

MagicQuant

策略开发指南

2016 年 9 月 1 日

文档信息

文档名称: MagicQuant 策略开发指南

版本号: 4.3.5

作者: 梁涛

目录

1	MagicQuant 简介	7
1.1	概述	7
1.2	升级变化	8
1.3	特色功能	9
1.4	注册登录	10
1.5	平台架构	11
1.6	下载安装	13
1.7	帮助体系	15
1.8	目录迁移	15
1.9	界面布局	16
1.10	数据管理	18
1.11	盘后下载	27
1.12	交易图表	28
1.13	账户管理	31
1.14	策略管理	34
1.15	工程管理	40
1.16	实盘交易	45
2	策略语言	49
2.1	面向对象	49
2.2	基本语法	50
2.3	数学函数	59
2.4	条件结构	61
2.5	循环逻辑	63
2.6	高级类型	67
2.7	方法	78
3	核心概念	79
3.1	策略	79
3.2	Tick	80
3.3	Bar	82
3.4	事件驱动	84
3.5	线程安全	86
3.6	策略运行	87
3.7	代码结构	94
3.8	事件种类	106
3.9	委托拒绝处理	110
3.10	委托状态	112
3.11	事件流图	113
3.12	事件隔离	114
4	策略开发	115
4.1	品种	115
4.2	时间	120
4.3	夜盘	122
4.4	行情	123
4.5	回溯	129
4.6	账户	136
4.7	查询机理	139
4.8	本地 GUID	141

4.9	委托.....	142
4.10	成交.....	144
4.11	持仓.....	146
4.12	交易.....	151
4.13	交互.....	158
4.14	全局字典.....	159
4.15	采样.....	161
4.16	数学.....	163
5	指标.....	164
5.1	指标概述.....	164
5.2	指标用法.....	166
5.3	MA	167
5.4	EMA	168
5.5	MACD.....	169
5.6	Bolling.....	170
5.7	自定义指标.....	171
6	复盘.....	175
6.1	概述.....	175
6.2	交易成本.....	176
6.3	策略评价.....	177
6.4	参数调优.....	180
6.5	Tick 回放.....	181
6.6	K 线回放.....	182
7	调试.....	183
7.1	日志调试.....	183
7.2	断点调试.....	184
7.3	弹出消息.....	186
8	通道配置.....	187
8.1	通道概述.....	187
8.2	LTS	188
8.3	万得 TDF	190
8.4	CTP.....	191
9	策略范例.....	192
9.1	日内波动.....	192
9.2	红三兵.....	194
9.3	观察区间突破.....	198
9.4	日内开盘突破.....	201
9.5	包 Package.....	204
9.6	R-Break	204
9.7	Dural Trust.....	204
9.8	轴枢点.....	204
9.9	多参数混合	204
9.10	多周期混合	204
9.11	多策略混合	205
9.12	股票 T+0 策略模板	205
9.13	股票 Alpha 策略	205
9.14	股票实盘策略.....	205
9.15	股票动量策略.....	205
9.16	唐奇安通道.....	206
9.17	双均线.....	206

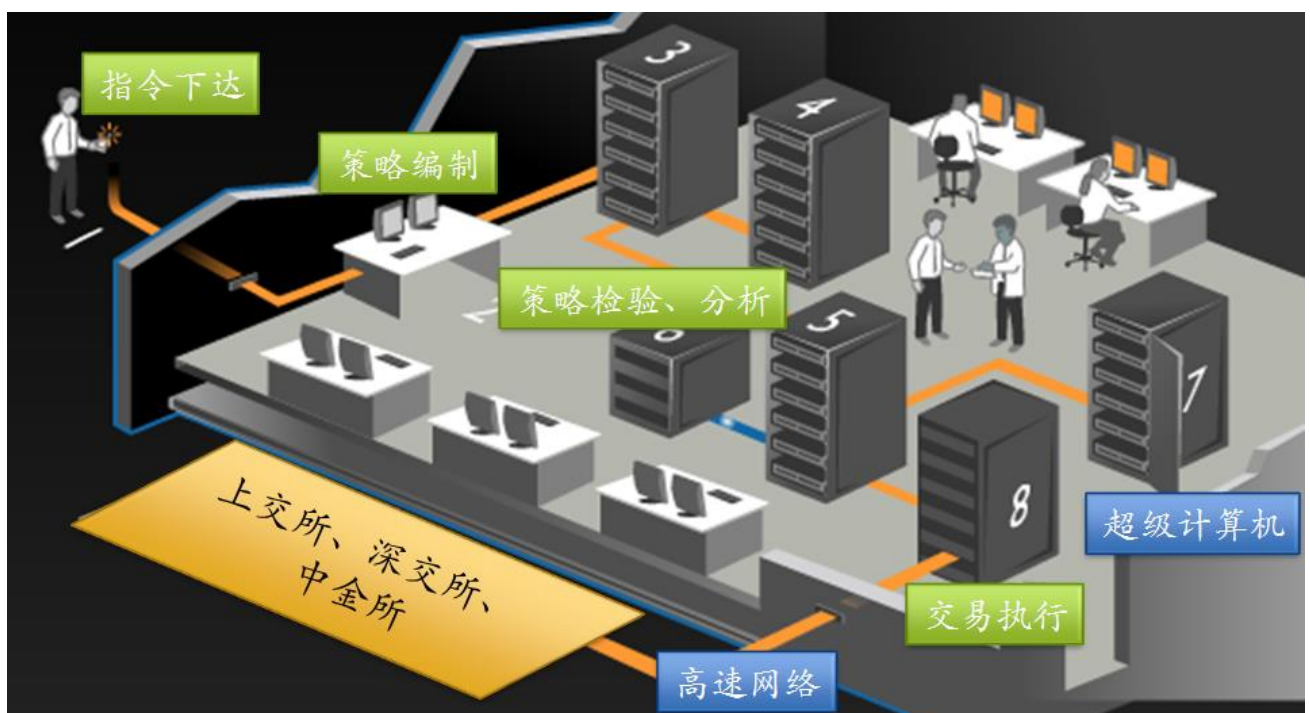
9.18	三均线.....	206
9.19	菲阿里四价.....	206
9.20	Aberration.....	206
9.21	横盘突破系统.....	207
9.22	空中花园.....	207
9.23	金肯特纳交易系统.....	207
9.24	布林强盗系统.....	207
9.25	动态突破系统.....	208
9.26	幽灵交易者.....	208
9.27	恒温器策略.....	208
9.28	网格交易策略.....	208
9.29	鳄鱼法则策略.....	210
9.30	做市商策略.....	210
9.31	多品种扫描.....	210
9.32	VWAP 拆单.....	211
9.33	撤单和追单.....	211
9.34	平仓及时反手.....	212
9.35	期现套利.....	213
9.36	Alpha 对冲.....	213
9.37	Beta 增强型对冲.....	214
9.38	区间突破止盈.....	215
9.39	布林带穿越.....	219
10	功能扩展.....	220
10.1	引用 DLL.....	220
10.2	数据库.....	221
10.3	辅助工具-统计盈亏.....	223
10.4	在 MQ 中使用 Python.....	225
10.5	在 MQ 中使用 Matlab.....	229
10.6	在 MQ 中使用 R.....	231
10.7	在 MQ 中下载结算单.....	231
10.8	提取数据子集.....	232
10.9	异常处理.....	234
10.10	实盘人工确认.....	235
10.11	限制策略使用.....	236
10.12	盘中调整参数.....	237
10.13	灾难恢复.....	238
11	常见问题.....	239
11.1	MQ 如何收费.....	239
11.2	MQ 支持哪些期货公司.....	241
11.3	怎样自己配置期货公司.....	241
11.4	GetBarSeries 和 OnBar.....	241
11.5	怎样进一步提高速度.....	242
11.6	不合法的登录.....	242
11.7	不能输出类库.....	242
11.8	刚开通 CTP 却收不到事件.....	242
11.9	怎么避免开盘前的委托.....	243
11.10	怎么获取 Tick 数据复盘.....	243
11.11	如何使用股票程序化交易功能.....	243
11.12	如何使用 UDP 行情.....	243

11.13	策略显示已经下单但平台并没有实际下单	244
11.14	股指期货当天平今委托的成交回报为什么不是平今?.....	245
11.15	集合已修改；可能无法执行枚举操作?	245
11.16	为什么 3852>3852?	246
11.17	创建工程时出错，实例化策略失败?	247

1 MagicQuant 简介

1.1 概述

金融工程主要分为四类工作：一是基于数学统计模型进行程序化交易；二是通过高频算法交易降低交易成本；三是对各种金融产品进行定价；四是量化风险管理。作为一款针对于国内股票和期货市场开发的量化交易平台，MagicQuant（以下简称 MQ）主要完成的是前两个任务。

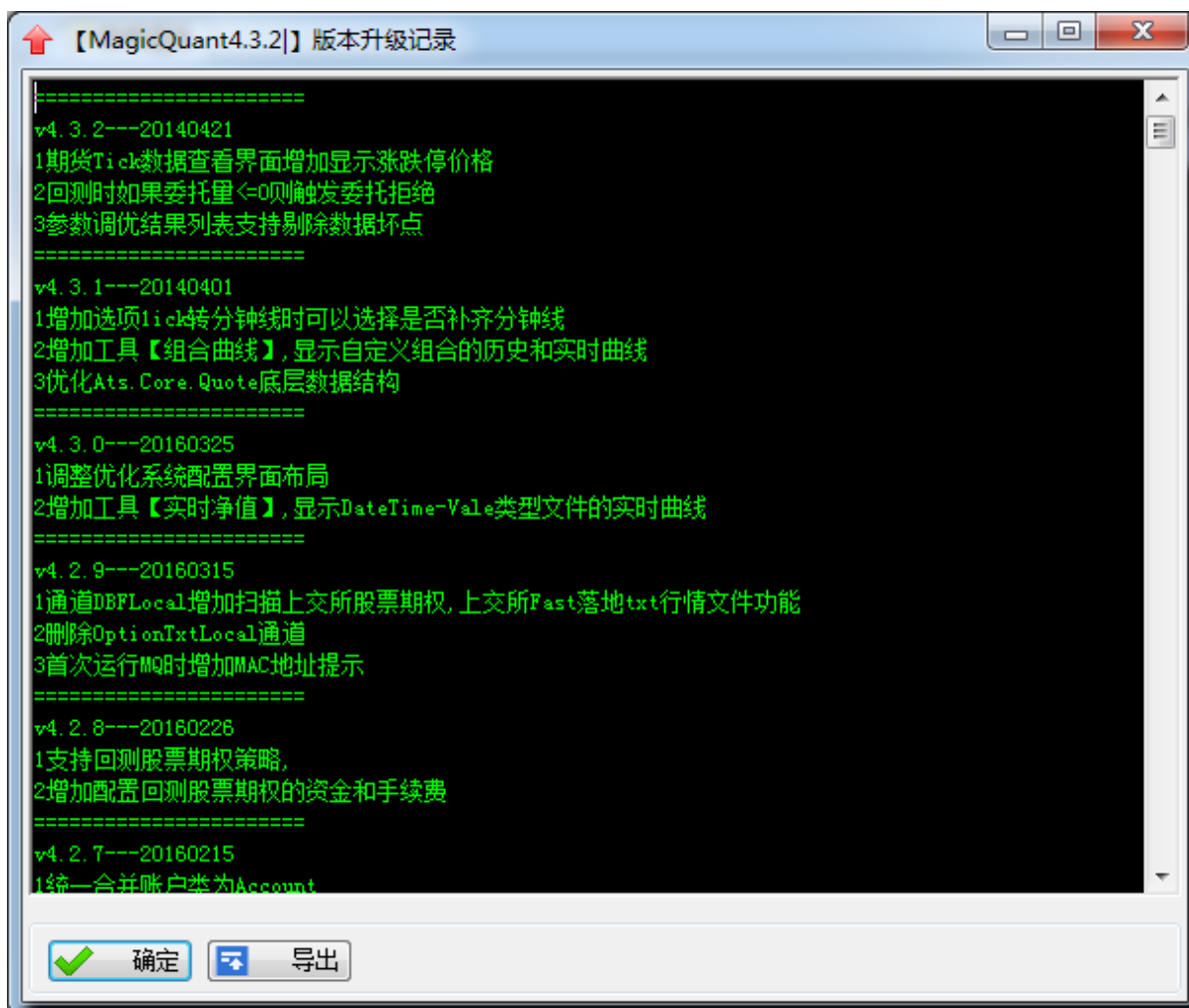


具体来说，MQ 软件可以帮助金融工程师实现了量化策略的建模、回测、调优、运营、实盘交易的全生命周期。MQ 的功能覆盖量化投机、算法交易、高频交易、策略研究、套利、对冲以及趋势交易等。MQ 致力于帮助金融工程师研究、开发并高效地执行一个最终可用于实盘的数量化策略。该平台将为金融工程师提供无以伦比的策略开发和交易体验。

本文档首先介绍 MQ 软件的使用操作，然后讲述 C# 策略语言的基本语法，接着介绍策略相关的核心概念，在上述知识框架的基础上重点介绍如何开发策略和指标，然后介绍复盘和调试功能，最后通过一些实际的例子介绍 MQ 的功能扩展和策略范例。

1.2 升级变化

MQ 的升级变化参考软件【菜单】→【帮助】→【关于】→【版本升级记录】。



1.3 特色功能

➤ 策略安全

MQ 始终将确保策略安全放在最高优先的位置，通过核级加密、不可逆文件混淆以及用户自定义密码等措施对策略文件施加最严格的安全保护。

➤ 事件驱动架构

与传统的程序化交易平台不同的是，**MQ** 采用了先进的“事件驱动”架构，从而迅捷地对各种复杂的市场事件（例如 **Tick** 价格变化、成交回报、撤单回报等事件）在第一时间作出反应。

➤ Tick 级别的高精度复盘

独特的复盘引擎结合 **MQ** 提供的海量历史数据可以进行 Tick 级别的高精度复盘，从而尽可能真实的还原策略在历史中的表现，为实盘交易提供一个坚实的实验基础。

➤ 多策略多账户高效并行

MQ 支持多个策略、多个账户的高效并行运行，方便管理多个交易账户。

➤ .NET 扩展

基于目前微软强大的 **.NET** 平台，基于 **MQ** 开发的策略可以最大范围的扩展功能。

➤ 股票期货对冲

作为对冲基金进行资产组合管理的利器，**MQ** 同时打通了国内期货和现货市场的交易通道，可以高效灵活地运行套利、对冲策略。

1.4 注册登录

➤ 注册

申请 SN 请发送邮件到客服邮箱 services@magicquant.com

申请 SN 需要计算机的 MAC 地址，Mac 地址获取方法请下载 QQ 群（233309671）共享内 MAC 地址查看工具

SN 文件请放置在 MQ 根目录，登陆账号为 admin 密码为 123456

➤ 登录

普通登录需要输入用户名和密码，服务器验证成功登录后就进入 MQ 的主界面了。登录时注意根据您的网络实际情况，选择【电信】或者【联通】。

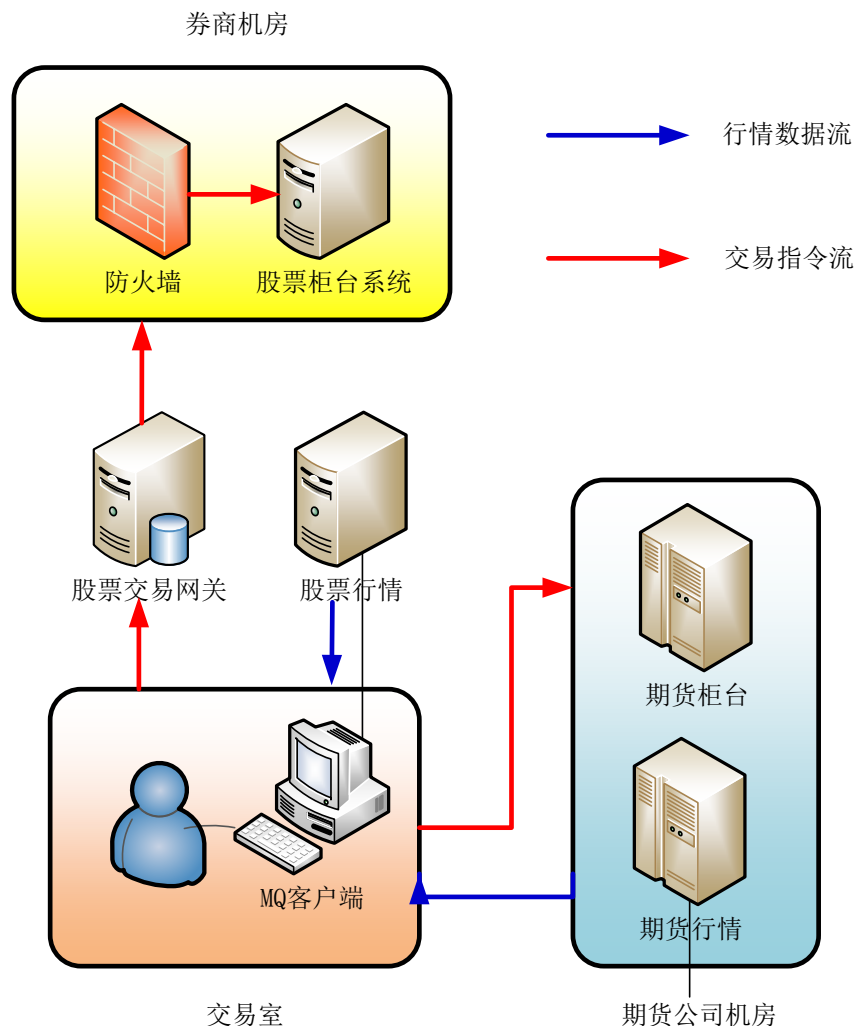
1.5 平台架构

➤ 网络拓扑图

下图是 MQ 的网络拓扑图。MQ 内部集成了国内主流券商、期货经纪商柜台系统的接口，可以轻松无缝地在国内的各大券商、期货经纪商直接进行交易。MQ 内嵌了股票和期货实时行情源，而且可以对股票全市场品种进行实时数据全推，策略可以最大宽度深度地扫描市场。

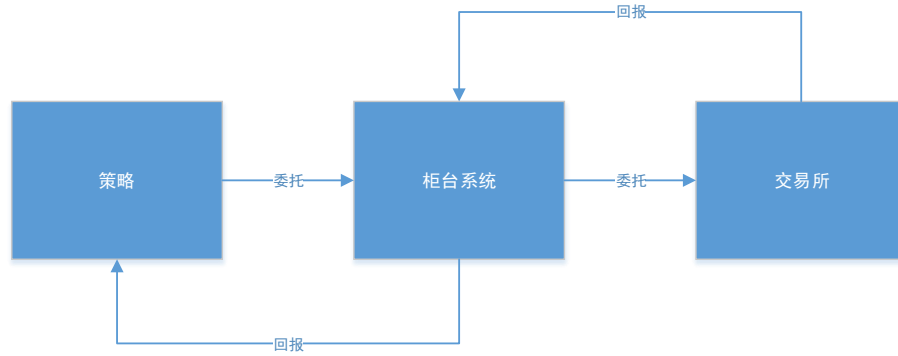
注意：由于目前国内大部分证券公司的柜台系统并不直接向互联网开放接口，因此如果想在实盘交易中使用 MQ 的股票程序化交易功能，需首先与开户的证券公司协商开通股票接口。如果没有开通股票接口，在实盘交易中调用股票交易的方法系统将抛出异常。

MQ 的期货程序化交易功能在任何支持 CTP 期货柜台系统的期货公司均可使用。



➤ 订单路由

如下图所示，订单从 MQ 量化终端中承载的策略 Strategy 中发出，通过适配器 Channel，直接发送到券商或者期货公司的柜台系统（不经过任何中间环节），经由柜台系统报送到交易所撮合主机。



1.6 下载安装

➤ 下载

MQ 主要需要安装 2 个程序包，一个是基础框架，另一个是量化终端。

模块	说明
.NET Framework 4.5.1	MQ 运行的基础框架（微软官方提供）
MagicQuant	MQ 量化终端

➤ MagicQuant

登录 MQ 主页：www.magicquant.com 然后进入【软件使用】栏目，在置顶的帖子中获取软件安装包的下载链接。下载软件安装包后进行安装。

也可以在官方网站的下载页面上下载 MQ：

<http://www.magicquant.com/bbx/download.html>



注意：该论坛是 MQ 的官方论坛，您有关于软件使用方面的问题都可以在这个论坛提问。该论坛只面向 MQ 用户开放，如果需要论坛账号请联系 MQ 管理员：

service@magicquant.com

➤ .NET Framework 4.5.1

由于 MagicQuant 是基于 Windows 平台和 .NET Framework 4.5.1 开发的，因此无法在 Linux 或者 Unix 环境下使用，只能在 Windows 平台下使用。

.NET Framework 4.5.1 的下载地址

<https://www.microsoft.com/zh-cn/download/details.aspx?id=40779>

➤ Visual Studio

MQ 本身不捆绑策略开发和编译的工具，用户使用微软提供的开发工具 [Visual Studio 2013](#) 开发编译策略。

➤ 网络端口

CTP 需要开放的端口各期货公司不同，具体可以参考 MQ 根目录的 **Config** 目录下 [ctpbrokers.xml](#)、[ltsbrokers.xml](#) 等配置文件中的端口。

1.7 帮助体系

MQ 的帮助体系主要包括

- 手册 [MagicQuant](#) 策略开发指南
- 单元测试
- [MagicQuant](#) 开发 API（[MagicQuant.CHM](#) 文件）
- 现场培训（需要可以联系 [MQ](#) 公司）
- QQ 群 [233309671](#) 交流

注意：在开发策略时，可以将 [MQ](#) 根目录的 [ats.core.xml](#) 文件拷贝到引用 [dll](#) 的目录下，这样开发策略时就有智能感知的效果，提高策略开发效率。

1.8 目录迁移

由于核心配置文件中 [atscore.xml](#) 中保存了路径相关信息，因此在对 [MQ](#) 进行目录迁移时，需要注意

如果将 [MQ](#) 移动了目录，需要手工删除根目录下的核心配置文件 [atscore.xml](#) 后，重启 [MQ](#)。重启时，[MQ](#) 会自动重建核心配置文件。此时再对系统进行配置，例如历史数据目录等即可完成 [MQ](#) 的目录移动工作。

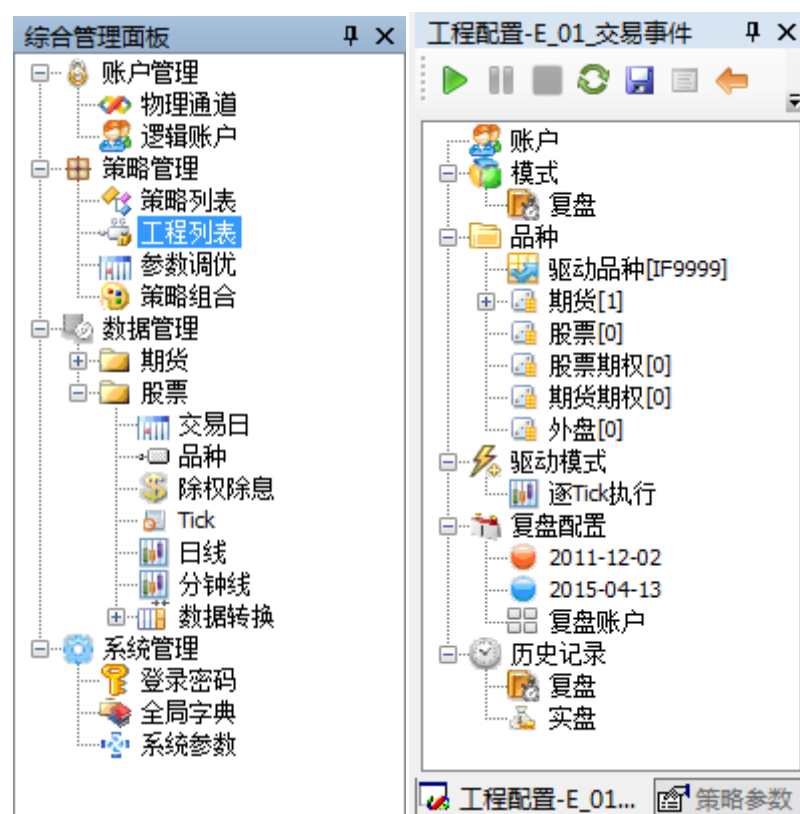
1.9 界面布局

MQ 采用经典的 Windows 程序布局方式，支持自由灵活的拖拽式界面操作，主要由菜单、工具栏、综合管理面板、主面板、消息面板以及工程配置构成。

菜单包括【文件】、【视图】、【工具】、【窗口】、【帮助】。

工具栏放置了一些常用的功能：【加载策略】、【新建工程】、【综合管理面板】、【交易账户】、【手工交易】、【开启实盘程序化交易】、【系统消息】。

综合管理面板中整合了常用的功能【账户管理】、【策略管理】、【工程管理】、【数据管理】、【系统管理】。



主面板是用户操作的主要区域，可以通过拖拽的方式动态调整界面布局。用户点击综合管理面板中的功能时，主面板就会出现对应的功能模块。

系统消息是 MQ 系统消息的提示窗口，这里可以看到系统的重要信息。

工程配置面板的功能是配置、启停工程、修改、导入导出策略参数。

MagicQuant3.2.5[专业版@zlling302]联通

文件 视图 工具 窗口 帮助 **菜单**

综合管理面板

- 账户管理
 - 物理通道
 - 逻辑账户
- 策略管理
 - 策略列表
 - 工程列表
 - 参数调优
 - 策略组合
- 数据管理
 - 期货
 - 股票
 - 交易日
 - 品种
 - 除权除息

主面板

工程

所有工程 新建工程

状态	工程
1 未加载	B_02测试最小价格不是...
2 未加载	E_01_交易事件

策略

导入策略

名称	路径
1 BaseTestStrategy	E:\MQ测试\20150305MG
2 T_01_显示Tick行情	E:\MQ测试\20150305MG
3 T_02_动态订阅退订	E:\MQ测试\20150305MG
4 Q_01_查询委托	E:\MQ测试\20150305MG
5 E_01_交易事件	E:\MQ测试\20150305MG
6 Q_04_查询成交	E:\MQ测试\20150305MG
7 D_01_加载本地Tick	E:\MQ测试\20150305MG
8 E_04_压力测试	E:\MQ测试\20150305MG
9 E_05_同时挂单	E:\MQ测试\20150305MG
10 Q_02_查询账户	E:\MQ测试\20150305MG
11 T_03_显示期权行情	E:\MQ测试\20150305MG
12 E_03_连续挂单撤单	E:\MQ测试\20150305MG
13 E_02_挂单用动态	E:\MQ测试\20150305MG

工程配置-E_01_交易事件

- 账户
 - 模式
 - 复盘
- 品种
 - 驱动品种[IF9999]
 - 期货[1]
 - 股票[0]
 - 股票期权[0]
 - 期货期权[0]
 - 外盘[0]
- 驱动模式
 - 逐Tick执行
- 复盘配置
 - 2011-12-02
 - 2015-04-13

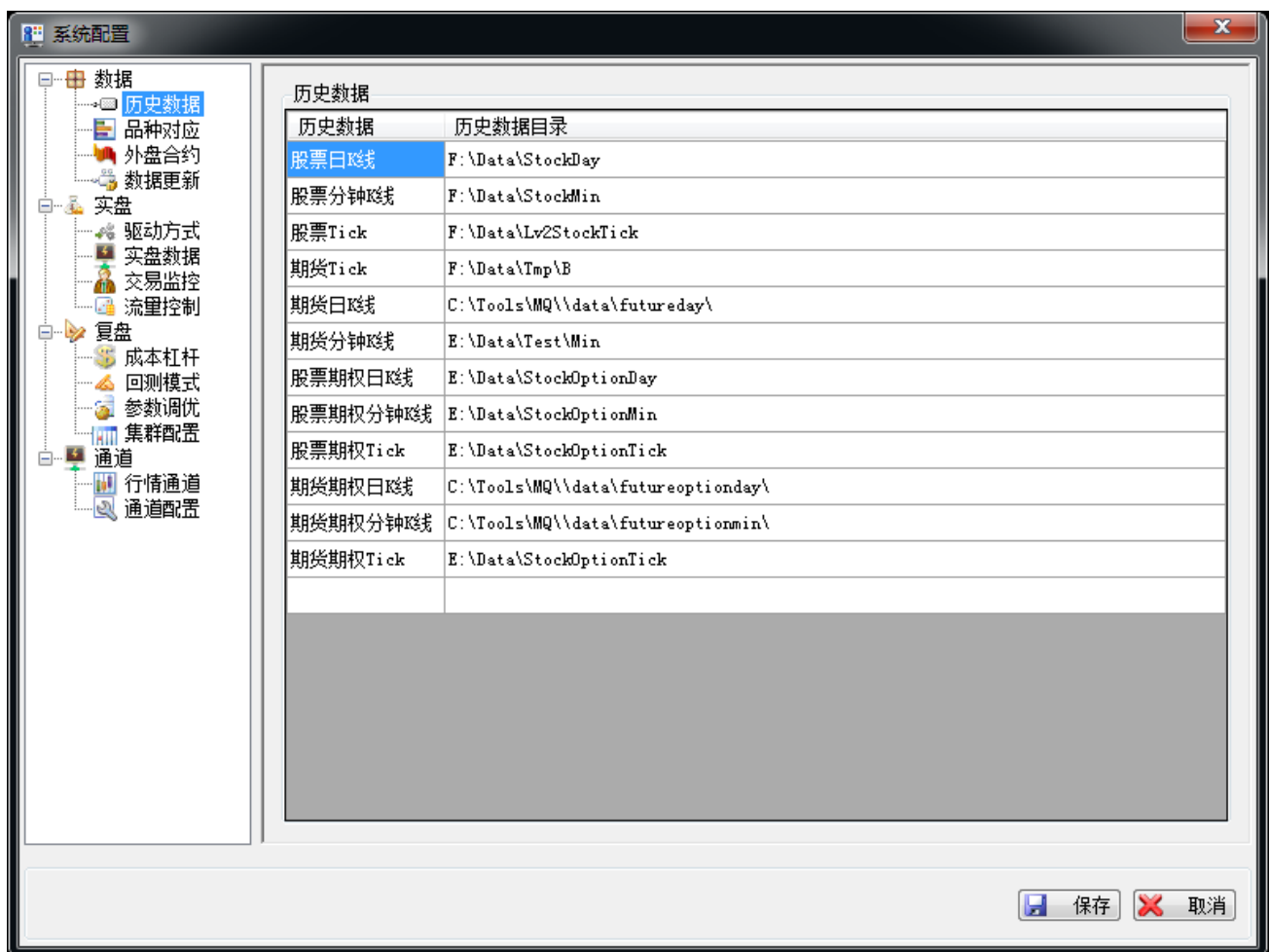
系统消息

工程配置面板

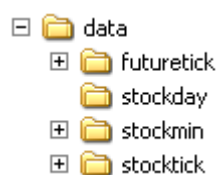
1.10 数据管理

➤ 数据路径配置

历史数据是保存在本地硬盘上的，主要包括 Tick、分钟线和日线。MQ 安装时自带的数
据，提供了部分重要期货和股票品种（股指期货、上证指数、深圳成指、沪深 300、深发展
和浦发银行等）部分时段的历史数据。这些数据位于 MQ 安装程序根目录下的 **data** 目录。通
过[工具-系统配置-数据路径](#)可以重新设置数据目录。



数据文件夹 **data** 是存放历史 K 线和 Tick 数据的文件夹：



其对应的数据内容如下表所示：

文件夹	内容	说明
Futuretick	期货 Tick	按照交易日组织，一天一个文件夹
FutureDay	期货日线	一个期货合约一个文件
FutureMin	期货分钟线	按照月组织，一月一个文件夹
Stockday	股票日线	一个股票的全部日线数据在一个单独的文件.day 中
StockMin	股票分钟线	按照月组织，一月一个文件夹
StockTick	股票 Tick	按照交易日组织，一天一个文件夹
stockoptionday	股票期权日线	
stockoptionmin	股票期权分钟线	
stockoptiontick	股票期权 Tick	
futureoptionday	期货期权日线	
futureoptionmin	期货期权分钟线	
futureoptiontick	期货期权 tick	

为控制安装文件大小，MQ 安装时自带的历史数据比较少。MQ 论坛的【数据中心】板块提供了免费的历史迷你数据包的下载。用户可以下载到本地后，解压缩到对应的文件夹。解压缩时请注意对应好文件夹的层次关系。数据中心网址：

<http://www.magicquant.com/forum.php?mod=forumdisplay&fid=50>

迷你数据包提供的内容包括股指期货 Tick 数据、核心股票的 Tick、分钟线、日线数据。核心股票指的是上证指数、深圳成指、沪深 300 指数、浦发银行以及深发展 5 个品种。

MQ 还提供了免费的主力合约 Tick 数据供交易者下载：

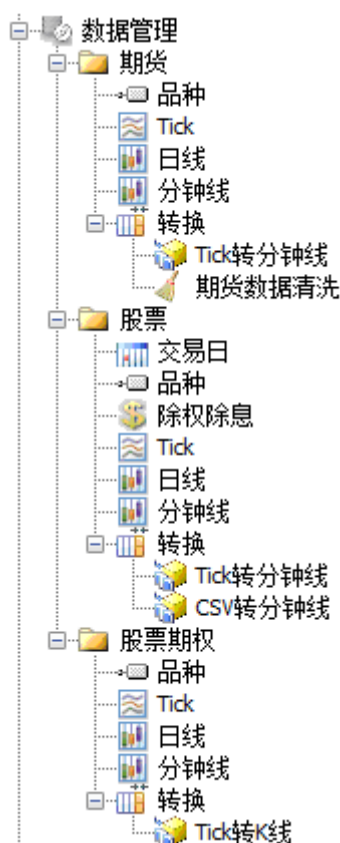
<http://www.magicquant.com/forum.php?mod=viewthread&tid=255&extra=page%3D1>

如果需要更多品种更长时间范围的历史数据，则需要联系 MQ 提供更多的历史数据包。

注意：如果某天不是交易日，但有一个空目录，会导致复盘无法通过，解决办法就是删除这个空的目录。

➤ 数据转换

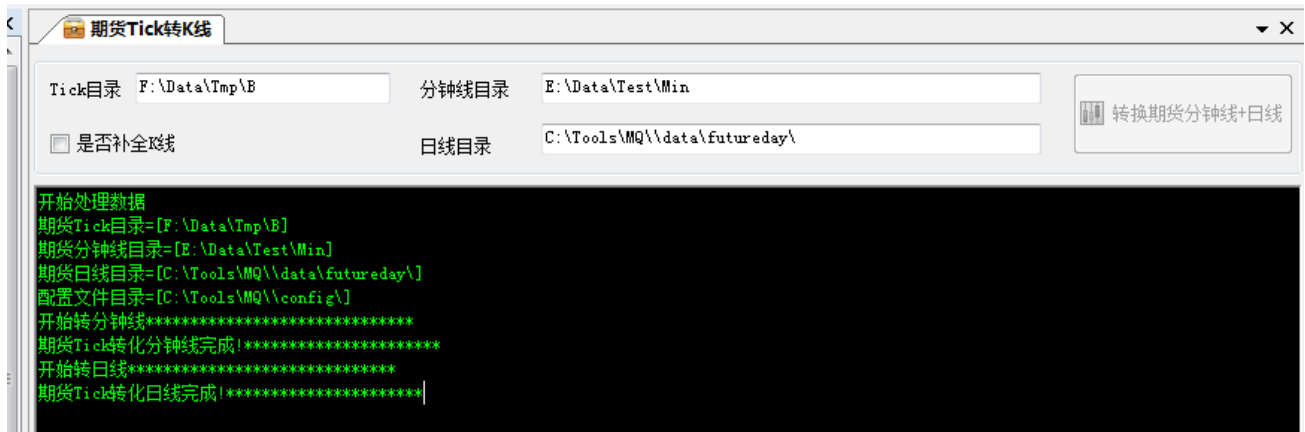
通过界面左侧导航栏的【数据管理】节点，可以对股票、期货以及股票期权数据进行管理。



➤ Tick 转分钟线

期货数据的 Tick 转换 K 线的功能是将本地期货的 Tick 文件转换成 1 分钟线和日线：

选项【是否补全 K 线】的含义是：如果某一分钟内由于行情清淡，成交稀少导致没有这 1 分钟的 K 线，会用上一分钟的收盘价来补全这一分钟的 K 线（如果没有选中，则不补全）。



➤ 期货数据清洗

期货数据清洗的功能是删除 Tick 数据中的坏点（例如开盘前和收盘后的异常数据、价格为 0 的数据点）。



其选项的含义分别为：

【全部清洗】删除原有的主力合约（9999）数据和索引 index 文件，然后重新构建；

【修复涨跌停价】根据昨结计算出涨跌停价格，如果原始 Tick 缺少涨跌停数据，则修补；

【强制修复涨跌停】不论原始 Tick 数据是否有涨跌停价格，都修补；

【无效文件删除】如果清洗后某个 tick 文件没有数据存在（全部为坏点或者是盘前盘后数据），则将该文件删除。

➤ 主力合约

MQ 将股指期货的主力合约、当月连续、下季连续以及指数合约等都单独处理好了，以便于进行长时间段的连续复盘。

注意：IF9999 这类数据是按照成交量最大的原则将数据拼接而成的，对于日内策略可以直接采用，但是对于隔夜策略在换仓日存在跳空的问题，从而影响复盘绩效。在使用主力合约进行复盘时，一个简便的处理技巧是在换仓日提前平仓，主力合约切换以后再開同方向同数量仓位。

下表是股指期货的特殊代码表：

代码	含义
IF9999	主力合约
IF9990	指数合约（按照成交量加权）
IF9998	当月连续
IF9997	下月连续
IF9996	下季连续
IF9995	隔季连续

特殊代码在复盘时可以使用，只要订阅特殊代码品种即可。

实盘模式时，建议使用真实合约代码。如果直接使用 9999 交易时，MQ 会根据品种映射表将 9999 转换成真实合约代码报单出去，同时事件回报里携带的品种代码 `InstrumentID` 字段会是真实合约的代码。

点击【菜单】→【工具】→【系统配置】→【品种映射】可以配置特殊代码映射表。

系统配置

数据路径
高级参数
行情通道
品种映射
外盘合约

合约映射表

	指数合约	实际合约
▶	a9999	a1401
	ag9999	ag1312
	al9999	al1310
	au9999	au1312
	b9999	b1309
	c9999	c1401
	CF9999	CF401
	cu9999	cu1312
	zn9999	zn1312
	FG9999	FG401
	fu9999	fu1407
	IF9999	IF1310
	IF9998	IF1310
	IF9997	IF1311
	IF9996	IF1312

保存
取消

注意：实盘模式候注意使用实际品种的代码，例如 IF1605，而不要用 IF9999。

➤ 数据浏览

数据管理功能可以直接浏览股票和期货的历史数据。

期货数据可以查看品种、Tick、日线和分钟线。

打开综合管理面板，选中【数据管理】->【期货】->【Tick】，然后在右侧主面板中选择要查看的品种和日期，下面就可以显示历史 Tick 数据。

工程

期货Tick

中国金融交易所

a1509

2015年 1月12日

查询

图形

	时间	最新	现量	买1价	卖1价	买1量	卖1量	高	开	低	涨幅	成交量	成交额	持仓
1	21:00:09.330	4440	2	4438	4447	3	8	4440	4440	4440	-0.045	2	88800	6432
2	21:00:11.520	4440	0	4438	4447	3	4	4440	4440	4440	-0.045	2	88800	6432
3	21:00:11.940	4440	0	4438	4447	3	3	4440	4440	4440	-0.045	2	88800	6432
4	21:00:13.120	4440	0	4438	4447	3	5	4440	4440	4440	-0.045	2	88800	6432
5	21:00:15.660	4440	0	4439	4447	2	3	4440	4440	4440	-0.045	2	88800	6432
6	21:00:19.060	4440	0	4440	4447	2	3	4440	4440	4440	-0.045	2	88800	6432
7	21:00:23.290	4447	6	4440	4448	2	2	4447	4440	4440	0.1126	8	355620	6432
8	21:00:25.800	4448	2	4440	4448	2	1	4448	4440	4440	0.1351	10	444580	6432
9	21:00:26.300	4448	0	4442	4448	2	1	4448	4440	4440	0.1351	10	444580	6432
10	21:00:27.570	4448	2	4442	4451	2	1	4448	4440	4440	0.1351	12	533540	6432
11	21:00:31.170	4453	4	4442	4454	2	4	4453	4440	4440	0.2476	16	711620	6432
12	21:00:32.970	4453	0	4442	4454	3	4	4453	4440	4440	0.2476	16	711620	6432
13	21:00:33.430	4453	0	4445	4454	2	3	4453	4440	4440	0.2476	16	711620	6432
14	21:00:37.920	4453	0	4447	4454	3	3	4453	4440	4440	0.2476	16	711620	6432
15	21:00:41.430	4453	24	4447	4454	3	3	4453	4440	4440	0.2476	40	1780340	6432
16	21:00:42.060	4450	22	4447	4454	3	2	4454	4440	4440	0.1801	62	2759420	6430
17	21:00:44.910	4450	0	4445	4454	2	2	4454	4440	4440	0.1801	62	2759420	6430
18	21:00:45.840	4450	0	4445	4453	2	1	4454	4440	4440	0.1801	62	2759420	6430
19	21:00:53.330	4450	0	4443	4453	10	1	4454	4440	4440	0.1801	62	2759420	6430
20	21:00:53.830	4450	4	4445	4453	1	1	4454	4440	4440	0.1801	66	2937420	6430
21	21:00:55.370	4450	0	4445	4453	6	1	4454	4440	4440	0.1801	66	2937420	6430
22	21:00:55.890	4450	0	4445	4453	1	1	4454	4440	4440	0.1801	66	2937420	6430
23	21:00:56.130	4450	0	4445	4452	1	1	4454	4440	4440	0.1801	66	2937420	6430
24	21:00:57.630	4450	0	4445	4452	11	1	4454	4440	4440	0.1801	66	2937420	6430
25	21:00:58.270	4450	0	4445	4452	12	1	4454	4440	4440	0.1801	66	2937420	6430
26	21:00:58.930	4450	0	4443	4452	10	1	4454	4440	4440	0.1801	66	2937420	6430
27	21:00:59.610	4450	0	4444	4452	11	1	4454	4440	4440	0.1801	66	2937420	6430
28	21:01:01.680	4450	0	4444	4452	12	1	4454	4440	4440	0.1801	66	2937420	6430
29	21:01:02.480	4444	2	4444	4452	11	1	4454	4440	4440	0.045	68	3026300	6430
30	21:01:02.980	4444	0	4444	4452	2	1	4454	4440	4440	0.045	68	3026300	6430

共有 7151 个 Tick

股票数据可以查看品种、除权除息表、Tick、日线以及分钟线。

例如，选中【数据管理】->【股票】->【Tick】，然后在右侧主面板中选择要查看的品种和日期，下面就可以显示客户端本地保存的 Tick 数据。

注意: 在 MagicQuant 中, 股票代码必须加后缀以区分是上交所【.SH】还是深交所【.SZ】的品种。例如 000001.SH 是上交所发布的上证指数, 而 000001.SZ 则是深交所的深发展股票代码。

工程

股票分钟

股票Tick

代码

000001

SH

2014年12月 1日

股票

查询

	日期	时间	最新	高	开	低	涨幅	成交量	成交额	买1价	买2价	买3价	买4价	买5价
1	2014-12-01	8:44:52	2682.84	2682.84	2682.84	2682.84	0	0	0	0	0	0	0	0
2	2014-12-01	9:25:15	2691.73	2691.73	2682.84	2682.84	0.3314	5518162	489070...	0	0	0	0	0
3	2014-12-01	9:30:06	2694.28	2694.28	2682.84	2682.84	0.4264	8921910	774323...	0	0	0	0	0
4	2014-12-01	9:30:11	2693.74	2694.28	2682.84	2682.84	0.4063	11119260	956101...	0	0	0	0	0
5	2014-12-01	9:30:16	2692.97	2694.28	2682.84	2682.84	0.3776	11797990	101731...	0	0	0	0	0
6	2014-12-01	9:30:21	2692.66	2694.28	2682.84	2682.84	0.366	12280440	105721...	0	0	0	0	0
7	2014-12-01	9:30:26	2693.91	2694.28	2682.84	2682.84	0.4126	12696290	109544...	0	0	0	0	0
8	2014-12-01	9:30:31	2692.24	2694.28	2682.84	2682.84	0.3504	13107770	113292...	0	0	0	0	0
9	2014-12-01	9:30:36	2693.73	2694.28	2682.84	2682.84	0.4059	13526030	117197...	0	0	0	0	0
10	2014-12-01	9:30:41	2693.74	2694.28	2682.84	2682.84	0.4063	14171150	122674...	0	0	0	0	0
11	2014-12-01	9:30:51	2693.51	2694.28	2682.84	2682.84	0.3977	15235430	132719...	0	0	0	0	0
12	2014-12-01	9:30:56	2693.69	2694.28	2682.84	2682.84	0.4044	15832700	136225...	0	0	0	0	0
13	2014-12-01	9:31:01	2693.17	2694.28	2682.84	2682.84	0.385	16405570	143132...	0	0	0	0	0
14	2014-12-01	9:31:06	2693.03	2694.28	2682.84	2682.84	0.3798	16875430	147060...	0	0	0	0	0
15	2014-12-01	9:31:11	2693.36	2694.28	2682.84	2682.84	0.3921	17449800	151917...	0	0	0	0	0
16	2014-12-01	9:31:16	2693.48	2694.28	2682.84	2682.84	0.3966	17982660	156539...	0	0	0	0	0
17	2014-12-01	9:31:21	2693.74	2694.28	2682.84	2682.84	0.4063	18554450	161870...	0	0	0	0	0
18	2014-12-01	9:31:26	2694.07	2694.28	2682.84	2682.84	0.4186	19121690	167095...	0	0	0	0	0
19	2014-12-01	9:31:31	2693.2	2694.28	2682.84	2682.84	0.3862	19621610	171257...	0	0	0	0	0
20	2014-12-01	9:31:36	2694	2694.28	2682.84	2682.84	0.416	20152710	175735...	0	0	0	0	0
21	2014-12-01	9:31:41	2693.99	2694.28	2682.84	2682.84	0.4156	20659790	180020...	0	0	0	0	0
22	2014-12-01	9:31:46	2693.55	2694.28	2682.84	2682.84	0.3992	21174910	184263...	0	0	0	0	0
23	2014-12-01	9:31:51	2693.27	2694.28	2682.84	2682.84	0.3888	21711110	188733...	0	0	0	0	0
24	2014-12-01	9:31:56	2693.73	2694.28	2682.84	2682.84	0.4059	22248540	193309...	0	0	0	0	0
25	2014-12-01	9:32:01	2693.48	2694.28	2682.84	2682.84	0.3966	22728280	197556...	0	0	0	0	0
26	2014-12-01	9:32:06	2693.01	2694.28	2682.84	2682.84	0.3791	23233000	201864...	0	0	0	0	0
27	2014-12-01	9:32:11	2692.46	2694.28	2682.84	2682.84	0.3586	23668870	205455...	0	0	0	0	0
28	2014-12-01	9:32:16	2693.29	2694.28	2682.84	2682.84	0.3895	24136900	209273...	0	0	0	0	0
29	2014-12-01	9:32:21	2693.12	2694.28	2682.84	2682.84	0.3832	24597180	213066...	0	0	0	0	0
30	2014-12-01	9:32:26	2693.61	2694.28	2682.84	2682.84	0.4014	25126180	217171...	0	0	0	0	0

<

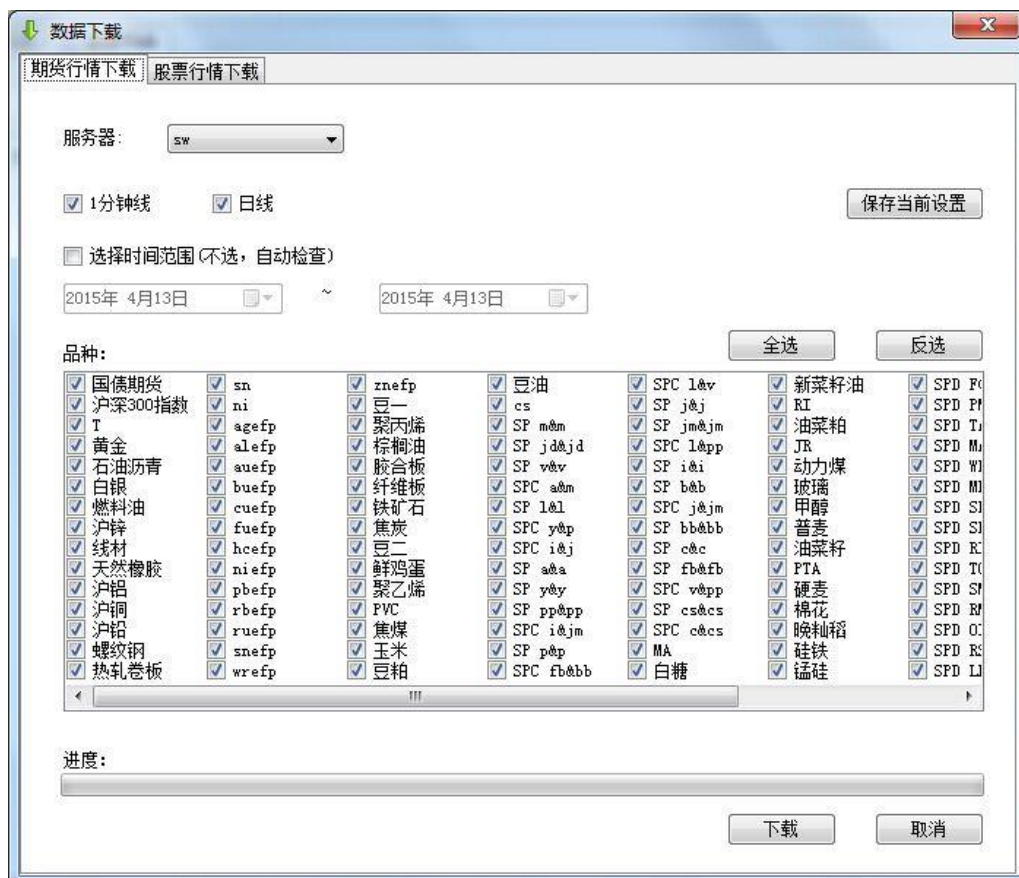
111

>

共有 2743 个 Tick

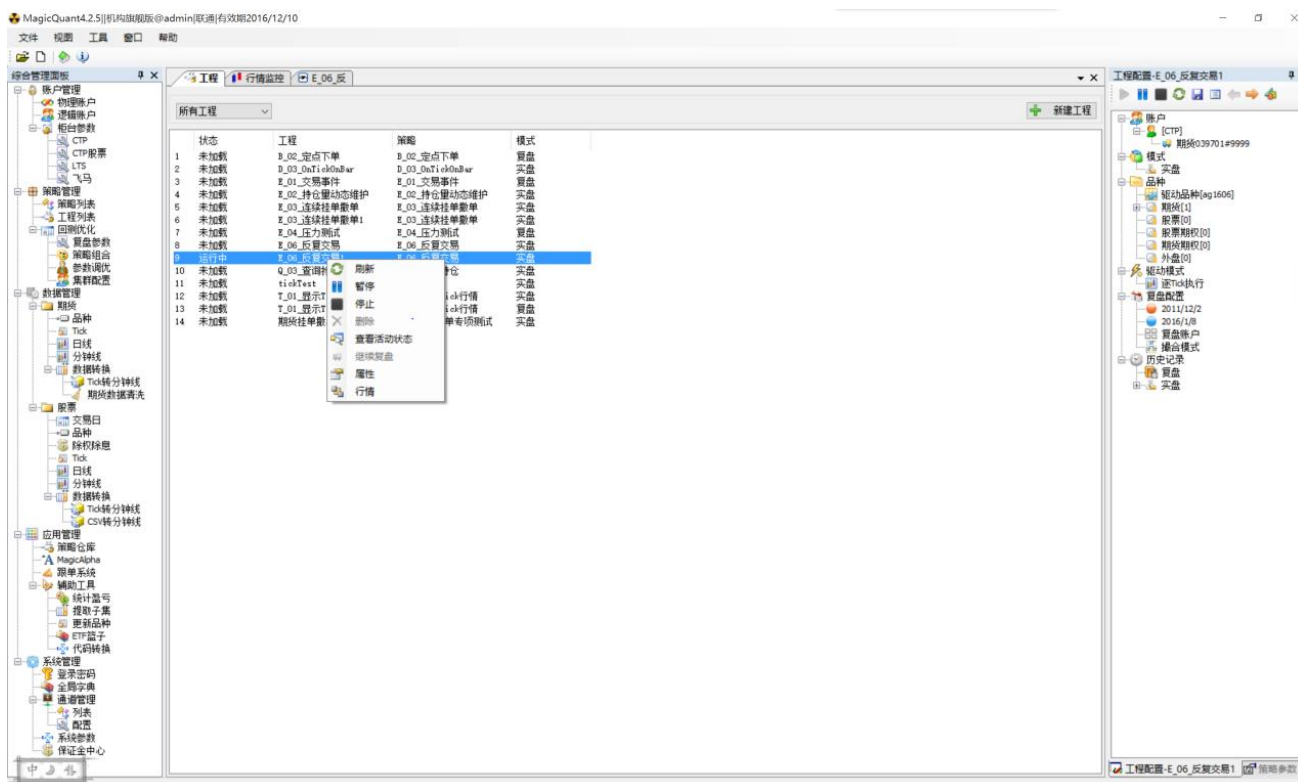
1.11 盘后下载

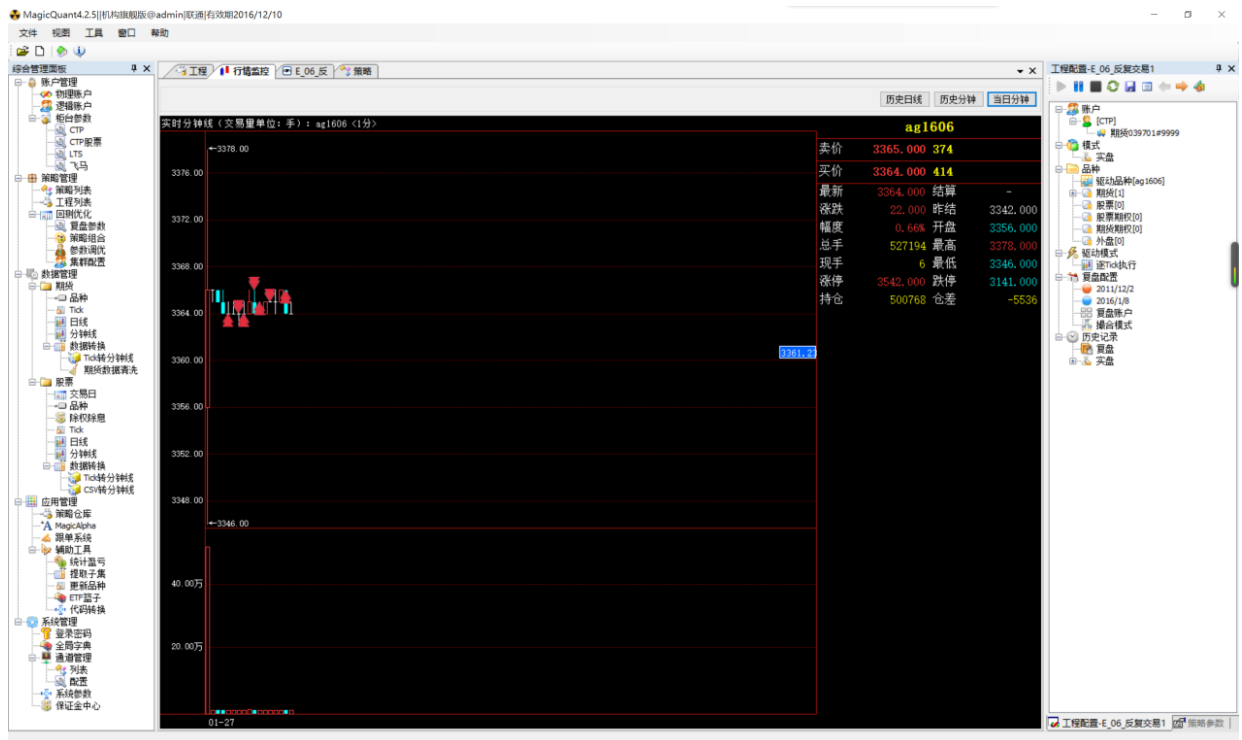
在盘后（非交易时间），选中【工具】→【数据下载】，可以下载期货的分钟线和日线。



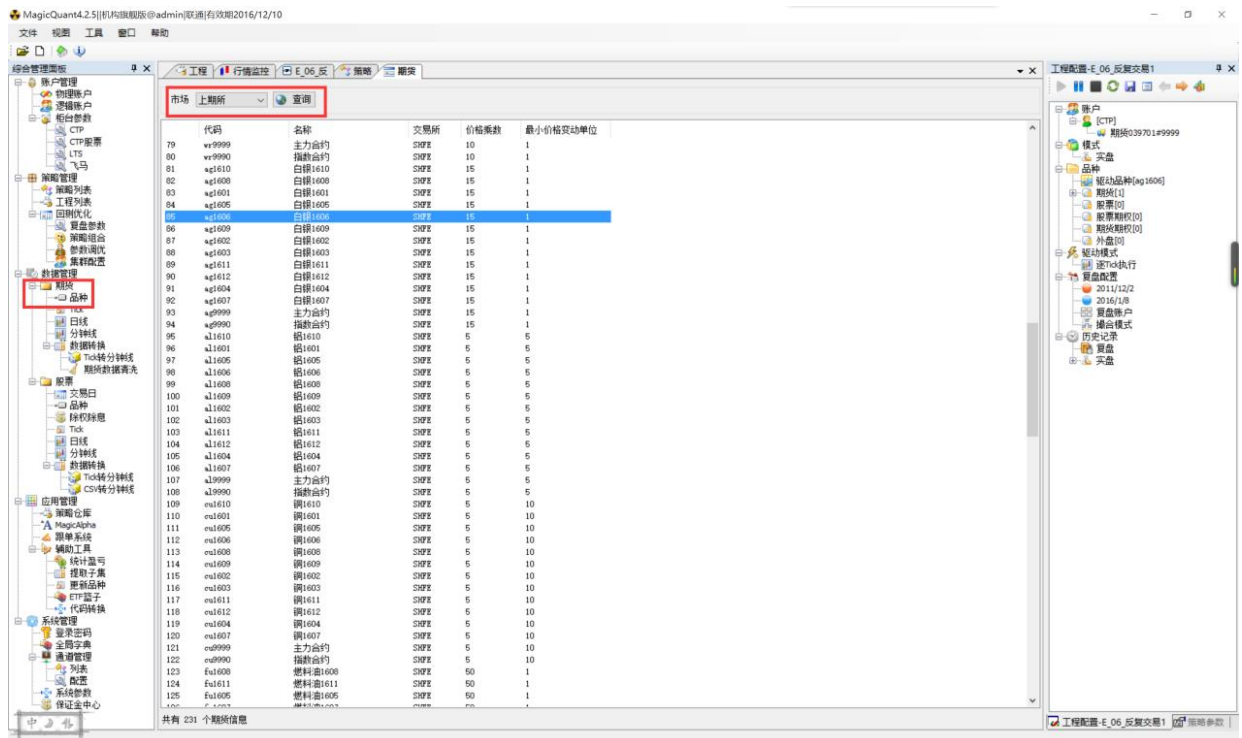
1.12 交易图表

MQ 可以将交易的成交信号显示到品种的行情的上，便于交易员监控交易过程。如果需要在实时交易过程中显示交易图表，注意需要在系统配置中，将实盘的“是否显示实盘成交”勾选上。目前只支持显示驱动品种的交易信号。点开工程列表，选择一个工程，右键选择【行情】选项，则会打开驱动品种的实时图表。



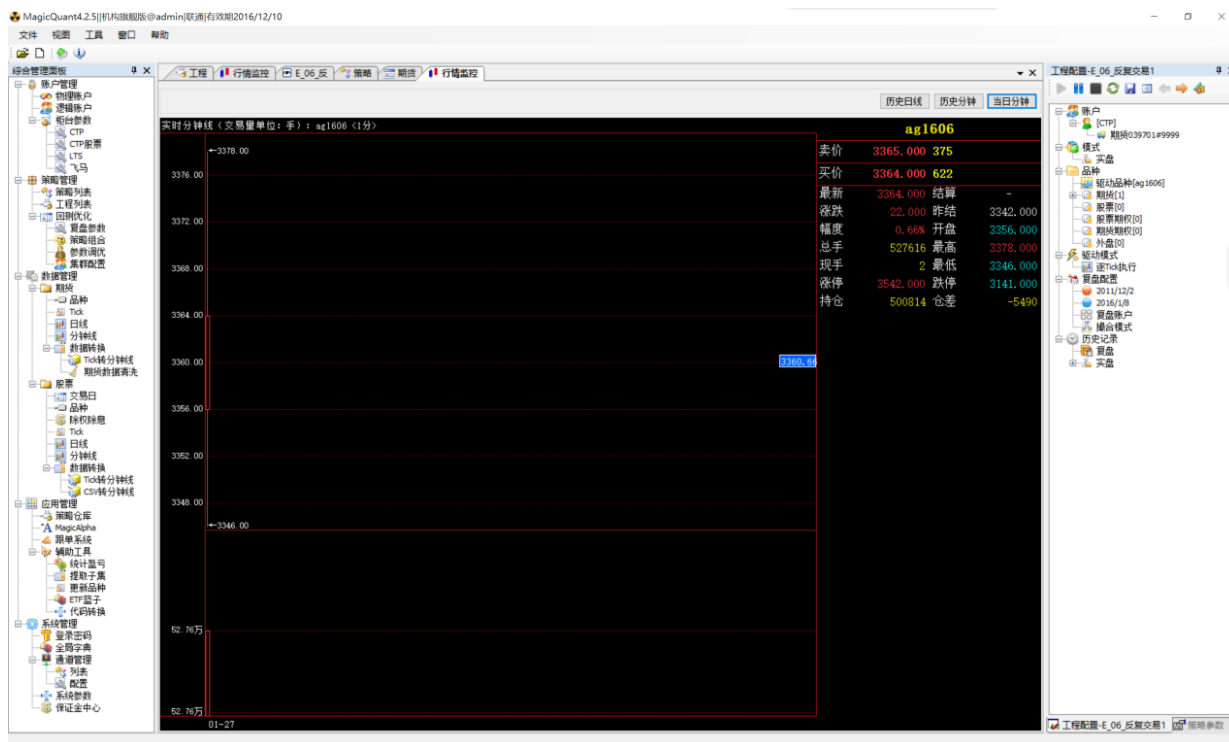


第二个调出实时图表的方法是：打开品种的浏览界面，选中一个品种，双击该品种，就可以显示该品种的实时图表。



The screenshot shows the '品种' (Contract) list table in the MagicQuant 4.2.5 interface. The table has the following columns: 代码 (Code), 名称 (Name), 交易所 (Exchange), 价格精度 (Price Precision), and 最小价格变动单位 (Minimum Price Change Unit). The table contains 231 contracts, with the first 100 contracts listed below:

代码	名称	交易所	价格精度	最小价格变动单位
79	vr9999	主力合约	SHFE	10
80	vr9990	指数合约	SHFE	10
81	ag1610	白铜1610	SHFE	15
82	ag1608	白铜1608	SHFE	15
83	ag1601	白铜1601	SHFE	15
84	ag1605	白铜1605	SHFE	15
85	ag1609	白铜1609	SHFE	15
86	ag1602	白铜1602	SHFE	15
87	ag1603	白铜1603	SHFE	15
88	ag1611	白铜1611	SHFE	15
89	ag1612	白铜1612	SHFE	15
90	ag1604	白铜1604	SHFE	15
91	ag1607	白铜1607	SHFE	15
92	ag1606	白铜1606	SHFE	15
93	ag9999	主力合约	SHFE	15
94	ag9990	指数合约	SHFE	15
95	al1610	铝1610	SHFE	5
96	al1601	铝1601	SHFE	5
97	al1605	铝1605	SHFE	5
98	al1606	铝1606	SHFE	5
99	al1608	铝1608	SHFE	5
100	al1609	铝1609	SHFE	5
101	al1602	铝1602	SHFE	5
102	al1603	铝1603	SHFE	5
103	al1611	铝1611	SHFE	5
104	al1612	铝1612	SHFE	5
105	al1604	铝1604	SHFE	5
106	al1607	铝1607	SHFE	5
107	al9999	主力合约	SHFE	5
108	al9990	指数合约	SHFE	5
109	cu1610	铜1610	SHFE	5
110	cu1601	铜1601	SHFE	5
111	cu1605	铜1605	SHFE	5
112	cu1606	铜1606	SHFE	5
113	cu1608	铜1608	SHFE	5
114	cu1609	铜1609	SHFE	5
115	cu1602	铜1602	SHFE	5
116	cu1603	铜1603	SHFE	5
117	cu1611	铜1611	SHFE	5
118	cu1612	铜1612	SHFE	5
119	cu1604	铜1604	SHFE	5
120	cu1607	铜1607	SHFE	5
121	cu9999	主力合约	SHFE	5
122	cu9990	指数合约	SHFE	5
123	fu1608	燃料油1608	SHFE	50
124	fu1611	燃料油1611	SHFE	50
125	fu1605	燃料油1605	SHFE	50



1.13 账户管理

➤ MQ 账户

MQ 账户是用户在 MQ 登录界面注册的账户。用户使用 MQ 账户登录 MQ 客户端。

➤ 物理通道

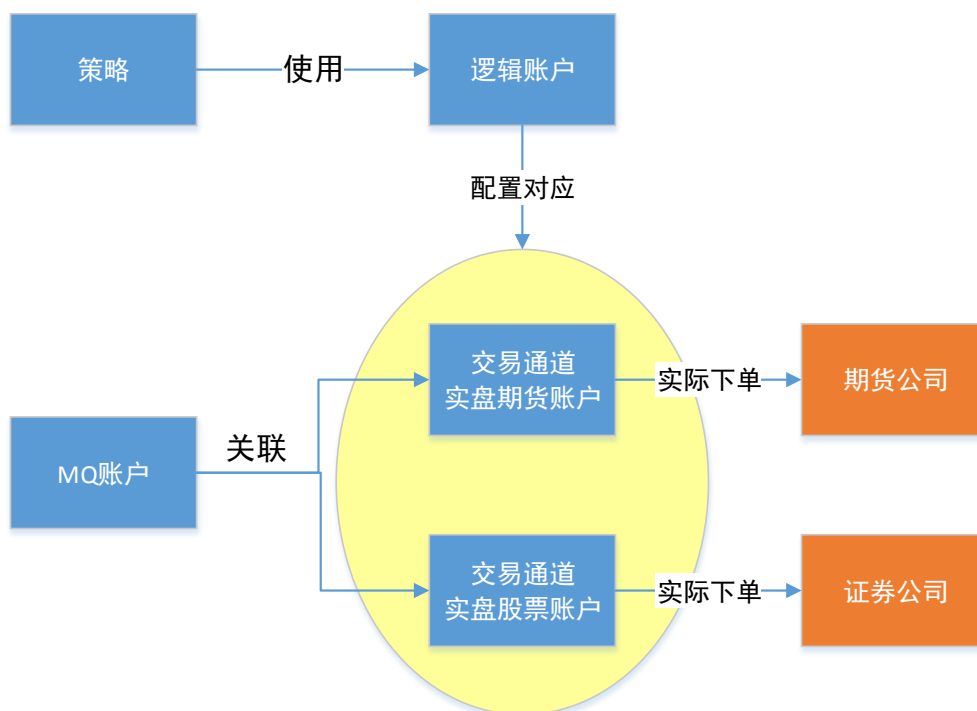
物理通道指的是用户在期货公司和证券公司开户的账户，是实际交易用的账户。由期货公司和证券公司提供物理通道的账号和密码。

➤ 逻辑账户

逻辑账户是策略使用的本地的一个虚拟账户，是策略里用来交易的账户。用户可以在 MQ 中创建一个逻辑账户，并且配置该逻辑账户对应的物理通道。一个逻辑账户最多对应 4 个物理通道：股票、期货、股票期权、期货期权。

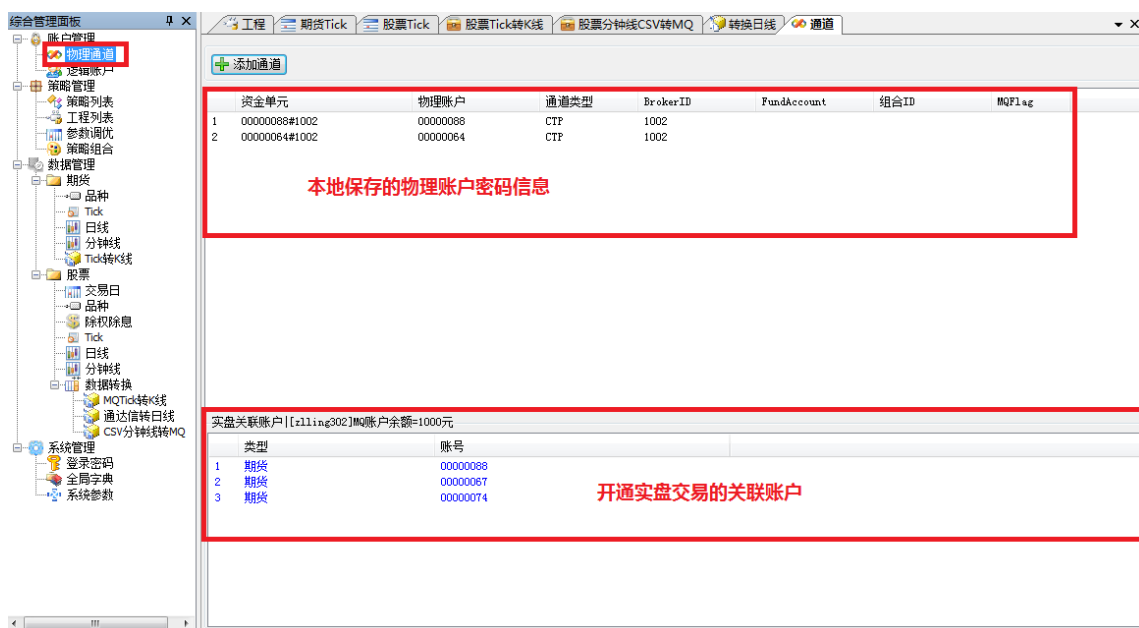
➤ 账户关联

账户关联指的是将 MQ 账户和物理通道建立绑定关系。关联后的物理通道才能在 MQ 中进行程序化交易。

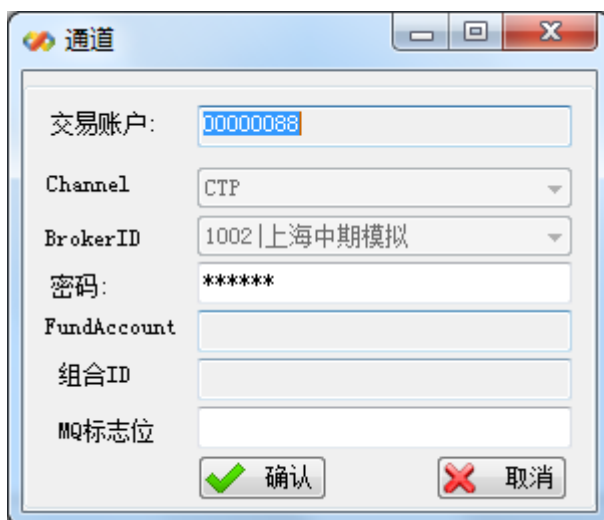


点击综合管理面板的【[账户关联](#)】，可以看到账户关联面板。账户关联面板上栏显示的是

本地保存的实盘交易账户和密码。中栏显示的是开通了程序化交易的实盘账户，这些实盘账户都是与当前登录的 MQ 账户关联的。



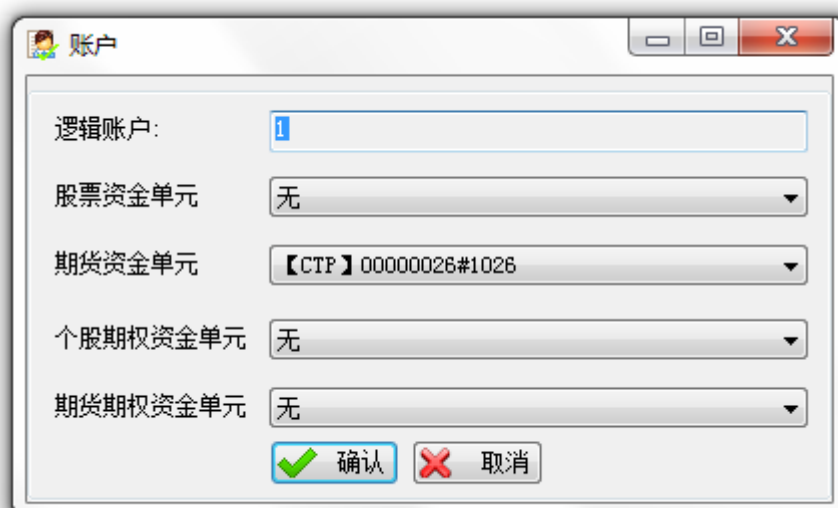
用户通过【通道管理】关联可以添加、删除、修改实盘的物理通道的信息。对于 CTP、LTS 柜台，FundAccount 和组合 ID 字段是不需要的。



对于飞马柜台，除了账户 ID 之外，还有一个 UserID，需要填在【组合 ID】的文本框中。



双击【逻辑账户】，单击【添加逻辑账户】，就可以为一个逻辑账户指定相应的物理账户（包括股票，期货，个股期权和期货期权），逻辑账户名称可以自行命名。如下图所示：

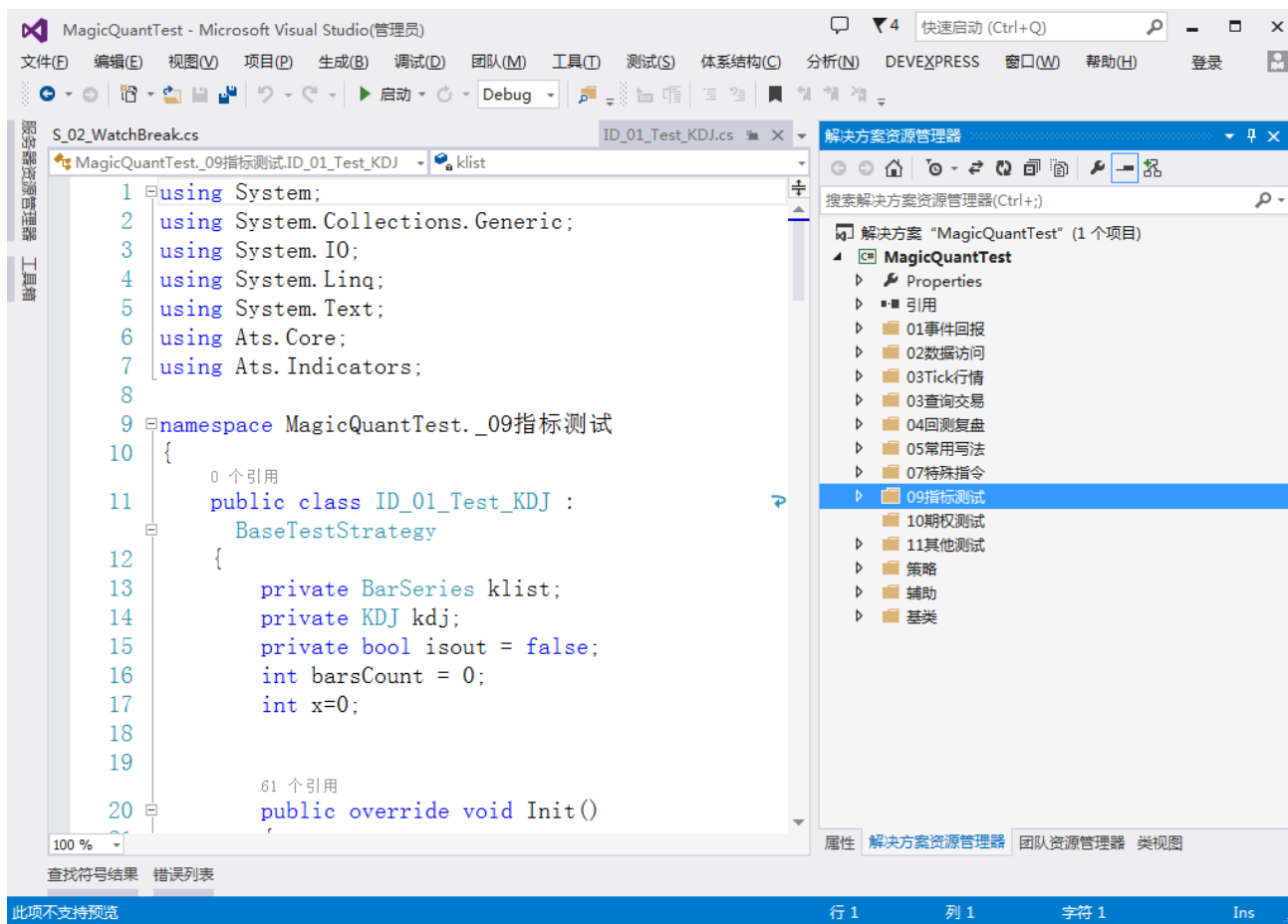


1.14 策略管理

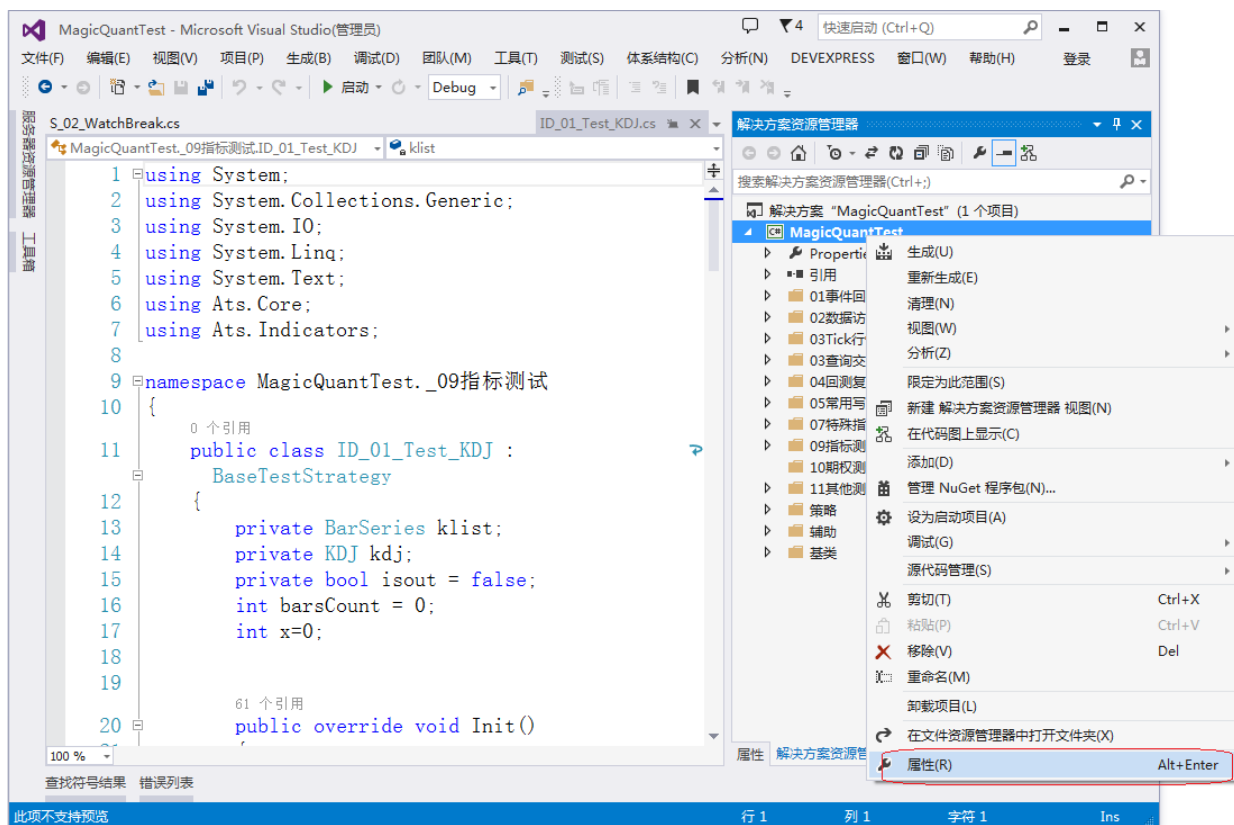
➤ 开发策略

单元测试是 MQ 提供的综合性的范例代码，里面包含大量的测试代码，范例策略和辅助类策略，还有日内波动策略的基类 **BaseStrategy**，是学习策略开发的优秀代码。基于 MQ 自带的单元测试项目中的空策略模板，用户可以开发自己的策略，编译以后会生成相应的策略 dll 文件。

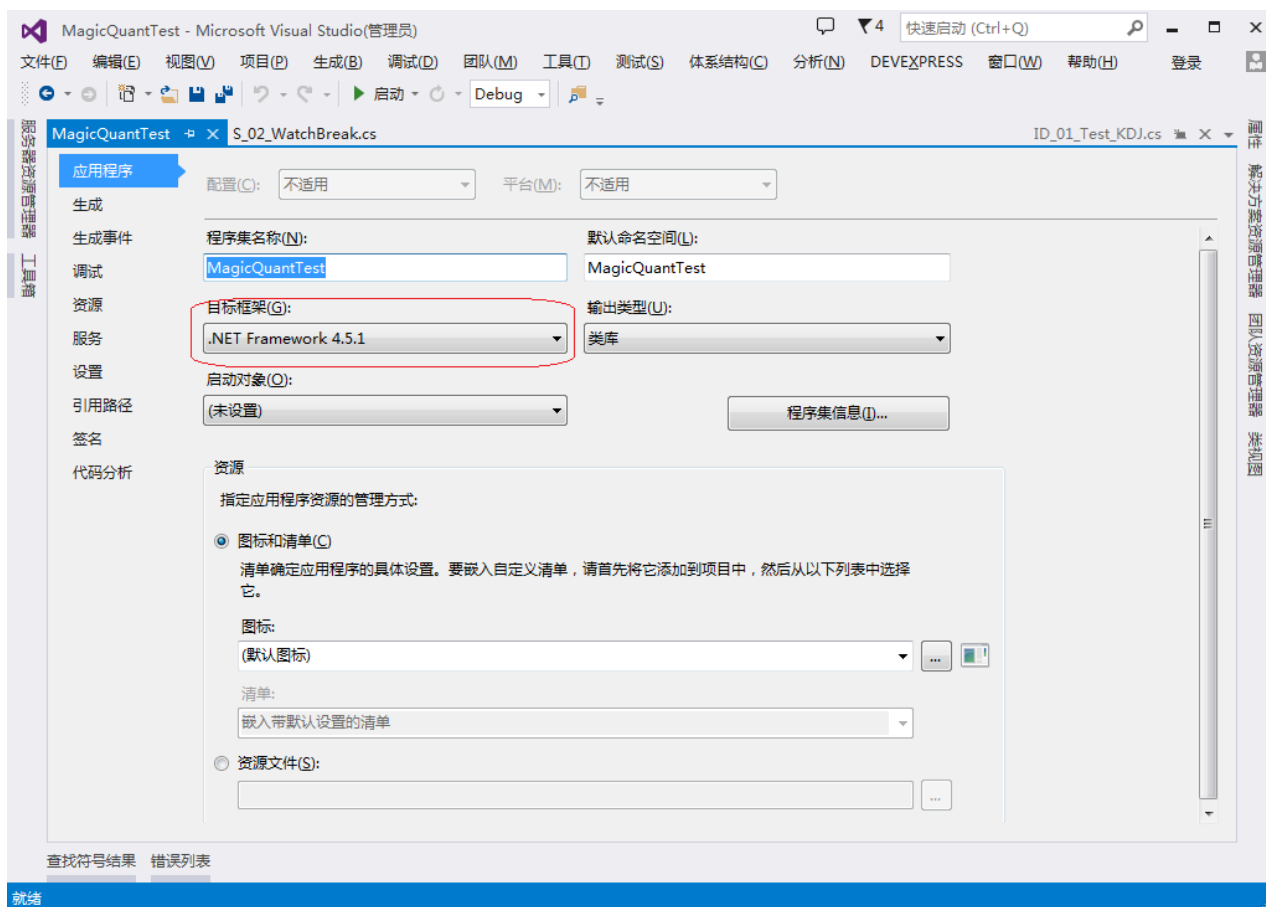
交易员安装好 VisualStudio 后，直接双击打开解决方案文件【MagicQuantTest.sln】，就可以打开策略源码。



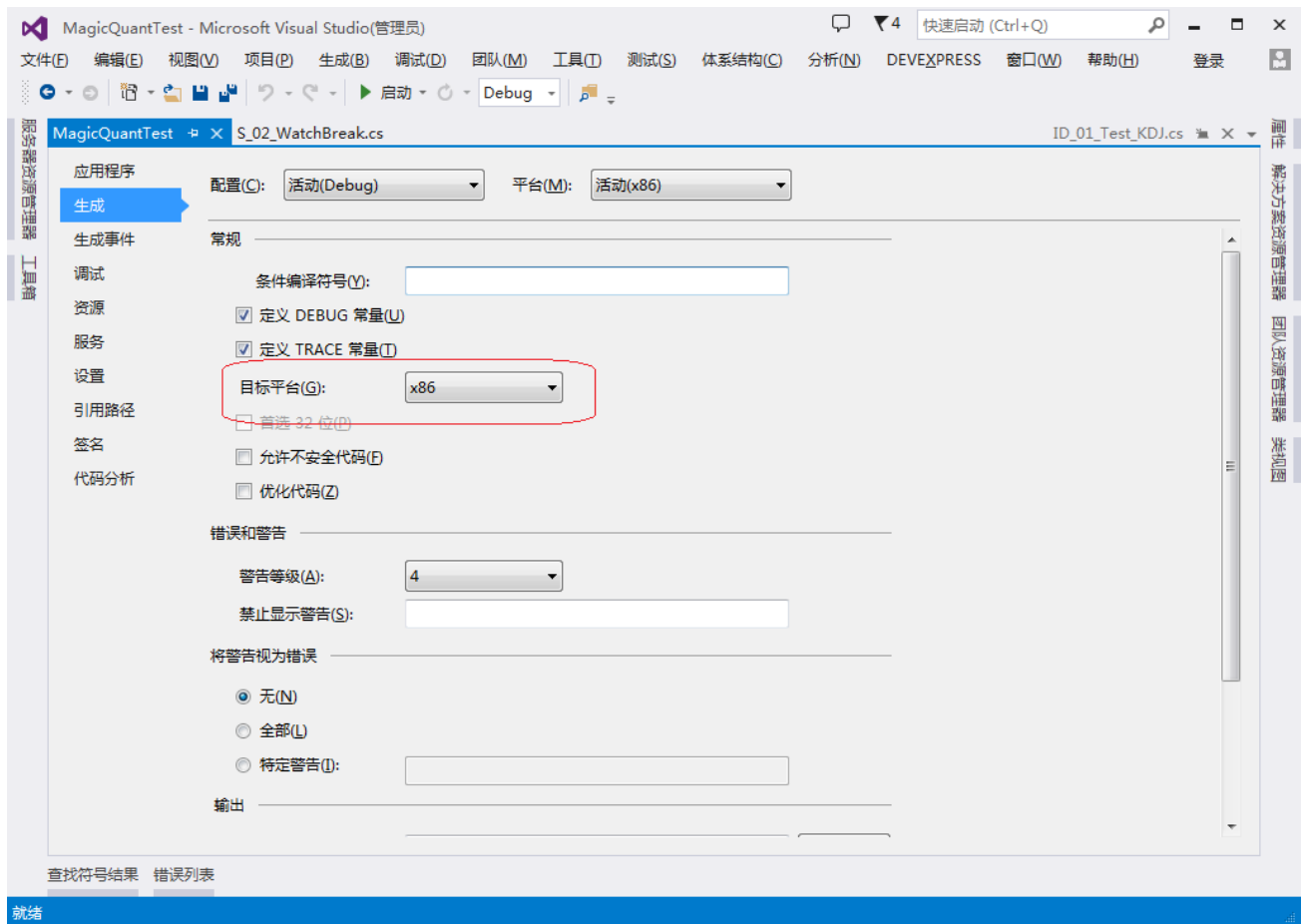
在工程名称上右键属性，可以打开工程属性面板，进而对工程进行配置。



工程属性-应用程序，设置目标框架为.NET Framework 4.5.1，目标平台是 x86。



工程属性-生成，设置目标平台为 x86。



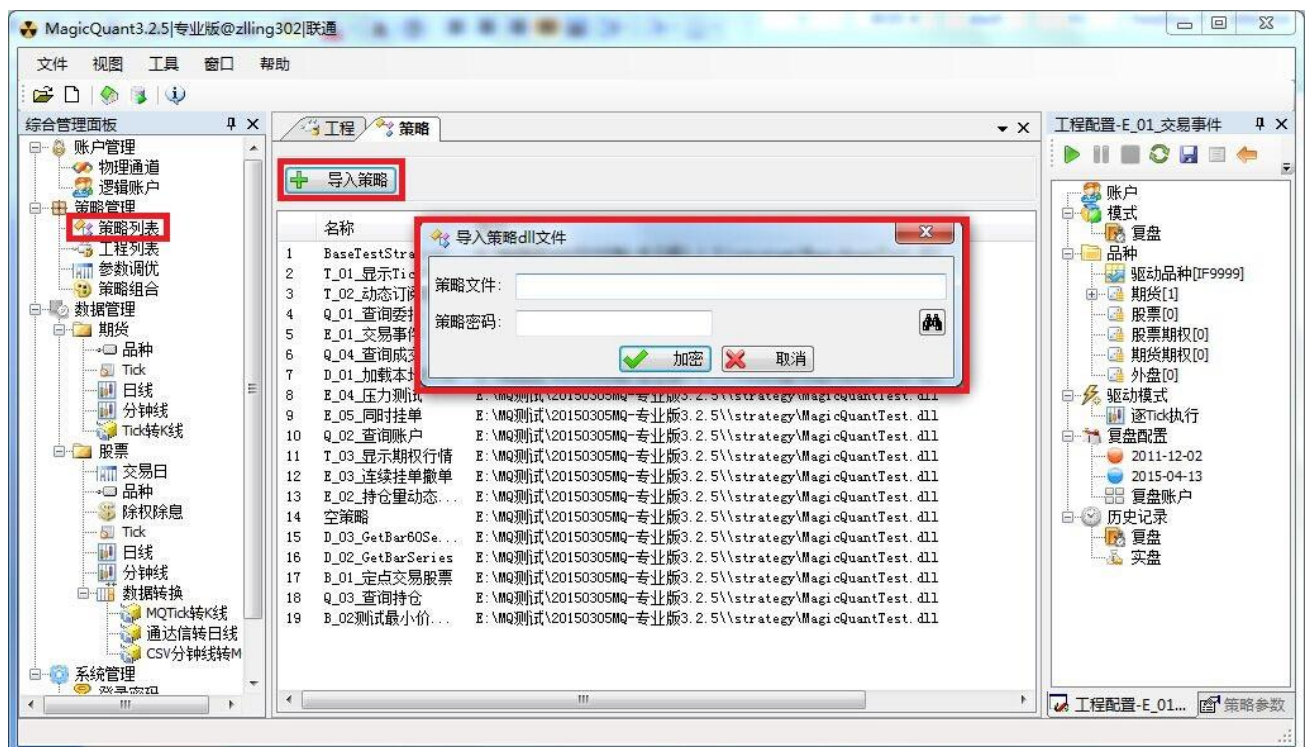
一个 Visual Studio 项目可以包含多个策略源码，因此一个策略 dll 中是可以有多个策略的。需要注意一个策略 dll 中的策略的类名不能相同。

注意：Ats.Core.dll 和 Ats.Indicators.dll 是必须引用的动态链接库，这两个文件在 MQ 的根目录下面。运行策略的 dll 文件必须和 MQ 根目录的 Ats.Core.dll 是同一个文件。

➤ 导入策略

通过综合管理面板可以进入策略管理界面，其中会列出所有已有的策略。在策略管理界面点击【导入策略】按钮，在弹出的窗口中，选择策略文件，同时输入策略密码，就可以将该策略 dll 文件加密后导入到 MQ 中。

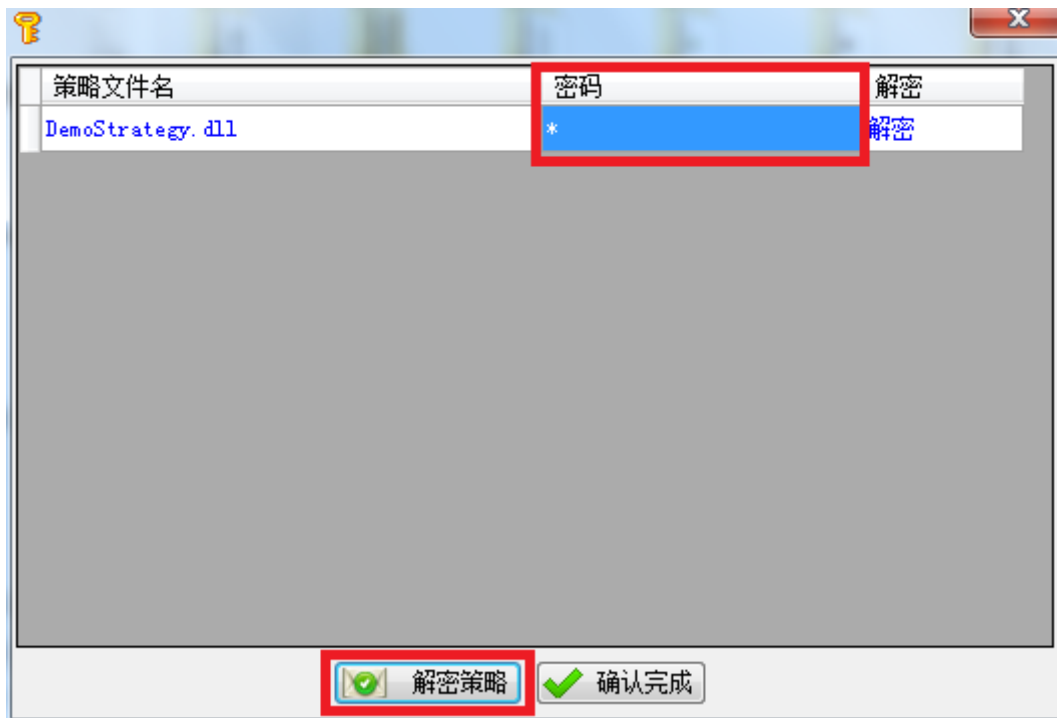
注意：新导入策略后，如果是实盘或者参数调优，需要重启 MQ，如果是普通复盘，不需要重启 MQ。



注意：因为出于策略安全考虑，MQ 每次启动程序时都需要强制输入该密码。该密码既不保存在客户端本地，也不保存在服务器端，只保存在用户的记忆中。一旦改密码丢失，已经导入的策略就不能被使用，所以请牢记您第一次载入策略时输入的策略密码。

➤ 解密策略

用户登录 MQ 以后，都需要输入策略密码，点击【解密策略】后，显示解密成功后才能加载策略到内存中。这样做的好处是即使策略 dll 遗失了，其他人也无法使用或者获取内部的策略信息。

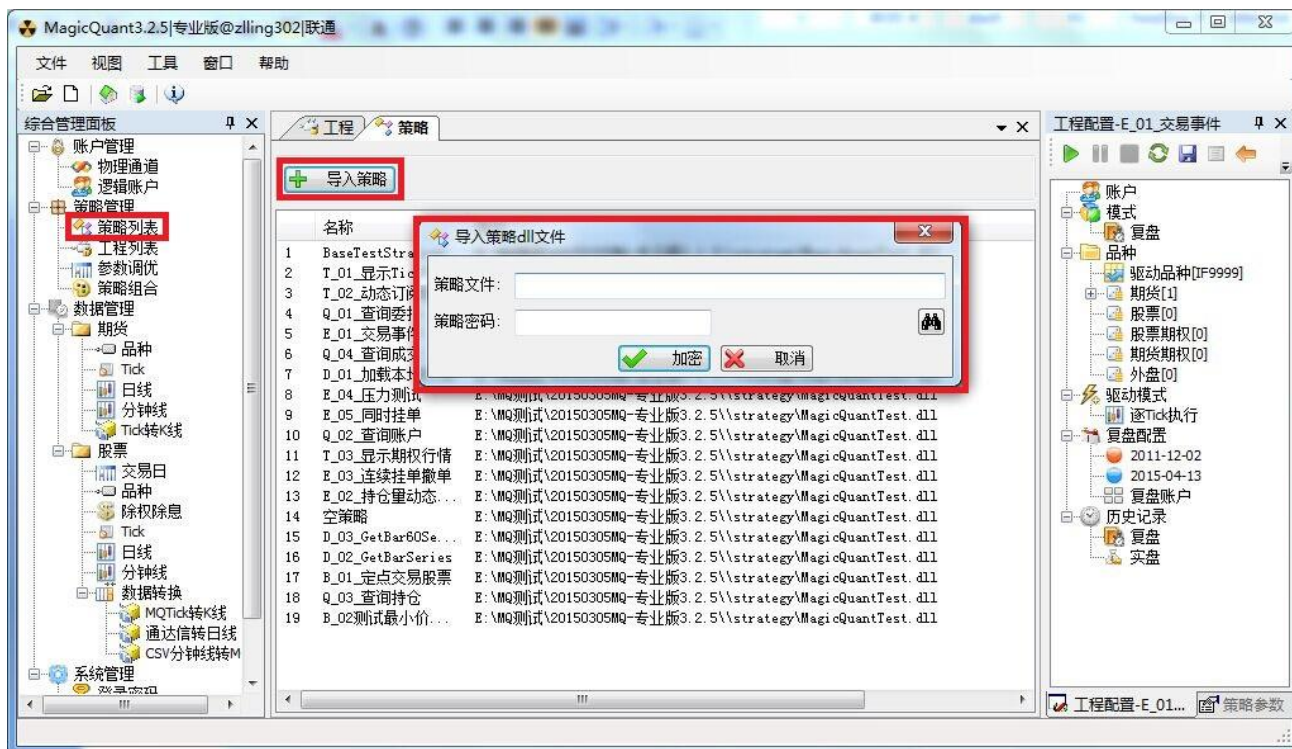


注意：和实盘交易账号的密码一样，MQ 的策略源码和 dll 文件均只在客户端计算机本地运行。由于 MQ 采取的多重加密手段，尽最大努力保护用户的策略安全。

1.15 工程管理

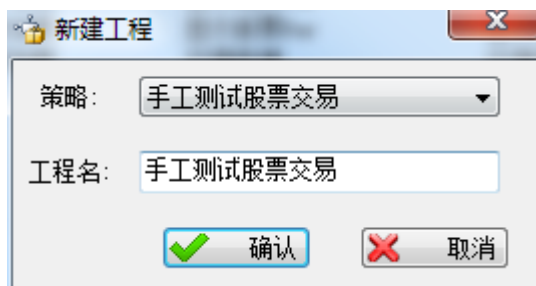
➤ 工程与策略的关系

工程是一个策略的实例，基于一个策略可以构造多个工程。例如：如果需要在多个账户上运行同一个策略，就可以创建多个工程，每个工程配置一个不同的账户即可。



➤ 创建工程

创建工程：点击【新建工程】按钮，在弹出窗口中选择策略，填写工程名称，即创建了一个工程。



工程有 5 种状态。

未加载：工程还没有被加载，也就是还没有被读入 MQ 内存中；

准备：工程已经加载，但还没有运行；

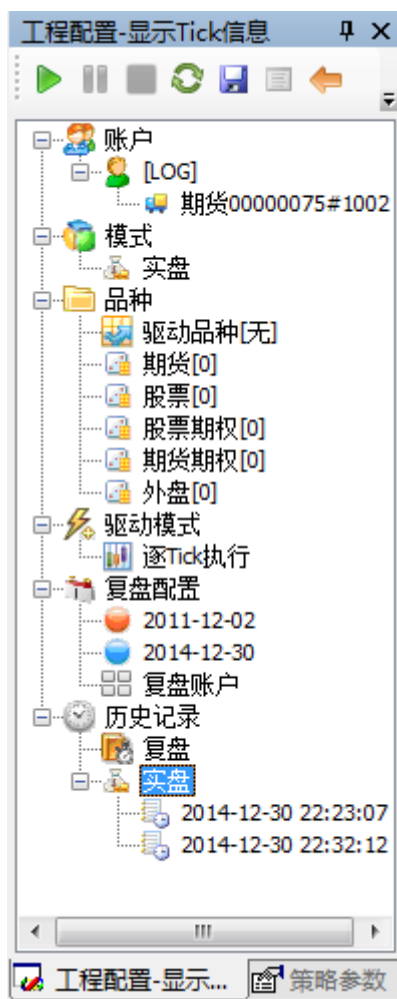
运行：工程正在运行，运行中的工程可以暂停；

暂停：工程被用户手工暂停，暂停中的工程可以恢复运行；

结束：工程运行结束。

➤ 工程配置

通过双击或者右键→【属性】，可以打开工程配置面板，对工程进行配置或者修改策略参数。**注意：策略运行、暂停时不能对工程进行配置，也不能修改策略参数。**



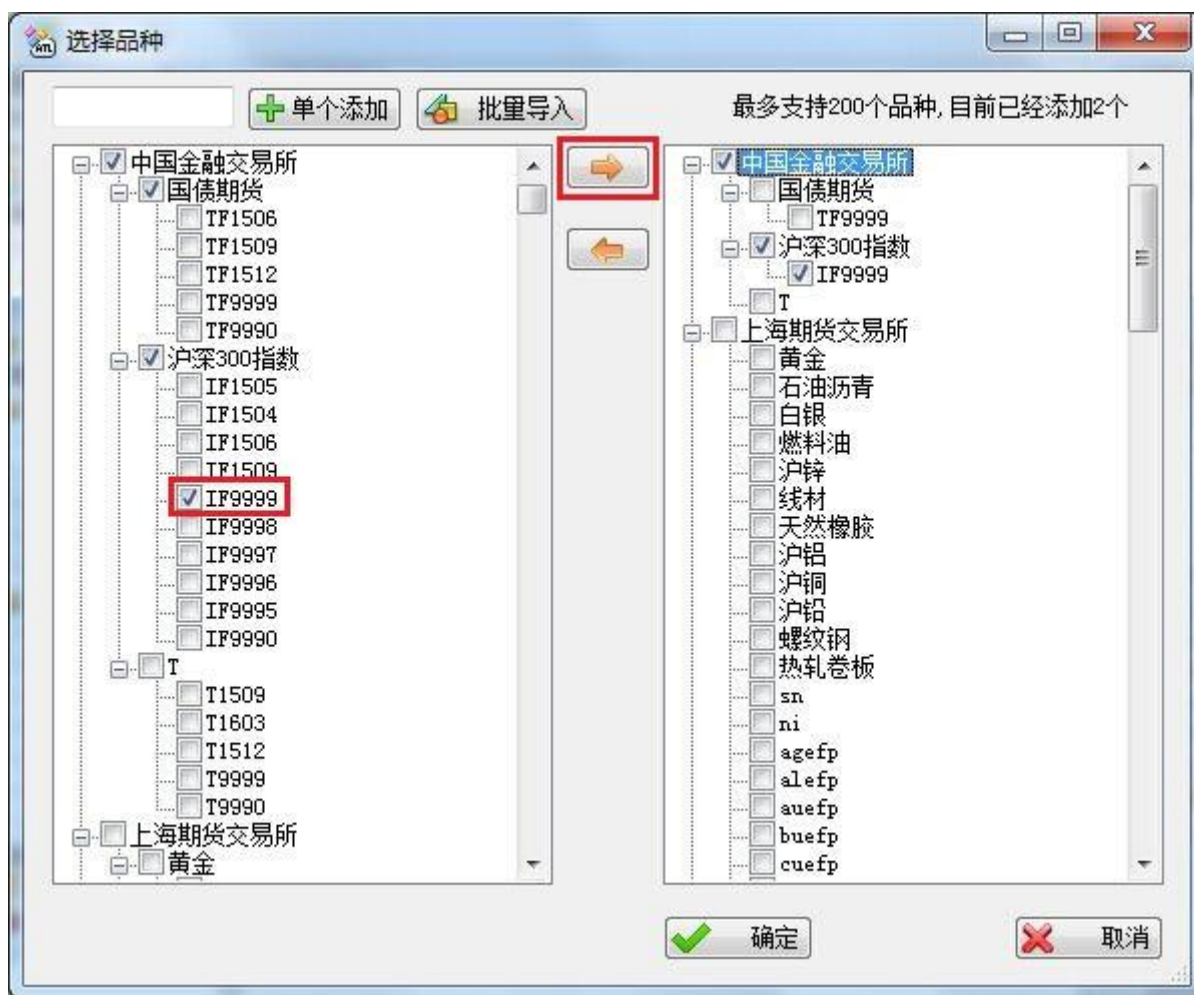
在【账户】上右键，选择【配置账户】，可以配置实盘交易的账户。

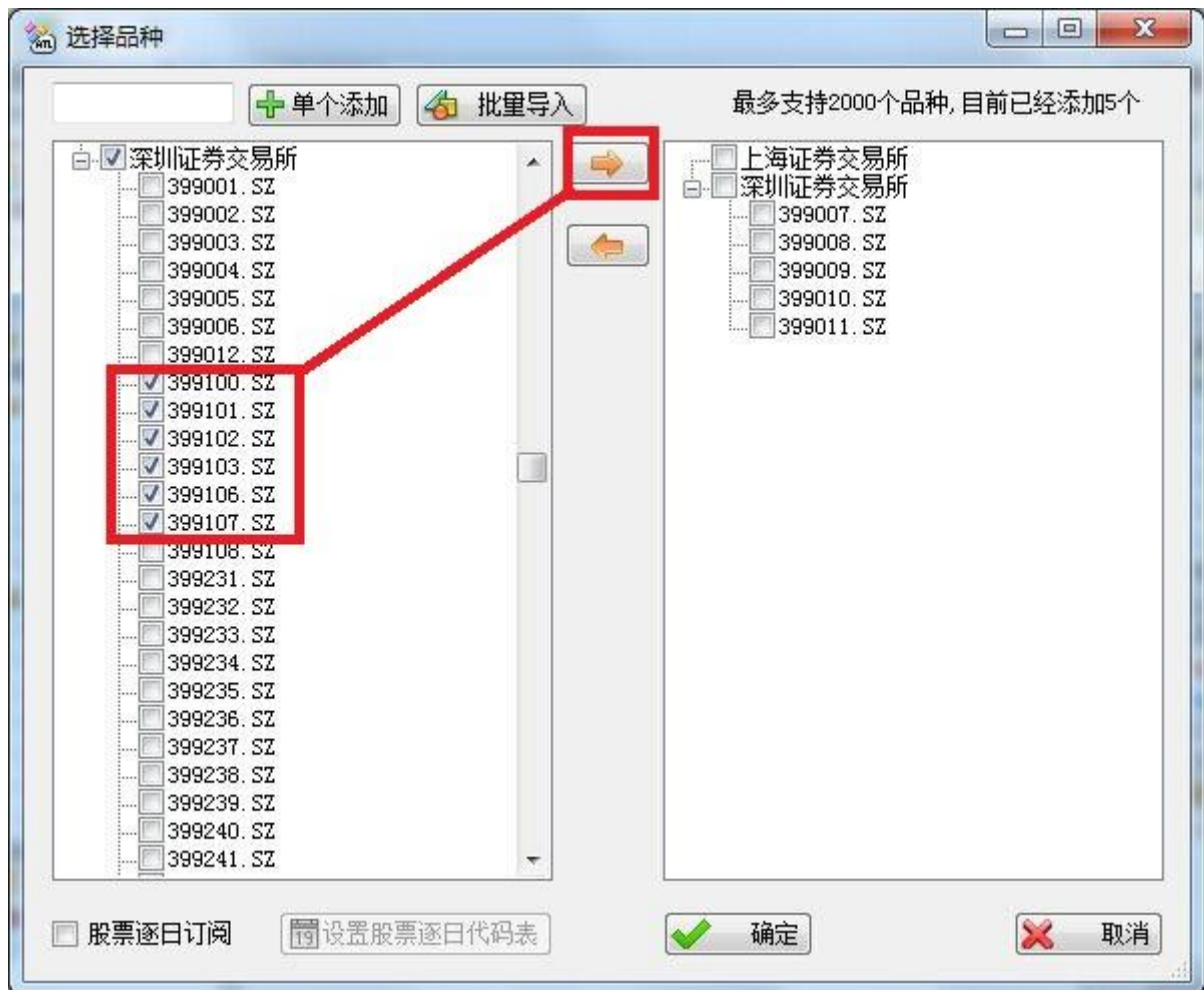
【模式】：工程有 2 种运行模式：复盘和实盘。复盘就是基于历史数据对策略进行回测；实盘是基于实时行情，使用真实账户进行真实交易。在界面上的复盘模式上右键可以修改工程的运行模式。策略运行过程中不能修改其工作模式。

在策略代码中，可以通过属性 `WorkMode` 获取当前策略的工作模式是实盘 `StrategyWorkMode.Live`，还是复盘（回测）`StrategyWorkMode.Simulate`。

【品种】：工程可以订阅多个股票和期货品种，只能配置一个品种作为驱动品种。在期货或者股票上右键选择【添加移除期货】或者【添加移除股票】，可以对订阅的品种进行调整。添加品种有三种方法：

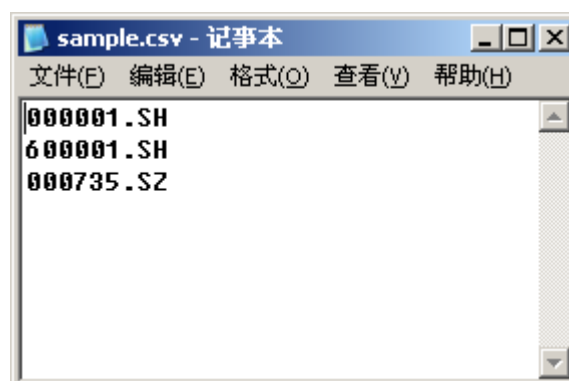
左侧的品种树是备选的品种列表，右侧的品种树是已经订阅的品种。勾选住品种前的复选框后，点击箭头按钮，可以添加或者移除品种。



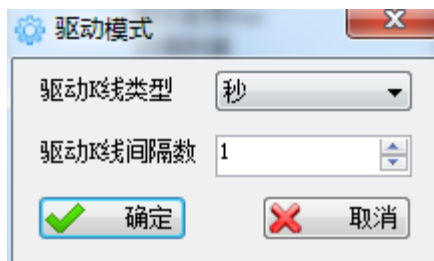


如果是添加个别品种，可以直接在左上角的文本框中输入期货合约代码或者股票代码，然后点击【单个添加】按钮。

如果要添加很多个品种，可以先把要添加的品种代码写到一个文本文件中，一行一个代码，然后点击批量导入。下面是一个样本的代码文本文件内容：

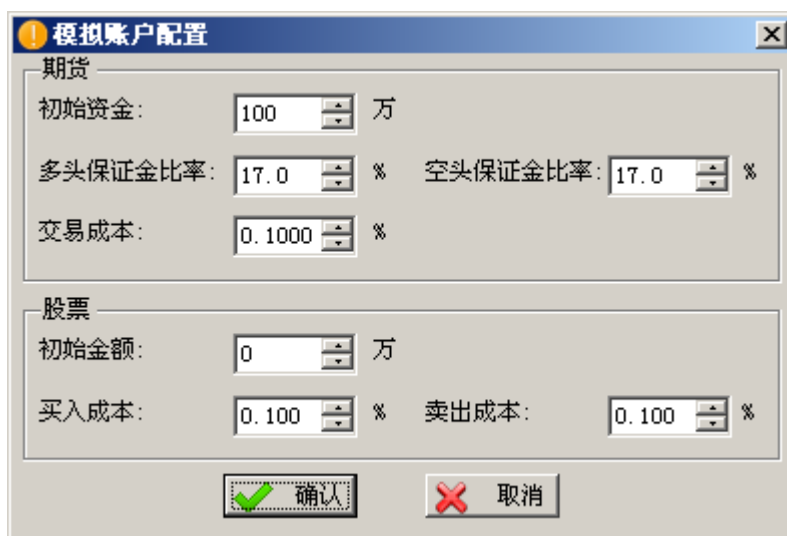


【执行方式】是驱动策略运行的机制，可以选择 Tick 驱动和 K 线驱动。双击执行方式下面的子节点，在弹出窗口上配置策略的驱动方式。下面就是一个基于 10 秒 K 线驱动的例子。



注意：策略都是由驱动品种的行情驱动的。

【复盘配置】可以配置复盘的起止日期和复盘账户资金和交易成本。



【历史记录】可以查询策略的历史日志信息。每次启动工程都会生成一个日志。

1.16 实盘交易

➤ 实时行情

MQ 的期货实时行情源来自于期货公司，券商或者第三方数据服务商的行情源。当用户配置了实盘账号以后，启动策略时，MQ 就会从行情源接收实时行情数据。行情源数据的质量和稳定性由行情源保证。行情的通道配置点击【菜单】→【工具】→【系统配置】→【行情通道】进行配置。



注意：由于交易通道和行情通道可以独立配置，因此用户可以使用真实行情源，同时使用模拟仿真账户进行交易。

➤ 启动实盘交易

启动实盘程序化交易的 4 个步骤：

- 1 在【账户关联】添加实盘的交易账户信息；
- 2 联系 MQ 关联并开通实盘交易，获得含有账户信息的序列号 SN 文件。
- 3 配置实时行情。
- 4 新策略导入后，启动实盘前一定要重启 MQ。

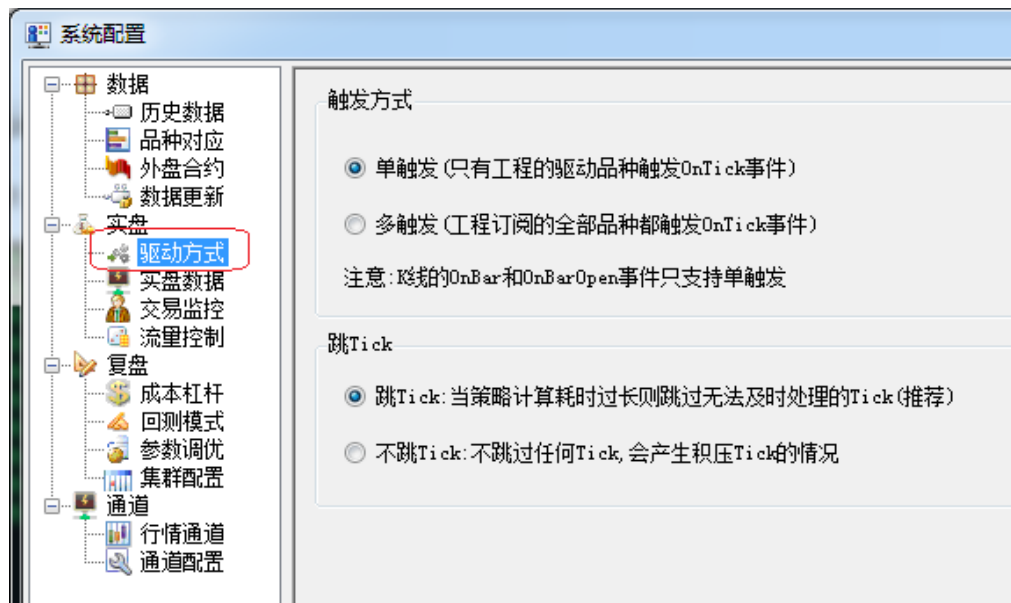
如果您需要开通实盘交易，请联系 MQ 管理员，提供您的开户期货公司、交易账号以及计算机的 MAC 地址，MQ 管理员将在服务器端完成关联和开通交易操作。

MQ 管理员联系方式：service@magicquant.com

注意：由于受 CTP 一般限制 6 个连接，因此在 MQ 中一个账户最多可以启动 5 个实盘工程（起码还需要留一个给人工监控使用）。如果策略要交易很多品种，需要开发支持多品种的策略，注意。每天开盘启动实盘策略之前一定要重启 MQ。

➤ 驱动方式

在系统配置的【实盘】→【驱动方式】页面中，可以配置策略的触发方式和是否跳 Tick。



➤ 数据落盘

➤ 盘中回补

➤ 交易监控

- 流量控制

2 策略语言

MQ 的策略语言是 C#语言。基于.NET Framework 强大的类库, MQ 提供了丰富的函数接口。用户可以像开发商业应用程序一样开发策略。原则上说, MQ 策略可以实现.NET 能实现的全部功能。例如操作数据库、访问网页、实现复杂回溯计算等。基于 MQ 的策略语言, 金融工程师可以将精力集中在交易思想的实现, 而不是编程技巧。

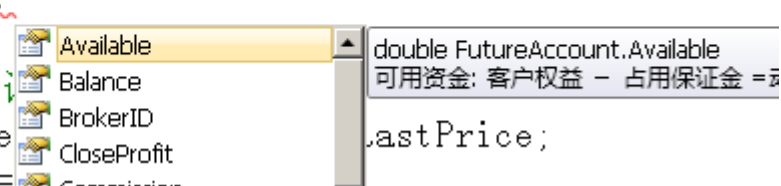
如果读者已经掌握了 C#语言的知识, 可跳过此章。

2.1 面向对象

C#语言处理的都是对象 Object。期货、期货账户、期货 Tick、股票 Tick、K 线、指标等都是对象。

对象具有属性和方法。属性用来描述对象的特征。例如期货账户的账号、余额、保证金等都是期货账户的属性。方法用来实现对象的行为。例如买入股票、发出限价委托时策略的方法。在 Visual Studio 中, 对象的属性和方法可以通过键入点 . 来调出。

```
public override void Init()
{
    MyFutureAccount.
    //采用最新价作为计
    double BasePrice
    if (BasePrice <=
```



2.2 基本语法

➤ 代码区块

一个代码区 **Block** 块由花括号 `{}` 包围的多行代码构成, 指定了程序要执行的一系列操作。下面的代码中首先判断变量 `a` 是否等于 `5`, 如果相等则调用 **MQ** 提供的 **Print** 方法, 在屏幕上显示 `ok` 和 `finish`。

```
if (a == 5)
{
    Print("ok");
    Print("finish");
}
```

注意: 在一个代码区块内部定义的变量只在这个代码块内有效。

➤ 注释

在策略代码中添加注释可以让代码清晰易懂, 便于维护。若需要对单行语句进行注释, 可以在句首使用 `//` 将该行文字注释, 若是需要对多行语句进行注释, 则可使用 `/* ...*/` 将段文字注释。

```
//这是一行注释
/*
 * 这是一段注释
 * 这是一段注释
 */
```

➤ 区分大小写

C# 是一个大小写敏感的语言, 因此定义变量时要区分大小写。因此变量 **STRATEGY** 和 **strategy** 是两个不同的变量。

➤ 变量及其类型

变量是在计算机内存中保存和读取信息的位置。由于内存的读写速度比文件读写快，因此变量的访问效率最高。下面介绍 C#语言中最常用的变量类型：

	数据类型	范例
<code>string</code>	文本	“MagicQuant”
<code>double</code>	浮点型	3.1415
<code>int</code>	整型	100
<code>bool</code>	布尔型	True False
<code>DateTime</code>	时间	2012 年 1 月 1 日 9:30

➤ 声明变量

声明变量时，首先要定义数据类型，然后设置一个在代码区块内唯一的变量名和初始值。

```
string a = "MagicQuant";  
double b = 0.618;  
int c = 5;  
bool d = true;  
DateTime t = new DateTime(2012, 1, 1, 9, 30, 0); // 定义时间为2012年1月1日9:30
```

➤ 变量及其类型

C#提供了常用的操作符进行数学和逻辑运算。操作符适用于整型、浮点型、字符串和布尔型的数据。

(1) 算术运算

操作符	运算	范例
+	加	1+2=3
-	减	5-4=1
*	乘	2*3=6
/	除	12/4=3
%	求余	6%2=0

下面是一个算术运算的例子：

```
int myInt = 0; // 声明变量myInt  
myInt = 5 + (3 * 4); // 执行计算  
Print(myInt.ToString()); // 显示结果
```

(2) 关系计算

通过关系计算对两个变量进行比较,例如判断两个变量是否相等,变量 1 是否大于变量 2。
关系计算的结果是一个布尔变量 (True 或 False)。

操作符	关系运算
==	是否等于
!=	是否不等于
<	是否小于
>	是否大于
<=	是否小于等于
>=	是否大于等于

下面是关系计算的示例:

```
int a = 4; int b = 5;
if (a > b)
{
    Print("a>b");
}
else
{
    Print("a<=b");
}
```

最后的结果会在界面显示 **a<=b**

(3) 逻辑运算

逻辑与`&&`和逻辑或`||`对两个布尔值或者表达式进行逻辑计算。

- 逻辑与`&&`

当两个布尔值都为 `True` 时，计算结果才为 `True`。

A	B	A&&B
True	True	True
True	False	False
False	True	False
False	False	False

如果是对两个表达式进行逻辑与计算，`&&`操作符必须对两个表达式都执行计算。如下面的代码：程序首先计算`a<b`，得出`True`，然后继续计算`b<c`，得出`False`，最后逻辑与的结果是`False`。

```
int a = 1; int b = 2; int c = 0;
if (a < b && b < c)
{
    Print("a<b and b<c");
}
```

- 逻辑或 ||

当两个布尔值中，只要有一个为 **True** 时，逻辑或||计算结果就为 **True**。

A	B	A B
True	True	True
True	False	True
False	True	True
False	False	False

如果是对两个表达式进行逻辑或计算，如果左边的布尔表达式为 **True**，则不会计算右边的布尔表达式，以节省计算资源。如下面的代码：程序首先计算 `a < b`，得出 **True**，此时直接返回结果 **True**，并不需要计算表达式 `b < c`。

```
int a = 1; int b = 2; int c = 3;
if (a < b || b < c)
{
    Print("a<b or b<c");
}
```

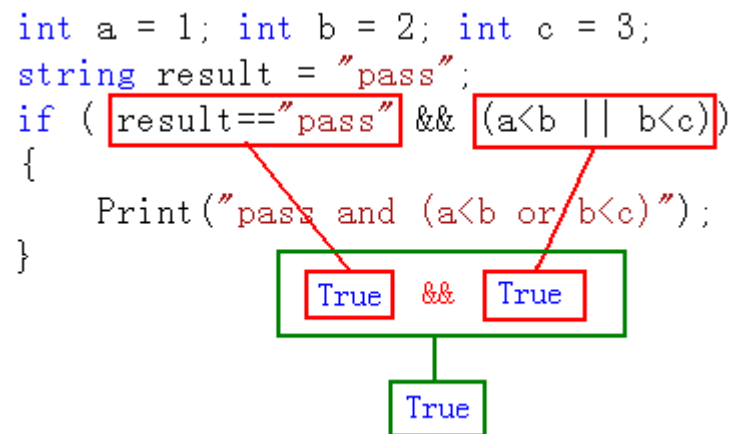
我们在开发策略时需要注意利用逻辑运算的这个优先计算的规则来保证策略的严谨性。策略在引用一个变量的属性前，一般先要确保变量本身有效，然后再看调用的属性是否有效。下面的代码示例中，首先判断期货队列 `AllFutures` 不是 `null`，在期货队列不是 `null` 的前提下再判断其长度大于 `0`，最后还要检查最新价格是否合理，防止数据坏点导致后续计算发生错误。

```
if ( AllFutures != null && AllFutures.Count > 0 && AllFutures[0].LastPrice > 0)
{
    double CurrentPrice = AllFutures[0].LastPrice;
}
```

C#语言可以对多个布尔表达式运用逻辑运算符**&&**和**||**，也可以用括号**()**实现相对较复杂的逻辑嵌套关系。下面的范例代码中程序首先计算表达式 `result=="pass"`，得到 **True**，然后再计算后面的逻辑或表达式**(a<b||b<c)**，得到 **True**，最终再将两个结果做逻辑与计算，最终结果

为 `True`。

```
int a = 1; int b = 2; int c = 3;
string result = "pass";
if ( result=="pass" && (a<b || b<c) )
{
    Print("pass and (a<b or b<c)");
}
```



The diagram illustrates the evaluation of the condition `result=="pass" && (a<b || b<c)`. The expression `result=="pass"` is evaluated to `True`, and the expression `(a<b || b<c)` is also evaluated to `True`. These two `True` values are combined using the logical AND operator `&&` to produce the final result `True`.

（4）赋值

基本的赋值运算符是`=`，同时 C# 还支持`+=`，`-=`，`*=`，`/=`这种简洁的赋值运算符。

运算符	含义
<code>=</code>	<code>x = y</code> 将值 <code>y</code> 赋给 <code>x</code>
<code>+=</code>	<code>x += y</code> 等价于 <code>x = x + y</code>
<code>-=</code>	<code>x -= y</code> 等价于 <code>x = x - y</code>
<code>*=</code>	<code>x *= y</code> 等价于 <code>x = x * y</code>
<code>/=</code>	<code>x /= y</code> 等价于 <code>x = x / y</code>
<code>++</code>	<code>x ++</code> 等价于 <code>x = x + 1</code>
<code>--</code>	<code>x --</code> 等价于 <code>x = x - 1</code>

➤ 类型转换

在数据类型匹配的前提下，简单类型的变量可以进行类型转换。

int 型转换成 string 型：

```
int a = 100;  
string b = a.ToString();
```

string 型转换成 int 型

```
string a = "100";  
int b = int.Parse(a);
```

double 型转换成 int 型

这种跨类型的转换需要借助 MQ 提供的 MathHelper。

```
double a = 3.14;  
int b = MathHelper.ConvertToInt(a);
```

➤ 字符串连接

字符串可以通过+连接。

```
string M = "Magic";  
string Q = "Quant";  
string MQ = M + Q;  
Print(MQ);
```

字符串也可以用+=运算符。

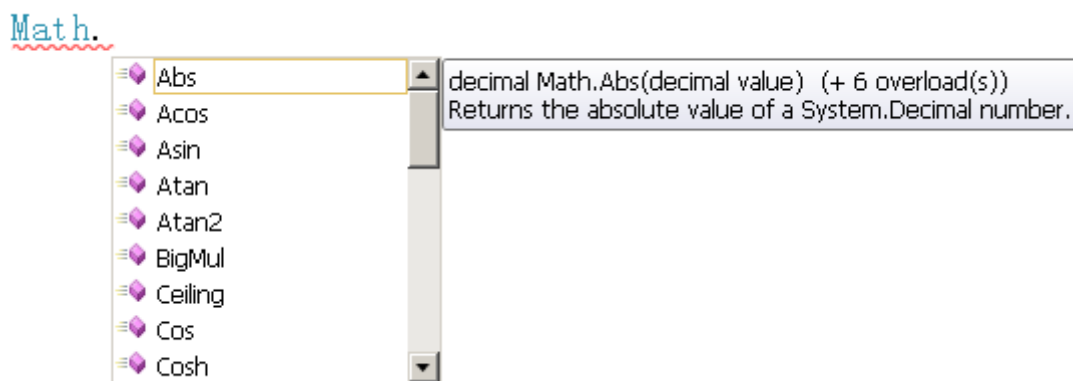
```
string str = "Magic";  
str += "Quant";  
Print(str);
```

2.3 数学函数

C#语言中包含了策略中需要用到的大部分基础数学函数，例如最大值、最小值、指数、对数、绝对值、三角函数等。C#将这些基础的数学函数封装在一个叫做 **Math** 的类库中。

➤ 智能感知

我们不一定能记全 C# 提供的所有方法，所以在开发时，可以利用 **Visual Studio** 的智能感知功能，以启发的方式快捷地调出 **Math** 库的相关方法。利用智能感知迅速调用相关方法的技巧也可以运用在其它类库上。在 **Visual Studio** 中，首先键入类名 **Math**，然后键入点 **.**，此时右侧会出现 **Math** 类库所有支持的方法，我们可以通过上下键方便地选择我们需要的方法。



➤ 常数

C# 内部提供了 2 个常数：圆周率 **PI** 和自然对数 **e**。

```
double pi = Math.PI;  
Print("PI=" + pi.ToString());  
double e = Math.E;  
Print("e=" + e.ToString());
```

➤ 常用方法

下表是一些策略中常用的基础数学方法。基于这些基础数学方法，可以降低我们开发策略的难度。

方法名	功能	范例	等价数学表达式
Abs	绝对值	Math.Abs(-1)	$ -1 $
Exp	以自然对数为底的指数	Math.Exp(2)	e^2
Log	以自然对数 e 为底的对数:	Math.Log(1.01)	$\log_e^{1.01}$
Pow	m 的 n 次方	Math.Pow(2, 3)	2^3
Max	两个数中的最大值	Math.Max(100, 99)	100
Min	两个数中的最小值	Math.Min(100, 99)	99
Round	四舍五入取固定的有效位数	Math.Round(9.36, 0)	9
Sqrt	平方根	Math.Sqrt(36)	$\sqrt{36}$

下面的实例代码中实现了经典的指数函数拟合公式 $f(x) = y_0 + A_1 * e^{-\frac{x}{t1}}$ 。

```
Future future = AllFutures[0];  
double x = future.LastTick.Change;  
double y = y0 + A1 * Math.Exp(-x / t1);
```

2.4 条件结构

C#语言支持对程序进行流程控制，包括条件结构和循环结构。

➤ if

if 结构，当特定的条件满足时执行指定的操作。

```
int x = 0;
if (x == 0)
{
    Print("MagicQuant");
}
```

➤ if-else

If-else 结构，当特定的条件满足时执行 if 后面的语句，否则执行 else 后面的语句，

```
int x = 1;
if (x == 0)
{
    Print("MagickQuant");
}
else
{
    Print("AutoTrade");
}
```

➤ if-else if-else

if-else if-else 结构是 if-else 结构的扩展，支持多重分支条件。

```
int x = 2;
if (x == 0)
{
    Print("MagicQuant");
}
elseif (x == 1)
{
    Print("AotuTrade");
}
else
{
    Print("CTP");
}
```

➤ switch

switch 结构可以针对一个变量，使用一系列数据进行匹配。

```
int x = 2;
switch (x)
{
    case 0:
        Print("x is 0");
        break;
    case 1:
        Print("x is 1");
        break;
    case 2:
        Print("x is 2");
        break;
    default:
        Print("no result");
        break;
}
```

当没有任何数据与其匹配时，则转入 **default** 分支。

2.5 循环逻辑

循环逻辑用于执行遍历算法的场合。

➤ for

for 结构是最常用的循环逻辑，其基本结构如下：

```
for (初始化; 条件表达式; 迭代计算)
{
    //执行代码;
}
```

例如我们想将策略订阅的所有期货品种代码显示出来，就可以用 **for** 结构。由于 **C#** 列表的索引都是从 **0** 开始的，因此初始化的时候定义 **i=0**，然后条件表达式 **i < AllFutures.Count** 用于判断是否索引移到了最后一个；迭代计算是在每次循环结束后执行，对索引 **i** 加 **1**。执行代码中将期货品种通过索引 **i** 取出，然后将期货品种代码显示在屏幕上。

```
for (int i = 0; i < AllFutures.Count; i++)
{
    Future future = AllFutures[i];
    Print(future.ID);
}
```

➤ Foreach

上一个例子中的 **AllFutures** 是一个列表，是多个期货品种的合集。如果要遍历数组、列表这种集合，**C#** 提供了一种更方便的循环结构 **foreach**，意思就是“对每一个，执行代码”。从下面的代码中可以看出，实现同样的功能，**foreach** 比 **for** 要简洁很多。

```
foreach (Future future in AllFutures)
{
    Print(future.ID);
}
```

➤ while

while 语句的含义是：当条件表达式的值为真(**True**)的时候重复执行某一系列操作，直到条件为假(**False**)时，循环才结束。

```
while (条件表达式)
{
    //执行操作
}
```

例如，策略需要将最近 **10** 个价格保存在内存中。策略首先构建一个 **List<double>** 型的浮点型数据队列，然后每来一个 **Tick**，就把最新的价格添加到队列中。如果队列的长度超过了 **10**，则移除最先加入的那个价格。

```
using System;
using Ats.Core;
using System.Collections.Generic; //由于用到了List，需要引用这个命名空间
namespace MyNameSpace
{
    public class TestStrategy : Strategy
    {
        List<double> PriceBuffer; //定义内存中的价格队列，类型为aList<double>
        public override void Init()
        {
            PriceBuffer = newList<double>(); //初始化
        }
        public override void OnTick(Tick tick)
        {
            Future future = AllFutures[0];
            if (future != null && future.LastPrice > 0) //安全性判断
            {
                PriceBuffer.Add(future.LastPrice); //将最新的价格添加到List中
                while (PriceBuffer.Count > 10) //如果队列长度>10
                {
                    PriceBuffer.RemoveAt(0); //移除最先加入队列的价格
                }
            }
        }
    }
}
```


➤ 死循环

在使用 **While** 循环的时候，假如条件表达式的值始终为 **True**，则循环操作会一直执行，永远不能退出的情况，这种情况我们称之为死循环，例如：

```
while (true)
{
    //执行操作
}
```

注意：如果策略代码中出现了死循环，策略将永远停止死循环的地方，导致后续操作都无法执行，进而对交易产生致命的影响。因此要尽量避免死循环的情况。

➤ Break

如果想从死循环中跳出，需要用 **Break** 语句。例如

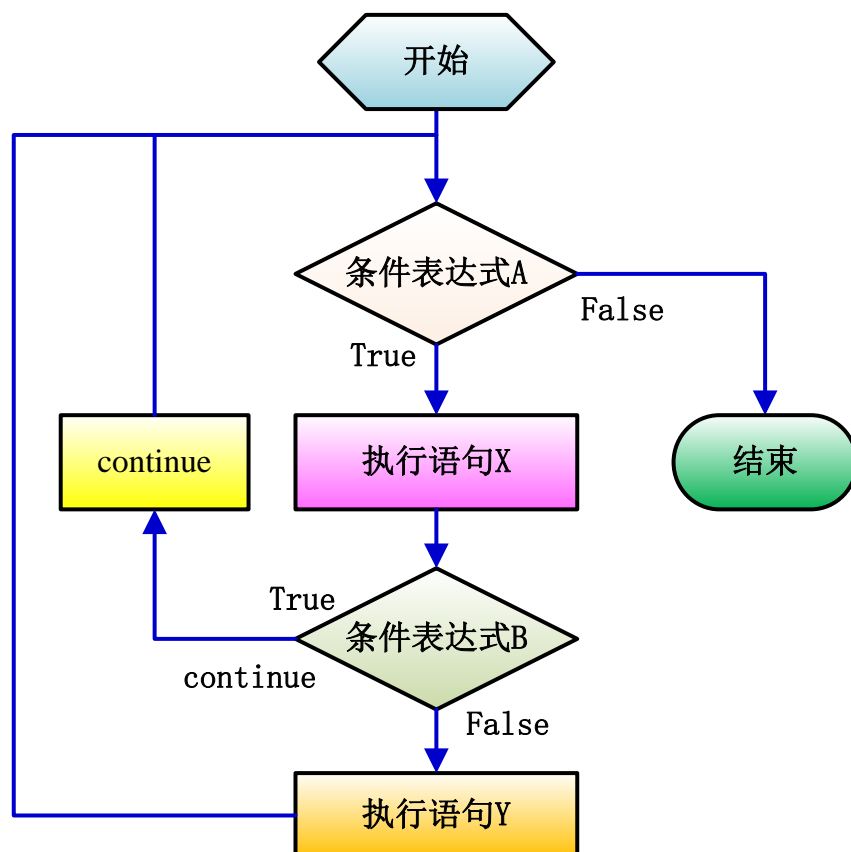
```
while (true)
{
    //此时已经进入死循环的逻辑
    if (跳出循环条件表达式)
        break;
}
```

➤ continue

有的时候在循环中，我们可能希望跳过后面的代码，进入下一次循环，在这种情况下，可以使用 **continue** 语句来达到目的，如下：

```
while (条件表达式)
{
    //执行语句
    if (条件表达式)
        continue; //跳过执行语句
    //执行语句
}
```

当条件表达式 A 满足时，循环被执行，在执行完执行语句 X 后，将判断条件表达式 B 的值，当条件表达式 B 为 True，将执行 **continue**，即跳过执行语句 Y，重新判断条件表达式 A 的值，进入下一次循环。否则将继续执行执行语句 Y。



2.6 高级类型

➤ 枚举

当变量的数值有几种可能的值时，我们可以用枚举类型 `enum`。

下面列出 MQ 中常用的一些枚举类型：

市场

```
public enum EnumMarket
{
    /// <summary>
    /// 股票
    /// </summary>
    股票 ,

    /// <summary>
    /// 期货
    /// </summary>
    期货 ,

    /// <summary>
    /// 个股期权
    /// </summary>
    股票期权 ,

    /// <summary>
    /// 期货期权
    /// </summary>
    期货期权 ,

    /// <summary>
    /// 指数
    /// </summary>
    指数,

    /// <summary>
    /// 外盘
    /// </summary>
    外盘,
}
```

```
/// <summary>
/// 买入卖出方向
/// </summary>
public enum EnumBuySell
{
    #region 普通

    /// <summary>
    /// 股票、期货买入
    /// </summary>
    买入,

    /// <summary>
    /// 股票、期货卖出
    /// </summary>
    卖出,
    #endregion

    #region ETF

    /// <summary>
    /// ETF申购
    /// </summary>
    ETF申购 ,

    /// <summary>
    /// ETF赎回
    /// </summary>
    ETF赎回 ,
    #endregion

    #region 两融

    #region 融资

    /// <summary>
    /// 向券商借钱买入股票(信用账户)
    /// 融资买入的品种须在公司规定的标的证券范围之内
    /// </summary>
    融资买入,

    /// <summary>
    /// 卖出信用账户内的证券, 偿还其融资负债
    /// </summary>
    卖券还款,

    /// <summary>
    /// 偿还融资负债
    /// </summary>
    现金还款,
    #endregion

    #region 融券

    /// <summary>
    /// 向券商借股票卖出做空
    /// 融券卖出的品种须在公司规定的标的证券及所提供的融券券源范围之内
    /// </summary>
```

```
融券卖出,

/// <summary>
/// 从二级市场买入股票还当初融券的负债
/// </summary>
买券还券 ,

/// <summary>
/// 利用已经持有的股票还当初融券的负债
/// </summary>
现券还券,
#endregion
#endregion

#region 期权

/// <summary>
/// 期权专用：备兑开仓和备兑平仓
/// 在拥有标的证券（含当日买入）的基础上
/// 卖出相应的认购期权（百分之百现券担保，不需现金保证金）
/// 即通过备兑开仓增加备兑持仓头寸
/// </summary>
备兑,

/// <summary>
/// 锁定现货，用做备兑开仓
/// 开仓+锁定 现货股票 = 冻结股票
/// 平仓+锁定 现货股票 = 解冻股票
/// 锁定解冻后，LTS会触发成交回报
/// </summary>
锁定,

#endregion

#region 开放式基金 Open Fund

OF申购,

OF赎回,

#endregion

#region 分级基金 Structed Fund

SF拆分,

SF合并,

#endregion
}
```

```
/// <summary>
/// 开平仓
/// </summary>
public enum EnumOpenClose
{
    /// <summary>
    /// 开仓
    /// </summary>
    开仓,

    /// <summary>
    /// 平仓
    /// </summary>
    平仓,

    /// <summary>
    /// 平今
    /// </summary>
    平今仓,

    /// <summary>
    /// 强平
    /// </summary>
    强平,

    /// <summary>
    /// 强减
    /// </summary>
    强减
}

/// <summary>
/// 价格类型
/// </summary>
public enum EnumOrderPriceType
{
    /// <summary>
    /// 限价单，都可用
    /// </summary>
    限价，

    /// <summary>
    /// 市价
    /// </summary>
    市价,

    /// <summary>
    /// 市价报入，剩余转限价
    /// </summary>
    市价剩余转限价，

    /// <summary>
    /// 五档即成剩撤
    /// </summary>
    五档即成剩撤，
}
```

```
/// <summary>
/// 投机套保标志
/// </summary>
public enum EnumHedgeFlag : byte
{
    /// <summary>
    /// 投机
    /// </summary>
    投机 ,

    /// <summary>
    /// 套利
    /// </summary>
    套利 ,

    /// <summary>
    /// 保值
    /// </summary>
    保值 ,
}
```

```
/// <summary>
/// 持仓方向
/// </summary>
public enum EnumPositionDirection
{
    /// <summary>
    /// 买入带来的持仓
    /// 权力仓
    /// </summary>
    多头,

    /// <summary>
    /// 卖出带来的持仓
    /// 义务仓
    /// </summary>
    空头,

    /// <summary>
    /// 一般用不到
    /// </summary>
    净持仓,

    /// <summary>
    /// 期权特有
    /// 持有现货, 卖出开仓
    /// </summary>
    备兑,
}
```

```
/// <summary>
/// Bar类型
/// K线类型（日K线、分钟K线等）
/// </summary>
public enum EnumBarType
{
    /// <summary>
    /// 秒线
    /// </summary>
    秒 = 1,
    /// <summary>
    /// 分钟线
    /// </summary>
    分钟,
    /// <summary>
    /// 小时
    /// </summary>
    小时,
    /// <summary>
    /// 日线
    /// </summary>
    日线,
    /// <summary>
    /// 周线
    /// </summary>
    周线,
    /// <summary>
    /// 月线
    /// </summary>
    月线,
    /// <summary>
    /// 年线
    /// </summary>
    年线,
    /// <summary>
    /// Tick, 只支持 1 个 tick
    /// </summary>
    Tick
}
```

```
/// <summary>
/// 2条曲线互相穿过关系
/// </summary>
public enum EnumCross
{
    /// <summary>
    /// 上穿
    /// </summary>
    Above,
    /// <summary>
    /// 下穿
    /// </summary>
    Below,
    /// <summary>
    /// 无
    /// </summary>
    None
}
```



```
}
```

➤ 自定义枚举类型

在策略中我们也可以自定义枚举类型。假设我们定义策略可以处于三种状态：无、做空、做多，策略可以根据市场环境的变化在这三种状态之间来回切换。那么就可以定义这么一个枚举类型：

```
public enum StrategyMode
{
    None,
    Long,
    Short
}
```

➤ 时间段

前面我们学过 `Datetime` 类型，有时我们想知道两个时刻间的信息，需要用到 `TimeSpan` 类型。两个 `DateTime` 类型的值相减，可以得到时间段 `TimeSpan`。进一步访问 `TimeSpan` 的 `Total*` 类属性（*表示秒、毫秒等具体单位），可以获取这个时间段的总长度。下面这个例子中计算出已经开盘了多少分钟。

```
//定义开盘时间
DateTime tOpen = new DateTime(Year, Month, Day, 9, 30, 0);
TimeSpan ts = Now - tOpen; //获取开盘到现在的时间差
Print("已经开盘"+ts.TotalMinutes.ToString()+"分钟");
```

例如有时候我们需要控制交易频率，避免过渡频繁交易，那么可以设置一个阈值 `DT`，计算当前距离上次交易时刻 `T0` 的时间段的总长度，如果时间段过短，则不进行新的交易。

```
//100秒内不进行交易
double DT = 100;

if ((Now - T0).TotalSeconds >= DT)
{
    //执行策略计算
    //...
    //触发交易
    T0 = Now; //记录交易时刻
}
```

➤ 动态数组 [List](#)

[List](#) 是动态数组，通过 [List](#) 可以实现对列表的搜索、排序和操作。[List](#) 的写法是 [List<X>](#)，[X](#) 可以为任意类型，例如：整型、浮点型、字符串、布尔等基本类型，或者 MQ 定义的 [Future](#)、[Stock](#)、[Tick](#) 等，也可以是用户自定义的类型。在策略中如果要使用 [List](#)，需要添加引用 [using System.Collections.Generic](#)。[List](#) 的操作主要包括：

添加：[Add](#)

移除：[RemoveAt](#)

是否存在：[Contains](#)

遍历：[for](#) 或者 [foreach](#)

访问：[\[i\]](#)，[i](#) 是索引位置。

继续沿用前面一个例子，策略需要最近 10 个 [Tick](#) 的算术平均价格，我们可以构建一个自己维护的 [Tick](#) 缓存，将到来的 [Tick](#) 添加到缓存中，当缓存中 [Tick](#) 的个数超过 10 时，将 [Buffer](#) 中第一个 [Tick](#) 移除。然后对 [Buffer](#) 进行算术平均计算。

```
using System;
using System.Collections.Generic;
using Ats.Core;
namespace MyNameSpace
{
    public class TestStrategy : Strategy
    {
        [Parameter(Display = "N", Description = "", Category = "参数")]
        int N = 10;

        List<Tick> Buffer = newList<Tick>(); // 自己维护一个Tick缓存
        public override void OnTick(Tick tick)
        {
            Buffer.Add(tick); // 添加最新的Tick到Buffer中
            while (Buffer.Count > N)
            {
                Buffer.RemoveAt(0); // 移除第1个Tick
            }

            if (Buffer.Count >= N)
            {
                // 遍历求算术平均
                double avgPrice = 0;
                double sumPrice = 0;
                for (int i = 0; i < Buffer.Count; i++)
                {
                    sumPrice += Buffer[i].LastPrice;
                }
            }
        }
    }
}
```

```
    }  
    avgPrice = sumPrice / Buffer.Count;  
    Print("近期均价=" + avgPrice.ToString());  
  }  
}  
}
```

➤ 可回溯动态数组 [AtsList](#)

[AtsList](#) 是在 [List](#) 基础上增加回溯功能的动态数组。通过 [AtsList](#) 可以用方法 [Ago\(N\)](#) 实现对列表的回溯。

字典 [Dictionary](#)

字典是键 [Key](#) 和值 [Value](#) 的对应关系的集合，写法是 [Dictionary<key, value>](#)，[key](#) 和 [value](#) 均可以是任意类型。在策略中如果要使用字典 [Dictionary](#)，需要添加引用 [using System.Collections.Generic](#)。

注意：这个“字典”是 .NET 的类，其数据是保存在内存中的，而 MQ 的全局字典是 MQ 提供的另一种基于文件的键-值访问方法，其数据是保存在硬盘上的，要注意两者的区别。

字典 [Dictionary](#) 的操作主要包括：

添加：[Add](#)

是否包含键：[ContainsKey](#)

移除：[Remove](#)

清空：[Clear](#)

访问：[\[key\]](#)，[key](#) 是键

例如策略对于持仓的每个股票设定一个止损阈值，一旦跌幅超过该阈值，立即卖出。策略中可以建立一个键是 [string](#) 型，值是 [double](#) 型的字典，用来保存股票代码和止损阈值的对应关系。通过遍历所有键值，获取对应的股票涨跌幅，进而判断是否需要卖出该品种。

※ 范例

§ 字典 Dictionary 范例 §

※源代码 \基础概念\字典 Dictionary 范例.cs

```
using System;
using System.Collections.Generic;
using Ats.Core;
namespace DemoStrategy
{
    publicclass 字典Dictionary范例: Strategy
    {
        Dictionary<string, double> DICT = newDictionary<string, double>();//定义字典

        public override void Init()
        {
            DICT = newDictionary<string, double>();
            DICT.Add("000001.SZ", -0.2);//添加股票及其止损阈值
            DICT.Add("600001.SH", -0.5);
            DICT.Add("000735.SZ", -0.9);
        }
        public override void OnTick(Tick tick)
        {
            foreach (Stock stock in AllStocks)
            {
                string StockCode = stock.StockCode;
                if (DICT.ContainsKey(StockCode))
                {
                    double Gate = DICT[StockCode];//通过键访问对应的值
                    if (stock.LastTick.Change <= Gate)//当跌幅超过该品种的止损阈值时
                    {
                        //查看是否持有该股票
                        StockPosition pos = MyStockPositions[StockCode];
                        if (pos != null&& pos.EnableVolume> 0)
                        {
                            //如果持有该股票，按照最新价格卖出
                            SellStock(StockCode, pos.EnableVolume, stock.LastPrice);
                        }
                    }
                }
            }
        }
    }
}
```

2.7 方法

方法是把一些特定的，经常被调用的语句集合写在一个代码块中，定义其方法名称、输入参数和返回类型。方法的好处是，我们可以直接调用方法，而不用关心方法的实现细节，有利于将策略的逻辑复杂度拆分到各个方法中，而策略的主干部分清晰易读，便于代码维护。同时开发好的一个方法还可以被多个策略调用，实现了代码复用，避免了代码重复开发。

➤ 定义方法

方法由访问控制、返回类型、名称以及参数构成，例如下面就是一个公有的，返回类型是字符串型，输入参数是整型的方法，方法名是 `MyMethod`。

```
public string MyMethod(int x)
```

访问控制主要有两种：`public` 是公有型的，`private` 是私有型的。一般策略中自己开发的方法都设置为私有型的。如果不特别注明访问控制，系统默认也是私有型的。

```
string MyMethod(int x)
```

参数是可选的，可以没有参数，也可以有多个参数。例如下面这个方法定义就有 2 个数。

```
string MyMethod(int x, int y)
```

➤ 没有返回值的方法

如果方法没有返回值，我们用关键词 `void` 作为方法的返回类型。

```
void MyMethod(int x)
```

➤ 有返回值的方法

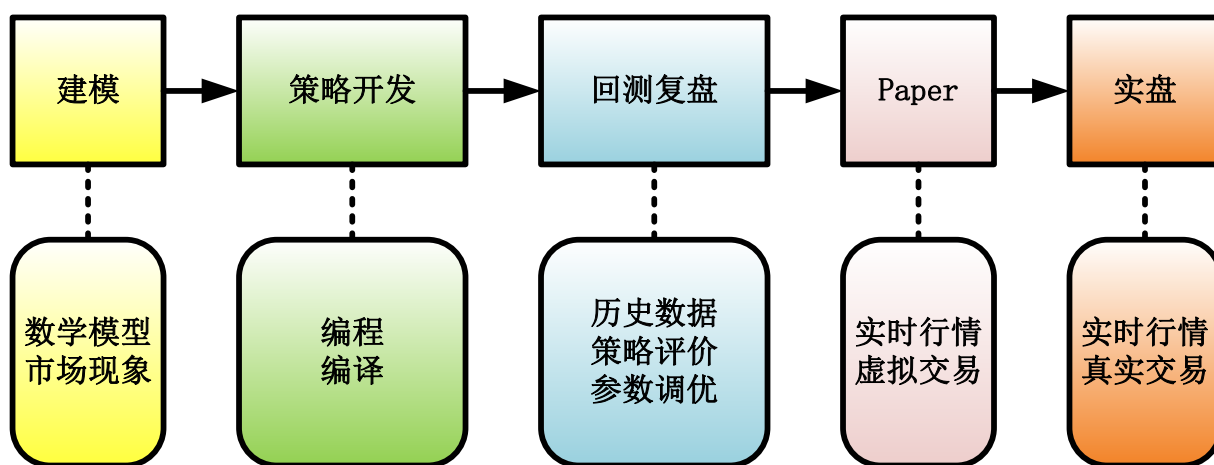
如果方法有返回值，在代码块的最后必须返回 `return` 相对应类型的数值。

```
double GetAvgValue(double x, double y, double z)
{
    double tmp = x + y + z;
    return tmp / 3;
}
```

3 核心概念

3.1 策略

策略([Strategy](#))是在市场中应对所有可能发生情况的一整套逻辑规则完整行动计划。一个完整的策略开发生命周期由五个环节构成：



- (1) 建模：金融工程师根据某些特定的数学模型、市场规律现象，构建出交易思想，进而整理形成策略。
- (2) 策略开发：通过计算机语言将策略编写为机器可以识别的代码，计算机编译策略代码生成自动化交易程序。
- (3) 回测复盘：基于历史数据（[Tick](#) 和 [K 线](#)），驱动策略运行，产生买卖信号，依据虚拟的成交记录计算出资产线，最大回撤、夏普比率等策略评价指标，并进一步对参数进行调优，以搜索出最佳参数。
- (4) 执行仿真交易，使用经纪商提供的模拟账户，用实时行情源驱动策略运行，观察策略在实盘下的表现。
- (5) 实盘：使用实时行情源驱动策略运行，集成实盘的下单通道，连接券商或者期货经纪商的柜台系统，用真实资金进行交易。

3.2 Tick

我国的股票市场和期货市场的 **Tick** 是若干秒一次的行情采集期间累计的成交量和最后的成交价，可能是几笔成交的集合。我国的期货市场目前是 500 毫秒推送一个 **Tick**；深交所约每 3 秒，上交所约每 5 秒推送一个 **Tick**。

通过传统的行情软件，我们可以看到 **Tick** 数据是实时地从先到后推送给到客户端的。**Tick** 是行情的最精确单位，是交易时间的最小颗粒。对于高频策略，实盘中捕捉超短线行情的细微变化需要靠 **Tick** 级别的价格信号；对于复盘，**Tick** 级别的数据能以最高准确度重现策略的绩效。因此对于程序化交易而言，实盘和复盘中的 **Tick** 数据都是必不可少的。这一点在期货投机策略中显得尤为重要。大部分期货策略都是基于 **Tick** 运算执行的。

时间	价格	现量	增仓	性质
15:14	2685.0	21	-17	空平
15:14	2685.0	23	-17	空平
15:14	2684.8	110	-35	多平
15:14	2684.8	16	-14	多平
15:14	2684.8	34	-32	多平
15:14	2684.6	42	-38	多平
15:14	2684.2	13	-10	多平
15:14	2684.2	20	-10	多平
15:14	2684.2	19	-13	空平
15:14	2684.2	18	-12	多平
15:14	2684.2	26	-17	空平
15:14	2684.0	14	-9	多平
15:14	2684.8	24	-3	空平
15:14	2684.6	30	-15	多平

期货 Tick 行情

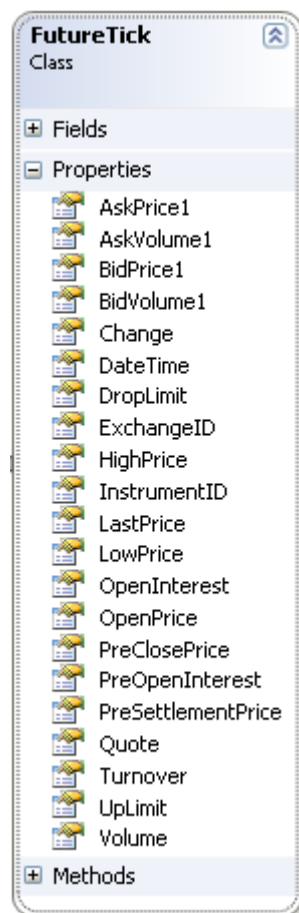
14:56	17.49	1165	B	87
14:56	17.48	53	B	5
14:56	17.49	153	B	12
14:56	17.50	2948	B	262
14:56	17.48	50	B	3
14:56	17.48	21	S	2
14:56	17.49	46		4
14:56	17.49	20	S	2
14:56	17.48	25	S	3
14:56	17.48	11	S	3
14:56	17.48	1	S	1
14:56	17.48	23	S	5
14:56	17.50	364	B	14
14:56	17.48	36	S	3
15:00	17.48	7084		0

股票 Tick 行情

MQ 中对于期货 **Tick** 和股票 **Tick** 都用统一的一个类 **Tick** 来表示。

期货和股票 **Tick** 中包含了期货和股票行情推送的实时信息，如最新价格、盘口、昨结算价等。这些信息都是 **Tick** 的属性。

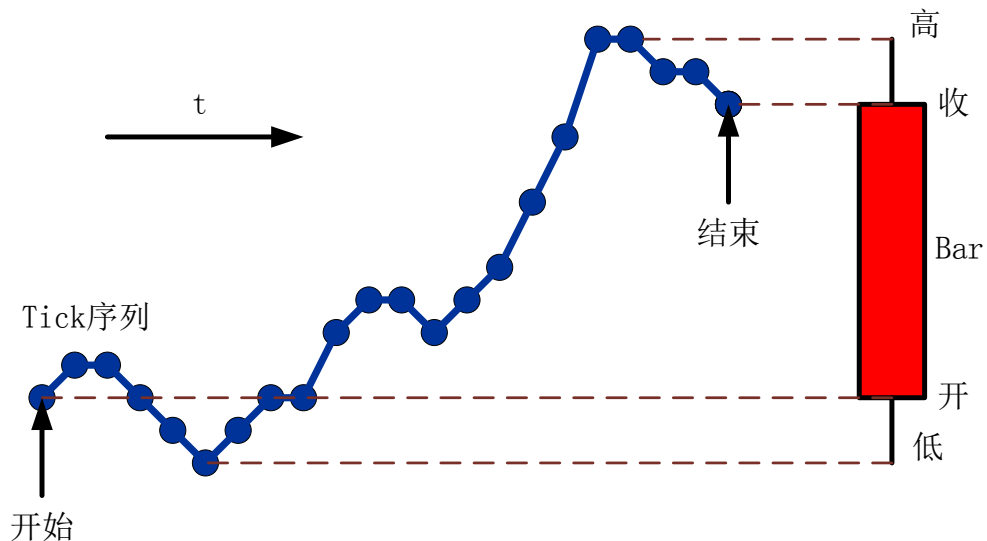
属性	含义
LastPrice	最新价格
Change	涨跌幅，基于昨结计算，单位是%
Volume	成交量
TurnOver	成交额
AskPrice1	卖一价
BidPrice1	买一价
Quote	盘口
OpenPrice	当天的开盘价
HighPrice	当天的最高价
LowPrice	当天的最低价
ClosePrice	收盘价，盘中相当于最新价
PreClosePrice	昨天的收盘价
PreSettlementPrice	昨天的结算价 ¹
OpenInterest	持仓量
UpLimit	涨停价
DropLimit	跌停价



¹由于数据源的原因，MQ 复盘的早期历史数据中的昨结是用昨收近似代替的。实盘交易时行情来自 CTP，其昨结是准确的。

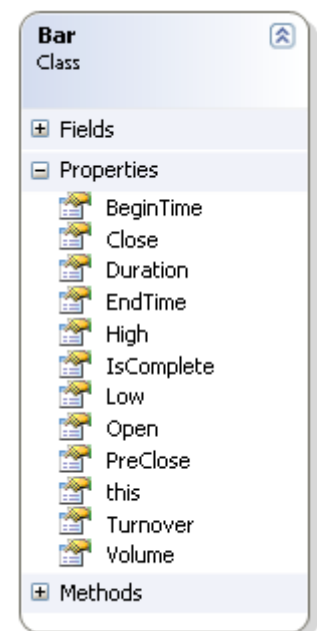
3.3 Bar

在一定时间段内的 **Tick** 序列就构成了一根 K 线（日本蜡烛图）。在 MQ 中，单根 K 线被称为 **Bar**。



股票 **Bar** 和期货 **Bar** 的数据结构是一样的，在 MQ 中都用 **Bar** 来表示。**Bar** 既包含了时间维度的信息，例如起止时间，也包括了空间上的信息，例如高开低收，还包含了成交量的信息。

属性	含义
Open	开
High	高
Low	低
Close	收
TurnOver	成交额
Volume	成交量
IsComplete	是否已经结束



策略研究的核心对象之一是价格和成交量构建成的时间序列。**Bar** 就是时间维度上价格在空间维度上变化构成的数据单元。如下图所示，多个数据单元 **Bar** 就构成了一个时间序列 **BarSeries**。



属性	含义
Open	开盘价序列
High	最高价序列
Low	最低价序列
Close	收盘价序列
TurnOver	成交额序列
Volume	成交量序列

3.4 事件驱动

整个 MQ 策略的运行是基于事件驱动的，这种事件驱动的机制使得 MQ 策略可以并行执行算法，及时响应各种事件的发生。所谓事件驱动就是你点什么按钮(即产生什么事件)，计算机就执行什么操作(即调用什么函数)。在 MQ 策略中用户点击开始，计算机就执行 **Init**，一个 **Tick** 数据到来，计算机就执行 **OnTick**，一根 K 线走完，计算机就执行 **OnBar**。事件驱动在整个策略开发中是很重要的概念，大家需要熟练掌握。期货、股票和期权策略都支持事件驱动。

事件就是在交易活动中发生的各种事情 (**Event**)。各种复杂的事件是驱动 MQ 策略运行的动力，其基本模式为：

每当 XXX 时，进行一次计算；

或者：

每当 XXX 时，驱动策略运行；

把 XXX 换成下面的内容：

到来一个新的 **Tick**

委托成交了

撤单成功了

委托被拒绝了

委托的状态变化了

就是事件驱动策略运行的机理。

MQ 中提供了开始【**Init**】、退出【**Exit**】、【**OnTick**】、【**OnAllTicks**】、【**OnBarOpen**】、【**OnBar**】、定时任务【**Task**】、委托回报【**OnOrder**】、成交回报【**OnTrade**】、委托拒绝【**OnOrderRejected**】、撤单成功【**OnOrderCanceled**】、撤单失败【**OnCancelOrderFailed**】以及系统报警【**OnSysWarning**】这 13 个事件。在未来的版本中还将提供更多复杂的事件，以满足复杂精细的策略微结构需求。

事件	关键词	触发
开始	<code>Init</code>	用户启动策略时触发
退出	<code>Exit</code>	用户停止策略时触发
<code>Tick</code> 数据到来	<code>OnTick</code>	当 <code>Tick</code> 数据到来时触发
多个 <code>Tick</code> 数据到来	<code>OnAllTicks</code>	当同一个时间切片的所有 <code>Tick</code> 到来时触发
<code>OnBar</code>	<code>OnBar</code>	当 <code>K</code> 线走完时触发
<code>OnBarOpen</code>	<code>OnBarOpen</code>	当 <code>K</code> 线刚产生时触发
定时任务	<code>Task</code>	到了指定的时刻时触发
委托回报	<code>OnOrder</code>	当委托状态改变时触发
成交回报	<code>OnTrade</code>	当返回一个成交时触发
委托拒绝	<code>OnOrderRejected</code>	当委托被拒绝时触发
撤单回报	<code>OnOrderCanceled</code>	撤单成功时触发
撤单失败回报	<code>OnCancelOrderFailed</code>	撤单失败时触发
系统警告回报	<code>OnSysWarning</code>	系统底层报警（例如自成交）

3.5 线程安全

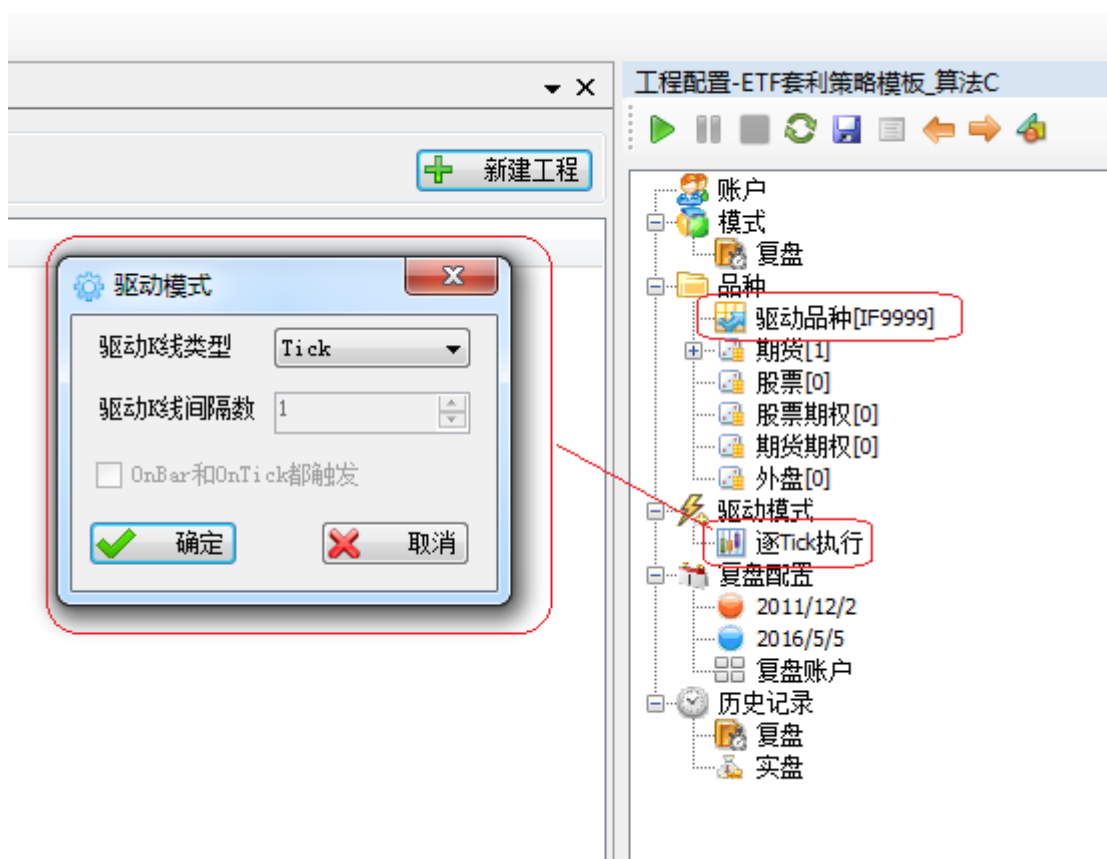
MQ 底层内核做了线程安全处理, 即 **MQ** 的各种回报和触发的事件(例如 **OnTick**, **OnTrade**) 等都是线程安全 **Thread Safe** 的。例如在代码执行 **OnTrade** 的逻辑过程中, 即使到来一个新的 **Tick**, **OnTick** 的代码也不会打断正在进行计算的 **OnTrade** 代码段, 而是要等到 **OnTrade** 代码段执行完毕后, 再执行新到的 **OnTick** 事件。

注意: 如果用户自己启动了新的线程, 则该线程不受 **MQ** 控制, 也没有线程安全的特性。因此在用户自己启动的线程中, 可能出现变量读写冲突的问题, 用户需要自己用锁等技术解决。不建议用户在 **MQ** 的策略中启动非受控线程。

3.6 策略运行

➤ 驱动品种

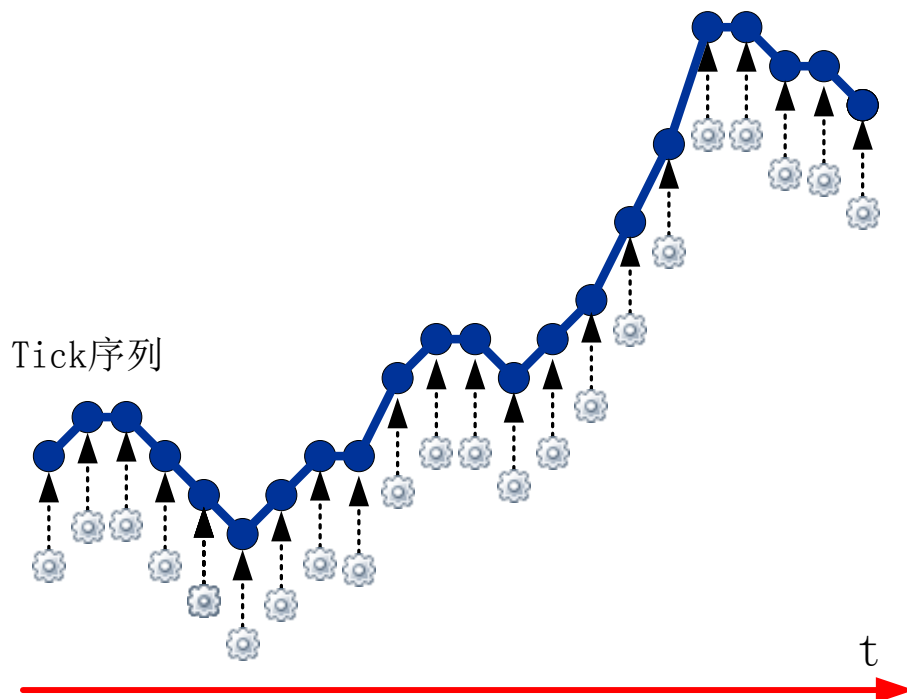
策略运行一般会设置一个驱动品种，如果是单纯期货策略，采用期货交易品种作为驱动品种；如果是单纯股票策略，则采用上证指数【000001.SH】、深证成指【399001.SZ】等交易所指数作为驱动品种。如果采用个股作为驱动品种，当遇到个股停牌时，策略就会由于没有 Tick 数据而无法正确执行。如果是期货股票的混合策略，一般采用 Tick 周期短的品种，即期货交易品种作为驱动品种。如果在期货股票的混合策略中采用股票行情驱动，则有可能由于股票行情更新速度太慢而错过期货的交易机会。



➤ Tick 驱动

当选择 Tick 驱动方式时，每当新来一个 Tick，算法都会被执行一次。

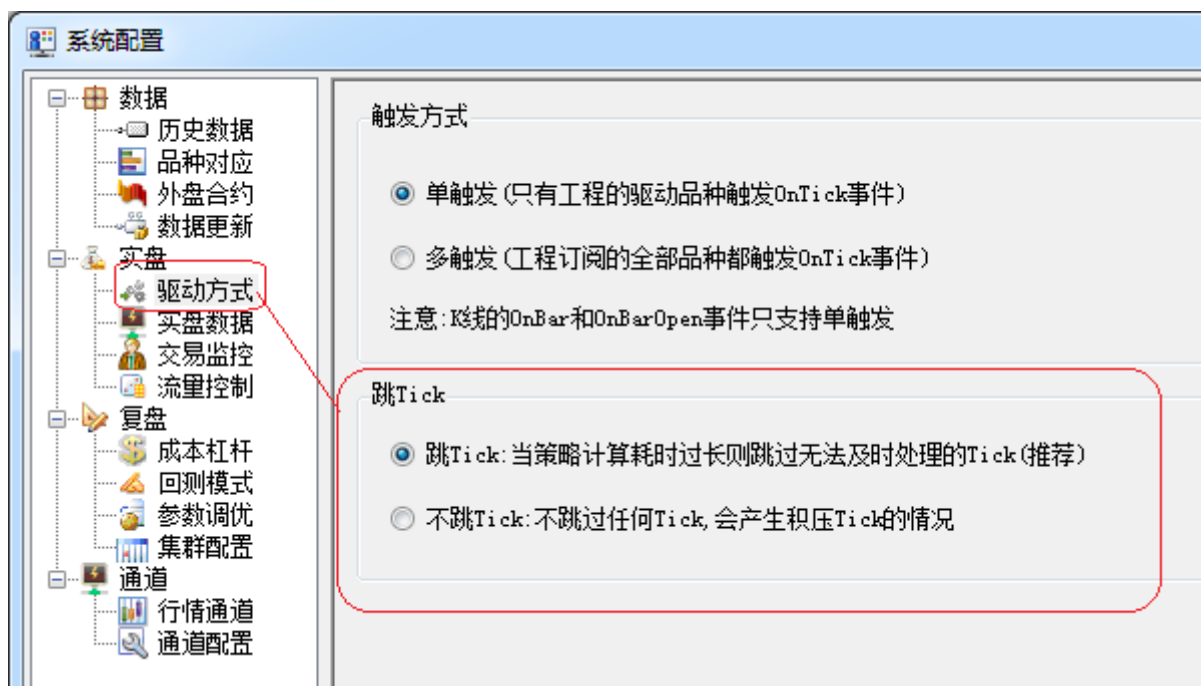
注意：对于某些非主力合约或者成交稀少的股票，如果一段时间内没有 Tick 数据推送给客户端，由于没有数据到来事件，这段时间的算法就不会被触发执行。



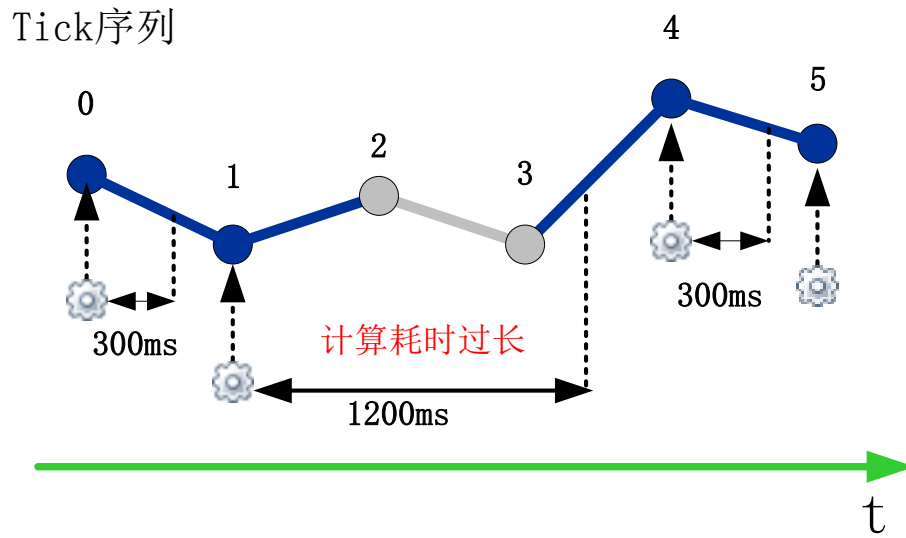
➤ 跳 Tick

在实盘交易中，若算法计算较复杂或者中间有文件读写、查询账户等耗时操作导致算法执行时间长于两个 Tick 间的时间间隔（例如期货的 Tick 间隔为 500 毫秒），程序会执行当前完成至当前代码的最后一行，然后在最新的 Tick 到来后执行下一次的算法计算。因此两次运算中间可能会有部分 Tick 没有参与计算，这在 MQ 中被称为“跳 Tick”。

为了确保策略逻辑能顺利执行，不会由于计算过慢而积压 Tick，MQ 默认是跳 Tick。配置方法是打开【工具】→【系统配置】→【实盘】→【驱动方式】，根据需要配置“跳 Tick”。



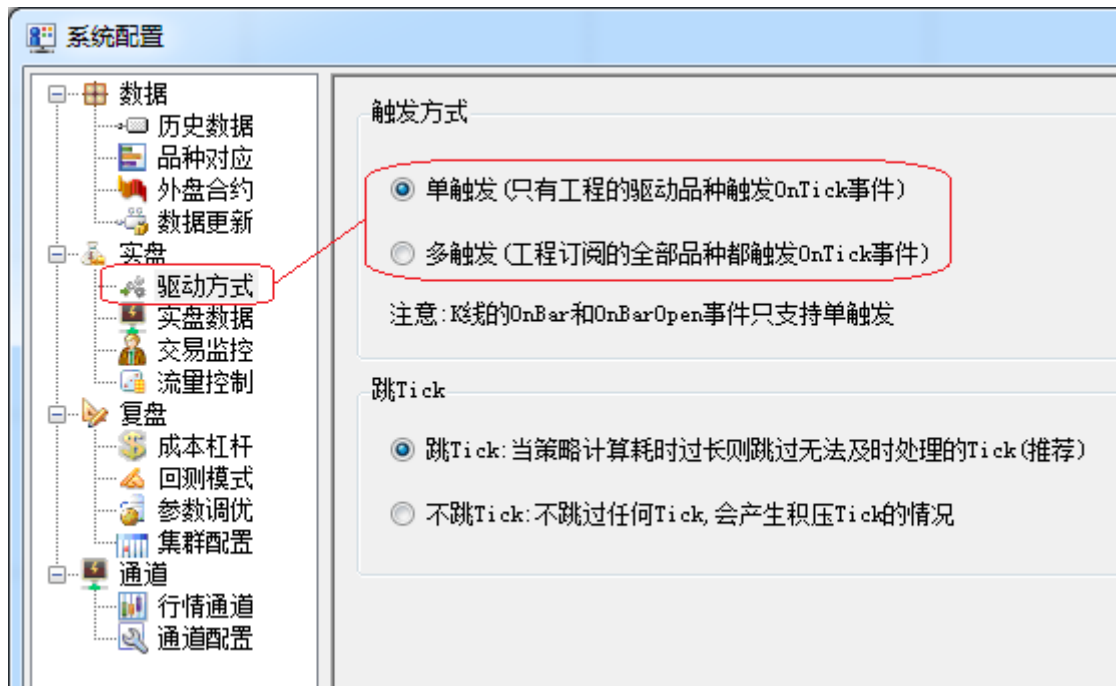
在下面的例子中，期货 Tick 序列的时间间隔为 500ms，算法执行一次平均耗时为 300ms，所以第 Tick[0]、Tick[1]、Tick[5] 都可以参与计算。但是在 Tick[1] 时，由于某种原因（如执行了大文件读写，打开数据库连接等耗时操作），算法计算耗时过长，超过了两个 Tick 的间隔，导致 Tick[2] 和 Tick[3] 都没有参与计算，而是等到算法执行完成后到来了最新的 Tick[4]，开始新一轮算法计算。



所以在实盘策略尤其是高频策略的开发中要特别注意算法的效率和避免不必要的文件读写，尽可能使用内存中的数据，以提高算法执行的速度，在交易中争取先机。

➤ 多驱动

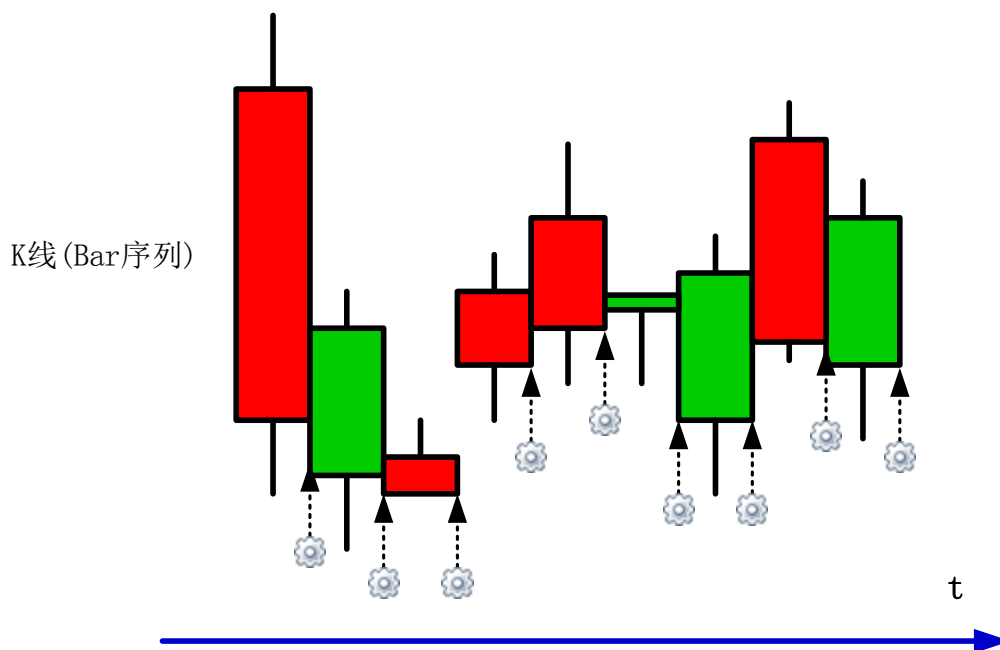
如果多品种策略需要每个品种的 **Tick** 到来时都触发 **OnTick** 事件，可以在系统配置的驱动方式页面里将【多触发】选中。



➤ K 线驱动

K 线驱动就是按照 **Bar** 来驱动策略运行，由于 K 线的周期是固定不变的，因此 K 线驱动等效于按照固定时间间隔执行算法。

K 线驱动只有驱动品种有效，且只支持单触发。



➤ OnTick+OnBar

策略如果同时支持 **OnTick** 和 **OnBar** 方法，可以在工程的驱动模式的配置界面上勾选中【**OnBar 和 OnTick 都触发**】。



➤ OnAllTicks

多品种策略如果需要同时处理一组到达时间相同或者相近的多个 **Tick**, 可以采用【**AllTicks**】的驱动类型。在策略中可以重载 **OnAllTicks** 方法, 就可以获取底层经过对齐处理的一组 **Tick**。

```
/// <summary>
/// 一组Tick到来
/// </summary>
/// <param name="buffer">Tick组</param>
public override void OnAllTicks(Dictionary<string, Tick> buffer)
{
    Print("一大波Tick到来");
    foreach (string instrumentId in buffer.Keys)
    {
        //品种instrumentId的Tick
        Tick tick = buffer[instrumentId];
        Print(tick.ToString());
    }
}
```

3.7 代码结构

MQ 的策略代码结构由代码头、参数、变量、初始化、执行、退出、定时任务、委托回报、成交回报等九个部分构成。下面是一个空策略的模板，**Strategy** 的关键代码架构都已经写好了，读者可以基于这个模板开发策略，在适当的位置填写自己策略的代码。

```
using System;
using System.Collections.Generic;
using Ats.Core;
using Ats.Indicators;
namespace MagicQuantTest
{
    public class 空策略 : Strategy
    {
        /// <summary>
        /// 策略启动
        /// </summary>
        public override void Init()
        {
        }

        /// <summary>
        /// 策略停止
        /// </summary>
        public override void Exit()
        {
        }

        /// <summary>
        /// 单个Tick到来
        /// </summary>
        /// <param name="tick">单个Tick</param>
        public override void OnTick(Tick tick)
        {
        }

        /// <summary>
        /// 一批Tick到来
        /// </summary>
        /// <param name="buffer">一组Tick</param>
        public override void OnAllTicks(Dictionary<string, Tick> buffer)
        {
        }

        /// <summary>
        /// 一根K线走完
        /// </summary>
        /// <param name="bar"></param>
        public override void OnBar(Bar bar)
        {
        }

        /// <summary>
        /// 一根K线创建
    }
}
```

```
/// </summary>
/// <param name="bar"></param>
public override void OnBarOpen(Bar bar)
{
}

/// <summary>
/// 成交回报
/// </summary>
/// <param name="trade">成交</param>
public override void OnTrade(Trade trade)
{
}

/// <summary>
/// 委托状态变化
/// </summary>
/// <param name="order">状态变化的委托</param>
public override void OnOrder(Order order)
{
}

/// <summary>
/// 委托拒绝
/// </summary>
/// <param name="order">被拒绝的委托</param>
public override void OnOrderRejected(Order order)
{
}

/// <summary>
/// 撤单成功
/// </summary>
/// <param name="order">成功撤单的委托</param>
public override void OnOrderCanceled(Order order)
{
}

/// <summary>
/// 撤单失败
/// </summary>
/// <param name="orderGuid">撤单失败的委托GUID</param>
/// <param name="msg">失败消息</param>
public override void OnCancelOrderFailed(string orderGuid, string msg)
{
}

/// <summary>
/// 系统警告
/// </summary>
/// <param name="arg"></param>
public override void OnSysWarning(SysWarningEventArgs arg)
{
    if (arg.ID == SysWarningID.自成交)
    {
        Print("注意自成交");
    }
}
```

```
}  
}
```

➤ 代码头

代码头位于整个代码的顶端，用于定义策略需要用到的命名空间。策略必须引用 [System](#)、[Ats.Core](#) 以及 [Ats.Indicators](#)。

```
//代码头  
using System;  
using Ats.Core;  
using Ats.Indicators;
```

下表列出了一般常用的类库引用方法。

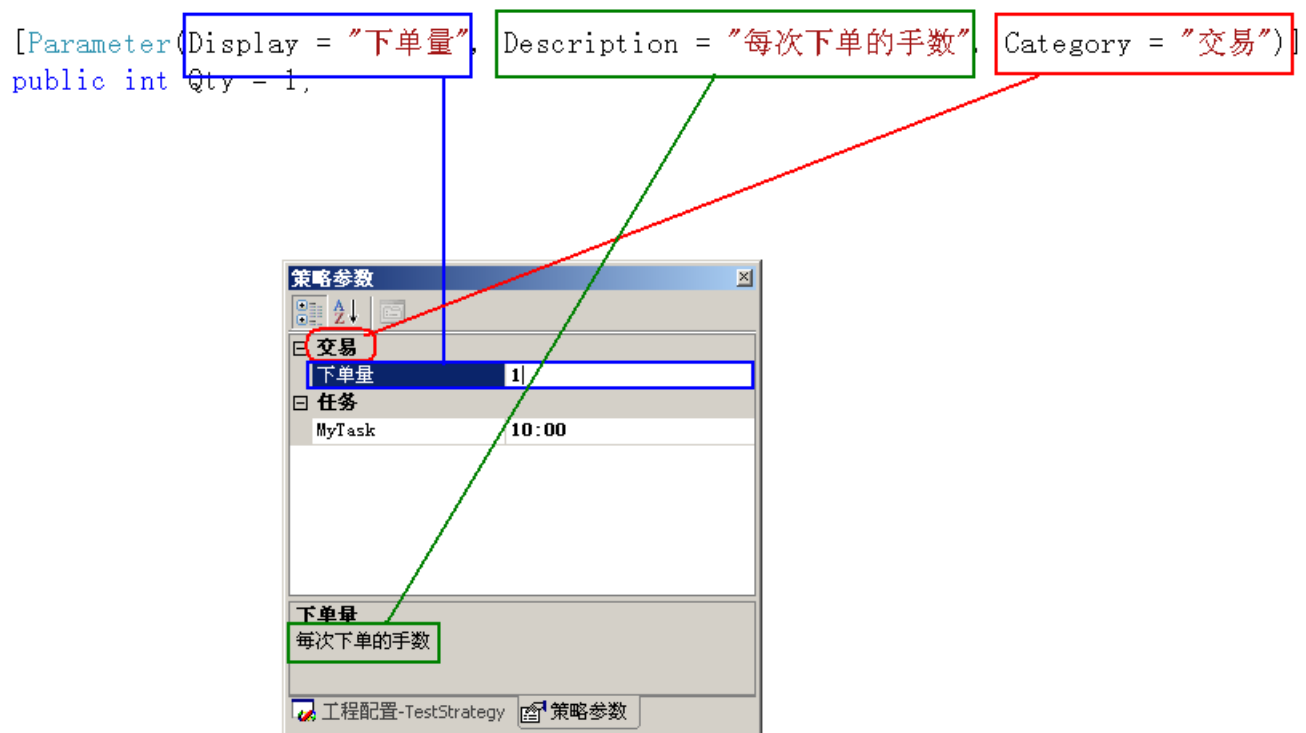
类库	主要功能	说明
System	底层系统类库	策略必须，系统内核
Ats.Core	MQ 内核方法	策略必须，MQ 内核
Ats.Indicators	MQ 指标	指标必须
System.Text	文本类	
System.Threading	线程类	
System.Windows.Forms	窗体类	UI 相关
System.IO	文件读写类	读写外部文件
System.Collections.Generic	集合类	处理 List、Dictionary 等集合
System.Xml	XML 类	处理 XML 文件

➤ 参数

参数是策略中用户可以配置的变量，共四种类型：字符串、整型、浮点型和布尔型。参数的变量名是代码中用来引用参数值的，显示名称【**Display**】是配置参数界面中显示的参数名。参数的描述【**Description**】是对参数的简介，以便于用户在界面配置正确理解该参数的含义。参数的分类【**Category**】的作用是在参数比较多时，将参数分门别类，便于配置管理。

用户一般在启动策略前配置好参数。策略运行过程中，用户也可以在参数配置界面上修改参数，修改完成后点击窗体别的地方，系统会提示用户确认参数修改。

下面的例子中，参数的变量名是 **Qty**，初始值是 **1**，界面显示的名称是“下单量”，描述是“每次下单的手数”，分类是“参数”。



注意：策略可以通过代码中修改参数值，但仅在运行中有效，一旦策略运行结束，下次启动时，参数又会恢复为当初设置的值。

```
public override void OnTick()
{
    Qty = 2; // 代码中修改参数值
}
```

➤ 变量

不同于参数，策略的变量是不可以由用户在 **MQ** 界面里配置初始值的，因此变量在策略启动时始终是其初始值，运行以后，在策略执行计算时，可以被调用和修改。一旦策略结束，其数据将从内存中释放，下次启动时变量又恢复为初始值。变量名可以是英文，也可以是中文。下面的例子中定义了一个整型变量作为 **Tick** 计数器，变量名为 **clc**，每来一个 **Tick** 就自增 **1**。

```
using System;
using Ats.Core;
namespace MyNameSpace
{
    public class TestStrategy : Strategy
    {
        int clc = 0;
        public override void OnTick(Tick tick)
        {
            clc++;
            string str = "[" + clc.ToString() + "]Tick, t=" + Now.ToString();
            Print(str);
        }
    }
}
```

➤ 初始化

当用户启动策略时，初始化事件被触发，策略执行 `Init` 里面的代码。初始化 `Init` 一般执行策略的准备工作，例如将外部文件数据预读到内存中，构造 `K` 线，设定变量的初始值，显示策略的参数配置等。

下面这个例子中，用户需要调用上证指数的日 `K` 线，回溯的天数 `backDays` 需要从一个外部文件读取。文本读取到的是一个字符串型的变量 `readStr`，通过 `int.Parse` 的方法解析成整型变量 `backDays`。最后 `backDays` 作为 `GetBarSeries` 的参数，构造了上证指数的日 `K` 线。

注意：`K` 线的构造工作一定要放在 `Init` 里面完成，而不能放在 `OnTick` 里面完成。

```
using System;
using Ats.Core;
using System.IO; // 引用文件读写类
namespace MyNameSpace
{
    public class TestStrategy: Strategy
    {
        BarSeries kLine; // 定义变量线，类型BarSeries

        public override void Init()
        {
            string FilePath = "c:\\cfg.txt"; // 定义局部变量，外部文件路径
            string readStr = File.ReadAllText(FilePath); // 将文件中的内容读取到内存中
            int backDays = int.Parse(readStr); // 将字符串解析成整型
            // 构造上证指数的日线，回溯backDays天
            kLine = GetBarSeries(EnumMarket.股票, "000001.SH", 1,
                                EnumBarType.日线,
                                EnumRestoration.不复权,
                                backDays);
        }
    }
}
```

➤ 驱动算法执行

当驱动数据到来事件发生时，策略就会执行 **OnTick** 或者 **OnBar** 里面的代码。一般来说，策略的核心算法都是写在 **OnTick** 或者 **OnBar** 中的。

对于 **Tick** 驱动的策略，每到来一个驱动品种的 **Tick**，**OnTick** 就会被执行一次计算；

对于 **K** 线驱动的策略，每当驱动品种的 **Bar** 结束时，**OnBar** 和 **OnBarOpen** 就会被执行一次计算。

```
public override void OnTick(Tick tick)
{
    //执行算法计算
}

public override void OnBar(Barbar)
{
    //执行算法计算
}

public override void OnBarOpen(Barbar)
{
    //执行算法计算
}
```

➤ 退出

用户停止策略事件触发时，策略会被执行 **Exit** 内的代码。退出 **Exit** 一般用于收盘时和停止策略时的收尾工作，例如将一些信息写盘保存。

下面这个例子中，策略将期货的最新价格保存到文件 **price.txt** 中去。

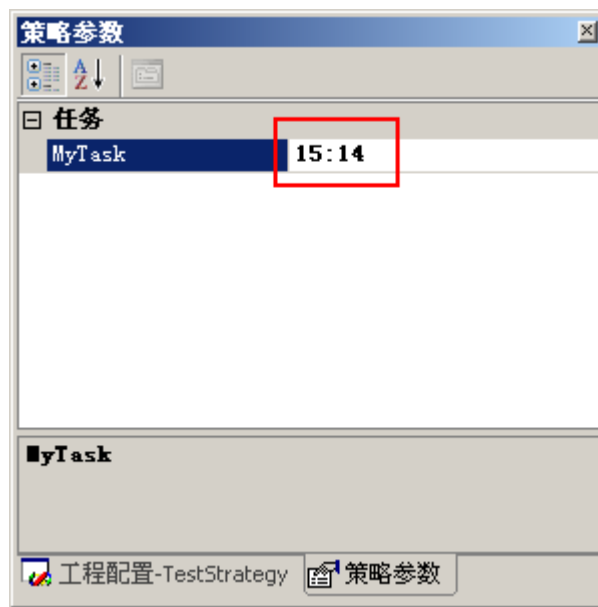
```
public override void Exit()
{
    double price = AllFutures[0].LastPrice; //获取最新价格
    string FilePath = "c:\\price.txt"; //文件路径
    File.AppendAllText(FilePath, price.ToString()); //将最新价格写入文件
    return true;
}
```

注意：如果收盘时，用户不点击停止策略按钮，将不会触发 **Exit** 事件。

➤ 定时任务

定时任务 **Task** 可以在用户指定的时刻 **T** 执行，并且只执行一次。一个策略可以有多个定时任务。定时任务前面需要加任务标志 [**Task**(Time = "15:14")], 设置定时任务的执行时刻。

定时任务的执行时间可以在工程的策略参数面板里面配置。



注意：如果启动策略的时间 **T1** 晚于某个 **Task** 设置的执行时间 **T2**，那么这个 **Task** 中的代码将不会得到执行。

➤ 委托回报

当期货委托状态变化时，会触发委托回报事件，策略可以得到该委托 **Order**。

```
public override void OnOrder(Order order)
{
    //委托状态变化时执行
    Print("委托回报" + order.ToString());
}
```

委托 **Order** 的一些重要属性见下表：

属性	含义
InstrumentID	期货代码（股票代码）
OrderSysID	委托系统 ID（唯一）
ExchangeID	交易所代码
LimitPrice	委托限价
Direction	买卖方向
OpenOrClose	开平标志
Volume	委托手数=未成交手数+成交手数
VolumeLeft	未成交手数
VolumeTraded	成交手数
InsertTime	委托时间

国内交易所代码：上海证券交易所(**SH**)、深证证券交易所(**SZ**)、中国金融交易所(**CFEX**)、上海期货交易所 (**SHFE**)、大连商品交易所 (**DCE**)、郑州商品交易所 (**CZCE**)。

➤ 成交回报

当期货成交时，触发成交回报事件，策略可以得到该成交 **Trade**。

```
public override void OnTrade(Trade trade)
{
    //成交回报时执行
    Print("成交回报" + trade.ToString());
}
```

成交 **Trade** 的一些重要属性见下表：

属性	含义
InstrumentID	品种代码
TradeID	委托成交 ID （唯一）
ExchangeID	交易所代码
OrderSysID	对应的系统委托 ID ，可以通过该 ID 找到对应的 Order
Price	成交价格
Direction	买卖方向
OpenOrClose	开平标志
Volume	成交手数
TradeTime	成交时间

下面的例子演示了如何根据期货成交 **Trade** 的 **OrderSysID** 找到对应的期货委托 **Order**。

```
using System;
using Ats.Core;
using Ats.Indicators;
namespace DemoStrategy
{
    public class 成交匹配委托 : Strategy
    {
        public override void OnTrade(Trade trade)
        {
            string TradeID = trade.TradeID;
            string OrderSysID = trade.OrderSysID;
            OrderSeries OrderLst = GetFutureOrders();
            foreach (Order order in OrderLst)
            {
                if (order.OrderSysID == OrderSysID)
                {
                    Print("成交[" + TradeID + "]对应委托:" + order.ToString());
                }
            }
        }
    }
}
```

➤ 撤单成功回报

当撤单成功时触发 **OnOrderCanceled**。

撤单成功回报往往用在细致的盘中操作，例如追单。

➤ 撤单失败回报

撤单失败是触发撤单失败回报 **OnCancelOrderFailed**。

➤ 系统报警

系统报警是一种风险控制的手段，当 **MQ** 发现一些系统底层风险时，会触发系统报警 **OnSysWarning**。

系统报警的参数是枚举类型 **SysWarningEventArgs**，目前版本支持自成交报警。

```
public override void OnSysWarning(SysWarningEventArgs arg)
{
    if (arg.ID == SysWarningID.自成交)
    {
```


}

}

3.8 事件种类

➤ OnOrder

委托状态改变事件

注意：期货委托报入以后，其状态可能改变多次。我们可以做一个实验，分别用激进挂单和保守挂单的方式测试 **OnOrder**。测试代码如下：

```
public override void Init()
{
    Wait(10 * 1000);
    //策略开始时执行
    Print("DemoStrategy的策略不能用于真实资金账户，只能用于复盘或者模拟测试?");

    Order order = LimitOrder(Qty, 委托价格, EnumBuySell.买入, EnumOpenClose.开仓);
    Print("提交委托时，其ID=" + order.OrderSysID);

    Wait(200);
    Scan();
}

public override void OnOrder(Order order)
{
    count++;
    Print("第" + count + "次委托回报, ID=" + order.OrderSysID);

    Print("OrderSubmitStatus=" + order.OrderSubmitStatus.ToString() + ", Status=" +
order.OrderStatus.ToString());
    Scan();
}

public override void OnTrade(Trade trade)
{
    //应该有多成交
    Print("*****");
    Print("成交对应委托ID=" + trade.OrderSysID);
    Print("成交回报:" + trade.ToString());
    int CurrPos = MarketPosition();
    Print("当前净持仓=" + CurrPos.ToString());
}
```

在上期模拟环境下测试，当策略用激进的价格打单时，策略日志如下：

```
Info[2013/7/10 20:18:10]$期货委托:开仓买入 1 手[IF1307]限价 2148 元@20:18:10.669
Info[2013/7/10 20:18:10]提交委托时，其 ID=
Info[2013/7/10 20:18:10]启动策略线程...
Info[2013/7/10 20:18:10]启动策略成功
Info[2013/7/10 20:18:10]第 1 次委托回报,ID=      470214
Info[2013/7/10 20:18:10]OrderSubmitStatus=InsertSubmitted,Status=AllTraded
Info[2013/7/10 20:18:11]*****
Info[2013/7/10 20:18:11]成交对应委托 ID=      470214
Info[2013/7/10 20:18:11]成交回报：期货 # 买入，开
仓,Volume=1,[IF1307]Price=2148.000,TradeTime=2013/7/11
20:17:54,ExchangeID=SHFE,BrokerID=1026,TradeID=118203,OrderSysID=470214,
Info[2013/7/10 20:18:11]当前净持仓=1
而用保守方式挂单时，策略日志如下：
Info[2013/7/10 20:20:23]$期货委托:开仓买入 1 手[IF1307]限价 2153.2 元@20:20:23.149
Info[2013/7/10 20:20:23]提交委托时，其 ID=
Info[2013/7/10 20:20:23]启动策略线程...
Info[2013/7/10 20:20:23]启动策略成功
Info[2013/7/10 20:20:23]第 1 次委托回报,ID=      480126
Info[2013/7/10 20:20:23]OrderSubmitStatus=Accepted,Status=NoTradeQueueing
Info[2013/7/10 20:21:19]第 2 次委托回报,ID=      480126
Info[2013/7/10 20:21:19]OrderSubmitStatus=Accepted,Status=AllTraded
Info[2013/7/10 20:21:19]*****
Info[2013/7/10 20:21:19]成交对应委托 ID=      480126
Info[2013/7/10 20:21:19]成交回报:期货 # 买入,开仓
,Volume=1,[IF1307]Price=2153.200,TradeTime=2013/7/11
20:21:02,ExchangeID=SHFE,BrokerID=1026,TradeID=121668,OrderSysID=480126,
Info[2013/7/10 20:21:19]当前净持仓=2
Info[2013/7/10 20:22:17]停止策略...
```

从策略中可以看出，由于激进打单，只出现了一次 **OnOrder**，而在保守打单中出现了 2 次

OnOrder。

➤ OnTrade

成交回报事件。

一个 Order 可能对应了一笔或者多笔成交,也就是说可能会触发一个或者多个 OnTrade。

例如开仓买入 10 手, 可能分成 3 笔成交: $10=2+3+5$ 。

成交回报在策略开发中非常重要, 经常运用在:

- 先开后平の場合: 直接挂止盈单
- 先平后开の場合: 反手

确认策略的委托成交, 改变状态位

➤ OnOrderCanceled

撤单成功回报事件

注意: 撤单成功只会触发一次; 如果一个委托成交了一部分, 也是可以撤单的

撤单成功回报在策略开发中非常重要, 经常运用在追单等场合。

➤ OnOrderRejected

委托被拒绝回报事件

在使用 **OnOrderRejected** 事件时，要注意区分委托被拒绝的原因。通过 **Order** 的 **OrderRejectReason** 属性可以获得被拒绝的原因。一般常见的拒绝委托的原因包括：

拒绝原因	说明
每秒请求超许可数	流控，报单频率过高被柜台拒绝
价格非最小单位倍数	价格不是 PriceTick 的整数倍，例如买入开仓 IF1609 价格=2699.265
价格高于涨停板	
价格低于跌停板	
资金不足	账户资金不够开仓或者买入股票
平今仓位不足	今仓数量不够平仓
平昨仓位不足	昨仓数量不够平仓
不支持报单类型	报单类型不支持，例如 FAK ， FOK
不支持套利类型报单	套利单类型不支持
保值额度不足	
网络连接失败	和柜台之间的网络连接中断
延迟开仓被 MQ 拒绝	委托数超过流量限制， MQ 拒绝延迟发送开仓单
MQ 暂停开仓	MQ 暂停发送开仓单若干秒
报单数量错误	委托数量超出交易所规定的范围，例如买入-1 股股票
提交委托失败	
未知	不知道的错误的，更详细的信息查看 Order 的 StatusMsg 字段

3.9 委托拒绝处理

通过委托拒绝事件，我们会知道委托被拒绝了，那么怎么才能避免这些非常规问题的出现呢？在 MQ 里，是由一系列的，基于实盘交易经验总结的常规处理方法的。

➤ 如何处理流控

那么，如果因为流控，触发了“每秒请求超许可数”的情况，应该如何处理呢？最简单的办法是在委托之间人为的加入延迟，例如：

连续预埋多个单子：

LimitOrder

LimitOrder

LimitOrder

...

这样肯定会触发流控！如果加入延迟以后，就不会触发流控了：

LimitOrder

Wait(200);

LimitOrder

Wait(200);

LimitOrder

...

➤ 如何避免价格非最小单位整数倍

在策略中，有时候，我们需要对几个数量进行加减乘除，以得到报单价格，这样得到的价格往往不是最小单位的整数倍，例如股指期货有可能计算出限价委托 2600.3965487，这样的价格如果直接报入，会被拒绝的。一个简便的解决办法是调用 MQ 提供的方法

TrimFuturePrice

这个方法会将价格整理到跳（PriceTick）的整数倍数上去，从而避免价格非最小单位整数倍的问题。

➤ 资金不足怎么处理

首先我们可以知道：资金不足对应的一定是开仓委托。那么可以根据需要对策略进行“回滚”(RollBack):放弃开仓,并将标志位恢复原位。

➤ 平今仓位不足怎么处理

如果平今仓位不足，需要分析原因：为什么会发出平仓单？要么是策略状态位错了；要么是手工把持仓给平了；

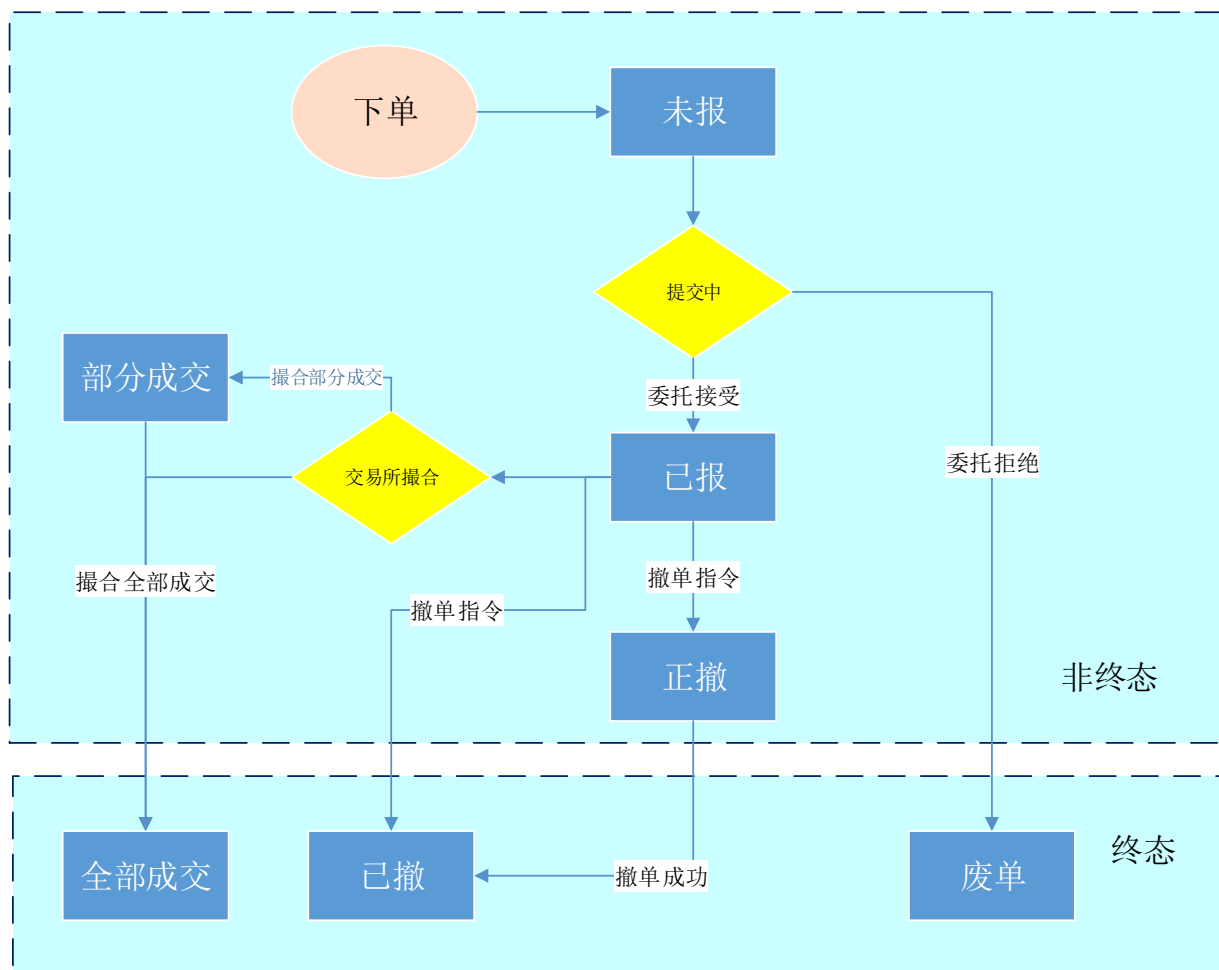
注意：不管是什么原因，当柜台系统已经反馈平今不足时，不能再发出平今委托，否则会出现死循环的情况。

3.10 委托状态

委托共有 7 种委托状态 `MQOrderStatus`，分为两类，一类是非终态；一类是终态。委托状态会随着交易过程而变化。通过 `Order` 的 `MQOrderStatus` 属性可以访问该委托的委托状态。

委托状态	类型	说明
未报	非终态	还没有提交
已报	非终态	已经提交
部分成交	非终态	已经提交，并且成交了一部分，但没有全部成交： <code>Order.Volume > Order.VolumeTraded</code>
正撤	非终态	已经发出撤单动作，还没有收到撤单回报
全部成交	终态	委托全部成交 <code>Order.VolumeTraded = Order.Volume</code>
废单	终态	委托被平台或者柜台系统拒绝
已撤	终态	委托已经撤单成功（可能有部分成交，也可能没有成交）

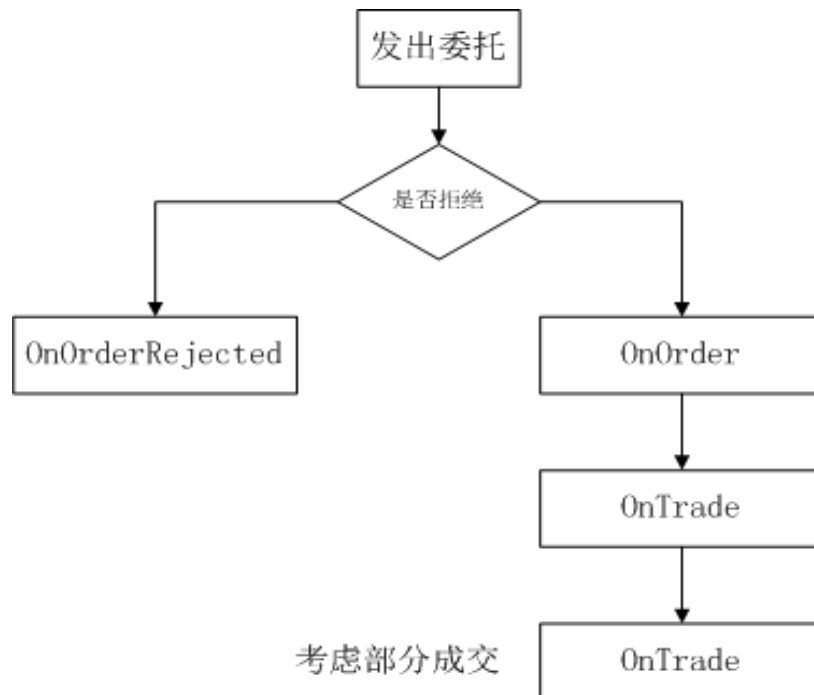
下图显示了委托状态的变化过程。



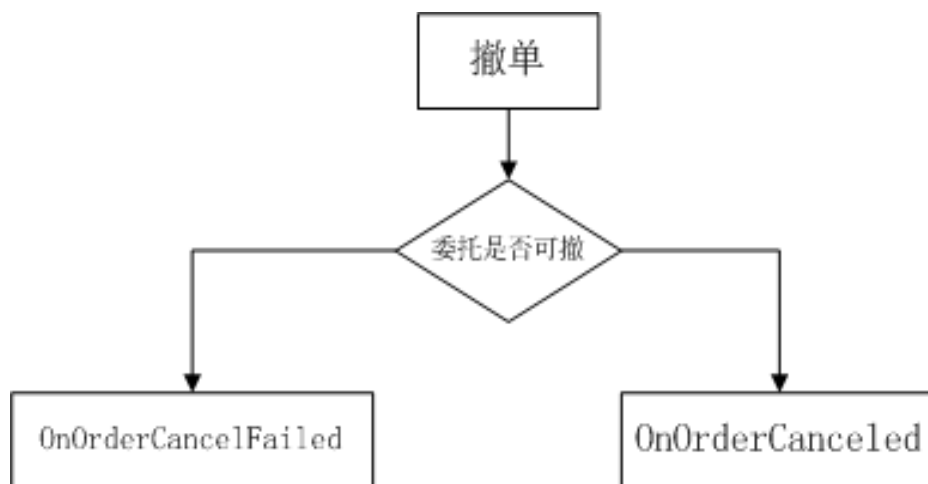
3.11 事件流图

事件流图是发出一个委托后，可能触发的事件流程图。理顺事件流图，对于理解精细结构的交易策略至关重要。

➤ 提交委托的事件流图



➤ 撤单的时间流图



上面这 2 个事件流图非常简单,但是后面要实现的更为复杂精细逻辑的基础。

3.12 事件隔离

同一个账户，运行 2 个策略，A 策略和 B 策略。

A 策略开仓，成交了，OnTrade 事件会在哪里触发呢？有四种可能：

- ①在 A 中触发，在 B 不中触发
- ②在 A 中不触发，在 B 中触发
- ③在 A 中触发，在 B 中也触发
- ④在 A 中不触发，在 B 不中触发

MQ 中的情况是：①在 A 中触发，在 B 不中触发。这就是事件隔离的基本概念。

注意：如果没有事件隔离，会导致策略逻辑混乱。

4 策略开发

在有了基本的 **C#** 语言知识和了解核心的概念之后，我们就可以开始编写自己的策略了。策略所做的工作可以简单的理解成：获取信息，然后根据获取的信息按照一定的逻辑规则执行操作。在 **MQ** 中，策略获取的信息包括：品种信息、实时行情、回溯历史数据；执行的操作包括：执行交易和交互。**MQ** 的策略都是继承自 **Ats.Core** 类库的 **Strategy** 基类。只要熟悉了 **Strategy** 基类提供的各种属性和方法，我们就能灵活自如的开发策略。

注意：在 **MQ** 里为了灵活的支持多个市场（股票，期货，股票期权和期货期权），大多数方法都是通过市场类型 **Market** 来指定操作的市场。

4.1 品种

MQ 的工程配置面板中可以配置策略订阅的品种。一个策略可以订阅多个股票和多个期货品种。策略是由这些品种中的一个驱动执行 **OnTick** 或者 **OnBar**，这个品种就是驱动品种 **TriggerInstrument**。

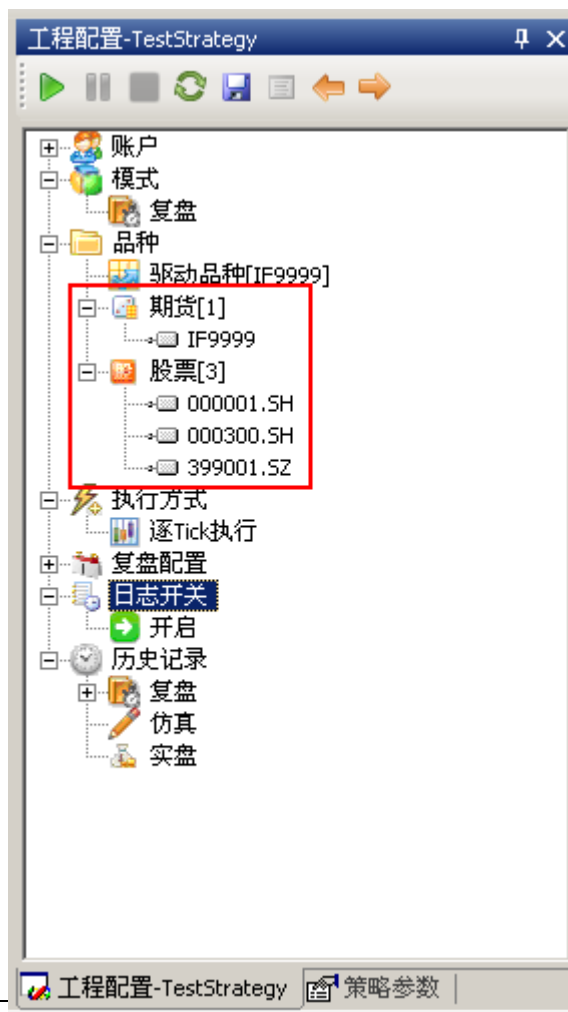
➤ 获取品种

策略订阅的股票和期货是分开记录的。股票队列是 **AllStocks**，期货队列是 **AllFutures**。如果要获取品种，可以通过索引访问：

```
Future future = AllFutures[0];
Stock stock = AllStocks[0];
```

根据索引从队列中获取品种时，需要注意索引不能超过队列长度。

```
int index = 3;
if (AllFutures.Count > index)
{
    Future future = AllFutures[index];
}
```



策略还可以通过品种代码直接访问品种：

```
Future future = AllFutures["IF1203"];
Stock stock = AllStocks["000001.SH"];
```

根据代码从队列中获取品种时，要考虑到没有订阅该品种的情况。

```
string IfCode = "IF1203";
if (AllFutures.Contains(IfCode))
{
    Future future = AllFutures[IfCode];
}
```

➤ 默认期货品种

为了使代码更加简洁，策略中引入了默认期货品种 **DefaultFuture** 的概念。

当策略所在的工程订阅了 1 个或者多个期货品种时，默认期货 **DefaultFuture** 是订阅的第一个期货品种，相应的 **DefaultFutureCode** 是 **DefaultFuture** 的代码。

注意：驱动品种是用户配置指定的；默认品种是订阅的第一个品种，不一定是驱动品种。

例如一个工程依次订阅了期货合约 **IF1205**、**IF1206** 和 **Cu1207**，那么默认品种就是 **IF1205**。

➤ 默认股票品种

当策略所在的工程订阅了 1 个或者多个股票品种时，默认股票 **DefaultStock** 就是第一个股票品种，相应的 **DefaultStockCode** 是 **DefaultStock** 的股票代码。

➤ 品种属性

获取品种以后，就可以直接访问品种的属性。下面的例子是访问期货品种的常用属性

```
Future future = AllFutures[0];
string str = future.ID;
str += ",最新价=" + future.LastPrice;
str += ",合约名称=" + future.Name;
str += ",昨收=" + future.PreClose;
str += ",合约数量乘数=" + future.VolumeMultiple;
str += ",涨停价=" + future.UpLimit;
str += ",跌停价=" + future.DropLimit;
Print(str);
```

下面是访问股票品种的常用属性：

```
Stock stock = AllStocks[0];
string str = stock.ID;
str += ",最新价=" + stock.LastPrice;
str += ",股票名称=" + stock.Name;
str += ",昨收=" + stock.PreClose;
```

```
str += ",涨停价=" + stock.UpLimit;  
str += ",跌停价=" + stock.DropLimit;  
Print(str);
```

下表列出了股票 **Stock** 的重要字段。

属性	含义
ID	股票代码，9 位代码，沪市后缀.SH，深市后缀.SZ
ExchangeID	交易所编号
Name	股票名称
LastTick	最新 Tick
LastPrice	最新价格
PreClose	昨收
UpLimit	涨停价
DropLimit	跌停价

注意：上表**红色部分属性**只有在订阅了该品种，并且有行情数据时才有效。因此在使用这些属性前，需要作非空检验：

```
if( stock.LastTick!=null )
```

下表列出了期货 **Future** 的重要字段。

属性	含义
ID	期货品种代码
Name	期货品种名称
ProductID	期货产品代码
ExchangeID	交易所代码
VolumeMultiple	合约数量乘数，例如股指期货是 300
PriceTick	最小变动价位，即“跳”，例如股指期货一跳是 0.2 点
OpenDate	上市日
ExpireDate	到期日
LongMarginRatio	多头保证金率，例如股指期货是 0.15
ShortMarginRatio	空头保证金率
IsArbitrage	是否套利合约
LastTick	最新 Tick
LastPrice	最新价
PreClose	昨收
PreSettlementPrice	昨结
UpLimit	涨停价
DropLimit	跌停价

注意：上表红色部分属性只有在订阅了该品种，并且有行情数据时才有效。因此在使用这些属性前，需要作非空检验：

```
if( future.LastTick!=null )
```

注意：交易所代码参考下表：

交易所代码	交易所
SH	上交所
SZ	深交所
CFFEX	中金所
SHFE	上期所
DCE	大商所
CZCE	郑商所

➤ 动态订阅

在实盘模式下，可以用 `BookInstrument` 方法动态订阅品种。

4.2 时间

时间在策略中是极为重要的环境参数，策略需要准确地知道当前的交易日和时刻，以作出判断和操作。

➤ 交易日

`Year`、`Month`、`Day` 是当前交易日的年、月、日。可以基于年月日构造时间变量。

```
T0 = new DateTime(Year, Month, Day, 10, 0, 0);
```

➤ 当前时刻

实盘时，通过 `Now` 可以获得操作系统本地时间（注意不等同于交易所时间）。复盘时 `Now` 是最新 `Tick` 的时间。

➤ TickNow

在策略开发中，一个更为实用的时间是最新 `Tick` 的时间戳，我们称之为 `TickNow`：

```
DateTime TickNow;  
public override void OnTick(Tick tick)  
{  
    TickNow = tick.DateTime;  
}
```

注意：在策略中不能简单地使用 `.NET` 的系统函数 `DateTime.Now` 获取当前时刻，`DateTime.Now` 获取到的是操作系统的时间，这在复盘时将引起策略逻辑错误。

注意：由于实盘时 `Now` 是操作系统的时间，因此如果要使用 `Now`，则要经常同步计算机时间，使得计算机本地时间与北京时间一致。

如果策略需要在某一个固定时刻 **T0**（例如 **10:00**）以后执行某项操作，而且特别要求如果启动策略的时间晚于 **T0** 也需要执行一次。

注意：用 **Task** 并不能满足后面的要求，假如用户是在 **T0** 之后（例如 **10:10**）启动的策略，那么 **Task** 中的代码将不会得到执行。

为了满足这种需求，我们可以利用时间比较的方法实现固定时间后执行任务的功能：

```
using System;
using Ats.Core;
using Ats.Indicators;
namespace DemoStrategy
{
    public class 固定时间后执行任务 : Strategy
    {
        ///<summary>
        ///固定时间
        ///</summary>
        DateTime T0;

        ///<summary>
        ///是否已经执行标志
        ///</summary>
        bool IfHasDone = false;

        public override void Init()
        {
            //设置基准时间为当天的10:00
            T0 = new DateTime(Year, Month, Day, 10, 0, 0);
        }

        public override void OnTick(Tick tick)
        {
            //没有执行过操作并且当前时刻Now在T0之后
            if (!IfHasDone && Now > T0)
            {
                //执行操作
                //...
                IfHasDone = true;
            }
        }
    }
}
```

4.3 夜盘

中国的股票，股指期货，股票期权是没有夜盘的。但是商品期货是有夜盘的，夜盘的交易时间段如下表所示。各个品种的交易时段在MQ的Config里面的TradingFrame.xml文件中维护。

交易所	代码	品种	夜盘时段
上期所	Cu	铜	21:00~1:00
	Zn	锌	21:00~1:00
	Al	铝	21:00~1:00
	Pb	铅	21:00~1:00
	Ni	镍	21:00~1:00
	Sn	锡	21:00~1:00
	rb	螺纹钢	21:00~23:00
	hc	热轧卷	21:00~23:00
	Bu	沥青	21:00~23:00
	Au	黄金	21:00~2:30
	Ag	白银	21:00~2:30
	ru	橡胶	21:00~23:00
大商所	M	豆粕	21:00~23:30
	Y	豆油	21:00~23:30
	A	豆一	21:00~23:30
	B	豆二	21:00~23:30
	P	棕榈油	21:00~23:30
	J	焦炭	21:00~23:30
	Jm	焦煤	21:00~23:30
	i	铁矿石	21:00~23:30
郑商所	Sr	白糖	21:00~23:30
	Cf	棉花	21:00~23:30
	Pta	PTA	21:00~23:30
	Me	甲醇	21:00~23:30
	Rm	菜粕	21:00~23:30
	fg	玻璃	21:00~23:30
	Oi	菜籽油	21:00~23:30
	tc	动力煤	21:00~23:30

注意：三个商品交易所的夜盘在交易所的业务清算中所属的交易日都是下一个交易日，而不是当天的自然日。

4.4 行情

行情是策略实时接收到的最新信息，**Tick**、买卖盘、K 线都包含有实时信息。实盘时的行情由行情源服务器主动推送给 **MQ**，复盘时的行情由 **MQ** 读取历史 **Tick** 数据在本地生成。

➤ 最新价

最新价是获取行情最快捷的途径。策略启动以后，对于所有订阅的品种，一旦有 **Tick** 到来，其属性 **LastPrice** 将会自动更新。

```
public override void OnTick(Tick tick)
{
    Stock stock = AllStocks[0];
    if (stock != null)
    {
        if (stock.LastPrice > 0)
        {
            Print(stock.StockCode + "-" + stock.LastPrice.ToString());
        }
    }
}
```

通过 **LastFuturePrice ()** 可以直接访问默认期货品种 **DefaultFuture** 的最新价。这方便了单品种策略的开发。

注意：在 **Init** 时，由于策略处于初始化过程中，是不能更新实时行情的。因此如果在 **Init** 中运用上面的方法，最新价 **LastPrice** 将为 **0**。所以在使用价格时，最好先做检验确保其 **>0**。

➤ Tick

有两种方法可以获取最新的 **Tick**：一是通过 **Future** 和 **Stock** 的属性 **LastTick** 获得最新的 **Tick**。

注意，如果该品种当天停牌，则不会有 **Tick** 数据。因此在使用 **Tick** 时要注意先判断是否为空。对变量是否为空进行预先判断是编程的好习惯。

```
Future future = AllFutures[0];
if (future != null) //判断期货是否为空
{
    Tick tick = future.LastTick; //
    if (tick != null) //判断Tick是否为空
    {
        //显示最新价格
        Print("最新价格=" + tick.LastPrice);
    }
    else
    {
        Print("期货Tick为空");
    }
}
else
{
    Print("期货为空");
}
```

二是通过 **LastFutureTick()** 可以直接访问默认期货品种 **DefaultFuture** 的最新 **Tick**。

➤ 成交量和成交额

Tick 里的成交量和成交额指的是当天总的成交量和成交额。因此，如果要计算当前 Tick 的成交量或成交额，可以将当前 Tick 的成交量或成交额减去前一 Tick 的成交量或成交额。

```
///
```

➤ 买卖盘口

买卖盘口 (Quote) 体现了买卖盘口的细节信息，包含了当前委托队列的数量和价格。在交易过程中，通过对买卖盘的研判可以给交易提供一些短线趋势的信息。因此买卖盘信息在短线交易中是很重要的。

Level1 股票行情源提供了 5 档盘口数据，Level1 期货行情源提供了 1 档盘口数据。Level2 的股票行情源提供了 10 档盘口数据，Level2 期货行情源提供了 5 档盘口数据。MQ 提供 Level1 和 Level2 的行情接入适配器。

卖⑤	0.945	12277
卖④	0.944	3934
卖③	0.943	1325
卖②	0.942	7031
卖①	0.941	704
买①	0.940	2200
买②	0.939	47
买③	0.938	400
买④	0.937	907
买⑤	0.936	1501

通过调用最新 Tick 的属性 Quote，可以访问到该 Tick 的买卖盘信息。买盘的前缀是 Bid，而卖盘的前缀是 Ask。

注意：MQ 的盘口 Quote 中股票买卖量 Volume 的单位是股，期货和期权的单位是张。

下面这个例子就是通过 Tick 的 Quote 得到买卖盘的信息。

```
Stock stock = AllStocks[0];
if (stock != null)
```

```
{
    StockTicktick = stock.LastTick;//获取Tick
    if (tick != null)
    {
        Quote quote = tick.Quote;//获取
        if (quote != null)
        {
            string str = stock.StockCode;
            str += "卖1价,=" + quote.AskPrice1.ToString();
            str += "卖2价,=" + quote.AskPrice2.ToString();
            str += "卖3价,=" + quote.AskPrice3.ToString();
            str += "买1价,=" + quote.BidPrice1.ToString();
            str += "买2价,=" + quote.BidPrice2.ToString();
            str += "买3价,=" + quote.BidPrice3.ToString();
            str += "卖1量,=" + quote.AskVolume1.ToString();
            str += "卖2量,=" + quote.AskVolume2.ToString();
            str += "卖3量,=" + quote.AskVolume3.ToString();
            str += "买1量,=" + quote.BidVolume1.ToString();
            str += "买2量,=" + quote.BidVolume2.ToString();
            str += "买3量,=" + quote.BidVolume3.ToString();
            Print(str);
        }
    }
}
```

➤ 加权平均委买/卖价

股票的 **Level2** 行情提供了加权平均委买价 **WeightedAvgBidPrice** 和加权平均委卖价 **WeightedAvgAskPrice**, 该数据是最新的以盘口委托量为权重的平均委托买入/卖出价格。其计算公式是:

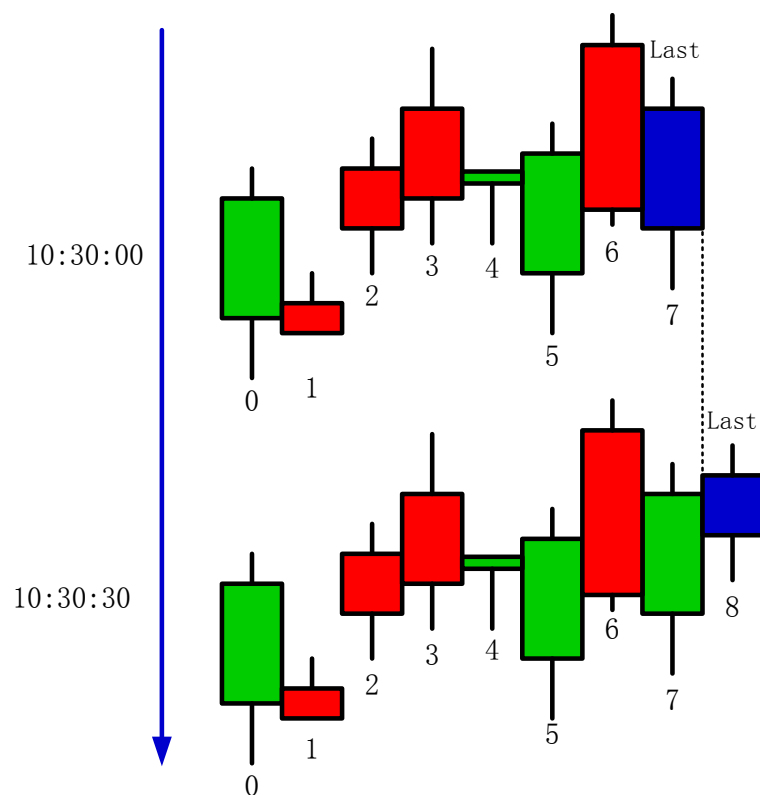
$$\text{WeightedAvgBidPrice} = \frac{\sum_{i=1}^N \text{BidVolume}_i * \text{BidPrice}_i}{\sum_{i=1}^N \text{BidVolume}_i}$$

$$\text{WeightedAvgAskPrice} = \frac{\sum_{i=1}^N \text{AskVolume}_i * \text{AskPrice}_i}{\sum_{i=1}^N \text{AskVolume}_i}$$

➤ K 线

K 线中的最新 K 线（标记为 **Last**）包含了实时行情的信息。当行情走完一个完整的 K 线周期，K 线中就会添加一个新 Bar，并且这个新 Bar 被设置为 **Last**。最新 Bar 的最高价、最低价、收盘价等信息会根据最新的 Tick 实时更新。注意最新 Bar 的收盘价就是最新价格。

下图中是一条 1 分钟 K 线 Bars，10:30:00 时，K 线由 8 根 Bar 构成，序号从 0 到 7，其最新 Bar 为 Bars[7]。一旦时间推移过 10:30:00（例如 10:30:30），K 线就会自动增加了一根新 Bar，序号为 8。因此 Bars.Last 就是 Bar 是 Bars[8]。



一般通过在 **Init** 时调用 **GetBarSeries** 方法来构造 K 线。

下面的代码演示了如何定义、构造、使用 K 线中实时行情。

```
BarSeries bars; // 定义K线
public override void Init()
{
    // Init时初始化K线，IF1203合约的1分钟K线，初始向前回溯100根K线
    bars = GetBarSeries(EnumMarket.期货, "IF1203", 1, EnumBarType.分钟, 100);
}
```

```
public override void OnTick(Tick tick)
{
    if (bars != null)
    {
        Bar bar = bars.Last();//最新的Bar
        if (bar != null)
        {
            string str = "";
            str += "开盘价,=" + bar.Open.ToString();
            str += "最高价,=" + bar.High.ToString();
            str += "最低价,=" + bar.Low.ToString();
            str += "收盘价,=" + bar.Close.ToString();
            str += "成交量,=" + bar.Volume.ToString();
            str += "成交额,=" + bar.Turnover.ToString();
            Print(str);
        }
    }
}
```

➤ 成交量和成交额

和 **Tick** 不一样的是, **Bar** 里的成交量和成交额指的是这根 **K** 线成交量和成交额。

➤ 跨周期调用

MQ 支持跨周期调用 **K** 线, 只要在构造 **K** 线的时候, 设置不同的周期即可。**MQ** 支持的 **K** 线包括秒线、分钟线、小时线、日线。

```
BarSeries bars1;//定义K线1
BarSeries bars2;//定义K线2
public override void Init()
{
    //构造IF1203合约的3分钟线, 初始回溯100根K线
    bars1 = GetBarSeries(EnumMarket.期货, "IF1203", 3, EnumBarType.分钟, 100);
    DateTime t0 = new DateTime(2010, 1, 1);
    //构造000735.SZ的日线, 前复权, 回溯到2010年1月?1日
    bars2 = GetBarSeries(EnumMarket.股票, "000735.SZ", 1,
        EnumBarType.日线, t0, EnumRestoration.前复权);
}
```


4.5 回溯

策略中经常用当前的数据和过去的数据进行比较计算，也就是**回溯 Ago**。例如：如果当天价格创 30 天内新高，则认为行情突破，执行做多。这里用到了过去 30 天的最高价，这就需要历史数据进行回溯。基于历史行情数据库和当天的实时行情，MQ 中提供了对 K 线和 Tick 的回溯。

注意：MQ 在回溯引擎中尽量规避未来函数，因此即使是在复盘时，回溯数据的时候只能获取历史数据，而不能获取未来数据。

➤ 回溯 K 线

对 K 线的回溯操作分为 2 步，第一步是回溯到指定的 Bar，第二步是根据 Bar 获取需要的数据。

```
if (bar != null)
{
    double vol = bar.Volume; //获取成交量
    double open = bar.Close; //获取收盘价
}
```

➤ 新高新低

策略有时需要调用过去 N 个 Bar 的最高值或者最低值。一种方法是利用循环的方法，遍历所有 Bar，然后把最高值或者最低值取出来。

```
//安全性判断, 确保bars不为空, 且个数大于0
if ( bars != null && bars.Count > 0)
{
    double min = bars[0].Low; //设置最低值初始值
    double max = bars[0].High; //设置最高值初始值
    //由于索引y=0的bar的数据已经作为初始值, 所以从索引开始循环
    for (int i = 1; i < bars.Count; i++)
    {
        //更新最低值
        min = Math.Min( min, bars[i].Low );
        //更新最高值
        max = Math.Max(max, bars[i].High);
    }
    Print("最低价="+min.ToString());
    Print("最高价="+ min.ToString());
}
```

另一种简洁的方法是直接调用 K 线的 **HighestHigh** 和 **LowestLow** 方法。

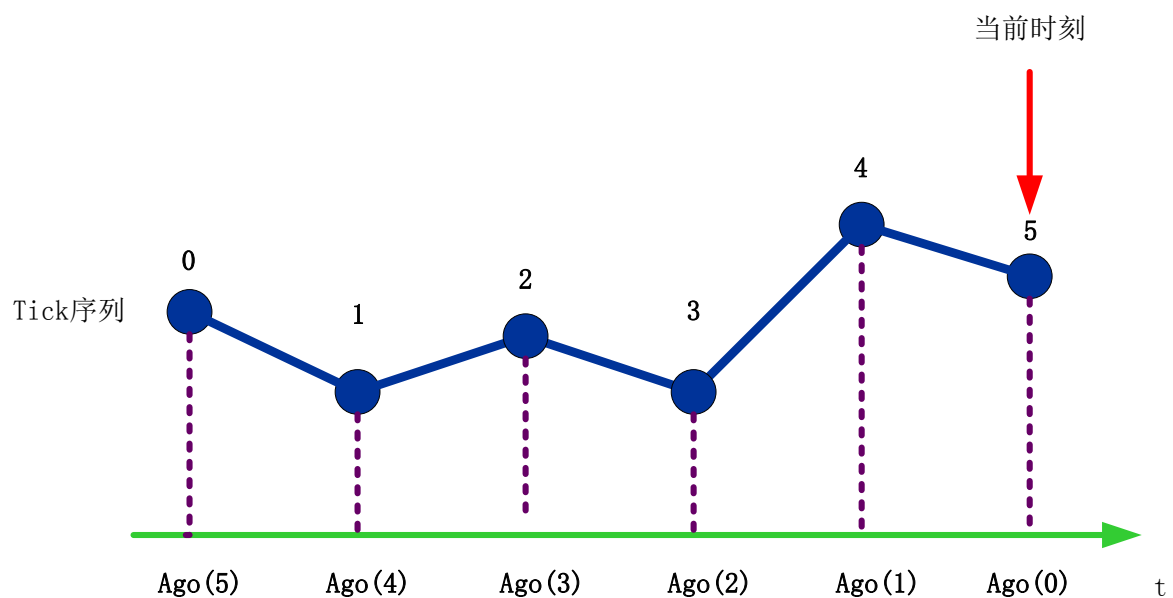
```
if (bars != null && bars.Count > 0)
{
    Print("最低价=" + bars.HighestHigh().ToString());
    Print("最高价=" + bars.LowestLow().ToString());
}
```

➤ 回溯特定 Tick

策略有时需要回溯更为精细颗粒的数据，即回溯 **Tick** 数据。

一种方法是通过 `AgoTick(Market,Code,N)` 直接回溯 **Tick**。该方法第一个参数是要回溯品种的代码，第二个参数是回溯个数 **N**，**N** 只能是正整数。注意：如果要访问最新的 **Tick**，则 **N=0**。

下图中用简写的 **Ago** 示意代表 `AgoTick`。



回溯 **Tick** 的范例代码如下。如果不给定品种代码，则会回溯默认期货品种或者默认股票品种的 **Tick**。

```
Tick f_tick = AgoTick(EnumMarket.期货, "IF1203", 100);
Tick s_tick = AgoTick(EnumMarket.股票, "000001.SH", 100);
```

注意：回溯 **Tick** 是不可以跨交易日的。也就是说如果回溯 **N** 足够大，即使本地数据完整，也不可以访问到上一个交易日的 **Tick** 数据。

MQ 底层在组装 **K** 线的时候会去读取本地的历史数据，因此回溯秒 **K** 线是可以跨日的。

➤ 回溯 Tick 时间序列

如果需要回溯当天启动策略以后的驱动品种的 Tick 序列, 可以使用 `GetTickBuffer()` 方法。具体使用方法是在 `Init` 时构造时间序列, 然后在 `OnTick` 方法内使用。

注意: MQ 中所有时间序列 (K 线、Tick 序列) 都必须在 `Init` 内部构造。如果在 `OnTick` 内构造, 则会出现严重的性能问题。

注意: `GetTickBuffer` 只维护驱动品种的策略启动后到最新的 Tick 时间序列。如果中途停止过程序, 则停止程序前那一部分 Tick 数据将无法通过 `GetTickBuffer` 获取。

```
using System;
using Ats.Core;
namespace DemoStrategy
{
    Public class 回溯期货Tick : Strategy
    {
        TickSeries tickLst;
        public override void Init()
        {
            //构造期货Tick时间序列
            tickLst = GetFutureTickBuffer();
        }
        public override void OnTick(Tick tick)
        {
            Tick agotick =AgoTick(EnumMarket.期货, 1);
            if (agotick != null)
            {
                Print("AgoTick=" + agotick.ToString());
            }
            agotick = tickLst.Last;
            if ( agotick != null)
            {
                Print("LastTick=" + agotick.ToString());
            }
        }
    }
}
```

建议用户自己在策略中维护 **Tick** 缓存，具体方法参考范例中的 **BufferManager**。

※范例

§ BufferManager §

※单元测试\基类\BufferManager.cs

```
using System;
using Ats.Core;
namespace MagicQuantTest
{
    /// <summary>
    /// 缓存管理器
    /// </summary>
    public class BufferManager
    {

        #region 变量
        /// <summary>
        /// 缓存Tick序列
        /// </summary>
        public TickSeries Buffer = new TickSeries();

        /// <summary>
        /// 跳
        /// </summary>
        public double PriceTick = 0.2;

        /// <summary>
        /// 数据是否合法
        /// 数据跨越秒数超过MaxT时才认为合法
        /// </summary>
        public bool IsValid = false;

        /// <summary>
        /// 容量
        /// 最大Tick数
        /// </summary>
        public int Capacity = 60;

        public int Count
        {
            get
            {
                return Buffer.Count;
            }
        }

        #endregion

        /// <summary>
        /// 最后一个Tick
        /// </summary>
        public Tick LastTick
        {

```

```
        get
        {
            return Buffer.Last;
        }
    }

    /// <summary>
    /// 构造函数
    /// </summary>
    /// <param name="_Capacity">容量</param>
    /// <param name="_PriceTick">跳</param>
    public BufferManager(int _Capacity, double _PriceTick)
    {
        Capacity = _Capacity;
        PriceTick = _PriceTick;
        Buffer = new TickSeries();
    }

    /// <summary>
    /// 添加Tick
    /// </summary>
    /// <param name="tick"></param>
    public void Add(Tick tick)
    {
        //将Tick添加到缓存中
        Buffer.Add(tick);

        //控制缓存内存占用
        while (Buffer.Count > Capacity)
        {
            Buffer.RemoveAt(0);
            IsValid = true;
        }
    }

    /// <summary>
    /// 回溯第N个Tick
    /// </summary>
    /// <param name="N"></param>
    /// <returns></returns>
    public Tick AgoTick(int N)
    {
        return Buffer.Ago(N);
    }

    /// <summary>
    /// 释放内存
    /// </summary>
    public void Dispose()
    {
        Buffer.Clear();
    }
}
```

➤ 回溯价格时间序列

我们还可以通过回溯价格时间序列 `GetPriceBuffer()` 的方法获取驱动品种的价格时间序列。价格时间序列里只包含价格信息。

注意：`GetPriceBuffer` 只维护驱动品种的策略启动后到最新的 `Tick` 时间序列。如果中途停止过程序，则停止程序前那一部分驱动品种的 `Tick` 数据将无法通过 `GetPriceBuffer` 获取。

```
using System;
using Ats.Core;
using Ats.Indicators;
namespace DemoStrategy
{
    public class 回溯价格时间序列: Strategy
    {
        ///<summary>
        ///价格时间序列
        ///</summary>
        DataSet dataLst;

        public override void Init()
        {
            //构造驱动品种的价格时间序列
            dataLst = GetPriceBuffer();
        }

        public override void OnTick(Tick tick)
        {
            if (dataLst.Count > 0)
            {
                Print( "LastPrice=" +dataLst.Last.ToString());
            }
        }
    }
}
```

4.6 账户

➤ 逻辑账户

MQ 的策略中使用逻辑账户作为交易的账户,用户在写策略的时候就不用去考虑填写账户 **ID** 了。逻辑账户是下单和回报中的账户,对策略来说非常重要。

在单账户的策略中可以简单的使用 **DefaultAccount** 做交易。**DefaultAccount** 是默认的逻辑账户。

在多账户策略中,可以先用 **IdxToLogAccount(i)**方法获取第 *i* 个逻辑账户 **ID**,然后就使用逻辑账户下单。在下面的例子中,策略首先获取账户个数,然后遍历所有逻辑账户,获取所有逻辑账户的账号。

```
int N = GetAccountCount(); // 获取策略配置的账户个数
for (int i = 0; i < N; i++)
{
    // 第1个逻辑账户的账号
    string AccountID = IdxToLogAccount(i);
    Print("第" + i + "个逻辑账户的账号是:" + AccountID);
}
```


➤ 物理账户

物理账户信息包括：账号，可用金额、总资产等，这些物理账户的信息可以通过 `GetAccount(Market)` 方法查询。

账户类型是 `Account`，主要包含下列信息：

属性	含义	说明
<code>ID</code>	期货帐号	
<code>BrokeID</code>	经纪公司代码	
<code>Balance</code>	动态权益	动态权益=期末结存+质押
<code>PreBalance</code>	静态权益	静态权益=昨日结存
<code>Avaiable</code>	可用资金	
<code>PositionMargin</code>	持仓保证金	持仓占用的保证金
<code>FrozenMargin</code>	冻结的保证金	委托未成交冻结的保证金
<code>PositionProfit</code>	浮动盈亏	浮动盈亏=持仓盈亏
<code>Commision</code>	手续费	当天累积的手续费
<code>FrozenCommision</code>	冻结手续费	
<code>CloseProfit</code>	平仓盈亏	
<code>Risk</code>	风险度	风险=持仓/权益

下面的例子中，调用 `GetAccount(Market)` 方法查询期货账户信息。

```
Account acc = GetAccount( MyMarket );
if (acc != null)
{
    Print("账户=" + acc.ToString());
    if ( acc.Balance <= 0)
    {
        Print("错误账户动态权益=" + acc.Balance.ToString());
    }
}
```

```
        Print("错误账户可用资金=" + acc.Available.ToString());
    }
    else
    {
        Print("账户动态权益=" + acc.Balance.ToString());
        Print("账户可用资金=" + acc.Available.ToString());
        Print("账户冻结保证金=" + acc.FrozenMargin.ToString());
    }
}
```

注意：调用 `GetAccount` 实际会去柜台系统查询账户信息，属于比较耗时的操作，应该尽量避免频繁调用。

➤ 调整现金

某些对冲策略会同时操作 1 个股票账户和 1 个期货账户。出于资产管理和风险控制的需要，资金会在两个账户之间动态分配。MQ 提供了修改现金的方法 `ModifyMoney(Market,x)`。如果现金调整量 >0 ，相当于增加资金，如果现金调整量 <0 ，相当于减少资金。

实盘时，出金入金的操作需要人工通过银期转账和银证转账实现。复盘时，可以在策略中调用调整现金的方法。为了保证策略绩效的真实性，修改现金时，一个账户增加的现金，应与另一个账户减少的现金金额相等。

```
double x = 10 * 10000; //转账金额=10万
ModifyMoney(EnumMarket.期货, -x); //期货账户减少10万现金
ModifyMoney(EnumMarket.股票, x); //股票账户增加10万现金
```

注意：目前 MQ 策略中不支持银期转账和银证转账，因此调整现金方法 `ModifyMoney` 只有在复盘时使用才有效。实盘时调用该方法，不会有任何效果。

4.7 查询机理

MQ 支持两种查询机制，一种是内存查询（通常用这种），另一种是柜台查询。查询的对象是委托、成交、持仓、可撤单列表。

内存查询是以 `Get` 作为前缀的方法（查询账户 `GetAccount` 除外），可以快速查询策略自己的对象。策略中使用内存查询时，在策略本地内存中查询目标对象，因此速度很快。例如 `GetCanCancelOrders` 查询当前策略自己的所有可撤单列表，而不能查到同一个账户其它策略或者手工提交的委托。

柜台查询是以 `Query` 作为前缀的方法，可以查询账户级别的对象。由于需要去柜台系统查询，因此其查询速度比内存查询慢。例如 `QueryCanCancelOrders` 查询账户所有的可撤单列表。

注意：开发策略时，不能过于频繁的调用查询柜台的方法，否则会影响策略执行效率，甚至导致部分 `Tick` 不参与计算。如果一定需要调用，建议采用间隔一段时间或者若干个 `Tick` 才调用一次柜台查询类的方法。

下表中对比列出了一些常用的查询命令：

内存查询命令	功能	柜台查询命令	功能
GetOrder	根据委托 GUID 从内存查询委托	QueryOrder	根据委托 GUID 从柜台查询委托
GetOrderByTrade	根据成交查询内存中对应的委托		
GetOrders	查询策略中指定逻辑账户所有的委托	QueryOrders	从柜台查询指定逻辑账户的当日所有委托
GetCanCancelOrders	查询策略中指定逻辑账户所有的可撤委托	QueryCanCancelOrders	从柜台查询指定逻辑账户的当日所有可撤委托
GetPosition	查询策略的持仓 (首次调用时查账户的持仓)	QueryPosition	从柜台查询指定逻辑账户的所有持仓
GetTrades	查询策略的所有成交	QueryTrades	从柜台查询指定逻辑账户的当日所有成交
		GetAccount	从柜台查询账户信息

4.8 本地 GUID

策略下单后返回的 **Order** 的委托系统编号 **OrderSysID** 是由交易所分配的，因此在策略发出委托这一瞬间，其 **OrderSysID** 为空。**MQ** 对委托 **Order** 和成交 **Trade** 增加了本地 **GUID** 字段。本地 **GUID** 是策略下单时立即能获得的全局唯一 **ID**。这个 **GUID** 也是策略用来查询或者撤单操作的一个编号。

GUID 字段	含义
Order.GUID	委托的本地 GUID
Trade.GUID	成交的本地 GUID
Trade.OrderGUID	成交对应的委托的 GUID

注意：如果策略或者 **MQ** 重启，则维护在内存中的本地 **GUID** 体系就被破坏了。因此用 **QueryOrders** 查询出来的委托，如果不是当前策略发出的委托，就没有本地 **GUID**。

4.9 委托

➤ 查询委托

从策略内存中查询委托，只会查出策略自己生命周期内的委托，主要有下面几个方法：

一是通过委托回报，直接获取刚刚发生委托状态变化的委托实例。

```
public override void OnOrder(Order order)
{
    //委托状态变化时执行
    Print("委托回报" + order.ToString());
}
```

二是调用 `GetOrders(Market, LogAccount)` 方法，查询工程自己的全部委托列表。

```
OrderSeries ordLst = GetOrders(EnumMarket.期货, DefaultAccount);
```

三是在下单时，就把委托保存作为策略的一个变量，例如：

```
Order MyOrder; //定义变量
public override void Init()
{
    Future future = AllFutures["IF1408"];
    string InsID = future.ID;
    string ExchangeID = future.ExchangeID;
    //把委托保存在变量里
    MyOrder = SendOrder(DefaultAccount, InsID,
        EnumMarket.期货, ExchangeID, future.LastPrice,
        1, EnumBuySell.买入, EnumOpenClose.开仓);
}
```

四是调用 `GetOrder (OrdGUID)` 方法，根据委托的 `GUID` 来查询委托。

从柜台查询委托的方法有两种：

一是通过方法 `QueryOrder (GUID)`，从柜台根据委托的 `GUID` 查询委托。

二是通过方法 `QueryOrders(Market,LogAccount)`，从柜台查询到指定账户当日的所有委托，包括手工下单或者别的策略提交的委托。

➤ 可撤委托（挂单）

从内存查询可撤委托是通过调用 `GetCanCancelOrders(Market,LogAccount)` 方法，可以获取

当前策略可以撤单的全部委托。

```
OrderSeries ordLst = GetCanCancelOrders(EnumMarket.期货); //查询期货可撤单列表
```

注意：由于MQ的事件隔离机制，通过GetCanCancelOrders不能查到手工下单或者别的策略提交的委托。

从柜台查询可撤委托是通过调用QueryCanCancelOrders(Market,LogAccount)方法，可以获取指定账户的全部委托，包括手工下单或者别的策略提交的委托。

4.10 成交

➤ 成交

属性	含义
TradeID	成交 ID
OrderGUID	对应的委托的 GUID
InstrumentID	品种代码
ExchangeID	交易所 ID
BrokerID	经纪公司代码
InvestorID	投资者代码
Direction	买卖方向，枚举 EnumBuySell 类型
OpenOrClose	开平标识，枚举 EnumOpenClose 类型
Volume	成交量
TradeBalance	成交金额
Price	成交价格
DateTime	成交时间
FundUnit	资金单元
BasketID	篮子 ID（如果不是篮子委托，则篮子 ID 为空）

通过 [GetTrades](#) 可以从内存中查询成交。

从内存查询成交是通过调用 [GetTrades \(Market,LogAccount\)](#)方法，获取策略的全部指定市场、指定逻辑账户的成交列表，例如：

```
TradeSeries期货成交列表= GetTrades(EnumMarket.期货, DefaultAccount);  
TradeSeries股票成交列表= GetTrades(EnumMarket.股票, DefaultAccount);
```

从柜台查询成交是通过 [QueryTrades\(Market,LogAccount\)](#)方法，获得账户的指定市场、指定逻辑账户的成交列表，例如：

```
TradeSeries期货成交列表 = QueryTrades(EnumMarket.期货, DefaultAccount);  
TradeSeries股票成交列表 = QueryTrades(EnumMarket.股票, DefaultAccount);
```

另一种方法是通过成交回报事件，获取事件回报中传入的成交的实例。

```
///<summary>  
///成交回报事件  
///</summary>  
///<param name="trade">成交</param>  
public override void OnTrade(Trade trade)  
{  
    Print("收到成交回报事件:"+trade.ToString() );  
}
```

4.11 持仓

持仓又称为头寸 **Position**。

➤ 持仓 **Position**

属性	含义
InstrumentID	品种代码
BrokerID	经纪公司代码
PosiDirection	多空方向，枚举 EnumPositionDirection 类型
HedgeFlag	套保标识，枚举 EnumHedgeFlag 类型
Volume	总持仓量
YdPosition	昨持仓量
TodayPosition	今持仓量
EnableVolume	可平仓量，可卖出量
Price	最新价格
PositionCost	持仓合约价值
UseMargin	占用保证金
PositionProfit	持仓盈亏（浮动盈亏）
CloseProfit	平仓盈亏

➤ 持仓数量变化逻辑

以 **T+1** 的股票交易为例，这里重点介绍一下持仓的几个重要数量的变化逻辑和数量关系。

TodayPosition: 今仓，当日买入的股票都是今仓，当日买入的股票，当日不可以卖出。

YdPosition: 昨仓，上一个交易日或者更早的交易日买入的持仓，当日卖出股票的指令，都会且只会卖出昨仓部分的持仓。当卖出股票时，昨仓会减少。

Volume: 持仓总量，账户当前的持仓总数量，包含昨仓+今仓。

EnableVolume: 可卖出量，等于昨仓扣减掉卖出委托的冻结量。

下面两个恒等式始终成立：

$$\text{Volume} = \text{TodayPosition} + \text{YdPosition}$$

$$\text{YdPosition} = \text{EnableVolume} + \text{卖出冻结}$$

另外，如果用户发出一个卖出指令，其数量超过 **EnableVolume**，则柜台系统会拒绝该委托，拒绝原因是：可卖数量不足。

下面是一个具体的业务场景：

开盘时，账户持有 **1000** 股 **600028.SH**，持仓的初始状态如下表所示。

状态	Volume	YdVolume	TodayVolume	卖出冻结	EnableVolume
初始状态	1000	1000	0	0	1000

10:00 用户以涨停价报单，买入 **100** 股，并且全部成交，此时持仓状态变为。

状态	Volume	YdVolume	TodayVolume	卖出冻结	EnableVolume
初始状态	1000	1000	0	0	1000
买入 100 股 全部成交	1100	1000	100	0	1000

10:01 用户以卖 1 价报单，卖出 600 股，发生了部分成交，成交了 400 股，还有 200 股挂在盘口上。

状态	Volume	YdVolume	TodayVolume	卖出冻结	EnableVolume
初始状态	1000	1000	0	0	1000
买入 100 股， 全部成交	1100	1000	100	0	1000
卖出 600 股， 成交 400 股， 剩余 200 股	700	600	100	200	400

10:02，用户撤掉那剩余的 200 股挂单，且撤单成功，冻结的 200 股卖出量被释放掉，可卖量从 400 股增加到 600 股。

状态	Volume	YdVolume	TodayVolume	卖出冻结	EnableVolume
初始状态	1000	1000	0	0	1000
买入 100 股 全部成交	1100	1000	100	0	1000
卖出 600 股 成交 400 股 剩余 200 股	700	600	100	200	400
撤掉 200 股挂单	700	600	100	0	600

10:03, 用户再次以跌停价挂出 200 股卖单, 这次卖单全部成交。

状态	Volume	YdVolume	TodayVolume	卖出冻结	EnableVolume
初始状态	1000	1000	0	0	1000
买入 100 股 全部成交	1100	1000	100	0	1000
卖出 600 股 成交 400 股 剩余 200 股	700	600	100	200	400
撤掉 200 股挂单	700	600	100	0	600
卖出 200 股 全部成交	500	400	100	0	400

10:04, 用户最后卖出 500 股, 由于可卖不足, 委托被柜台系统拒绝。

状态	Volume	YdVolume	TodayVolume	卖出冻结	EnableVolume
初始状态	1000	1000	0	0	1000
买入 100 股 全部成交	1100	1000	100	0	1000
卖出 600 股 成交 400 股 剩余 200 股	700	600	100	200	400
撤掉 200 股挂单	700	600	100	0	600
卖出 200 股 全部成交	500	400	100	0	400
卖出 500 股 委托拒绝	500	400	100	0	400

➤ 查询持仓

从内存查询持仓可以通过调用 `GetPosition(Market,LogAccount)` 方法获取策略的所有持仓。

从柜台查询持仓可以通过调用 `QueryPosition(Market,LogAccount)` 方法获取账户的所有持仓。

注意：策略需要订阅品种的价格，`GetPosition` 才能查询到该品种的持仓。

➤ 融资融券的持仓

股票信用账户的持仓中，如果是融资买入的股票，其 `PosiDirection` 是多头；如果是融券卖出的股票，其 `PosiDirection` 是空头。

特别的，如果一个信用账户中，对于同一个股票，既有融资买入的持仓，也有融券卖出的持仓，那么查询持仓时，会返回两行 `Position`，一行是多头头寸，一行是空头头寸。

4.12 交易

➤ 交易方法

交易是策略的核心执行组件，是策略算法计算推导得出的终极结果。交易主要实现的功能就是下单和撤单。

方法	功能说明
SendOrder	统一的下单方法，支持全部市场，单位是股或者张
LimitOrder	期货限价委托，单位是张
BuyStock	限价买入股票，单位是股
SellStock	限价卖出股票，单位是股
PurchaseETF	申购基金，单位是股
RedeemETF	赎回基金，单位是股
CancelOrder	撤策略自己的单个委托，不能撤其他策略的委托
ACancelOrder	撤逻辑账户的委托，可以不是策略自己的，注意和 CancelOrder 区分开

➤ 统一下单

下单都是通过统一的 [SendOrder](#) 方法提交委托，该方法返回 [Order](#) 的实例。[SendOrder](#) 可以提交期货、股票、股票期权和期货期权的委托。[SendOrder](#) 中有些参数配置有默认值，如果不输入这些参数，则会用默认值下单。

```
public Order SendOrder(  
    string LogAccount,  
    string InsID,  
    EnumMarket InsType,  
    string ExchID,  
    double price,  
    int Volume,  
    EnumBuySell Direction,  
    ...  
)
```

```
EnumOpenClose OpenClose = EnumOpenClose.开仓,
EnumOrderPriceType PriceType = EnumOrderPriceType.限价,
EnumOrderTimeForce TimeForce = EnumOrderTimeForce.当日有效,
EnumHedgeFlag HedgeFlag = EnumHedgeFlag.投机);
```

参数	含义	默认值	类型
LogAccount	逻辑账户		string
InsID	品种代码		Int
InsType	市场类型（股票、期货、股票期权、期货期权）		EnumMarket
ExchID	交易所 ID		string
Price	限价		Double
Volume	下单量（期货：张，股票：股）		Int
Direction	买卖方向		EnumBuySell
OpenClose	开平标志（针对期货和期权）	开仓	EnumOpenClose
PriceType	价格类型	限价	EnumOrderPriceType
TimeForce	时间类型	当日有效	EnumOrderTimeForce
HedgeFlag	套期保值标志（针对期货）	投机	EnumHedgeFlag

参数说明：

LogAccount：下单的对象是逻辑账户，而不是物理账户。可以通过**IdxToLogAccount(i)**方法获取策略的第*i*个逻辑账户ID。一个逻辑账户有其对应的股票、期货、股票期权和期货期权账户。因此策略下单时不用考虑物理通道，而只要输入一个统一的逻辑账户即可。

InsID：下单的品种代码。注意股票的品种代码需要带上后缀，上交所的后缀是.SH，深交所品种的代码后缀是.SZ。

InsType:委托的市场，枚举 **EnumMarket** 类型目前支持四种市场：股票、期货、股票期权和期货期权。

ExchID: 交易所 ID。

交易所 ID 参考下表：

交易所代码	交易所
SH	上交所
SZ	深交所
CFFEX	中金所
SHFE	上期所
DCE	大商所
CZCE	郑商所

策略中可以根据品种的属性直接获取交易所 ID，如下面的代码：

```
Stock stock = AllStocks[0]; // 订阅的第一个股票
string ExchID1 = stock.ExchangeID;

Future future = AllFutures[0]; // 订阅的第一个期货
string ExchID2 = future.ExchangeID;

Option optionS = AllStockOptions[0]; // 订阅的第一个股票期权
string ExchID3 = optionS.ExchangeID;

Option optionF = AllFutureOptions[0]; // 订阅的第一个期货期权
string ExchID4 = optionF.ExchangeID;
```

➤ 期货限价委托

期货限价委托方法 `LimitOrder` 是以限价方式提交期货委托。

最基础的限价委托函数方法是：

`LimitOrder(FutureCode, Qty, LimitedPrice, Direction, OpenorClose)`

其它的期货限价委托方法都是由该方法衍生出来的。该方法返回委托 `Order` 实例。

参数	含义	类型
<code>FutureCode</code>	期货代码	<code>String</code>
<code>Volume</code>	委托手数	<code>Int</code>
<code>LimitedPrice</code>	委托限价	<code>Double</code>
<code>Direction</code>	买卖方向	枚举 <code>EnumBuySell</code>
<code>OpenorClose</code>	开平标志	枚举 <code>EnumOpenClose</code>

注意：委托手数 `Volume` 必须是大于 0 的正整数。例如：

```
LimitOrder("IF1205", 3, 2360, EnumBuySell.买入, EnumOpenClose.开仓);
```

开平标志：如果平仓时，没有该方向的头寸，或者开仓时保证金不足，则会成为废单。

注意：期货的委托价格必须是期货合约最小价格变动单位（称之为“跳”）的整数倍。期货 `Future` 的 `TickPrice` 属性就是最小价格变动单位（跳）。例如在下面的策略代码中，我们想开仓买入合约，为了确保成交，需要以当前价格的基础上加 2 跳的价格下单。

```
string IFCode= "IF1203";
Future future = AllFutures[IFCode];
if (future != null)
{
    //委托价格=最新价格+2跳
    double OrderPrice = future.LastPrice+2*future.PriceTick;
    //发出限价单
    LimitOrder(IFCode, 1, OrderPrice,
                EnumBuySell.买入, EnumOpenClose.开仓);
}
```

买卖方向和开平标志的实际操作含义参见下表：

买卖	开平	多空	实际操作	范例
买入	开仓	做多	新开多头头寸	LimitOrder("IF1203", 3, 2360, EnumBuySell. 买入, EnumOpenClose. 开仓);
买入	平仓	做多	平掉空头头寸	LimitOrder ("IF1203", 3, 2360, EnumBuySell. 买入, EnumOpenClose. 平仓);
卖出	开仓	做空	新开空头头寸	LimitOrder ("IF1203", 3, 2360, EnumBuySell. 卖出, EnumOpenClose. 开仓);
卖出	平仓	做空	平掉多头头寸	LimitOrder ("IF1203", 3, 2360, EnumBuySell. 卖出, EnumOpenClose. 平仓);

如果是交易默认期货品种，即订阅的第一个期货品种，限价委托的输入参数可以省略期货代码。

LimitOrder(Volume, LimitedPrice, Direction, OpenorClose)

其效果等价于：

LimitOrder(默认期货代码, Qty, LimitedPrice, Direction, OpenorClose)

注意：中金所不区别平仓和平今，因此如果需要平股指期货的持仓，开平标志使用平今和平仓都可以，柜台系统会自动进行转换。上期所区别平仓和平今，因此如果需要平上期所合约的持仓，对于昨仓，需要使用平仓，对于今仓，需要使用平今。

➤ 撤单

撤单是将未成交的委托撤掉。撤单的写法是：`CancelOrder(order)`，`order` 是一个可撤单的委托。

例如，策略需要实现一个定时批量撤单的功能，将所有已经提交委托 5 秒但是未成交的单子立刻撤掉。

```
OrderSeries lstOrder = GetCanCancelOrders(EnumMarket.期货); // 查询所有可撤单列表
foreach (Order order in lstOrder) // 遍历可撤单
{
    TimeSpan ts = Now - order.InsertTime; // 计算该委托至今的TimeSpan
    double 时间差 = ts.TotalSeconds; // 转换成时间差，单位是秒
    if (时间差 > 5) // 如果时间差 > 5秒
    {
        CancelOrder(order); // 撤单
    }
}
```

注意：根据国内期货交易所目前的规定，账户单个合约一天之内最多只能撤单 500 次（以交易所最新公告为准），但套利账户不受此限制。

➤ 买卖股票

买卖股票的方法分别是 **BuyStock** 和 **SellStock**, 其底层最终会调用万能下单方法 **SendOrder**。

买入股票 **BuyStock** 和卖出股票 **SellStock** 就是限价委托买入和卖出股票, 需要输入股票代码、委托股数、委托限价, 具体写法分别是:

```
//买入300股浦发银行
BuyStock("600000.SH", 300, 9.5);
//卖出200股深发展
SellStock("000001.SZ", 200, 17.2);
```

注意: 买入股票的委托股数只能是 **100** 的整数倍, 卖出股票的委托股数可以不是 **100** 的整数倍。如果买入股票时, 账户金额不足, 或者卖出股票时, 股票持仓量不足, 均会产生废单。

➤ ETF 申购和赎回

PurchaseETF: 申购是用股票账户中的一篮子股票向基金公司换 **ETF** 基金份额。

RedeemETF: 赎回是用股票账户中的 **ETF** 基金份额向基金公司换回一篮子股票。

申购赎回指令只需要输入 **ETF** 基金代码和交易单位: 份就可以了。

注意: 申购赎回的最小交易单位一般是若干万股, 而不是 **100** 股。

➤ 融资融券

融资买入是向券商借钱买入股票, 使用下单 **SendOrder**, 买卖方向为 **EnumBuySell.融资买入**。

融券卖出是向券商借入股票在二级市场卖出, 使用万能下单 **SendOrder**, 买卖方向为 **EnumBuySell.融券卖出**。

注意: 融资融券需要券商有可融额度, 如果额度不足, 则会拒绝融资融券委托。融券卖出的委托限价不能低于该股票的最近成交价。

4.13 交互

MQ 提供了多重策略与用户交互的方式。

➤ 显示日志

文本信息可以通过 `Print(str)` 方法显示在策略监控屏幕上。

```
Print("当前时刻="+Now.ToString());
```

➤ 报警声音

`PlaySound` 方法可以用声音的方式提醒关注策略的运行状态。例如当检测到期货账户风险超过 **80%** 发出警告，当策略开始和退出的时候发出相应的提示音。

报警播放的声音来源于 MQ 安装根目录下 **wav** 文件夹下的声音文件。

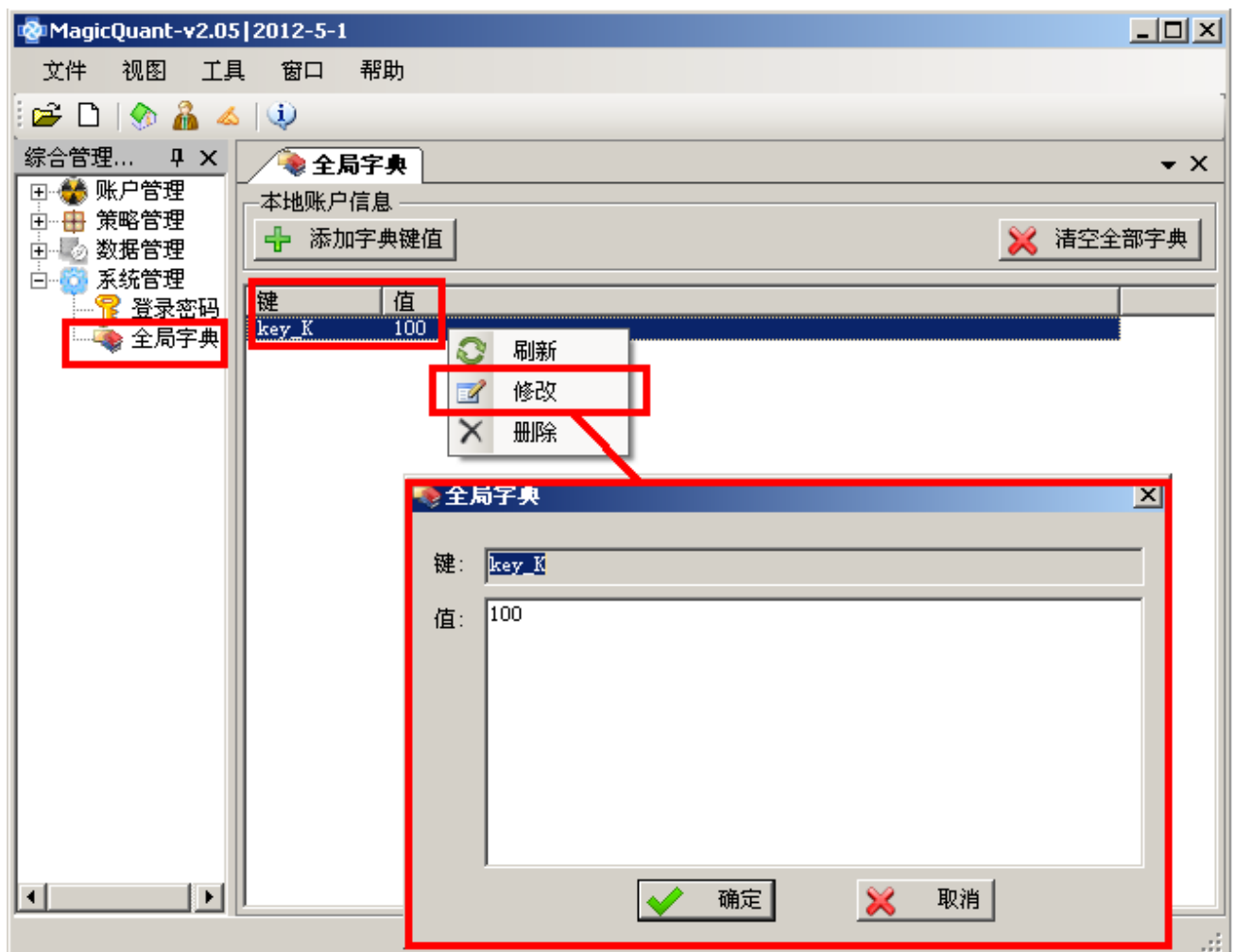
```
using System;
using Ats.Core;
namespace MyNameSpace
{
    public class 声音报警范例 : Strategy
    {
        int clc = 0;
        public override void Init()
        {
            PlaySound(EnumSound.ding); //提示策略开始
        }
        public override void OnTick(Tick tick)
        {
            clc++;
            if (clc > 100)
            {
                Account MyFutureAccount = GetAccount(EnumMarket.期货); //查询期货账户
                if (MyFutureAccount.Risk > 0.8) //账户风险过高
                {
                    PlaySound(EnumSound.alarm); //声音报警
                }
                clc = 0; //计数器归零
            }
        }
        public override void Exit()
        {
            PlaySound(EnumSound.logout); //提示策略退出
        }
    }
}
```

4.14 全局字典

全局字典 **Dict** 是 **MQ** 提供的轻量级文件型数据存取功能，适用于策略存取简单类型的变量。类似于 **.NET** 的 **Dictionary** 类型，全局字典是一系列由键-值（**Key-Value**）构成的数据。全局字典保存在计算机硬盘的物理文件上，因此即使程序异常中断或者跨交易日，只要硬盘没有损坏，其保存的数据就不会丢失。通过 **MQ** 的综合管理面板→系统管理→全局字典，可以直接添加、删除、编辑全局字典中的内容。

注意：MQ 的 **Dict** 是存储在物理硬盘的文件中的；**.NET** 的 **Dictionary** 是存储在内存中的。

注意：MQ 的全局字典的键和值都是字符串形式，且键不允许重复。



策略通过 3 个方法访问 MQ 的全局字典：

DictContains(key)

功能：判断全局字典中是否存在键 **key**，如果存在返回 **true**，不存在则返回 **false**。

GetDict(key)

功能：根据键 **key** 获取对应的值 **value**，如果字典中不存在键 **key**，则返回空字符串。

SetDict(key, value)

功能：根据键 **key** 设置对应的值为 **value**，如果字典中不存在键 **key**，则新建一个字典项，其键为 **key**，其值为 **value**；如果字典中存在键 **key**，则将其对应的值修改为 **value**。

由于全局字典的值只能保存字符串，因此将全局字典的值取出来以后，还要根据其类型，利用 **Parse** 方法做相应的解析转换。在下面的例子中整型变量 **x** 和 **double** 型变量 **y** 的值分别保存在全局字典中，其对应的键分别为 **key_x** 和 **key_y**。

```
string str_x =GetDict( "key_x ");
int x = int.Parse( str_x);//解析为int

string str_y = GetDict("key_y ");
double y = double.Parse(str_x); //解析为double
```

在解析时，为了避免解析出错导致程序异常，还可以使用 **Tryparse** 方法。使用 **TryParse** 方法时，即使解析的字符串和目标类型不匹配也不会抛出异常。用户可以根据需要选择是用 **Parse** 还是 **TryParse**。

```
string str_x =GetDict( "key_x ");
int x = 0;
int.TryParse(str_x, out x);//解析为int

string str_y = GetDict("key_y ");
double y = 0;
double.TryParse(str_y, out y);//解析为double
```

注意：策略运行时如果删除正在使用的字典值，策略再次读取该字典时将由于找不到键而无法获得正确的值。

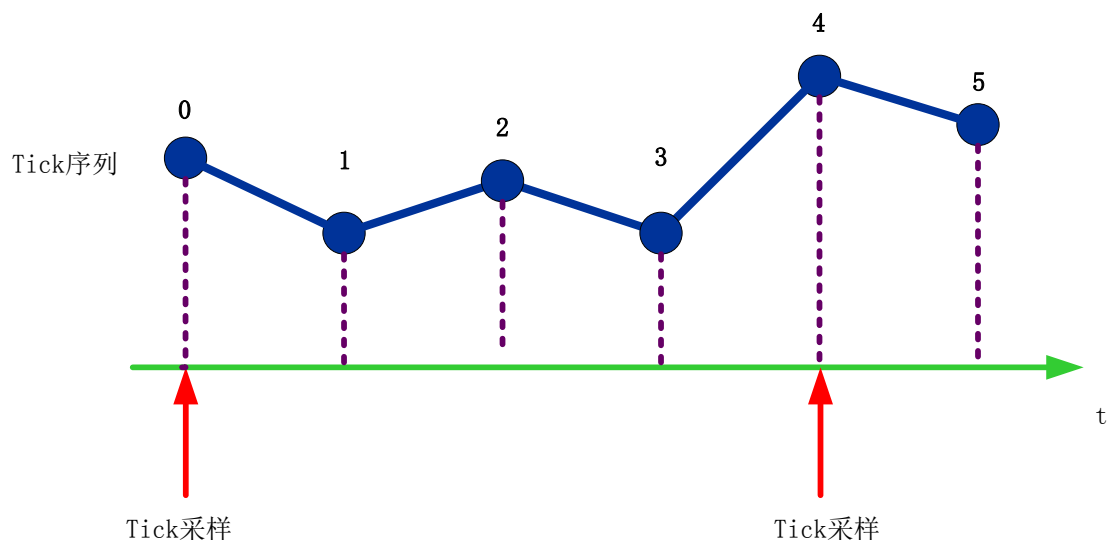
4.15 采样

有时候策略需要间隔一定周期才执行某些特定操作，这里需要应用采样的功能。MQ 中可以很简洁的实现 Tick 采样和时间采样。

➤ Tick 采样

当策略是由 Tick 数据驱动时，我们可以使用 Tick 采样。Tick 采样的思路是利用计数器来实现每隔一段 Tick 才执行特定操作。两次 Tick 采样之间间隔的 Tick 个数是固定的。

例如查询期货账户比较耗时，我们不能每个 Tick 都去查询账户，但又希望尽可能获得准确的账户数据，例如可用资金、风险等信息。这个时候就可以用 Tick 采样的方法，每隔一定数量的 Tick 执行一次 `GetAccount` 的方法(而不是每个 Tick 都执行)，从而提高策略运行效率，降低柜台系统的压力。如下图所示，两次 Tick 采样之间间隔固定的 Tick 数。



利用 Tick 采样的方法实现低频率查询账户的代码如下：

```
using System;
using Ats.Core;
using Ats.Indicators;
namespace DemoStrategy
{
    public class Tick采样查询账户 : Strategy
    {
        int clc = 0; //计数器

        public override void OnTick(Tick tick)
```

```
{
    clc++;
    if (clc > 100)
    {
        //每隔100个Tick（相当于每50秒）查询一次期货账户
        Account MyFutureAccount = GetAccount(EnumMarket.期货);
        clc = 0;
    }
}
}
```

4.16 数学

MQ 的 **MathHelper** 类中提供了一些简单的数学方法，方便策略对时间序列数据进行加工计算。**MathHelper** 目前提供的方法都是静态方法，包括：计算时间序列的最大值、最小值、平均值、标准差等。

```
using System;
using Ats.Core;
using Ats.Indicators;
namespace DemoStrategy
{
    public class 使用Mathhelper : Strategy
    {
        DataSeries PriceLst; //定义价格时间序列

        public override void Init()
        {
            //回溯获取驱动品种的价格时间序列
            PriceLst = GetPriceBuffer();
        }

        public override void OnTick(Tick tick)
        {
            //求平均值
            double Avg = MathHelper.Average(PriceLst);

            //求标准差
            double Std = MathHelper.Standard(PriceLst, 0, PriceLst.Count, Avg);

            //求最大值
            double Max = MathHelper.Max(PriceLst);

            //求最小值
            double Min = MathHelper.Min(PriceLst);
        }
    }
}
```

5 指标

5.1 指标概述

指标是基于原始的价格数据逐项进行一定规则的计算得出一套新的时间序列,例如均线就是对过去 N 个价格求平均。**MQ** 提供了一些常用的指标,例如 **MA**、**EMA**、**MACD**、**Bolling** 等,随着版本的更新会加入更多的基础指标。用户也可以基于 **Indicator** 基类,自定义开发指标。

Indicator 重要的属性如下表所示:

属性	含义	说明
IsValid	是否可用	使用指标前需要判断 IsValid
Count	数据个数	数据序列的个数
First	第一个数据	索引为 0 的数据
Last	最后一个数据	索引为 Count-1 的数据
OrigBar	原始数据序列	

Indicator 重要的方法如下表所示：

方法	含义	说明
Ago	回溯	回溯 N 会被自动限制在[0,Count-1]之间
Calcuat	指标算法	指标的核心算法写在这里
Cross	交叉关系	可以判断指标与时间序列之间的交叉关系 交叉关系有三种：上穿 Above 、下穿 Below 、无 None

5.2 指标用法

下面以移动平均线 **MA** 为例介绍如何在策略中使用指标。假设我们需要基于分钟 **K** 线获得 5 分钟均线。

首先在变量区定义分钟 **K** 线和指标。

```
BarSeries bars; //定义分钟K线
MA maquick; //定义指标
```

然后在 **Init** 时构造基础数据和指标。

注意：构造后要调用 **AddIndicator** 方法把指标加载到策略中去（相当于告诉策略我要计算这个指标），然后 **MQ** 会自动计算指标的各项数值。如果不对指标调用 **AddIndicator**，那么策略将不会对指标进行计算。

```
public override void Init()
{
    //构造K线，然后基于K线构造指标
    bars = GetBarSeries( EnumMarket.期货, 1, EnumBarType.分钟, 0);
    maquick = new MA(bars, 5);
    AddIndicator(maquick);
}
```

最后在需要使用指标的地方（例如 **OnTick**、**Exit**、**OnOrder**、**OnTrade**）调用。使用前可以通过 **IsValid** 验证其有效性。

```
if (maquick.IsValid )
{
    //...使用指标
}
```

5.3 MA

MA 指标是英文(Moving average)的简写，叫移动平均线指标。移动平均线 **MA** 本质上是一种趋势追踪指标，便于识别趋势已经终结或者反转，新的趋势是否正在形成。定义一根均线需要知道平均的对象 **X** 和回溯 **N**，就可以计算移动平均线的值。

$$MA = \frac{X_t + X_{t-1} + X_{t-2} + \cdots + X_{t-N+1}}{N}$$

其中：

X_i = 序列中的第 i 个数据；

t 是计算的起点（一般是最后一根 **K** 线或者最新 **Tick**）；

N = 回溯求平均的个数

关于均线 **MA** 指标的用法，可以参考策略范例中的[双均线策略](#)。

MA 有 3 中构造方法。

```
public MA(BarSeries barlist, BarData bardata, int N)
```

barlist 是 **K** 线，**bardata** 决定使用 **K** 线的哪个数据作（高、开、低、收）为计算基础，**N** 是计算平均值的回溯值。

```
public MA(BarSeries barlist, int N)
```

这种构造方法实际上是默认使用 **Close** 作为计算基础。

```
public MA(ISeries series, int N)
```

该构造方法直接使用了序列数据（例如价格时间序列）作为计算基础。

5.4 EMA

EMA (Exponential Moving Average), 指数平均数指标。**EMA** 也是一种趋向类指标, 指数平均数指标是以指数式递减加权的移动平均。

EMA 的计算公式如下:

$$EMA_t = EMA_{t-1} + \frac{2}{N+1} (X_t - EMA_{t-1})$$

N=回溯求平均的个数

X_t=计算起点 (一般是最后一根 K 线或者最新 Tick) 的数据;

与 **MA** 类似, **EMA** 也有三种构造方法:

一是根据 K 线、数据类型 **BarData** 和回溯 **N** 构造:

```
public EMA(BarSeries barlist, BarData bardata, int N)
```

二是默认 **BarData** 为 **Close**:

```
public EMA(BarSeries barlist, int N)
```

三是直接基于数据序列构造:

```
public EMA(ISeries series, int N)
```


5.5 MACD

MACD (Moving Average Convergence and Divergence) 称为指数平滑异同移动平均线，是从双移动平均线发展而来的，由快的移动平均线减去慢的移动平均线。

MACD 的计算公式如下：

$$\text{MACD} = \text{EMA}(P_t, n) - \text{EMA}(P_t, m)$$

P_t =最新数据（一般是最后一根 K 线或者最新 Tick）

n =短周期回溯参数，默认值为 12

m =长周期回溯参数，默认值为 26

默认回溯 $N=9$

MACD 有两类构造函数：

➤ 以 K 线数据为基础数据构造的布林带

```
public MACD(BarSeries barlist, BarData bardata, int Short, int Long, int N)
```

其中 **barlist** 是 K 线数据，**bardata** 是数据类型，**Short** 是短周期参数，**Long** 是长周期参数，**N** 是回溯参数。

其他的构造函数都是以该构造函数为基础衍生出来的。

```
public MACD(BarSeries barlist, BarData bardata) 缺省 Short=1, Long=26, N=9
```

```
public MACD(BarSeries barlist, int Short, int Long, int N) 缺省数据类型为 Close
```

```
public MACD(BarSeries barlist) 缺省数据类型为 Close, Short=1, Long=26, N=9
```

➤ 以数据序列为基础数据构造的布林带

```
public MACD(ISeries barlist, int Short, int Long, int N)
```

其中 **barlist** 是 K 线数据，**bardata** 是数据类型，**Short** 是短周期参数，**Long** 是长周期参数，**N** 是回溯参数。

```
public MACD(ISeries barlist) 缺省 Short=12, Long=26, N=9
```

5.6 Bolling

布林线(Bolling) 是根据统计学中的标准差原理设计出来的一种非常实用的技术指标。它由三条轨道线组成, 其中上下两条线分别可以看成是价格的压力线和支撑线, 在两条线之间是一条价格平均线。其计算公式如下:

$$\text{Bolling} = \text{MA}(\text{P}, \text{N}) \pm k * \text{Std}(\text{P}, \text{N})$$

N=回溯周期数

P=最新数据 (一般是最后一根 K 线或者最新 Tick)

Std=标准差

K=系数

类似的, Bolling 指标也有多个构造函数。

➤ 以 K 线为基础数据构造布林带

```
public Bolling(BarSeries barlist, BarData bardata, int N, double K)
```

这是以 K 线为基础数据的构造函数, barlist 是基础数据, bardata 表示数据类型, N 是回溯周期数, K 是标准差前的乘数。

```
public Bolling(BarSeries barlist, int N, double K) 缺省 K 线的数据类型为 Close。
```

```
public Bolling(BarSeries barlist, BarData bardata) 缺省回溯 N=20, 系数 K=2
```

```
public Bolling(BarSeries barlist, int N) 缺省 K 线的数据类型为 Close, 系数 K=2
```

```
public Bolling(BarSeries barlist) 缺省 K 线的数据类型为 Close, 回溯 N=20, 系数 K=2
```

➤ 以数据序列为基础数据构造布林带

```
public Bolling(ISeries barlist, int N, double K)
```

这是以数据序列为基础数据的构造函数, N 是回溯周期数, L 是标准差前的乘数。

```
public Bolling(ISeries barlist, int N) 缺省系数 K=2
```

```
public Bolling(ISeries barlist) 缺省回溯 N=20, 系数 K=2
```

5.7 自定义指标

在熟悉 **Indicator** 基类结构的基础上，我们可以开发自定义指标。指标源码编译通过后，将生成的 **dll** 文件放到 **MQ** 根目录下，策略就可以调用到该指标了。下面是 **MA** 指标的源码，大家可以参考该源码学习如何开发一个自定义指标。

```
using System;
using Ats.Core;
namespace Ats.Indicators
{
    /// <summary>
    /// 均线
    /// </summary>
    public class MA : Indicator
    {
        #region 私有变量
        private DataSeries _malist = new DataSeries();//基础数据

        /// <summary>
        /// 计算平均值的数据个数
        /// </summary>
        private readonly int _N;

        /// <summary>
        /// 数据类型, 缺省Close
        /// </summary>
        private readonly BarData _bardata = BarData.Close;

        /// <summary>
        /// 当前位置
        /// </summary>
        private int _curIdx;//
        private double _latestValue = 0;//最新值
        private double _sum = 0;//和
        #endregion

        #region 属性

        /// <summary>
        /// 长度
        /// </summary>
        public int N
        {
            get { return _N; }
        }

        /// <summary>
        /// 数据类型
        /// </summary>
        private BarData BarData
        {
            get { return _bardata; }
        }
    }
}
```

```
/// <summary>
/// 根据索引访问数据
/// </summary>
/// <param name="index"></param>
/// <returns></returns>
public override double this[int index]
{
    get
    {
        if ( IsValid && index >= 0 && index <= _curIdx)
        {
            return _malist[index];
        }
        return 0;
    }
}

/// <summary>
/// 最后一个值
/// </summary>
public override double Last
{
    get
    {
        return IsValid ? _malist[_malist.Count - 1] : 0;
    }
}

/// <summary>
/// 第一个值
/// </summary>
public override double First
{
    get
    {
        return IsValid ? _malist[0] : 0;
    }
}

#endregion

#region 构造函数

/// <summary>
/// 构造函数
/// </summary>
/// <param name="barlist">K线</param>
/// <param name="n">均值N</param>
public MA(BarSeries barlist, int n)
: base(barlist)
{
    if (n < 1) n = 1;
    _N = n;

    _curIdx = n - 1;

    for (var i = 0; i < n; ++i)
    {
        _malist.Add(0);
    }
}
```

```
/// <summary>
/// 均线构造函数
/// </summary>
/// <param name="barlist">K线</param>
/// <param name="bardata">数据类型</param>
/// <param name="n">均值N</param>
public MA(BarSeries barlist,
    BarData bardata,
    int n)
    : base(barlist, bardata)
{
    if (n < 1) n = 1;

    _N = n;
    _curIdx = n - 1;
    _bardata = bardata;

    for (var i = 0; i < n; ++i)
    {
        _malist.Add(0);
    }
}

public MA(ISeries series, int N)
    : base(series)
{
    if (N < 1) N = 1;

    _N = N;
    _curIdx = N - 1;

    for (var i = 0; i < N; ++i)
    {
        _malist.Add(0);
    }
}
#endregion

#region 方法
/// <summary>
/// 指标核心算法
/// </summary>
public override void Calculate()
{
    #region
    // 数据不足, 直接返回
    if (OrigBar.Count < N) return;

    // 初始化第一个有效数据
    if (!IsValid)
    {
        //求和
        for (var i = 0; i < N; ++i)
        {
            _sum += OrigBar[i];
            _malist[i] = Math.Round(_sum / (i + 1), 6);
        }
    }
}
```

```
        _latestValue = OrigBar[N - 1];
        //_malist[N - 1] = Math.Round( _sum / N, 6 );
        IsValid = true; //指标数据有效
    }

    // 如果原始_curIdx数据有更新, 那么更新ma对应数据
    var dvalue = OrigBar[_curIdx] - _latestValue;
    if (dvalue != 0)
    {
        _sum += dvalue;
        _malist[_curIdx] = Math.Round(_sum / N, 6);
    }

    // 如果_curIdx后面还有原始数据, 出来对应的ma
    for (var i = _curIdx + 1; i < OrigBar.Count; ++i)
    {
        _sum += (OrigBar[i] - OrigBar[i - N]);
        _malist.Add( Math.Round( _sum / N, 6 ));
    }
    // 设置_curIdx到最新位置, 更新_latestValue
    _curIdx = OrigBar.Count - 1;
    _latestValue = OrigBar[_curIdx];
    #endregion
}

/// <summary>
/// 指标回溯算法
/// </summary>
/// <param name="n"></param>
/// <returns></returns>
public override double Ago(int n)
{
    //注意将回溯n限制在合理范围
    n = Math.Min(Math.Max(0, n), _curIdx + 1 - N);
    return _malist.Ago(n);
}


/// <summary>
/// 释放资源
/// </summary>
public override void Dispose()
{
    if (_malist != null)
    {
        _malist.Clear();
    }
}
#endregion
}
```

6 复盘

6.1 概述

只有做到尽可能精确的复盘（回测），才能获得预期的实盘效果。基于海量的历史 Tick 和 K 线数据，MQ 可以进行 Tick 级的策略回测。

MQ 的复盘账户默认是 2 个账户：股票账户和期货账户。其绩效考量的是两个账户的资产总值。如果我们的策略只做股票，可以将期货账户的初始金额设置为 0，如果只做期货，可以将股票账户的初始金额设置为 0。



模拟账户配置对话框，包含期货和股票两个账户的配置项。

期货配置：

- 初始资金：100 万
- 多头保证金比率：17.0 %
- 空头保证金比率：17.0 %
- 交易成本：0.1000 %

股票配置：

- 初始金额：0 万
- 买入成本：0.100 %
- 卖出成本：0.100 %

底部有“确认”（带绿色对勾图标）和“取消”（带红色叉号图标）按钮。

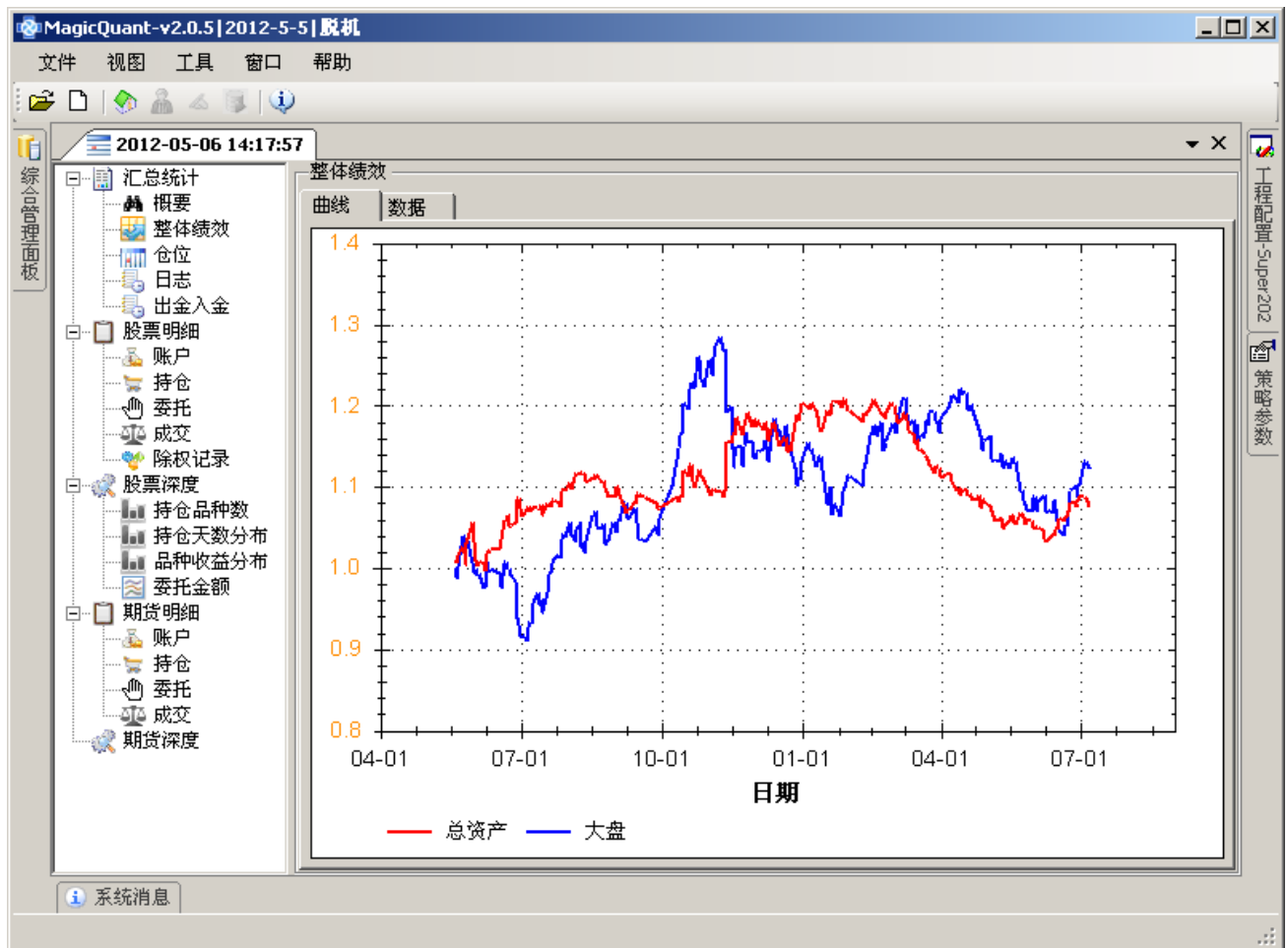
6.2 交易成本

准确合理的设置交易成本对于精确评估策略的绩效是至关重要的。实际交易中的成本包含两部分：一部分是显性成本，主要包括交易佣金、印花税、过户费等固定成本。具体的数值需要根据实际情况配置。一部分是隐性成本，主要包括冲击成本和资金成本。冲击成本对于价格变动迅速或者流动性较差的品种而言是需要做更为审慎的考量。

策略方面建议对于大资金策略交易该类品种时，加入各类拆单技术，委托量不要超过当时买一卖一档的挂单量，交易量占品种总成交量的比例比较小，委托频率要比较低，这样回测的结果才是真实可信的。

6.3 策略评价

在工程配置的【历史记录】-【复盘】子目录下可以双击打开复盘结果。左侧的树结构会列出各项内容，单击树的节点，在主界面可以显示对应的子项目。



【汇总统计】是对策略的整体评价，其下又分为【概要】、【整体绩效】、【仓位】、【日志】以及【出金入金】。

【概要】中包含汇总信息（盈亏、年化回报率、最大回撤、夏普系数）、复盘配置以及策略参数等。

【整体绩效】显示策略的总资产线。

【仓位】显示策略的总仓位。该项针对用股指期货对冲股票的策略有意义。

$$\text{股票仓位} = \frac{\text{股票市值}}{\text{股票账户总资产}}$$

$$\text{整体总仓位} = \frac{\text{股票市值} + \text{期货净市值}}{\text{股票账户总资产} + \text{期货账户动态权益}}$$

例如一个策略的股票账户和期货账户如下，由于股指期货的对冲，其整体总仓位=0%，实现了完全对冲。

项目	数值
股票市值	300 万
股票账户余额	700 万
股票账户总资产	=300+700=1000 万
股票总仓位	$= \frac{300}{1000} = 30\%$
多头股指期货持仓	1 手
空头股指期货持仓	-5 手
净持仓	-4 手
股指期货价格	2500
期货净市值	$= -4 * 300 * 2500 = -300 \text{ 万}$
期货账户动态权益	400 万
整体总仓位	$= \frac{300 + (-300)}{1000 + 400} = 0\%$

【日志】是策略运行过程中记录的日志，帮助我们检查策略的运行过程和逻辑是否正确。

【出金入金】中显示的是策略中调用修改账户现金 `ModifyFutureMoney(x)` 和 `ModifyStockMoney(x)` 的记录。

【股票明细】是股票账户交易的全部流水明细，包括【账户】、【持仓】、【委托】、【成交】、【除权记录】等。

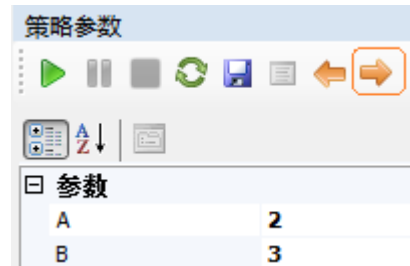
【股票深度】是在【股票明细】基础上进行数据挖掘得到的衍生数据，包括【持仓品种数】、【持仓天数分布】、【品种收益分布】、【委托金额】。

【期货明细】是期货账户交易的全部流水明细，包括【账户】、【持仓】、【委托】、【成交】等。

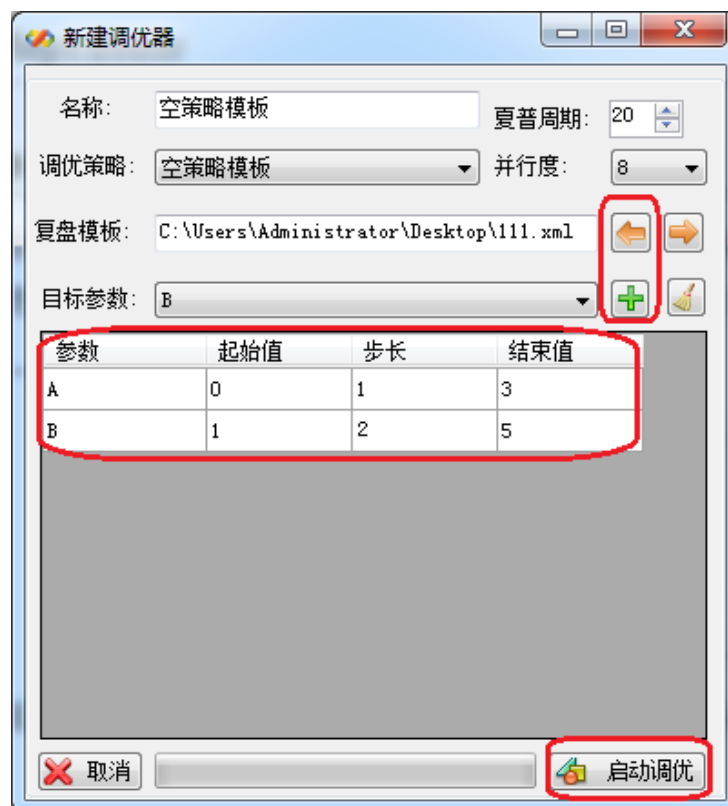
6.4 参数调优

参数调优就是针对多个参数，搜索最优参数组合的过程。MQ 支持搜索的参数值类型包括整型 Int、浮点型 Double 和布尔型 Bool。某策略有 2 个参数 A 和 B。A 参数的范围从 0 到 3，搜索步长为 1，B 参数的范围从 1 到 5，搜索步长为 2。

第一步将该策略的工程配置导出成 XML 文件。



然后点击【综合管理面板】→【策略管理】→【参数调优】，在弹出的【新建调优器】窗口中，导入该策略的工程配置文件，然后把目标参数添加到参数空间表中，并配置好起始值、步长、结束值，然后点击【参数调优】按钮即可。MQ 会在设定的参数空间中进行遍历式搜索。

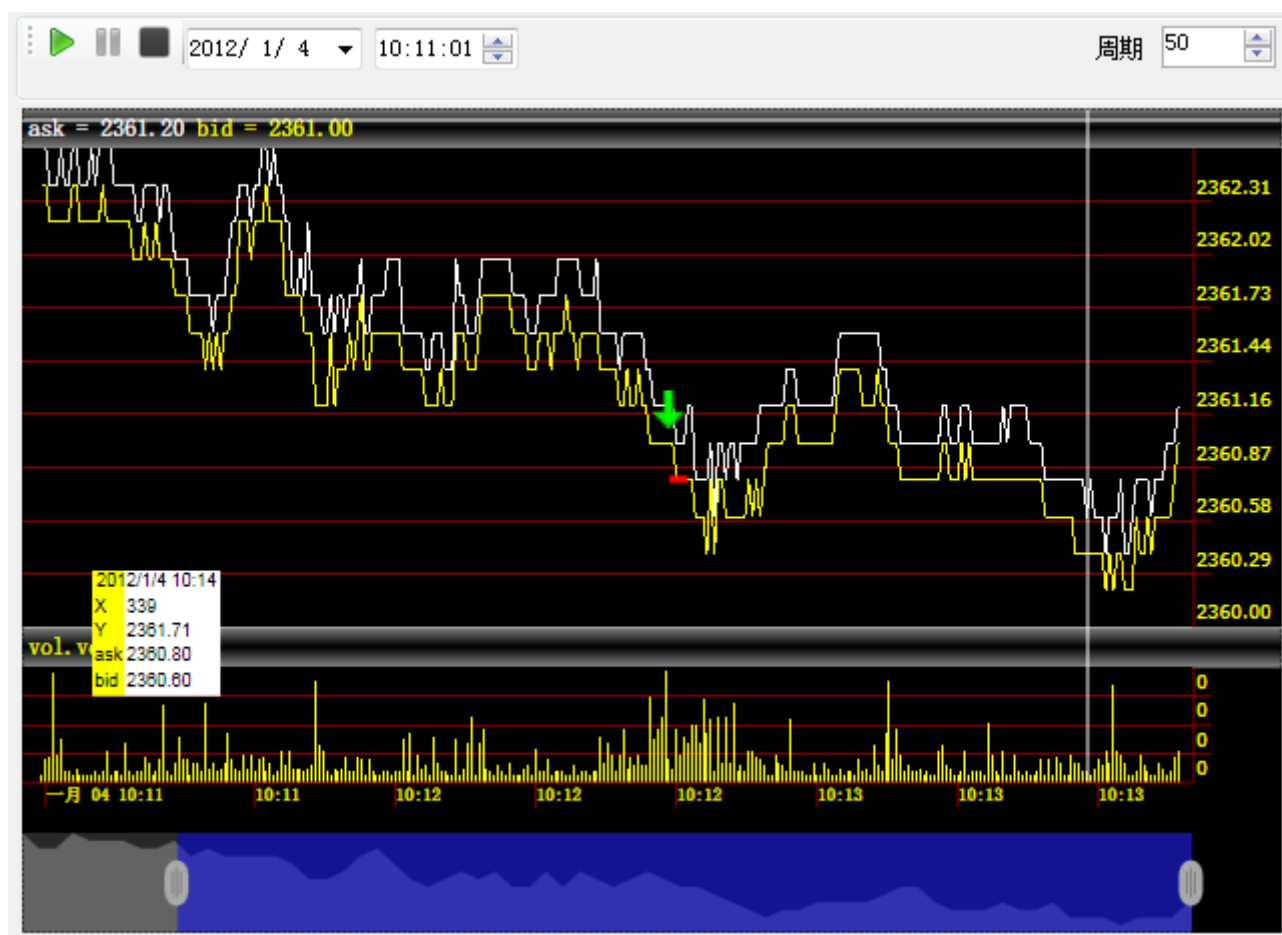


注意：在对新导入的策略进行参数调优前，需要重启 MQ。

6.5 Tick 回放

在复盘结果左侧的导航栏中，点击【期货深度】→【Tick 回放】，可以逐 Tick 重现策略的开仓平仓委托细节。委托图例的含义参考下表：

图例	含义
↑	买入开仓
↓	卖出开仓
-	卖出平仓
-	买入平仓



6.6 K 线回放

在复盘结果左侧的导航栏中，点击【期货深度】→【K 线回放】，可以在 K 线级别重现策略的开仓平仓委托细节。图中的连线表示开平成交对的盈亏连线。黄色的连线表示盈利的成交，紫红色的连线表示亏损的成交。



7 调试

策略开发好后，通过编译后说明策略没有语法错误，但并不能说明不存在逻辑错误，需要通过调试帮助我们排查策略逻辑中的错误。

7.1 日志调试

日志调试就是将策略运行中的信息输出到 MQ 日志中，当系统出现错误时通过对日志的检查找出策略的问题。同时对于策略发生交易动作时，也可以通过日志记录当时的信息。在策略中一些容易出错的环节输出日志是一个良好的策略开发习惯。

注意：MQ 底层的日志在根目录的 Log 文件下，主要用于排查平台层面的问题。策略的日志在工程界面就能通过双击浏览，主要用于监控策略，排查策略层面的问题。

在下面的例子中，策略首先检查订阅品种中是否包含 IF9999，如果没有订阅，则通过日志提示。策略开仓的逻辑很简单，如果发现 IF9999 的涨幅>1%，就开仓做多，并记录当时 IF9999 的涨跌幅。

```
string IFCode = "IF9999";
if ( AllFutures.Contains(IFCode))
{
    Future future = AllFutures[IFCode];
    if (future != null&& future.LastTick != null&& future.LastPrice>0 )
    {
        if (future.LastTick.Change > 1)
        {
            Print("IF9999涨跌幅="+ future.LastTick.Change.ToString());
            LimitOrder(IFCode, 1, future.LastPrice, EnumBuySell.买入);
        }
    }
}
else
{
    Print("订阅品种中没有IF9999，请检查");
}
```

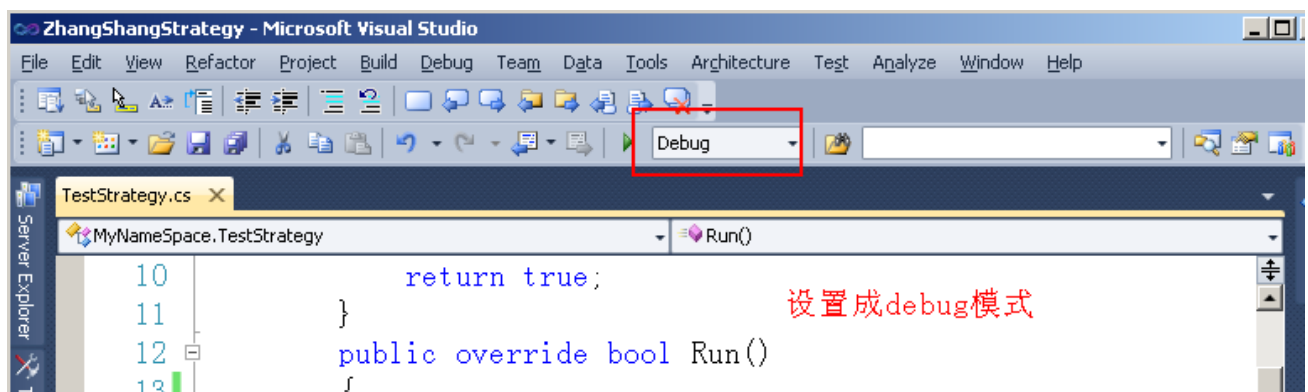
7.2 断点调试

日志调试虽然可以帮助我们排查问题，但是有一些不足。日志调试要求我们在写代码的时候考虑到可能出错的情况，并着重把我们需要知道的信息显示出来。如果开发策略时没有相应的提示信息，则不能提供出有用的信息。

我们需要更为强大的断点调试功能，让我们可以在策略运行中清晰地看到各个变量变化的过程，同时可以通过断点迅速定位到出错的位置。

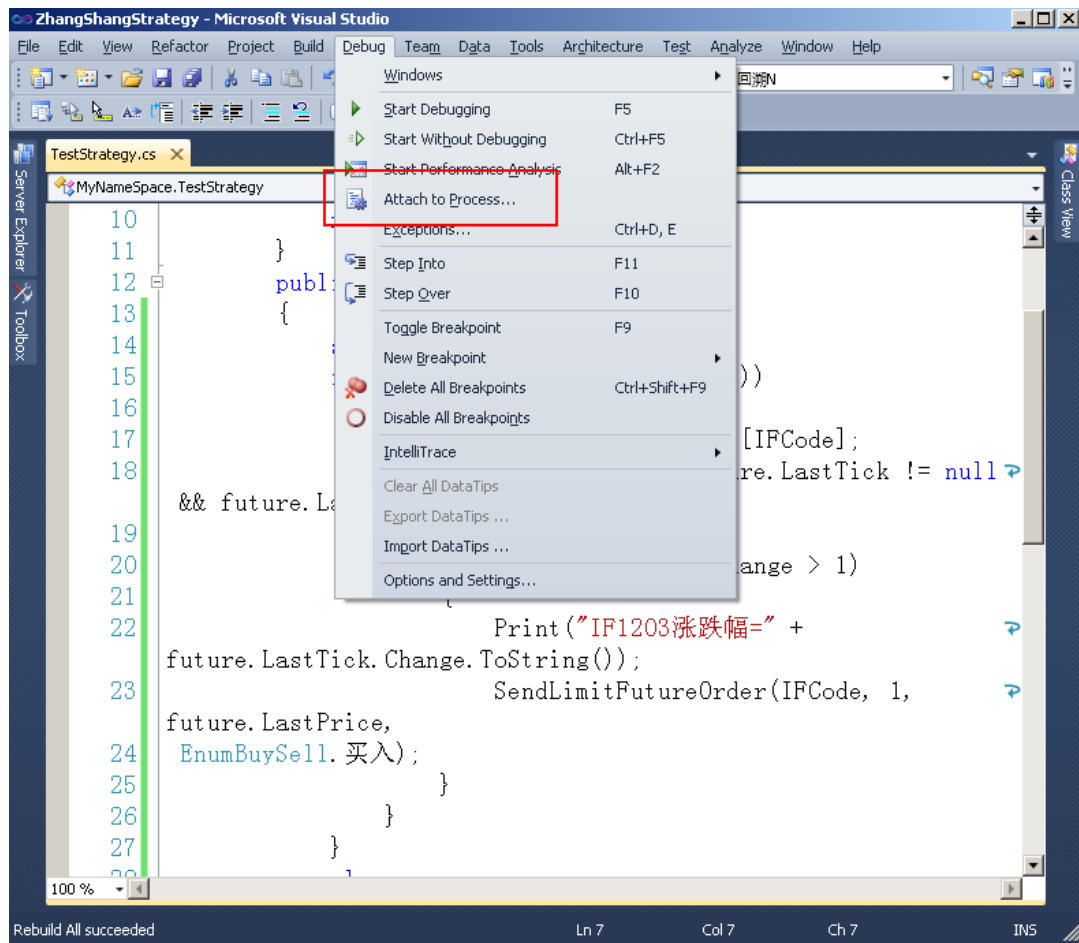
我们可以利用 **Visual Studio** 的调试引擎来对 **MQ** 进行调试。

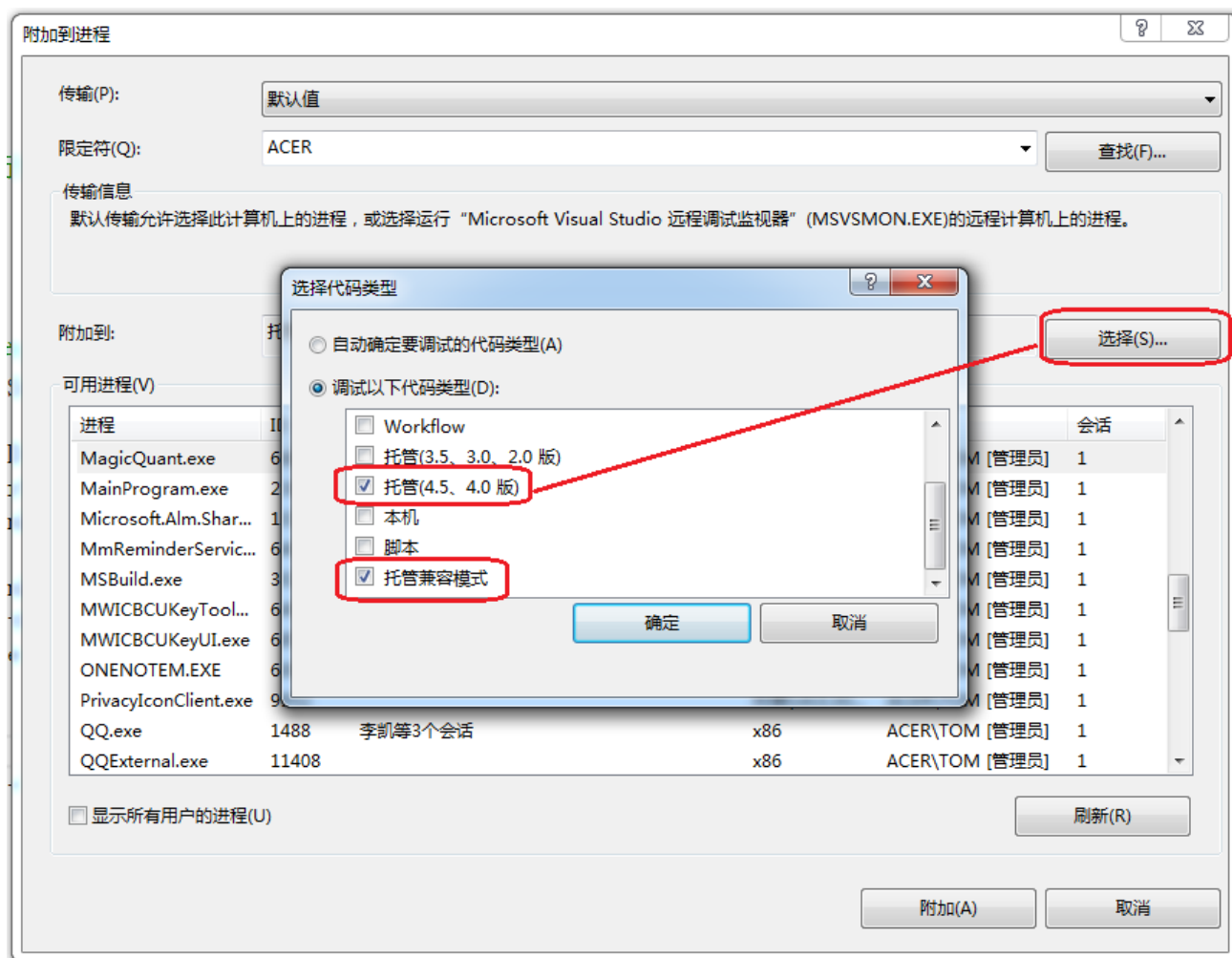
首先运行 **MQ**，然后用 **Visual Studio** 打开策略源码的解决方案文件(.sln)，并将编译选项配置成 **debug** 模式。



然后编译好策略 **dll**，运行 **MQ**，将编译好的策略 **dll** 加载到 **MQ** 里面。完成创建工程，配置好策略参数和订阅品种等准备工作。最后在 **Visual Studio** 里将 **MQ** 的主进程 **MagicQuant.exe** 附加到 **Visual Studio** 的调试器上。

指令	说明
F5	调试直接运行到指定断点
F10	单步调试
F11	单步调试，跟入调用的方法





7.3 弹出消息

策略可以调用方法 [ShowMessage](#) 来弹出消息。在弹出消息窗口中，如果用户点击“确认”，策略将继续执行；如果用户点击“取消”，策略将终止执行；如果用户不点击按钮，策略将一直等待。范例代码如下：

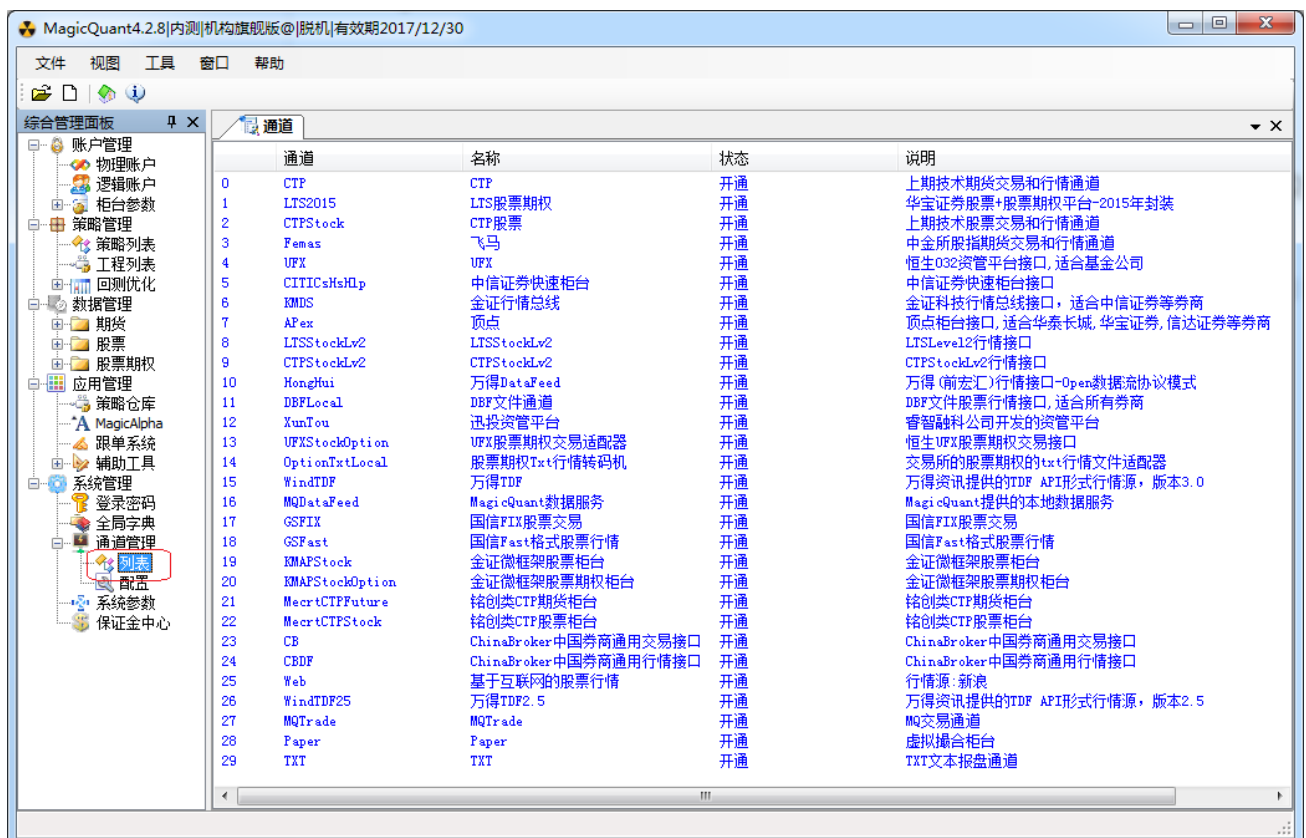
```
ShowMessage("确认" + MyDirection.ToString() + 委托数量.ToString() + "张" + 品种代码 + "@价格" + 委托价格.ToString());
```

8 通道配置

8.1 通道概述

MQ 采用通道 Channel 的方式对接外部的行情源和交易柜台，这样做的好处是 MQ 平台可以很灵活的根据用户的交易环境进行配置，实现插件式的引用。

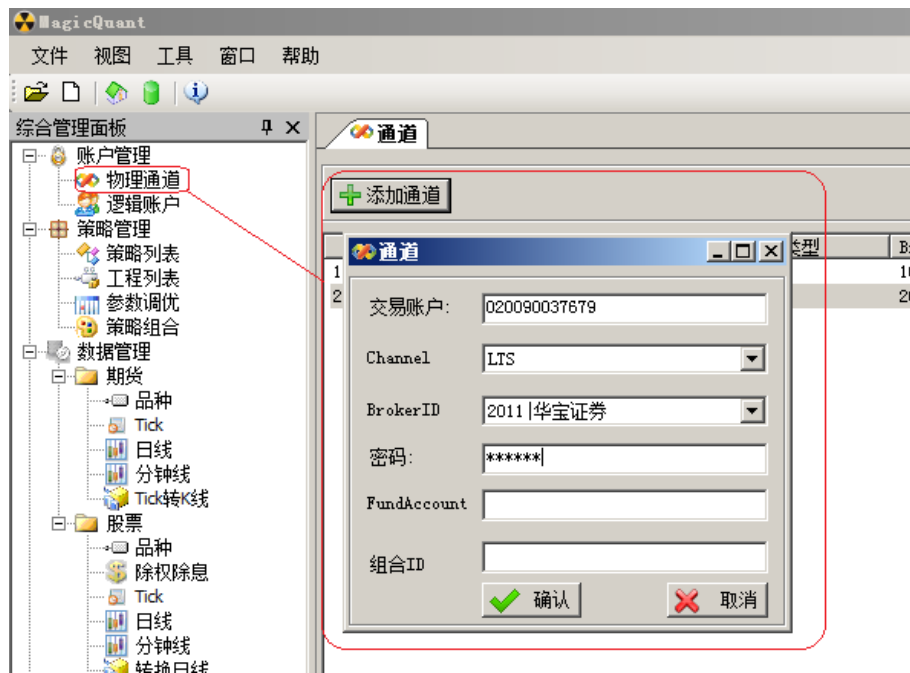
用户在【综合管理面板】→【系统管理】→【通道管理】→【列表】中可以看到 MQ 目前支持的通道。如果需要开通某个通道，可以联系 MQ 的客服人员进行开通。



8.2 LTS

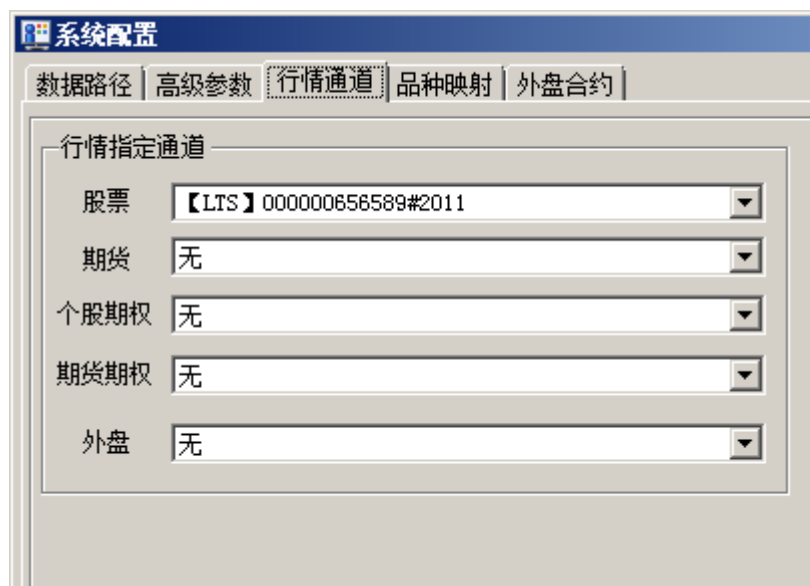
MQ 支持接入华宝证券的 **LTS** 柜台系统。接入前需要做如下准备工作：

华宝证券的交易和行情是分开两个账户配置的。华宝证券 **LTS** 柜台系统的交易和行情的账号是一样的，密码可能是不一样的。在【账户管理】→【物理通道】菜单中，新建一个交易通道和行情通道。其中交易通道选择 **Channel** 为 **LTS2015**，行情通道选择 **Channel** 为 **LTS2015**。



注意：由于交易和行情的账号相同，直接添加的话，系统会提示重复。因此在添加 **LTS** 行情通道的时候，设置【组合 ID】为 **1**，这样就可以解决资金单元冲突的问题。

然后在【工具】→【系统配置】→【行情通道】里指定好股票或者股票期权使用的通道。



8.3 万得 TDF

MQ 支持接入万得提供的 DataFeed 实时行情源。使用该通道前，需要联系 MQ 的工作人员提供相关的适配器文件。

用户需要购买万得的 DataFeed 账户才能使用该行情源。

在【账户管理】-->【物理通道】里面添加一个物理通道，Channel 选择【HongHui】，输入万得方面提供的账户和密码，然后确认。

最后在菜单【工具】-->【系统配置】-->【行情通道】处对应的地方选择万得的物理通道。

8.4 CTP

9 策略范例

注意：本手册中所有的代码、策略以及范例旨在为阐述、讲解函数、语法以及语句的实现等学习目的，而非提供商业用途。**本手册不保证策略范例在实际交易中的有效性及可赢利性。**请对其进行正确研判并结合市场情况改进后方可使用。使用本文档所有代码导致的交易结果，均由交易者自己承担。

策略范例都在单元测试代码包中，可以从 QQ 群 233309671 的群共享中下载最新的版本。我们将不断丰富完善策略范例，帮助量化交易者实现各种类型的交易模型。

9.1 日内波动

日内波动是一种交易模式，指的是持仓时间短，不留过夜持仓的交易方式。日内波动策略获利的源泉是标的价格在日内的大幅波动，因此而得名。

状态机模型可以描述日内波动策略的行为特点。一个基本的原型是策略只有三种状态：

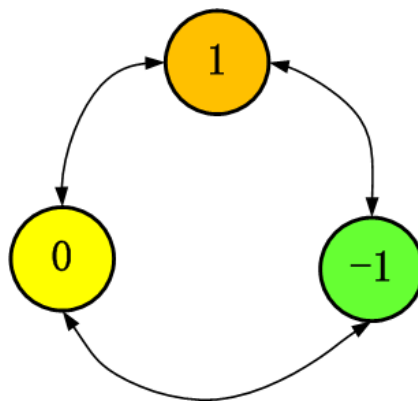
1 看多

-1 看空

0 无方向

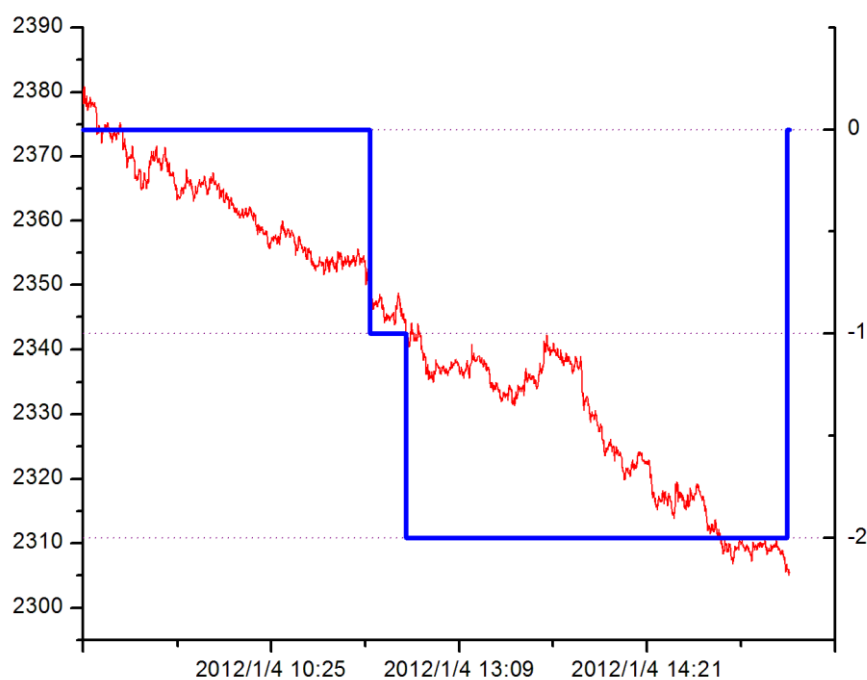
我们把看多标识成 1，看空标识成 -1，无方向标识成 0。这 3 种状态正好分别对应了交易上的三种持仓状态：持有多单，持有空单，不持仓。策略在任何一个时刻只能处于这三个状态中的一种（且只能是一种）。策略可以在三种状态之间来回转换。

在实际交易中，将三态状态机扩展到多态状态机就可以实现更多张持仓的日内波动策略。



这种用多个离散的整型数字表示策略状态位的方法，我们就称之为状态机。使用状态机这种方式的好处是逻辑清晰，状态切换无缝，并且比较容易实现多策略混合。下图的红线是标的的价格，蓝线则是策略的状态机。状态机随着时间的推移，依次发生切换：

0→-1→-2→0



基于 MQ 的策略 API，我们进一步封装了交易接口，基于 Tick 驱动的策略运行方式，开发了日内波动的策略基类 BaseStrategy，实现了下单，追单，成交回报处理等细节交易逻辑。继承 BaseStrategy 开发日内波动策略，会大幅度提高日内波动策略开发的效率。

9.2 红三兵

红三兵亦称“三红兵”，是三根阳线，依次上升，形成红三兵形态。它是一种很常见的 K 线组合，这种 K 线组合出现时，后势看涨的情况居多。我们用程序的语言描述这种交易逻辑：

基于的 K 线周期：M 分钟。例如 M=5 时，就是基于 5 分钟 K 线计算信号。



入场信号：

如果 K 线连续出现 N 根阳线，做多；

如果 K 线连续出现 N 根阴线，做空；

出场信号：

如果策略处于做空状态中，K 线连续出现 N-1 根阳线，平仓出场；

如果策略处于做多状态中，K 线连续出现 N-1 根阴线，平仓出场；

日内波动策略，收盘前平仓。

策略采用 Tick 驱动，调用 K 线，入场信号和出场信号都只在 K 线走完的时候进行计算。

※源代码\单元测试\策略\日内波动\DS_01_3Red.cs

```
using System;
using Ats.Core;
using Ats.Indicators;
namespace MagicQuantTest
{
    /// <summary>
    /// 红三兵策略
    /// </summary>
    public class DS_01_3Red : BaseStrategy
    {
        //入场信号:
        //如果K线连续出现N根阳线, 做多;
        //如果K线连续出现N根阴线, 做空;
        //出场信号:
        //如果策略处于做空状态中, K线连续出现N-1根阳线, 平仓出场;
        //如果策略处于做多状态中, K线连续出现N-1根阴线, 平仓出场;
        //策略采用Tick驱动, 调用K线, 入场信号和出场信号都只在K线走完的时候进行计算。

        [Parameter(Display = "M", Description = "", Category = "红三兵")]
        public int M = 5;

        [Parameter(Display = "N", Description = "连续N根阳线做多, 连续N根阴线做空", Category = "红三兵")]
        public int N = 3;

        BarSeries kLine;

        #region 策略事件

        public override void Init()
        {
            PreInit("策略版本 V1", "05", true, new DateTime(2019, 1, 1));

            //查询K线, 把K线作为变量放在内存中
            kLine = GetBarSeries(EnumMarket.期货, DefaultFutureCode, M, EnumBarType.分钟);
        }

        /// <summary>
        /// Tick触发
        /// </summary>
        /// <param name="tick">最新的Tick</param>
        public override void OnTick(Tick tick)
        {
            if ( PreOnTick(tick) )
            {
                Calculate(tick);

                AutoTrade();
            }
        }

        /// <summary>
        /// 红三兵
    }
}
```

```
/// </summary>
/// <param name="tick"></param>
public override void Calculate(Tick tick)
{
    if (StrategySignal == 0)
    {
        //没有持仓时，关注入场信号
        int tmp = GetRecentKLineDirection(kLine, N);
        if (tmp == 1)
        {
            //如果K线连续出现N根阳线，做多
            StrategySignal = 1;
        }
        else if (tmp == -1)
        {
            //如果K线连续出现N根阴线，做空
            StrategySignal = -1;
        }
    }
    else
    {
        //有持仓时，关注出场信号
        int tmp = GetRecentKLineDirection(kLine, N-1);
        if (StrategySignal > 0)
        {
            if (tmp == -1)
            {
                //如果策略处于做多状态中，K线连续出现N-1根阴线，平仓出场
                StrategySignal = -1;
            }
        }
        else if (StrategySignal < 0)
        {
            if (tmp == 1)
            {
                //如果策略处于做空状态中，K线连续出现N-1根阳线，平仓出场
                StrategySignal = 1;
            }
        }
    }
}

/// <summary>
/// 计算最近backN根K线的连续状态
/// </summary>
/// <param name="bars">K线</param>
/// <param name="backN">回溯N</param>
/// <returns>1连续阳线 -1连续阴线 0都不是</returns>
public int GetRecentKLineDirection(BarSeries bars, int backN)
{
    #region

    if (bars != null) //非空判断是必须的
    {
        if (bars.Count >= backN + 1)
        {
            //由于最新的一根K线是刚刚新创建的，因此不参与连续阳线或者阴线的计算
        }
    }
}
```

```
int sum = 0;
//判断连续N根K线阳线或者阴线
for (int i = 1; i <= 1 + backN; i++)
{
    int tmp = 0;
    Bar bar = bars.Ago(i);
    //这里用MinDouble, 避规Double的精度问题
    if (bar.BarChange > MinDouble)
    {
        tmp = 1;
    }
    else if (bar.BarChange < -MinDouble)
    {
        tmp = -1;
    }
    sum += tmp;
}
if (sum >= backN)
{
    return 1; //连续阳线
}
else if (sum <= -backN)
{
    return -1; //连续阴线
}
}
}
return 0;
#endregion
}
#endregion
}
```

9.3 观察区间突破

观察区间突破策略的思路是开盘以后，在一个观察窗口内（蓝色竖虚线左侧），例如 30 分钟，根据价格的最高点（上轨）和最低点（下轨）建立一个价格区间。

入场信号：

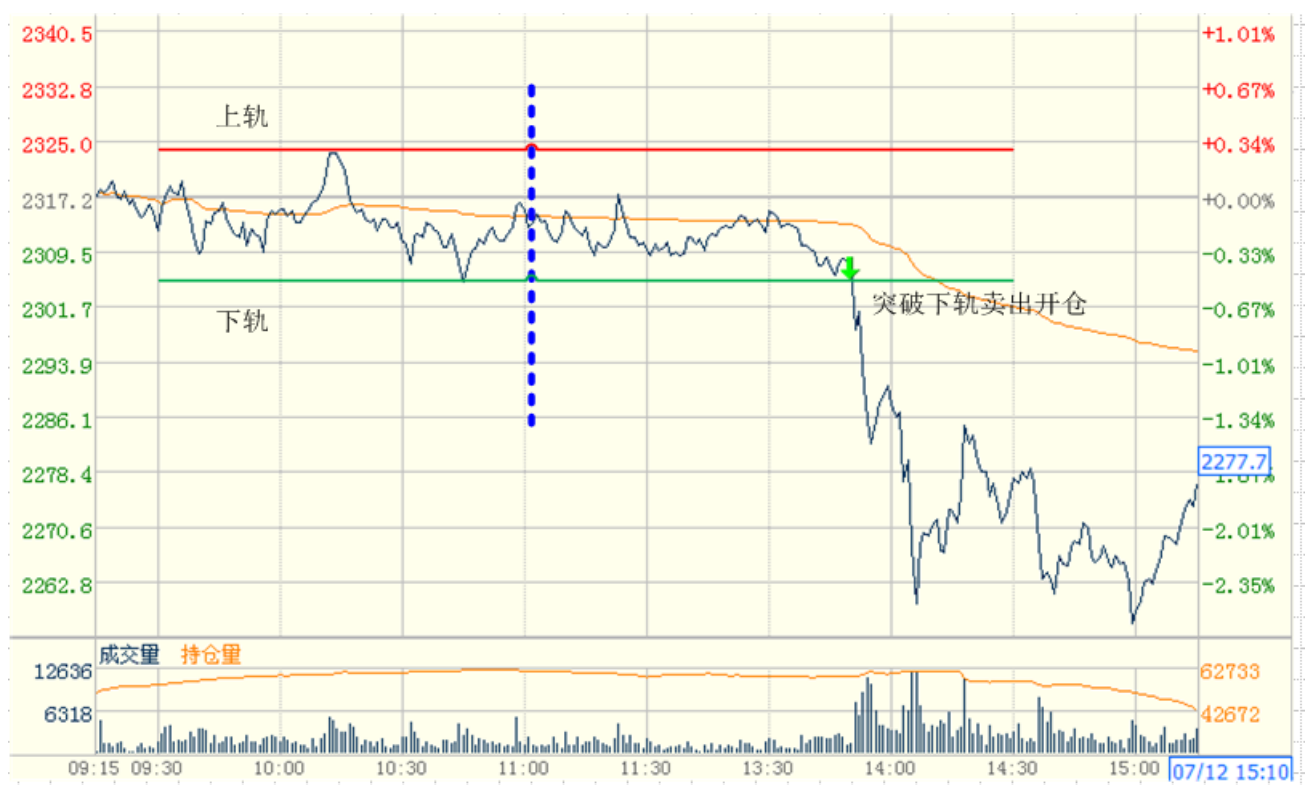
当观察结束后（蓝色竖虚线右侧），如果价格上涨突破上轨，则做多；

当观察结束后（蓝色竖虚线右侧），如果价格下跌突破下轨，则做空；

出场信号：

如果信号发生反转，则平掉原有持仓，按照新的信号入场。

日内波动策略，收盘前平仓。



※源代码\单元测试\策略\日内波动\DS_02_WatchBreak.cs

```
using System;
using Ats.Core;
using Ats.Indicators;
using System.Collections.Generic;
using System.IO;

namespace MagicQuantTest
{
    /// <summary>
    /// 观察区间突破
    /// </summary>
    public class DS_02_WatchBreak : BaseStrategy
    {
        #region 观察区间突破
        //观察区间突破策略的思路是开盘以后，在一个观察窗口内（蓝色竖虚线左侧）
        //例如30分钟，根据价格的最高点（上轨）和最低点（下轨）建立一个价格区间。
        //入场信号：
        //当观察结束后（蓝色竖虚线右侧），如果价格上涨突破上轨，做多；
        //当观察结束后（蓝色竖虚线右侧），如果价格下跌突破下轨，做空；
        //出场信号：
        //如果信号发生反转，则平掉原有持仓，按照新的信号入场。
        //日内波动策略，收盘前平仓。
        #endregion

        [Parameter(Display = "开盘观察区间", Description = "开盘观察区间", Category = "时间框架")]
        public double 开盘观察区间 = 100;

        #region

        /// <summary>
        /// 上轨
        /// </summary>
        double UpLine = -100000;

        /// <summary>
        /// 下轨
        /// </summary>
        double DownLine = 100000;

        /// <summary>
        /// 停止观察时间
        /// </summary>
        DateTime TCut;
        #endregion

        #region 策略事件

        /// <summary>
        /// 策略启动的时候执行
        /// </summary>
        public override void Init()
        {
            PreInit("策略版本 V12015-4-21 ", "WB", true, new DateTime(2018, 1, 1), 100, true );

            //启动策略的时候计算好停止观察时间
        }
    }
}
```

```
TCut = tOpen.AddMinutes(开盘观察区间);
Print("停止观察时间=" + TCut.ToString());
}

/// <summary>
/// Tick触发
/// </summary>
/// <param name="tick">新的行情Tick</param>
public override void OnTick(Tick tick)
{
    if ( PreOnTick(tick))
    {
        Calculate(tick);

        AutoTrade();
    }
}

/// <summary>
/// 观察区间突破
/// </summary>
/// <param name="tick">最新的Tick</param>
public override void Calculate( Tick tick )
{
    if ( TickNow <= TCut)
    {
        #region Tick的时间在停止观察时刻之前，维护上轨和下轨
        UpLine = Math.Max( UpLine, LastPrice); //维护价格区间上轨
        DownLine = Math.Min(DownLine, LastPrice); //维护价格区间下轨
        #endregion
    }
    else
    {
        if ( LastPrice >= UpLine )
        {
            //如果价格上涨突破上轨，做多
            StrategySignal = 1;
        }
        else if (LastPrice <= DownLine)
        {
            //如果价格下跌突破下轨，做空
            StrategySignal = -1;
        }
    }
}
#endregion
}
```


9.4 日内开盘突破

开盘价，是买卖双方消化隔夜各种多空消息后达成的对当天价格的一致初始预期。

策略逻辑：

价格在开盘价之上运行，代表了当天多头强势；

价格在开盘价之下运行，代表了当天空头强势；

开盘突破系统，以开盘价为基础，触发买卖信号。这套系统的原型可以非常简单，简单到开盘价之上，做多、开盘价之下，做空。

时间过滤：为了避免价格往复开盘价所带来的尴尬，我们采用时间过滤的方法，过滤噪声。

入场信号：

价格在开盘价上持续 N 分钟，做多；

价格在开盘价下持续 N 分钟，做空；

出场信号：

日内波动策略，收盘前平仓。

※源代码\单元测试\策略\日内波动\DS_03_OpenStand.cs

```
using System;
using Ats.Core;
using Ats.Indicators;

namespace MagicQuantTest
{
    /// <summary>
    /// 日内开盘突破
    /// </summary>
    public class DS_03_OpenStand : BaseStrategy
    {
        [Parameter(Display = "持续时间阈值", Description = "单位:分钟", Category = "时间框架")]
        public double 持续时间阈值 = 80;

        double 持续时间 = 0;

        double 开盘价 = 0;

        /// <summary>
        /// 内部标志位
        /// </summary>
        public int MyFlag = 0;
    }
}
```

```
DateTime 开始计算时间;

#region 策略事件

public override void Init()
{
    PreInit("策略版本 V1", "OS", true, new DateTime(2019, 1, 1));

    开始计算时间 = Now;
}

/// <summary>
/// Tick触发
/// </summary>
/// <param name="tick"></param>
public override void OnTick(Tick tick)
{
    if ( PreOnTick(tick) )
    {
        Calculate(tick);

        AutoTrade();
    }
}

/// <summary>
/// 日内开盘突破
/// </summary>
/// <param name="tick"></param>
public override void Calculate(Tick tick )
{
    #region 策略的核心逻辑
    开盘价 = tick.OpenPrice;
    //计算开盘持续时间
    持续时间 = (TickNow - 开始计算时间).TotalMinutes;
    //注意考虑中午休市的时间
    if (TickNow >= PMBegin && 开始计算时间 <= AMEnd)
    {
        持续时间 = 持续时间 - (PMBegin - AMEnd).TotalMinutes;
    }
    int 当前位置 = Math.Sign(LastPrice - 开盘价);
    //通过Math.Sign判断价格在开盘价上面还是下面
    if ( 当前位置 != 0)
    {
        if ( MyFlag == 0)
        {
            开始计算时间 = TickNow; //标记初始状态
            MyFlag = 当前位置;
        }
        else
        {
            if ( MyFlag * 当前位置 > 0)
            {
                //价格持续在开盘基准一边
                if (持续时间 >= 持续时间阈值)
                {
                    if ( 当前位置 > 0)

```

```
{
    if (StrategySignal <= 0)
    {
        //价格在开盘价上持续N分钟，做多
        Print("持续时间=" + 持续时间.ToString()
            + "分钟位于开盘价之上做多, t=" + Now.ToString());
        StrategySignal = 1;
    }
}
else if (当前位置 < 0 )
{
    if (StrategySignal >= 0)
    {
        //价格在开盘价下持续N分钟，做空
        Print("持续时间=" + 持续时间.ToString()
            + "分钟位于开盘价之下做空, t=" + Now.ToString());
    }
    StrategySignal = -1;
}
}
else
{
    //价格如果穿到另一边去的话，则变换符号
    开始计算时间 = TickNow;
    MyFlag = 当前位置;
}
}
}
#endregion
}
#endregion
}
```

9.5 包 Package

9.6 R-Break

9.7 Dural Trust

9.8 轴枢点

9.9 多参数混合

9.10 多周期混合

9.11 多策略混合

9.12 股票 T+0 策略模板

9.13 股票 Alpha 策略

9.14 股票实盘策略

9.15 股票动量策略

9.16 唐奇安通道

9.17 双均线

双均线策略中使用 2 根均线（一根快速均线和一根慢速均线）作为买卖的信号。当快速均线向上穿过慢速均线时做多；而快速均线向下穿过慢速均线时做空。

※源代码 \策略范例\双均线策略.cs

9.18 三均线

三均线系统是在双均线系统基础上衍生变化得来的，本质上也是一个均线系统。其策略逻辑是建立在最新价、快速均线、中速均线、满速均线基础上的。

当（最新价>快速均线&&快速均线>中速均线&&中速均线>慢速均线）时，做多；当快速均线<中速均线时，结束做多，平多头仓；

当（最新价<快速均线&&快速均线<中速均线&&中速均线<慢速均线）时，做空；当快速均线>中速均线时，结束做空，平空头仓；

※源代码 \策略范例\三均线策略.cs

9.19 菲阿里四价

9.20 Aberration

9.21 横盘突破系统

9.22 空中花园

9.23 金肯特纳交易系统

9.24 布林强盗系统

9.25 动态突破系统

9.26 幽灵交易者

9.27 恒温器策略

9.28 网格交易策略

网格交易旨在震荡市中从市场噪声中获利,其基本思路是在当前价格上下一定范围内布置一系列“网格”,价格之上每隔一定距离布置一张空单,价格之下每隔一定距离布置一张多单。所有单子都设定止盈目标,但是不设置止损目标。如果某个价格位置的单子获利成交,则在该价位再布置同样的网格。

这种策略的优势是在震荡市中对于价格在一定范围内来回拉锯可以双向持续获利,但是在

强趋势突破行情时，会出现鱼死网破的情况。因此网格交易策略的关键因素在于风险控制。常用的风险控制的方法有：动态调整网格间距、设置账户亏损阈值、只运用在低波动性品种上、动态移动网格中枢等。

范例中演示一种最简单的网格：以价格均线作为中枢，在中枢上下距离一定倍数标准差的位置部署网格，等待价格以自然波动的方式“撞击”网格成交。一旦成交以后，立即挂上止盈单，并且撤掉原有的网格，以最新的价格均线作为中枢再次部署网格。

9.29 鳄鱼法则策略

9.30 做市商策略

9.31 多品种扫描

9.32 VWAP 拆单

9.33 撤单和追单

9.34 平仓及时反手

策略中经常需要先平仓，再反手。但是如果保证金额刚好只够一手，那么同时打出开仓和平仓单，必然导致废单。

细致的做法是：先平仓，等平仓单成交后立即（注意不是下一个 Tick）开仓。要实现这种精确的策略微流程控制，就需要用到成交回报。

在范例策略中假设只需要平 1 张单，再反手开一张，就用到 **OnTrade**。

※源代码 \策略范例\解决反手保证金不足.cs

9.35 期现套利

期现套利就是利用现货和期货的基差超过一定阈值实现套利。

※源代码 \策略范例\完全复制期限套利策略.cs

9.36 Alpha 对冲

对冲（[Hedge](#)）指特意降低另一项投资的风险或者取得某种较确定的收益。一般来说对冲是同时进行两笔行情相关、方向相反、数量相当、盈亏基本相抵的交易。

9.37 **Beta** 增强型对冲

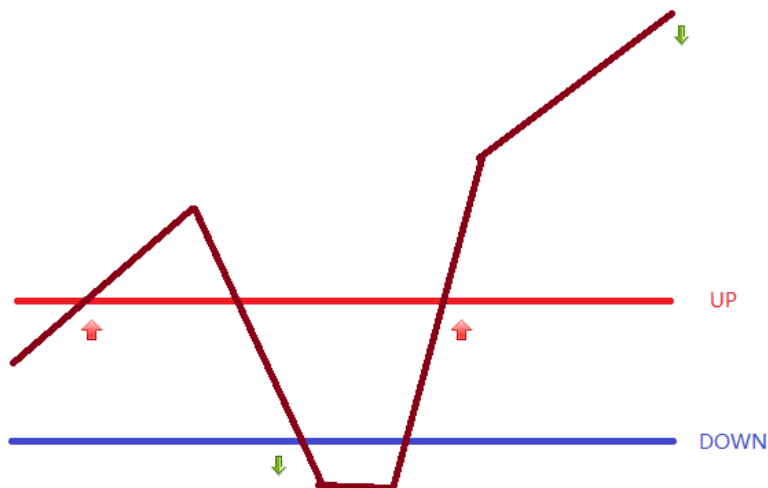
9.38 区间突破止盈

策略原理：

价格一旦突破某一个区间，会有一段较大的趋势行情。

交易逻辑：

- 1 如果价格突破区间上轨 **UP**，则持有一份多头头寸；
- 2 如果价格跌破区间下轨 **DOWN**，则持有 1 份空头头寸；
- 3 如果达到目标盈利金额则止盈出场，不再开仓；
- 4 策略可能隔夜。



如上图所示，按照策略的逻辑，首先在价格突破 **UP** 时开一份多单，当价格跌破 **DOWN** 时，把多单平掉，开空单；后面价格又涨上来突破 **UP**，则平掉空单，开多单；最后价格大幅上涨，累积盈利超过阈值，卖出平仓，不再开仓。

开发思路

在策略开发过程中，我们需要根据策略的逻辑流程解决下面几个问题：

1 策略的状态切换

这个策略在运行过程中是由多个状态切换衔接构成的，我们借鉴状态机的思路，将策略的状态用 `int` 型变量 `State` 标记：

```
/// <summary>
/// 策略状态
/// 0初始状态，等待建仓
/// 1初始建仓，已经挂单，等待成交
/// 2初始建仓完成，等待止盈或者反手
/// 3触发止盈，平仓离场
/// 4触发反手，先平仓
/// 5反手过程中，平仓全部完成，进行开仓；开仓完成后切换到2
/// 6平仓完成，不再开仓
/// </summary>

public int State = 0;
```

2 隔夜

隔夜策略的一个关键技术就是需要将策略的状态保存到文件中，同时在策略启动时从文件中读取恢复出来。在本策略中，策略的状态包括 `State`、方向 `Direction`、持仓数量 `Volume` 以及最新价格 `LastPrice`。

```
/// <summary>
/// 保存策略状态信息
/// </summary>
public void SaveState()
{
    //将策略状态写入文件, 便于下次启动策略的时候读取信息
    //DateTime, State, Direction, Volume, LastPrice
    //每次保存都是追加新的信息，便于跟踪状态变化
    string str = "";
    str += Now.ToString() + ",";
    str += State.ToString() + ",";
    str += Direction.ToString() + ",";
    str += Volume.ToString() + ",";
    str += LastPrice.ToString() + ",";
    str += "\r\n";
    File.AppendAllText(StateFile, str, Encoding.Default);
}

/// <summary>
/// 载入状态信息
/// </summary>
public void LoadState()
```



```
{
    //需要从历史文件中提取的信息
    //从后面往前读取，总是加载最新的信息
    if (File.Exists(StateFile))
    {
        //读取解析
        string[] lines = File.ReadAllLines(StateFile, Encoding.Default);
        char[] sp = ", ".ToCharArray();
        //遍历文本数据
        for (int i = lines.Length - 1; i > 0; i--)
        {
            string str = lines[i];
            if (str.Length > 5)
            {
                //用 , 区分开
                string[] data = str.Split( sp );
                //按照顺序解析
                //DateTime, State, Direction, Volume, LastPrice
                DateTime time = DateTime.Parse(data[0]);
                State = int.Parse(data[1]);
                Direction = int.Parse(data[2]);
                Volume = int.Parse(data[3]);
                LastPrice = double.Parse(data[4]);

                Print("解析完成，最新的状态的时间=" + time.ToString());

                break;
            }
        }
    }
    else
    {
        Print("找不到状态文件[" + StateFile + "]");
    }
}
```

3 反手交易

策略有反手的逻辑，即当持有多单时，如果出现做空信号，需要平掉多头持仓，开新的空头持仓；当持有空单时，如果出现做多信号，需要平掉空头持仓，开新的多头持仓。反手交易需要先平仓，等平仓委托成交后，再开仓。如果同时发出平仓和开仓的委托，可能由于资金不足导致开仓委托被拒绝。因此策略发生反手信号的时候，先发出平仓委托，然后在成交回报事件 `OnTrade` 中等再发出开仓委托。

```
else if (State == 4)
{
    #region 反手:如果平仓全部完成，则进入开仓环节

    if (trade.OpenOrClose == EnumOpenClose.开仓)
    {
```

```
}
else
{
    TradeVolume += trade.Volume;
    //如果是平仓，则减少存货
    Volume -= trade.Volume;

    if (Volume <= 0)
    {
        //存货平光了
        TdVolume = 0;
        YdVolume = 0;
        Volume = 0;
        TradeVolume = 0;

        double ordPrice = LastPrice;

        if (trade.Direction == EnumBuySell.买入)
        {
            ordPrice = GetBuyPrice(LastTick);
        }
        else if (trade.Direction == EnumBuySell.卖出)
        {
            ordPrice = GetSellPrice(LastTick);
        }
        //立刻同方向开仓
        Print("反手的平仓全部完成，进入开仓环节");
        LimitOrder(Qty, ordPrice, trade.Direction, EnumOpenClose.开仓);

        SwitchState(5);
    }
}
#endregion
}
```

4 计算策略盈亏

期货公司的柜台系统无法统计策略的盈亏（尤其在一个账户中运行多个策略的情况），这时需要策略自己记录维护盈亏。我们在这个策略中采取的办法是在日内波动策略模板 **BaseStrategy** 中用过的成交统计法，即记录策略所有的成交，然后根据成交统计。

9.39 布林带穿越

策略原理:

布林带喇叭口张开时, 跟随价格的趋势交易。

交易逻辑:

当布林带的张口幅度超过设定阈值 2%时:

- 1 如果价格上穿中轨, 建仓一份多头头寸, 止损位置为布林带的中轨;
- 2 如果价格下穿中轨, 建仓一份空头头寸, 止损位置为布林带的中轨。

开发思路

代码描述

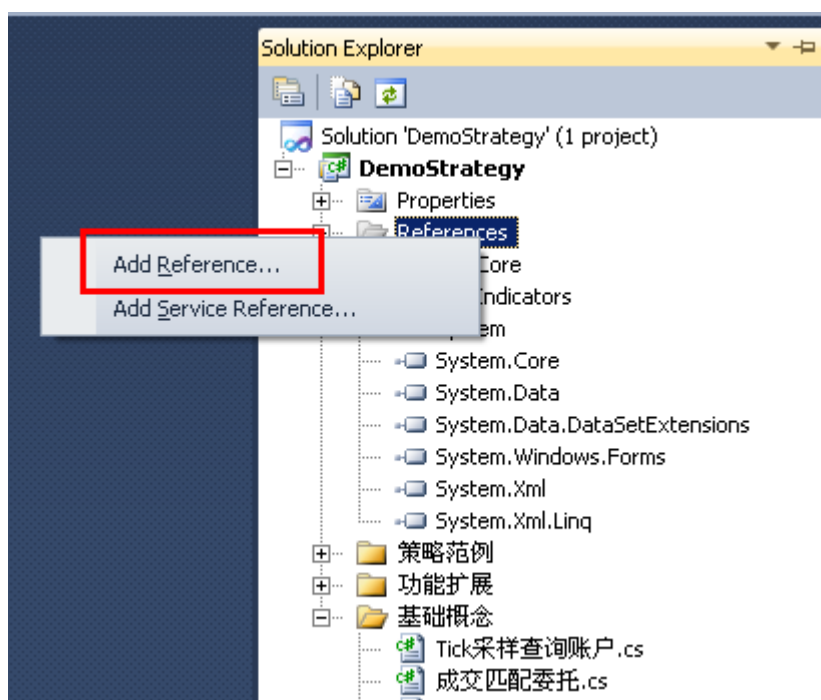
10 功能扩展

基于 C# 语言，策略开发者可以对 MQ 进行功能扩展，开发出专业强大的程序化交易工具。MQ 灵活扩展的特性给了策略开发人员更大的发挥空间，获得更好的程序化交易体验。本章介绍一些策略中常用的功能扩展。

10.1 引用 DLL

策略如果需要使用别的 .NET 开发的 dll，只需要在 Visual Studio 中添加对该 dll 的引用，在开发策略是就可以使用该 dll 提供的方法。

注意：如果策略中使用了这类第三方 dll 文件，需要将该 dll 放到 MQ 程序的根目录下，策略在运行时才能正确找到 dll 文件并成功运行。



用户也可以自己开发组件，然后以 dll 的方式将组件“安装”到 MQ 中，以供自己的策略调用。

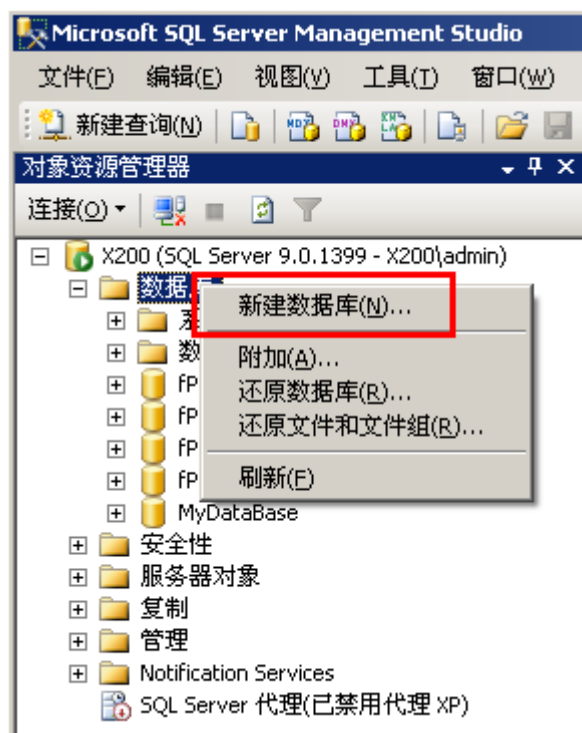
10.2 数据库

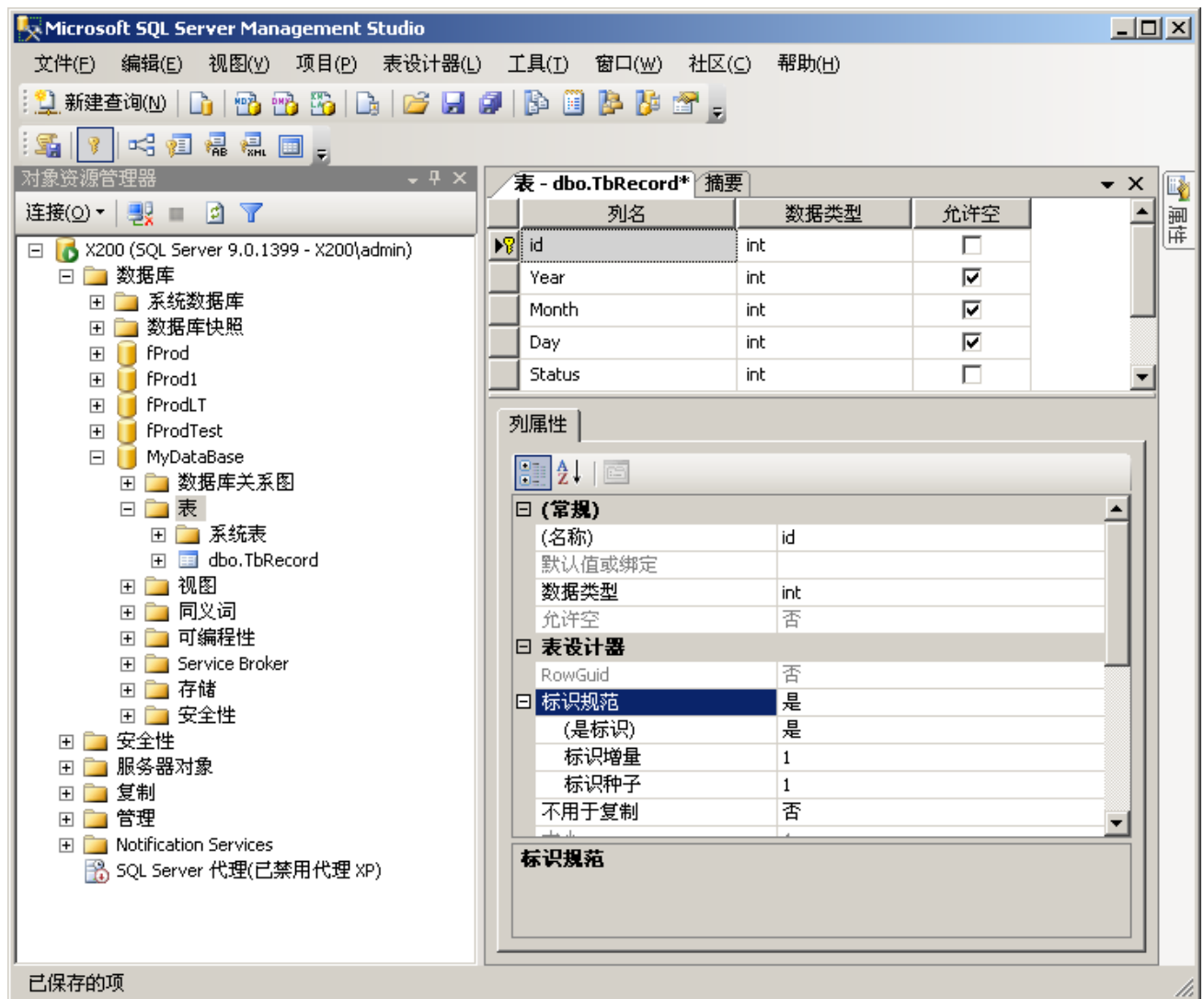
作为结构化的数据存储工具，数据库既可以保存数据，也可以读取数据。在很多策略中，结合数据库技术，可以获得更强大的数据处理和分析能力。通过对 **SQLServer** 数据库的操作进行了一定程度的封装。使用数据库之前，需要添加引用 **Ats.Util.dll** 文件。

在使用数据库的策略范例中，我们练习对数据库进行增删查改的操作。

操作	SQL 语句范例	SqlHelper 用法
增加	Select * from TbRecord;	SqlHelper. ExecuteNonQuery
删除	Delete from TbRecord where id=1;	SqlHelper. ExecuteNonQuery
查询	Select * from TbRecord;	SqlHelper.ExecuteDataset
修改	Update TbRecord set status=1 where id=1;	SqlHelper. ExecuteNonQuery

假设已经安装好 **SQLServer**，新建一个数据库 **MyDataBase**。，然后新建一个表 **TbRecord**，共有 4 个列：id、Year、Monty、Day、Status。





※源代码 \

```
using System.Data;
using Ats.Core;
using Ats.Util;

namespace MagicQuantTest
{
    publicclass 连SQLServer : Strategy
    {

        [Parameter(Display = "ConnStr", Description = "数据库连接字符串", Category = "参数")]
        public string ConnStr = @"Data Source=127.0.0.1;Initial Catalog=DataBaseName;User ID=sa;Password=5";

        public override void Init()
        {
            //连接SQLServer数据库
            string sql = "delete from table1;";
            SqlHelper.ExecuteDataset(ConnStr, CommandType.Text, sql);

            //查询数据库
            sql = "select * from table1 ";
        }
    }
}
```

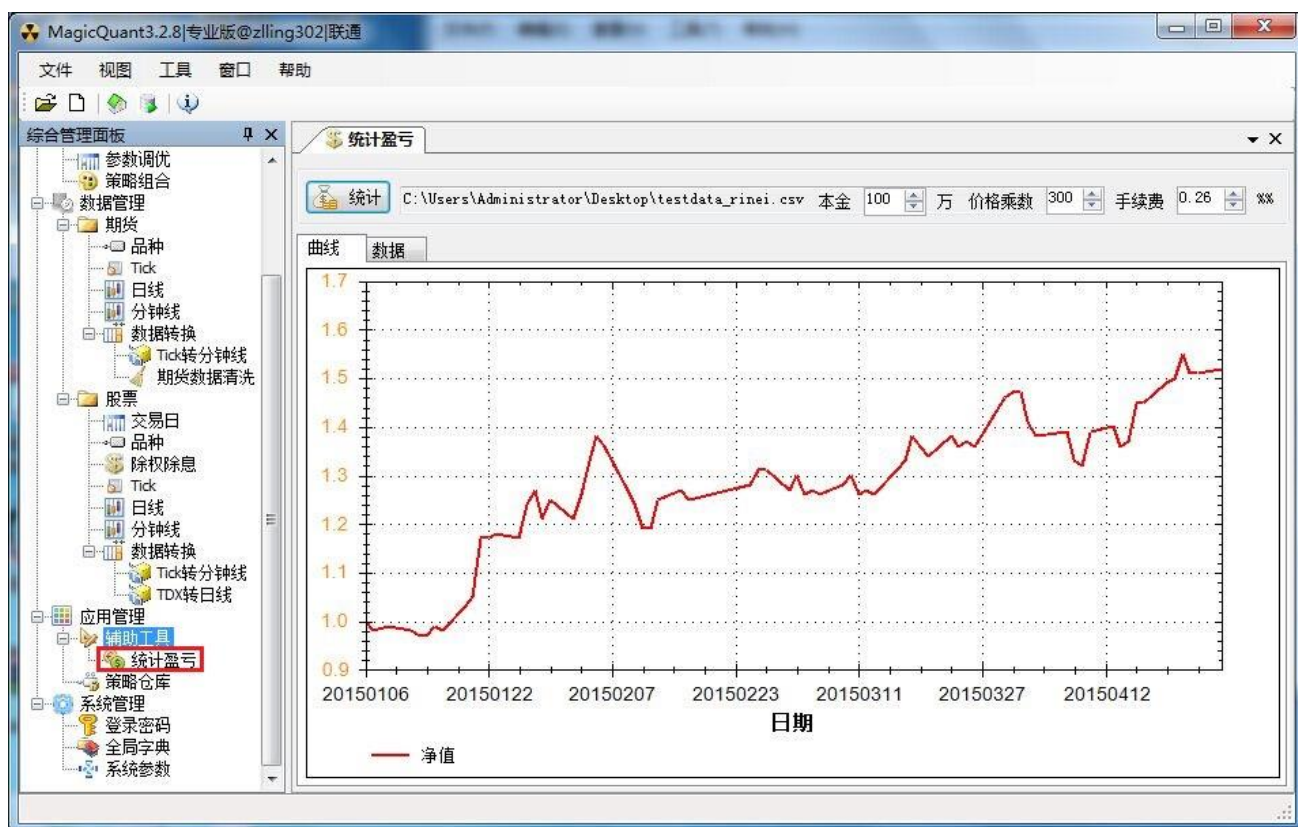
```

        DataSet ds = SqlHelper.ExecuteDataset(ConnStr, CommandType.Text, sql);
    }
}

```

10.3 辅助工具-统计盈亏

在【综合管理面板】上选择【辅助工具】，【统计盈亏】，可以对历史交易数据进行统计和分析。



点击【统计后】可以直接载入历史交易的 CSV 统计文件后可以查看历史数据分析

点击【数据】后可以查看逐笔交易的详细信息：

曲线	数据							
	交易日	买量	卖量	买金额	卖金额	手续费	毛利润	净利润
1	2015-01-06	0	0	0	0	0	0	0
1	2015-01-07	4	4	4385760	4365000	227.5	-20760	-20987.5
2	2015-01-09	3	3	3214920	3223860	167.4	8940	8772.6
3	2015-01-12	1	1	1068360	1058280	55.3	-10080	-10135.3
4	2015-01-13	1	1	1063620	1053420	55	-10200	-10255
5	2015-01-14	3	3	3170940	3168840	164.8	-2100	-2264.8
6	2015-01-15	3	3	3230160	3254580	168.6	24420	24251.4
7	2015-01-16	2	2	2218740	2208600	115.1	-10140	-10255.1
8	2015-01-19	1	1	994860	1044720	53	49860	49807
9	2015-01-20	8	8	8093400	8113200	421.4	19800	19378.6
10	2015-01-21	4	4	4092900	4219200	216.1	126300	126083.9
11	2015-01-22	14	14	14956920	14956320	777.7	-600	-1377.7
12	2015-01-23	21	21	22659300	22669500	1178.5	10200	9021.5
13	2015-01-26	18	18	19467120	19459140	1012.1	-7980	-8992.1
14	2015-01-27	21	21	22502520	22566900	1171.8	64380	63208.2
15	2015-01-28	29	29	30911340	30942420	1608.2	31080	29471.8
16	2015-01-29	32	32	33538920	33485160	1742.6	-53760	-55502.6
17	2015-01-30	38	38	39622920	39664860	2061.5	41940	39878.5
18	2015-02-02	26	26	26422500	26379720	1372.9	-42780	-44152.9
19	2015-02-03	22	22	22473540	22524300	1169.9	50760	49590.1
20	2015-02-04	32	32	33199140	33265500	1728.1	66360	64631.9

CSV 文件格式为:

成交时间	买卖	开平	成交数量	成交价格	品种代码	成交编号	委托编号
2015-1-7 9:23	卖出	开仓	1	3643.8	IF1501	26678	127448
2015-1-7 9:44	买入	平仓	1	3663	IF1501	94478	497693
2015-1-7 9:50	买入	开仓	1	3668.2	IF1501	115018	602734

10.4 在 MQ 中使用 Python

在 MagicQuant 中可以调用 Python 脚本的计算功能。MQ 把 Python 脚本看做一个数值计算的黑盒子。策略输入参数给 Python 这个黑盒子，然后调用 Python 的计算方法，最后 Python 输出结果给策略。

- (1) 安装 Python 的运行环境和需要的算法包，编辑好 Python 脚本。
- (2) 策略调用方法 `AddPythonScript` 加载 Python 脚本。
- (3) 策略调用方法 `SetPythonPara` 设置 Python 脚本的参数。
- (4) 策略调用方法 `ExecutePython` 编译执行 Python 脚本。
- (5) 策略调用方法 `GetPythonOutput` 提取 Python 的计算函数。
- (6) 运行计算函数。
- (7) 策略调用方法 `GetPythonOutput` 提取计算结果。

具体代码参考《综合测试代码》里的《使用 Python 脚本》和下面的操作流程：

➤ 编辑 Python 脚本

在外部编译器重编译好 Python 脚本，保存在文件中。

注意：如果 Python 脚本编译不通过，则无法在 MQ 策略中执行。

一个简单的 Python 脚本如下所示：

#输出结果

Price=0

def Split(tick):

 global Price

 Price=tick.LastPrice+Jump*0.2

➤ 添加脚本

在策略初始化的时候，首先从文件中把脚本的文本读取出来。然后用方法 `AddPythonScript(脚本文本)` 将 Python 脚本添加到策略内存中去。如果有多个脚本需要添加，则需要指定脚本的 ID。

```
public override void Init()
{
    // 初始化时加载，编译脚本
```

```
if (File.Exists(脚本路径))
{
    string StrText = File.ReadAllText(脚本路径);
    if (StrText != "")
    {
        //将脚本代码添加到MagicQuant策略中
        PythonID = AddPythonScript(StrText);
        Print("创建Python脚本, ID=" + PythonID);
    }
}
```

➤ 传入参数

如果脚本中需要参数, 可以用方法**SetPythonPara(参数名, 参数值, 脚本ID)**将参数传给脚本。
传入参数以后, 需要执行一次脚本**ExecutePython()**, 以使得参数生效。

```
//传入参数Jump
SetPythonPara("Jump", Jump);
//修改参数后需要执行一次脚本
CompileOK = ExecutePython();
if (CompileOK)
{
    Print("脚本[" + PythonID + "]编译成功");
}
else
{
    Print("脚本[" + PythonID + "]编译失败");
}
```

➤ 调用Python方法

在数据更新时, 需要调用**Python**内部的计算方法。首先**Func**定义**Python**内部的方法接口。
然后用**GetPythonOutput**将**Python**里对应的计算方法取出来。

```
//定义接口方法
Func<object, object> MyFun;
//将脚本中的方法输出给MyFun
bool getFunOK = GetPythonOutput<Func<object, object>>("Split", out MyFun);
```

获取到方法后, 执行方法**MuFun(输入变量)**。输入变量可以是**.NET**的类, 也可以是数值。
计算成功后, 再用**GetPythonOutput**把计算结果取出来。

```
if (getVOK)
{
    double Price = Convert.ToDouble(MyValue);
    Print("脚本计算的结果="+Price.ToString() );
}

else
{
    Print("获取脚本计算结果失败");
}
```

完整的调用Python脚本的策略代码:

```
using System;
using Ats.Core;
using Ats.Indicators;
using System.IO;

namespace MagicQuantTest
{
    public class 使用Python脚本 : Strategy
    {
        [Parameter(Display = "脚本路径", Description = "", Category = "脚本")]
        string 脚本路径 = "c:\\MQPython.py";

        [Parameter(Display = "Jump", Description = "", Category = "加跳")]
        int Jump = 1;
        string PythonID = "";
        bool CompileOK = false;
        //这个脚本的功能是在Tick的最新价基础上加跳
        //算法输入参数
        //Jump=2
        //输出结果
        //Price=0
        //def Split(tick):
        //    global Price
        //    global Jump
        //    Price=tick.LastPrice+Jump*0.2
        public override void Init()
        {
            #region 初始化时加载, 编译脚本
            if (File.Exists(脚本路径))
            {
                string StrText = File.ReadAllText(脚本路径);
                if (StrText != "")
                {
                    //将脚本代码添加到MagicQuant策略中
                    PythonID = AddPythonScript(StrText);
                    Print("创建Python脚本, ID=" + PythonID);
                    //传入参数Jump
                    SetPythonPara("Jump", Jump);
                    //修改参数后需要执行一次脚本
                    CompileOK = ExecutePython();
                    if (CompileOK)
                    {
                        Print("脚本[" + PythonID + "]编译成功");
                    }

                    else
                    {
                        Print("脚本[" + PythonID + "]编译失败");
                    }
                }
            }
            else
            {
                Print("脚本内容为空");
            }
        }
    }
}
```

```
        else
        {
            Print("脚本文件[" + 脚本路径 + "]不存在");
        }
        #endregion
    }

    public override void OnTick(Tick tick)
    {
        //计算策略信号时调用脚本
        //把Tick对象作为变量传给Python脚本
        if (CompileOK)
        {
            //定义接口方法
            Func<object, object> MyFun;
            //将脚本中的方法输出给MyFun
            bool getFunOK = GetPythonOutput<Func<object, object>>("Split", out MyFun);
            if (getFunOK)
            {
                //执行Python脚本中的方法, 把变量Tick传给方法
                MyFun(tick);
                //把计算结果取出来
                object MyValue;
                bool getVOK = GetPythonOutput<object>("Price", out MyValue);
                if (getVOK)
                {
                    double Price = Convert.ToDouble(MyValue);
                    Print("脚本计算的结果="+Price.ToString() );
                }
                else
                {
                    Print("获取脚本计算结果失败");
                }
            }
            else
            {
                Print("获取脚本方法失败");
            }
        }
    }
}
```

10.5 在 MQ 中使用 Matlab

在 MQ 中调用 Matlab 的基本原理是，将 Matlab 的函数代码编译成 .NET 能识别的动态链接库文件（.dll），然后在策略中引用并调用 Matlab 的函数。

注意如果使用 32 位 MQ，就必须用 32 位的 Matlab；如果使用 64 位 MQ，就必须用 64 位的 Matlab。具体操作步骤如下：

1 在 Matlab 中编写函数

在 Matlab 的 .m 文件中输入函数代码，例如：

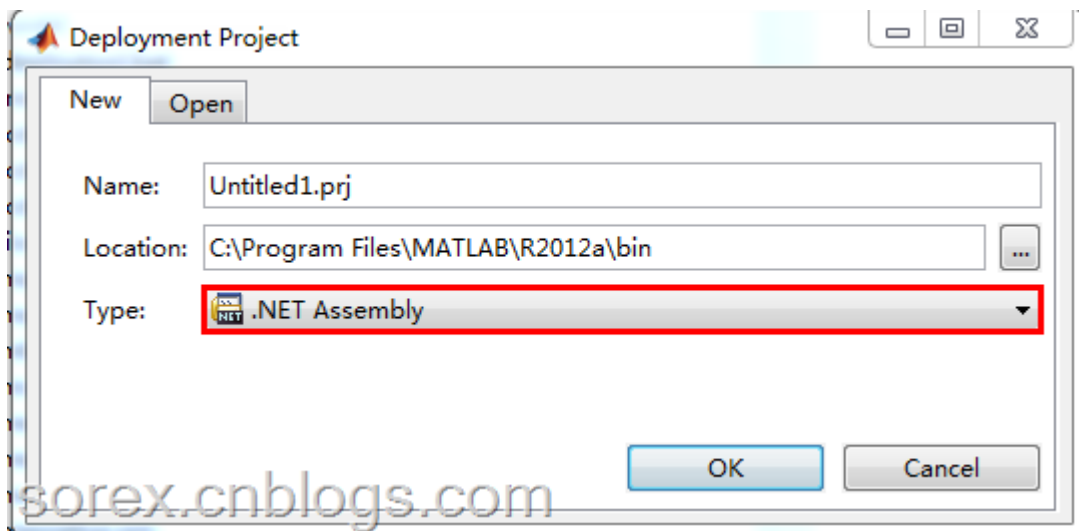
```
function result = TestFun(x, y)
result = power(x, y);
end
```

然后保存

2 发布 Matlab 工程

点击 Matlab 的 File→New→ Deployment Project

注意类型 Type 要选择 .Net Assembly



3 编译代码

添加一个类 Class，假设命名为 MatX 类，然后将上面我们新建的文件放进去，将 Matlab 的 .m 文件加进去。

然后点击编译按钮，会在 Matlab 的项目目录里看到生成的 dll 文件

4 引用 dll 文件

在编辑策略代码的 **VS** 里面，在解决方案管理器的引用节点上右键，添加引用，然后选中前面生成的 **dll**

在策略代码里，就可以用下面的方法调用 **Matlab** 的函数：

```
TestClass tc1 = new TestClass();  
var z3 = tc1.TestFun((MwNumericArray)x.ToArray(), (MwNumericArray)y.ToArray()).ToArray();
```

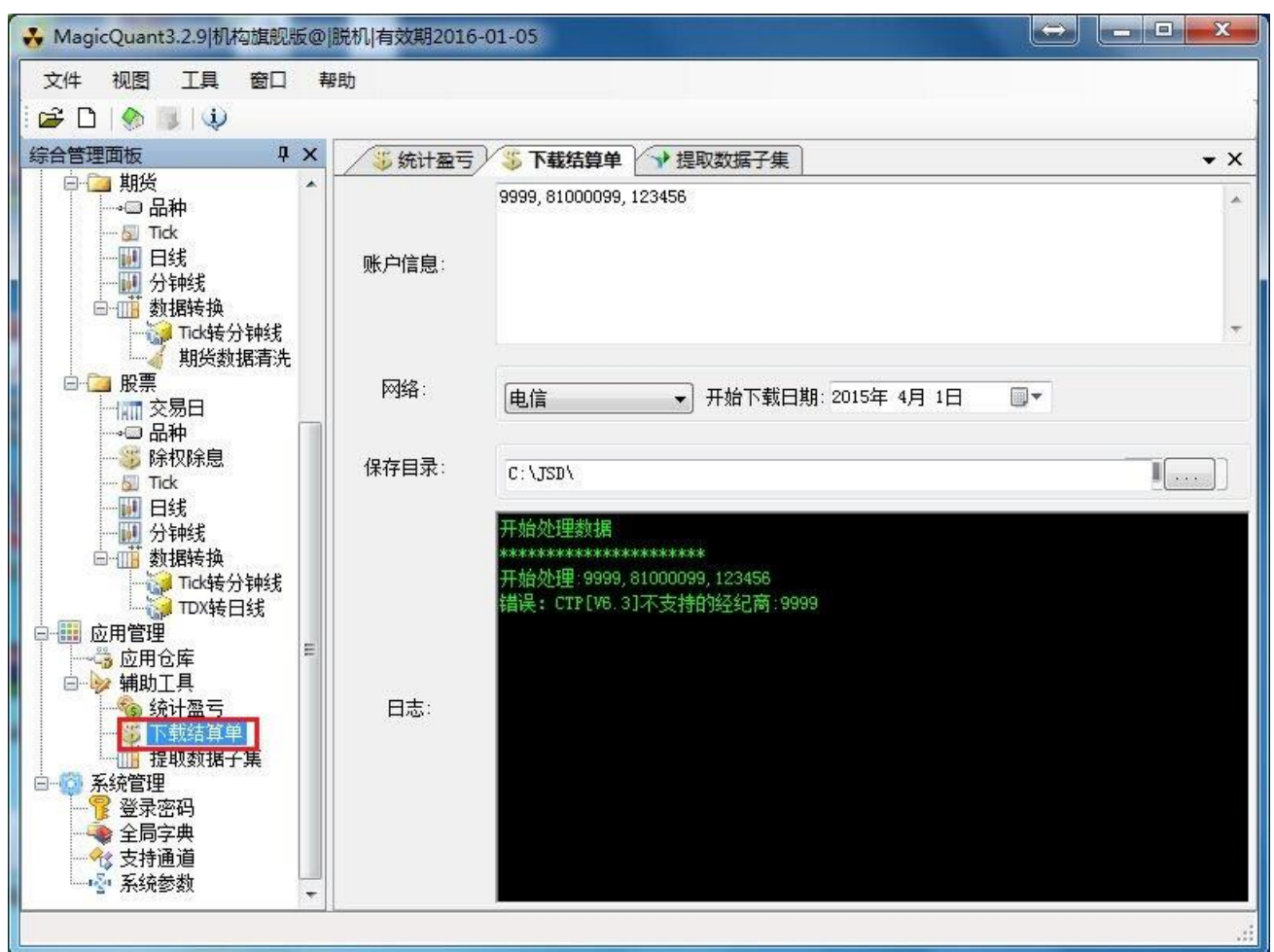
5 使用 **dll** 文件

策略编译好后，要把 **dll** 文件拷贝到运行的 **MQ** 的根目录里面，同时在机器上安装 **Matlab** 运行需要的环境，再启动策略。

10.6 在 MQ 中使用 R

10.7 在 MQ 中下载结算单

在【综合管理面板】上选择【辅助工具】，【下载结算单】，可以下载账户交易数据记录进行分析。



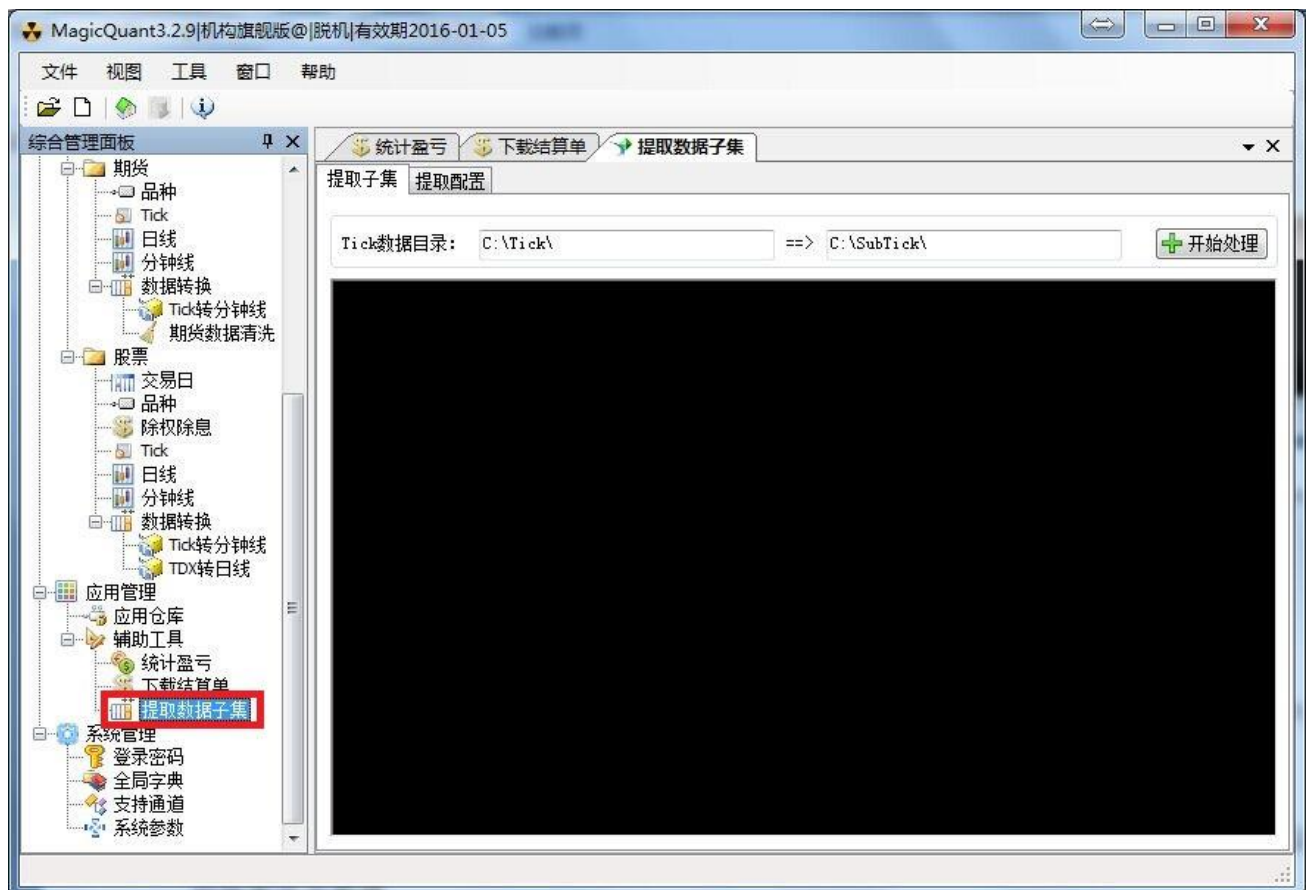
在账户信息配置中填写好 **brokerid,交易账号,交易密码(以逗号进行分隔)**

以及设置好日期和输出目录后可以通过点击下载按钮后从 CTP 服务器端下载指定日期中的结算单！

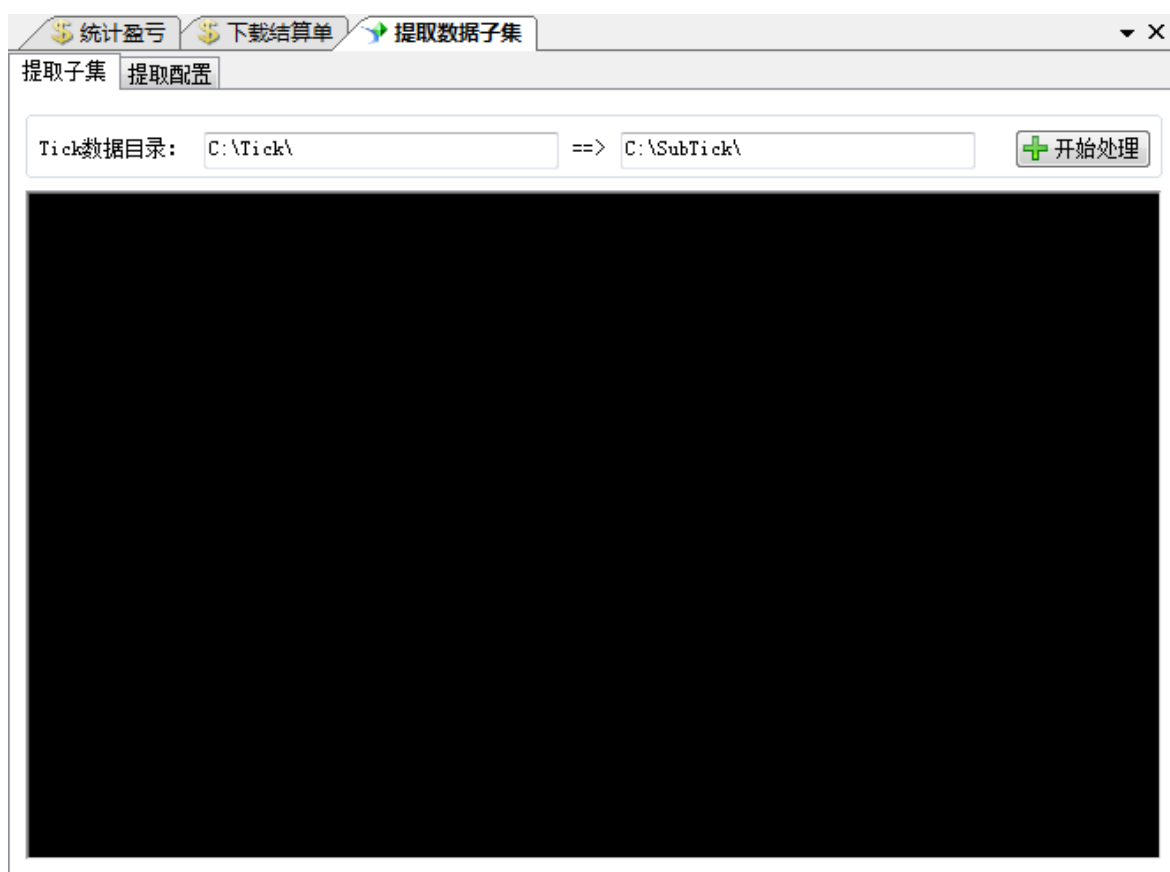
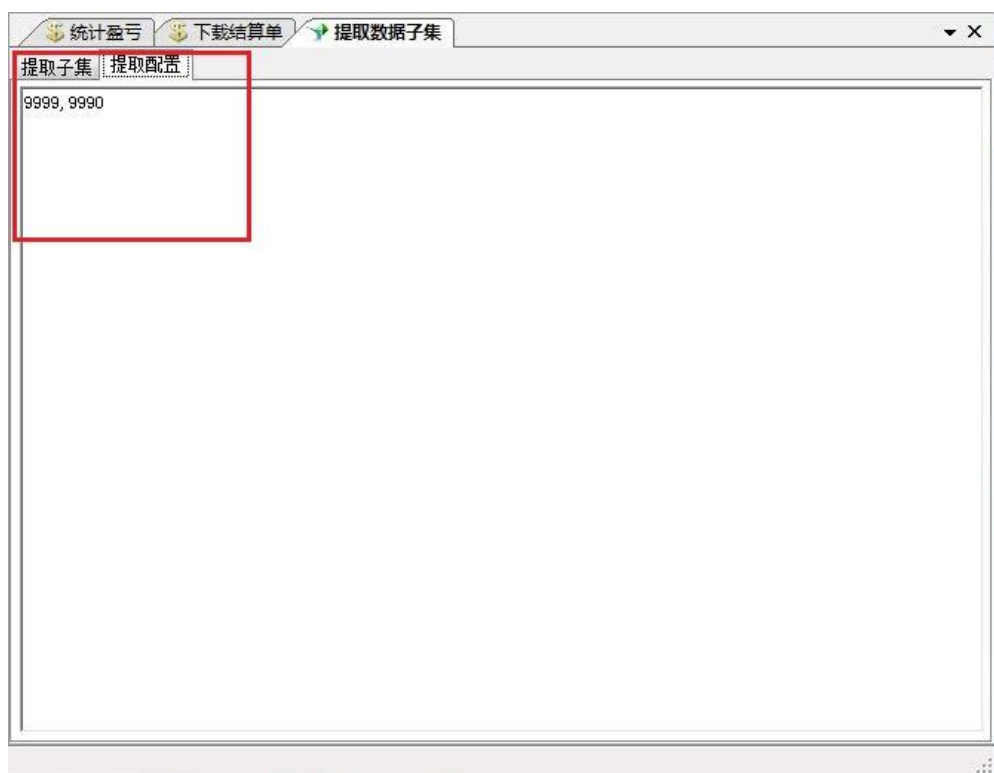
10.8 提取数据子集

当本地历史数据品种过多的并且只需要用到其中某几个品种 tick 的时候可以用数据子集功能将需要的数据单独提取出来！提取方式：

在【综合管理面板】上选择【辅助工具】，【提取数据子集】，可以提取指定标的品种的所有历史数据



在界面中选择【提取配置】设置好需要提取的数据代码以及配置好历史数据目录和数据输出目录后单击开始处理即可将需要的数据子集提取出来！



10.9 异常处理

当策略中遇到异常时，如果不对异常进行处理，那么 MQ 就会抛出异常，并中断策略运行。为了避免策略中断，更柔性地处理异常，需要用 **try-catch** 机制进行异常处理。异常处理的语法结构：

```
try
{
    //可能出错的代码块
}
catch (Exception ex)
{
    //对错误信息进行处理分析
}
```

※源代码 \功能扩展\异常处理.cs

10.10 实盘人工确认

在实盘交易时，为了确保策略的参数、状态处于正常范围内，可以通过人工确认窗口，由交易监控人员确定参数，人工判断系统状态正常后才让策略继续运行。

※源代码 \功能扩展\人工确认.cs

10.11 限制策略使用

策略的安全性是 MQ 考虑的重点问题。MQ 本身对策略进行了高强度的加密，因此使用反编译工具是无法看到策略中的内容的。用户希望进一步加强其安全性，就可以在策略中加入限制代码，使得该策略只能运行在特定的账户和特定的时间范围内，从而最大程度上保护策略开发者的利益。

※源代码 \功能扩展\限制策略使用.cs

10.12 盘中调整参数

我们基于 MQ 的全局字典功能，让策略每次都从字典中获取变量的值，这样一旦用户通过 MQ 的字典界面修改了该值，策略可以立即根据用户的指示调整参数，从而实现盘中调整参数的功能。

※源代码 \功能扩展\盘中调整参数.cs

10.13 灾难恢复

如果在实盘中由于停电或者用户需要，策略会被中途强行停止，这样内存中的数据就全部丢失了。当再次启动策略时，我们需要能够将策略内存中的变量恢复到灾难发生前的状态。这种场景下就需要用到灾难恢复的技术。最常用的方法是将需要恢复的变量保存到硬盘中，然后每次启动策略时，从硬盘中读取这些数据，恢复到策略的内存中。

下面的例子就演示了基于 MQ 的字典功能，将一个 **double** 型的变量 K 保存到 MQ 的全局字典 **Dict** 中，如果程序异常中断，则可以通过字典将数据恢复出来。

※源代码 \功能扩展\灾难恢复.cs

11 常见问题

11.1 MQ 如何收费

如果您所在的期货公司开通了 MagicQuant 代收费服务，MagicQuant 机构版服务费为交易所手续费的 5%。

目前开通代收费业务的期货公司有：

期货公司	联系方式【QQ】
东证期货	8724846
宝城期货	3254028
美尔雅期货	457667466
银河期货	104254922
上海中期	409171034
国信期货	19202238
安信期货	1914740676

如果您所在的期货公司没有开通 MagicQuant 代收费服务，可以通过结算单直接交费。

MagicQuant 机构旗舰版实盘客户按月提供授权，每月初客户通过以下步骤来向我们申请授权：

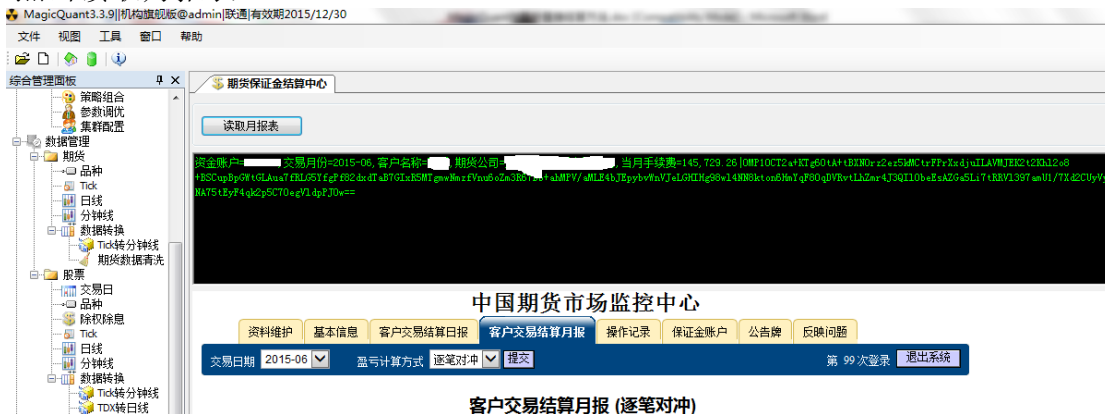
1，登陆 MQ，点击保证金中心



- 2, 登陆保证金中心
- 3, 点击客户交易结算月报



- 4, 点击读取月报表



- 5, 点击复制到剪贴板
- 6, 粘贴到邮件正文, 并发送邮件到 services@magicquant.com
- 7, 在淘宝充值, 链接
<http://item.taobao.com/item.htm?spm=a1z10.1-c.w1031-2470818356.1.oPbGhp&id=14357063373>, 充的金额为当月手续费的 5%。
- 8, 客服发送邮件更新授权

11.2 MQ 支持哪些期货公司

MQ 支持全部部署 CTP 或者飞马柜台系统的期货公司。

11.3 怎样自己配置期货公司

自己编译 Config 目录下的 ctpbrokerx.xml 文件即可。

11.4 GetBarSeries 和 OnBar

这二者毫无关系。用户首先要了解数据和驱动的含义和区别。

通过 [GetBarSeries](#) 方法获得的 K 线，是数据；

[OnBar](#) 是通过 MQ 界面配置的驱动模式配置的，是驱动。

数据是数据，驱动是驱动，这两件东西完全是两码事。

任何策略，不管是 Tick 驱动，还是 K 线驱动，都可以调用 [GetBarSeries](#) 方法获得 K 线。这个获得的 K 线在策略运行中会自动随着数据的到来而更新。

而 [OnBar](#) 事件，只有当策略被配置成 K 线驱动时，才会以配置的 K 线周期驱动运行。

例如：某个策略，初始化了 2 个 K 线：K1 和 K2，周期分别是 3 分钟，5 分钟；同时该策略配置用 1 分钟 K 线驱动，那么这个策略会每 1 分钟触发一次 [OnBar](#) 事件。

11.5 怎样进一步提高速度

为了获得更快的运行速度，可以采用下列方法

工具--->高级配置

1 Tick 数据不落盘

方法：工具-->系统配置-->高级参数，把 Tick 数据落盘全面的复选框的钩去掉；

这样 MQ 收到的数据就不会保存在本地，减少磁盘读写，从而提高速度；

2 关掉日志

把根目录下的 [log4net.xml](#) 文件删除

这样 MQ 就不会记录系统底层日志（代价就是不便于排查系统问题），减少磁盘读写，从而提高速度

11.6 不合法的登录

如果底层日志出现信息“**不合法的登录**”，说明期货账户的账户或者密码不正确。

11.7 不能输出类库

在编译策略时，不能直接按 **F5**，而是要用 **VisualStudio** 菜单栏上的“生成”→重新生成解决方案功能。

11.8 刚开通 CTP 却收不到事件

开通 **CTP** 的第一天不能交易，因为这一天 **CTP** 柜台系统是没有事件回报的。

11.9 怎么避免开盘前的委托

集合竞价期间报入的委托有时候是无效的，可以在代码中加一段保护，例如：

```
if (TickNow < tOpen)
{
    //过滤集合竞价期间的Tick, tOpen是开盘时间
    return;
}
```

11.10 怎么获取 Tick 数据复盘

MQ 提供一部分免费的历史数据供复盘使用：

更多的 Tick 数据可以联系管理员购买。

11.11 如何使用股票程序化交易功能

11.12 如何使用 UDP 行情

第一步：将 UDP 行情源服务器 IP 地址和端口配置到期货商列表中

期货商列表文件位于 config 目录下，文件名为 ctpbrokers.xml

例如申万期货的 UDP 行情 IP 为 180.1.11.81 和 180.1.11.82，行情端口为 18213，交易端口为 41205

那么应该配置为：

```
<broker id="88888" name="申万期货" defgroup="上海">
    <group name="上海">
        <dx>
            <quote>
                <server address="tcp://180.1.111.81:41213">
```

```
</server>

        <server address="tcp://180.1.111.82:41213">

</server>

        </quote>

        <trade>

            <server address="tcp://180.1.111.81:41205">

                </server>

                <server address="tcp://180.1.111.82:41205">

                    </server>

                </trade>

            </dx>

        </group>

    </broker>
```

第二步：将通道配置中 CTP 的 UDP 选项改成 true

通道配置文件位于 config 目录下，文件名为 chanel.xml

例如：

```
<channel id="CTP" name="CTP">
  <param name="addressfile">config\ctpbrokers.xml</param>
  <param name="timeout">120</param>
  <param name="udp">true</param>
</channel>
```

第三步：可以将 ctpbrokers.xml 和 chanel.xml 设成只读，避免自动更新该配置文件。

11.13 策略显示已经下单但平台并没有实际下单

检查一下登陆期货实盘账户连接数总数是否超过 6 个，包括使用该账户的 MQ 工程总数

和快期连接数，如果超过 6 个的话，会出现这个问题，需要关闭多余的连接。

11.14 股指期货当天平今委托的成交回报为什么不是平今？

中金所的股指期货合约不区别平仓和平今（CTP 和飞马等柜台系统会智能进行转换处理），因此如果要平掉股指期货的持仓，可以用平仓，也可以用平今，同时中金所目前返回的成交回报里都是平仓。

11.15 集合已修改；可能无法执行枚举操作？

原因是.NET 中不能对一个正在遍历的集合进行移除操作。例如：有一个集合 `List`

```
for( int i = 0 ; i < List.Count; i++ )
{
    List.Removeat(0); //移除集合中某一个元素
}
```

这样做就会报错，因为程序在遍历集合时，对集合又进行了操作，因此报错。一个避免这个问题的办法是：遍历集合的同时，把需要的操作对象在内存中记录好。然后再针对这个操作记录进行操作。参考代码如下：

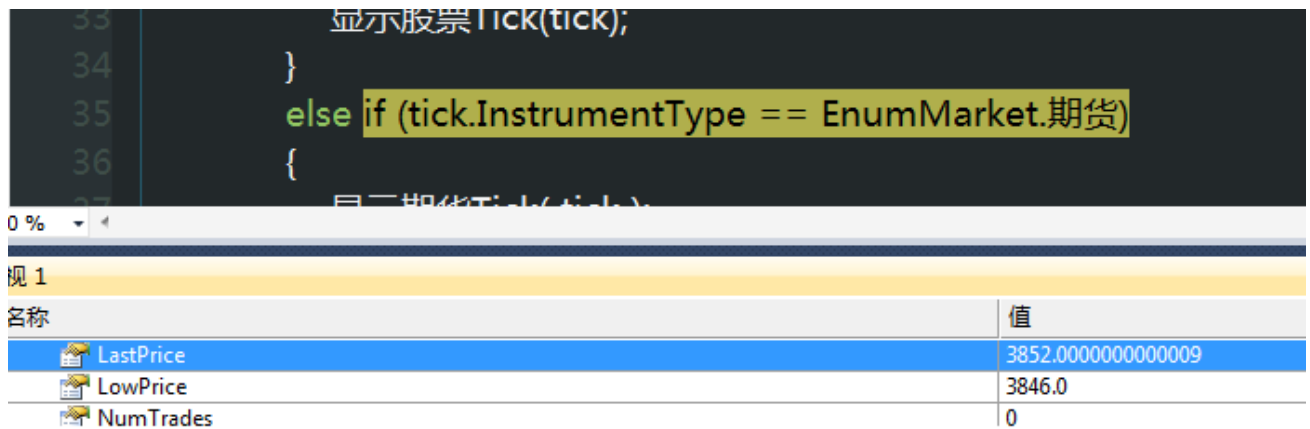
```
List<string> X = new List<string>();
For( int i = 0 ; i < List.Count; i++ )
{
    X.Add( List[i] ); //记录（而不直接操作）集合中某一个元素
}

//然后统一操作

For( int i = 0 ; i < X.Count; i++ )
{
    List.Remove(X[i]);
}
```

11.16 为什么 3852>3852?

在.NET 里，double 数值类型是一个近似类型，存在精度问题。



某个品种的最新价是 3852，但是在.NET 编译器里显示为 3852.00000000000009，因此当两个 double 值进行比较时会出现精度问题。

有几种解决办法：

- (1) 使用 Decimal 数据类型。
- (2) 对于品种价格的比较，使用多少“跳”作为单位

以股指期货为例，其跳 PriceTick=0.2

```
int PriceTickN = (int)tick.LastPrice/0.2;
```

这样 PriceTickN 作为一个整型变量，就可以进行精确的比较。

- (3) 在比较时使用一个保护的 MinDouble

例如：

```
double MinDouble = 0.000001;
```

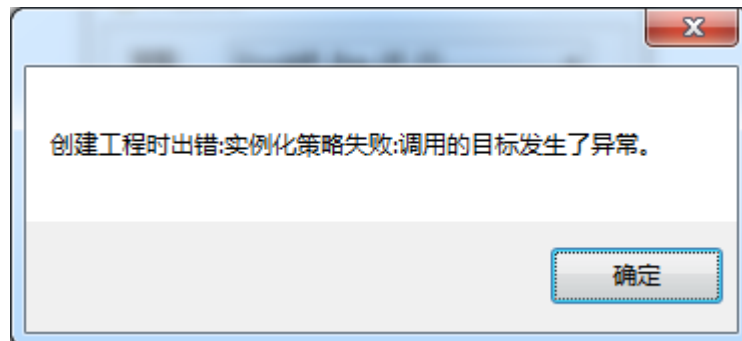
```
if( A > B )
```

可以写成：

```
if(A>B+MinDouble)
```

11.17 创建工程时出错，实例化策略失败？

创建工程时，MQ 提示出错：实例化策略失败，如下图所示。



MQ 创建工程时，需要把策略实例化，这里需要量化工程师检查自己的策略的变量有没有异常的值。例如下面的变量就会报错，因为 2 月没有 31 日：

```
Public DateTime tExpire = newDateTime(2016, 2, 31);
```