可能会有所更改。

在 MQL5 中处理项目时，请记住平台中的术语“项目”是指两个不同的实体：

- ⌚ Mproj 文件形式的本地项目
- ⌚ MQL5 云存储中的文件夹

本地项目允许你系统化并收集构建特定 MQL 程序所需的有关源代码、资源和设置的所有信息。此类项目只保存在你的计算机上，可以引用不同文件夹中的文件。

扩展名为 *mproj* 的文件具有广泛使用的通用 JSON (JavaScript 对象表示法) 文本格式。它方便、简单，非常适合说明任何主题领域的信息：所有信息都被分组到具有命名特性的对象或数组

中,支持不同类型的值。这些特征使得 JSON 在概念上非常接近 OOP 语言;此外,它源自面向对象的 JavaScript,这一点很容易从名字中看出。

云存储的运行基于一个版本控制系统和一个名为 SVN (Subversion) 的软件。此处,项目是本地目录 *MQ5/Shared Projects* 中的顶层文件夹,另一个文件夹被分配到该文件夹,该文件夹具有相同的名称,但位于 MQ5 存储服务器上。在项目文件夹中,你可以组织子文件夹的层级。顾名思义,网络项目可以与其他开发人员共享,一般为公开项目 (内容可以由在 mq5.com 上注册的任何人下载)。

该系统提供云中的文件夹映射和本地驱动器上的文件夹映射之间的按需同步 (使用特殊的用户命令),反之亦然。既可以将其他人的项目更改“拉入”你的电脑,也可以将你的编辑“推送”到云端。完整的文件夹映射和选择性文件都可以同步,这当然包括 mq5 文件、mqh 头文件、多媒体、设置 (Set 文件)以及 mqproj 文件。有关云存储的更多信息,请参阅 MetaEditor 和 SVN 系统的文档。

值得注意的是,mqproj 文件的存在并不意味着在其基础上创建任何云项目,就像创建共享文件夹并不强制你使用 mqproj 项目一样。

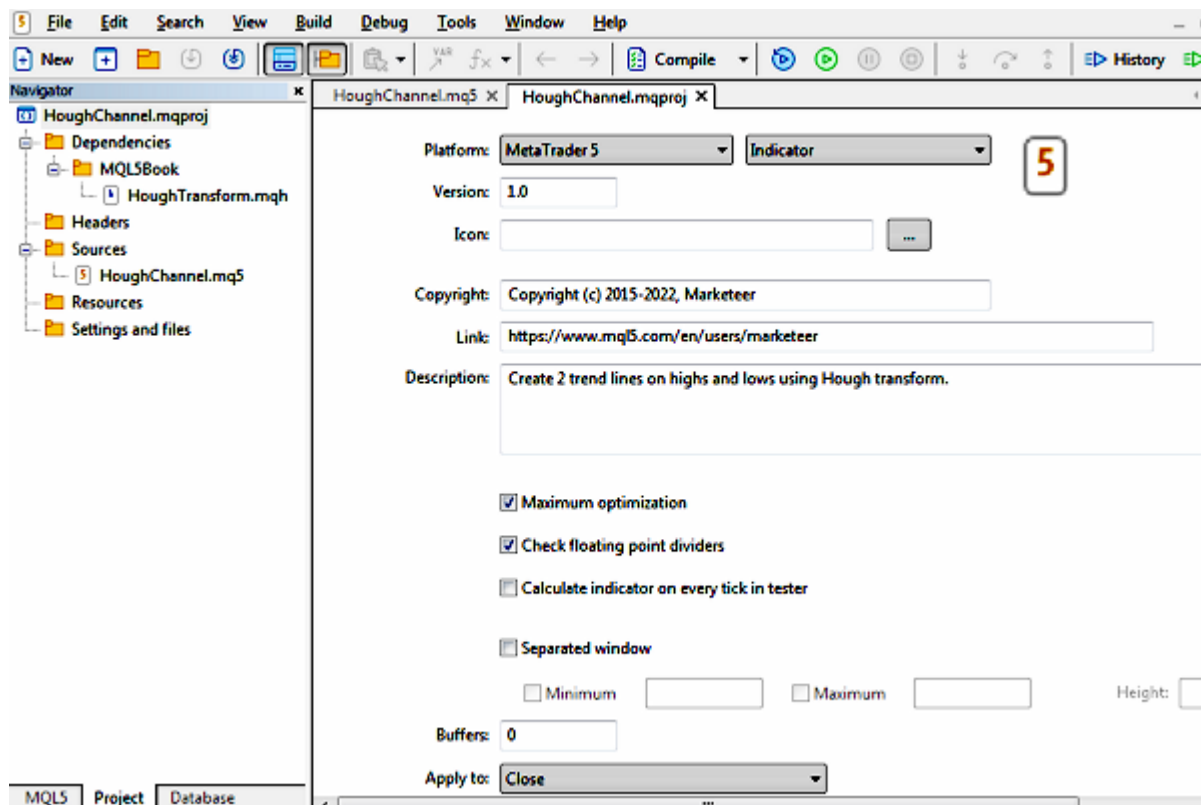
在撰写本文时,mqproj 文件只能描述一个 (而不是多个程序)的结构。然而,由于在开发复杂项目时这种需求很常见,所以将来可能会将这种功能添加到 MetaEditor 中。

在本章中,我们将描述创建和组织 mqproj 项目的主要函数,并给出一组例子。

7.8.1 处理本地项目的一般规则

可以使用 *New project* 或者 *New project from source file* 命令从 MetaEditor 主菜单或者 *Navigator* 上下文菜单创建本地项目 (mqproj 文件)。在后一种情况下,必须首先在 *Navigator* 中选择文件,或者在 *Open* 对话框中选择文件。因此,指定的 mq5 文件将立即包含在项目中。提到的第一个命令启动 MQ5 向导,你应该在其中选择程序类型或空项目选项 (可以稍后再将源文件添加到项目中)。按照向导的常规步骤选择项目的 MQ5 程序类型。

该项目包含几个逻辑部分,类似于包含所有组件的树 (层次结构)。它们显示在 *Navigator* 的左面板中,在一个单独的 *Project* 选项卡中。



导航器和指标项目特性

创建项目后，或稍后通过双击树形结构的根节点，将在窗口的右侧打开一个用于设置 MQL 程序特性的面板。特性集根据程序类型而异。

大多数特性对应于源代码中的 `#property` 指令。这些特性优先：如果在项目和源代码中都指定了它们，将使用项目中的值。

一些开发人员可能喜欢在对话框中交互地设置特性，而不是在源代码中硬编码。此外，你可以在不同的项目中使用同一个 `mq5` 文件，并使用不同的设置构建 MQL 程序的版本（无需更改源代码）。

有些特性仅在项目中可用。例如，这包括启用/禁用编译优化和内置的被零除检查。

在项目编译期间，系统自动分析依赖项，即包含的头文件、资源等。依赖关系出现在项目层次结构的不同分支中。特别是，使用尖括号 (`<filename>`) 的 `#include` 指令中包含的标准 MQL5/Include 文件夹中的头文件属于 **Dependencies**，而使用双引号包含的自定义头文件 (`#include "filename"`) 属于 **Headers** 部分。

此外，用户可以将与完成的软件产品相关的文件添加到项目中，这些文件可能是其正常操作或演示所需要的（例如，具有经过训练的神经网络模型的文件），但不直接嵌入源代码中。出于这些目的，你可以使用 **Settings and Files** 分支。它的上下文菜单包含将单个文件或整个目录添加到项目中的命令。

特别是，我们将进一步考虑不仅包括客户端 MQL 程序，还包括服务器部分的项目示例。

New file 和 **New folder** 将一个新元素添加到包含项目文件的文件夹中：这样的元素总是相对于项目本身进行搜索（在 `mqproj` 文件中，它们以 `relative_to_project` 特性（值为 `true`）进行标记，详见后文）。

Add an existing file 和 *Add an existing folder* 命令从 MQL5 文件夹内的现有目录结构中选择一个或多个元素，m qproj 文件内的这些元素相对于根 MQL5 被引用 (*relative_to_project* 特性为 *false*)。

relative_to_project 特性只是 MetaTrader 5 开发人员定义的少数几个以 JSON 格式表示项目的特性之一。注意，编辑项目 (层次结构和特性) 之后就会构建 JSON 格式的 m qproj 文件。

这是上图中项目文件的样子。

```
{
  "platform"      : "mt5",
  "program_type"  : "indicator",
  "copyright"     : "Copyright (c) 2015-2022, Marketeer",
  "link"          : "https://www.mql5.com/en/users/marketeer",
  "version"       : "1.0",
  "description"   : "Create 2 trend lines on highs and lows using Hough transform.",
  "optimize"      : "1",
  "fpzerocheck"   : "1",
  "tester_no_cache" : "0",
  "tester_everytick_calculate" : "0",
  "unicode_character_set" : "0",
  "static_libraries" : "0",

  "indicator":
  {
    "window": "0"
  },

  "files":
  [
    {
      "path": "HoughChannel.mq5",
      "compile": true,
      "relative_to_project": true
    },
    {
      "path": "MQL5\\Include\\MQL5Book\\HoughTransform.mqh",
      "compile": false,
      "relative_to_project": false
    }
  ]
}
```

我们将在接下来的章节中更详细地讨论 JSON 格式的技术特性，因为我们将在我们的演示项目中应用它。

请务必注意，项目引用的所有文件并不都存储在 m qproj 文件中，因此，复制到新位置或仅将项目文件移动到另一台计算机不会恢复它。为了能够迁移一个项目，须为其建立一个共享项目，并将项目的所有内容上传到云中。但是，这可能需要重组本地文件系统结构，因为所有组件都必须在共享项目文件夹中，而 m qproj 格式没有这样的要求。

7.8.2 用于复制交易和信号的 Web 服务的项目计划

作为我们将在本章中开发的一个端到端的演示项目，我们将采用一个简单但同时技术相当先进的产品：一个客户机-服务器复制交易系统。客户端部分将是 MQL 程序，它使用 [套接字](#) 技术。考虑到 MQL5 只允许使用客户机套接字，因此需要为套接字服务器选择一个替代平台(下面将详细介绍)。因此，该项目将需要几种不同技术的共生，并使用我们已经学习过的多个 MQL5 API，包括在它们的基础上开发的应用程序代码。

由于基于套接字的客户端-服务器架构，该系统可用于不同的场景：

- 便于在一台计算机上的终端之间复制交易；
- 在不同计算机上的终端之间建立专用(个人)通信信道，不仅包括在本地网络中，还包括通过互联网；
- 组织需要登记的公开开放或封闭的信号服务；
- 监控交易；
- 远程管理你自己的帐户。

在所有情况下，客户端程序将扮演两个角色：数据的发布者(发布者、发送者)和订阅者(接收者)。

我们不会发明自己的网络协议，而是使用现有的流行的 WebSocket 标准。他们的客户端实现内置于所有浏览器中，我们需要在 MQL5 中重复这一实现(不同程度的完整性)。当然，WebSocket 支持也适用于大多数流行的 Web 服务器。因此，在任何情况下，我们的开发不仅可以适用于其他服务器(如果其他人适合的话)，还可以与提供类似 Web 服务的知名网站集成。这里的要点是严格遵循他们的 API 规范(建立在 WebSocket 之上)。

开发比一个独立程序更复杂的软件系统时，重要的是要制定一个行动计划，甚至可能要设计一个技术项目，包括模块的结构、它们的相互作用以及编码的顺序。

因此，我们的计划包括：

1. WebSocket 协议的理论分析；
2. 使用 WebSocket 服务器的实现来选择和安装 Web 服务器；
3. 创建一个简单的回显服务器(将传入消息的副本发送回客户机)来熟悉这项技术；
4. 创建一个简单的客户端网页，从浏览器测试回显服务器的功能；
5. 创建一个简单的聊天服务器，向所有连接的客户端发送消息，并为其创建一个测试网页；
6. 在可识别的提供商和订户之间创建消息服务器，并为其创建测试 Web 客户端；
7. MQL5 中 WebSocket 的设计与实现；
8. 创建一个简单的脚本作为回显服务器的客户端；
9. 创建一个简单的 EA 交易作为聊天服务器客户端；
10. 最后，在 MQL5 中创建一个交易复制器，它将根据设置充当信息提供者(监视帐户变化和状态)和信息消费者(复制交易)。

但是在我们开始实施这个计划之前，我们需要安装一个 Web 服务器。

基于 Nodejs 的 Web 服务器

为了组织我们项目的服务器部分，我们需要一个 Web 服务器。我们将使用最轻、技术最先进的节点。它的服务器端脚本可以用 JavaScript 编写，JavaScript 与交互式 Web 浏览器中使用的语

言相同。从系统的客户机和服务器部分的统一编写的角度来看,这是方便的;一般来说,任何 Web 服务的客户端部分迟早都是需要的,例如,用于管理、注册和显示服务使用的漂亮统计数据。

任何了解 MQL5 的人实际上都了解 JavaScript,所以相信自己。侧栏中讨论了主要的区别。

MQL5 vs JavaScript

JavaScript 是一种解释语言,而 MQL5 是编译语言。对于我们这些开发人员来说,这会减少我们很多负担,因为我们不需要单独的编译阶段来获得一个工作程序。不用担心 JavaScript 的效率:所有的 JavaScript 运行时都是按需使用 JavaScript 的 JIT (just-in-time) 编译,也就是第一次访问模块的时候。这个过程在每个会话中自动隐式地发生一次,之后脚本以编译后的形式执行。

MQL5 指的是静态类型化的语言,即在描述变量时,我们必须显式指定其类型,编译器监控类型兼容性。相比之下,JavaScript 是一种动态类型语言:变量的类型由我们放入的值决定,并且可以在变量的生命周期中改变。这提供了灵活性,但需要谨慎,以避免不可预见的错误。

从某种意义上说,JavaScript 是一种比 MQL5 更面向对象的语言,因为其中几乎所有的实体都是对象。例如,函数也是对象,而作为对象特性描述符的类也是 (原型的)对象。

JavaScript 本身“收集垃圾”,即释放应用程序为对象分配的内存。在 MQL5 中,我们必须为动态对象提供及时的 *delete* 调用。

JavaScript 语法包含许多方便的“缩写”,而 MQL5 中实现相同功能则需要编写更冗长的代码结构。例如,为了将指向另一个函数的参数传递给 MQL5 中的某个函数,我们需要使用 *typedef*,单独定义一个与这个原型匹配的函数,然后将它的标识符作为参数传递。而在 JavaScript 中,你可以定义你所指向的函数 (完整地!)而不是指针参数。

如果你是 Web 开发人员或者已经熟悉 node.js,可以跳过安装和配置步骤。

可以从官方网站下载 node.js nodejs.org。安装有不同的版本,例如,使用安装程序或解压缩归档文件。安装后,你将在指定的 *node.exe* 目录中收到一个可执行文件以及若干支持文件和文件夹。

如果安装程序没有将 node.js 添加到系统路径中,可以通过在安装 node.js 的文件夹 (*node.exe* 文件所在的位置)中运行以下命令来为当前 Windows 用户添加:

```
setx PATH "%CD%"
```

或者,你可以从系统特性对话框 (*Computer-> Properties -> Extra options ->Environment Variables* 编辑 Windows 环境变量;具体对话框类型取决于操作系统的版本)。无论如何,通过这种方式,我们将确保从计算机上的任何文件夹运行 node.js 的能力,这在将来对我们是有用的。

你可以通过运行以下命令 (在 Windows 命令行中)来检查 node.js 的运行状况:

```
node -v
npm version
```

第一个命令输出 node.js 的版本,第二个命令输出一个重要的内置 node.js 服务, *npm* 包管理器。

包是一个现成的模块,它向 node.js 添加特定的功能。node.js 本身非常小,如果没有包,它将需要大量的例行编码。

请求最多的包存储在 Web 上的中央存储库中，可以下载并安装在 nodejs 的特定副本上或全局安装（如果机器上有几个 nodejs 的副本，则为 nodejs 的所有副本）。使用以下命令将软件包安装到特定副本：

```
npm install <package name>
```

在安装 nodejs 的文件夹中运行该软件包。此命令将把软件包放在本地，不会影响已经存在的或以后可能出现在计算机上的 nodejs 的其他副本，也不会影响意外的编辑。

我们尤其需要 ws 软件包，它实现了 WebSocket 协议。也就是说，你需要运行以下命令：

```
npm install ws
```

等待该过程完成。运行结果是，<nodejs_install_path>/node_modules/ 文件夹应包含一个新的子文件夹 **ws**，其中包含必要的内容（你可以查看 README.md 文件中包的说明，以确保它是一个 WebSocket 协议库）。

该软件包中包含了服务器和客户端的实现。我们不使用后者，而是在 MQL5 中编写自己的客户端。

Nodejs 服务器的所有功能都集中在 **/node_modules** 文件夹中。就用途而言，它与 MetaTrader5 中的标准文件夹 **MQL5/Include** 相当。用 JavaScript 编写应用程序时，我们将以一种特殊的方式包含或“导入”必要的模块，类似于在 MQL5 中使用 **#include** 指令包含 mqh 头文件。

7.8.4 WebSocket 协议的理论基础

WebSocket 协议是建立在 TCP/IP 网络连接之上的，这些网络连接的特征是 IP 地址（或替换它的域名）和端口号。HTTP/HTTPS 协议，我们已经在关于 [网络功能](#) 的章节中实践过的 HTTP/HTTPS 协议，就是基于相同的原理工作的。其中，标准端口号是 80（用于不安全的连接）和 443（用于安全的连接）。WebSocket 没有专用的端口号，所以 Web 服务提供商可以选择任何可用的端口号。我们所有的例子都将使用 9000 端口。

将 URL 指定为 WebSocket 协议前缀时，我们使用 **ws**（用于非安全连接）和 **wss**（用于安全连接）。

就数据传输而言，WebSocket 格式比 HTTP 更高效，因为它使用的控制数据少得多。

WebSocket 服务的初始连接建立完全重复一个 HTTP/HTTPS Web 页面请求：你需要发送一个带有专门准备的标头的 GET 请求。这些标头的一个特征是存在文本行：

```
Connection: Upgrade
Upgrade: websocket
```

以及一些报告 WebSocket 协议版本和特殊随机生成的字符串的附加行。客户端和服务端之间“握手”过程中涉及的密钥。

```
Sec-WebSocket-Key: ...
Sec-WebSocket-Version: 13
```

实际上，“握手”意味着服务器检查客户机请求的那些选项的可用性，并以标准的 HTTP 标头作为响应，确认切换到 WebSocket 模式或拒绝。拒绝的最简单的原因可能是，如果你试图通过 WebSocket 连接到一个简单的 Web 服务器，其中没有提供 WebSocket 服务器或者不支持所需的版本。

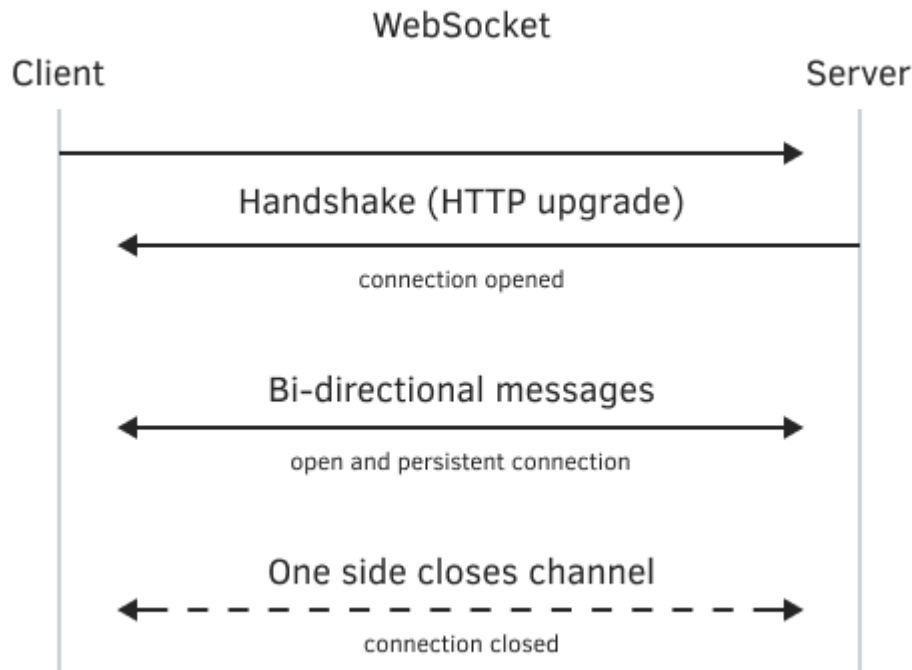
WebSocket 协议的当前版本的代号为 Hybi 和 13 号。更早、更简单的版本 Hixie 可能有助于向后兼容。在下文中，我们将只使用 Hybi，尽管 Hixie 实现也包括在内。

服务器响应中的以下 HTTP 标头表示连接成功：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: ...
```

这里的 `Sec-WebSocket-Accept` 字段由服务器根据 `Sec-WebSocket-Key` 计算和填充，以确认符合协议。所有这些都受规范的约束 [RFC6455](#) 也将我们的 MQL 项目中得到支持。

为清晰起见，下图显示了该过程：



客户端和服务端之间通过 WebSocket 协议进行交互

建立 WebSocket 连接后，客户机和服务器可以交换打包到特殊块中的信息：帧和消息。一条消息可以由一个或多个帧组成。根据规范，帧大小被限制为天文数字 2^{63} 字节（ $9223372036854775807 \sim 9.22$ 艾字节！），但是特定的实现当然可能具有更普通的限制，因为这种理论限制对于在一个分组中发送似乎是不实际的。

在任何时候，客户端或服务器都可以终止连接，只要事先“礼貌地说再见”（见下文）或简单地关闭网络套接字。

帧可以是不同的类型，如每个帧开头的标头（4 到 16 个字节）所指定的。作为参考，我们先列出操作码（它们存在于标头的第一个字节）和不同类型帧的用途。

- 0 -连续帧（继承前一帧的特性）；
- 1 -带文本信息的帧；
- 2 -带二进制信息的帧；
- 8 -请求关闭并确认关闭连接的帧（发送用于“礼貌告别”）；
- 9 -ping 帧，可由任一端定期发送，以确保连接得到物理保存；
- 10 -pong 帧，为响应 ping 帧而发送。

消息中的最后一帧在标头中用一个特殊的位来标记。当然，当消息由一帧组成时，它也是最后一帧。有效载荷的长度也在标头中传递。

7.8.5 基于 WebSocket 协议的 Web 服务的服务器组件

为了组织所有项目的公共服务器组件，我们将在 *MQL5/Experts/MQL5Book/p7/* 内部创建一个单独的 **Web** 文件夹。理想情况下，将 **Web** 作为子文件夹放在 *Shared Projects* 共享项目中会很方便。事实是，MetaTrader 5 的标准发行版中提供了 *MQL5/Shared Projects*，并被保留用于云存储项目。因此，以后，通过使用共享项目的功能，将有可能把我们项目的文件上传到服务器（不仅仅是 Web 文件，还有 MQL 程序）。

稍后，当我们使用 MQL5 客户端程序创建 *mqlproj* 文件时，会将该文件夹中的所有文件添加到项目的 *Settings and Files* 部分，因为所有这些文件构成了项目的组成部分，即服务器部分。

由于已经为项目服务器分配了一个单独的目录，因此必须确保可以从该目录中的 *nodejs* 导入模块。默认情况下，*nodejs* 在当前目录的 */node_modules* 子文件夹中查找模块，我们将从项目中运行服务器。因此，在我们将放置该项目的 Web 文件的文件夹中，运行命令：

```
mklink /j node_modules {drive:/path/to/folder/nodejs}/node_modules
```

其结果是，将出现一个名为 *node_modules* 的“符号”目录链接，指向已安装的 *nodejs* 中同名的原始文件夹。

要检查 *WebSockets* 功能，最简单方法是使用回显服务。它的操作模型是将任何接收到的消息返回给发送者。让我们考虑如何以最小的配置来组织这样的服务。*wsintro.js* 文件中包含了一个例子。

首先，我们连接 *ws* 包（模块），它为 *nodejs* 提供 *WebSocket* 功能，这个包是我们随 Web 服务器一起安装的。

```
// JavaScript
const WebSocket = require('ws');
```

require 函数的工作方式类似于 MQL5 中的 *#include* 指令，但会另外返回一个模块对象，其中包含 *ws* 包中所有文件的 API。得益于此，我们可以调用 *WebSocket* 对象的方法和特性。在这种情况下，我们需要在端口 9000 上创建一个 *WebSocket* 服务器。

```
// JavaScript
const port = 9000;
const wss = new WebSocket.Server({ port: port });
```

这里我们看到 *new* 运算符调用了常见的 MQL5 构造函数，但是一个未命名的对象（结构）被作为参数传递，在这个参数中，像在映射中一样，可以存储一组命名特性及它们值。在这种情况下，只使用一个 *port* 特性，它的值被设置为等于上述 *port* 变量（更准确地说，是一个常量）。基本上，我们可以在运行脚本时在命令行中传递端口号（以及其他设置）。

服务器对象进入 *wss* 变量。成功后，我们向命令行窗口发出信号，表明服务器正在运行（等待连接）。

```
// JavaScript
console.log('listening on port: ' + port);
```

console.log 调用类似于 MQL5 中通常 *Print*。还要注意，JavaScript 中的字符串不仅可以用双引号，还可以用单引号括起来，甚至可以用反斜杠 *`this is a \${template}text`*，这增加了一些有用的功能。

随后，对于 `wss` 对象，我们分配一个“连接”事件处理程序，它指向一个新的客户端连接。显然，支持对象事件列表是由包的开发者定义的，在这种情况下，就是我们使用 `ws` 包。所有这些都反映在文档中。

处理程序由 `on` 方法绑定，该方法指定事件的名称和处理程序。

```
// JavaScript
wss.on('connection', function(channel)
{
    ...
});
```

处理程序是一个未命名（匿名）函数，直接定义在回调代码在新连接上执行时需要引用参数的地方。这个函数是匿名的，因为它只在这里使用，JavaScript 允许这种语法简化。该函数只有一个参数，即新连接的对象。我们可以自己选择参数的名称，在本例中是 `channel`。

在处理程序内部，应该为与特定通道中新消息的到达相关的“消息”事件设置另一个处理程序。

```
// JavaScript
channel.on('message', function(message)
{
    console.log('message: ' + message);
    channel.send('echo: ' + message);
});
...
```

它还使用一个匿名函数，该函数只有一个参数，即接收消息的对象。我们将它打印到控制台日志中进行调试。但是最重要的事情发生在第二行：通过调用 `channel.send`，我们向客户端发送一条响应消息。

为了进一步完善，让我们向“连接”处理程序添加我们自己的欢迎消息。完成后，如下所示：

```
// JavaScript
wss.on('connection', function(channel)
{
    channel.on('message', function(message)
    {
        console.log('message: ' + message);
        channel.send('echo: ' + message);
    });
    console.log('new client connected!');
    channel.send('connected!');
});
```

重要的是要理解，虽然在代码中绑定“消息”处理程序比发送“hello”更重要，但是消息处理程序将在以后调用，并且只有当客户端发送消息时才调用。

我们已经查看了组织回显服务的脚本大纲。不过，最好是测试一下。这可以通过使用常规浏览器以最有效的方式完成，但这需要将脚本稍微复杂化一些：将它变成尽可能小的 Web 服务器，用尽可能小的 WebSocket 客户端返回网页。

回显服务和测试网页

我们现在要查看的回显服务器脚本位于 `wsecho.js` 文件中。要点之一是，不仅需要支持 `http/ws` 服务器上的开放协议，还需要支持受保护协议 `https/wss`。这种可能性将在我们所有的示例中提供（包括基于MQ15的客户机），但你需要在服务器上执行一些操作。

你应该从包含加密密钥和证书的几个文件开始。这些文件通常从授权来源（即认证中心）获得，但出于提供信息的目的，你可以自己生成这些文件。当然，你自己生成的这些文件不能在公共服务器上使用，带有此类证书的页面会在任何浏览器中触发警告（地址栏左侧的页面图标以红色突出显示）。

证书设备的描述以及证书自身的生成过程超出了本书的范围，但是本书包含了两个现成的文件：持续时间有限的 `MQ15Book.crt` 和 `MQ15Book.key`（还有其他扩展文件）。这些文件必须传递给 Web 服务器对象的构造函数，以便服务器能够通过 HTTPS 协议工作。

我们将在脚本启动命令行中传递证书文件的名称。例如，像这样：

```
node wsecho.js MQ15Book
```

如果运行脚本时没有附加参数，服务器将使用 HTTP 协议工作。

```
node wsecho.js
```

在脚本内部，命令行自变量可通过内置对象 `process.argv` 获得，前两个参数始终包含 `node.exe` 服务器的名称和要运行的脚本的名称（在本例中为 `wsecho.js`），因此我们通过 `splice` 方法将其丢弃。

```
// JavaScript
const args = process.argv.slice(2);
const secure = args.length > 0 ? 'https' : 'http';
```

根据现存的证书名称，`secure` 变量将获取创建服务器后应该加载的包的名称：`https` 或 `http`。代码中总共有 3 个依赖项：

```
// JavaScript
const fs = require('fs');
const http1 = require(secure);
const WebSocket = require('ws');
```

我们已经了解了 `ws` 包的所有内容；`https` 和 `http` 包提供了一个 Web 服务器实现，内置的 `fs` 包提供了文件系统的使用。

Web 服务器设置被格式化为 `options` 对象。此处，我们可以看到命令行中的证书名称是如何使用 `${args[0]}` 表达式在字符串中用斜线引号替换的。然后通过 `fs.readFileSync` 方法读取相应的文件对。

```
// JavaScript
const options = args.length > 0 ?
{
  key : fs.readFileSync(`${args[0]}.key`),
  cert : fs.readFileSync(`${args[0]}.crt`)
} : null;
```

Web 服务器是通过调用 `createServer` 方法创建的，我们将选项对象和一个匿名函数（HTTP 请求处理程序）传递给该方法。该处理程序有两个参数：带有 HTTP 请求的 `req` 对象和我们用来发送响应的 `res` 对象（HTTP 头文件和 Web 页面）。

```
// JavaScript
http1.createServer(options, function (req, res)
{
  console.log(req.method, req.url);
  console.log(req.headers);

  if(req.url == '/') req.url = "index.htm";

  fs.readFile('./' + req.url, (err, data) =>
  {
    if(!err)
    {
      var dotoffset = req.url.lastIndexOf('.');
      var mimetype = dotoffset == -1 ? 'text/plain' :
      {
        '.htm' : 'text/html',
        '.html' : 'text/html',
        '.css' : 'text/css',
        '.js' : 'text/javascript'
      }[ req.url.substr(dotoffset) ];
      res.setHeader('Content-Type',
        mimetype == undefined ? 'text/plain' : mimetype);
      res.end(data);
    }
    else
    {
      console.log('File not found: ' + req.url);
      res.writeHead(404, "Not Found");
      res.end();
    }
  });
}).listen(secure == 'https' ? 443 : 80);
```

主索引页面 (也是唯一的页面)是 *index.htm* (待编写)。此外, 处理程序还可以发送 *js* 和 *css* 文件, 后面我们会用到这些文件。根据是否启用了保护模式, 通过在 443 或 80 标准端口上调用 *listen* 方法来启动服务器 (如果你的计算机上已经安装了这些端口, 请更改为其他端口)。

要接受 Web 套接字端口 9000 上的连接, 我们需要部署另一个采用了相同选项的 Web 服务器实例。但在这种情况下, 服务器存在的唯一目的是处理 HTTP 请求, 将连接“升级”到 Web Sockets 协议。

```
// JavaScript
const server = new http1.createServer(options).listen(9000);
server.on('upgrade', function(req, socket, head)
{
  console.log(req.headers); // TODO: we can add authorization!
});
```

这里, 在“upgrade”事件处理程序中, 我们接受任何已经通过握手的连接, 并将头打印到日志中, 但是如果我们正在做一个封闭的(付费的)服务, 我们可能会请求用户授权。

最后, 我们创建一个 WebSocket 服务器对象, 和前面的介绍性例子一样, 唯一的区别是一个现成的 Web 服务器被传递给了构造函数。所有连接的客户端都按序列号进行计数和欢迎。

```
// JavaScript
var count = 0;

const wsServer = new WebSocket.Server({ server });
wsServer.on('connection', function onConnect(client)
{
    console.log('New user:', ++count);
    client.id = count;
    client.send('server#Hello, user' + count);

    client.on('message', function(message)
    {
        console.log('%d : %s', client.id, message);
        client.send('user' + client.id + '#' + message);
    });

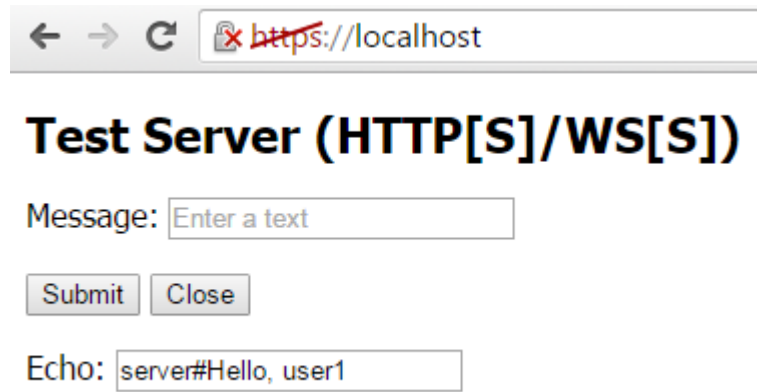
    client.on('close', function()
    {
        console.log('User disconnected:', client.id);
    });
});
```

对于所有事件（包括连接、断开和消息），调试信息都显示在控制台中。

好了，有 Web Socket 服务器支持的 Web 服务器准备好了。现在我们需要为它创建一个客户端网页 *index.htm*。

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Server (HTTP[S]/WS[S])</title>
  </head>
  <body>
    <div>
      <h1>Test Server (HTTP[S]/WS[S])</h1>
      <p><label>
        Message: <input id="message" name="message" placeholder="Enter a text">
      </label></p>
      <p><button>Submit</button> <button>Close</button></p>
      <p><label>
        Echo: <input id="echo" name="echo" placeholder="Text from server">
      </label></p>
    </div>
  </body>
  <script src="wsecho_client.js"></script>
</html>
```

该页面是一个表单，有一个输入字段和一个发送消息的按钮。



WebSocket上的回显服务网页

该页面使用 `wsecho_client.js` 脚本，该脚本提供 WebSocket 客户端响应。在浏览器中，WebSocket 内置为“本地”JavaScript 对象，因此你不需要连接任何外部设备：只需使用所需的协议和端口号调用 `web socket` 构造函数。

```
// JavaScript
const proto = window.location.protocol.startsWith('http') ?
    window.location.protocol.replace('http', 'ws') : 'ws: ';
const ws = new WebSocket(proto + '//' + window.location.hostname + ':9000');
```

URL 由当前网页的地址 (`window.location.hostname`) 构成，因此 WebSocket 连接是针对同一服务器的。

随后，`ws` 对象允许你对事件做出反应并发送消息。在浏览器中，打开的连接事件被称为“打开”；它通过 `onopen` 特性连接。相同语法（与服务器实现略有不同）也用于新消息到达事件 - 它的处理程序被分配给 `onmessage` 特性。

```
// JavaScript
ws.onopen = function()
{
    console.log('Connected');
};

ws.onmessage = function(message)
{
    console.log('Message: %s', message.data);
    document.getElementById('echo').value = message.data;
};
```

传入消息的文本显示在 id 为“echo”的表单元素中。请注意，消息事件对象（处理程序参数）不是 `data` 特性中可用的消息。这是 JavaScript 中的一个实现特征。

对表单按钮的反应是使用 `addEventListener` 方法为每个 `button` 标记对象分配的。这里我们看到了 JavaScript 中说明匿名函数的另一种方式：自变量列表的圆括号可以是空的，箭头后的函数体可以是 `(arguments) => { ... }`。

```
// JavaScript
const button = document.querySelectorAll('button'); // request all buttons
// button "Submit"
button[0].addEventListener('click', (event) =>
{
    const x = document.getElementById('message').value;
    if(x) ws.send(x);
});
// button "close"
button[1].addEventListener('click', (event) =>
{
    ws.close();
    document.getElementById('echo').value = 'disconnected';
    Array.from(document.getElementsByTagName('button')).forEach((e) =>
    {
        e.disabled = true;
    });
});
```

为了发送消息，我们调用 **ws.send** 方法，为了关闭连接，我们调用 **ws.close** 方法。

这就完成了演示回显服务的第一个客户机/服务器脚本示例的开发。你可以使用前面显示的一个命令运行 **wsecho.js**，然后在浏览器中打开位于 **http://localhost** 或者 **https://localhost** (取决于服务器设置) 的页面。表单出现在屏幕上后，尝试与服务器聊天，并确保服务正在运行。

逐渐将这个例子变复杂，我们将为复制交易信号的 Web 服务铺平道路。但下一步将是聊天服务，其原理类似于交易信号的服务：来自一个用户的消息被传输给其他用户。

聊天服务和测试网页

新服务器脚本名为 **wschat.js**，它与 **wsecho.js** 之间有大量重复。让我们列出主要的区别。在 Web 服务器 HTTP 请求处理程序中，将初始页面从 **index.htm** 更改为 **wschat.htm**。

```
// JavaScript
http1.createServer(options, function (req, res)
{
    if(req.url == '/') req.url = "wschat.htm";
    ...
});
```

为了存储连接到聊天的用户相关信息，我们将说明 **clients** 映射数组。**Map** 是一个标准的 JavaScript 关联容器，可以使用任意类型的键将任意值写入其中 (包括对象)。

```
// JavaScript
const clients = new Map(); // added this line
var count = 0;
```

在新的用户连接事件处理程序中，我们将把作为函数参数接收的 **client** 对象添加到当前客户端序列号下的映射中。

```
// JavaScript
wsServer.on('connection', function onConnect(client)
{
  console.log('New user:', ++count);
  client.id = count;
  client.send('server#Hello, user' + count);
  clients.set(count, client);           // added this line
  ...
}
```

在 `onConnect` 函数中, 我们为特定客户端的新消息到达事件设置了一个处理程序, 而我们就是在这个嵌套的处理程序中发送消息的。然而, 这一次我们遍历了映射的所有元素 (也就是说, 遍历了所有客户端), 并将文本发送给每个客户端。该循环是通过 `forEach` 方法从映射调用数组来组织的, 将对每个元素 (`elem`) 执行的下一个匿名函数被传递给相应的方法。循环结构的例子清楚地展示了 JavaScript 中流行的 *functional-declarative* 编程范式 (与 MQL5 中的 *imperative* 方法形成对比)。

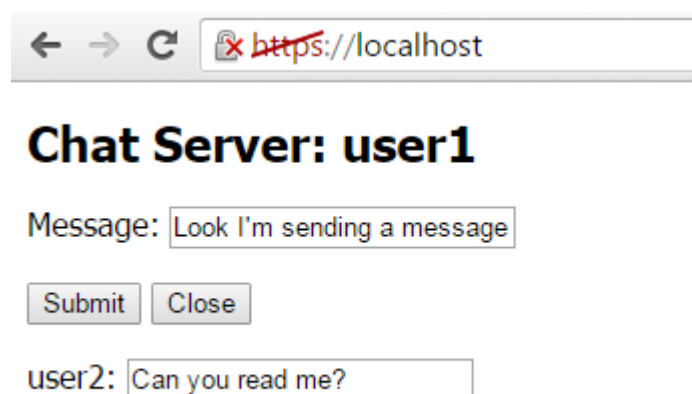
```
// JavaScript
client.on('message', function(message)
{
  console.log('%d : %s', client.id, message);
  Array.from(clients.values()).forEach(function(elem) // added a loop
  {
    elem.send('user' + client.id + '#' + message);
  });
});
```

需要注意的是, 我们会将邮件的副本发送给所有客户, 包括原作者。它可能会被过滤掉, 但是出于调试的目的, 最好确认消息已经发送。

与之前的回显服务的最后一个区别是, 当客户端断开连接时, 需要将其从映射中删除。

```
// JavaScript
client.on('close', function()
{
  console.log('User disconnected:', client.id);
  clients.delete(client.id);           // added this line
});
```

关于用 `wschat.htm` 替换 `index.htm` 页面, 我们在这里添加了一个“字段”来显示消息的作者 (*origin*), 并连接了一个新的浏览器脚本 `wschat_client.js`。该脚本可解析消息 (我们使用 '#' 符号将作者与文本分开), 并用收到的信息填充表单字段。因为从 WebSocket 协议的角度来看没有什么变化, 所以我们不会提供源代码。



基于 WebSocket 的聊天服务网页

你可以用 `wschat.js` 聊天服务器启动 `nodejs`, 然后从几个浏览器选项卡连接到它。每个连接都会在头文件中显示一个唯一编号。单击 **Submit** 后, **Message** 字段中的文本将发送给所有客户端。然后, 客户端表单显示消息的作者 (左下方的标签) 和文本本身 (底部中间的字段)。

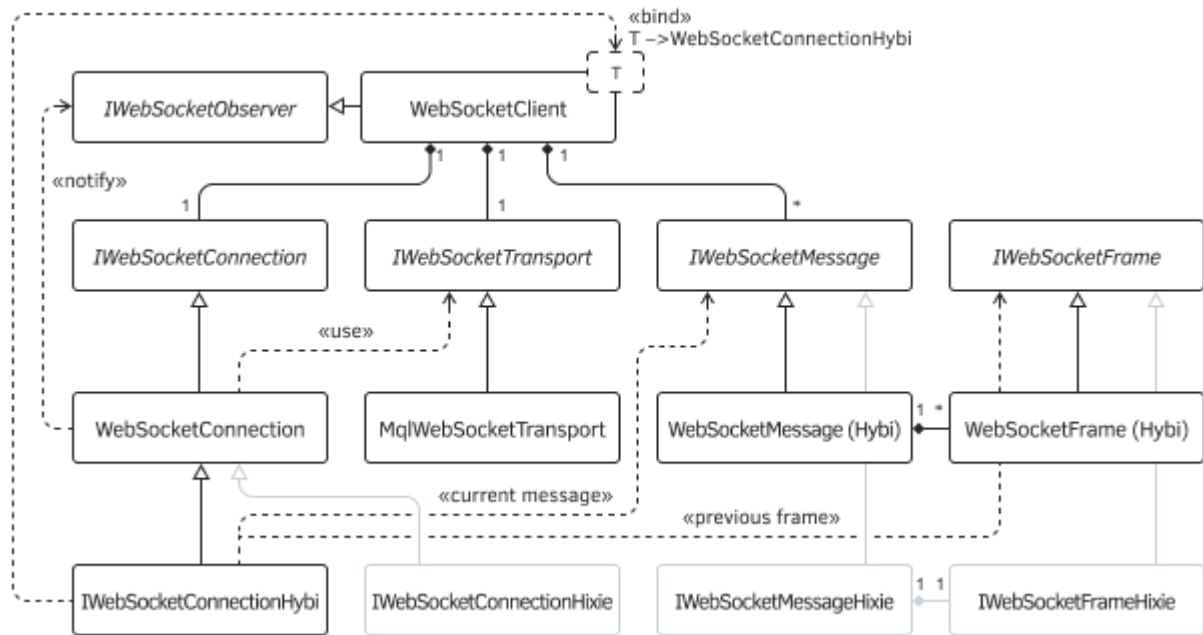
因此, 我们已经确保支持 WebSocket 的 Web 服务器已经准备好了。让我们转向用 MQL5 编写协议的客户端部分。

7.8.6 MQL5 中的 WebSocket 协议

我们之前已经研究过 WebSocket 协议的理论基础。完整的规范非常广泛, 要详细介绍其实现方式需要占用大量篇幅。因此, 我们提出了现成类的一般结构及其编程接口。所有文件都位于 `MQL5/Include/MQL5Book/ws/` 目录中。

- `wsinterfaces.mqh` - 所有接口、常量和类型的一般抽象描述。
- `wstransport.mqh` - `MqWebSocketTransport` 类, 可基于 MQL5 套接字函数实现 `WebSocketTransport` 低级网络数据传输接口;
- `wsframe.mqh` - `WebSocketFrame` 和 `WebSocketFrameHixie` 类, 可实现 `WebSocketFrame` 接口, 该接口隐藏了分别为 Hybi 和 Hixie 协议生成 (编码和解码) 帧的算法;
- `wsmessag.mqh` - `WebSocketMessage` 和 `WebSocketMessageHixie` 类, 它们实现 `WebSocketMessage` 接口, 分别用于规范化基于 Hybi 和 Hixie 协议从帧构建消息的过程;
- `wsprotocol.mqh` - 继承自 `WebSocketConnection` 的 `WebSocketConnectionHybi`、`WebSocketConnectionHixie` 类; 这些类根据规范对帧、消息、问候和断开的形成进行协调管理, 为此使用了上述接口;
- `wsclient.mqh` - `WebSocket` 客户端的现成实现; `WebSocketClient` 模板类, 支持 `WebSocketObserver` 接口 (用于事件处理) 并期望 `WebSocketConnectionHybi` 或 `WebSocketConnectionHixie` 作为参数化类型;
- `wstools.mqh` - `WSTools` 名称空间中有用的实用工具。

这些头文件都将作为 `#include` 指令的依赖项自动包含在我们未来的 `mqlproj` 项目中。



MQL5 中的 WebSocket 类图

底层网络接口 *IWebSocketTransport* 有以下方法。

```

interface IWebSocketTransport
{
    int write(const uchar &data[]); // write the array of bytes to the network
    int read(uchar &buffer[]);      // read data from network into byte array
    bool isConnected(void) const;   // check for connection
    bool isReadable(void) const;    // check for the ability to read from the network
    bool isWritable(void) const;    // check for the possibility of writing to the net
    int getHandle(void) const;      // system socket descriptor
    void close(void);               // close connection
};

```

从这些方法的名称中不难猜出将使用哪些 MQL5 API 套接字函数来构建它们。但如果需要，也可以通过他们自己的方式实现这个接口，例如，通过一个 DLL。

实现此接口的 *MqlWebSocketTransport* 类需要在创建实例时建立网络连接的协议、主机名和端口号。此外，你可以指定超时值。

帧类型在 `WS_FRAME_OPCODE` 枚举中收集。

```

enum WS_FRAME_OPCODE
{
    WS_DEFAULT = 0,
    WS_CONTINUATION_FRAME = 0x00,
    WS_TEXT_FRAME = 0x01,
    WS_BINARY_FRAME = 0x02,
    WS_CLOSE_FRAME = 0x08,
    WS_PING_FRAME = 0x09,
    WS_PONG_FRAME = 0x0A
};

```

用于处理框架的接口包含与框架实例相关的静态和常规方法。静态方法充当工厂，由发送方创建所需类型的帧 (*create*) 和传入帧 (*decode*)。

```

class IWebSocketFrame
{
public:
    class StaticCreator
    {
    public:
        virtual IWebSocketFrame *decode(uchar &data[], IWebSocketFrame *head = NULL) =
        virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const string data = NULL,
            const bool deflate = false) = 0;
        virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const uchar &data[],
            const bool deflate = false) = 0;
    };
    ...

```

由于存在 *Creator* 模板和返回该模板的 *getCreator* 方法的实例 (假设返回 "singleton"), 则工厂方法在子类中是强制存在的。

```

protected:
    template<typename P>
    class Creator: public StaticCreator
    {
    public:
        // decode received binary data in IWebSocketFrame
        // (in case of continuation, previous frame in 'head')
        virtual IWebSocketFrame *decode(uchar &data[],
            IWebSocketFrame *head = NULL) override
        {
            return P::decode(data, head);
        }
        // create a frame of the desired type (text/closing/other) with optional text
        virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const string data = NULL,
            const bool deflate = false) override
        {
            return P::create(type, data, deflate);
        };
        // create a frame of the desired type (binary/text/closure/other) with data
        virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const uchar &data[],
            const bool deflate = false) override
        {
            return P::create(type, data, deflate);
        };
    };
public:
    // require a Creator instance
    virtual IWebSocketFrame::StaticCreator *getCreator() = 0;
    ...

```

接口的其余方法提供了对帧中数据的所有必要操作 (编码/解码、接收数据和各种标志)。

```

// encode the "clean" contents of the frame into data for transmission over the ne
virtual int encode(uchar &encoded[]) = 0;

// get data as text
virtual string getData() = 0;

// get data as bytes, return size
virtual int getData(uchar &buf[]) = 0;

// return frame type (opcode)
virtual WS_FRAME_OPCODE getType() = 0;

// check if the frame is a control frame or with data:
// control frames are processed inside classes
virtual bool isControlFrame()
{
    return (getType() >= WS_CLOSE_FRAME);
}

virtual bool isReady() { return true; }
virtual bool isFinal() { return true; }
virtual bool isMasked() { return false; }
virtual bool isCompressed() { return false; }
};

```

IWebSocketMessage 接口包含用于执行类似操作的方法，但是是在消息层级。

```

class IWebSocketMessage
{
public:
    // get an array of frames that make up this message
    virtual void getFrames(IWebSocketFrame *&frames[]) = 0;

    // set text as message content
    virtual bool setString(const string &data) = 0;

    // return message content as text
    virtual string getString() = 0;

    // set binary data as message content
    virtual bool setData(const uchar &data[]) = 0;

    // return the contents of the message in "raw" binary form
    virtual bool getData(uchar &data[]) = 0;

    // sign of completeness of the message (all frames received)
    virtual bool isFinalised() = 0;

    // add a frame to the message
    virtual bool takeFrame(IWebSocketFrame *frame) = 0;
};

```

考虑到帧和消息的接口，定义了 WebSocket 连接 *IWebSocketConnection*。

```

interface IWebSocketConnection
{
    // open a connection with the specified URL and its parts,
    // and optional custom headers
    bool handshake(const string url, const string host, const string origin,
        const string custom = NULL);

    // low-level read frames from the server
    int readFrame(IWebSocketFrame *&frames[]);

    // low-level send frame (e.g. close or ping)
    bool sendFrame(IWebSocketFrame *frame);

    // low-level message sending
    bool sendMessage(IWebSocketMessage *msg);

    // custom check for new messages (event generation)
    int checkMessages();

    // custom text submission
    bool sendString(const string msg);

    // custom posting of binary data
    bool sendData(const uchar &data[]);

    // close the connection
    bool disconnect(void);
};

```

通过 *IWebSocketObserver* 接口方法接收关于断开连接和新消息的通知。

```

interface IWebSocketObserver
{
    void onConnected();
    void onDisconnect();
    void onMessage(IWebSocketMessage *msg);
};

```

特别是, *WebSocketClient* 类是这个接口的继承者, 默认情况下, 它只是将信息输出到日志。类构造函数需要一个地址来连接到 **ws** 或 **wss** 协议。

```

template<typename T>
class WebSocketClient: public IWebSocketObserver
{
protected:
    IWebSocketMessage *messages[];

    string scheme;
    string host;
    string port;
    string origin;
    string url;
    int timeout;
    ...
public:
    WebSocketClient(const string address)
    {
        string parts[];
        URL::parse(address, parts);

        url = address;
        timeout = 5000;

        scheme = parts[URL_SCHEME];
        if(scheme != "ws" && scheme != "wss")
        {
            Print("WebSocket invalid url scheme: ", scheme);
            scheme = "ws";
        }

        host = parts[URL_HOST];
        port = parts[URL_PORT];

        origin = (scheme == "wss" ? "https://" : "http://") + host;
    }
    ...

    void onDisconnect() override
    {
        Print(" > Disconnected ", url);
    }

    void onConnected() override
    {
        Print(" > Connected ", url);
    }

    void onMessage(IWebSocketMessage *msg) override
    {
        // NB: message can be binary, print it just for notification
        Print(" > Message ", url, " ", msg.getString());
        WsTools::push(messages, msg);
    }
}

```

```

    }
    ...
};

```

WebSocketClient 类将所有消息对象收集到一个数组中并负责删除这些对象（如果 MQL 程序未执行删除）。

连接是在 *open* 方法中创建的。

```

template<typename T>
class WebSocketClient: public IWebSocketObserver
{
protected:
    IWebSocketTransport *socket;
    IWebSocketConnection *connection;
    ...
public:
    ...
    bool open(const string custom_headers = NULL)
    {
        uint _port = (uint)StringToInteger(port);
        if(_port == 0)
        {
            if(scheme == "ws") _port = 80;
            else _port = 443;
        }

        socket = MqlWebSocketTransport::create(scheme, host, _port, timeOut);
        if(!socket || !socket.isConnected())
        {
            return false;
        }

        connection = new T(&this, socket);
        return connection.handshake(url, host, origin, custom_headers);
    }
    ...

```

文本和二进制数据的重载 *send* 方法提供了发送数据的最便捷的方式。

```

bool send(const string str)
{
    return connection ? connection.sendString(str) : false;
}

bool send(const uchar &data[])
{
    return connection ? connection.sendData(data) : false;
}

```

若要检查新的传入消息，可以调用 *checkMessages* 方法。根据其 *blocking* 参数，该方法在循环中等待消息直至超时，或者在没有消息时立即返回。消息将转到 *IWebSocketObserver::onMessage* 处理程序。


```

void checkMessages(const bool blocking = true)
{
    if(connection == NULL) return;

    uint stop = GetTickCount() + (blocking ? timeout : 1);
    while(ArraySize(messages) == 0 && GetTickCount() < stop && isConnected())
    {
        // all frames are collected into the appropriate messages, and they become
        // available through event notifications IWebSocketObserver::onMessage,
        // however, control frames have already been internally processed and remove
        if(!connection.checkMessages()) // while no messages, let's make micro-pause
        {
            Sleep(100);
        }
    }
}

```

另一种接收消息的方法是在 *readMessage* 方法中实现的：它返回一个指向调用代码的消息的指针（也就是说，*onMessage* 应用处理程序不是必要的）。之后，MQL 程序负责释放对象。

```

IWebSocketMessage *readMessage(const bool blocking = true)
{
    if(ArraySize(messages) == 0) checkMessages(blocking);

    if(ArraySize(messages) > 0)
    {
        IWebSocketMessage *top = messages[0];
        ArrayRemove(messages, 0, 1);
        return top;
    }
    return NULL;
}

```

该类还允许你更改超时、检查连接并关闭连接。

```

void setTimeout(const int ms)
{
    timeOut = fabs(ms);
}

bool isConnected() const
{
    return socket && socket.isConnected();
}

void close()
{
    if(isConnected())
    {
        if(connection)
        {
            connection.disconnect(); // this will close socket after server acknowledged
            delete connection;
            connection = NULL;
        }
        if(socket)
        {
            delete socket;
            socket = NULL;
        }
    }
}
};

```

上述类库允许为回显和聊天服务创建客户端应用程序。

7.8.7 MQL5 中回显和聊天服务的客户端程序

我们编写一个简单脚本 *MQL5/Experts/MQL5Book/p7/wsEcho/wsecho.mq5* 来连接到回显服务（注意，这是一个脚本，但是我们将它放在 *MQL5/Experts/MQL5Book/p7/* 文件夹中，使它成为与 Web 相关的 MQL 程序的一个容器，因为所有后续的例子都将是 EA 交易）。因为在这一章中，我们考虑在项目中创建软件复合体，我们将设计脚本作为 *mqlproj* 项目的一部分，它也将包括服务器组件。

脚本的输入参数允许你指定服务的地址和消息的文本。默认采用不安全连接。如果要启动支持 TLS 的 *wsecho.js* 服务器，需要将协议更改为安全的 *wss*。请记住，建立一个安全的连接需要比平时更长的时间（几秒钟）。

```

input string Server = "ws://localhost:9000/";
input string Message = "My outbound message";

#include <MQL5Book/AutoPtr.mqh>
#include <MQL5Book/ws/wsclient.mqh>

```

在 *OnStart* 函数中，我们为给定的地址创建一个 WebSocket 客户端 (*wss*) 实例，并调用 *open* 方法。如果连接成功，我们通过在阻塞模式下调用 *wss.readMessage* 来等待来自服务的欢迎消息

默认情况下，最多等待 5 秒)。我们在结果对象上使用自动指针，这样就不会在最后手动调用 *delete*。

```
void OnStart()
{
    Print("\n");
    WebSocketClient<Hybi> wss(Server);
    Print("Opening...");
    if(wss.open())
    {
        Print("Waiting for welcome message (if any)");
        AutoPtr<IWebSocketMessage> welcome(wss.readMessage());
        ...
    }
}
```

WebSocketClient 类包含事件处理程序存根，包括 *onMessage* 简单方法，该方法将问候语打印到日志中。

然后我们发送消息，再次等待服务器的响应。还将记录回显消息。

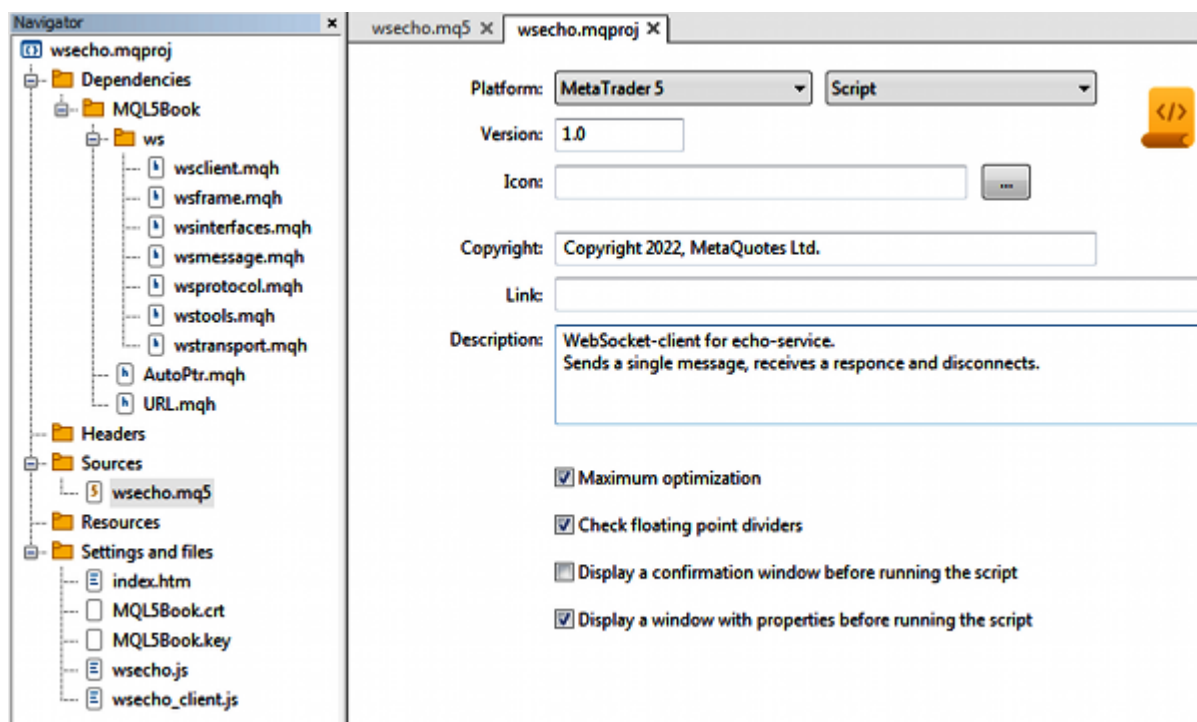
```
Print("Sending message...");
wss.send(Message);
Print("Receiving echo...");
AutoPtr<IWebSocketMessage> echo(wss.readMessage());
}
...
```

最后，我们关闭连接。

```
if(wss.isConnected())
{
    Print("Closing...");
    wss.close();
}
}
```

基于这个脚本文件，让我们创建一个项目文件 (*wsecho.mqproj*)。我们用版本号 (1.0)、版权信息和描述来填充项目特性。我们将回显服务服务器文件添加到 *Settings and Files* 分支中 (这至少会提醒开发人员存在测试服务器)。编译后，依赖项 (头文件)将出现在层次结构中。

一切应该看起来像下面的截图。



回显服务项目、客户端脚本和服务端

如果脚本位于 *Shared Projects* 文件夹中，例如，在 *MQL5/Shared Projects/MQL5Book/wsEcho/* 中，那么在成功编译之后，它的 `ex5` 文件将自动移动到 *MQL5/Scripts/Shared Projects/MQL5Book/wsEcho/* 文件夹中，并且相应的条目将显示在编译日志中。这是在共享项目中编译任何 MQL 程序的标准行为。

在本章的所有示例中，不要忘记在测试 MQL 脚本之前启动服务器。在这种情况下，运行以下命令：在 *web* 文件夹中的 `node.exe wsecho.js`。

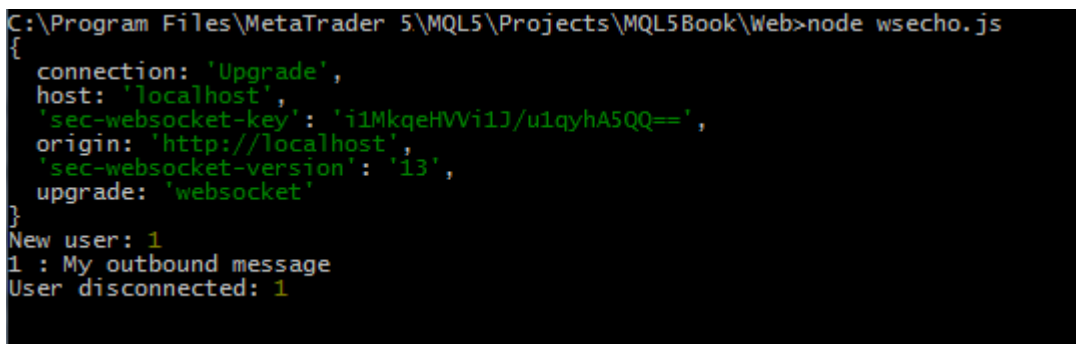
然后，运行 `wsecho.ex5` 脚本。日志将显示正在发生的操作，以及消息通知。

```

Opening...
Connecting to localhost:9000
Buffer: 'HTTP/1.1 101 Switching Protocols'
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: mIpas63g5xGMqJcKtreHKpSbY1w=
,
Headers:
                                [,0]                                [,1]
[0,] "upgrade"                  "websocket"
[1,] "connection"              "Upgrade"
[2,] "sec-websocket-accept"    "mIpas63g5xGMqJcKtreHKpSbY1w="
> Connected ws://localhost:9000/
Waiting for welcome message (if any)
> Message ws://localhost:9000/ server#Hello, user1
Sending message...
Receiving echo...
> Message ws://localhost:9000/ user1#My outbound message
Closing...
Close requested
Waiting...
SocketRead failed: 5273 Available: 1
> Disconnected ws://localhost:9000/
Server close ack

```

以上 HTTP 标头是握手过程中服务器的响应。如果我们查看服务器运行的控制台窗口，我们会发现服务器从我们的客户机接收到的 HTTP 标头。



```

C:\Program Files\MetaTrader 5\QL5\Projects\QL5Book\Web>node wsecho.js
{
  connection: 'Upgrade',
  host: 'localhost',
  'sec-websocket-key': 'i1MkqeHVVi1J/u1qyhA5QQ==',
  origin: 'http://localhost',
  'sec-websocket-version': '13',
  upgrade: 'websocket'
}
New user: 1
1 : My outbound message
User disconnected: 1

```

回显服务服务器日志

此外，这里还显示了用户的连接、消息和断开。

让我们为聊天服务做一个类似的工作：在 MQL5 中创建一个 WebSocket 客户机，为它创建一个项目，并进行测试。这一次，客户端程序的类型将是 EA 交易，因为聊天需要图表上的键盘对交互事件的支持。EA 交易附在本书的 *MQL5/QL5Book/p7/wsChat/wschat.mq5* 文件夹中。

为了演示在处理程序方法中接收事件的技术，让我们定义我们自己的类 *MyWebSocket*，该类派生自 *WebSocketClient*。

```

class MyWebSocket: public WebSocketClient<Hybi>
{
public:
    MyWebSocket(const string address, const bool compress = false):
        WebSocketClient(address, compress) { }

    /* void onConnected() override { } */

    void onDisconnect() override
    {
        // we can do something else and call (or not call) the legacy code
        WebSocketClient<Hybi>::onDisconnect();
    }

    void onMessage(IWebSocketMessage *msg) override
    {
        // TODO: we could truncate copies of our own messages,
        // but they are left for debugging
        Alert(msg.getString());
        delete msg;
    }
};

```

收到一条消息时，我们不会在日志中显示，而是作为一个警报显示，在此之后应该删除该对象。

在全局上下文中，我们说明了 `wss` 类的对象，而用户从键盘输入的 `message` 字符串将在其中累积。

```

MyWebSocket wss(Server);
string message = "";

```

`OnInit` 函数包含了必要的准备，特别是启动定时器和打开连接的准备。

```

int OnInit()
{
    ChartSetInteger(0, CHART_QUICK_NAVIGATION, false);
    EventSetTimer(1);
    wss.setTimeout(1000);
    Print("Opening...");
    return wss.open() ? INIT_SUCCEEDED : INIT_FAILED;
}

```

需要计时器来检查来自其他用户的新消息。

```

void OnTimer()
{
    wss.checkMessages(false); // use a non-blocking check in the timer
}

```

在 `OnChartEvent` 处理程序中，我们响应击键：所有字母数字键都被转换成字符并附加到 `message` 字符串。如有必要，可以按 `Backspace` 删除最后一个字符。所有键入的文本都会在图表注释中更新。消息完成后，按 `Enter` 键将其发送到服务器。

```

void OnChartEvent(const int id, const long &lparam, const double &dparam,
    const string &sparam)
{
    if(id == CHARTEVENT_KEYDOWN)
    {
        if(lparam == VK_RETURN)
        {
            const static string longmessage = ...
            if(message == "long") wss.send(longmessage);
            else if(message == "bye") wss.close();
            else wss.send(message);
            message = "";
        }
        else if(lparam == VK_BACK)
        {
            StringSetLength(message, StringLen(message) - 1);
        }
        else
        {
            ResetLastError();
            const short c = TranslateKey((int)lparam);
            if(_LastError == 0)
            {
                message += ShortToString(c);
            }
        }
        Comment(message);
    }
}

```

如果我们输入文本“long”，程序将发送一个特别准备的相当长的文本。如果消息文本是“bye”，程序将关闭连接。此外，当程序退出时，连接将被关闭。

```

void OnDeinit(const int)
{
    if(wss.isConnected())
    {
        Print("Closing...");
        wss.close();
    }
}

```

让我们为 EA 交易创建一个项目 (*wschat.mqproj* 文件)，填充它的特性，并将后端添加到分支 **Settings and Files**。这次我们将从内部展示项目文件。在 *mqproj* 文件中，**Dependencies** 分支存储在“files”特性中，而 **Settings and Files** 分支存储在“tester”特性中。

```

{
  "platform" : "mt5",
  "program_type" : "expert",
  "copyright" : "Copyright 2022, MetaQuotes Ltd.",
  "version" : "1.0",
  "description" : "WebSocket-client for chat-service.\r\nType and send text messages for
  "optimize" : "1",
  "fpzerocheck" : "1",
  "tester_no_cache" : "0",
  "tester_everytick_calculate" : "0",
  "unicode_character_set" : "0",
  "static_libraries" : "0",
  "files":
  [
    {
      "path": "wschat.mq5",
      "compile": true,
      "relative_to_project": true
    },
    {
      "path": "MQL5\\Include\\MQL5Book\\ws\\wsclient.mqh",
      "compile": false,
      "relative_to_project": false
    },
    {
      "path": "MQL5\\Include\\MQL5Book\\URL.mqh",
      "compile": false,
      "relative_to_project": false
    },
    {
      "path": "MQL5\\Include\\MQL5Book\\ws\\wsframe.mqh",
      "compile": false,
      "relative_to_project": false
    },
    {
      "path": "MQL5\\Include\\MQL5Book\\ws\\wstools.mqh",
      "compile": false,
      "relative_to_project": false
    },
    {
      "path": "MQL5\\Include\\MQL5Book\\ws\\wsinterfaces.mqh",
      "compile": false,
      "relative_to_project": false
    },
    {
      "path": "MQL5\\Include\\MQL5Book\\ws\\wsmessage.mqh",
      "compile": false,
      "relative_to_project": false
    },
    {
      "path": "MQL5\\Include\\MQL5Book\\ws\\wstransport.mqh",
      "compile": false,
      "relative_to_project": false
    },
  ],

```



```

{
  "path":"MQL5\\Include\\MQL5Book\\ws\\wsprotocol.mqh",
  "compile":false,
  "relative_to_project":false
},
{
  "path":"MQL5\\Include\\VirtualKeys.mqh",
  "compile":false,
  "relative_to_project":false
}
],
"tester":
[
  {
    "type":"file",
    "path":"..\\Web\\MQL5Book.crt",
    "relative_to_project":true
  },
  {
    "type":"file",
    "path":"..\\Web\\MQL5Book.key",
    "relative_to_project":true
  },
  {
    "type":"file",
    "path":"..\\Web\\wschat.htm",
    "relative_to_project":true
  },
  {
    "type":"file",
    "path":"..\\Web\\wschat.js",
    "relative_to_project":true
  },
  {
    "type":"file",
    "path":"..\\Web\\wschat_client.js",
    "relative_to_project":true
  }
]
}

```

如果 EA 交易在 *Shared Projects* 文件夹内部，例如，在 *MQL5/Shared Projects/MQL5Book/wsChat/* 中，在成功编译之后，它的 `ex5` 文件将自动移动到 *MQL5/Experts/Shared Projects/MQL5Book/wsChat/* 文件夹。

启动 `node.exe wschat.js` 服务器。现在，你可以在不同的图表上运行 EA 交易的若干个副本。基本上，该服务涉及不同终端甚至不同计算机之间的“通信”，但你也可以从一个终端进行测试。

这是一个 EURUSD 和 GBPUSD 图表进行通信的例子。

```

(EURUSD,H1)
(EURUSD,H1) Opening...
(EURUSD,H1) Connecting to localhost:9000
(EURUSD,H1) Buffer: 'HTTP/1.1 101 Switching Protocols
(EURUSD,H1) Upgrade: websocket
(EURUSD,H1) Connection: Upgrade
(EURUSD,H1) Sec-WebSocket-Accept: Dg+aQdCBwNExE5mEQsfk5w9J+uE=
(EURUSD,H1)
(EURUSD,H1) '
(EURUSD,H1) Headers:
(EURUSD,H1)                                     [,0]                                     [,1]
(EURUSD,H1) [0,] "upgrade"                                     "websocket"
(EURUSD,H1) [1,] "connection"                                     "Upgrade"
(EURUSD,H1) [2,] "sec-websocket-accept"                       "Dg+aQdCBwNExE5mEQsfk5w9J+uE="
(EURUSD,H1) > Connected ws://localhost:9000/
(EURUSD,H1) Alert: server#Hello, user1
(GBPUSD,H1)
(GBPUSD,H1) Opening...
(GBPUSD,H1) Connecting to localhost:9000
(GBPUSD,H1) Buffer: 'HTTP/1.1 101 Switching Protocols
(GBPUSD,H1) Upgrade: websocket
(GBPUSD,H1) Connection: Upgrade
(GBPUSD,H1) Sec-WebSocket-Accept: NZENnc8p05T4amvngeop/e/+gFw=
(GBPUSD,H1)
(GBPUSD,H1) '
(GBPUSD,H1) Headers:
(GBPUSD,H1)                                     [,0]                                     [,1]
(GBPUSD,H1) [0,] "upgrade"                                     "websocket"
(GBPUSD,H1) [1,] "connection"                                     "Upgrade"
(GBPUSD,H1) [2,] "sec-websocket-accept"                       "NZENnc8p05T4amvngeop/e/+gFw="
(GBPUSD,H1) > Connected ws://localhost:9000/
(GBPUSD,H1) Alert: server#Hello, user2
(EURUSD,H1) Alert: user1#I'm typing this on EURUSD chart
(GBPUSD,H1) Alert: user1#I'm typing this on EURUSD chart
(GBPUSD,H1) Alert: user2#Got it on GBPUSD chart!
(EURUSD,H1) Alert: user2#Got it on GBPUSD chart!

```

因为我们的消息发送给每个人，包括发件人，所以它们在日志中是重复的，但是在不同的图表上。

通信在服务器端也是可见的。

```

C:\Program Files\MetaTrader 5\QL5\Projects\QL5Book\Web>node wschat.js
{
  connection: 'Upgrade',
  host: 'localhost',
  'sec-websocket-key': 'ZBiifDgZZxrYB3HDz5S6Vg==',
  origin: 'http://localhost',
  'sec-websocket-version': '13',
  upgrade: 'websocket'
}
New user: 1
{
  connection: 'Upgrade',
  host: 'localhost',
  'sec-websocket-key': 'a2UZhk69Ulw1Xb1UtIqCw==',
  origin: 'http://localhost',
  'sec-websocket-version': '13',
  upgrade: 'websocket'
}
New user: 2
1 : I'm typing this on EURUSD chart
2 : Got it on GBPU$D chart!

```

聊天服务服务器日志

现在我们有组织交易信号服务的所有技术组件。

7.8.8 交易信号服务和测试 Web 页

交易信号服务在技术上与聊天服务相同，但是它的用户（或者说是客户连接）必须扮演两个角色之一：

- 消息提供者
- 消息消费者

此外，信息不应该对每个人都可用，而是根据一些订阅方案运作。

为了确保这一点，当连接到服务时，用户将被要求提供特定的标识信息，这些信息因角色而异。

提供者必须指定一个在所有信号中唯一的公共信号标识符（PUB_ID）。基本上，同一个人可能会产生不止一个信号，因此应该能够获得多个识别标志。在这个意义上，我们不会通过引入单独的提供者标识符（作为特定的人）和其信号的标识符来使服务复杂化。相反，只支持信号标识符。对于一个真正的信号服务，这个问题需要和授权一起解决，这一点我们在本书之外已经讨论过了。

该标识符将用于发布该信号，或仅需将其传递给有意订阅此信号的人员然而，不应仅凭公开标识符就让任何获知者都具备访问该信号的能力。在最简单的情况下，开账户监控可采用此方式，但我们将专门演示信号场景中的访问限制方案。

为此，提供者必须向服务器提供一个只有他们自己知道而公众不知道的秘密密钥（PUB_KEY）。生成特定订户的访问密钥将需要该密钥。

消费者（订阅者）还必须有一个唯一标识符（SUB_ID，这里我们也是省略了授权）。用户订阅需要的信号时，须向信号提供者提交标识符（实际操作中需同步完成确认付款，该流程通常由服务器自动化处理）提供者形成一个快照，其中包含提供者的标识符、订户的标识符和秘密密钥。在我们的服务中，这将通过从 PUB_ID:PUB_KEY:SUB_ID 字符串计算 SHA256 散列来完成，之后，结果字节被转换为十六进制格式的字符串。这将是特定用户的特定提供商的信号的访问密钥（子密钥或访问密钥）。提供者（在真实系统中，由服务器本身自动完成）将这个密钥转发给订户。

因此，当连接到服务时，订户将必须指定订户标识符 (SUB_ID)、期望信号的标识符 (PUB_ID)和访问密钥 (SUB_KEY)。因为服务器知道提供者的秘密密钥，所以它可以为给定的 PUB_ID 和 SUB_ID 的组合重新计算访问密钥，并将其与提供的 SUB_KEY 进行比较。匹配意味着正常的消息传递过程继续。这种差异将导致错误消息，并断开伪订户与服务的连接。

值得注意的是，在我们的演示中，为了简单起见，不存在正常注册的用户和信号，因此标识符的选择是任意的。对我们来说，重要的是跟踪标识符的唯一性，以便知道在线发送信息的发送方和接收方。因此，我们的服务不保证标识符（例如，“超级趋势”）昨天、今天和明天属于同一个用户。根据“先到先得”原则来预留名称。只要提供商用给定的标识符持续连接，信号就被传递。如果提供者断开连接，那么标识符在任何下一个连接中变得可供选择。

唯一会一直忙碌的标识符是“Server”：服务器用它来发送它的连接状态消息。

为了在服务器文件夹中生成访问密钥，使用了简单的 `JavaScript``access.js`。在命令行上运行该 `JavaScript` 文件时，需要将上面类型的 PUB_ID:PUB_KEY:SUB_ID (标识符和它们之间的密钥，由 ':' 符号连接)作为唯一的参数传递

如果未指定该参数，脚本将为一些演示标识符 (PUB_ID_001, SUB_ID_100) 和一个秘密 (PUB_KEY_FFF) 生成一个访问密钥。

```
// JavaScript
const args = process.argv.slice(2);
const input = args.length > 0 ? args[0] : 'PUB_ID_001:PUB_KEY_FFF:SUB_ID_100';
console.log('Hashing "', input, '"');
const crypto = require('crypto');
console.log(crypto.createHash('sha256').update(input).digest('hex'));
```

使用以下命令运行脚本：

```
node access.js PUB_ID_001:PUB_KEY_FFF:SUB_ID_100
```

我们得到这样的结果：

```
fd3f7a105eae8c2d9afce0a7a4e11bf267a40f04b7c216dd01cf78c7165a2a5a
```

顺便说一下，你可以使用 `CryptEncode` 函数在纯 MQL5 环境中检查和重复该算法。

分析了概念部分之后，让我们继续实际的实现。

信号服务的服务器脚本将放在 `MQL5/Experts/MQL5Book/p7/Web/wspubsub.js` 文件中。在其中设置服务器与我们之前所做的一样。但是，除此之外，你还需要连接 `access.js` 中使用的相同的“crypto”模块。主页被称为 `wspubsub.htm`。

```
// JavaScript
const crypto = require('crypto');
...
http1.createServer(options, function (req, res)
{
    ...
    if(req.url == '/')
    {
        req.url = "wspubsub.htm";
    }
    ...
});
```

我们将为信号提供商和消费者分别定义两个映射，而不是一个连接客户端的映射。

```
// JavaScript
const publishers = new Map();
const subscribers = new Map();
```

在这两个映射中，键均为提供者 ID，但是第一个存储提供者的对象，第二个存储订阅每个提供者的订阅者的对象（对象数组）。

为了在握手过程中传输标识符和密钥，我们将使用 WebSocket 规范允许的特殊标头，即 Sec-WebSocket-Protocol。我们约定：用符号 ‘-’ 将标识符和键拼接：提供者应生成类似 X-MQL5-publisher-PUB_ID-PUB_KEY 的字符串，订阅者应生成类似 X-MQL5-subscriber-SUB_ID-PUB_ID-SUB_KEY 的字符串。

如果在没有 Sec-WebSocket-Protocol-X-MQL5-... 标头的情况下尝试连接到我们服务，则会立即停止并关闭。

在新的客户端对象中（在 “connection” 事件处理程序参数 *onConnect(client)* 中），这个标题很容易从 *client.protocol* 特性中提取。

让我们以简化形式展示注册和发送信号提供者消息的过程，省略了错误处理（附上完整代码）。值得注意的是，消息文本是以 JSON 格式生成的（我们将在下一节详细讨论）。特别是，消息的发送者在 “origin” 特性中传递（此外，当消息由服务本身发送时，该字段包含字符串 “Server”），来自提供者的应用程序数据放在 “msg” 特性中，这可能不仅仅是文本，还可以是任何内容的嵌套结构。

```
// JavaScript
const wsServer = new WebSocket.Server({ server });
wsServer.on('connection', function onConnect(client)
{
  console.log('New user:', ++count, client.protocol);
  if(client.protocol.startsWith('X-MQL5-publisher'))
  {
    const parts = client.protocol.split('-');
    client.id = parts[3];
    client.key = parts[4];
    publishers.set(client.id, client);
    client.send('{"origin":"Server", "msg":"Hello, publisher ' + client.id + '"}');
    client.on('message', function(message)
    {
      console.log('%s : %s', client.id, message);

      if(subscribers.get(client.id))
        subscribers.get(client.id).forEach(function(elem)
        {
          elem.send('{"origin":"publisher ' + client.id + '", "msg":"'
            + message + '"}');
        });
    });
    client.on('close', function()
    {
      console.log('Publisher disconnected:', client.id);
      if(subscribers.get(client.id))
        subscribers.get(client.id).forEach(function(elem)
        {
          elem.close();
        });
      publishers.delete(client.id);
    });
  }
  ...
}
```

订阅者的算法前半部分与此类似，但新增了访问密钥的计算环节，以及它与连接客户端传输的内容的比较。

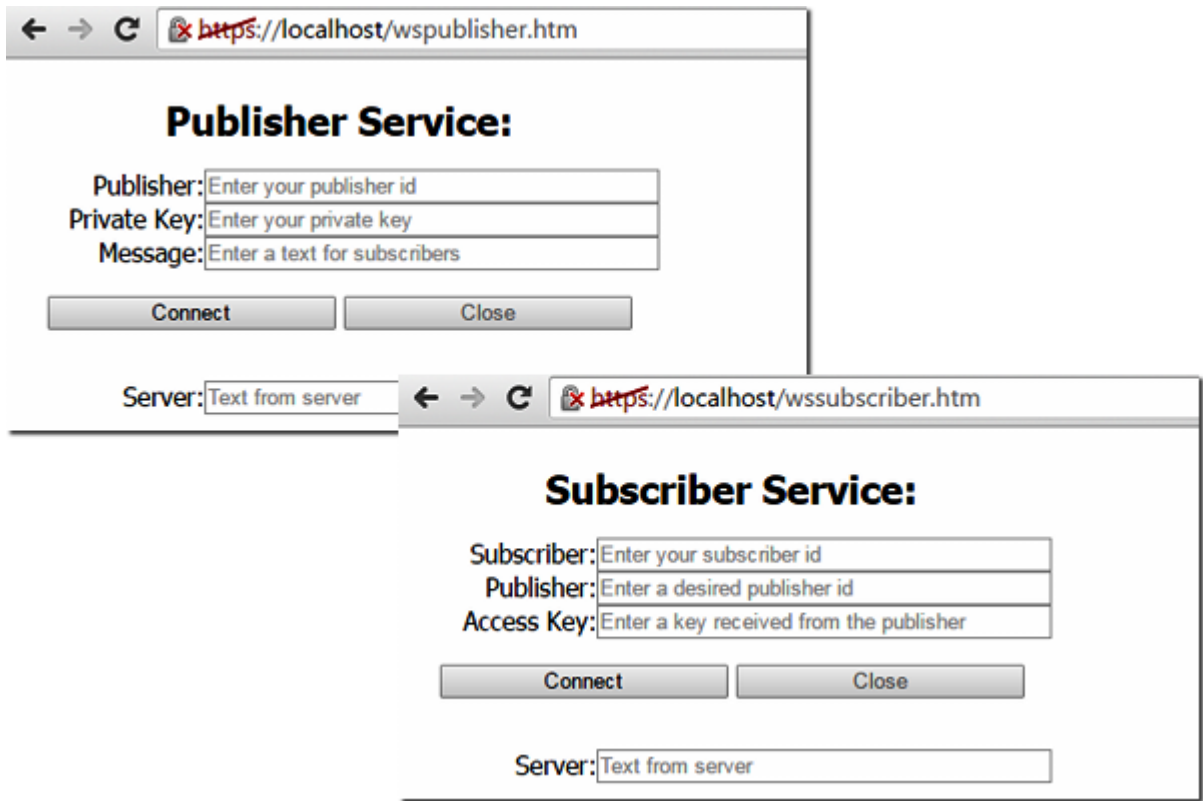
```
// JavaScript
else if(client.protocol.startsWith('X-MQL5-subscriber'))
{
    const parts = client.protocol.split('-');
    client.id = parts[3];
    client.pub_id = parts[4];
    client.access = parts[5];
    const id = client.pub_id;
    var p = publishers.get(id);
    if(p)
    {
        const check = crypto.createHash('sha256').update(id + ':' + p.key + ':' +
            + client.id).digest('hex');
        if(check !== client.access)
        {
            console.log(`Bad credentials: '${client.access}' vs '${check}'`);
            client.send({'origin':"Server", "msg":"Bad credentials, subscriber "
                + client.id + '"'});
            client.close();
            return;
        }

        var list = subscribers.get(id);
        if(list == undefined)
        {
            list = [];
        }
        list.push(client);
        subscribers.set(id, list);
        client.send({'origin':"Server", "msg":"Hello, subscriber "
            + client.id + '"'});
        p.send({'origin':"Server", "msg":"New subscriber " + client.id + '"'});
    }

    client.on('close', function()
    {
        console.log('Subscriber disconnected:', client.id);
        const list = subscribers.get(client.pub_id);
        if(list)
        {
            if(list.length > 1)
            {
                const filtered = list.filter(function(el) { return el !== client; });
                subscribers.set(client.pub_id, filtered);
            }
            else
            {
                subscribers.delete(client.pub_id);
            }
        }
    });
});
```

}

wspubsub.htm 客户端页面上的用户界面邀请你点击一个链接，该链接指向两个页面中的一个，这两个页面都包含了针对提供者 (*wspublisher.htm* + *wspublisher_client.js*) 或订阅者 (*wssubscriber.htm* + *wssubscriber_client.js*) 的表单。



信号服务测试客户端 Web 页面

它们的实现继承了之前考虑的 JavaScript 客户端的特性，但是在 Sec-WebSocket-Protocol-MQL5-标头定制方面还有一处细节优化。

一直到现在，我们都是交换简单的短信。但是对于信令服务，你需要传输大量的结构化信息，而 JSON 更适合这一点。因此，客户端可以解析 JSON，尽管它们并没有将它用于预期的目的，因为即使在 JSON 中找到了买卖给定数量的特定股票的命令，浏览器也不知道如何处理。

我们需要在 MQL5 中为我们的信号服务客户端添加 JSON 支持。同时，你可以在 *wspubsub.js* 服务器上运行，并根据信号提供者和使用者的指定细节测试它们的选择性连接。出于你自己利益考虑，我们建议你自行完成。

7.8.9 MQL5 中的信号服务客户端程序

因此我们决定，服务消息中的文本将采用 JSON 格式。

在最常见的版本中，JSON 是对象的文本说明，类似于 MQL5 中对结构的说明。对象带有花括号，在花括号内，它的特性用逗号分隔：每个特性都有一个带引号的标识符，后跟一个冒号和特性值。此处支持几种基本类型的特性：字符串、整数和实数、布尔值 *true/false* 和空值 *null*。此外，特性值可以是对象或数组。数组用方括号说明，方括号中的元素用逗号分隔。例如，


```
{
  "string": "this is a text",
  "number": 0.1,
  "integer": 789735095,
  "enabled": true,
  "subobject" :
  {
    "option": null
  },
  "array":
  [
    1, 2, 3, 5, 8
  ]
}
```

顶层的数组基本上都是有效的 JSON。例如，

```
[
  {
    "command": "buy",
    "volume": 0.1,
    "symbol": "EURUSD",
    "price": 1.0
  },
  {
    "command": "sell",
    "volume": 0.01,
    "symbol": "GBPUSD",
    "price": 1.5
  }
]
```

为了减少使用 JSON 的应用程序协议中的流量，通常将字段名缩写成几个字母（通常是一个字母）。

特性名称和字符串值带有双引号。如果要在字符串中指定引号，必须用反斜杠对其进行转义。

JSON 的使用使得该协议具有通用性和可扩展性。例如，对于正在设计的服务（交易信号，以及在更一般的情况下，账户状态复制），可以假设消息结构如下：

```

{
  "origin": "publisher_id",    // message sender ("Server" in technical message)
  "msg" :                      // message (text or JSON) as received from the sender
  {
    "trade" :                  // current trading commands (if there is a signal)
    {
      "operation": ...,        // buy/sell/close
      "symbol": "ticker",
      "volume": 0.1,
      ... // other signal parameters
    },
    "account":                 // account status
    {
      "positions":             // positions
      {
        "n": 10,               // number of open positions
        [ { ... }, { ... } ] // array of properties of open positions
      },
      "pending_orders":        // pending orders
      {
        "n": ...
        [ { ... } ]
      }
      "drawdown": 2.56,
      "margin_level": 12345,
      ... // other status parameters
    },
    "hardware":                 // remote control of the "health" of the PC
    {
      "memory": ...,
      "ping_to_broker": ...
    }
  }
}

```

其中某些特征可能支持也可能不支持客户端程序的特定实现（它们会忽略不“理解”的任何内容）。此外，在同级的特性名称没有冲突的情况下，每个信息提供者都可以向 JSON 添加自己的特定数据。消息服务将转发该信息。当然，接收端的程序必须能够解析这些特定的数据。

本书还介绍了一个名为 *ToyJson* 的 JSON 解析器（“toy” JSON，*toyjson.mqh* 文件），它很小，效率很低，并且不支持格式规范的全部功能（例如，在处理 *escape* 序列方面）。它是专门为这个演示服务编写的，针对预期的、不太复杂的交易信号信息结构进行了调整。这里不再赘述，我们从信号服务的 MQL 客户端的源代码中就能看出它的使用原理。

为你的项目和此项目的进一步开发考虑，你可以选择 mql5.com 网站上代码库中可用的其他 JSON 解析器。

每个 *ToyJson* 的一个元素（容器或特性）由 *JsValue* 类对象说明。定义了 *put(key, value)* 方法的几个重载，可用于在 JSON 对象或 *put(value)* 中添加命名内部特性，以在 JSON 数组中添加值。该对象还可以表示基本类型的单个值。要读取 JSON 对象的属性，可以对 *JsValue* 应用运算符的表示法 `[]`，后跟带圆括号的特性名。显然支持通过整数索引来访问 JSON 数组内部。

形成相关对象 *JsValue* 的所需配置后，可以使用 *stringify(string&buffer)* 方法将其序列化为 JSON 文本。

toyjson.mqh 中的第二个类 - *JsParser* - 允许你执行相反的操作：将带有 JSON 说明的文本转换成 *JsValue* 对象的层次结构。

考虑到使用 JSON 的类，让我们编写一个 EA 交易 *MQL5/Experts/MQL5Book/p7/wsTradeCopier/wstradecopier.mq5*，它将能够在事务复制服务中扮演两个角色：提供有关账户交易的信息，或者从服务接收这些信息以复制此类交易。

从政治角度来看，发送信息的数量和内容由提供商决定，并且可能会因使用服务的场景（目的）而有很大不同。特别是，可以只复制正在进行的交易或整个账户余额以及挂单和保护级别。在本例中，我们将只指出信息传输的技术实现，然后你可以自行选择一组特定的对象和特性。

在代码中，我们将描述从内置结构继承的 3 种结构体，它们用 JSON 进行信息“打包”：

- *MqlTradeRequestWeb* - *MqlTradeRequest*
- *MqlTradeResultWeb* - *MqlTradeResult*
- *DealMonitorWeb* - *DealMonitor**

严格地说，列表中的最后一个结构体不是内置的，而是由我们在 *DealMonitor.mqh* 文件中定义的，但它是在交易特性的标准集中填写的。

每个派生结构的构造函数根据传输的主要来源（交易请求、其结果或成交）填充字段。每个结构都实现了 *asJsValue* 方法，该方法返回一个指向 *JsValue* 对象的指针，该对象反映了结构的所有特性：使用 *JsValue::put* 方法将这些特性添加到 JSON 对象中。例如，下面是 *MqlTradeRequest* 的使用情况：

```

struct MqlTradeRequestWeb: public MqlTradeRequest
{
    MqlTradeRequestWeb(const MqlTradeRequest &r)
    {
        ZeroMemory(this);
        action = r.action;
        magic = r.magic;
        order = r.order;
        symbol = r.symbol;
        volume = r.volume;
        price = r.price;
        stoplimit = r.stoplimit;
        sl = r.sl;
        tp = r.tp;
        type = r.type;
        type_filling = r.type_filling;
        type_time = r.type_time;
        expiration = r.expiration;
        comment = r.comment;
        position = r.position;
        position_by = r.position_by;
    }

    JsValue *asJsValue() const
    {
        JsValue *req = new JsValue();
        // main block: action, symbol, type
        req.put("a", VerboseJson ? EnumToString(action) : (string)action);
        if(StringLen(symbol) != 0) req.put("s", symbol);
        req.put("t", VerboseJson ? EnumToString(type) : (string)type);

        // volumes
        if(volume != 0) req.put("v", TU::StringOf(volume));
        req.put("f", VerboseJson ? EnumToString(type_filling) : (string)type_filling);

        // block with prices
        if(price != 0) req.put("p", TU::StringOf(price));
        if(stoplimit != 0) req.put("x", TU::StringOf(stoplimit));
        if(sl != 0) req.put("sl", TU::StringOf(sl));
        if(tp != 0) req.put("tp", TU::StringOf(tp));

        // block of pending orders
        if(TU::IsPendingType(type))
        {
            req.put("t", VerboseJson ? EnumToString(type_time) : (string)type_time);
            if(expiration != 0) req.put("d", TimeToString(expiration));
        }

        // modification block
        if(order != 0) req.put("o", order);
        if(position != 0) req.put("q", position);
    }
}

```

```

    if(position_by != 0) req.put("b", position_by);

    // helper block
    if(magic != 0) req.put("m", magic);
    if(StringLen(comment)) req.put("c", comment);

    return req;
}
};

```

我们将所有特性转移到 JSON (这适用于账户监控服务), 但是你可以保留有限的一组特性。

对于枚举的特性, 我们提供了两种方法用 JSON 将它们表示为整数和枚举元素的字符串名称。方法的选择是使用 **VerboseJson** 输入参数 (理想情况下, 它不应该直接写在结构代码中, 而是通过构造函数参数编写)。

```
input bool VerboseJson = false;
```

只传递数字可以简化编码, 因为在接收端, 为了执行“镜像”操作, 将数字转换为所需的枚举类型就足够了。但是人们难以通过数字来感知信息, 可能还需要分析具体情况 (消息)。因此, 支持字符串表示的方法是有意义的, 因为它更“友好”, 尽管这需要对接收算法进行额外操作。

输入参数还分别为提供者和订阅者指定服务器地址、应用程序角色和连接细节。

```

enum TRADE_ROLE
{
    TRADE_PUBLISHER, // Trade Publisher
    TRADE_SUBSCRIBER // Trade Subscriber
};

input string Server = "ws://localhost:9000/";
input TRADE_ROLE Role = TRADE_PUBLISHER;
input bool VerboseJson = false;
input group "Publisher";
input string PublisherID = "PUB_ID_001";
input string PublisherPrivateKey = "PUB_KEY_FFF";
input string SymbolFilter = ""; // SymbolFilter (empty - current, '*' - any)
input ulong MagicFilter = 0; // MagicFilter (0 - any)
input group "Subscriber";
input string SubscriberID = "SUB_ID_100";
input string SubscribeToPublisherID = "PUB_ID_001";
input string SubscriberAccessKey = "fd3f7a105eae8c2d9afce0a7a4e11bf267a40f04b7c216ddc";
input string SymbolSubstitute = "EURUSD=GBPUSD"; // SymbolSubstitute (<from>=<to>,...)
input ulong SubscriberMagic = 0;

```

提供者组中的参数 **SymbolFilter** 和 **MagicFilter** 允许你将受监控的交易活动限制为给定的交易品种和幻数。**SymbolFilter** 中的空值表示只控制图表的当前交易品种, 要拦截任何交易, 请输入 * 符号。为此, 信号提供者将使用 **FilterMatched** 函数, 该函数接受交易的交易品种和幻数。

```

bool FilterMatched(const string s, const ulong m)
{
    if(MagicFilter != 0 && MagicFilter != m)
    {
        return false;
    }

    if(StringLen(SymbolFilter) == 0)
    {
        if(s != _Symbol)
        {
            return false;
        }
    }
    else if(SymbolFilter != s && SymbolFilter != "*")
    {
        return false;
    }

    return true;
}

```

订阅者输入组中的 *SymbolSubstitute* 参数允许用另一个交易品种替换消息中收到的交易品种，该交易品种将用于复制交易。在跨经纪商环境中，该功能尤其有用，可确保同一金融工具的交易代码名称不同。但该参数也执行重复信号许可筛选功能：只有这里指定的交易品种能进行交易。例如，要允许 EURUSD 交易品种的信号交易（即使没有交易代码替换），你需要在参数中设置 "EURUSD=EURUSD" 字符串。信号消息的交易品种显示在 = 交易品种的左边，交易的交易品种显示在右边。

字符替换列表在初始化期间由 *FillSubstitutes* 函数处理，然后由 *FindSubstitute* 函数用来替换和解析交易。

```

string Substitutes[][2];

void FillSubstitutes()
{
    string list[];
    const int n = StringSplit(SymbolSubstitute, ',', list);
    ArrayResize(Substitutes, n);
    for(int i = 0; i < n; ++i)
    {
        string pair[];
        if(StringSplit(list[i], '=', pair) == 2)
        {
            Substitutes[i][0] = pair[0];
            Substitutes[i][1] = pair[1];
        }
        else
        {
            Print("Wrong substitute: ", list[i]);
        }
    }
}

string FindSubstitute(const string s)
{
    for(int i = 0; i < ArrayRange(Substitutes, 0); ++i)
    {
        if(Substitutes[i][0] == s) return Substitutes[i][1];
    }
    return NULL;
}

```

为了与服务通信，我们定义了一个从 *WebSocketClient* 派生的类。首先，当消息到达 *onMessage* 处理程序时，需要根据信号开始交易。在分析了提供者端信号的形成和发送之后，我们稍后会回到这个问题。

```

class MyWebSocket: public WebSocketClient<Hybi>
{
public:
    MyWebSocket(const string address): WebSocketClient(address) { }

    void onMessage(IWebSocketMessage *msg) override
    {
        ...
    }
};

MyWebSocket wss(Server);

```

OnInit 中的初始化可开启计时器（用于定期调用 *wss.checkMessages(false)*）并编写带用户详情的自定义头文件，具体取决于所选的角色。然后我们用 *wss.open(custom)* 调用打开连接。

```

int OnInit()
{
    FillSubstitutes();
    EventSetTimer(1);
    wss.setTimeout(1000);
    Print("Opening...");
    string custom;
    if(Role == TRADE_PUBLISHER)
    {
        custom = "Sec-WebSocket-Protocol: X-MQL5-publisher-"
            + PublisherID + "-" + PublisherPrivateKey + "\r\n";
    }
    else
    {
        custom = "Sec-WebSocket-Protocol: X-MQL5-subscriber-"
            + SubscriberID + "-" + SubscribeToPublisherID
            + "-" + SubscriberAccessKey + "\r\n";
    }
    return wss.open(custom) ? INIT_SUCCEEDED : INIT_FAILED;
}

```

复制机制，即拦截交易并将有关交易的信息发送到 Web 服务，是在 *OnTradeTransaction* 处理程序中启动的。正如我们所知，这不是唯一的方法，在 *OnTrade* 中分析账户状态的“快照”是有可能的。

```

void OnTradeTransaction(const MqlTradeTransaction &transaction,
    const MqlTradeRequest &request,
    const MqlTradeResult &result)
{
    if(transaction.type == TRADE_TRANSACTION_REQUEST)
    {
        Print(TU::StringOf(request));
        Print(TU::StringOf(result));
        if(result.retcode == TRADE_RETCODE_PLACED // successful action
            || result.retcode == TRADE_RETCODE_DONE
            || result.retcode == TRADE_RETCODE_DONE_PARTIAL)
        {
            if(FilterMatched(request.symbol, request.magic))
            {
                ... // see next block of code
            }
        }
    }
}

```

我们跟踪成功完成的交易请求的事件（即满足指定筛选条件）。随后，请求的结构、请求的结果和交易都被转换成 JSON 对象。所有这些都放在一个公共容器 *msg* 中，分别名为“req”、“res”和“deal”。回顾将容器本身作为“msg”特性包含在 web 服务消息中。


```

// container to attach to service message will be visible as "msg" property:
// {"origin" : "this_publisher_id", "msg" : { our data is here }}
JsValue msg;
MqlTradeRequestWeb req(request);
msg.put("req", req.asJsValue());

MqlTradeResultWeb res(result);
msg.put("res", res.asJsValue());

if(result.deal != 0)
{
    DealMonitorWeb deal(result.deal);
    msg.put("deal", deal.asJsValue());
}
ulong tickets[];
Positions.select(tickets);
JsValue pos;
pos.put("n", ArraySize(tickets));
msg.put("pos", &pos);
string buffer;
msg.stringify(buffer);

Print(buffer);

wss.send(buffer);

```

填充后，容器作为字符串输出进入 *buffer*，打印到日志，并发送到服务器。

我们可以向这个容器添加其他信息：账户状态（资金回撤率、杠杆率）、挂单的数量和特性等等。因此，为了展示扩展信息内容的可能性，我们在上面增加了未平仓仓位数量。为了根据筛选器选择仓位，我们使用 *PositionFilter* 类对象 (*PositionFilter.mqh*):

```

PositionFilter Positions;

int OnInit()
{
    ...
    if(MagicFilter) Positions.let(POSITION_MAGIC, MagicFilter);
    if(SymbolFilter == "") Positions.let(POSITION_SYMBOL, _Symbol);
    else if(SymbolFilter != "") Positions.let(POSITION_SYMBOL, SymbolFilter);
    ...
}

```

基本上，为了增加可靠性，对于复制器来说，分析仓位的状态是有意义的，而不仅仅是拦截交易。

这就是对信号提供者角色中涉及的 EA 交易部分的分析总结。

正如我们所声明的，作为订阅者，EA 交易接收 *MyWebSocket::onMessage* 方法中的消息。此处用 *JsParser::jsonify* 解析传入的消息，从 *obj["msg"]* 特性中检索由传输方形成的容器。

```

class MyWebSocket: public WebSocketClient<Hybi>
{
public:
    void onMessage(IWebSocketMessage *msg) override
    {
        Alert(msg.getString());
        JsValue *obj = JsParser::jsonify(msg.getString());
        if(obj && obj["msg"])
        {
            obj["msg"].print();
            if(!RemoteTrade(obj["msg"])) { /* error processing */ }
            delete obj;
        }
        delete msg;
    }
};

```

RemoteTrade 函数实现信号分析和交易操作。这里给出了缩写，省略了潜在错误处理。该函数支持两种表示枚举的方式：表示为整数值或字符串元素名称。通过应用运算符 `[]`，对传入的 JSON 对象进行必要的特性（命令和信号属性）检查，包括连续几次（访问嵌套的 JSON 对象）。

```

bool RemoteTrade(JsValue *obj)
{
    bool success = false;

    if(obj["req"]["a"] == TRADE_ACTION_DEAL
        || obj["req"]["a"] == "TRADE_ACTION_DEAL")
    {
        const string symbol = FindSubstitute(obj["req"]["s"].s);
        if(symbol == NULL)
        {
            Print("Suitable symbol not found for ", obj["req"]["s"].s);
            return false; // not found or forbidden
        }

        JsValue *pType = obj["req"]["t"];
        if(pType == ORDER_TYPE_BUY || pType == ORDER_TYPE_SELL
            || pType == "ORDER_TYPE_BUY" || pType == "ORDER_TYPE_SELL")
        {
            ENUM_ORDER_TYPE type;
            if(pType.detect() >= JS_STRING)
            {
                if(pType == "ORDER_TYPE_BUY") type = ORDER_TYPE_BUY;
                else type = ORDER_TYPE_SELL;
            }
            else
            {
                type = obj["req"]["t"].get<ENUM_ORDER_TYPE>();
            }
        }

        MqlTradeRequestSync request;
        request.deviation = 10;
        request.magic = SubscriberMagic;
        request.type = type;

        const double lot = obj["req"]["v"].get<double>();
        JsValue *pDir = obj["deal"]["entry"];
        if(pDir == DEAL_ENTRY_IN || pDir == "DEAL_ENTRY_IN")
        {
            success = request._market(symbol, lot) && request.completed();
            Alert(StringFormat("Trade by subscription: market entry %s %s %s - %s",
                EnumToString(type), TU::StringOf(lot), symbol,
                success ? "Successful" : "Failed"));
        }
        else if(pDir == DEAL_ENTRY_OUT || pDir == "DEAL_ENTRY_OUT")
        {
            // closing action assumes the presence of a suitable position, look for it
            PositionFilter filter;
            int props[] = {POSITION_TICKET, POSITION_TYPE, POSITION_VOLUME};
            Tuple3<long,long,double> values[];
            filter.let(POSITION_SYMBOL, symbol).let(POSITION_MAGIC,
                SubscriberMagic).select(props, values);
        }
    }
}

```

```

for(int i = 0; i < ArraySize(values); ++i)
{
    // need a position that is opposite in direction to the deal
    if(!TU::IsSameType((ENUM_ORDER_TYPE)values[i]._2, type))
    {
        // you need enough volume (exactly equal here!)
        if(TU::Equal(values[i]._3, lot))
        {
            success = request.close(values[i]._1, lot) && request.completed(
                Alert(StringFormat("Trade by subscription: market exit %s %s %s",
                    EnumToString(type), TU::StringOf(lot), symbol,
                    success ? "Successful" : "Failed")));
        }
    }
}

if(!success)
{
    Print("No suitable position to close");
}
}
}
return success;
}

```

该实现不分析交易价格、对交易的可能限制、止损水平和其他时刻。我们只是以当前的本地价格重复交易。此外，平仓时，会检查交易量是否完全相等，这适用于对冲账户，但不适用于净额结算，如果交易量少于仓位（在冲销的情况下可能更多，但此处不支持 DEAL_ENTRY_INOUT 选项），则可能部分平仓。所有这些要点都应最终落实到实际应用中。

我们在同一个终端的不同图表上启动服务器 *node.exewspubsub.js* 和 EA 交易 *wstradecopier.mq5* 的两个副本。通常的场景假设需要在不同的账户上启动 EA 交易，但是“矛盾的”选项也适合于检查性能：我们将把信号从一个交易品种复制到另一个交易品种。

在 EA 交易的一个副本中，我们将保留默认设置，保留发布者的角色。应放在 EURUSD 图表上。在 GBPUSD 图表上运行的第二个副本中，我们将角色更改为订阅者。输入参数 *SymbolSubstitute* 中的字符串“EURUSD=GBPUSD”允许基于 EURUSD 信号的 GBPUSD 交易。

连接数据将被记录，带有 HTTP 头文件和问候语（我们之前已看到过），这里略去不讲。

我们来买入 EURUSD，并确保它以 GBPUSD 的相同数量被“复制”。

以下是日志的片段（切记，由于在终端的同一副本中使用了两个 EA 交易，交易消息将被发送到两个图表，因此，为了便于分析日志，你可以交替设置筛选“EURUSD”和“USDUSD”）：

```
(EURUSD,H1) TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 0
(EURUSD,H1) DONE, D=1439023682, #=1461313378, V=0.01, @ 0.99886, Bid=0.99886, Ask=0.9
(EURUSD,H1) {"req" : {"a" : "TRADE_ACTION_DEAL", "s" : "EURUSD", "t" : "ORDER_TYPE_BU
» "f" : "ORDER_FILLING_FOK", "p" : 0.99886, "o" : 1461313378}, "res" : {"code" : 1000
» "o" : 1461313378, "v" : 0.01, "p" : 0.99886, "b" : 0.99886, "a" : 0.99886}, "deal"
» "o" : 1461313378, "t" : "2022.09.19 16:45:50", "tmsec" : 1663605950086, "type" : "DE
» "entry" : "DEAL_ENTRY_IN", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" : 0.
» "s" : "EURUSD"}, "pos" : {"n" : 1}}
```

该片段显示了所执行请求的内容和结果，以及向服务器发送一个 JSON 字符串的缓冲区。

在接收端，立即在 GBPUSD 图表上显示一个警报，给出一条来自服务器的“raw”形式的消息，该消息在 *JsParser* 中成功解析后被格式化。“origin”特性以原始形式存储，服务器以这种形式告诉我们信号的源。

```

(GBPUSD,H1) Alert: {"origin":"publisher PUB_ID_001", "msg":{"req" : {"a" : "TRADE_ACT
» "s" : "EURUSD", "t" : "ORDER_TYPE_BUY", "v" : 0.01, "f" : "ORDER_FILLING_FOK", "p"
» "o" : 1461313378}, "res" : {"code" : 10009, "d" : 1439023682, "o" : 1461313378, "v"
» "p" : 0.99886, "b" : 0.99886, "a" : 0.99886}, "deal" : {"d" : 1439023682, "o" : 146
» "t" : "2022.09.19 16:45:50", "tmsec" : 1663605950086, "type" : "DEAL_TYPE_BUY",
» "entry" : "DEAL_ENTRY_IN", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" : 0.
» "p" : 0.99886, "s" : "EURUSD"}}, "pos" : {"n" : 1}}}

(GBPUSD,H1) {
(GBPUSD,H1)   req =
(GBPUSD,H1)   {
(GBPUSD,H1)   a = TRADE_ACTION_DEAL
(GBPUSD,H1)   s = EURUSD
(GBPUSD,H1)   t = ORDER_TYPE_BUY
(GBPUSD,H1)   v = 0.01
(GBPUSD,H1)   f = ORDER_FILLING_FOK
(GBPUSD,H1)   p = 0.99886
(GBPUSD,H1)   o = 1461313378
(GBPUSD,H1)   }
(GBPUSD,H1)   res =
(GBPUSD,H1)   {
(GBPUSD,H1)   code = 10009
(GBPUSD,H1)   d = 1439023682
(GBPUSD,H1)   o = 1461313378
(GBPUSD,H1)   v = 0.01
(GBPUSD,H1)   p = 0.99886
(GBPUSD,H1)   b = 0.99886
(GBPUSD,H1)   a = 0.99886
(GBPUSD,H1)   }
(GBPUSD,H1)   deal =
(GBPUSD,H1)   {
(GBPUSD,H1)   d = 1439023682
(GBPUSD,H1)   o = 1461313378
(GBPUSD,H1)   t = 2022.09.19 16:45:50
(GBPUSD,H1)   tmsec = 1663605950086
(GBPUSD,H1)   type = DEAL_TYPE_BUY
(GBPUSD,H1)   entry = DEAL_ENTRY_IN
(GBPUSD,H1)   pid = 1461313378
(GBPUSD,H1)   r = DEAL_REASON_CLIENT
(GBPUSD,H1)   v = 0.01
(GBPUSD,H1)   p = 0.99886
(GBPUSD,H1)   s = EURUSD
(GBPUSD,H1)   }
(GBPUSD,H1)   pos =
(GBPUSD,H1)   {
(GBPUSD,H1)   n = 1
(GBPUSD,H1)   }
(GBPUSD,H1)   }
(GBPUSD,H1)   警报: 订购交易: 市场买入 ORDER_TYPE_BUY 0.01 GBPUSD - 成功

```

上面最后一次买入表示 GBPUSD 的一次成功交易。在账户的交易选项卡上，应显示 2 个仓位。

一段时间后，我们平仓 EURUSD 仓位，GBPUSD 仓位应自动平仓。

```
(EURUSD,H1) TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_FOK, @
(EURUSD,H1) DONE, D=1439025490, #=1461315206, V=0.01, @ 0.99881, Bid=0.99881, Ask=0.9
(EURUSD,H1) {"req" : {"a" : "TRADE_ACTION_DEAL", "s" : "EURUSD", "t" : "ORDER_TYPE_SE
» "f" : "ORDER_FILLING_FOK", "p" : 0.99881, "o" : 1461315206, "q" : 1461313378}, "res
» "d" : 1439025490, "o" : 1461315206, "v" : 0.01, "p" : 0.99881, "b" : 0.99881, "a" :
» "deal" : {"d" : 1439025490, "o" : 1461315206, "t" : "2022.09.19 16:46:52", "tmsec" :
» "type" : "DEAL_TYPE_SELL", "entry" : "DEAL_ENTRY_OUT", "pid" : 1461313378, "r" : "D
» "v" : 0.01, "p" : 0.99881, "m" : -0.05, "s" : "EURUSD"}, "pos" : {"n" : 0}}
```

如果第一次交易的类型为 DEAL_ENTRY_IN, 现在则为 DEAL_ENTRY_OUT。该警报确认接收消息以及重复仓位被成功平仓。

```
(GBPUSD,H1) Alert: {"origin":"publisher PUB_ID_001", "msg":{"req" : {"a" : "TRADE_ACT
» "s" : "EURUSD", "t" : "ORDER_TYPE_SELL", "v" : 0.01, "f" : "ORDER_FILLING_FOK", "p"
» "o" : 1461315206, "q" : 1461313378}, "res" : {"code" : 10009, "d" : 1439025490, "o"
» "v" : 0.01, "p" : 0.99881, "b" : 0.99881, "a" : 0.99881}, "deal" : {"d" : 143902549
» "o" : 1461315206, "t" : "2022.09.19 16:46:52", "tmsec" : 1663606012990, "type" : "DE
» "entry" : "DEAL_ENTRY_OUT", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" : 0
» "p" : 0.99881, "m" : -0.05, "s" : "EURUSD"}, "pos" : {"n" : 0}}}
...
(GBPUSD,H1) 警报:认购交易:市场卖出 ORDER_TYPE_SELL 0.01 GBPUSD - 成功
```

最后, 在 EA 交易 *wstradecopier.mq5* 旁, 我们创建一个项目文件 *wstradecopier.mqproj*, 向其中添加说明和必要的服务器文件 (在 *MQL5/Experts/p7/MQL5Book/Web/* 旧目录中)。

综上所述: 我们已经组织了一个技术上可扩展的多用户系统, 用于通过套接字服务器交换交易信息。由于 web sockets (永久开放连接) 的技术特性, 信号服务的这种实现更适合于短期和高频交易, 以及控制报价中的套利情况。

解决这个问题需要将不同平台上的多个程序结合起来, 并连接大量依赖项, 这通常象征了到项目级别的过渡。开发环境也得到扩展, 超越了编译器和源代码编辑器的领域。特别是, 项目中的客户端或服务部分, 通常涉及不同程序员的工作, 这些程序员对这些客户端或服务负责。在这种情况下, 将项目放入云端共享以及版本控制变得必不可少。

请注意, 通过 MetaEditor 在 *MQL5/Shared Projects* 文件夹中开发项目时, 来自标准目录 *MQL5/Include* 的头文件不包含在共享存储中。另一方面, 在你的项目中创建一个专用的文件夹 *Include*, 并将必要的标准 *mqh* 文件转移到其中, 这将导致信息的重复和头文件版本的潜在差异。而在 MetaEditor 中进行这种操作, 情况可能会改善。

公共项目的另一特点是需要管理用户并为他们授权。在上一个例子中, 这一点只是被识别出来, 而没有被实施。然而, *mq5.com* 网站提供了一个基于 OAuth 协议 (该协议广为人知) 的现成解决方案。任何拥有 *mq5.com* 账户的个人都可以熟悉 OAuth 的原理, 并为他们的 Web 服务配置该协议: 你只需在配置文件中找到 *Applications* 一节 (类似于 <https://www.mql5.com/zh/users/<login>/apps> 的链接)。通过在 *mq5.com* 应用程序中注册 Web 服务, 你将能够通过 *mq5.com* 网站为用户授权。

7.9 原生 Python 支持

自动化交易的成功很大程度上取决于实现这一想法的技术广度。正如我们在前面几节中已经看到的, *MQL5* 能让你超越交易任务的范围, 能够与外部服务集成 (例如, 基于网络功能和自定义交易品种), 使用关系数据库处理和存储数据, 以及连接任意库。

最后一点允许你与任何以 DLL 格式提供 API 的软件进行交互。一些开发人员使用这种方法来连接工业分布式 DBMS (而非内置的 SQLite)、数学软件包 (如 R 或 MATLAB) 和其他编程语言。

Python 已经成为最流行的编程语言之一。其特点是：核心紧凑，功能由软件包补充，这些软件包是用于构建应用程序解决方案的现成脚本的集合。交易员受益于基本面市场分析（统计计算、数据可视化对象）和交易假设测试（包括机器学习）软件包的广泛选择和功能。

顺应这一趋势，MQ 于 2019 年在 MQL5 中引入了 Python 支持。这种更紧密的“开箱即用”集成允许将技术分析和交易算法完全转移到 Python 环境中。

从技术角度来看，集成是通过在 Python 中安装“MetaTrader5”包来实现的，该包组织了与终端的进程间交互（在撰写本文时，通过 ipykernel/RPC 机制组织交互）。

在该软件包的函数中，包含了内置 MQL5 函数的所有等效函数，可用于获取有关终端、交易帐户、**Market Watch** 中的交易品种、报价、分时报价、市场深度、订单、仓位和交易的信息。此外，该软件包允许你切换交易账户，发送交易订单，检查保证金要求，并实时评估潜在的利润/损失。

然而，与 Python 的集成则有一些限制。特别要注意的是，在 Python 中不可能实现诸如 **OnTick**、**OnBookEvent** 等事件处理。因此，有必要使用无限循环来检查新价格，就像我们在 MQL5 脚本中必须做的那样。对交易订单执行的分析也同样困难：在没有 **OnTradeTransaction** 的情况下，需要更多的代码来了解仓位是完全平仓还是部分平仓。为了绕过这些限制，你可以组织 Python 脚本和 MQL5 的交互，例如，通过套接字组织交互。mq5.com 网站上的文章提供了实现这类桥接的例子。

因此，对于处理报价、分时报价或交易账户历史的机器学习任务，将 Python 与 MetaTrader 5 结合使用似乎是再自然不过的事情了。遗憾的是，在 Python 中你无法获取指标读数。

7.9.1 安装 Python 和 MetaTrader5 包

要学习本章中的资料，必须将 Python 安装到电脑。如果还没有安装它，请从 <https://www.python.org/downloads/windows> 下载最新版本的 Python（例如，在撰写本文时是 3.10 版）。

安装 Python 时，建议选中“Add Python to PATH（将 Python 添加到 PATH）”标志，以便可以从任何文件夹的命令行运行 Python 脚本。

下载并运行 Python 后，从命令行安装 MetaTrader5 模块（这里 **pip** 是一个标准的 Python 包管理器程序）：

```
pip install MetaTrader5
```

随后可以使用以下命令行检查软件包更新：

```
pip install --upgrade MetaTrader5
```

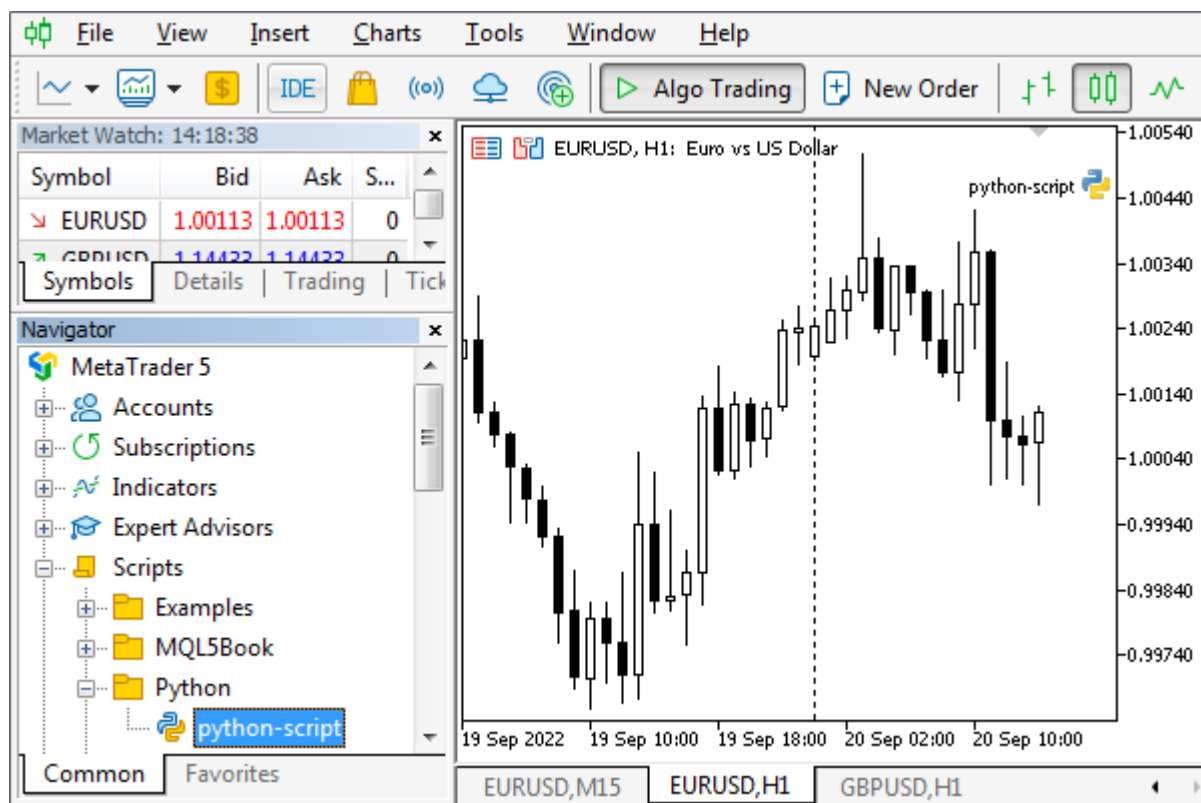
添加其他常用软件包的语法都类似。特别是，许多脚本需要数据分析和可视化软件包：分别为 **pandas** 和 **matplotlib** 软件包。

```
pip install matplotlib
pip install pandas
```

可以直接从 MetaEditor 中的 MQL5 向导创建新的 Python 脚本。除了脚本名称，用户还可以选择导入多个软件包的选项，如 **TensorFlow**、**NumPy** 或 **Datetime**。

默认情况下，建议将脚本放在 **MQL5/Scripts** 文件夹中。新创建和现有 Python 脚本在 MetaTrader 5 导航器中显示，用一个特殊的图标标记，并且可以以通常的方式从导航器中启动。Python 脚

本可以在图表上与其他 MQL5 脚本和 EA 交易并行执行。要在脚本循环执行时停止脚本，只需将其从图表中删除即可。



在终端中运行 Python 脚本

从终端启动的 Python 脚本通过命令行参数接收交易品种的名称和图表的时间范围。例如，我们可以在 EURUSD,H1 图表上运行以下脚本，其中的自变量以 `sys.argv` 数组的形式给出：

```
import sys

print('The command line arguments are:')
for i in sys.argv:
    print(i)
```

结果将输出到 EA 交易日志：

```
The command line arguments are:
C:\Program Files\MetaTrader 5\MQL5\Scripts\MQL5Book\Python\python-args.py
EURUSD
60
```

此外，通过在编辑器 **Settings** 对话框、**Compilers** 选项卡中指定 Python 安装位置，可以直接从 MetaEditor 运行 Python 脚本 - 随后扩展名为 `*.py` 的文件的编译命令，成为运行命令。

最后，Python 脚本也可以在其本机环境中运行，方法是从命令行或另一个适用于 Python 的 IDE（集成开发环境），如 Jupyter Notebook，将它们作为参数在 `python.exe` 调用中传递。

如果在终端中启用了算法交易，那么在默认情况下也会启用从 Python 进行交易。为了在使用第三方 Python 库时进一步保护账户，平台设置提供了“通过外部 Python API 禁用自动交易”选项。这样，Python 脚本可以选择性地阻止交易，方便 MQL 程序来使用。启用此选项时，Python 脚

本中的交易函数调用将返回错误 10027 (TRADE_RETCODE_CLIENT_DISABLES_AT), 表明客户端禁用了算法交易。

MQL5 与 Python 的对比

Python 是一种解释型语言, 而 MQL5 是编译语言。对于我们这些开发人员来说, 这会减少我们很多负担, 因为我们不需要单独的编译阶段来获得一个工作程序。但是, Python 中脚本的执行速度明显低于在 MQL5 中编译的脚本的执行速度。

Python 是一种动态类型语言: 变量的类型是由我们赋予它的值决定的。一方面, 这带来了灵活性, 但另一方面, 我们也需要谨慎, 以避免不可预见的错误。MQL5 使用静态类型, 即在说明变量时, 我们必须显式指定其类型, 同时编译器监控类型兼容性。

Python 会自己“清理垃圾”, 即释放应用程序为对象分配的内存。在 MQL5 中, 我们必须跟踪动态对象的 *delete* 调用。

在 Python 语法中, 源代码缩进是非常重要的。如果你需要编写一个包含多个嵌套语句的复合语句 (例如, 循环或条件语句), 那么 Python 会使用空格或制表符来实现这一目的 (它们在块中必须具有相同的大小)。不允许混合使用制表符和空格进行缩进。不正确的缩进会导致错误。在 MQL5 中, 我们通过花括号 {...} 将多个语句包裹形成复合语句块, 这里没有使用格式化, 你可以在不破坏程序性能的情况下应用喜欢的样式。

Python 函数支持两种类型的参数: 命名参数和位置参数。第二种类型对应于我们在 MQL5 中常用的类型: 每个参数的值必须严格按照它在自变量列表中的顺序传递 (根据函数原型)。相比之下, 命名参数是作为名称和值 (中间用 = 连接) 的组合传递的, 因此可以按任何顺序指定, 例如 *func(param2 = value2, param1 = value1)*。

7.9.2 适用于 Python 的 MetaTrader 5 软件包函数概述

Python 中可用的 API 函数可以有条件地分为 2 组: 在 MQL5 API 中有完全等效的函数和仅在 Python 中可用的函数。第二组之所以存在, 部分是因为 Python 和 MetaTrader 5 之间的连接必须进行技术组织以使用应用函数。这解释了两组函数的存在的原因和目的 *initialize* 和 *shutdown*: 第一个建立到终端的连接, 第二个终止连接。

重要的是, 在初始化过程中, 可以启动所需的终端副本 (如果尚未执行) 并可以选择特定的交易账户。此外, 还可以在已经打开的终端连接的上下文中更改交易账户: 这是通过 *login* 函数完成的。

连接到终端后, Python 脚本可以使用 *version* 函数获取终端版本概述。有关终端的完整信息可通过 *terminal_info* 获取, 该函数完全等效于三个 *TerminalInfo* 函数, 就好像它们在一次调用中结合起来一样。

下表列出了 Python 应用函数及其在 MQL5 API 中的对应函数。

Python	MQL5
last_error	<i>GetLastError</i> (注意! Python 有原生错误代码)
account_info	<i>AccountInfoInteger</i> 、 <i>AccountInfoDouble</i> 、 <i>AccountInfoString</i>
terminal_info	<i>TerminalInfoInteger</i> 、 <i>TerminalInfoDouble</i> 、 <i>TerminalInfoDouble</i>

Python	MQL5
symbols_total	<i>SymbolsTotal</i> (所有交易品种, 包括自定义和禁用交易品种)
symbols_get	<i>SymbolsTotal</i> + <i>SymbolInfo</i> 函数
symbol_info	<i>SymbolInfoInteger</i> 、 <i>SymbolInfoDouble</i> 、 <i>SymbolInfoString</i>
symbol_info_tick	<i>SymbolInfoTick</i>
symbol_select	<i>SymbolSelect</i>
market_book_add	<i>MarketBookAdd</i>
market_book_get	<i>MarketBookGet</i>
market_book_release	<i>MarketBookRelease</i>
copy_rates_from	<i>CopyRates</i> (按柱线数, 从日期/时间开始)
copy_rates_from_pos	<i>CopyRates</i> (按柱线数, 从柱线号开始)
copy_rates_range	<i>CopyRates</i> (在日期/时间范围内)
copy_ticks_from	<i>CopyTicks</i> (按分时报价数, 从指定时间开始)
copy_ticks_range	<i>CopyTicksRange</i> (在指定时间范围内)
orders_total	<i>OrdersTotal</i>
orders_get	<i>OrdersTotal</i> + <i>OrderGet</i> 函数
order_calc_margin	<i>OrderCalcMargin</i>
order_calc_profit	<i>OrderCalcProfit</i>
order_check	<i>OrderCheck</i>
order_send	<i>OrderSend</i>
positions_total	<i>PositionsTotal</i>
positions_get	<i>PositionsTotal</i> + <i>PositionGet</i> 函数
history_orders_total	<i>HistoryOrdersTotal</i>
history_orders_get	<i>HistoryOrdersTotal</i> + <i>HistoryOrderGet</i> 函数
history_deals_total	<i>HistoryDealsTotal</i>
history_deals_get	<i>HistoryDealsTotal</i> + <i>HistoryDealGet</i> 函数

Python API 中的函数有多个特征。

如前所述, 函数可以有命名参数: 函数被调用时, 这些参数与名称和值一起指定, 在每对名称和值中, 它们通过等号 = 组合在一起。命名参数的指定顺序并不重要 (与位置参数不同, 位置参数用于 MQL5 中, 必须遵循函数原型指定的严格顺序)。

Python 函数针对 Python 原生的数据类型进行操作。这不仅包括常见的数字和字符串，还包括几种复合类型，这些有点类似于 MQL5 数组和结构。

例如，许多函数返回特殊的 Python 数据结构体：*tuple* 和 *namedtuple*。

元组是任意类型的元素序列。它可以被认为是一个数组，但与数组不同，元组的元素可以是不同的类型。也可以将元组视为一组结构体字段。

可以发现命名元组与结构体更相似，元组中的每个元素都给定了一个 ID。只有通过索引才可以访问公共元组中的元素（带方括号，如在 MQL5 中，即 `[i]`）。但是，我们可以对命名元组应用解引用运算符（点 `.`），以获取它的“特性”，就像在 MQL5 结构（*tuple.field*）中一样。

此外，元组和命名元组不能在代码中编辑（即它们是常量）。

另一种流行的类型是字典：存储键和值对的关联数组，键和值的类型可以不同。可使用运算符 `[]` 访问字典值，而键（无论是什么类型，例如字符串）用一对方括号表示，这使得字典与数组有几分相似。一个字典不能有两对相同的键，也就是说，键始终是唯一的。特别是，使用 *namedtuple._asdict()* 方法可以很容易地将命名元组转换成字典。

7.9.3 将 Python 脚本连接到终端和账户

initialize 函数建立与 MetaTrader 5 终端的连接，具有两种形式：简短（无参数）和完整（有多个可选参数，第一个 *path*，它是位置参数，其余都是命名参数）。

`bool initialize()`

`bool initialize(path, account = <ACCOUNT>, password = <"PASSWORD">, server = <"SERVER">, timeout = 60000, portable = False)`

path 参数设置终端文件（*metatrader64.exe*）的路径（注意，与所有其他参数不同，这是一个未命名的参数，因此如果指定了该参数，则必须在列表的第一位）。

如果没有指定路径，模块将尝试自己查找可执行文件（开发者没有透露确切的算法）。为了消除歧义，请使用第二种形式的带参数的函数。

在 *account* 参数中，可以指定交易账户的账号。如果未指定，则将使用选定终端实例中的最后一个交易账户。

交易账户的密码在 *password* 参数中指定，也可以省略：如果省略，存储在终端数据库中的指定交易账户的密码将被自动替换。

server 参数的处理方式与交易服务器名称（在终端中指定）类似：如果未指定，则自动替换终端数据库中为指定交易账户保存的服务器。

timeout 参数是指为连接设定的超时时间（以毫秒为单位）（如果超过该时间，将会发生错误）。默认值为 60000（60 秒）。

portable 参数包含用于在便携模式下启动终端的标志（默认为 *False*）。

如果成功连接到 MetaTrader 5 终端，该函数返回 *True*，否则返回 *False*。

如果必要，在调用 *initialize* 时，可以启动 MetaTrader 5 终端。

例如，与特定交易账户的连接展示如下。

```
import MetaTrader5 as mt5
if not mt5.initialize(login = 562175752, server = "MetaQuotes-Demo", password = "abc"
    print("initialize() failed, error code =", mt5.last_error())
    quit()
...

```

login 函数还使用指定的参数连接到交易账户。但这意味着与终端的连接早已建立，即该函数通常用来更改账户。

bool login(account, password = <"PASSWORD">, server = <"SERVER">, timeout = 60000)

account 参数中提供了交易账号。这是一个必需的未命名参数，意味着它必须出现在列表的第一位。

password、**server** 和 **timeout** 参数与 **initialize** 函数的相关参数相同。

如果成功连接到交易账户，函数返回 **True**，否则返回 **False**。

shutdown()

shutdown 函数关闭先前建立的与 MetaTrader 5 终端的连接。

上述函数的例子将在 [下一节](#) 提供。

连接建立后，脚本可以找到终端的版本。

tuple version()

version 函数返回关于 MetaTrader 5 终端版本的简短信息，格式是三个值的元组：版本号、内部版本号和内部版本日期。

字段类型	说明
整数	MetaTrader 5 终端版本 (当前版本为 500)
整数	内部版本号 (例如，3456)
字符串	内部版本日期 (例如 '25 Feb 2022')

如果出现错误，该函数将返回 **None**，并且可以使用 **last_error** 获得错误代码。

有关终端的更完整信息，可以使用 **terminal_info** 函数获取。

7.9.4 错误检查：last_error

last_error 函数返回关于上一个 Python 错误的信息。

int last_error()

整数错误代码不同于分配给 MQL5 错误并由标准 **GetLastError** 函数返回的代码。在下表中，缩写 IPC 是指术语“进程间通信”。

常 量	意 义	说 明
RES_S_OK	1	成功
RES_E_FAIL	-1	常规错误
RES_E_INVALID_PARAMS	-2	自变量无效
RES_E_NO_MEMORY	-3	内存错误
RES_E_NOT_FOUND	-4	未找到历史
RES_E_INVALID_VERSION	-5	不支持的版本
RES_E_AUTH_FAILED	-6	授权错误
RES_E_UNSUPPORTED	-7	不支持的方法
RES_E_AUTO_TRADING_DISABLED	-8	算法禁用
RES_E_INTERNAL_FAIL	-10000	一般PC 错误
RES_E_INTERNAL_FAIL_SEND	-10001	发送PC 数据时出错
RES_E_INTERNAL_FAIL_RECEIVE	-10002	接收PC 数据时出错
RES_E_INTERNAL_FAIL_INIT	-10003	PC 内部错误
RES_E_INTERNAL_FAIL_CONNECT	-10003	无PC
RES_E_INTERNAL_FAIL_TIMEOUT	-10005	PC 超时

在以下脚本 (*MQL5/Scripts/MQL5Book/Python/init.py*) 中，在连接到终端时出现错误的情况下，显示错误代码并退出。

```
import MetaTrader5 as mt5
# show MetaTrader5 package version
print("MetaTrader5 package version: ", mt5.__version__) # 5.0.37

# let's try to establish a connection or launch the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    quit()
... # the working part of the script will be here
# terminate the connection to the terminal
mt5.shutdown()
```

7.9.5 获取交易账户信息

account_info 函数可获取当前交易账户的所有信息。

`namedtuple account_info()`

该函数以命名元组 (*namedtuple*) 的结构体返回信息。如果出现错误，则结果为 *None*。

使用此函数，你可以使用一个调用来获取在 MQL5 中由 [AccountInfoInteger](#)、[AccountInfoDouble](#) 和 [AccountInfoString](#) 提供的所有信息，包括支持特性的所有变体。元组中字段的名称对应于没有 "ACCOUNT_" 前缀的枚举元素的名称，简化为小写。

本书中包含以下 *MQL5/Scripts/MQL5Book/Python/accountinfo.py* 脚本。

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    quit()

account_info = mt5.account_info()
if account_info != None:
    # display trading account data as is
    print(account_info)
    # display data about the trading account in the form of a dictionary
    print("Show account_info()._asdict():")
    account_info_dict = mt5.account_info()._asdict()
    for prop in account_info_dict:
        print(" {}={}".format(prop, account_info_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

结果应该是这样的。

```

AccountInfo(login=25115284, trade_mode=0, leverage=100, limit_orders=200, margin_so_m
Show account_info()._asdict():
login=25115284
trade_mode=0
leverage=100
limit_orders=200
margin_so_mode=0
trade_allowed=True
trade_expert=True
margin_mode=2
currency_digits=2
fifo_close=False
balance=99511.4
credit=0.0
profit=41.82
equity=99553.22
margin=98.18
margin_free=99455.04
margin_level=101398.67590140559
margin_so_call=50.0
margin_so_so=30.0
margin_initial=0.0
margin_maintenance=0.0
assets=0.0
liabilities=0.0
commission_blocked=0.0
name=MetaQuotes Dev Demo
server=MetaQuotes-Demo
currency=USD
company=MetaQuotes Software Corp.

```

7.9.6 获取终端信息

`terminal_info` 函数允许你获取连接的 MetaTrader 5 终端的状态和参数。

`namedtuple terminal_info()`

如果成功，该函数将以命名元组 (*namedtuple*) 的结构体返回信息，如果出错，则返回 *None*。

在这个函数的一次调用中，你可以获取在 MQL5 中由 *TerminalInfoInteger*、*TerminalInfoDouble*，和 *TerminalInfoDouble* 提供的所有信息，包括支持特性的所有变体。元组中字段的名称对应于没有 "TERMINAL_" 前缀的枚举元素的名称，简化为小写。

例如 参见 *MQL5/Scripts/MQL5Book/Python/terminalinfo.py*：


```

import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    quit()

# display brief information about the MetaTrader 5 version
print(mt5.version())
# display full information about the settings and the state of the terminal
terminal_info = mt5.terminal_info()
if terminal_info != None:
    # display terminal data as is
    print(terminal_info)
    # display the data as a dictionary
    print("Show terminal_info()._asdict():")
    terminal_info_dict = mt5.terminal_info()._asdict()
    for prop in terminal_info_dict:
        print(" {}={}".format(prop, terminal_info_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

```

应该像下面这样。

```

[500, 3428, '14 Sep 2022']
TerminalInfo(community_account=True, community_connection=True, connected=True,...
Show terminal_info()._asdict():
community_account=True
community_connection=True
connected=True
dlls_allowed=False
trade_allowed=False
tradeapi_disabled=False
email_enabled=False
ftp_enabled=False
notifications_enabled=False
mqid=False
build=2366
maxbars=5000
codepage=1251
ping_last=77850
community_balance=707.10668201585
retransmission=0.0
company=MetaQuotes Software Corp.
name=MetaTrader 5
language=Russian
path=E:\ProgramFiles\MetaTrader 5
data_path=E:\ProgramFiles\MetaTrader 5
commondata_path=C:\Users\User\AppData\Roaming\MetaQuotes\Terminal\Common

```

7.9.7 获取有关金融工具的信息

MetaTrader5 软件包的函数组提供有关金融工具的信息。

`symbol_info` 函数以命名元组结构体的形式返回关于一个金融工具的信息。

`namedtuple symbol_info(symbol)`

所需金融工具的名称在 `symbol` 参数中指定。

一次调用提供了使用三个 MQL5 函数 `SymbolInfoInteger`、`SymbolInfoDouble`，和 `SymbolInfoString` 以及所有特性所能够获得的所有信息。命名元组中字段的名称与指定函数中使用的枚举元素的名称相同，但没有 "SYMBOL_" 前缀，并且是小写的。

如果出现错误，该函数将返回 `None`。

注意！为确保成功执行函数，必须在 `Market Watch` 中选择请求的交易品种。这可以在 Python 中通过调用 `symbol_select` 来完成（见下文）。

示例 (`MQL5/Scripts/MQL5Book/Python/eurjpy.py`):

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    quit()

# make sure EURJPY is present in the Market Watch, or abort the algorithm
selected = mt5.symbol_select("EURJPY", True)
if not selected:
    print("Failed to select EURJPY")
    mt5.shutdown()
    quit()

# display the properties of the EURJPY symbol
symbol_info = mt5.symbol_info("EURJPY")
if symbol_info != None:
    # display the data as is (as a tuple)
    print(symbol_info)
    # output a couple of specific properties
    print("EURJPY: spread =", symbol_info.spread, ", digits =", symbol_info.digits)
    # output symbol properties as a dictionary
    print("Show symbol_info(\"EURJPY\")._asdict():")
    symbol_info_dict = mt5.symbol_info("EURJPY")._asdict()
    for prop in symbol_info_dict:
        print(" {}={}".format(prop, symbol_info_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

结果：

```

SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_deals=0, se
EURJPY: spread = 17, digits = 3
Show symbol_info()._asdict():
custom=False
chart_mode=0
select=True
visible=True
...
time=1585069682
digits=3
spread=17
spread_float=True
ticks_bookdepth=10
trade_calc_mode=0
trade_mode=4
...
trade_exemode=1
swap_mode=1
swap_rollover3days=3
margin_hedged_use_leg=False
expiration_mode=7
filling_mode=1
order_mode=127
order_gtc_mode=0
...
bid=120.024
ask=120.041
last=0.0
...
point=0.001
trade_tick_value=0.8977708350166538
trade_tick_value_profit=0.8977708350166538
trade_tick_value_loss=0.8978272580355541
trade_tick_size=0.001
trade_contract_size=100000.0
...
volume_min=0.01
volume_max=500.0
volume_step=0.01
volume_limit=0.0
swap_long=-0.2
swap_short=-1.2
margin_initial=0.0
margin_maintenance=0.0
margin_hedged=100000.0
...
currency_base=EUR
currency_profit=JPY
currency_margin=EUR
...

```

`bool symbol_select(symbol, enable = None)`

`symbol_select` 函数将指定的交易品种添加到 *Market Watch* 或删除该交易品种。交易品种在第一个参数中指定。第二个参数作为 *True* 或 *False* 传递，分别表示显示或隐藏交易品种。

如果省略了第二个可选的未命名参数，那么根据 Python 的类型转换规则，`bool(none)` 相当于 `False`。

该函数类似于 `SymbolSelect`。

`int symbols_total()`

`symbols_total` 函数返回 MetaTrader 5 终端中所有金融工具的数量，将自定义交易品种和当前未显示在 *Market Watch* 窗口中的交易品种考虑在内。该函数类似于 `SymbolsTotal(false)`。

接下来，`symbols_get` 函数返回一个元组数组，其中包含所有金融工具或收藏的金融工具的信息，这些金融工具的名称与可选命名参数 `group` 中的指定筛选条件匹配。

`tuple[] symbols_get(group = "PATTERN")`

数组元组中的每个元素都是一个命名元组，具有全套的交易品种特性（我们在上面描述 `symbol_info` 函数的上下文中看到了类似的元组）。

由于只有一个参数，所以在调用函数时可以省略它的名称。

如果出现错误，该函数将返回一个特殊值 `None`。

`group` 参数允许你按名称选择交易品种，也可以在搜索字符串的开头和/或结尾使用替换（通配符）字符 `*`。`*` 表示 0 个或任意数量的字符。因此，你可以搜索名称中出现的子字符串，该子字符串在指定片段之前或之后带有任意数量的其他字符。例如，“EUR*”表示以“EUR”开头且具有任何扩展名（或仅“EUR”）的交易品种。“*EUR*”筛选器将返回名称中任意位置处包含“EUR”子字符串的交易品种。

此外，`group` 参数可包含由逗号分隔的多个条件。可以使用 `*` 将每个条件指定为一个掩码。要排除交易品种，可以使用逻辑非交易品种 `!`。在这种情况下，所有条件都按顺序应用，即首先需要指定包含条件，然后指定排除条件。例如，`group="*, !*EUR*"` 意味着我们需要首先选择所有交易品种，然后排除名称中（任意位置处）包含“EUR”的交易品种。

例如，要显示除 4 种主要外汇货币之外的交叉货币汇率相关信息，可以运行以下查询：

```
crosses = mt5.symbols_get(group = "*,!*USD*,!*EUR*,!*JPY*,!*GBP*")
print('len(*,!*USD*,!*EUR*,!*JPY*,!*GBP*):', len(crosses)) # the size of the resultir
for s in crosses:
    print(s.name, ":", s)
```

结果示例：

```
len(*,!*USD*,!*EUR*,!*JPY*,!*GBP*): 10
AUDCAD : SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_de
AUDCHF : SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_de
AUDNZD : SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_de
CADCHF : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDCAD : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDCHF : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDSGD : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
CADMXN : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
CHFMXN : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDMXN : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
```

`symbol_info_tick` 函数可用于获取指定金融工具最近的分时报价。

`tuple symbol_info_tick(symbol)`

唯一的强制参数指定金融工具的名称。

该信息作为一个元组返回，其字段与 `MqlTick` 结构中的字段相同。该函数类似于 `SymbolInfoTick`。

`None` 如果发生错误，则不返回任何值。

为使该函数正常工作，必须在 `Market Watch` 中启用该交易品种。我们在 `MQL5/Scripts/MQL5Book/Python/gbpusdtick.py` 脚本中进行演示。

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    quit()

# try to include the GBPUSD symbol in the Market Watch
selected=mt5.symbol_select("GBPUSD", True)
if not selected:
    print("Failed to select GBPUSD")
    mt5.shutdown()
    quit()

# display the last tick of the GBPUSD symbol as a tuple
lasttick = mt5.symbol_info_tick("GBPUSD")
print(lasttick)
# display the values of the tick fields in the form of a dictionary
print("Show symbol_info_tick(\"GBPUSD\")._asdict():")
symbol_info_tick_dict = lasttick._asdict()
for prop in symbol_info_tick_dict:
    print(" {}={}".format(prop, symbol_info_tick_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

结果应如下所示：

```
Tick(time=1585070338, bid=1.17264, ask=1.17279, last=0.0, volume=0, time_msc=15850703
Show symbol_info_tick._asdict():
time=1585070338
bid=1.17264
ask=1.17279
last=0.0
volume=0
time_msc=1585070338728
flags=2
volume_real=0.0
```

7.9.8 订阅订单簿变更

Python API 包括三个用于处理 订单簿 的函数。

bool market_book_add(symbol)

market_book_add 函数订阅以接收关于订单簿更改 (指定交易品种) 的事件。所需金融工具的名称在单个未命名的参数中指定。

该函数返回一个布尔成功指示。

该函数类似于 *MarketBookAdd*。在完成订单簿的处理后, 应该通过调用 **market_book_release** 来取消订阅 (见下文)。

tuple[] market_book_get(symbol)

market_book_get 函数请求指定交易品种的订单簿的当前内容。结果作为 *BookInfo* 记录的元组 (数组) 返回。每个条目都类似于 *MqlBookInfo* 结构体, 从 Python 的角度来看, 这是一个带有 "type"、"price"、"volume"、"volume_real" 字段的命名元组。如果出现错误, 将返回 *None* 值。

注意, 出于某种原因, 在 Python 中, 该字段被称为 *volume_dbl*, 但在 MQL5 中, 相应的字段被称为 *volume_real*。

要使用此函数, 你必须首先使用 **market_book_add** 函数订阅以接收订单簿事件。

该函数类似于 *MarketBookGet*。请注意, Python 脚本不能直接接收 *OnBookEvent* 事件, 而应该以循环方式轮询订单簿的内容。

bool market_book_release(symbol)

market_book_release 函数取消对指定交易品种的订单簿更改事件的订阅。如果成功, 该函数将返回 *True*。该函数类似于 *MarketBookRelease*。

我们举一个简单的例子 (参见 *MQL5/Scripts/MQL5Book/Python/eurusdbook.py*)。

```

import MetaTrader5 as mt5
import time # connect a pack for the pause

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    mt5.shutdown()
    quit()

# subscribe to receive DOM updates for the EURUSD symbol
if mt5.market_book_add('EURUSD'):
    # run 10 times a loop to read data from the order book
    for i in range(10):
        # get the contents of the order book
        items = mt5.market_book_get('EURUSD')
        # display the entire order book in one line as is
        print(items)
        # now display each price level separately in the form of a dictionary, for clar
        for it in items or []:
            print(it._asdict())
        # let's pause for 5 seconds before the next request for data from the order boc
        time.sleep(5)
    # unsubscribe to order book changes
    mt5.market_book_release('EURUSD')
else:
    print("mt5.market_book_add('EURUSD') failed, error code =", mt5.last_error())

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

```

结果示例：

```
(BookInfo(type=1, price=1.20036, volume=250, volume_dbl=250.0), BookInfo(type=1, price=1.20035, volume=250, volume_dbl=250.0), BookInfo(type=1, price=1.20029, volume=100, volume_dbl=100.0), BookInfo(type=1, price=1.20028, volume=50, volume_dbl=50.0), BookInfo(type=1, price=1.20026, volume=36, volume_dbl=36.0), BookInfo(type=2, price=1.20023, volume=36, volume_dbl=36.0), BookInfo(type=2, price=1.20022, volume=50, volume_dbl=50.0), BookInfo(type=2, price=1.20021, volume=100, volume_dbl=100.0), BookInfo(type=2, price=1.20014, volume=250, volume_dbl=250.0), BookInfo(type=1, price=1.20035, volume=250, volume_dbl=250.0), BookInfo(type=1, price=1.20029, volume=100, volume_dbl=100.0), BookInfo(type=1, price=1.20027, volume=50, volume_dbl=50.0), BookInfo(type=1, price=1.20025, volume=36, volume_dbl=36.0), BookInfo(type=2, price=1.20023, volume=36, volume_dbl=36.0), BookInfo(type=2, price=1.20022, volume=50, volume_dbl=50.0), BookInfo(type=2, price=1.20021, volume=100, volume_dbl=100.0), BookInfo(type=2, price=1.20014, volume=250, volume_dbl=250.0), BookInfo(type=1, price=1.20037, volume=250, volume_dbl=250.0), BookInfo(type=1, price=1.20031, volume=100, volume_dbl=100.0), BookInfo(type=1, price=1.2003, volume=50, volume_dbl=50.0), BookInfo(type=1, price=1.20028, volume=36, volume_dbl=36.0), BookInfo(type=2, price=1.20025, volume=36, volume_dbl=36.0), BookInfo(type=2, price=1.20023, volume=50, volume_dbl=50.0), BookInfo(type=2, price=1.20022, volume=100, volume_dbl=100.0), BookInfo(type=2, price=1.20016, volume=250, volume_dbl=250.0), ...
```

7.9.9 读取报价

Python API 允许您使用三个函数来获取价格数组（柱线），这三个函数在指定请求数据范围的方式上有所不同：按柱线编号或按时间。所有函数都类似于不同形式的 *CopyRates*。

对于所有函数，前两个参数用于指定交易品种名称和时间范围。时间范围列于 `TIMEFRAME` 枚举中，它类似于 MQL5 中的 `ENUM_TIMEFRAMES` 枚举。

请注意：在 Python 中，该枚举的元素以 `TIMEFRAME_` 为前缀，而 MQL5 中类似枚举的元素以 `PERIOD_` 为前缀。

标识符	说明
TIMEFRAME_M1	1 分钟
TIMEFRAME_M2	2 分钟
TIMEFRAME_M3	3 分钟
TIMEFRAME_M4	4 分钟
TIMEFRAME_M5	5 分钟
TIMEFRAME_M6	6 分钟

标识符	说明
TIMEFRAME_M10	10 分钟
TIMEFRAME_M12	12 分钟
TIMEFRAME_M15	15 分钟
TIMEFRAME_M20	20 分钟
TIMEFRAME_M30	30 分钟
TIMEFRAME_H1	1 小时
TIMEFRAME_H2	2 小时
TIMEFRAME_H3	3 小时
TIMEFRAME_H4	4 小时
TIMEFRAME_H6	6 小时
TIMEFRAME_H8	8 小时
TIMEFRAME_H12	12 小时
TIMEFRAME_D1	1 天
TIMEFRAME_W1	1 周
TIMEFRAME_MN1	1 个月

这三个函数都将柱线作为 `numpy` 批处理数组返回，其中包含命名列 `time`、`open`、`high`、`low`、`close`、`tick_volume`、`spread` 和 `real_volume`。`numpy.ndarray` 数组类似于命名元组，但更高效。要访问列，请使用方括号表示法，例如 `array['column']`。

`None` 如果发生错误，则不返回任何值。

所有函数参数都是强制的和未命名的。

`numpy.ndarray copy_rates_from(symbol, timeframe, date_from, count)`

`copy_rates_from` 函数请求从指定日期 (`date_from`) 开始、`count` 根柱线的数据。该日期可以由 `datetime` 对象设置，或者设置为自 1970.01.01 以来经历的秒数。

创建 `datetime` 对象时，Python 使用本地时区，而 MetaTrader 5 终端使用 UTC (GMT，无偏移) 存储分时报价和柱线的开盘时间。因此，要执行使用时间的函数，必须以 UTC 格式创建 `datetime` 变量。要配置时区，可以使用 `pytz` 软件包。例如 参见

`MT5/Scripts/MT5Book/Python/eurusdrates.py` :

```

from datetime import datetime
import MetaTrader5 as mt5
import pytz                                # import the pytz module to work with the timezone
# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    mt5.shutdown()
    quit()

# set the timezone to UTC
timezone = pytz.timezone("Etc/UTC")

# create a datetime object in the UTC timezone so that the local timezone offset is r
utc_from = datetime(2022, 1, 10, tzinfo = timezone)

# get 10 bars from EURUSD H1 starting from 10/01/2022 in the UTC timezone
rates = mt5.copy_rates_from("EURUSD", mt5.TIMEFRAME_H1, utc_from, 10)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

# display each element of the received data (tuple)
for rate in rates:
    print(rate)

```

接收数据的范例：

```

(1641567600, 1.12975, 1.13226, 1.12922, 1.13017, 8325, 0, 0)
(1641571200, 1.13017, 1.13343, 1.1299, 1.13302, 7073, 0, 0)
(1641574800, 1.13302, 1.13491, 1.13293, 1.13468, 5920, 0, 0)
(1641578400, 1.13469, 1.13571, 1.13375, 1.13564, 3723, 0, 0)
(1641582000, 1.13564, 1.13582, 1.13494, 1.13564, 1990, 0, 0)
(1641585600, 1.1356, 1.13622, 1.13547, 1.13574, 1269, 0, 0)
(1641589200, 1.13572, 1.13647, 1.13568, 1.13627, 1031, 0, 0)
(1641592800, 1.13627, 1.13639, 1.13573, 1.13613, 982, 0, 0)
(1641596400, 1.1361, 1.13613, 1.1358, 1.1359, 692, 1, 0)
(1641772800, 1.1355, 1.13597, 1.13524, 1.1356, 1795, 10, 0)

```

`numpy.ndarray copy_rates_from_pos(symbol, timeframe, start, count)`

`copy_rates_from_pos` 函数请求从指定 *start* 索引开始、*count* 根柱线的数据。

MetaTrader 5 终端仅在图表上用户可见的历史范围内呈现柱线。用户可见的柱线数量由设置中的“窗口中的最大柱线数”参数设定。

以下例子 (*MQL5/Scripts/MQL5Book/Python/ratescorr.py*) 展示了基于报价的几种货币的相关矩阵的图形表示。

```

import MetaTrader5 as mt5
import pandas as pd          # connect the pandas module to output data
import matplotlib.pyplot as plt # connect the matplotlib module for drawing

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    mt5.shutdown()
    quit()

# create a path in the sandbox for the image with the result
image = mt5.terminal_info().data_path + r'\MQL5\Files\MQL5Book\ratescorr'

# the list of working currencies for calculating correlation
sym = ['EURUSD', 'GBPUSD', 'USDJPY', 'USDCHF', 'AUDUSD', 'USDCAD', 'NZDUSD', 'XAUUSD']

# copy the closing prices of bars into DataFrame structures
d = pd.DataFrame()
for i in sym:          # last 1000 M1 bars for each symbol
    rates = mt5.copy_rates_from_pos(i, mt5.TIMEFRAME_M1, 0, 1000)
    d[i] = [y['close'] for y in rates]

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

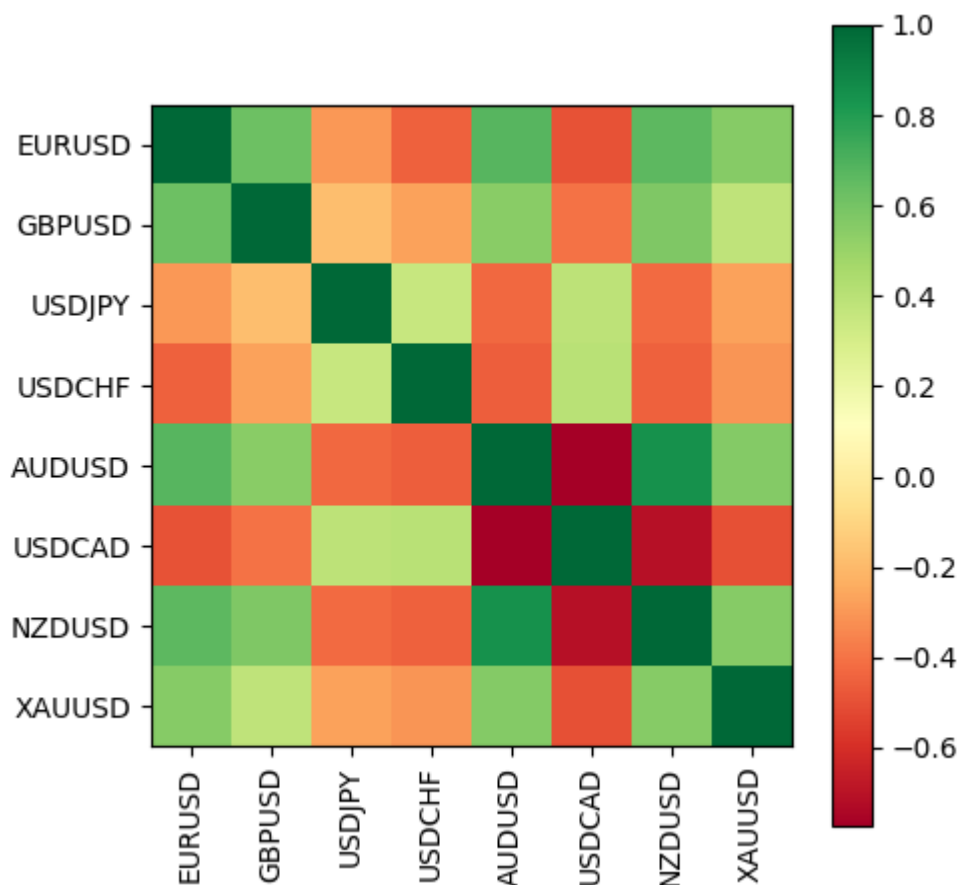
# calculate the price change as a percentage
rets = d.pct_change()

# compute correlations
corr = rets.corr()

# draw the correlation matrix
fig = plt.figure(figsize = (5, 5))
fig.add_axes([0.15, 0.1, 0.8, 0.8])
plt.imshow(corr, cmap = 'RdYlGn', interpolation = 'none', aspect = 'equal')
plt.colorbar()
plt.xticks(range(len(corr)), corr.columns, rotation = 'vertical')
plt.yticks(range(len(corr)), corr.columns)
plt.show()
plt.savefig(image)

```

图像文件 *ratescorr.png* 是在 MetaTrader 5 的当前工作副本的沙盒中形成的。如果你 Python 安装不包括可选功能 “`tk`” 和 “`IDLE`” 或者如果你没有添加 `pip install tk` 软件包，则通过调用 `plt.show()` 在单独的窗口中交互显示图像可能不起作用。



外汇货币相关矩阵

`numpy.ndarray copy_rates_range(symbol, timeframe, date_from, date_to)`

`copy_rates_range` 函数允许您获取 `date_from` 和 `date_to` 之间的指定日期和时间范围内的柱线：这两个值作为 UTC 时区中自 1970 年开始的秒数给出（由于 Python 使用 `datetime` 当地时区，因此应使用 `pytz` 模块进行转换）。结果包括开盘时间在 `time >= date_from` 到 `time <= date_to` 的柱线。

在下面的脚本中，我们将请求特定时间范围内的柱线。

```

from datetime import datetime
import MetaTrader5 as mt5
import pytz                # connect the pytz module to work with the timezone
import pandas as pd       # connect the pandas module to display data in a tabular form

pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)     # max. table width to display

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    quit()

# set the timezone to UTC
timezone = pytz.timezone("Etc/UTC")
# create datetime objects in UTC timezone so that local timezone offset is not applied
utc_from = datetime(2020, 1, 10, tzinfo=timezone)
utc_to = datetime(2020, 1, 10, minute = 30, tzinfo=timezone)

# get bars for USDJPY M5 for period 2020.01.10 00:00 -2020.01.10 00:30 in UTC timezone
rates = mt5.copy_rates_range("USDJPY", mt5.TIMEFRAME_M5, utc_from, utc_to)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

# create a DataFrame from the received data
rates_frame = pd.DataFrame(rates)
# convert time from number of seconds to datetime format
rates_frame['time'] = pd.to_datetime(rates_frame['time'], unit = 's')

# output data
print(rates_frame)

```

结果示例：

```

time open high low close tick_volume spread real_volume
0 2020-01-10 00:00:00 109.513 109.527 109.505 109.521 43 2 0
1 2020-01-10 00:05:00 109.521 109.549 109.518 109.543 215 8 0
2 2020-01-10 00:10:00 109.543 109.543 109.466 109.505 98 10 0
3 2020-01-10 00:15:00 109.504 109.534 109.502 109.517 155 8 0
4 2020-01-10 00:20:00 109.517 109.539 109.513 109.527 71 4 0
5 2020-01-10 00:25:00 109.526 109.537 109.484 109.520 106 9 0
6 2020-01-10 00:30:00 109.520 109.524 109.508 109.510 205 7 0

```

7.9.10 读取分时报价历史

Python API 包括两个用于读取分时报价历史的函数：`copy_ticks_from` 表示从指定日期开始的分时报价数，`copy_ticks_range` 表示指定时间段内的分时报价数。

这两个函数都有四个必需的未命名参数，第一个参数指定交易品种。第二个参数指定所请求的分时报价的初始时间。第三个参数 (在 `copy_ticks_from` 函数中)表示传递的分时报价数或 (在 `copy_ticks_range` 函数中)表示分时报价的结束时间。

最后一个参数确定将返回哪种分时报价。它可以包含以下标志之一 (COPY_TICKS):

标识符	说明
COPY_TICKS_ALL	所有分时报价
COPY_TICKS_INFO	包含买价和/或卖价变化的分时报价
COPY_TICKS_TRADE	包含最后价格和/或交易量变化的分时报价

这两个函数都 (从 `numpy` 包)将分时报价作为 `numpy.ndarray` 数组返回，该数组包含命名列 `time`、`bid`、`ask`、`last` 和 `flags`。`flags` 字段的值是 `TICK_FLAG` 枚举的位标志的组合：每一位表示分时报价特性对应字段的一次变化。

标识符	已更改分时报价特性
TICK_FLAG_BID	买价
TICK_FLAG_ASK	卖价
TICK_FLAG_LAST	最后价格
TICK_FLAG_VOLUME	交易量
TICK_FLAG_BUY	最后买入价格
TICK_FLAG_SELL	最后卖出价格

`numpy.ndarray copy_ticks_from(symbol, date_from, count, flags)`

`copy_ticks_from` 函数请求从指定时间 (`date_from`) 开始的给定数量的 (`count`) 分时报价。

该函数类似于 `CopyTicks`。

`numpy.array copy_ticks_range(symbol, date_from, date_to, flags)`

`copy_ticks_range` 函数允许你获取指定时间范围的分时报价。

该函数类似于 `CopyTicksRange`。

在下面的例子中 (`MQL5/Scripts/MQL5Book/Python/copyticks.py`)，我们生成了一个带有分时报价图表的交互式网页 (注意：这里使用了 `plotly` 软件包；要在 Python 中安装该软件包，请运行命令 `pip install plotly`)。

```

import MetaTrader5 as mt5
import pandas as pd
import pytz
from datetime import datetime

# connect to terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    quit()

# set the name of the file to save to the sandbox
path = mt5 terminal_info().data_path + r'\MQL5\Files\MQL5Book\copyticks.html'

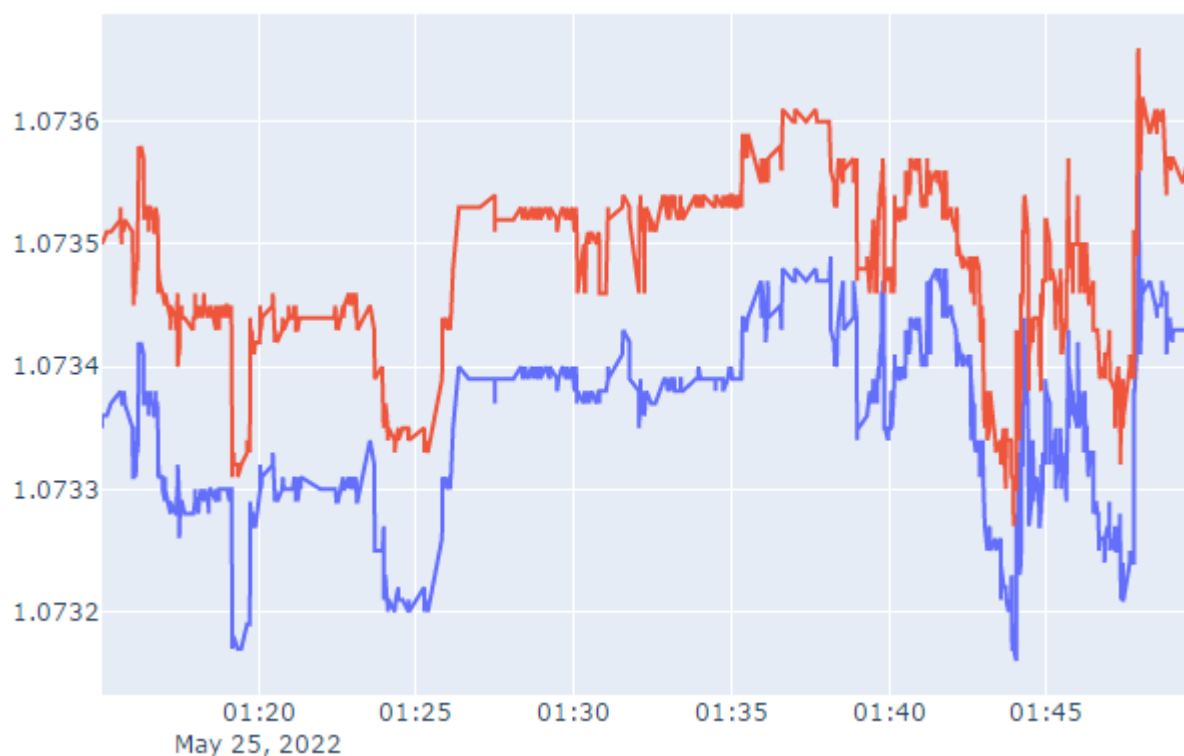
# copy 1000 EURUSD ticks from a specific moment in history
utc = pytz.timezone("Etc/UTC")
rates = mt5.copy_ticks_from("EURUSD", \
datetime(2022, 5, 25, 1, 15, tzinfo = utc), 1000, mt5.COPY_TICKS_ALL)
bid = [x['bid'] for x in rates]
ask = [x['ask'] for x in rates]
time = [x['time'] for x in rates]
time = pd.to_datetime(time, unit = 's')

# terminate the connection to the terminal
mt5.shutdown()

# connect the graphics package and draw 2 rows of ask and bid prices on the web page
import plotly.graph_objs as go
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
data = [go.Scatter(x = time, y = bid), go.Scatter(x = time, y = ask)]
plot(data, filename = path)

```

结果可能像这样。



Python 脚本中接收到包含分时报价的图表

在 `MQL5/Files/MQL5Book` 子目录中生成 `copyticks.html` Web 页。

7.9.11 保证金要求的计算和利润估算

Python 开发人员可以使用 `order_calc_margin` 和 `order_calc_profit` 函数直接计算脚本中提议的交易操作的保证金和潜在损益。在成功执行的情况下，任何函数的结果都是一个实数；否则，结果为 `None`。

`float order_calc_margin(action, symbol, volume, price)`

`order_calc_margin` 函数返回完成指定交易操作 `action` 所需的保证金金额（以账户货币为单位），可以是 `ENUM_ORDER_TYPE` 枚举中的两个元素之一：`ORDER_TYPE_BUY` 或 `ORDER_TYPE_SELL`。以下参数指定了金融工具的名称、交易额和开盘价。

该函数类似于 `OrderCalcMargin`。

`float order_calc_profit(action, symbol, volume, price_open, price_close)`

`order_calc_profit` 函数返回指定交易类型、交易品种和交易量的损益金额（以账户货币为单位），以及市场买入和卖出价格之间的差额。

该函数类似于 `OrderCalcProfit`。

建议在发送订单之前，检查保证金和交易操作的预期结果。

7.9.12 检查和发送交易订单

如有必要，你可以直接从 Python 脚本进行交易。`order_check` 和 `order_send` 这两个函数允许你预先检查，然后执行交易操作。

对于这两个函数，唯一的参数是请求结构 `TradeRequest`（它可以在 Python 中初始化为字典，参见例子）。结构字段与 `MqlTradeRequest` 完全相同。

`OrderCheckResult order_check(request)`

`order_check` 函数检查交易请求字段的正确性以及完成所需交易操作的资金充足性。

函数的结果作为 `OrderCheckResult` 结构体返回。它重复了 `MqlTradeCheckResult` 的结构体，但是还包含具有原始请求副本的 `request` 字段。

`order_check` 函数类似于 `OrderCheck`。

示例 (`MT5/Scripts/MT5Book/python/ordercheck.py`):

```

import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
...
# get account currency for information
account_currency=mt5.account_info().currency
print("Account currency:", account_currency)

# get the necessary properties of the deal symbol
symbol = "USDJPY"
symbol_info = mt5.symbol_info(symbol)
if symbol_info is None:
    print(symbol, "not found, can not call order_check()")
    mt5.shutdown()
    quit()

point = mt5.symbol_info(symbol).point
# if the symbol is not available in the Market Watch, add it
if not symbol_info.visible:
    print(symbol, "is not visible, trying to switch on")
    if not mt5.symbol_select(symbol, True):
        print("symbol_select({}) failed, exit", symbol)
        mt5.shutdown()
        quit()

# prepare the query structure as a dictionary
request = \
{
    "action": mt5.TRADE_ACTION_DEAL,
    "symbol": symbol,
    "volume": 1.0,
    "type": mt5.ORDER_TYPE_BUY,
    "price": mt5.symbol_info_tick(symbol).ask,
    "sl": mt5.symbol_info_tick(symbol).ask - 100 * point,
    "tp": mt5.symbol_info_tick(symbol).ask + 100 * point,
    "deviation": 10,
    "magic": 234000,
    "comment": "python script",
    "type_time": mt5.ORDER_TIME_GTC,
    "type_filling": mt5.ORDER_FILLING_RETURN,
}

# run the test and display the result as is
result = mt5.order_check(request)
print(result) # [?this is not in the help log?]

# convert the result to a dictionary and output element by element
result_dict = result._asdict()
for field in result_dict.keys():
    print(" {}={}".format(field, result_dict[field]))
# if this is the structure of a trade request, then output it element by element :

```

```

    if field == "request":
        traderequest_dict = result_dict[field]._asdict()
        for tradereq_filed in traderequest_dict:
            print("        traderequest: {}={}".format(tradereq_filed,
                traderequest_dict[tradereq_filed]))

# terminate the connection to the terminal
mt5.shutdown()

```

结果：

```

账户货币:USD
OrderCheckResult(retcode=0, balance=10000.17, equity=10000.17, profit=0.0, margin=100
retcode=0
balance=10000.17
equity=10000.17
profit=0.0
margin=1000.0
margin_free=9000.17
margin_level=1000.017
comment=Done
request=TradeRequest(action=1, magic=234000, order=0, symbol='USDJPY', volume=1.0, pr
traderequest: action=1
traderequest: magic=234000
traderequest: order=0
traderequest: symbol=USDJPY
traderequest: volume=1.0
traderequest: price=144.128
traderequest: stoplimit=0.0
traderequest: sl=144.028
traderequest: tp=144.228
traderequest: deviation=10
traderequest: type=0
traderequest: type_filling=2
traderequest: type_time=0
traderequest: expiration=0
traderequest: comment=python script
traderequest: position=0
traderequest: position_by=0

```

OrderSendResult order_send(request)

order_send 函数从终端向交易服务器发送一个请求以进行交易操作。

函数的结果作为 **OrderSendResult** 结构体返回。它重复了 **MqlTradeResult** 的结构体，但是还包含具有原始请求副本的 **request** 字段。

该函数类似于 **OrderSend**。

示例 (**MT5/Scripts/MT5Book/python/ordersend.py**):

```

import time
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
...
# assign the properties of the working symbol
symbol = "USDJPY"
symbol_info = mt5.symbol_info(symbol)
if symbol_info is None:
    print(symbol, "not found, can not trade")
    mt5.shutdown()
    quit()

# if the symbol is not available in the Market Watch, add it
if not symbol_info.visible:
    print(symbol, "is not visible, trying to switch on")
    if not mt5.symbol_select(symbol, True):
        print("symbol_select({}) failed, exit", symbol)
        mt5.shutdown()
        quit()

# let's prepare the request structure for the purchase
lot = 0.1
point = mt5.symbol_info(symbol).point
price = mt5.symbol_info_tick(symbol).ask
deviation = 20
request = \
{
    "action": mt5.TRADE_ACTION_DEAL,
    "symbol": symbol,
    "volume": lot,
    "type": mt5.ORDER_TYPE_BUY,
    "price": price,
    "sl": price - 100 * point,
    "tp": price + 100 * point,
    "deviation": deviation,
    "magic": 234000,
    "comment": "python script open",
    "type_time": mt5.ORDER_TIME_GTC,
    "type_filling": mt5.ORDER_FILLING_RETURN,
}

# send a trade request to open a position
result = mt5.order_send(request)
# check the execution result
print("1. order_send(): by {} {} lots at {}".format(symbol, lot, price));
if result.retcode != mt5.TRADE_RETCODE_DONE:
    print("2. order_send failed, retcode={}".format(result.retcode))
    # request the result as a dictionary and display it element by element
    result_dict = result._asdict()
    for field in result_dict.keys():

```

```

    print("    {}={}".format(field, result_dict[field]))
    # if this is the structure of a trade request, then output it element by element
    if field == "request":
        traderequest_dict = result_dict[field]._asdict()
        for tradereq_filed in traderequest_dict:
            print("        traderequest: {}={}".format(tradereq_filed,
                traderequest_dict[tradereq_filed]))
    print("shutdown() and quit")
    mt5.shutdown()
    quit()

print("2. order_send done, ", result)
print("    opened position with POSITION_TICKET={}".format(result.order))
print("    sleep 2 seconds before closing position #{}".format(result.order))
time.sleep(2)
# create a request to close
position_id = result.order
price = mt5.symbol_info_tick(symbol).bid
request = \
{
    "action": mt5.TRADE_ACTION_DEAL,
    "symbol": symbol,
    "volume": lot,
    "type": mt5.ORDER_TYPE_SELL,
    "position": position_id,
    "price": price,
    "deviation": deviation,
    "magic": 234000,
    "comment": "python script close",
    "type_time": mt5.ORDER_TIME_GTC,
    "type_filling": mt5.ORDER_FILLING_RETURN,
}
# send a trade request to close the position
result = mt5.order_send(request)
# check the execution result
print("3. close position #{}: sell {} {} lots at {}".format(position_id,
symbol, lot, price));
if result.retcode != mt5.TRADE_RETCODE_DONE:
    print("4. order_send failed, retcode={}".format(result.retcode))
    print("    result", result)
else:
    print("4. position #{} closed, {}".format(position_id, result))
    # request the result as a dictionary and display it element by element
    result_dict = result._asdict()
    for field in result_dict.keys():
        print("    {}={}".format(field, result_dict[field]))
    # if this is the structure of a trade request, then output it element by element
    if field == "request":
        traderequest_dict = result_dict[field]._asdict()
        for tradereq_filed in traderequest_dict:
            print("        traderequest: {}={}".format(tradereq_filed,

```

```
traderequest_dict[tradereq_filed]))
```

```
# terminate the connection to the terminal
mt5.shutdown()
```

结果：

```
1. order_send(): by USDJPY 0.1 lots at 144.132
2. order_send done, OrderSendResult(retcode=10009, deal=1445796125, order=1468026008,
opened position with POSITION_TICKET=1468026008
sleep 2 seconds before closing position #1468026008
3. close position #1468026008: sell USDJPY 0.1 lots at 144.124
4. position #1468026008 closed, OrderSendResult(retcode=10009, deal=1445796155, order
retcode=10009
deal=1445796155
order=1468026041
volume=0.1
price=144.124
bid=144.124
ask=144.132
comment=Request executed
request_id=2
retcode_external=0
request=TradeRequest(action=1, magic=234000, order=0, symbol='USDJPY', volume=0.1, pr
traderequest: action=1
traderequest: magic=234000
traderequest: order=0
traderequest: symbol=USDJPY
traderequest: volume=0.1
traderequest: price=144.124
traderequest: stoplimit=0.0
traderequest: sl=0.0
traderequest: tp=0.0
traderequest: deviation=20
traderequest: type=1
traderequest: type_filling=2
traderequest: type_time=0
traderequest: expiration=0
traderequest: comment=python script close
traderequest: position=1468026008
traderequest: position_by=0
```

7.9.13 获取活动订单的数量和列表

Python API 提供了以下用于处理活动订单的函数。

int orders_total()

orders_total 函数返回活动订单的数量。

该函数类似于 *订单总计*。

可以使用 **orders_get** 函数获得关于每个订单的详细信息，该函数有几个选项，能够按交易品种或订单号进行筛选。无论哪种方式，该函数都返回命名元组 **TradeOrder** 的数组（字段名称匹配

`ENUM_ORDER_PROPERTY_enumerations` 不带 "ORDER_" 前缀并简化为小写)。如果出现错误,则结果为 *None*。

```
namedtuple[] orders_get()
namedtuple[] orders_get(symbol = <"SYMBOL">)
namedtuple[] orders_get(group = <"PATTERN">)
namedtuple[] orders_get(ticket = <"TICKET">)
```

不含参数的 `orders_get` 函数返回所有交易品种的订单。

可选的命名参数 *symbol* 可以为订单选择指定特定的交易品种名称。

可选的命名参数 *group* 旨在使用通配符 '*' 替代任意数量的任何字符,包括模式中给定位置的零字符)和条件逻辑非字符 '!' 来指定搜索模式。筛选模板的操作原理在 [获取有关金融工具的信息](#) 一节中说明。

如果指定了 *ticket* 参数,则搜索某个订单。

在一次函数调用中,您可以获取所有活动订单。它类似于 [OrdersTotal](#)、[OrderSelect](#), 和 [OrderGet](#) 函数的组合使用。

在下面的例子 ([MQL5/Scripts/MQL5Book/Python/ordersget.py](#)) 中,我们使用不同的方式请求有关订单的信息。

```

import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)     # max. table width to display

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    quit()

# display information about active orders on the GBPUSD symbol
orders = mt5.orders_get(symbol = "GBPUSD")
if orders is None:
    print("No orders on GBPUSD, error code={}".format(mt5.last_error()))
else:
    print("Total orders on GBPUSD:", len(orders))
    # display all active orders
    for order in orders:
        print(order)
print()

# getting a list of orders on symbols whose names contain "*GBP*"
gbp_orders = mt5.orders_get(group="*GBP*")
if gbp_orders is None:
    print("No orders with group=\"*GBP*\", error code={}".format(mt5.last_error()))
else:
    print("orders_get(group=\"*GBP*\")={}".format(len(gbp_orders)))
    # display orders as a table using pandas.DataFrame
    df = pd.DataFrame(list(gbp_orders), columns = gbp_orders[0]._asdict().keys())
    df.drop(['time_done', 'time_done_msc', 'position_id', 'position_by_id',
            'reason', 'volume_initial', 'price_stoplimit'], axis = 1, inplace = True)
    df['time_setup'] = pd.to_datetime(df['time_setup'], unit = 's')
    print(df)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

```

结果示例如下：

```

Total orders on GBPUSD: 2
TradeOrder(ticket=554733548, time_setup=1585153667, time_setup_msc=1585153667718, tim
TradeOrder(ticket=554733621, time_setup=1585153671, time_setup_msc=1585153671419, tim

orders_get(group="*GBP*")=4
ticket time_setup time_setup_msc type ... volume_current price_open sl tp price_curre
0 554733548 2020-03-25 16:27:47 1585153667718 3 ...0.2 1.25379 0.0 0.0 1.16803 GBPUSD
1 554733621 2020-03-25 16:27:51 1585153671419 2 ...0.2 1.14370 0.0 0.0 1.16815 GBPUSD
2 554746664 2020-03-25 16:38:14 1585154294401 3 ...0.2 0.93851 0.0 0.0 0.92428 EURGBP
3 554746710 2020-03-25 16:38:17 1585154297022 2 ...0.2 0.90527 0.0 0.0 0.92449 EURGBP

```


7.9.14 获取未平仓仓位数量和清单

positions_total 函数返回未平仓仓位数量。

```
int positions_total()
```

该函数类似于 *PositionsTotal*。

为获取每个仓位的详细信息，可以使用 *positions_get* 函数，该函数有多个选项。所有变量都返回一个命名元组 *TradePosition* 的数组，其索引对应仓位特性（参见 *ENUM_POSITION_PROPERTY_enumerations* 的元素，不带 “POSITION_” 前缀，小写）。如果出现错误，则结果为 *None*。

```
namedtuple[] positions_get()
namedtuple[] positions_get(symbol = <"SYMBOL">)
namedtuple[] positions_get(group = <"PATTERN">)
namedtuple[] positions_get(ticket = <"TICKET">)
```

不含参数的函数返回所有未平仓仓位。

包含 *symbol* 参数的函数允许选择指定符号的仓位。

包含 *group* 参数的函数通过搜索掩码提供筛选，该掩码使用通配符 '*'（替换任何字符）和条件逻辑非 '!'. 有关详细信息，参见 [获取有关金融工具的信息](#) 一节。

包含 *ticket* 参数的版本使用特定票证选择仓位（POSITION_TICKET 特性）。

positions_get 函数可用于在一次调用中获取所有仓位及其特性，这使它类似于 *PositionsTotal*、*PositionSelect* 和 *PositionGet* 之类的函数。

在 *MQL5/Scripts/MQL5Book/Python/positionsget.py* 脚本中，我们请求特定交易品种的仓位和搜索掩码。

```

import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)     # max. table width to display

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    quit()

# get open positions on USDCHF
positions = mt5.positions_get(symbol = "USDCHF")
if positions == None:
    print("No positions on USDCHF, error code={}".format(mt5.last_error()))
elif len(positions) > 0:
    print("Total positions on USDCHF =", len(positions))
    # display all open positions
    for position in positions:
        print(position)

# get a list of positions on symbols whose names contain "*USD*"
usd_positions = mt5.positions_get(group = "*USD*")
if usd_positions == None:
    print("No positions with group=\"*USD*\", error code={}".format(mt5.last_error()))
elif len(usd_positions) > 0:
    print("positions_get(group=\"*USD*\") = {}".format(len(usd_positions)))
    # display the positions as a table using pandas.DataFrame
    df=pd.DataFrame(list(usd_positions), columns = usd_positions[0]._asdict().keys())
    df['time'] = pd.to_datetime(df['time'], unit='s')
    df.drop(['time_update', 'time_msc', 'time_update_msc', 'external_id'],
            axis=1, inplace=True)
    print(df)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

```

结果可能是这样的：

```

Total positions on USDCHF = 1
TradePosition(ticket=1468454363, time=1664217233, time_msc=1664217233239, time_update
time_update_msc=1664217233239, type=1, magic=0, identifier=1468454363, reason=0, volu
sl=0.0, tp=0.0, price_current=0.9853, swap=-0.01, profit=6.24, symbol='USDCHF', comme
positions_get(group="*USD*") = 2
ticket time type ... identifier volume price_open ..._current swap profit symbol comm
0 1468454363 2022-09-26 18:33:53 1 ...1468454363 0.01 0.99145 ...0.98530 -0.01 6.24 U
1 1468475849 2022-09-26 18:44:00 0 ...1468475849 0.01 1.06740 ...1.08113 0.00 13.73 G

```

7.9.15 读取订单和交易的历史

也可以使用 Python 脚本处理账户历史中的订单和交易。为了达到这些目的，提供了函数 `history_orders_total`、`history_orders_get`、`history_deals_total` 和 `history_deals_get`。

```
int history_orders_total(date_from, date_to)
```

history_orders_total 函数返回指定时间间隔内交易历史中的订单数量。每个参数都由 *datetime* 对象设置，或者设置为自 1970.01.01 以来经历的秒数。

该函数类似于 *HistoryOrdersTotal*。

history_orders_get 函数有多个版本，支持按交易品种名称、订单号或位置 ID 中的子字符串进行订单筛选。所有变量都返回一个命名元组 *TradeOrder* 的数组（字段名称匹配 *ENUM_ORDER_PROPERTY_enumerations* 不带 "ORDER_" 前缀并且是小写的）。如果没有匹配的订单，数组将为空。如果出现错误，该函数将返回 *None*。

```
namedtuple[] history_orders_get(date_from, date_to, group = <"PATTERN">)
```

```
namedtuple[] history_orders_get(ticket = <ORDER_TICKET>)
```

```
namedtuple[] history_orders_get(position = <POSITION_ID>)
```

第一个版本选择指定时间范围内的订单（类似于 *history_orders_total*）。在可选的命名参数 *group* 中，你可以为交易品种名称的子字符串指定搜索模式（你可以在其中使用通配符 '*' 和非 '!', 参见 获取有关金融工具的信息一节）。

第二个版本旨在按订单号搜索特定订单。

最后一个版本按仓位 ID (ORDER_POSITION_ID 特性) 选择订单。

任一选项都相当于调用多个 MQL5 函数：*HistoryOrdersTotal*、*HistoryOrderSelect* 和 *HistoryOrderGet* 函数。

这里是一个 *historyordersget.py* 脚本的例子，让我们看看如何获取不同条件下的历史订单的数量和列表。

```

from datetime import datetime
import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)     # max. table width for display
...
# get the number of orders in the history for the period (total and *GBP*)
from_date = datetime(2022, 9, 1)
to_date = datetime.now()
total = mt5.history_orders_total(from_date, to_date)
history_orders=mt5.history_orders_get(from_date, to_date, group="*GBP*")
# print(history_orders)
if history_orders == None:
    print("No history orders with group=\"*GBP*\", error code={}".format(mt5.last_errc))
else :
    print("history_orders_get({}, {}, group=\"*GBP*\")={{} of total {}".format(from_date,
to_date, len(history_orders), total))

# display all canceled historical orders for ticket position 0
position_id = 0
position_history_orders = mt5.history_orders_get(position = position_id)
if position_history_orders == None:
    print("No orders with position #{}".format(position_id))
    print("error code =", mt5.last_error())
elif len(position_history_orders) > 0:
    print("Total history orders on position #{}: {}".format(position_id,
len(position_history_orders)))
    # display received orders as is
    for position_order in position_history_orders:
        print(position_order)
    # display these orders as a table using pandas.DataFrame
    df = pd.DataFrame(list(position_history_orders),
columns = position_history_orders[0]._asdict().keys())
    df.drop(['time_expiration', 'type_time', 'state', 'position_by_id', 'reason', 'vol',
'price_stoptlimit', 'sl', 'tp', 'time_setup_msc', 'time_done_msc', 'type_filling', 'e',
axis = 1, inplace = True)
    df['time_setup'] = pd.to_datetime(df['time_setup'], unit='s')
    df['time_done'] = pd.to_datetime(df['time_done'], unit='s')
    print(df)
...

```

脚本的结果 (用缩写给出):

```
history_orders_get(2022-09-01 00:00:00, 2022-09-26 21:50:04, group="*GBP*")=15 of total 44
```

Total history orders on position #0: 14

```
TradeOrder(ticket=1437318706, time_setup=1661348065, time_setup_msc=1661348065049, time_done=166134
time_done_msc=1661348083632, time_expiration=0, type=2, type_time=0, type_filling=2, state=2, magic
position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=0.9
sl=0.0, tp=0.0, price_current=0.99311, price_stoplimit=0.0, symbol='EURUSD', comment='', external_i
TradeOrder(ticket=1437331579, time_setup=1661348545, time_setup_msc=1661348545750, time_done=166134
time_done_msc=1661348551354, time_expiration=0, type=2, type_time=0, type_filling=2, state=2, magic
position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=0.9
sl=0.0, tp=0.0, price_current=0.99284, price_stoplimit=0.0, symbol='EURUSD', comment='', external_i
TradeOrder(ticket=1437331739, time_setup=1661348553, time_setup_msc=1661348553935, time_done=166134
time_done_msc=1661348563412, time_expiration=0, type=2, type_time=0, type_filling=2, state=2, magic
position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=0.9
sl=0.0, tp=0.0, price_current=0.99286, price_stoplimit=0.0, symbol='EURUSD', comment='', external_i
...
```

```
ticket time_setup time_done type ..._initial price_open price_current symbol comment
0 1437318706 2022-08-24 13:34:25 2022-08-24 13:34:43 2 0.01 0.99301 0.99311 EURUSD
1 1437331579 2022-08-24 13:42:25 2022-08-24 13:42:31 2 0.01 0.99281 0.99284 EURUSD
2 1437331739 2022-08-24 13:42:33 2022-08-24 13:42:43 2 0.01 0.99285 0.99286 EURUSD
...
```

我们可以看到，在 9 月份，只有 44 个订单，其中 15 个订单包括英镑货币（由于未平仓仓位，这个数字是一个奇数）。历史记录包含 14 个取消的订单。

int history_deals_total(date_from, date_to)

history_deals_total 函数返回指定期间的历史交易数。

该函数类似于 *HistoryDealsTotal*。

history_deals_get 函数有多种形式，用于选择能够按订单订单号或仓位 ID 筛选的交易。所有形式的函数都返回一个命名元组 *TradeDeal* 的数组，其中的字段反映 *ENUM_DEAL_PROPERTY_enumerations* 中的特性（前缀 “DEAL_” 已从字段名中删除，并应用小写字母）。如果出错，则返回 *None*。

namedtuple[] history_deals_get(date_from, date_to, group = <"PATTERN">)

namedtuple[] history_deals_get(ticket = <ORDER_TICKET>)

namedtuple[] history_deals_get(position = <POSITION_ID>)

该函数的第一种形式类似于使用 *history_orders_get* 请求历史订单。

第二种形式允许按订单号选择由特定订单生成的交易（DEAL_ORDER 特性）。

最后，第三个表单请求已经形成仓位的交易，该仓位具有给定 ID（DEAL_POSITION_ID 特性）。

该函数允许你在一次调用中获取所有事务及其特性，这类似于 *HistoryDealsTotal*、*HistoryDealSelect* 和 *HistoryDealGet* 函数。

下面是测试脚本 *historydealsget.py* 的主要部分。

```

# set the time range
from_date = datetime(2020, 1, 1)
to_date = datetime.now()

# get trades for symbols whose names do not contain either "EUR" or "GBP"
deals = mt5.history_deals_get(from_date, to_date, group="*,!*EUR*,!*GBP*")
if deals == None:
    print("No deals, error code={}".format(mt5.last_error()))
elif len(deals) > 0:
    print("history_deals_get(from_date, to_date, group='*,!*EUR*,!*GBP*') =",
          len(deals))
    # display all received deals as they are
    for deal in deals:
        print(" ", deal)
    # display these trades as a table using pandas.DataFrame
    df = pd.DataFrame(list(deals), columns = deals[0]._asdict().keys())
    df['time'] = pd.to_datetime(df['time'], unit='s')
    df.drop(['time_msc', 'commission', 'fee'], axis = 1, inplace = True)
    print(df)

```

结果示例：

```

history_deals_get(from_date, to_date, group="*,!*EUR*,!*GBP*") = 12
TradeDeal(ticket=1109160642, order=0, time=1632188460, time_msc=1632188460852, type=2
TradeDeal(ticket=1250629232, order=1268074569, time=1645709385, time_msc=164570938581
TradeDeal(ticket=1250639814, order=1268085019, time=1645709950, time_msc=164570995061
TradeDeal(ticket=1250639928, order=1268085129, time=1645709955, time_msc=164570995550
TradeDeal(ticket=1250640111, order=1268085315, time=1645709965, time_msc=164570996514
TradeDeal(ticket=1250640309, order=1268085512, time=1645709973, time_msc=164570997362
TradeDeal(ticket=1250640400, order=1268085611, time=1645709978, time_msc=164570997870
TradeDeal(ticket=1250640616, order=1268085826, time=1645709988, time_msc=164570998827
TradeDeal(ticket=1250640810, order=1268086019, time=1645709996, time_msc=164570999699
TradeDeal(ticket=1445796125, order=1468026008, time=1664199450, time_msc=166419945048
TradeDeal(ticket=1445796155, order=1468026041, time=1664199452, time_msc=166419945256
TradeDeal(ticket=1446217804, order=1468454363, time=1664217233, time_msc=166421723323

ticket order time t... e... .. position_id volume price profit symbol comment external_id
0 1109160642 0 2021-09-21 01:41:00 2 0 0 0.00 0.00000 10000.00
1 1250629232 1268074569 2022-02-24 13:29:45 0 0 1268074569 0.01 1970.98000 0.00 XAUUS
2 1250639814 1268085019 2022-02-24 13:39:10 1 1 1268074569 0.01 1970.09000 -0.89 XAUU
3 1250639928 1268085129 2022-02-24 13:39:15 1 0 1268085129 0.01 1969.98000 0.00 XAUUS
4 1250640111 1268085315 2022-02-24 13:39:25 0 1 1268085129 0.01 1970.17000 -0.19 XAUU
5 1250640309 1268085512 2022-02-24 13:39:33 1 0 1268085512 0.10 1970.09000 0.00 XAUUS
6 1250640400 1268085611 2022-02-24 13:39:38 0 1 1268085512 0.10 1970.22000 -1.30 XAUU
7 1250640616 1268085826 2022-02-24 13:39:48 1 0 1268085826 1.10 1969.95000 0.00 XAUUS
8 1250640810 1268086019 2022-02-24 13:39:56 0 1 1268085826 1.10 1969.88000 7.70 XAUUS
9 1445796125 1468026008 2022-09-26 13:37:30 0 0 1468026008 0.10 144.13200 0.00 USDJPY
10 1445796155 1468026041 2022-09-26 13:37:32 1 1 1468026008 0.10 144.12400 -0.56 USDJ
11 1446217804 1468454363 2022-09-26 18:33:53 1 0 1468454363 0.01 0.99145 0.00 USDCHF

```

7.10 内置对并行计算的支持:OpenCL

OpenCL 是一种开放并行编程标准,它允许你创建在现代处理器的多个核心上同时执行的应用程序,这些处理器具有不同的架构,特别是图形 (GPU) 或中央 (CPU) 处理器。

换句话说,OpenCL 允许你使用中央处理器的所有核心或显卡的所有计算能力来计算一个任务,最终减少了程序的执行时间。因此,OpenCL 的使用对于计算密集型任务非常有用,但需要注意的是,解决这些任务的算法必须可分为并行线程。这些包括,例如训练神经网络、傅立叶变换或求解大维数的方程组。

例如,关于交易细节,可以使用脚本、指标或 EA 交易来提高性能,该脚本、指标或 EA 交易对多个符号和时间范围的历史数据进行复杂而冗长的分析,并且每个符号和时间范围的计算都不依赖于其他符号和时间范围。

与此同时,初学者经常会有这样一个问题,即是否可以使用 OpenCL 来加速 EA 交易的测试和优化。答案是否定的。测试重现了连续交易的真实过程,因此每下一根柱线或逐笔交易明细都取决于前一根柱线或逐笔交易明细的结果,这使得一次计算无法并行化。至于优化,测试程序的代理只支持 CPU 核心。这是由于报价或逐笔交易明细的全面分析、跟踪头寸和计算余额及权益的复杂性。而如果不考虑复杂性,则可以将所有模拟交易环境的计算和所需的可靠性转移到 OpenCL,从而在显卡核心上实现自己的优化引擎。

OpenCL 的意思是开放计算语言。它类似于 C 和 C++ 语言,因此也类似于 MQL5。然而,为了编写(即“编译”)一个 OpenCL 程序,将输入数据传递给这个程序,并在多个核心上并行运行程序,获取计算结果,则需要使用一个特殊的编程接口(一组函数)。这个 **OpenCL API** 也可用于希望实现并行执行的 MQL 程序。

要使用 OpenCL,没有必要在电脑上安装显卡,因为有一个中央处理器就足够了。在任何情况下,都需要制造商提供的特殊驱动程序(要求 OpenCL 1.1 版及更高版本)。如果你的电脑装有使用了显卡的游戏或其他软件(例如,科技、视频编辑器等),那么必要的软件层很可能已经可用。这可以通过使用 OpenCL 调用在终端中运行 MQL 程序来检查(至少是终端交付的一个简单例子,参见下文)。

如果没有 OpenCL 支持,你将在日志中看到错误。

OpenCL OpenCL not found, please install OpenCL drivers

如果你的电脑上装有合适的设备,并且已启用 OpenCL 支持,则终端将显示一条消息,显示该设备(也可能有多个设备)的名称和类型。例如:

**OpenCL Device #0: CPU GenuineIntel Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz with OpenCL
OpenCL Device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 2.1**

为各种设备安装驱动程序的过程见 [mql5.com](#) 文章中的说明。。可支持来自 Intel、AMD、ATI 和 Nvidia 的当下最流行的设备。

在核心数量和分布式计算速度方面,中央处理器的性能明显不如显卡,但一个好的多核中央处理器就足以显著提高性能。

重要事项:如果你的计算机装有支持 OpenCL 的显卡,那么就不需要在 CPU 上安装 OpenCL 软件模拟!

OpenCL 设备驱动程序自动化核心之间的计算分布。例如,如果你需要对不同向量执行一百万次相同类型的计算,而只有一千个核心可用,那么当前一个任务准备就绪并且核心被释放时,驱动程序将自动启动下一个任务。

使用上述 OpenCL API 的函数,在 MQL 程序中设置 OpenCL 运行时环境的准备操作仅执行一次。

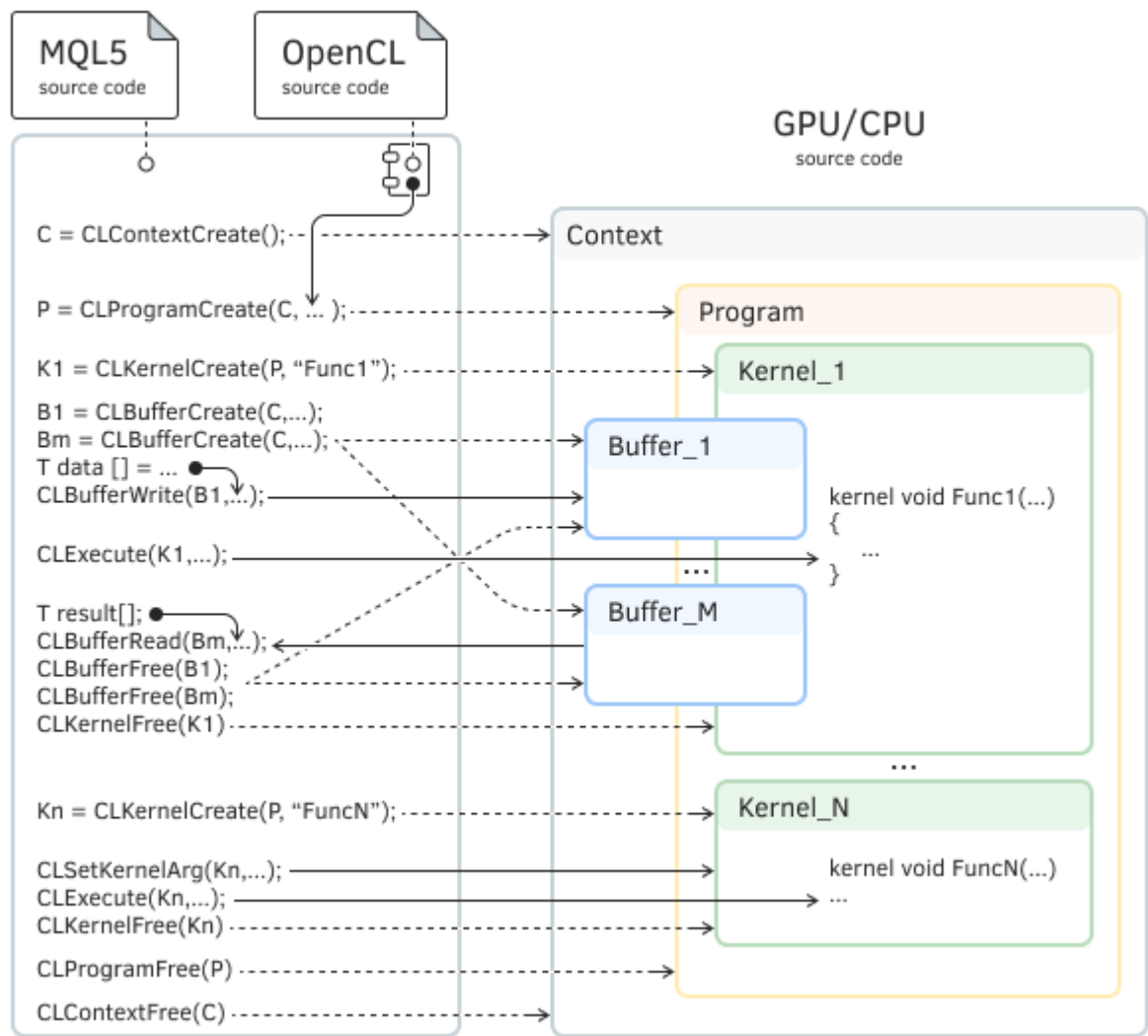
1. 为 OpenCL 程序创建上下文 (选择设备, 如显卡、CPU 或任何可用设备): **CLContextCreate(CL_USE_ANY)**。该函数将返回一个上下文描述符 (一个整数, 我们将其表示为有条件的 **ContextHandle**)。
2. 在接收到的上下文中创建 OpenCL 程序: 使用 **CLProgramCreate** 函数调用, 基于 OpenCL 语言的源代码进行编译, 代码的文本通过参数 **Source :CLProgramCreate(ContextHandle, Source, BuildLog)** 传递给该函数。该函数将返回程序句柄 (整数 **ProgramHandle**)。这里需要注意的是, 在该程序的源代码内部, 必须包含标有特殊关键字 **__kernel** (或 **kernel**) 的函数 (至少一个): 它们包含要并行化的算法部分 (参见下面的例子)。当然, 为了简化 (分解源代码), 程序员可以将内核函数的逻辑子任务划分到其他辅助函数中, 并从内核中调用: 同时, 不需要用 **kernel** 这个词来标记辅助函数。
3. 按 OpenCL 程序代码中标记为内核形成的函数的名称注册要执行的内核: **CLKernelCreate(ProgramHandle, KernelName)**。调用该函数将返回一个内核句柄 (一个整数, 比如 **KernelHandle**)。你可以在 OpenCL 代码中创建多个不同的函数, 并将它们注册为不同的内核。
4. 如有必要, 为按引用传递给内核的数据数组, 以及为返回值/数组创建缓冲区: **CLBufferCreate(ContextHandle, Size * sizeof(double), CL_MEM_READ_WRITE)**, 等。缓冲区也通过描述符进行标识和管理。

然后, 如有必要 (例如, 在指标或 EA 交易事件处理程序中), 将根据以下方案直接执行一次或多次计算:

- I. 使用 **CLSetKernelArg(KernelHandle,...)** 和/或 **CLSetKernelArgMem(KernelHandle,..., BufferHandle)** 传递输入数据和/或绑定输入/输出缓冲区。第一个函数提供标量值的设置, 第二个函数相当于按引用传递或接收值 (或值的数组)。在此阶段, 数据从 MQL5 迁移到 OpenCL 执行核心。**CLBufferWrite(BufferHandle,...)** 将数据写入缓冲区。内核执行期间, OpenCL 程序将可以使用参数和缓冲区。
- II. 通过调用特定内核 **CLExecute(KernelHandle,...)** 执行并行计算。内核函数能够将其工作结果写入输出缓冲区。
- III. 使用 **CLBufferRead(BufferHandle)** 获取结果。在此阶段, 数据从 OpenCL 移回到 MQL5。

计算完成后, 应释放所有描述符: **CLBufferFree(BufferHandle)**、**CLKernelFree(KernelHandle)**、**CLProgramFree(ProgramHandle)** 和 **CLContextFree(ContextHandle)**。

该顺序通常如下图所示。



MQL 程序和 OpenCL 附件之间的交互方案

建议在单独的文本文件中编写 OpenCL 源代码，然后使用 资源变量将文件连接到 MQL5 程序。

终端提供的标准头文件库包含一个用于 OpenCL 的包装类: *MQL5/Include/OpenCL/OpenCL.mqh*。

OpenCL 的使用示例可以在 *MQL5/Scripts/Examples/OpenCL/* 文件夹中找到。特别是 *MQL5/Scripts/Examples/OpenCL/Double/Wavelet.mq5* 脚本，它产生时间序列的小波变换（你可以根据随机 Weierstrass 模型或当前金融工具的价格增量取一条人工曲线）。在任何情况下，算法的初始数据都是一个数组，它是一组二维图像。

运行这个脚本时，与运行任何其他包含 OpenCL 代码的 MQL 程序一样，终端将选择最快的设备（如果有多个设备，并且程序没有选定特定设备，或者之前没有定义特定设备的话）。有关这方面的信息显示在 *Journal* 选项卡（终端日志，而不是 EA 交易）中。

```

Scripts script Wavelet (EURUSD,H1) loaded successfully
OpenCL device #0: GPU NVIDIA Corporation NVIDIA GeForce GTX 1650 with OpenCL 3.0 (16
OpenCL device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 3.0
OpenCL device performance test started
OpenCL device performance test successfully finished
OpenCL device #0: GPU NVIDIA Corporation NVIDIA GeForce GTX 1650 with OpenCL 3.0 (16
OpenCL device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 3.0
Scripts script Wavelet (EURUSD,H1) removed

```

作为执行的结果，脚本以通常的方式（在 CPU 上以串行方式）和并行方式（在 OpenCL 内核上）在 **Experts** 选项卡中显示计算速度测量记录。

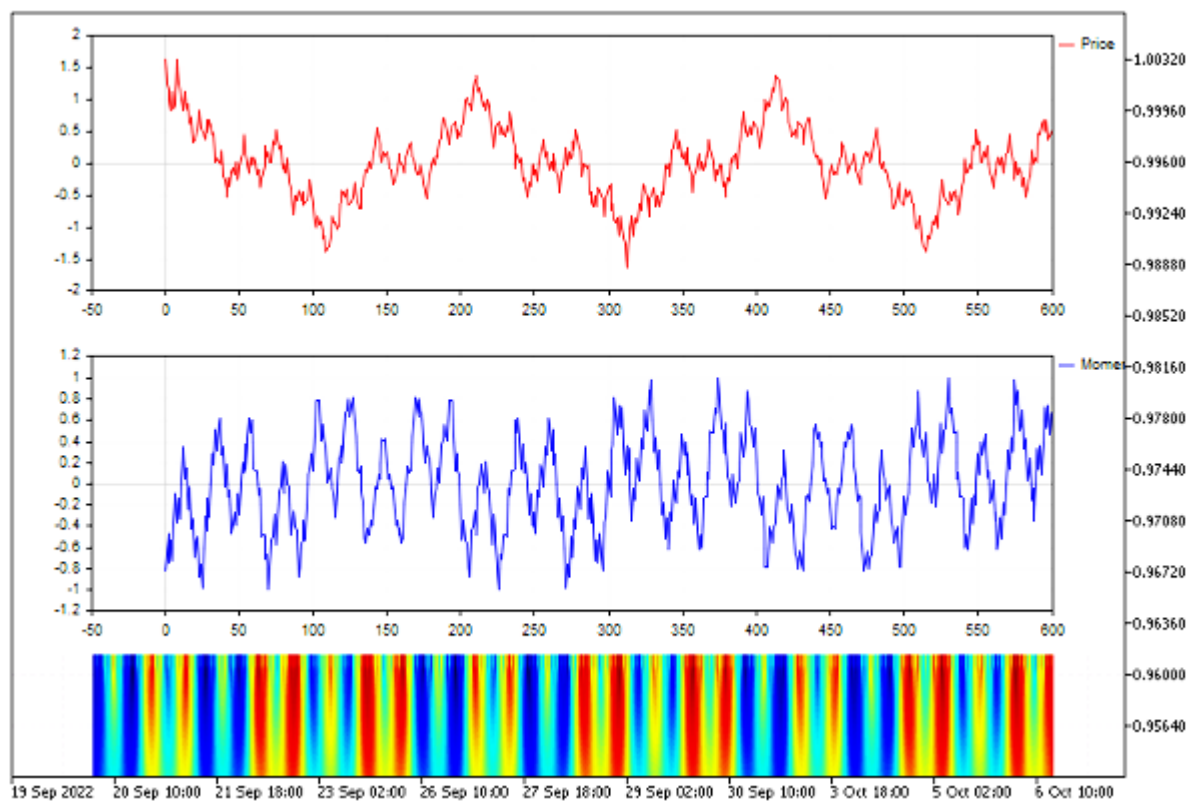
```

OpenCL: GPU device 'Intel(R) UHD Graphics 630' selected
time CPU=5235 ms, time GPU=125 ms, CPU/GPU ratio: 41.880000

```

速度比，根据任务的具体情况，可以达到几十。

该脚本在图表上显示原始图像、其增量形式的导数以及小波变换的结果。



原始模拟序列、其增量以及小波变换

请注意，在脚本完成工作后，图形对象仍保留在图表上。需要手动删除它们。

以下是小波变换的 `OpenCL` 源代码示例，在一个单独的文件 `MQL5/Scripts/Examples/OpenCL/Double/Kernels/wavelet.cl` 中实现。

```

// 需要增加两倍的计算精度
// 默认情况下, 如果没有这个指令, 我们会得到浮点数)
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

//辅助功能小程序
double Morlet(const double t)
{
    return exp(-t * t * 0.5) * cos(M_2_PI * t);
}

// OpenCL kernel函数
__kernel void Wavelet_GPU(__global double *data, int datacount,
    int x_size, int y_size, __global double *result)
{
    size_t i = get_global_id(0);
    size_t j = get_global_id(1);
    double a1 = (double)10e-10;
    double a2 = (double)15.0;
    double da = (a2 - a1) / (double)y_size;
    double db = ((double)datacount - (double)0.0) / x_size;
    double a = a1 + j * da;
    double b = 0 + i * db;
    double B = (double)1.0;
    double B_inv = (double)1.0 / B;
    double a_inv = (double)1.0 / a;
    double dt = (double)1.0;
    double coef = (double)0.0;

    for(int k = 0; k < datacount; k++)
    {
        double arg = (dt * k - b) * a_inv;
        arg = -B_inv * arg * arg;
        coef = coef + exp(arg);
    }

    double sum = (float)0.0;
    for(int k = 0; k < datacount; k++)
    {
        double arg = (dt * k - b) * a_inv;
        sum += data[k] * Morlet(arg);
    }
    sum = sum / coef;
    uint pos = (int)(j * x_size + i);
    result[pos] = sum;
}

```

有关 OpenCL 语法、内置函数和操作原理的完整信息, 请访问 [科纳斯组织的官方网站](#)。。

特别有趣的是, OpenCL 不仅支持常见的标量数字数据类型 (从 `char` 开始, 以 `double` 结束), 还支持 `(u)charN`, `(u)shortN`, `(u)intN`, `(u)longN`, `floatN`, `doubleN` 向量, 其中 $N = \{2|3|4|8|16\}$, 表示向量的长度。在本例中, 没有使用这种方法。

除了上面提到的 *kernel* 关键字之外, *get_global_id* 函数在并行计算的组织中起到一个重要的作用:它允许你在代码中找到当前正运行的计算子任务的编号。很明显,不同子任务中的计算应该是不同的(否则使用多个核心就没有意义了)。在本例中,由于任务涉及二维图像的分析,使用两个正交坐标来识别其片段会更加方便。在上面的代码中,我们使用两个调用 *get_global_id(0)* 和 *get_global_id(1)* 来获取它们。

实际上,我们在调用 MQL5 函数 *CLEExecute* 时为任务设置了数据维度(参见下文)。

在 *Wavelet.mq5* 文件中,OpenCL 源代码包含在指令中:

```
#resource "Kernels/wavelet.cl" as string cl_program
```

图像大小通过宏设置:

```
#define SIZE_X 600
#define SIZE_Y 200
```

为了管理 OpenCL,使用了带有 *COpenCL* 类的标准库。它的方法具有相似的名称,并且在内部使用 MQL5 API 中对应的内置 OpenCL 函数。建议先熟悉一下这部分内容。

```
#include <OpenCL/OpenCL.mqh>
```

以一种简化的形式(没有错误检查和可视化),启动变换的 MQL 代码如下所示。*CWavelet* 类中总结了与小波变换相关的操作。

```
class CWavelet
{
protected:
    ...
    int      m_xsize;           //沿轴的图像大小
    int      m_ysize;
    double   m_wavelet_data_GPU[]; //结果在这里显示
    COpenCL  m_OpenCL;         //包装对象
    ...
};
```

通过 *CalculateWavelet_GPU* 方法组织主要的并行计算。

```

bool CWavelet::CalculateWavelet_GPU(double &data[], uint &time)
{
    int datacount = ArraySize(data); //图像大小 (点数)

    //根据源代码编译cl程序
    m_OpenCL.Initialize(cl_program, true);

    //从 cl 文件注册一个 kernel 函数
    m_OpenCL.SetKernelsCount(1);
    m_OpenCL.KernelCreate(0, "Wavelet_GPU");

    //注册两个缓存器, 进行输入和输出数据, 写入输入数组 register 2 buffers for input
    m_OpenCL.SetBuffersCount(2);
    m_OpenCL.BufferFromArray(0, data, 0, datacount, CL_MEM_READ_ONLY);
    m_OpenCL.BufferCreate(1, m_xsize * m_ysize * sizeof(double), CL_MEM_READ_WRITE);
    m_OpenCL.SetArgumentBuffer(0, 0, 0);
    m_OpenCL.SetArgumentBuffer(0, 4, 1);

    ArrayResize(m_wavelet_data_GPU, m_xsize * m_ysize);
    uint work[2]; //分析一个二维图形的任务 - 即维度 2
    uint offset[2] = {0, 0}; //从最开头开始 (或者你可以跳过一些内容)
    work[0] = m_xsize;
    work[1] = m_ysize;

    //设置输入数据
    m_OpenCL.SetArgument(0, 1, datacount);
    m_OpenCL.SetArgument(0, 2, m_xsize);
    m_OpenCL.SetArgument(0, 3, m_ysize);

    time = GetTickCount(); //速度测量的切断时间
    //开始在 GPU 上进行计算, 二维任务
    m_OpenCL.Execute(0, 2, offset, work);

    //将结果存入输出缓存器
    m_OpenCL.BufferRead(1, m_wavelet_data_GPU, 0, 0, m_xsize * m_ysize);

    time = GetTickCount() - time;

    m_OpenCL.Shutdown(); //释放所有资源 - 调用所有必要函数 CL***Free
    return true;
}

```

在示例的源代码中, 有一个注释掉的行调用 *PreparePriceData* 用于创建一个基于实际价格的输入数组: 你可以用 *PrepareModelData* 调用 (该调用生成一个对数) 激活它, 而不是前面的行。

```

void OnStart()
{
    int momentum_period = 8;
    double price_data[];
    double momentum_data[];
    PrepareModelData(price_data, SIZE_X + momentum_period);

    // PreparePriceData("EURUSD", PERIOD_M1, price_data, SIZE_X + momentum_period);

    PrepareMomentumData(price_data, momentum_data, momentum_period);
    ... //系列和增量的可视化
    CWavelet wavelet;
    uint time_gpu = 0;
    wavelet.CalculateWavelet_GPU(momentum_data, time_gpu);
    ... //小波变换结果的可视化
}

```

一组特殊的错误代码 (带有 `ERR_OPENCL_` 前缀, 从代码 `5100` 开始, `ERR_OPENCL_NOT_SUPPORTED`) 已分配给 `OPENCL` 操作。这些代码在 [帮助中说明](#)。如果 `OpenCL` 程序的执行出现问题, 则终端会将详细的诊断信息输出到日志, 展示错误代码。

结语

本节是本书的结尾。在七大主题和众多章节的学习过程中，我们探索了 MQL5 编程的各个方面，从语言基础内容开始，逐步深入到相关的复杂技术，让使读者能够从创建单一的交易员专用工具，逐步掌握复杂交易系统和产品的开发。

你获得的知识将帮助你实现自己的各种想法，并让你在专业算法交易领域取得成功：

- 开发应用程序并通过 [市场](#) 进行销售，“市场”是最大的 MetaTrader 程序商店，为作者提供了现成的基础设施。用户可以通过“市场”接触庞大的受众，并提供产品保护、许可及集成式收款系统。
- 通过 [自由职业者](#) 功能开发定制应用程序。可以查看所有开发订单，享受便捷的工作系统和支付保障。
- 在 [代码库](#) 发布代码，从而分享你的经验。向 MQL5 社区的数千名交易员展示你的程序。

当然，也要不断学习。我们的 www.mql5.com 网站提供了丰富的信息和现成的算法：

- [编程文章](#) 在这个板块中，专业的程序编写者会给出解决实际问题的方法。
- [论坛](#) 在这个板块中，你可以交流经验并向其他开发者寻求建议。
- [代码库](#) 包含程序源代码，可助你了解 MQL5 语言功能和创建自己的程序。

最后我想说的是，软件开发不仅涉及编程，还涉及许多其他同样重要的领域：编写技术规范（即使只是为自己编写）、设计、原型制作、用户界面创建、提供文档以及后续支持。所有这些方面都会显著影响你作为程序员的工作效率和最终成果的质量。

特别是，大多数实际任务都可以分解为不同语言程序员长期使用的标准算法和原理。包括设计模式、为特定任务优化的数据结构集合以及用于自动化开发的工具。MetaTrader 5 平台中会用到所有这些方面，同时要借助 MQL5 以及其他工具。而这本书只是你专业发展道路上的第一步。

请加入交易机器人开发者社区 www.mql5.com !

