Trading

发明者量化 FMZ QUANT

商品期货量化
交易实战

Commodity futures quantitative trading combat

全新CTA趋势策略实战手册
实战应用案例

通过Python+发明者量化建模，手把手教你
搭建属于自己的量化交易系统

基于真实的数据进行策略解析、应用与回测
循循渐进的案例式教学，普通人也可以玩转量化交易

发明者量化编著
The inventor quantitative product

目录

第 1 章　量化交易基础 1
1.1　什么是量化交易 1
1.1.1　量化交易概述 1
1.1.2　量化交易发展 1
1.1.3　量化交易的特点 2
1.1.4　量化交易都有哪些入门策略？ 3
1.2　为什么选择量化交易 4
1.2.1　量化交易与主观交易的区别 4
1.2.2　量化交易比主观交易更好？ 4
1.2.3　量化交易一定能赚钱吗？ 5
1.2.4　量化交易的风险 5
1.3　量化交易需要准备哪些 5
1.3.1　策略构思 5
1.3.2　建立模型 6
1.3.3　回测调优 6
1.3.4　仿真交易 7
1.3.5　实盘交易 7
1.4　一个完整的策略有哪些要素 7
1.4.1　策略选择 8
1.4.2　买卖什么 8
1.4.3　买卖多少 8
1.4.4　何时买卖 9
1.4.5　如何买卖 9
1.4.6　交易心态 10
1.5　温故知新 10
第 2 章　Python 编程入门 11
2.1　为什么要学习 Python 11
2.1.1　Python 的特点 11
2.1.2　发明者量化支持的 Python 版本 12
2.2　Python 基础语法 12
2.2.1　编码 12
2.2.2　变量命名 12
2.2.3　关键字 13

2.2.4 注释......13
2.2.5 缩进......13
2.2.6 代码块......14
2.2.7 空行......14
2.2.8 导入模块......14
2.3 Python 变量和数据类型......15
2.3.1 变量......15
2.3.2 标准数据类型......15
2.3.3 Number（数字）......15
2.3.4 String（字符串）......16
2.3.5 List（列表）......17
2.3.6 Dictionary（字典）......18
2.3.7 Python 数据类型转换......19
2.4 Python 数据运算......19
2.4.1 算术运算符......20
2.4.2 关系运算符......20
2.4.3 赋值运算符......21
2.4.4 逻辑运算符......22
2.4.5 运算符优先级......23
2.5 Python 数字和字符串......24
2.5.1 数字类型转换......24
2.5.2 内置数学函数......24
2.5.3 访问字符串中的值......25
2.5.4 拼接字符串......25
2.5.5 其他常用函数......26
2.6 Python 列表和字典......26
2.6.1 列表索引......26
2.6.2 列表切片......27
2.6.3 列表修改删除......27
2.6.4 二维列表......28
2.6.5 列表增加元素......28
2.6.6 列表反向排序......29
2.6.7 创建字典......29
2.6.8 访问字典元素......30
2.6.9 字典添加修改元素......30
2.6.10 字典删除元素......30
2.7 Python 条件语句和循环语句......32
2.7.1 条件语句......32

2.7.2 循环语句......35
2.7.3 break 语句......36
2.7.4 continue 语句......37
2.8 Python 日期和时间......37
2.8.1 time 模块......37
2.8.2 什么是时间戳......37
2.8.3 时间戳转换时间......38
2.9 Python 常用内置函数......38
2.9.1 len()函数......38
2.9.2 range()函数......39
2.9.3 split()函数......40
2.9.4 type()函数......40
2.9.5 isinstance()函数......40
2.9.6 取整函数......41
2.10 Python 异常处理......41
2.10.1 语法错误......42
2.10.2 异常错误......42
2.10.3 异常捕获......42
2.11 温故知新......43
第 3 章 认识发明者量化......43
3.1 平台简介......43
3.1.1 平台架构......44
3.1.2 托管者程序......44
3.1.3 平台功能......45
3.1.4 机器人......46
3.1.5 策略库......47
3.1.6 交易终端......47
3.1.7 策略编写界面......48
3.1.8 账户管理......49
3.2 配置交易所和部署托管者......50
3.2.1 配置交易所......50
3.2.2 在 Windows 上部署托管者......52
3.2.3 在 linux 上部署托管者......54
3.2.4 一键租用托管者......55
3.3 创建管理策略和机器人......57
3.3.1 创建策略......57
3.3.2 管理策略......59
3.3.3 创建机器人......61

3.3.4 管理机器人 62
3.4 模拟级和实盘级回测系统 64
3.4.1 模拟级别回测系统 64
3.4.2 底层 K 线周期 65
3.4.3 实盘级别回测系统 65
3.5 全局常量和数据结构 66
3.5.1 exchange 交易所对象 66
3.5.2 exchanges 交易所对象列表 67
3.5.3 Order 结构 68
3.5.4 Position 结构 69
3.5.5 Trade 结构 70
3.5.6 Ticker 结构 70
3.5.7 Record 结构 71
3.5.8 Depth 结构 71
3.5.10 Account 结构 72
3.5.11 策略参数 72
3.6 获取 Tick、深度、历史 K 线数据 73
3.6.1 exchange.GetTicker() 73
3.6.2 exchange.GetDepth() 74
3.6.3 exchange.GetRecords() 74
3.6.4 商品期货策略框架 74
3.7 获取和取消订单、获取当前挂单 76
3.7.1 exchange.SetContractType(ContractType) 76
3.7.2 exchange.SetDirection(Direction) 77
3.7.3 exchange.Buy(Price, Amount) 77
3.7.4 exchange.Sell(Price, Amount) 78
3.7.5 exchange.CancelOrder(orderId) 79
3.7.6 exchange.GetOrders() 79
3.7.7 exchange.GetOrder(orderId) 80
3.8 IO 扩展函数 81
3.8.1 IO 函数切换行情模式 81
3.8.2 IO 函数判断与期货公司前置机连接状态 82
3.8.3 IO 函数获取交易所所有合约 82
3.8.4 exchange.IO("api", …) 82
3.8.5 exchange.IO("wait") 83
3.9 账户 API 获取账户和持仓信息 84
3.9.1 exchange.GetAccount() 84
3.9.2 exchange.GetPosition() 85

3.10 常用日志信息函数......87
3.10.1 Log(…)......87
3.10.2 LogProfit(Profit)......88
3.10.3 LogStatus(Msg)......89
3.10.4 Chart(…)......90
3.10.5 LogReset()......92
3.10.6 EnableLog(IsEnable)......93
3.11 常用内置函数......93
3.11.1 Sleep(Millisecond)......93
3.11.2 GetCommand()......94
3.11.3 IsVirtual()......95
3.11.4 _G(K, V)......95
3.11.5 _D(Timestamp, Fmt)......97
3.11.6 _N(Num, Precision)......97
3.11.7 _C(function, args…)......98
3.11.8 _Cross(Arr1, Arr2)......99
3.12 常用指标函数以及图表绘制......100
3.12.1 内置的 TA 指标库......101
3.12.1 绘制图表......103
3.13 策略参数及交互......107
3.13.1 策略参数......107
3.13.2 策略交互......109
3.14 内置模板类库及经典策略架构......110
3.14.1 模板类库......111
3.14.2 经典策略架构......111
3.15 温故知新......112
第 4 章 CTA 之趋势跟踪策略......113
4.1 什么是 CTA 策略......113
4.1.1 CTA 策略的分类......114
4.1.2 趋势策略......114
4.1.3 反转策略......114
4.1.4 量化 CTA 策略......114
4.2 经典 MACD 交易策略......115
4.2.1 MACD 简介......115
4.2.2 MACD 原理......116
4.2.3 MACD 计算方法......116
4.2.4 MACD 使用方法......116
4.2.5 MACD 的有效性......117

4.2.6 MACD 策略逻辑 117
4.2.7 MACD 策略编写 117
4.2.8 完整策略代码 119
4.3 利用平均趋向指数辅助 MACD 策略 120
4.3.1 什么是平均趋向指数 120
4.3.2 ADX 的计算方式 121
4.3.3 策略逻辑 121
4.3.4 策略编写 122
4.3.5 完整策略代码 123
4.4 自适应动态双均线策略 124
4.4.1 传统均线弊端 125
4.4.2 考夫曼均线原理 125
4.4.3 考夫曼均线计算 125
4.4.4 策略逻辑 127
4.4.5 策略编写 127
4.4.6 完整策略代码 129
4.5 日内高低点突破策略 130
4.5.1 什么是日内交易 130
4.5.2 策略逻辑 131
4.5.3 策略编写 131
4.5.4 完整策略代码 134
4.6 增强版唐奇安通道策略 135
4.6.1 唐奇安通道简介 136
4.6.2 原始策略逻辑 136
4.6.3 改进后的策略逻辑 136
4.6.4 策略编写 137
4.6.5 完整策略代码 139
4.7 hans123 日内突破策略 140
4.7.1 策略原理 140
4.7.2 策略编写 141
4.7.3 完整策略代码 143
4.8 菲阿里四价策略 144
4.8.1 菲阿里简介 145
4.8.2 策略逻辑 145
4.8.3 策略编写 146
4.8.4 完整策略代码 149
4.9 阿隆指标 AROON 策略 150
4.9.1 阿隆指标简介 150

4.9.2 阿隆指标的计算方法......151
4.9.3 如何使用阿隆指标......151
4.9.4 基于阿隆指标构建交易策略......152
4.9.5 完整策略......154
4.10 简易波动 EMV 策略......156
4.10.1 EMV 计算公式......156
4.10.2 EMV 用法......156
4.10.3 策略实现......157
4.10.4 策略回测......159
4.10.5 完整策略......160
4.11 动态阶梯突破策略......161
4.11.1 什么是突破策略......162
4.11.2 突破策略理论......162
4.11.3 策略逻辑......162
4.11.4 策略编写......164
4.11.5 完整策略......166
4.12 Dual Thrust 日内交易策略......168
4.12.1 Dual Thrust 简介......168
4.12.2 Dual Thrust 上下轨......168
4.12.3 策略逻辑......169
4.12.4 策略编写......169
4.12.5 策略回测......171
4.12.6 完整策略......172
4.13 经典恒温器策略......173
4.13.1 策略简介......173
4.13.2 市场波动指数......173
4.13.3 策略逻辑......174
4.13.4 策略编写......174
4.13.5 策略回测......177
4.13.6 完整策略代码......178
4.14 R-breaker 策略......180
4.14.1 策略原理......180
4.14.2 计算方式......180
4.14.3 策略逻辑......181
4.14.4 策略编写......181
4.14.5 完整策略......183
4.15 温故知新......185
第 5 章 CTA 之回归策略......185

5.1 布林带跨期套利策略......185
5.1.1 策略原理......186
5.1.2 策略逻辑......186
5.1.3 策略编写......187
5.1.4 策略回测......190
5.2 期现套利图表......190
5.2.1 什么是套利......191
5.2.2 期现套利方法......191
5.2.3 期现套利方法......192
5.2.4 获取数据......192
5.2.5 现货和基差图表......193
5.2.6 图表展示......196
5.3 乖离率 BIAS 策略......196
5.3.1 乖离率简介......196
5.3.2 乖离率的原理......197
5.3.3 乖离率计算公式......197
5.3.4 策略逻辑......198
5.3.5 策略编写......198
5.3.6 策略回测......199
5.3.7 完整策略......200
5.4 温故知新......201
第 6 章 用科学的方法回测策略......202
6.1 使用 Tick 数据让回测更精准......202
6.1.1 回测需要哪些数据......202
6.1.2 基于 Bar 数据的回测......203
6.1.3 基于 Tick 数据的回测......203
6.1.4 盘口数据回测引擎原理......204
6.1.5 如何选择最佳回测方式......204
6.2 回测绩效报告详解......205
6.2.1 回测配置参数......205
6.2.2 年化收益率......206
6.2.3 年化波动率......206
6.2.4 最大回撤比率......207
6.2.5 夏普比率......207
6.3 如何避免回测陷阱......208
6.3.1 未来函数......208
6.3.2 偷价......208
6.3.3 成本冲击......208

6.3.4　幸存者偏差 209
6.3.5　过拟合 209
6.4　递进和交叉回测 210
6.4.1　样本内和样本外回测 210
6.4.2　样本递进回测 211
6.4.3　样本交叉回测 212
6.5　温故知新 213
第 7 章　风险管理与投资组合 213
7.1　认识期货中的风险 214
7.2.1　系统性风险 214
7.2.2　人为主观风险 214
7.2.3　策略风险 214
7.2.4　资金管理的意义 215
7.2.5　资金管理的方法 215
7.2　等价鞅资金管理 216
7.2.1　什么是马丁格尔 216
7.2.2　正向马丁格尔 217
7.2.3　正向马丁格尔代码 217
7.2.4　反向马丁格尔 220
7.2.5　反向马丁格尔代码 220
7.3.6　马丁格尔在期货市场上的应用 223
7.3　反等价鞅资金管理 223
7.3.1　什么是凯利公式 224
7.3.2　凯利公式计算方式 224
7.3.3　用数据验证凯利公式 224
7.3.4　凯利公式在量化交易中的应用 227
7.3.5　凯利公式的局限性 227
7.4　构建投资组合和风险控制 227
7.4.1　投资分散与均衡 227
7.4.2　投资组合分类 228
7.4.3　构建投资组合 228
7.4.4　收益与风险 229
7.5　温故知新 229
第 8 章　交易技巧及交易理念 230
8.1　常用止盈止损方法 230
8.1.1　止损的成本 230
8.1.2　止损的意义 231
8.1.3　如何止损 231

8.1.4　止损的本质......232
8.2　量化交易与基本面数据......233
8.3.1　常用的基本面数据......233
8.3.2　基本面分析铁三角......233
8.3.3　获取基本面数据......234
8.3.4　绘制基本面数据图表......236
8.3　交易中的常用的数理知识......237
8.4.1　VWAP 算法......237
8.4.2　TWAP 算法......238
8.4.3　布朗运动......238
8.4.4　维纳过程......239
8.4.5　伊藤引理......239
8.4.6　马尔科夫过程......240
8.4　建立概率思维，提升交易格局......240
8.5.1　交易来自生活......240
8.5.2　概率思维......241
8.5.3　久赌必赢......241
8.5.4　概率的变化......242
8.5.5　交易中的大数定律......242
8.5　温故知新......242

第 1 章　量化交易基础

作为一种新型的金融投资方法，量化交易利用计算机技术，结合数学建模和统计学分析，从大量的历史数据中提炼出交易策略，通过计算机强大的运算能力实现自动交易，减少了交易者因情绪影响做出的非理性交易。

本章涉及到的知识点有：

- 量化交易概念：量化交易发展历程和基本概念。
- 量化交易特点：量化交易与主观交易优缺点。
- 量化交易要素：量化交易流程和策略要素内容。

1.1　什么是量化交易

量化交易作为交易与计算机结合的产物，正在改变着现代金融市场的格局。如今已经有不少交易者者将目光转向了这一领域。如何最大限度地降低风险并尽可能取得收益？也是许多交易者孜孜以求的目标。

1.1.1　量化交易概述

很多人一听到“量化交易”就会觉得高端大气、一夜暴富。人工智能时代，伴随着深度学习、大数据、云计算等先进技术的兴起，更是赋予它神秘的色彩。似乎只要运用量化交易，就能构建出“完美无缺”的交易策略。

在一定程度上，量化交易已经被神话了。量化交易其实就是借助计算机，并利用统计学、数学等方法，通过科学的投资体系，从中找到一套正期望的交易信号系统。这个信号系统会告诉我们应该在什么时间以什么价格进行买卖。

1.1.2　量化交易发展

追本溯源，早在 19 世纪，法国股票经纪人助理朱尔斯•雷格纳特就采用量化方法来分析价格数据变化，从中发现市场价格涨跌规律，并提出了股票价格变化的现代理论，随后出版了《概率计算和股票交易哲学》一书，在书中详细阐述了自己发现的市场涨跌规律（正态分布）：“价格的偏差与时间的平方根成正比”，最后以理性量化的投资方法获取交易上的成功。

现如今，在互联网+大数据+云计算+人工智能的时代背景下，量化交易也得到了快速发展。曾经的全球金融腹地伦敦金丝雀码头，早已变成了 IT 公司集散地。世界顶尖投行，也都在培养自己的量化团队，试图跻身到“得策略者得天下”的金融大战之中，这些开发交易模型的 IT 团队也被称为 Quant Team。

反观国内，无论是硬件设备还是投研实力，都还在发展初期阶段。但已经有越来越多的机构和专业投资者意识到量化交易的好处，并参与到这一领域，特别是在商品期货市场逐步规范、市场有序开放，量化交易更具有广阔的成长空间。

1.1.3 量化交易的特点

量化交易脱胎自主观交易，主观交易每次下单前需要人为判断行情，这在实际交易中很难保持一致，尤其是当行情波动剧烈，账户盈亏时时刻刻左右交易者的心智，使交易者很难做出正确判断。而主观交易的缺点正是量化交易的优点，具体如下：

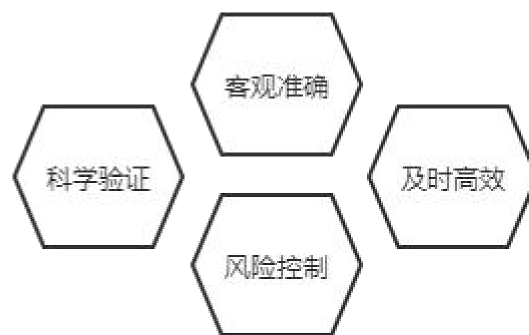

图 1.1　量化交易特点

- 科学验证：当编写完交易策略后，如果用模拟盘测试它的有效性，可能会付出很大的时间成本，如果用实盘测试，更有可能损失真金白银。但量化交易中的回测功能，可以通过大量的历史数据，以科学的方式去检验交易系统。
- 客观准确：在交易中，我们真正的敌人是自己，心态管理说起来容易，做起来难。贪婪、恐惧、侥幸等人性的弱点，在交易市场中会数倍放大，量化交易则可以屏蔽这些弱点，在交易中做出更理性的决策。
- 及时高效：在主观交易中，人的反应速度是无法快过电脑的，并且人的体力和精力也无法 24 小时运行，在机会稍纵即逝的交易市场，量化交易完全可以代替主观交易，寻找更多的交易机会，及时快速地跟踪市场变化。
- 风险控制：量化交易能从历史数据中挖掘价格未来可能重复的规律，这些规律可以转化为较大概率取胜的策略。还可以构建多种不同的投资组合，降低系统性风险，平滑资金曲线。

注意：主观交易并非一无是处，在量化交易中，计算机很难识别千变万化的 K 线形态，比如：双重顶底、头肩定底、V 型反转等等，但主观交易就很容易分辨出来。另外相对来说，主观交易更加细腻，比如对于一些似是而非的交易信号会选择性回避。

1.1.4　量化交易都有哪些入门策略？

开盘突破策略：一般情况下开盘半小时往往能决定一天的走势，该策略以开盘后半小时，价格是阳线还是阴线，作为判断日内趋势走向的标准。如果是阳线就开仓买入，如果是阴线就开仓卖出，收盘前平掉仓位。这是一个非常简单的交易策略。

唐奇安通道策略：该策略可以说是量化交易的雏形，其规则是：如果当前价格高于前 N 根 K 线内最高价就买入，如果当前价格低于前 N 根 K 线内最低价就卖出。著名的海龟交易法则用的就是修正版的唐奇安通道策略。

图 1.2　唐奇安通道策略

跨期套利策略：该策略是套利交易中最普遍的一种，根据同一个交易品种，不同交割月份合约的价格为基础，如果两者价格出现了较大的价差幅度，就可以同时买卖不同时期的期货合约，进行跨期套利。

假设主力合约与次主力合约的价差长期维持在-50~50 左右。如果某一天价差达到 70，预计价差会重新回归到-50~50 之内。那么就可以卖出主力合约，同时买入次主力合约，来做空这个价差。反之亦然。

图 1.3　量化交易策略分类

1.2　为什么选择量化交易

很多人在探讨量化交易时会以复杂的策略编程为切入点，这无形中给量化交易披上了一层神秘的面纱。本节将以通俗易懂的语言，为量化交易做一个简单“素描”，即便是毫无基础的小白也能轻松理解。

1.2.1　量化交易与主观交易的区别

主观交易更重视人为的分析和盘感，即使出现了买卖信号，也会选择性地下单交易，宁可错过行情，也不愿做错。人的感觉是复杂多变的且不可靠的，大多数交易者一旦发生连续亏损，往往就转而用另一种方法。交易的随机性较强，容易被浮动盈亏困扰，导致难以稳定盈利。

量化交易通过对交易的理解，制定一致性的买卖策略。在交易中，对所有的走势都一视同仁，开仓平仓全部系统化处理，宁可做错，也不愿错过。它还具有完整的评价体系，通过历史数据回测，确定策略更适合哪一类的行情和品种，并搭配多种策略和品种实现盈利。

简而言之，主观交易是量化交易的基础，量化交易是主观交易的提炼。主观交易更像是练武，最后能成功与否，天赋占大多数，有十年不悟的，也有一朝悟道的。量化交易更像是健身，只要刻苦努力，就算没有天赋，也能练出一身肌肉。

1.2.2　量化交易比主观交易更好？

一个成功的主观交易者，从某种角度上说，也是一个量化交易者。因为一个成功的主观交易者，必然有一套自己行之有效的规则和方法，也就是交易系统。成功的主观交易必须建立在交易规则和交易纪律之上，而交易规则其实就是主观交易中的量化部分。

相反，成功的量化交易者，也都脱胎自主观交易，因为量化交易策略的开发，其实就是主观交易方法的具体实现。如果一个对市场的理解和认知，从一开始就是错误的，那么开发出来的交易策略，长期以来也是难以获利的。

所以从长期稳定盈利的角度讲，决定一个交易者最终能否成功，关键因素是交易理念，而不在于是主观交易还是量化交易。量化交易表面上看似高大上，其盈利的本质与主观交易没有本质的区别，它们就像是一件事物的两面性既对立又统一。但是不可否认，从交易工具上来说，量化交易确实有很多优势。

- ❑ 复盘更快：想要检验一个交易策略，就需要计算大量的历史数据，量化交易几分钟之内就能计算出结果。这个速度要比主观交易快许多倍。
- ❑ 更加科学：评价一个策略是否优秀，依靠的是数据（比如：夏普比率、最大回撤率、年化收益），优秀的策略和交易理念往往具有可证伪性。
- ❑ 更多机会：国内商品期货有几十个交易品种，主观交易不可能同时盯盘，但是量化交易可以全市场实时盯盘，不错过任何交易机会，增加交易效率。

1.2.3　量化交易一定能赚钱吗？

当然能，但长期坚持下来却是一件很难的事。赚钱与否并不取决于量化交易本身，它只是一个工具，量化交易只是把交易思想用程序化、规则化、数量化实现出来，程序代替的只是执行力。难的是长期稳定地赚钱，因为市场是动态博弈，交易思路也要跟着市场转变。

1.2.4　量化交易的风险

量化交易也有风险，为什么呢？因为量化交易是在历史数据中去挖掘规律，形成交易策略。但是金融市场是一个生态体系，其规律和人性是一个相互作用的动态过程，归根到底还是人的市场。市场的规律会被人性所影响，而人性中间的贪婪、恐惧都会随着市场的变化而变化。所以市场上很少有一成不变的规律，再厉害的交易策略也很难应对这种突如其来的规律变化。

通过上面的解释可以看到，量化交易不是一种独特的交易方法，它只是一种交易工具，帮助我们分析交易逻辑，完善交易策略。无论是价值派、技术派，无论做的是股票、债券、商品还是期权，其实都可以量化。相比于靠个人经验做决策的交易者，量化交易者手中的武器就是市场证据和理性。

注意：量化交易的风险大部分来源于市场的风险，所以先学会交易，再学会量化。

1.3　量化交易需要准备哪些

一个完整的量化交易生命周期，不仅仅只是交易策略本身。它至少由六个环节构成，包括：策略构思、建立模型、回测调优、仿真交易、实盘交易、策略监控等。

图 1.4　量化交易流程

1.3.1　策略构思

首先，做量化交易必须先回到交易市场，要在市场中多观察价格，理解市场波动的规律，并尝试推断每一个交易逻辑，最后总结出交易策略。这里并没有捷径，阅读经典的投资书籍或许有帮助，或者不断地坚持做交易，在失败中总结经验。

对于初学者来说，开发交易策略最好的方式就是模仿。直接利用现成的技术分析指标构建策略逻辑，写入买卖规则，这样就可以得到一个简单的交易策略。例如单均线的策略

逻辑是：如果价格高于最近 10 天的平均价格就买入，如果价格低于最近 10 天的平均价格就卖出。

当然，随着市场经验的积累，形成自己的交易方式后，策略逻辑的选择会越来越多样化，再进阶到更加系统的量化交易。如果能做一个有量化思维的交易者，无论是在股票还是期货市场上，这都是一件值得庆幸的事。

1.3.2 建立模型

其次，你需要掌握一个量化交易工具，用来编写交易策略，实现你的交易想法。市面上的常用软件都可以。但是如果你想成为一名高端的量化交易者，就需要学会一门计算机语言，这里推荐使用 Python 编程语言，因为它是科学计算的权威语言，并且提供各种开源的分析包，文件处理，网络，数据库等。

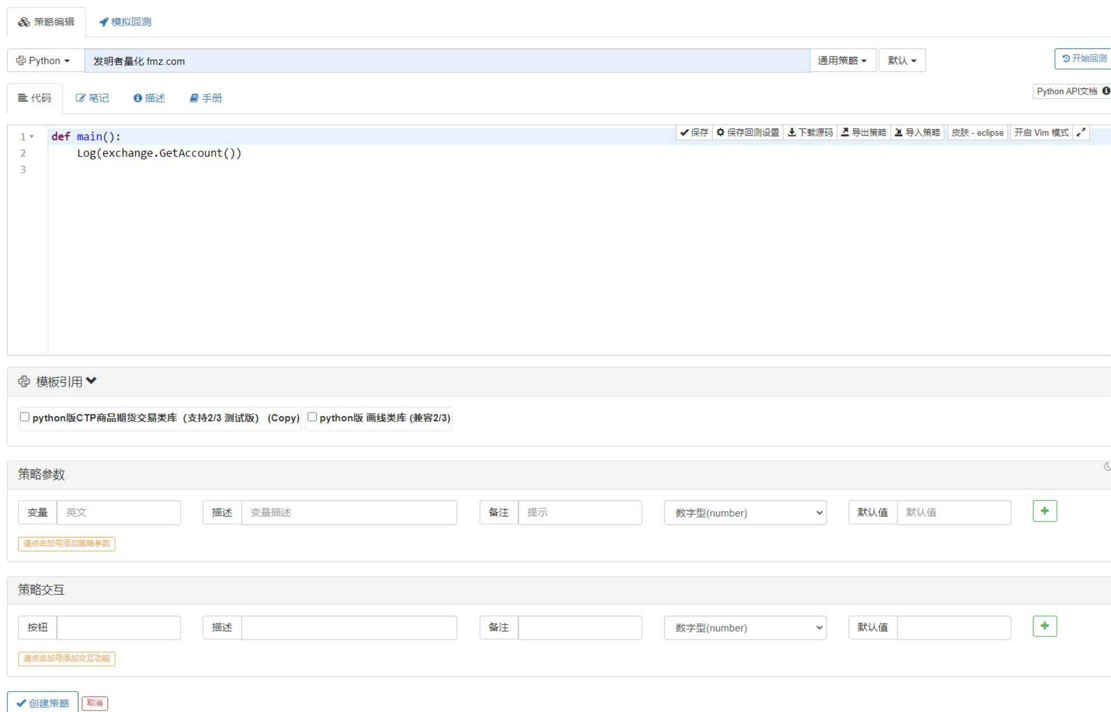

图 1.5　发明者量化策略编辑界面

1.3.3 回测调优

当编写完策略后，下一步就是对策略进行回测，以及参数的筛选和优化。可以利用不同的参数对策略进行回测，观察该策略的夏普比率、最大回撤、年化收益等。通过对策略的不断调试和修改，最终得到一个完善的量化交易策略。

比如，可以把 2010 年~2015 年的历史数据作为样本内数据，2016~2020 年的历史数据作为样本外数据。先用样本内数据优化出几组表现好的参数，再用这些参数对样本外数据

进行回测。

一般情况下，样本外的回测结果没有样本内的回测结果好，但是如果样本外与样本内的结果大相径庭，那么这个策略可能是无效的，就要观察分析，判断策略失效的原因。

如果发现策略失效由于是样本外数据，某几次极端行情导致的大幅亏损，那么就可以增加一个固定止损条件来规避这种风险；如果发现策略失效是由于交易次数过多，那么就可以将交易逻辑收紧，降低交易频率。

如果一开始交易逻辑本身就是错误的，再怎么修改也很难得到一个赚钱的策略，这个时候就需要重新审视自己的策略思路了。另外，在参数优化中，可用的参数组越多越好，说明策略的适用性广泛。

注意：核心的策略参数越少越好，如果参数过多很容易造成数据拟合。在回测时，交易次数太少的策略其回测结果可能是幸存者偏差。如果回测的结果是一个超级赚钱的资金曲线，很多情况下是策略逻辑写错了。

1.3.4　仿真交易

当交易逻辑正确，样本内外回测都赚钱时，先不要急着在真实账户上交易。尤其对于初学者来说，一定要先用仿真账户运行至少 3 个月，如果是中低频隔夜策略，则需要更长的仿真交易时间。在未来一段完全未知的仿真行情中，观察策略在仿真交易中表现，仔细核对回测信号与仿真交易信号是否吻合，下单时的价格与成交时的价格是否有偏差，如果表现与预期相符合，那么说明策略有效。

1.3.5　实盘交易

通过一段时间仿真交易检验之后，就可以将策略放入实战中进行交易了。不过在量化交易的过程中也要保持警惕，防范极端行情。在实盘交易中，策略的期望一般都要打折扣的，很难达到回测时的状态。

1.4 一个完整的策略有哪些要素

一个完整的策略，其实就是交易者给自己定的各种规则，它包括了交易的各个方面，并且不给交易者留下一点点主观想象的余地，每个买卖决定，策略都会给出答案。它至少包含策略选择、品种选择、资金管理、下单交易、极端行情应对、交易心态等等。

图 1.6　策略要素

1.4.1 策略选择

从专业的角度讲，主流的交易策略可以分为趋势交易、配对交易、一揽子交易、事件驱动、高频交易、期权策略等等，当然，策略的分类方式不是固定的。如下图：

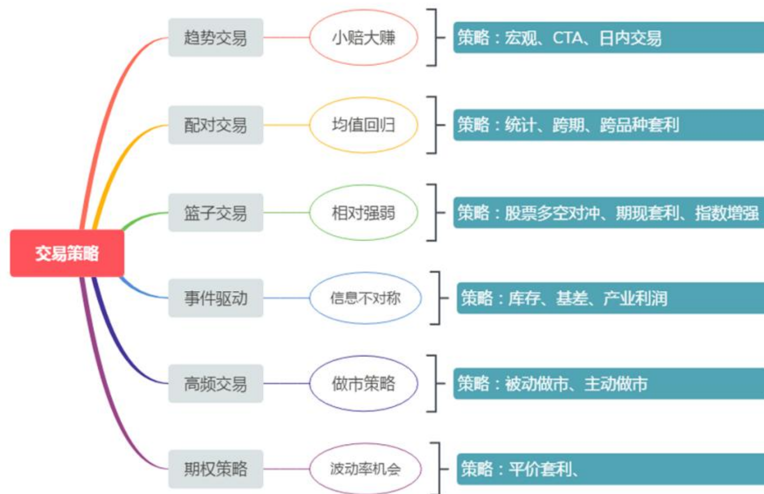

图 1.7　量化交易策略分类

对于刚入门的量化交易初学者来说，不必管这么多名词概念，一步一步从最简单的开始。趋势交易是一个不错的选择，其特点是策略逻辑简单，用有限的亏损换取无限的利润，长期来看是正期望策略。

1.4.2 买卖什么

做过交易的人应该知道，每个品种都有各自的性格。有些品种性格很“火爆”，流动性好、波动率高；有些品种性格很“温顺”，常年都在一定区间内震荡，波动率低。相比之下通常工业品比农产品波动率高。

在选择交易品种时，需要考虑品种的波动率情况，波动率高的品种，往往很容易走出一波不错的趋势行情。如果是趋势跟踪策略，尽量选择工业品，从品种属性上来讲，工业品往往比农产品波动率要大。

不同的策略适应不同的品种，选择合适交易品种，对期货交易这项大工程来说尤为关键。总的来说，没有绝对好的品种也没有绝对不好的品种。根据投资风格的不同，以及风险承受力的不同，需要针对自己的标准进行相应的调整。

1.4.3 买卖多少

交易是赔钱容易赚钱难的行当，当账户资金亏损 50%后，挽回损失则需要 100%的盈利。

就算赚很多次 100%，但只需要赔一次 100%就全部亏光。所以成熟的交易策略应该包含资金管理。

事实上，很多以传统技术指标构建的交易策略，最大回撤率会很大，甚至超过 50%。但一个风险很大的策略完全不能用吗？显然不是，最大回撤率完全可以通过资金管理控制，也就是买卖多少。

如果把仓位降低一半，那么风险也会降低一半，如果把仓位再降低一半，那么风险就会降低更多，这就是一个简单的资金管理方法。但是一味的降低仓位也不是较好的办法，因为降低仓位也就意味着降低了利润，如何取舍就要看交易者对风险的承受能力。

1.4.4 何时买卖

一个好的买点，是成功的一半，它能够迅速摆脱成本区。但是价格不是上涨就是下跌，从大数定律的角度讲，好的买点很难超过 50%，一味的追求胜率反而是舍本求末。笔者认为开仓不是决定最终盈利的核心，对于趋势策略来说，交易的核心是开仓之后，如何尽可能优化处理持仓，达到赢冲输缩的目的。

不管是短线策略，还是长线策略，比的不是看谁持仓时间长，而是风险收益比。换言之，影响策略绩效的最终结果是如何出场，出场方法又可以分为两种：止损出场和止盈出场，这两个部分也是关乎交易策略成败的分水岭。

1.4.5 如何买卖

如果说何时买卖是艺术，那么如何买卖就是技巧。在价格瞬息万变的环境中，酌情使用订单类型和下单方式，可以增加订单成交率，也可以降低滑点，减少交易成本。交易技巧通常需要考虑下列几种情况：

1、委托下单类型和方式：

委托下单的类型和方式有许多种，比如可以用：排队限价单、对手价、最新价、超价、涨停价、跌停价、买一价、买二价、卖一价、卖二价，或者先用排队价，再用超价，分批报单，或者把大单拆成一个个小单，或者干脆直接把单子全部报出去。

2、撤单

如果下单没有成交，就要考虑是继续等待还是撤单，继续等待意味着可能错失行情。如果撤单就要考虑是否继续追单。

3、追单

如果撤单后继续追单是按最新价去追，还是对手价，还是涨跌停价，如果追单仍未成交是否继续追单。当价格与最初的信号相差甚远时，是不限成本追单，还是放弃这个信号。

4、涨跌停价

有时候当下单信号出现，刚好是涨跌停价格。那么是否在涨跌停价挂单排队成交，如果没有成交怎么办，尤其是在持仓与行情反向的时候，如何对冲补救。

5、集合竞价

开盘集合竞价时，哪个报价最多就采用哪个价格开盘，盘面是不显示价格的，只能根据自己的预判进行申报，这里面充满不确定性，要不要参与，以及怎么参与。

6、夜盘

有些商品期货品种夜盘是从 21:00 至次日 02:30，人的精力是有限的，这段时间做不做，人工做还是让电脑来做。

7、重大节日

重大节日的超长假期之前，仓位需不需要保留。如果保留的话如何控制风险。如果节后价格反向跳空后如何处理。

8、极端行情和突发事件

价格瞬间涨跌停、连续涨跌停、乌龙指事件、黑天鹅行情等价格踩踏事件发生时，或者突然断电、断网、电脑故障、软件宕机、银期转账暂停、自然灾害等，出现时如何应对。

1.4.6 交易心态

市场会无形中放大交易者的情绪，影响交易的负面情绪有很多中，其中贪婪、恐惧和侥幸是交易中常见的三种负面情绪。因此交易者需要一个强大的交易心理体系，在不同阶段对上述三种情绪加以控制甚至利用。

注意：交易不仅考察技术基本功，还考察交易者心态，可以说人性的弱点在交易过程中都会被展现无遗和放大。只有不断学习和总结经验教训，不断历练，才能克服人性的思维共性和心理弱点。

没有完美的策略，也没有更好的策略，只有更适合自己性格的策略。结合自身的性格和资金情况一起去衡量该策略是否适合自己，如果适合自己的话，要充分评估自己坚持下去的可能性有多大，最坏的结果要事先规划好，如果最惨的一面你都想好了，那么执行下去的可能性就相对较大。

1.5 温故知新

学完本章内容，读者需要回答：

1．量化交易与主观交易的优缺点是什么？

2．量化交易的流程是什么？

3．期货交易有哪些订单类型？

在下一章中，读者会了解到：

1．Python 基础语法

2．Python 数据类型和条件循环语句

3．Python 常用的内置函数

第 2 章　Python 编程入门

Python 是一个面向对象的脚本语言，凭借极其简洁高效的语言特性，以及数据分析方面的巨大优势，在金融领域得到了广泛的应用。本章内容通过对 Python 语言学习，将其作为策略开发工具，为期货量化交易提供助力。

本章涉及到的知识点有：

- ❑ 基础语法：了解 Python 编程基本概念
- ❑ 数据类型：学习常用的数据类型，及基本的数据操作
- ❑ 数据运算：学习常用的数据运算和关系运算
- ❑ 条件语句和循环语句：掌握 Python 代码结构和逻辑
- ❑ 常用内置函数：掌握 Python 的函数用法

2.1　为什么要学习 Python

量化交易离不开数据分析，而 Python 有很多像 talib、pandas，NumPy 和 matplotlib 这些以数据分析和处理为主的第三方库，使得 Python 成为量化交易策略开发的首选编程语言。从数据获取到策略回测再到实盘交易，Python 已经覆盖了整个量化交易应用链。

2.1.1　Python 的特点

完整的量化交易流程可以分为这些步骤：获取数据、分析计算数据、处理数据、下单交易等。在数据分析方面，Python 既精于计算又能保持较好的性能，特别是在时间序列分析数据（K 线就是时间序列数据）处理，Python 有更加简洁高效的优势。

另外，比起其他编程语言，Python 的语法更加简单容易，不需要大量的计算机系统理论知识，学习曲线比较平缓，即使是非专业的初学者也可以轻松掌握。有意思的是 Python 代码与英语区别不大，具有极高的可读性。

Python 在量化交易领域是一门比较全面而且平衡的编程语言，既可以满足量化交易策略程序运行时的性能，又能轻松处理各种复杂的数学运算、建模分析、统计分析、机器学习等数据处理任务。并且有众多的工具库（包）支持，非常方便实现量化交易策略开发过程中的各种需求。

并且市面上的很多量化交易平台，基本上都支持 Python 编程语言，使得各个量化交易平台所编写出的策略很容易学习、研究、迁移、二次开发。

在量化交易领域，Python 特点可以归纳为：

- ❑ 语法简单，不需要考虑计算机底层细节问题，初学者更容易入门。
- ❑ 生态丰富，大量成熟的第三方库，带来的是无与伦比的便利。
- ❑ 应用广泛，许多量化交易平台都支持 Python 语言，方便学习研究。

- 跨平台、多线程、数据库等方面都有很好的支持。
- 扩展性强，代码通俗易懂，易于维护。
- 学习资料十分丰富，有众多活跃的社区可以进行讨论、学习、研究。

2.1.2　发明者量化支持的 Python 版本

发明者量化交易平台支持 Python 各个版本，如果同时安装了 Python 2 和 Python 3，可以在策略中编写：#!python3 或者#!python2 即可设置当前使用的 Python 版本。由于 Python 官方宣布，2020 年 1 月 1 日，停止 Python2 的更新，所以本书代码以 Python3 为主。

注意：在发明者量化交易平台使用 Python 语言开发策略，就如同使用原生 Python 一样，没有任何区别。编写完 Python 策略后，回测策略或者实际机器人运行策略，如果不在代码中指定 Python 版本，则默认为 Python3 执行策略代码。

2.2　Python 基础语法

Python 语言与 C 和 Java 语言有很多相似之处，但又比这 2 个语言更为简洁。Python 的变量无需声明，可以直接给变量赋值。并且代码块强制以 4 个空格缩进，来区分代码之间的层次。

2.2.1　编码

Python 可以在代码文件开头设置编码，如果不设置则默认为：UTF-8 编码。除非特殊需要，一般不用设置，使用默认 UTF-8 编码即可。你也可以设置为：cp-1252 字符集。

```
# -*- coding: cp-1252 -*-
```

2.2.2　变量命名

顾名思义变量就是一个可以变化的量，它就像一个盒子，里面可以存放各种东西。在编写 Python 代码时，对于声明的变量，在变量名称命名时需要注意，以下是 Python 变量命名规则：

1．变量名是区分大小写的。
2．变量名只能由字母、数字、下划线组成，且不能以数字开头。
3．变量名不能包含空格。
4．Python 的关键字和函数名不能作为变量名。
5．避免使用小写字母 l 和大写字母 O，否则可能会错看成 1 和 0。

```
name = "发明者量化"
```

注意：Python 的变量在赋值的时候不需要类型声明。在使用该变量之前，必须对其赋值，赋值之后变量才会创建。

2.2.3 关键字

在使用 Python 语言编写代码时，有一些特殊的名词是不能作为变量名、函数名或者其它用途使用的，这些名词叫做“关键字”或者“保留字”。Python 自带的 keyword 模块可以输出这些系统关键字。例如我们在编写的 Python 策略代码中使用：

```
import keyword
def main():
   Log(keyword.kwlist)
```

输出结果为：

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global',
'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass',
'raise', 'return', 'try', 'while', 'with', 'yield']
```

注意：这些'False'、'None'、'True'等等名词，都有它们自己的用途，是已经预先保留的关键字，不能再用作其它的命名。另外 Python 是一种动态语言，关键字会随着时间的变化而改变。

2.2.4 注释

为了提高代码可读性，可以在代码中添加解释和说明。良好的代码注释可以传达代码作用和上下文关系，便于理解策略逻辑，也方便日后维护策略。Python 的单行注释由一个“#”号开头，之后跟上注释文本：

```
# 第一个注释，单行注释，Log 函数是用于输出一条信息的函数
Log("你好，发明者！")
```

如果注释的内容比较多，可以使用多行注释三个连续的单引号'''或者三个连续的双引号""""，一次性注释多行的内容（包含一行），具体格式如下：

```
'''
第 1 行注释
第 2 行注释
'''
```

程序在运行时会忽略已经注释的代码，所以基本不会影响代码的运行速度。除此之外，注释还以帮助调试程序 BUG，如果觉得某段代码可疑，可以先把该段代码注释起来，代码可以再次正常运行，则说明 BUG 是由于这段代码引起的。合理的利用注释，可以缩小 BUG 的范围，提高调试策略的效率。

2.2.5 缩进

Python 的缩进是一种独特的语法，也是该语言的一大特点，它没有像其他语言一样用花括号{}分隔代码块，而是使用 Tab 键或 4 个空格进行代码缩进，以此来控制代码的作用域，相同缩进行的代码处于同一个作用域范围。

需要注意的是，空格和 tab 缩进不能混在一起用，否则会报错。使用空格缩进时，如果空格数量不一致，也会引起报错，例如：

```
if True:
   print ("Answer")
   print ("True")
else:
   print ("Answer")
  print ("False")                # 缩进不一致，会导致运行错误
```

2.2.6 代码块

缩进行相同的一组语句构成一个代码块，很多关键字例如：while、def、class、if 等关键字使用时，在冒号“:”后换行，缩进相同的代码构成代码块。例如：

```
if 1 < 2 :
   Log("1 小于 2 为真")              # -----
   Log("计算一下 2 比 1 大多少？")   #  代码块
   Log("计算：2-1=", 2-1)          # -----
elif 1 > 2 :                      # 1>2 结果是 False
   Log("1 大于 2 位真")              # 所以这个条件不会触发
```

输出结果为：

```
1 小于 2 为真
计算一下 2 比 1 大多少？
计算：2-1= 1
```

2.2.7 空行

通常在编写代码时，习惯于在函数之间或者类成员函数之间使用空行分隔，表示新的一段代码。这个并不是语法，仅仅是编写策略时的习惯，便于之后代码阅读，主要作用是分隔两段功能或者含义不同的代码。

2.2.8 导入模块

模块就像已经制造好的汽车零部件，通过生产线把各个零件组装成一体。编程也是同样的道理，在编写策略时，可以用过“import”导入模块。模块的好处是提高了策略开发效率，一般写在代码开头，有以下四种形式：

- 整个模块导入，写为：import module
- 从某个模块中导入某个函数，写为：from module import def
- 从某个模块中导入多个函数，写为：from module import def1，def2
- 某个模块中的全部函数导入，写为：from module import *

2.3 Python 变量和数据类型

变量其实是内存中的值，当变量创建的时候，Python 会自动识别值的类型，并根据类型分配到指定的内存中。变量可以存储不同的数据类型，包括：数字、字符串、列表、字典等等。

2.3.1 变量

Python 中变量不需要声明，但每个变量在使用之前必须赋值，变量赋值之后该变量才成功创建。使用“=”等号给变量赋值，等号左边为变量名称，等号右边为储存在变量中的值。例如：

```
pi = 3.1415926535897
name = "圆周率"
year = 2019
Log(pi, name, year)
```

输出结果为：

```
3.1415926535897 圆周率 2019
```

2.3.2 标准数据类型

Python 中的变量仅仅只是一个名字（name），它关联了内存中的一个数据（object）。而变量类型实际上指的是该变量关联在内存中数据（object）的类型。Python3 有六个标准类型，它们分别是：

- Number（数值）
- String（字符串）
- List（列表）
- Tuple（元组）
- Set（集合）
- Dictionary（字典）

通常在编写一般的策略代码时用的最多的就是 Number（数值）、String（字符串）、List（列表）、Dictionary（字典）这些数据类型。在接下来的章节中将重点讲解这些常用的基本数据类型的使用。

2.3.3 Number（数字）

Number 数据类型用于储存数字，常用的数字类型为：整型（int）、浮点型（float）。整型就是不带小数点的数字，正整数和负整数都是整数类型。数字是不可变类型，一旦改变其数据类型的值，那么就意味着重新分配内存空间。

注意：Python 没有布尔类型（bool），但数字 1 和 0 来可以用来表示 True 和 False。True、False 也是关键字。例如：

```
value_int = 10          # int 类型，整型变量，简单理解就是整数字变量。
value_float_1 = 3.14    # float 类型，浮点型变量，简单理解就是有小数部分的变量。
Value_float_2 = 3.00    # 值为 3.00 的变量也是浮点类型变量。
```

由于 Python 属于动态语言，很多时候需要判断对象的类型，可以使用内置的 type 函数，调用它就能查询对象类型信息。例如：

```
def main():
    value_int = 10           # int 类型，整型变量，简单理解就是整数字变量。
    value_float = 3.14       # float 类型，浮点型变量，简单理解就是有小数部分的变量
                             # 值为 3.00 的变量也是浮点类型变量。
    Log(type(value_int))     # 打印变量 value_int 的类型。
    Log(type(value_float))   # 打印变量 value_float 的类型。
```

输出结果为：

```
<class 'int'>
<class 'float'>
```

这个例子中，分别定义了 int（整数）类型和 float（浮点）类型变量，Log 函数打印了 type()函数返回的变量类型。无论是整型还是浮点型变量，都是用来表示数字，用于计算。

注意：在浮点型、整型变量混合计算时，Python 会把整型先转换为浮点型。另外数字的除法使用运算符“/”，返回一个浮点数，使用“//”返回一个整数。

2.3.4 String（字符串）

字符串是若干个字符的集合，表示文本的数据类型，Python 中的字符串用单引号''或者双引号""括起来，可以使用反斜杠\转义特殊字符。字符串的第一个索引是 0，第二个索引是 1，依此类推。也可以对字符串相加、截取、复制等操作。例如：

```
def main():
    str = 'hello fmz'
    Log(str)              # 输出字符串
    Log(str[0:-1])        # 输出第一个到倒数第二个的所有字符
    Log(str[0])           # 输出字符串第一个字符
    Log (str[2:5])        # 输出从第三个开始到第五个的字符
    Log (str[2:])         # 输出从第三个开始的后的所有字符
    Log (str * 2)         # 输出字符串两次
    Log (str + "!!")      # 连接字符串
```

输出结果为：

```
hello fmz
hello fm
H
Llo
llo fmz
hello fmzhello fmz
hello fmz!!
```

注意：Python 中的字符串不能改变，也就是说当字符串被创建完成后，就不能再改变它的状态了。例如在下面的例子中，重新给字符串的第一个索引位置赋值，会引起报错：

```
def main():
    str = 'hello fmz'
    Log(str[0])
    str[0] = 'H'
```

输出结果为：

```
h
Traceback (most recent call last): File "<string>", line 1481, in Run File
"<string>", line 9, in <module> File "<string>", line 4, in main TypeError:
'str' object does not support item assignment
```

2.3.5　List（列表）

列表就像是备忘清单，每一个编号记录着清单详情，它是有序数据的集合，通过编号就可以引用列表中的数据。列表也是策略开发中使用比较频繁的数据类型，商品期货 API 接口返回的大部分数据都是以列表形式呈现。Python 的列表可以存储不同类型的元素，包括：数字、字符串、列表、字典等等。

列表使用方括号“[]”包含元素，其中每个元素中间使用逗号“，”作为间隔符。和字符串类似，列表也可以通过索引获取其中的元素，也可以使用索引截取列表中的一部分，列表被截取后返回一个新的列表。例如：

```
def main():
    list = ["abc", 10, 3.14, ["1", 2, 3.0]]
    Log(list)                # 输出整个列表
    Log(list[0])             # 输出列表中的第一个元素
    Log(list[1:3])           # 从第二个元素开始输出到第三个元素
    Log(list[2:])            # 从第三个元素开始输出所有元素
    Log(list*2)              # 两个 list 列表连接在一起
    Log(list[-1][-1])        # 输出列表中嵌套的列表的最后一个元素
    Log(list + list[-1])     # 连接两个列表
```

输出结果为：

```
['abc', 10, 3.14, ['1', 2, 3.0]]
abc
[10, 3.14]
[3.14, ['1', 2, 3.0]]
['abc', 10, 3.14, ['1', 2, 3.0], 'abc', 10, 3.14, ['1', 2, 3.0]]
3.0
['abc', 10, 3.14, ['1', 2, 3.0], '1', 2, 3.0]
```

和 Python 字符串不一样的地方是，列表中的元素是可以改变的，包括：索引、切片、增删改查等基本操作。例如：

```
def main():
    list = ["abc", 10, 3.14, ["1", 2, 3.0]]
    Log("修改 list[0]之前：", list)
```

```
    list[0] = "hello fmz!"
    Log("修改 list[0]之后：", list)
```

输出结果为：

```
修改 list[0]之前： ['abc', 10, 3.14, ['1', 2, 3.0]]
修改 list[0]之后： ['hello fmz!', 10, 3.14, ['1', 2, 3.0]]
```

Python 有很多适用于列表的函数，例如：len()函数可以获取列表里面有多少个元素，append()函数可以向列表尾部添加一个元素，pop()函数可以移除一个元素，默认移除最后一个元素。例如：

```
def main():
    list = ["abc", 10, 3.14, ["1", 2, 3.0]]
    list.append("aaa")
    Log(list)
    list.pop()
    Log(list)
```

输出结果为：

```
['abc', 10, 3.14, ['1', 2, 3.0], 'aaa']
['abc', 10, 3.14, ['1', 2, 3.0]]
```

2.3.6 Dictionary（字典）

字典也是 Python 语言常用的一种数据结构，它是存放具有映射关系的数据，定义了键和值之间一对一的映射关系，它是一个无序、可变和有索引的集合。字典的数据用花括号“{}”包括，结构形式如下：

```
def main():
    dict1 = {
        "name" : "TOM",
        "age" : 18,
        "address" : {
            "city" : "xxx",
            "street" : "yyy"
        }
    }

    Log(dict1)
    Log("姓名：", dict1["name"], "年龄：", dict1["age"], "地址，城市：",
        dict1["address"]["city"], "街道：", dict1["address"]["street"])
```

可以看到，字典中的数据是一个键名对应一个键值，例如：name 这个键名（keyName）对应 TOM 这个键值（keyValue）。和列表类似，字典也可以嵌套，如上所示：address 这个键名对应的键值也是一个字典。

输出结果为：

```
{'name': 'TOM', 'age': 18, 'address': {'city': 'xxx', 'street': 'yyy'}}
姓名： TOM 年龄： 18 地址，城市： xxx 街道： yyy
```

注意：因为字典是通过键来访问值的，所以字典中键名（keyName）必须是唯一的，并

且键名必须使用不可变类型。也可以使用内置的函数 keys()输出所有键名，使用函数 values()输出所有键值，例如：

```
def main():
    dict1 = {
        "name" : "TOM",
        "age" : 18,
        "address" : {
            "city" : "xxx",
            "street" : "yyy"
        }
    }
    Log(dict1.keys())
    Log(dict1.values())
```

输出结果为：

```
dict_keys(['name', 'age', 'address'])
dict_values(['TOM', 18, {'city': 'xxx', 'street': 'yyy'}])
```

2.3.7 Python 数据类型转换

Python 提供了几种数据类型转换函数，可以将一种数据类型转变为另一种数据类型。比如：浮点数转换为整数、整数转换为字符串等等。通常情况下不同的数据类型是可以相互转换的，这也意味着：整数可以转换为浮点数、字符串也可以转换为整数等等。

1、将 x 转换为 int 类型：ret = int(x)

2、将 x 转换为 float 类型：ret = float(x)

3、将 x 转换为 string 类型：ret = str(x)

```
def main():
    pi = 3.14
    Log(int(pi))
    strPi = "3.14"
    Log(float(strPi))
    Log(type(str(pi)))
```

输出结果为：

```
3
3.14
<class 'str'>
```

2.4 Python 数据运算

计算机里面的数据运算与数学运算类似，数据运算也是有优先级的。但 Python 的数据运算更具有丰富多样性，支持以下常用数据运算：

- ❑ 算术运算

- 关系（比较）运算
- 赋值运算
- 逻辑运算

例如以下代码，a、b 被称为操作数，“+”号就被称为运算符。编程语言中的运算符有很多，在 Python 中有很多类型的运算符，包括：算术运算符、关系运算符、赋值运算符、逻辑运算符。

2.4.1 算术运算符

算术运算也就是数学运算，其运算规则与数学运算规则一样。算术运算符就是用来对操作数进行数学运算，主要有：+、-、*、/、%、**、//等运算符。例如：

```
def main():
    a = 3
    b = 2
    Log("加法运算符 + 计算结果:", a + b)
    Log("减法运算符 - 计算结果:", a - b)
    Log("乘法运算符 * 计算结果:", a * b)
    Log("除法运算符 / 计算结果:", a / b)
    Log("求模运算符 % 计算结果:", a % b)        # 计算相除(a/b)时的余数。
    Log("幂运算符 ** 计算结果:", a ** b)        # 计算 a 的 b 次方。
    Log("整除运算符 // 计算结果:", a // b)      # 向下取接近除数的整数。
```

输出结果为：

```
加法运算符 + 计算结果: 5
减法运算符 - 计算结果: 1
乘法运算符 * 计算结果: 6
除法运算符 / 计算结果: 1.5
求模运算符 % 计算结果: 1
幂运算符 ** 计算结果: 9
整除运算符 // 计算结果: 1
```

2.4.2 关系运算符

关系运算也称比较运算，关系运算符主要是用于对操作数进行数字大小关系比较。主要有：==、!=、>、<、>=、<= 等运算符。如果关系运算成立，返回 True（真），反之返回 False（假）。例如：

```
def main():
    a = 3
    b = 2
    c = 2
    Log("c:", c, "b:", b, "使用 c == b 判断，两边操作数是否相等，返回：", c == b)
    Log("a:", a, "b:", b, "使用 a == b 判断，两边操作数是否相等，返回：", a == b)
    Log("a:", a, "b:", b, "使用 a != b 判断，两边操作数是否不等，返回：", a != b)
```

```
    # a 值为 3，b 值为 2， 3 > 2，关系表达式是成立的，返回结果 True，即为真。
    Log("a:", a, "b:", b, "使用 a > b 判断，两边操作数大小关系，返回：", a > b)
    # 2 < 2 不成立，返回 False，即为假。
    Log("c:", c, "b:", b, "使用 c < b 判断，两边操作数大小关系，返回：", c < b)
    Log("c:", c, "b:", b, "使用 c >= b 判断，两边操作数大小关系，返回：", c >= b)
    Log("b:", b, "a:", a, "使用 b <= a 判断，两边操作数大小关系，返回：", b <= a)
```

输出结果为：

```
c: 2 b: 2 使用 c == b 判断，两边操作数是否相等，返回： True
a: 3 b: 2 使用 a == b 判断，两边操作数是否相等，返回： False
a: 3 b: 2 使用 a != b 判断，两边操作数是否不等，返回： True
a: 3 b: 2 使用 a > b 判断，两边操作数大小关系，返回： True
c: 2 b: 2 使用 c < b 判断，两边操作数大小关系，返回： False
c: 2 b: 2 使用 c >= b 判断，两边操作数大小关系，返回： True
b: 2 a: 3 使用 b <= a 判断，两边操作数大小关系，返回： True
```

2.4.3 赋值运算符

赋值运算是把右边的值传递给左边的变量，可以直接传递，也可以经过运算后再传递，比如加减乘除、函数调用、逻辑运算等。赋值运算符主要有：=、+=、-=、*=、/=、%=、**=、//=。例如：

```
def main():
    a, b = 3, 2
    a = b              # 把 b 的值赋值给 a，打印 a，显示 2
    Log(a)
    a, b = 3, 2
    a += b             # 等价于 a = a + b，打印 a，显示 5
    Log(a)
    a, b = 3, 2
    a -= b             # 等价于 a = a - b，打印 a，显示 1
    Log(a)
    a, b = 3, 2
    a *= b             # 等价于 a = a * b，打印 a，显示 6
    Log(a)
    a, b = 3, 2
    a /= b             # 等价于 a = a / b，打印 a，显示 1.5
    Log(a)
    a, b = 3, 2        # 等价于 a = a % b，打印 a，显示 1
    a %= b
    Log(a)
    a, b = 3, 2        # 等价于 a = a ** b，打印 a，显示 9
    a **= b
    Log(a)
    a, b = 3, 2        # 等价于 a = a // b，打印 a，显示 1
    a //= b
    Log(a)
```

运行结果为：

```
2
5
1
6
1.5
1
9
1
```

2.4.4　逻辑运算符

Python 中的逻辑运算与高中数学的逻辑运算类似，比如 a 为真命题，b 为假命题，那么“非 a”为假，“a 且 b”为假，“a 或 b”为真。Python 的逻辑运算符有 and、or、not。

and 又称为“与”操作符，假设有 x，y（x，y 可以是表达式，也可以是数值），x and y 这样就组成了一个逻辑表达式。如果 x 为 False，那么 x and y 返回 False，否则返回 y 的计算值。例如：

```
def main():
    x = 10
    y = 20
    z = False
    Log(x and y)     # x and y 这个逻辑表达式返回的值为 y 的值，即 20
    Log(z and y)     # z 值为 False，为假，则 z and y 这个逻辑表达式返回的值为 False
```

输出结果为：

```
20
False
```

or 又称为“或”运算符。同样假设有 x，y 两个表达式或者数值，x or y 组成一个逻辑表达式。如果 x 为 True，那么 x and y 返回 x 的值，否则返回 y 的计算值。例如：

```
def main():
    x = 10
    y = 20
    z = False
    Log(x or y)   # x 的值为 10，为真，表达式 x or y 的值为 10
    Log(z or y)   # z 的值为 False，为假，表达式 x or y 的值为 y 的计算值，即 20
```

输出结果为：

```
10
20
```

最后来看一下 not 操作符，not 操作符又称“非”操作符。假设有 x 这个表达式或者数值，not x 组成一个逻辑表达式。如果 x 为 True，则 not x 返回 False，如果 x 为 False，则 not x 返回 True。例如：

```
def main():
    x = 10
    y = 20
```

```
    z = False
    Log(not (x or y))     # x or y 为真，所以 not (x or y)为假
    Log(not (z and y))    # z and y 为假，所以 not (z and y)为真。
```

输出结果为：

```
False
True
```

在 Python 中，in 和 not in 是用于逻辑判断的另一种方式，可以简单理解为 in 左边的内容是否存在于 in 右边的内容，如果存在返回 True，如果不存在返回 False。例如：

```
def main():
    a = "hello FMZ!"
    b = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    c = {"name": "FMZ", "age": 10}
    Log('a' in a)
    Log('F' in a)
    Log(10 in b)
    Log(0 in b)
    Log('say' in c)
    Log('name' in c)
```

输出结果为：

```
False
True
False
True
False
True
```

2.4.5 运算符优先级

Python 的运算优先级是一个很重要的概念，在一个表达式中有多个运算符时，算数优先级决定了先执行哪个运算符。

运算符说明	Python 运算符	优先级
乘方	**	7
乘、除、取模、整除	*、/、%、//	6
加、减	+、-	5
比较运算符	<=、<、>、>=	4
等于运算符	==、!=	3
赋值运算符	=、+=、-=、*=、/=	2
逻辑运算符	not、and、or	1

上表从高到低列出了运算符的优先级，同一行运算符的优先级是从左到右顺序排列。先执行具有较高优先级的运算，然后执行较低优先级的运算。如果想要改变默认的运算顺序，可以使用圆括号，例如：11 + 2 -5 * (3 + 2 – (5 + 1))中，小括号最内层的 5+1 最先计算。

2.5 Python 数字和字符串

数字和字符串几乎是所有编程语言里面最基本的数据类型，也是通过策略代码实现量化交易的基础。Python 语言中有很多处理数字和字符串的函数，这些内置的函数通常能解决大多数策略开发需求。

2.5.1 数字类型转换

Python 内部提供了几种强制类型转换的函数，可以将一种数据类型转换为另一种数据类型，其中有些类型是可以相互转换的。例如：整型转换为浮点型，浮点型转换为整型，也可以将部分字符串转换为数字。

```
def main():
    Log(int(3.14))   # 将浮点型转换为整型
    Log(float(10))   # 将整型转换为浮点型
    Log(int('100'))  # 将字符串转换为整型
```

2.5.2 内置数学函数

Math 库提供了很多复杂数学运算函数，包括：自然常数、圆周率、绝对值，四舍五入等函数。这些函数并不能直接访问，需要使用“import”导入 math 模块，通过静态对象调用才能使用。例如：

```
import math                       # 导入 math 数学库

def main():
    a = -10
    Log(math.e)                   # 打印自然常数
    Log(math.pi)                  # 打印圆周率
    Log(abs(a))                   # 计算 a 的绝对值
    Log(math.ceil(math.pi))       # math.ceil(x)返回数值变量 x 的上入整数
    Log(math.exp(1))              # math.exp(x) 返回自然常数 e 的 x 次幂
    Log(math.fabs(a))             # math.fabs(x)返回 x 的绝对值，返回值为浮点类型
    Log(math.floor(math.pi))      # math.floor(x)返回数值变量 x 的下舍整数
    Log(math.log(100, 10))        # math.log(x, y)返回以 y 为基数的 x 的对数
    Log(max(a, math.pi))          # 求传入的参数中的最大值，参数可以是列表
    Log(min([a, math.pi, 0]))     # 求传入的参数中的最小值，参数可以是列表
    Log(math.modf(math.pi))       # modf(x)返回 x 的整数部分和小数部分
    Log(round(math.pi, 1))        # round(x, n)计算浮点数 x 的四舍五入值
    Log(math.sqrt(100))           # math.sqrt(x)计算 x 的平方根
```

输出结果为：

```
2.718281828459045
3.141592653589793
10
```

```
4
2.718281828459045
10.0
3
2.0
-10
3.1
```

2.5.3　访问字符串中的值

字符串是由多个字符组成，字符与字符之间是有顺序的，而这个顺序号被称为索引。字符串的索引是从 0 开始，以此类推。例如有一个字符串 stringA = “Hello FMZ”，那么它在内存中的实际存储顺序如下：

H	e	l	l	o		F	M	Z	!
0	1	2	3	4	5	6	7	8	9

如果要选取字符串区间内容，则需要遵循左闭右开的原则，即从“起始”位开始，到“结束”位的前一位结束（不包含结束位本身)。倒数第一个元素的索引是-1。例如：

```
def main():
    stringA = "Hello FMZ!"
    Log(stringA[6:9])
    Log(strintA[-1])
```

输出结果为：

```
FMZ
!
```

2.5.4　拼接字符串

在 Python 中字符串拼接有很多种方式：直接通过加号（+）拼接，或者通过逗号（，）拼接。但如果需要拼接大量字符串时，这两种方法就非常低效了，这时候可以使用 Python 内置的 join()函数进行拼接。例如：

```
def main():
    a = "hello,"
    b = "FMZ!"
    Log(a + b)
    Log(a, b)
    Log(' '.join([a, b]))
```

输出结果为：

```
hello,FMZ!
hello, FMZ!
hello, FMZ!
```

2.5.5 其他常用函数

除此之外，Python 还有一些其他常用的函数用于处理字符串：

- len()函数：用于返回字符串的字符个数。
- lower()函数：将字符串中的所有字符转换为小写。
- upper()函数：将字符串中的所有字符转换为大写。
- replace()函数：替换字符串中部分字符。
- split()函数：字符串分割函数。

例如：

```
def main():
    stringA = "Hello FMZ!"
    Log(len(stringA))
    Log(stringA.lower())
    Log(stringA.upper())
    Log(stringA.replace("FMZ", "发明者量化"))
    arr = stringA.split(" ")  # 以空格分割 stringA
    Log(arr[0])
    Log(arr[1])
```

输出结果为：

```
10
hello fmz!
HELLO FMZ!
Hello 发明者量化!
Hello
FMZ!
```

2.6 Python 列表和字典

列表和字典都是 Python 语言最常用的数据结构，列表是有序数据的集合，字典是无序数据的集合。列表中每一个元素都有它的索引，字典中每个元素都包含键值对。

2.6.1 列表索引

列表是 Python 中最基本的数据结构，列表中的每个元素都有一个索引，即一个数值，用于标记列表中元素的位置，第一个元素的索引为 0，第二个元素的索引为 1，依次类推。列表中的元素可以是不同类型的数据，例如：

```
def main():
    arr = ["Tom", 18, ["12345678@qq.com", 135123456789]]
    Log("姓名：", arr[0])
    Log("年龄：", arr[1])
    Log("联系方式，邮箱：", arr[2][0])
```

```
    Log("联系方式，电话：", arr[2][1])
```

输出结果为：

```
姓名： Tom
年龄： 18
联系方式，邮箱： 12345678@qq.com
联系方式，电话： 135123456789
```

2.6.2　列表切片

通过列表切片，可以获取一个列表中的部分元素。列表切片与字符串类似，也是需要遵循左闭右开的原则，即从“起始”位开始，到“结束”位的前一位结束（不包含结束位本身)。倒数第一个元素的索引是-1。也可以用 len()函数获取列表中的元素个数：

```
def main():
    arr = [1, 2, 3, 4, 5, 6]
    Log(arr[1:3])
    Log(arr[-1])
    Log(len(arr))
```

输出结果为：

```
[2, 3]
6
6
```

2.6.3　列表修改删除

列表是可变的数据类型，列表中的元素可以被修改、删除。直接使用赋值操作符就能修改列表中的元素。例如把列表中索引为 1 的元素修改为 22（原本是 2）。

```
def main():
    arr = [1, 2, 3, 4, 5, 6]
    arr[1] = 22
    Log(arr)
```

输出结果为：

```
[1, 22, 3, 4, 5, 6]
```

Python 提供了 4 种删除列表元素的函数，每一种方法分别适应于不同的场景。包括：del、pop、remove、clear。例如：

```
def main():
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    del arr[1]        # 根据索引值删除元素
    Log(arr)
    arr.pop()         # 根据索引值删除元素，默认删除最后一个元素
    Log(arr)
    arr.remove(5)     # 根据元素值进行删除
    Log(arr)
    arr.clear()       # 删除列表所有元素
```

```
    Log(arr)
```

输出结果为：

```
[1, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 3, 4, 5, 6, 7, 8, 9]
[1, 3, 4, 6, 7, 8, 9]
[]
```

2.6.4 二维列表

列表中的元素可以是任何一种数据类型，其中包括列表。如果一个列表中包含列表，那么这个列表就是二维列表。例如：

```
def main():
    arr = [[1, 2], [3, 4], [5, 6]]
    Log(arr[0][0])  # 获取 arr 列表第 1 个元素中的第 1 个元素
    Log(arr[1][0])  # 获取 arr 列表第 2 个元素中的第 1 个元素
    Log(arr[2][0])  # 获取 arr 列表第 3 个元素中的第 1 个元素
```

在量化交易中心，二维列表多用于技术指标中。如果要获取二维列表中列表元素里面的值，可以参考上面的例子。输出结果为：

```
1
3
5
```

众所周知，MACD 指标一共有 3 个数据，包括：dif 线、dea 线、macd 量柱。如果使用 talib 库中的 MACD 指标计算，返回的则是一个二维数组。第一个元素就是 MACD 指标中的 dif 线的数据，第二个元素是 dea 线的数据，第三个元素是量柱数据。例如：

```
def main():
    macd = [[1, 2, 3], [1.1, 2.2, 3.3], [1.11, 2.22, 3.33]] # MACD 值
    Log("dif 线：", macd[0])
    Log("dea 线：", macd[1])
    Log("macd 量柱：", macd[2])
    Log("当前 Bar 的 dif 指标值：", macd[0][-1])
```

输出结果为：

```
dif 线： [1, 2, 3]
dea 线： [1.1, 2.2, 3.3]
macd 量柱： [1.11, 2.22, 3.33]
当前 Bar 的 dif 指标值： 3
```

2.6.5 列表增加元素

在 Python 中 append 函数用来想列表尾部追加元素，如果所追加的元素是个列表，那么这个列表将作为一个整体来追加。例如：

```
def main():
    arr = [1, 2, 3, 4]
    arr.append("100")
```

```
    Log(arr)
    arr.append([99, 100])
    Log(arr)
```

输出结果为：

```
[1, 2, 3, 4, '100']
[1, 2, 3, 4, '100', [99, 100]]
```

注意：列表增加元素后，列表的长度也会自动增加。

2.6.6　列表反向排序

reverse 是列表中一个非常实用的内置函数，它可以让列表中的元素反向排序，该函数可以返回一个逆序序列的迭代器（用于遍历该逆序序列）。例如：

```
def main():
    arr = [1, 2, 3, 4]
    arr.reverse()
    Log(arr)
```

输出结果为：

```
[4, 3, 2, 1]
```

2.6.7　创建字典

字典是也一种可变的数据类型，字典中的键和值是一一对应的，其中键（key）就是数据的名字，值就是数据的内容。字典使用冒号 “:” 分隔键（key）值（value）对，然后每个键值对用逗号 “,” 分隔。最后使用花括号“{}”包裹起来。例如：

```
dict = {key1 : value1, key2 : value2 , key3 : value3}
```

字典中键（key）必须是唯一的，值（value）是可以重复的，值也可以是任何数据类型，但是键必须是不可变的数据类型，例如：数值，字符串都可以作为键。字典的创建方式很简单，例如：

```
def main():
    boy = {
        "name" : "tom",
        "age" : 18,
        "Email" : "123456789@qq.com"
    }

    Log(boy)
```

输出结果为：

```
{'name': 'tom', 'age': 18, 'Email': '123456789@qq.com'}
```

2.6.8　访问字典元素

汉语字典可以通过拼音查汉字，Python的字典访问也是基于这个原理，可以通过键（key）访问字典中的值（value）。具体方法是：在字典变量名后面写中括号“[]”，然后在中括号内写要访问的键名。例如：

```
def main():
    boy = {
        "name" : "tom",
        "age" : 18,
        "Email" : "123456789@qq.com"
    }

    Log("名字：", boy["name"])
    Log("年龄：", boy["age"])
    Log("地址：", boy["address"])   # 访问字典中不存在的键，会报错。
```

输出结果为：

```
名字： tom
年龄： 18
Traceback (most recent call last): File "<string>", line 1481, in Run File
"<string>", line 15, in <module> File "<string>", line 10, in main KeyError:
'address'
```

注意：如果访问字典并不存在的键，程序就会报错。

2.6.9　字典添加修改元素

字典是可变的数据类型，这也就意味着字典可以增删改查。那么如何增加和修改字典中的元素呢？和访问字典中键值的方式一样，只不过是对其赋值操作。例如：

```
def main():
    boy = {
        "name" : "tom",
        "age" : 18,
        "Email" : "123456789@qq.com"
    }
    boy["height"] = "180cm"              # 如果字典中没有该键，就创建一个键并赋值
    boy["Email"] = "abcdefg@qq.com"      # 如果字典中该键存在，就更新该键值
    Log(boy)
```

输出结果为：

```
{'name': 'tom', 'age': 18, 'Email': 'abcdefg@qq.com', 'height': '180cm'}
```

2.6.10　字典删除元素

Python 字典有 4 种删除元素的方法，可以适应于不同的应用场景。其中一个 del 关键字，del 是全局方法，既能删除单个元素又能删除字典，例如：

```
def main():
    boy = {
        "name" : "tom",
        "age" : 18,
        "Email" : "123456789@qq.com"
    }

    del boy["age"]   # 删除字典中的键值对
    Log(boy)
    del boy          # 删除整个字典
    Log(boy)
```

输出结果为：

```
{'name': 'tom', 'Email': '123456789@qq.com'}
Traceback (most recent call last): File "<string>", line 1481, in Run File
"<string>", line 16, in <module> File "<string>", line 11, in main
UnboundLocalError: local variable 'boy' referenced before assignment
```

上面的例子中，字典的 age 键和键值，都被删除了。使用 del 关键字，如果后面跟的要删除的内容是一个字典（del boy），那么删除的就是整个字典 boy。如果要清空一个字典内容，可以直接调用字典的 clear 函数。例如：

```
def main():
    boy = {
        "name" : "tom",
        "age" : 18,
        "Email" : "123456789@qq.com"
    }

    boy.clear()     # 清空字典的操作
    Log(boy)
```

输出结果为：

```
{}
```

注意：字典值可以是任意 python 对象，但是字典键必须是不可变类型。并且字典中相同的键不允许出现两次，如果创建一个字典时出现两次相同的键，那么只会记录最后一个。例如：

```
def main():
    boy = {
        "name" : "tom",
        "age" : 18,
        "name" : "jack",
        "Email" : "123456789@qq.com"
    }
    Log(boy)
```

输出结果为：

```
{'name': 'jack', 'age': 18, 'Email': '123456789@qq.com'}
```

2.7 Python 条件语句和循环语句

编程与生活息息相关，比如红灯停，绿灯行就是条件语句。条件语句和循环语句在量化交易也很常用，策略之所以会实时根据行情变化，发现潜在的交易机会，是因为它在循环语句中重复的判断交易信号是否成立。之所以会自动下单交易，是因为它可以根据条件语句执行下单动作。

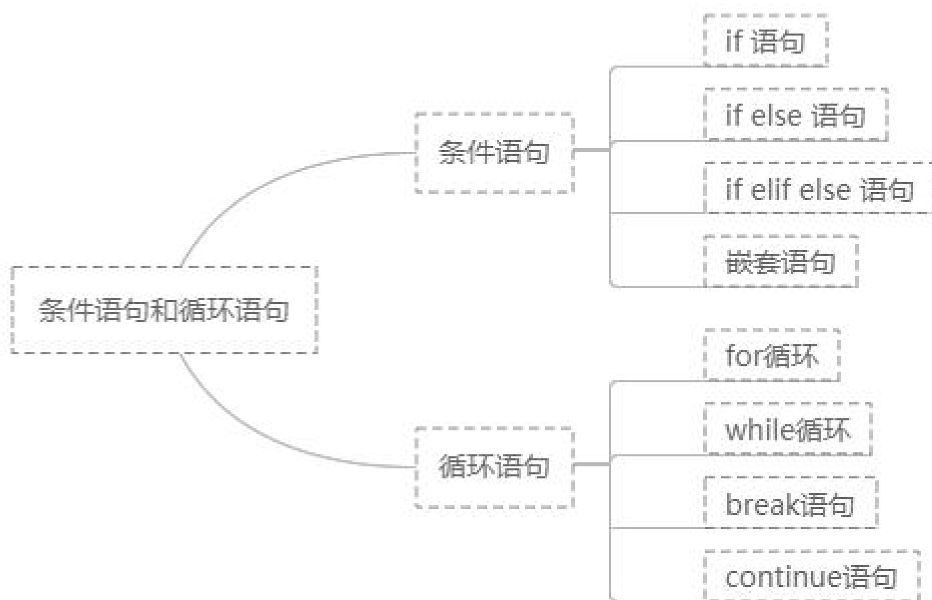

图 2.1 条件语句和循环语句

2.7.1 条件语句

计算机在执行代码时，是按照从上到下顺序，一行一行的执行。但很多时候这种按顺序结构执行代码有很大的局限性。假如有一个策略逻辑是只有在均线金叉时才能买入，这个时候就需要用到 if 条件语句了。

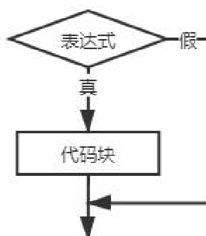

图 2.2　条件语句（1）

if 语句是选择结构，它就像是一个开关，通过对条件进行判断，然后根据判断结果执行不同的代码，这个条件可以是单一的值，也可以是由运算符组词的复杂语句，只要这个条件能得到一个值，if 语句都能判断它是否成立。如果条件成立，那么就会执行 if 里面的代码块，否则就会跳过 if 语句。例如：

```
def main():
    a = 5
    b = 10
    if a > 6:
        Log(a)
    if b > 6:
        Log(b)
```

输出结果为：

```
10
```

通常情况下，if 和 else 可以组合成“如果...否则...”的条件语句。如果条件成立，那么就会执行 if 里面的代码块，else 里面的代码块将会被跳过不执行。如果条件不成立，那么 if 里面的代码块将会被跳过不执行，然后执行 else 里面的代码块。

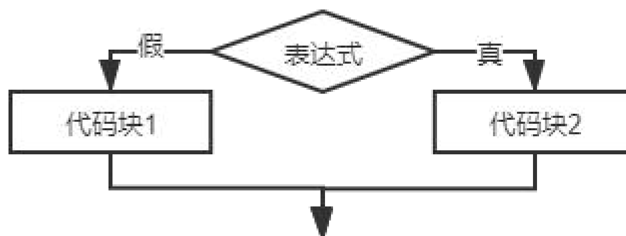

图 2.3　条件语句（2）

例如：

```
def main():
    a = 5
    b = 10
    if a > 6:
        Log(a)
    else:
        Log(b)
```

输出结果为：

```
10
```

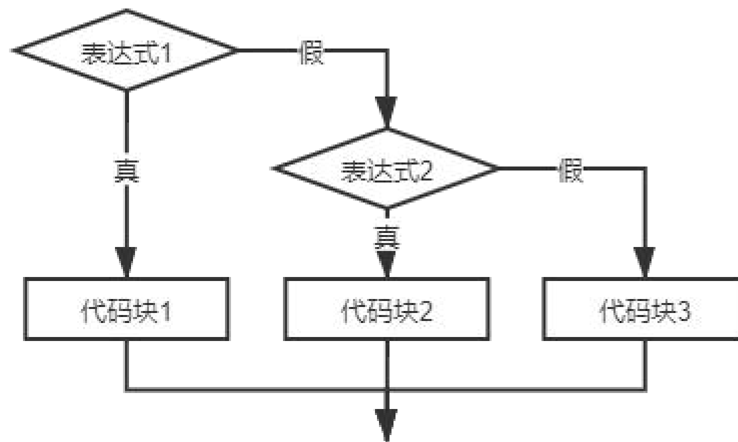

图 2.4　条件语句（3）

还有一种 if elif else 形式的语句，这种语句依次判断表达式的值，当某一个表达式的值为真时，则执行对应的代码块。如果所有的表达式为假，则执行 else 里面的代码块。例如：

```
def main():
    a = 5
    b = 10
    if a > 100:
        Log(a)
    elif b > 100:
        Log(b)
    else:
        Log('a、b 都小于 100')
```

输出结果为：

```
a、b 都小于 100
```

if 语句同样可以用于嵌套，在嵌套 if 语句中，可以把 if...elif...else 结构放在另外一个 if...elif...else 结构中。例如：

```
def main():
    boy = {
        "name" : "tom",
        "age" : 18,
        "Email" : "123456789@qq.com"
    }

    if boy["age"] == 20:
        Log("tom is 20 years old!")
    elif boy["age"] == 19:
        Log("tom is 19 years old!")
    else:
        Log("tom is not 20 or 19 years old!I don`t know his age.")
```

```
        if boy["Email"] == "123456789@gmail.com":
            Log("Although I don't know Tom's age, I can email him!")
            Log("123456789@gmail.com")
        elif boy["Email"] == "123456789@qq.com":
            Log("Although I don't know Tom's age, I can email him!")
            Log("123456789@qq.com")
        else:
            Log("I don`t know his email address!")
```

输出结果为：

```
tom is not 20 or 19 years old!I don`t know his age.
Although I don't know Tom's age, I can email him!
123456789@qq.com
```

2.7.2　循环语句

循环是让计算机重复的做某件事情，Python 语言提供了 2 种循环语句，分别是 for 循环和 while 循环。for 一般用于有限次数循环。while 一般用于不定次数循环，某些条件触发退出循环。循环逻辑如图：

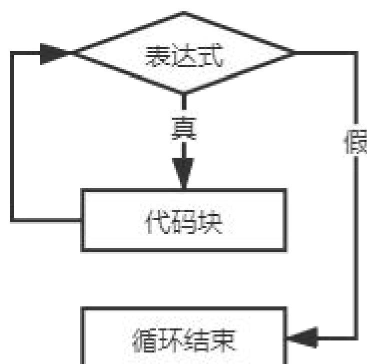

图 2.5　循环语句

for 循环可以遍历指定的次数，通常用于遍历一个有限的数据或者处理有限的任务。例如字符串、列表、字典等。例如使用 for 循环语句将一个字符串的字符逐个打印出来：

```
def main():
    stringA = "abc123"
    for char in stringA:
        Log(char)
    else:
        Log("打印结束")
```

输出结果为：

```
a
b
```

```
c
1
2
3
打印结束
```

如果数据是无限的，或者需要处理无限重复的任务，可以使用 while 循环语句。while 循环语句在每次开始循环之前，都会先判断条件语句是否为真，只要条件语句为真，就会执行循环体内的代码块。例如：

```
def main():
    a = 0
    while a < 100:
        a = a + 1
        Log(a)
```

输出结果为：

```
1
2
...
100
```

2.7.3 break 语句

break 语句是循环语句的搭档，当循环时出现 break 语句，循环就会立刻终止。如果是双层 for 循环，那么 break 语句只会终止当前的 for 循环。例如：

```
def main():
    arr1 = [1,2,3,4]
    arr2 = ["a", "b", "c", "d"]
    for i in arr1:
        for j in arr2:
            if j == "b":
                break
            Log("i:", i, "  j:", j)
```

输出结果为：

```
i: 1 j: a
i: 2 j: a
i: 3 j: a
i: 4 j: a
```

上面的例子使用了 2 个 for 循环，分别遍历 arr1 和 arr2，当在遍历 arr2 时遇到了 break 语句，就跳出了当前的 for 循环，所以 arr2 中第二个（b）、第三个（c）、第四个（d）这几个元素都不会被打印出来，但是 arr1 中的元素都打印了出来，说明 break 语句只是跳出了 for j in arr2 这个循环。

2.7.4 continue 语句

continue 语句有点像 break 语句，和 break 语句不同的是，它不是终止整个循环，而是跳过本次循环，并强制执行下一次循环。例如：

```
def main():
    arr = ["a", "b", "c", "d"]
    for i in arr:
        if i == "c":
            continue
        Log(i)
```

输出结果为：

```
a
b
d
```

上面的输出结果“c”这个字符串没有打印。因为在循环体内 if 语句判断 i == c 时，执行了 continue 语句。直接跳过了后面的 Log(i)代码，执行了下一次循环。continue 语句和 break 语句类似，都是只能作用于当前循环，不影响外层的循环（如果有的话）。

2.8 Python 日期和时间

量化交易经常需要和时间打交道，特别是对于一些日内策略或者交易频率比较高的策略来说，日期和时间的处理至关重要。Python 提供了 time、calendar、datetime 等模块用于处理日期和时间，其中较为常用的是 time、datetime 模块。

2.8.1 time 模块

Python 语言中处理时间需要使用 time 模块，导入 time 模块非常简单，使用 import 关键字即可。引入 time 模块以后，就可以调用该模块中的一些函数，做时间数据的处理，例如例子中的 time.time()函数，读取当前时间的秒级时间戳。例如：

```
import time              # 引入 time 模块

def main():
    Log(time.time())
```

输出结果为：

```
1595984400.0
```

2.8.2 什么是时间戳

时间戳是指自 1970 年 1 月 1 日（00:00:00 GMT）至当前时间的总秒数，常用的有秒级时间戳和毫秒级时间戳。时间戳具有唯一性，用于验证某个时间点存在的数据。严格来说

不管在地球哪个地方哪个时区，任意时间点的时间戳都是相同的。例如：

```
import time              # 引入 time 模块

def main():
    now = time.time()
    Log(now)             # 打印当前时间戳
    Log(type(now))
```

输出结果为：

```
1598931147.2031229
<class 'float'>
```

2.8.3 时间戳转换时间

从上面的例子中可以看到，时间戳是一个数字，在商品期货中，所有的数据都是基于时间戳。但如果数据以时间戳形式显示出来，看起来不直观，这不利于观察和分析数据，所以就需要把时间戳转换为传统的时间格式。

把时间戳转换为时间，可以使用 Python 语言 time 库中的函数转换，也可以使用 FMZ 平台的_D()函数转换。例如：

```
import time

def main():
    ts = time.time()          # 使用 time.time()获取当前的秒级别时间戳
    strTs = _D(ts)            # 将时间戳转换为可读的时间字符串
    Log("当前时间：", strTs)  # 打印当前的可读的时间字符串
```

输出结果为：

```
当前时间： 2020-07-29 09:00:00
```

注意：时间戳是不分时区全球统一的，在量化交易中一般不需要考虑时区的问题。

2.9 Python 常用内置函数

2.9.1 len()函数

Python 的 len()函数返回对象的元素数量或长度，可以适应于字符串、列表、字典等数据。字符串返回字符数量，列表返回元素数量，字典返回键值对数量。例如：

```
def main():
    a = "hello FMZ!"
    b = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    c = {"name": "FMZ", "age": 10}
    Log(len(a))
    Log(len(b))
    Log(len(c))
```

输出结果为：

```
10
10
2
```

2.9.2 range()函数

range()函数返回一个可以迭代的对象，这个对象是给定范围所生成一系列数字，常常与for 循环语句搭配使用。该函数至少需要一个参数，例如：

```
def main():
    for i in range(5):
        Log(i)
```

输出结果为：

```
0
1
2
3
4
```

上面的例子中 range(5)产生了一个从 0 开始到 4 的数列，遵循左开右闭的原则，从 0 开始（包含 0）到 5 结束（不包含 5）。也可以给 range 传 2 个参数，第一个参数确定起始数字，第二个参数确定结束数字，同样遵循左开右闭原则。例如：

```
def main():
    for i in range(2, 5):
        Log(i)
```

输出结果为：

```
2
3
4
```

range()函数还可以和 len()函数搭配使用，通过使用列表索引，遍历一个列表（区别于 for i in arr，注意变量 i 具体代表什么）。例如：

```
def main():
    arr = ["a", "b", "c", "d"]
    Log("第一个循环：")
    for i in arr:
        Log(i)

    Log("第二个循环：")
    for i in range(len(arr)):
        Log(i, "使用 i 访问列表中元素：", arr[i])
```

输出结果为：

```
第一个循环：
a
b
```

```
c
第二个循环：
0 使用 i 访问列表中元素： a
1 使用 i 访问列表中元素： b
2 使用 i 访问列表中元素： c
```

上面的例子，第一个循环执行时，每次打印 i 变量，显示的是字母，说明是每次从 arr 列表中取出元素，赋值给 i，然后打印 i，显示的就是列表中的元素。第二个循环执行时，每次打印 i 变量，显示的是数值，说明每次循环时 i 是列表中元素的索引。

2.9.3 split()函数

split 函数对字符串进行切片分割，返回分割后的字符串列表。下面的例子展示了 split()函数的使用方法：

```
def main():
    a = "hello FMZ!"
    b = a.split(" ")
    Log(b)
```

输出结果为：

```
['hello', 'FMZ!']
```

2.9.4 type()函数

type()函数在 Python 语言中是既简单又实用的对象数据类型查询方法，它是一个内部函数，调用它传入要查询的对象，就能够得到一个返回值，从而得知该对象类型信息。例如：

```
def main():
    a = "hello FMZ!"
    b = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    c = {"name": "FMZ", "age": 10}
    Log(type(a))
    Log(type(b))
    Log(type(c))
```

输出结果为：

```
<class 'str'>
<class 'list'>
<class 'dict'>
```

2.9.5 isinstance()函数

isinstance()函数常用于判断一个对象是否是一个已知的类型，类似 type()。区别在于：type()函数不会认为子类是一种父类类型，不考虑继承关系。isinstance()函数会认为子类是一种父类类型，考虑继承关系。例如：

```
def main():
    a = 3.14
    b = 4
    Log(isinstance(a, float))
    Log(isinstance(b, float))
```

输出结果为：

```
True
False
```

上面的例子中，第一个参数是要判断的对象，第二个参数为要对比的类型。可以看到变量 a 和 float 浮点类型对比是相同的，所以 isinstance(a, float)返回了 True。但变量 b 和 float 是不同的所以返回 False。

2.9.6　取整函数

在量化交易中，对于数据取整处理是不可避免的，取整方式包括：向下取整、四舍五入取整、向上取整等等。Python 提供了很多浮点数取整的相关函数。

- int()函数向下取整
- round()函数四舍五入取整
- math 模块中的 ceil()方法向上取整

```
import math

def main():
    a = 3.14156
    Log(int(a))
    Log(round(a))
    Log(round(a, 3))
    Log(math.ceil(a))
```

输出结果为：

```
3
3
3.142
4
```

2.10 Python 异常处理

在编写 Python 策略时，避免不了出现错误，理想的情况是在策略启动时，通过 Python 自检发现错误。但实际上 Python 并不能主动找出所有的错误，有一些错误只有在运行过程中才能被发现，所有就需要用一种恰当的方式将错误源及信息呈现出来，并对错误进行修正以提高策略的健壮性。

2.10.1 语法错误

语法错误通常是初学者经常遇到的情况，例如少写了括号、布尔值 True 字符 T 需要大写等等，不过这种错误在 Python 启动时，通过对代码的解析会自动终止程序，并报出错误位置和信息。例如：

```
def main()
   Log(1)
```

输出结果为：

```
Traceback (most recent call last): File "<string>", line 1481, in Run File
"<string>", line 1 def main() ^ SyntaxError: invalid syntax
```

2.10.2 异常错误

异常错误比较隐蔽，通常在策略运行中才能被发现。例如：除法运算时，除数为 0。把一个值为空值的变量（None），当做字典使用。整型变量和字符串相加使用未定义的变量参与运算。例如：

```
def main():
   Log(10 / 0)
```

输出结果为：

```
Traceback (most recent call last): File "<string>", line 1481, in Run File
"<string>", line 7, in <module> File "<string>", line 2, in main
ZeroDivisionError: division by zero
```

2.10.3 异常捕获

为了检索隐藏的异常错误，或者为了避免异常错误的发生，导致正在运行的策略异常停止。可以使用 try…except 捕获异常。当执行 try 后的代码块时，如果发生异常错误，会被异常检测捕获，as 把 Exception 这个异常类型的错误信息附加到 as 后的 e 这个变量中。例如：

```
def main():
   try:
      Log(10 / 0)
   except Exception as e:
      Log('错误', e)
   Log('hello FMZ')
```

运行结果为：

```
错误 division by zero
hello FMZ
```

上面的例子，在程序运算 10/0 时，并没有引发程序停止，而是打印了一条日志，并且最后一条日志 Log("hello FMZ")也执行了。异常捕获不仅可以提示代码错误的原因，还可以防止程序因为异常导致终止运行。

2.11　温故知新

学完本章内容，读者需要回答：

1．Python 都有哪些数据类型？

2．Python 运算优先规则是什么？

3．哪个关键字能终止循环？

4．如何把时间戳转换为时间？

5．Type()函数与 isinstance()函数有什么区别？

在下一章中，读者会了解到：

1．量化软件基本操作

2．常用的行情和交易 API

3．绘制图表

第 3 章　认识发明者量化

学完第 1 章的量化交易基础，以及第 2 章的 Python 编程语言知识，就可以利用这些知识开发策略了。但只有量化平台的支持，才能支撑起整个量化工程。量化平台一般包括这几个功能：策略编写、策略回测、策略分析、模拟交易、实盘交易等等。本章将深入了解发明者量化平台。

本章主要涉及到的知识点有：

- 平台架构：对平台架构和功能有大致的了解。
- 配置交易所和托管者：打通模拟和实盘交易通道。
- 策略回测和运行：学会创建、回测、运行策略。
- 详解 API 和内置函数：学会调用各种 API，让函数为我所用。

3.1　平台简介

发明者量化交易平台(原 BotVS)是国内最专业的量化社区之一，创建于 2014 年。在该平台可以学习、编写、分享、买卖量化策略，在线回测和使用模拟盘模拟交易，运行、公开、围观实盘机器人。支持商品期货 CTP、易盛接口。

该平台适用于量化交易初学者，即使无基础也可以快速入门，平台功能强大灵活，也可以能满足进阶需要。支持使用 Javascript、Python、C++等主流编程语言，也支持可视化语言和 My 语言（兼容文华财经）实现策略。

3.1.1 平台架构

发明者量化交易平台系统，并不是单一的传统软件，而是一个集实盘、回测、分析工具、分布式部署、分享展示、社区交流、等多元一体的量化交易平台软件系统。该平台的架构特殊，具有很多优势。

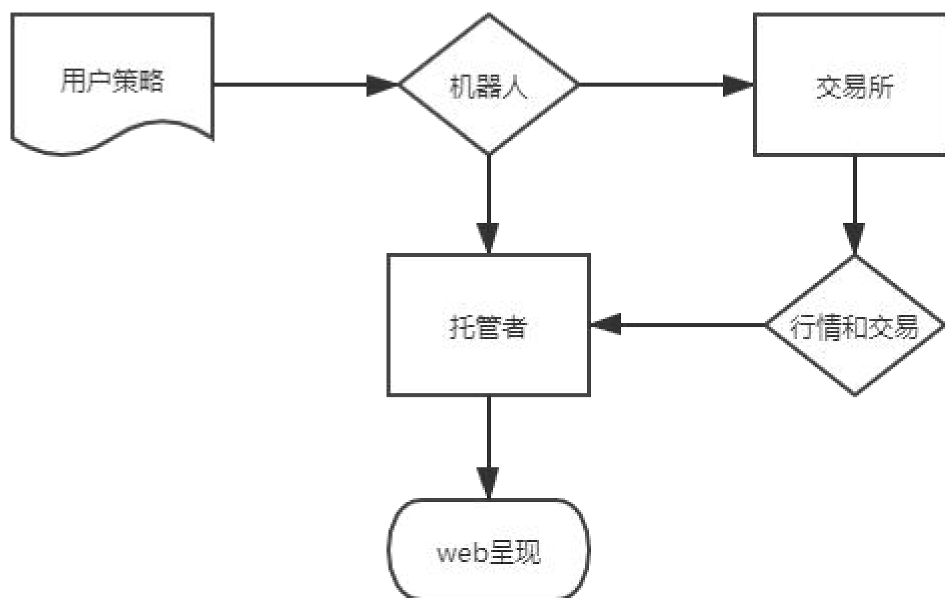

图 3.1　平台架构

特有的量化交易机器人，专门负责执行交易策略，机器人通常部署在用户自己的服务器或电脑的托管者程序上。另外托管者程序负责和发明者量化交易平台网站通信，进行传递日志、直接访问交易所获取行情和交易等任务。

如果发明者量化交易平台网站出现短暂问题，也不会影响策略机器人的执行。用户可以将托管者部署到任何地方来提高交易速度，通过网站或者手机随时随地的管理机器人、查看日志、修改参数。

注意：发明者量化聚焦于实盘交易，通过将计算密集和高延迟的任务从交易关键路径分离，以保证平台内部最低的延迟、最好的性能和最高的吞吐。

3.1.2 托管者程序

托管者是交易机器人的载体，是管理机器人的程序，主要负责处理底层数据、访问接口、通信等任务。托管者程序有适用于主流操作系统的各个版本，例如：

- Linux 32 位/64 位操作系统
- Windows 32 位/64 位操作系统，命令行版/界面版

- Mac OSX 操作系统（苹果电脑操作系统）
- ARM Linux 操作系统

托管者如何和发明者量化交易平台通信？在部署运行托管者程序时，需要输入一个地址，该地址即为发明者量化交易平台通信的地址，这个地址中有一个用户唯一标识 ID，地址输入完以后，还需要另外一个配置，配置上发明者量化交易平台账号的密码。使用这两样信息来让托管者程序和发明者量化交易平台上的账户对应起来。例如 windows 界面版托管者配置时，如图：

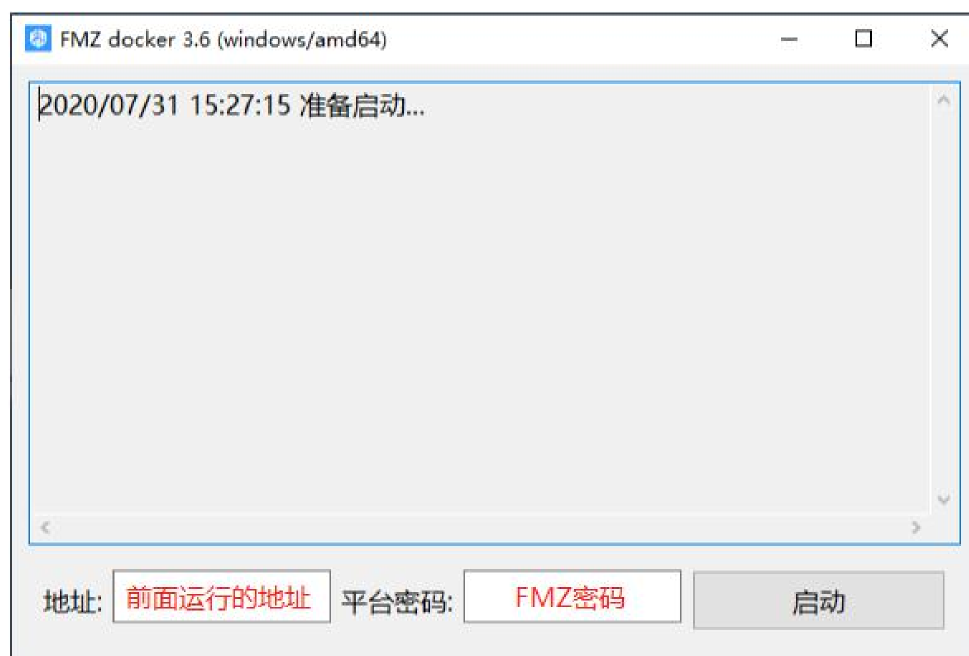

图 3.2　fmz 托管者程序

3.1.3　平台功能

- 控制中心：用户的操作界面。
- 策略：用户公开和出售的策略在这里，可以点击相应的标签筛选某一类的策略。
- 围观：用户公开运行的机器人，可以进行围观评论。
- 文库：发明者量化交易平台官方出品的一些精品文章。
- 社区：用户发帖提问交流平台。
- 众包：发布需求和代写策略，由用户自行联系，发明者量化交易平台官方不担保。
- 产品：发明者量化交易平台提供的面向机构或者交易所的产品。
- API 文档：编写策略所需要的 API 介绍，关于 API 有问题可以在这里搜索。
- 工具：发明者量化交易平台开发的工具，包括行情图表和一个简单行情计算器。

图 3.3　fmz 控制中心

3.1.4　机器人

机器人是交易策略的执行者，一个机器人只能执行一个策略，但一个策略可以给多个机器人执行。下图是：发明者量化平台机器人页面，用来管理托管者上的机器人、显示机器人的运行状态。

图 3.4　fmz 机器人

1、点击创建新的机器人。
2、机器人的名字，点击可进入此机器人管理页面。
3、此机器运行的策略名称，点击可进入策略编辑页面。
4、机器人的排序方式，可以根据名称、策略、状态、创建时间、盈利等排序。

5、机器人状态，有正在运行、停止、出现错误三种状态。
6、公开自己的机器人，可以在围观中让其它用户看到。
7、停止或重启机器人。
8、搜索机器人。
9、当前机器人可运行的时间。

3.1.5 策略库

顾名思义策略库是存放策略的仓库，同时策略库也是编程和回测策略的入口，不仅可以在策略库中新建策略，还可以分别对不同的策略进行分组。对于一些优质策略可以以出租或出售的方式授权分享给别人。

注意：可以在授权管理中，分别管理策略和机器人授权。

图 3.5 fmz 策略库

1、新建策略
2、注册码管理，管理出售或分享策略的注册码，接下来会详细介绍。
3、策略分组。
4、策略名称。
5、添加和管理新的分组，可以把不同类型的策略分组，方便管理。

3.1.6 交易终端

交易终端包含图表、账户、下单等功能，可以通过终端就可以撤单，下单，查看行情账户等。丰富的图表功能内置多种技术分析指标，全品种数据任意调用，毫秒级交易延迟，快速完成期货交易。

图 3.6　fmz 交易终端

3.1.7　策略编写界面

点击策略库就可以进入策略编辑界面，该界面主要包含两大功能：策略编写和策略回测。

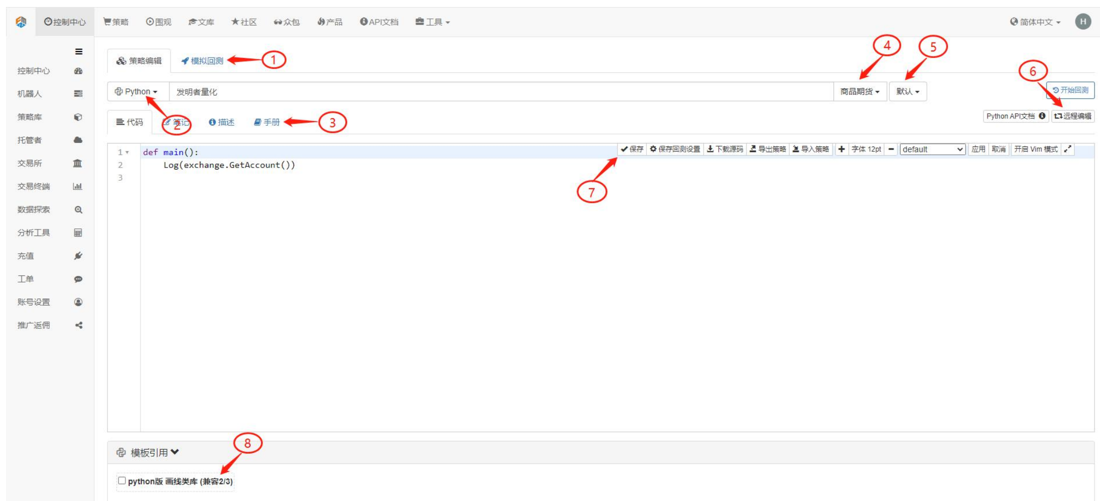

图 3.7　fmz 策略编辑

1、点此进入回测，关于回测的具体说明将在后续教程中讲解。

2、策略使用的语言选择，策略创建后不可切换语言。

3、笔记是编写策略的记录，仅自己可见；描述是策略说明，策略公开后其他人可以在策略页面看到；手册是策略的使用说明，购买或复制策略的可以看到。

4、策略类型，分为通用策略和模板，关于模板以后教程会有详细说明。

5、策略分组。

6、远程编辑，包含常用的代码编辑器插件，可在本地编写策略，自动同步到发明者量化交易平台。后续教程中会有讲解。

7、保存，可在编辑状态使用 Ctrl+S 的快捷键。

8、选择要引用的模板，要先在策略广场把模板复制下来。

3.1.8　账户管理

图 3.8　fmz 账号设置

1、消息中心：各种消息汇总，包括论坛回帖、工单信息等消息。

2、身份认证：身份认证即身份主页，可以修改头像，及设置名字和简介。

3、修改密码：重置账户密码。

4、谷歌验证：最好开启谷歌二次验证，提高安全性。

5、额度预警：当账户可用余额小于此值时，您将会收到邮件与微信通知，设置为 0 禁止使用此功能，没有充值或者更改此设置的情况下，24 小时内只通知一次

6、推送设置：这里可以绑定微信，Telegram，邮箱等，用于接收机器人的推送消息。

7、API 接口：FMZ 也有自己的 API，基本上机器人的所有操作都可以通过 API 完成，提供了丰富的扩展性。

8、子账户组：可以创建只有部分权限的子账户功能，可用于策略修改共享账户等作用。

注意：推送功能需要预先绑定相关账号。

3.2 配置交易所和部署托管者

在本节内容中，我们首先学习如何添加配置交易所和部署托管者程序。在发明者量化交易平台上做量化交易，运行量化交易机器人，需要三个必要条件，与之对应的操作就是：

1、部署托管者程序（托管者程序是机器人的载体，托管者程序负责和期货公司前置机交互，处理底层数据、负责和发明者网站通信等）。

2、在发明者量化交易平台上配置交易所（商品期货账号、密码、期货公司前置机信息）

3、所需要运行的交易策略（包括交易策略代码，参数配置信息等）。

3.2.1 配置交易所

图 3.9 fmz 交易所

在控制中心页面，点击交易所就可以跳转到交易所页面。然后点击“添加交易所”按钮，即可添加期货账户信息。

图 3.10 fmz 添加交易所（1）

例如：选择 CTP 协议，选择宏源期货主席（看穿式监管），选择对应的网络节点后，系统会自动匹配行情服务器和交易服务器，以及 Broker ID，剩下的只需要填入账号密码即可。

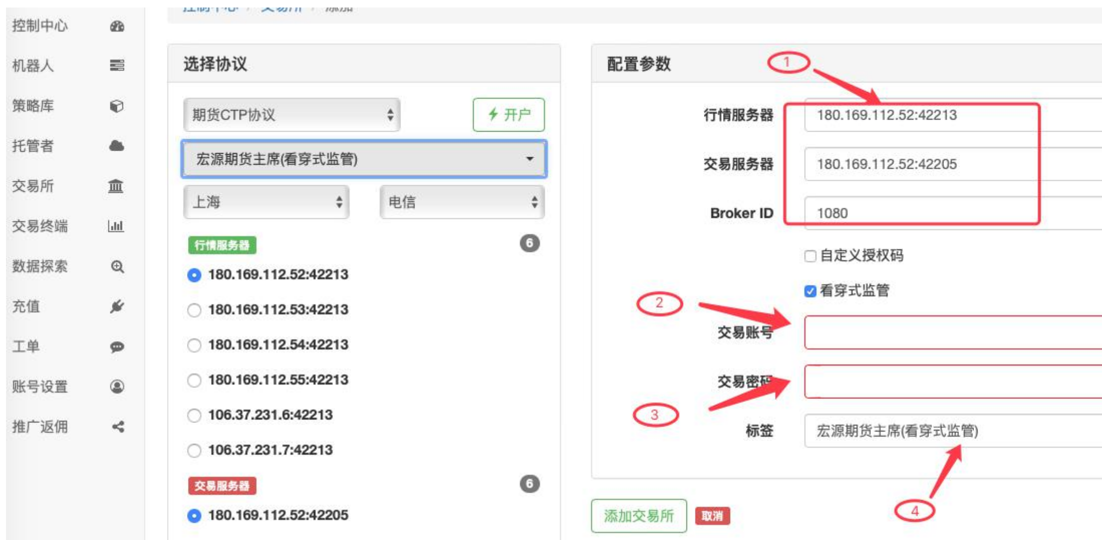

图 3.11 fmz 添加交易所（2）

1、 在选择了期货公司之后，行情服务器、交易服务器、Broker ID 会自动填充，也可以根据自己的网络，选择期货公司其它的线路节点。通常不用设置，直接使用默认。
2、 配置期货公司资金账号（CTP 登录验证）。
3、 配置期货公司资金账号的密码（CTP 登录验证）。
4、 设置标签，可以使用默认显示的，如果需要区分不同的账户，可以自行修改。

注意：如果需要配置自定义授权码，勾选配置即可，发明者量化交易平台已经对以下期货公司自动配置，就不用勾选了，只用配置对所在期货公司的资金账号和密码即可。

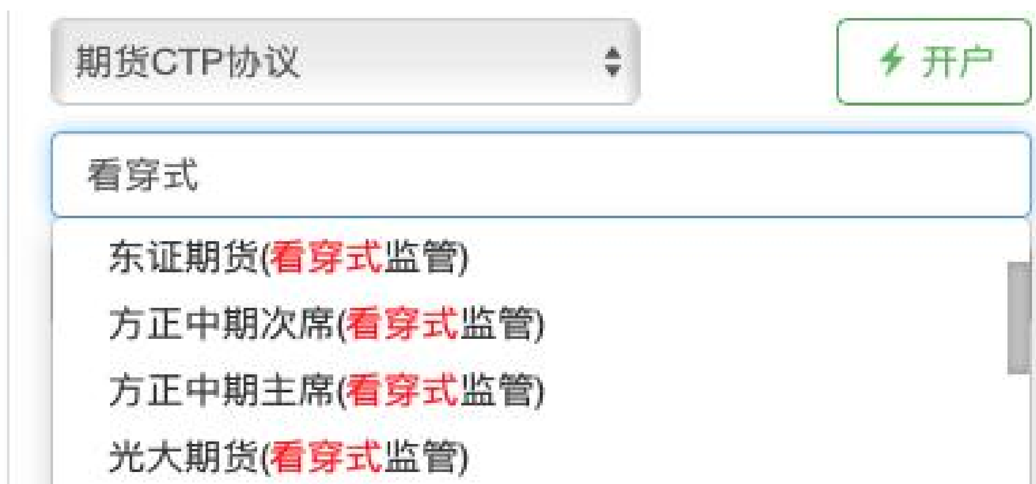

图 3.12 fmz 添加交易所（3）

或者可以直接在配置时搜索“看穿式”查询，选择对应的交易所。添加好期货公司配置信息后，在交易所页面会出现已经添加好的信息。

图 3.13 fmz 添加交易所（4）

3.2.2 在 Windows 上部署托管者

部署托管者有两种方式：即通过发明者量化交易平台一键部署到阿里云服务器上。也可以手动部署在自己的设备、服务器、电脑上。在控制中心，点击“托管者”，跳转到托管者管理页面。点击按钮“部署托管者”，跳转到部署托管者页面。

图 3.14 fmz 添加托管者（1）

托管者程序支持多种操作系统，可以部署到各种设备上，我们以 windows 系统为例，看下如何手动部署一个托管者程序。

- Windows
- Linux

- ❑ Mac
- ❑ ARM Linux

图 3.15 fmz 添加托管者（2）

可以选择对应操作系统的托管者，下载后解压缩运行。然后复制粘贴“界面版”字样后的地址信息。Windows 所对应的地址是：node.fmz.com/xxxxxxx，“x”是数字，每一个账户分别对应不同的数字。

图 3.16 fmz 添加托管者（3）

填入运行地址，填入当前发明者量化交易平台账号的密码，点击启动，即可运行。如果显示如下信息，说明托管者正常启动了。

2019/XX/XX 12:03:30 Login OK, SID: 9XXXX6, PID: 3XXXX6

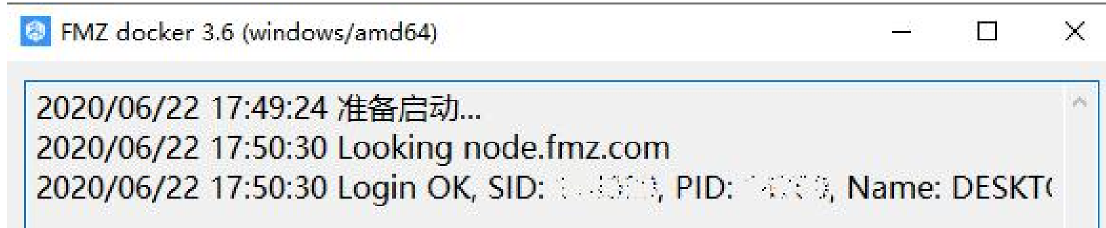

图 3.17　fmz 添加托管者（4）

注意：一个设备或者服务器上可以运行多个托管者，一个托管者上可以运行多个机器人。即使配置最低的阿里云服务器上，同时运行 6~7 个机器人也是没有压力的。部署托管者、配置交易所后，不要轻易修改发明者量化交易平台的账号密码，修改密码会导致配置的交易所配置失效，需要重新配置交易所信息、重启托管者才能正常使用（已经运行起来的没有影响）。

3.2.3　在 linux 上部署托管者

相对来说，在 linux 系统上部署比在 windows 系统复杂一点，但如果想长期从事量化交易，建议使用便宜又好用的 Linux 服务器。

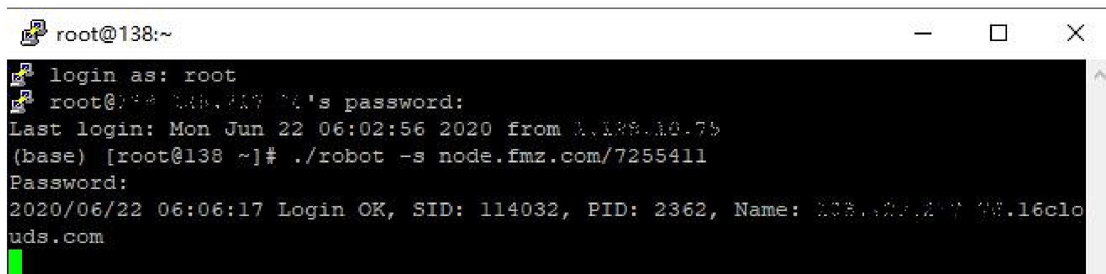

图 3.18　fmz 添加托管者（5）

1、 购买服务器，一般选择 CentOS 系统，最低配置即可。
2、 登录服务器，windows 推荐用 Xshell 客户端，SSH 登录。
3、 下载托管者，如同上例中部署 windows 系统的托管者时，下载 linux 版本的即可。根据 linux 系统的位数具体选择下载，此处以 64 位为例。服务器输入：wget https://www.fmz.com/dist/robot_linux_amd64.tar.gz 下载，如提示 wget 不存在，运行 yum install wget -y 安装 wget.
4、 运行 tar -xzvf robot_linux_amd64.tar.gz 解压。
5、 测试托管者运行，在解压后的目录中运行命令：
    ./robot -s node.fmz.com/2XXXX8 -p 密码

“运行地址”和 windows 例子中的一样。-p 后面输入空格，然后跟上发明者量化交易平台账户的密码。2XXXX8 这里是举个例子，每个账户的地址都不一样。提示如：201X/XX/XX 05:04:10 Login OK, SID: 62086, PID: 7226, Name: host.localdomain 则运行成功，如果遇到权限问题，运行 chmod +x robot 后重试。

6、托管者运行在前台关闭 SSH 连接即断开需要在后台运行，按 ctrl + C 结束测试。

7、后台运行命令：nohup ./robot -s node.fmz.com/2XXXX8 -p 密码 & 。当然也可以使用 linux 系统的 screen 软件工具，在后台运行托管者。部署完之后就可以在托管者页面看到部署的托管者。

3.2.4 一键租用托管者

图 3.19 fmz 添加托管者（6）

图 3.20 fmz 添加托管者（7）

对于网络稳定或者没有稳定的电脑，觉得操作起来比较麻烦，可以选择一键租用托管者，点击“一键租用托管者”，然后点击“立即租用”，输入发明者量化交易平台账户密码后，即可自动化部署。稍等几分钟后，就正常运行了。

图 3.21　fmz 添加托管者（8）

图 3.22　fmz 添加托管者（9）

一键租用托管者采用阿里云的 Linux 服务器，内置 Python3 版本，以及常用的第三方 Python 库，包括：

- TA-Lib
- Numpy
- Matplotlib

注意：一键租用托管者不支持机器学习 Tensorflow 库。

3.3　创建管理策略和机器人

当配置好交易所、部署好托管者之后，就只剩下选择所需要让机器人执行的交易策略了。本节学习如何创建管理策略和机器人。机器人可以理解为以托管者程序为底层支持，在托管者上运行起来的策略实例，操作控制配置好的期货账户（通过交易所配置信息）。

3.3.1　创建策略

策略创建方式有 2 种：新建策略和在策略广场中复制别人分享的开源策略。我们先来看创建一个新策略，点击“新建策略”按钮：

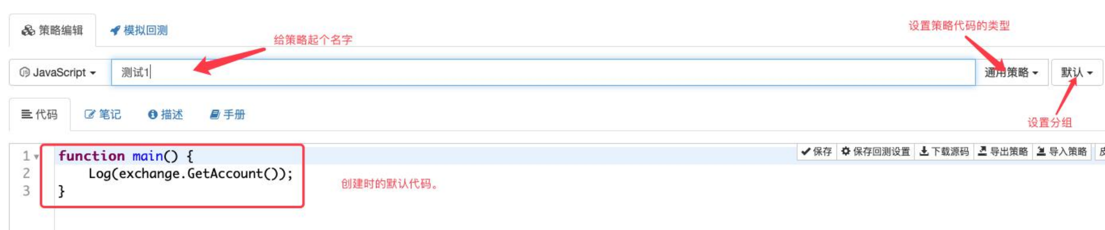

图 3.23　fmz 创建策略（1）

填入策略名称，点击保存之后，会显示在“策略库”页面。也可以在页面右边选择策略类型和设置策略分组。

图 3.24　fmz 创建策略（2）

也可以从策略广场复制策略，创建到自己的策略库中。点击“策略”按钮会跳转到策略广场页面，选中一个策略点击策略名字，跳转到策略描述页面。跳转到策略描述页面后向下滑动，滑动到如下位置，如图：

图 3.25　fmz 创建策略（3）

图 3.26 fmz 创建策略（4）

注意：复制的策略在实盘运行前请务必检测此策略代码的安全性。

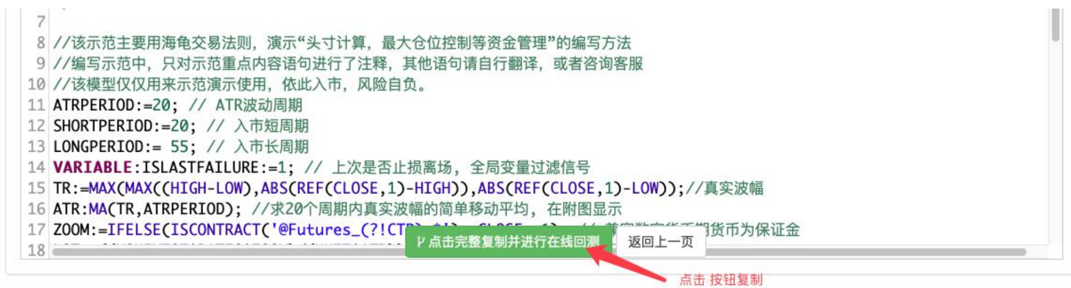

图 3.27 fmz 创建策略（5）

点击后，会自动跳转到该策略的编辑页面，可以修改策略或者直接点击保存，即可保存到当前账号的策略库中。

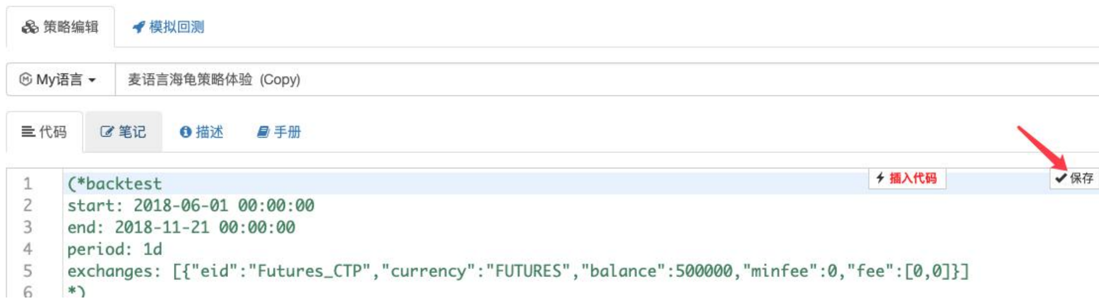

图 3.28 fmz 创建策略（6）

3.3.2 管理策略

图 3. 29 fmz 管理策略（1）

图 3. 30 fmz 管理策略（2）

管理策略时，可以预先设置一些分组，例如设置一个名为“MY 语言”的分组，设置过分组后，就可以点击策略名称，拖动，放入设置好的分组标签中，例如：

图 3. 31 fmz 管理策略（3）

创建分组既是创建策略标签，在这个分组中就可以看到刚才从策略广场复制创建的策

略了。可以点击策略名称，进入策略编辑页面。也可以点击右侧“操作项”，对策略库中的策略，复制、删除、公开、售卖、创建机器人运行。

图 3.32　fmz 管理策略（4）

其中，点击“编辑”选项可以跳转至该策略编辑页面。点击“公开”可以将策略以两种方式分享，内部分享：设置时间、复制次数等信息之后，会生成一个复制码和复制地址，可以通过打开这个复制地址，输入复制码获取策略。公开分享：点击公开分享后，策略会显示在策略广场中，其它用户可以复制获取。

点击“售卖”可以以三种方式进行出售，内部出售：和复制码类似，点击“内部出售”按钮后，设置使用时间，并发机器人个数等信息之后，会生成一个注册码和注册地址，租用该策略的用户拿到这个注册码、注册地址后，打开这个注册地址，输入注册码，即可获得该策略的使用权限，但是无法看到策略代码。软件注册：用于发明者量化交易平台 PC 端的启动。公开出售：申请在策略广场上架，需要审核，并且具备上架条件。

注意：对于出售、分享等操作创建的注册码、复制码等信息可以从“授权管理”页面进行管理。

图 3.33　fmz 管理策略（5）

3.3.3 创建机器人

图 3.34 fmz 创建机器人（1）

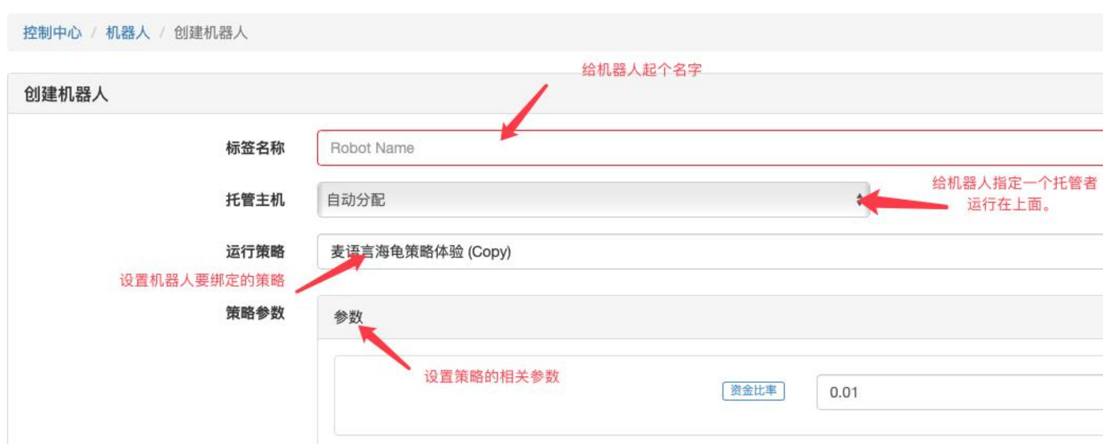

图 3.35 fmz 创建机器人（2）

如果当前已经配置好了交易所配置信息，已经部署了托管者，策略库中直接创建或者复制过来创建了策略。点击“创建机器人”按钮，即可跳转至创建机器人页面。设置好以上内容，我们往下滑动，需要给机器人继续配置：

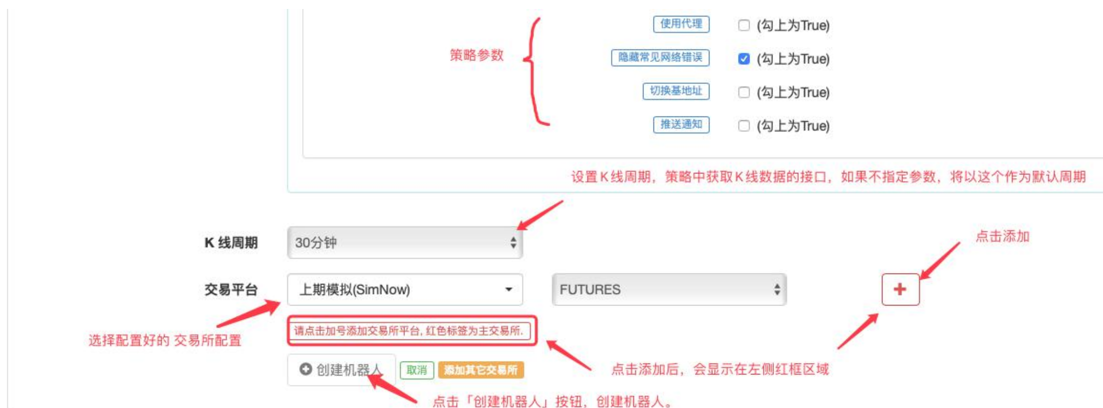

图 3.36 fmz 创建机器人（3）

机器人配置设置完毕后，点击“创建机器人”按钮后，就可以跳转到机器人管理页面，在这个页面显示了机器人的运行概况。如下图所示：

图 3.37　fmz 创建机器人（4）

也可以点击机器人名称，进入机器人运行页面，了解机器人详细情况。一般情况下在编写策略时，需要 Log 一些必要信息或数据，方便在机器人运行时查看。

图 3.38　fmz 创建机器人（5）

由于机器人没有连接到期货公司前置机（未开市时间），图表信息，状态栏信息都没有显示出来，这篇教程学习完之后，可以在开市时间动手尝试下运行一个机器人。

3.3.4　管理机器人

对运行中的机器人可以开启监控功能：如下图所示，在状态栏中，可以显示每个机器人运行的状态，以及每个机器人具体的盈利额度：

图 3.39　fmz 管理机器人（1）

图 3.40　fmz 管理机器人（2）

有时候策略会因为某些原因出现错误，导致机器人停止，错失开平仓机会，可以在创建机器人后，点击“监控”按钮，一旦出现机器人停止，邮箱与绑定的微信将收到通知。

图 3.41　fmz 管理机器人（3）

图 3.42　fmz 管理机器人（4）

- 点击停止按钮：机器人会停止，操作按钮也会发生变化。
- 点击删除按钮：点击后，确定删除之后，会删除机器人。

- ❑ 点击重启按钮：点击后，会重启该机器人。

注意：以上是手动管理机器人的使用方法，除此之外发明者量化还开放了扩展 API，可以使用程序批量自动管理机器人。

3.4　模拟级和实盘级回测系统

回测就像沙盘推演，它是量化交易中不可获取的环节，回测的目的是：过滤策略代码 BUG、检验策略逻辑是否有效等等，本节将介绍发明者量化 2 种回测方式。

3.4.1　模拟级别回测系统

图 3.43　回测配置（1）

发明者量化交易平台提供的模拟级别回测系统是基于 on Tick 回测机制运行，即：行情接口数据随着回测中时间序列上移动而逐个放出。把策略程序放在一个行情数据随着回测时间流动而实时变化的沙盒环境中运行。区别于 on bar 机制，即：新 K 线 BAR 出现时，才让策略程序运行一次。回测参数配置上可以设置：

图 3.44　回测配置（2）

1、回测时间范围：确定了回测时间区间。

2、默认 K 线周期：程序默认的 K 线周期大小。

3、日志、收益、图表条数：设置回测系统打印信息数量。

4、底层 K 线周期：模拟级别回测时用于控制回测数据粒度大小的周期参数。

5、滑点：成交撮合时的模拟滑点。

6、容错：在容错模式回测时，模拟发生错误的概率设置。点击开始回测按钮右侧小三角可以进行容错回测。

7、延迟：模拟网络延迟，让数据返回延迟一定时间。

8、柱长：K 线数据的 Bar 数量上限。
9、手续费相关：手续费费率设置。
10、深度、市场成交数据：设置深度档位、是否需要分笔数据。
11、设置市场类型：设置回测市场类型。
12、添加回测配置按钮。

注意：以上设置，使用鼠标放置在页面控件上，会显示详细说明。

3.4.2 底层 K 线周期

模拟级别回测是按照回测系统的底层 K 线数据，按照一定算法在给定的底层 K 线 Bar 的最高价、最低价、开盘价、收盘价的数值构成的框架内，模拟出 ticker 数据插值到这个 Bar 的时间序列中。实盘级别回测没有底层 K 线选项（因为 ticker 数据都是真实的，不用底层 K 线来模拟生成）。模拟级别回测中，基于 K 线数据模拟生成的 ticker。这个 K 线数据就是底层 K 线。在实际使用模拟级别回测中，底层 K 线周期必须小于设置的默认 K 线周期。否则，由于底层 K 线周期较大，生成的 ticker 数量不足，调用 API 获取指定周期的 K 线时，数据会有失真的情况。在使用大周期 K 线回测时，可以适当调大底层 K 线周期，以增加回测速度。底层 K 线如何生成模拟 tick 数据的相关链接帖子：

https://www.fmz.com/bbs-topic/662

可以理解为底层 K 线周期控制着 tick 的粒度，底层 K 线周期 1 分钟或者 5 分钟，生成的 tick 数据粒度（密度）肯定是不一样的。粒度越小价格变动跳跃越小，回测精细度越高。但是回测时间会相对较长。如果比较追求回测速度，可以适当调大底层 K 线周期，以损失一点回测精细度，来加快回测速度。对于中低频趋势策略来说影响不大。

3.4.3 实盘级别回测系统

图 3.45 回测配置（3）

实盘级别回测使用真实的 ticker 级别数据。对于基于 ticker 级别数据回测的策略来说，使用实盘级别回测更贴近真实环境。实盘级别回测，ticker 数据是真实记录的数据，并非模拟生成。在回测时逐个 ticker 数据放出，回测粒度较为精细。回测耗时也相对较多。适合交易频率相对较高的策略、基于盘口数据计算的策略等。实盘级别回测，除了提供 tick 数据，

还支持真实的深度数据回放，回放深度快照数据用于回测一些基于订单薄的策略。同样也快照了市场上逐笔成交数据。

由于实盘级别回测，数据量特别大。所以只能支持一定时间范围内的回测，时间范围不能选择太大，否则会超出数据承载限制。实盘级别回测只支持有限的几个交易市场，具体可以参看回测页面上的选项说明。

3.5 全局常量和数据结构

在发明者量化交易平台中，有很多 API 函数，每个函数都有各自的功能，它们返回的数据也不尽相同。通过本节对全局常量和数据结构的学习，可以知道这些函数返回的结果都有哪些意义。

3.5.1 exchange 交易所对象

exchange 交易所对象是在编写策略时最常用的，因为绝大多数 API 函数都是该对象的方法。exchange 交易所对象在策略代码中就代指了在创建机器人时或者回测时，添加的交易所。这些已经添加到平台的交易所，在添加时都绑定了交易所的 API KEY（访问密钥）或者资金账号、资金密码（对于商品期货）。

所以在使用例如：exchange.GetAccount()函数获取账户信息时，可以访问到对应 API KEY 或者资金账号、资金密码的账户的信息数据（API KEY 即授权程序访问接口的密钥，验证资金账号、资金密码即登录验证的凭证）。exchange 即添加的第一个交易所对象，如下图回测系统和实盘机器人：

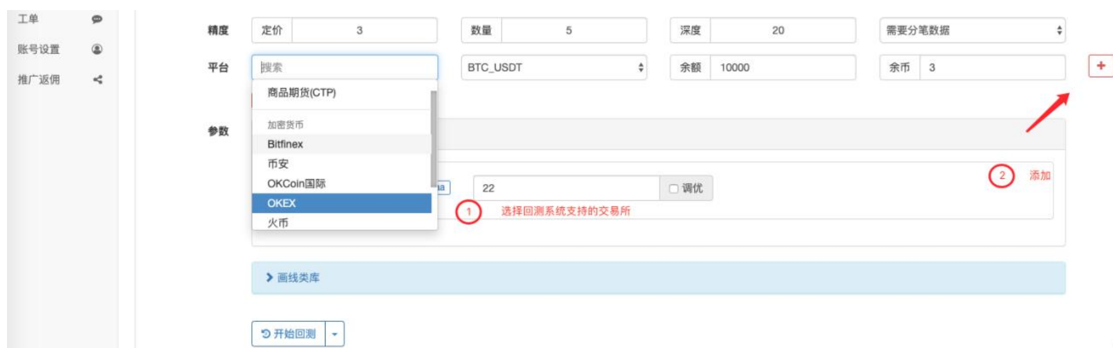

图 3.46　回测配置（4）

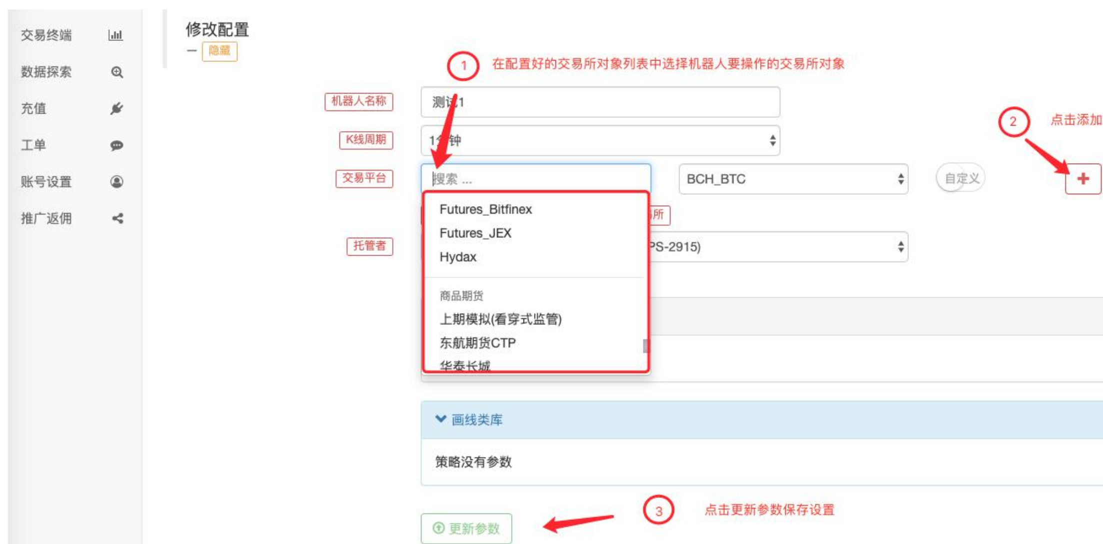

图 3.47　交易所对象（1）

在添加了一个交易所对象之后，可以在策略代码中写入代码，打印该交易所对象的名字、标签信息，回测实盘均可：

```
def main():
    Log("实盘机器人页面或者回测页面上，添加的第一个交易所对象名字：",
        exchange.GetName(), "，标签：", exchange.GetLabel())
```

3.5.2　exchanges 交易所对象列表

学习完了 exchange 的概念，exchanges 的概念就更加容易理解了。exchanges 就是一系列的交易所对象放在一个列表中（数组），因为在发明者量化交易平台上一个策略可以设计成多交易所、多账户的架构，所以可以添加多个交易所对象。exchange 的数组，包含多个交易所对象，exchanges[0]即是 exchange，如图：

图 3.48　交易所对象（2）

添加的交易所对象对应策略代码中的 exchanges[0]、exchanges[1]、exchanges[2]... ，以

此类推。同样，在实盘或者回测可以使用以下代码测试，遍历 exchanges 交易所对象数组，逐个打印交易所对象的名称、标签信息。

```
def main():
    for i in range(len(exchanges)):
        Log("添加的交易所对象索引（第一个为 0 以此类推）：", i, "名称：",
            exchanges[i].GetName(), "标签：", exchanges[i].GetLabel())
```

3.5.3 Order 结构

订单结构，由 exchange.GetOrder()，exchange.GetOrders()函数返回。发明者量化平台定义、封装的订单数据结构，该结构中属性：Status 、Type 、Offset 的值为固定的几种取值。

```
{
    Info: {...},             // 请求交易所接口返回的原始数据，回测时无此属性
    Id: 123456,              // 交易单唯一标识
    Price: 1000,             // 下单价格
    Amount: 10,              // 下单数量
    DealAmount: 10,          // 成交数量
    AvgPrice: 1000,          // 成交均价
    Status: 1,               // 订单状态
    Type: 0,                 // 订单类型
    Offset: 0,               // 订单的开平仓方向
    ContractType: ""         // 订单的合约代码
}
```

- Info 属性：接口返回的原始数据即封装之前的数据内容。
- Id 属性：订单的 ID，用于取消某个订单，查询某个订单时用作参数。
- Price 属性：订单的委托价格。
- Amount 属性：订单的委托数量。
- DealAmount 属性：订单成交部分的数量。
- AvgPrice 属性：订单的成交均价。
- Status 属性：订单状态属性，例如挂单状态、完全成交状态、撤销状态。
- Type 属性：标记订单是买单还是卖单。
- Offset 属性：在期货交易时，标记订单是开仓单还是平仓单。

以下表格定义了 Order 结构中的 Status 属性，分别表示了订单未完成、已完成、已取消、未知状态等 4 个值。

常量名	定义	值
ORDER_STATE_PENDING	未完成	0
ORDER_STATE_CLOSED	已完成	1
ORDER_STATE_CANCELED	已取消	2
ORDER_STATE_UNKNOWN	未知状态	3

在编写代码时，可以直接使用 ORDER_STATE_PENDING 判断订单状态，因为 ORDER_STATE_PENDING 非常容易看明白，判断的状态是挂单状态，而用 0 则非常不直

观。

```
# 第一种写法
if order["Status"] == ORDER_STATE_PENDING:
    Log("订单状态值:", order["Status"])

# 第二种写法
if order["Status"] == 0:
    Log("订单状态值:", order["Status"])
```

以下表格定义了 Order 结构中的 Type 属性，分别表示订单为买单、订单为卖单，当订单为买单时值为 0，当订单为卖单时值为 1。

常量名	定义	值
ORDER_TYPE_BUY	订单为买单	0
ORDER_TYPE_SELL	订单为卖单	1

下面的表格定义了 Order 结构中的 Offset 属性，分别表示订单开仓方向、订单平仓方向，当订单为开仓方向时值为 0，当订单为平仓方向值为 1。

常量名	定义	值
ORDER_OFFSET_OPEN	订单为开仓方向	0
ORDER_OFFSET_CLOSE	订单为平仓方向	1

注意：GetOrder 需要传入订单号参数，获取的是某一个订单的 Order 结构体。而 GetOrders 不需要传入参数，获取所有未完成的订单。返回值的是 Order 结构体数组。如果当前交易对没有挂单时，调用 exchange.GetOrders()返回空数组，即：[]。

3.5.4 Position 结构

Position 结构是期货交易中的持有仓位信息，由 exchange.GetPosition()函数返回此结构数组。

```
{
    Info: {...},                // 请求交易所接口返回的原始数据，回测时无此属性
    MarginLevel: 10,            // 杆杠大小
    Amount: 100,                // 持仓量
    FrozenAmount: 0,            // 仓位冻结数量
    Price: 10000,               // 持仓均价
    Profit: 0,                  // 持仓浮动盈亏
    Type: 0,                    // 持仓方向
    ContractType: "quarter",    // 持仓的合约代码
    Margin: 1                   // 仓位占用的保证金
}
```

- Info 属性：同上。
- MarginLevel 属性：持仓的杠杆数值。
- Amount 属性：该仓位持仓数量。
- FrozenAmount 属性：仓位冻结数量。

- Price 属性：持仓均价，持仓后，加仓会影响该值。
- Profit 属性：持仓盈亏。
- Type 属性：仓位类型，多头仓位、空头仓位。
- ContractType 属性：合约代码。
- Margin 属性：保证金。

GetPosition()函数返回的是 Position 结构数组，其中的 Type 属性代表持仓方向，PD_LONG 代表多头仓位，PD_SHORT 代表空头仓位，PD_LONG_YD 代表昨日多头仓位，PD_SHORT_YD 代表昨日空头仓位，如下表所示：

常量名	定义	值
PD_LONG	多头仓位	0
PD_SHORT	空头仓位	1
PD_LONG_YD	昨日多头仓位	2
PD_SHORT_YD	昨日空头仓位	3

注意：GetPosition 函数获取的是所有持仓品种的持仓信息,如果没有持仓返回空数组，所以引用前要先判断。

3.5.5 Trade 结构

获取所有交易历史(非自己)，由 exchange.GetTrades()函数返回。这个是整个市场最近时间的成交记录。

```
{
    Time: 1567736576000,     // 时间(Unix timestamp 毫秒)
    Price: 1000,             // 价格
    Amount: 1,               // 数量
    Type: 0                  // 订单类型
}
```

- Time 属性：毫秒时间戳，记录市场上这笔成交的时间。
- Price 属性：市场上这笔成交记录的成交价格。
- Amount 属性：市场上这笔成交记录的数量。
- Type 属性：标记这笔成交是买单主动成交，还是卖单主动成交。

3.5.6 Ticker 结构

市场行情由 exchange.GetTicker()函数返回。在商品期货中每秒返回 2 个 Tick 数据，通常指的是盘口数据。

```
{
    Info    : {...},             // 请求交易所接口返回的原始数据，回测时无此属性
    High    : 1000,              // 最高价
    Low     : 500,               // 最低价
    Sell    : 900,               // 卖一价
```

```
    Buy    : 899,                // 买一价
    Last   : 900,                // 最后成交价
    Volume : 10000000,           // 最近成交量
    Time   : 1567736576000       // 毫秒级别时间戳
}
```

- Info 属性：同上。
- High 属性：一般为 24 小时内的最高价。
- Low 属性：一般为 24 小时内的最低价。
- Sell 属性：当前的卖一价格。
- Buy 属性：当前的买一价格。
- Last 属性：当前的最新成交价。
- Volume 属性：最新成交量。
- Time 属性：毫秒级别时间戳，用于标记时间。

3.5.7 Record 结构

标准 OHLC 结构数据包含了开盘价、最高价、最低价、收盘价、成交量、时间等数据，它是组成 K 线的最基本数据，由 exchange.GetRecords()函数返回此结构数组。其中每个 Record 结构数据都代表一个 K 线。

```
{
    Time: 1567736576000,         // K 线时间戳
    Open: 1000,                  // 开盘价
    High: 1500,                  // 最高价
    Low: 900,                    // 最低价
    Close: 1200,                 // 收盘价
    Volume: 1000000              // 交易量
}
```

- Time 属性：K 线 BAR 起始时间戳，比较这个 K 线柱的起始时间。
- Open 属性：开盘价。
- High 属性：最高价。
- Low 属性：最低价。
- Close 属性：收盘价。
- Volume 属性：成交量。

3.5.8 Depth 结构

市场深度，由 exchange.GetDepth()函数返回。返回值是 Depth 结构体，结构体包含两个结构体数组，分别是 Asks[]和 Bids[]，其中每个数组中包含价格 Price、数量 Amount 以及时间戳 Time。

```
{
```

```
    Asks    : [...],              // 卖单数组,MarketOrder 数组,按价格从低向高排序
    Bids    : [...],              // 买单数组,MarketOrder 数组,按价格从高向低排序
    Time    : 1567736576000       // 毫秒级别时间戳
}
```

Depth 数据结构的 Asks 键，为卖单列表，列表中每个数据均为 MarketOrder 数据。Depth 数据结构的 Bids 键，为买单列表，列表中每个数据均为 MarketOrder 数据。比如我要获取卖二价格：

```
def main():
    depth = exchange.GetDepth()
    price = depth["Asks"][1].Price
    Log("卖二价为:", price)
```

3.5.10 Account 结构

Account 结构是由 exchange.GetAccount()函数返回的账户信息，主要包含 3 个数据：账户余额、账户冻结余额，以及请求交易所接口返回的原始数据。

```
{
    Info: {...},          // 请求交易所接口返回的原始数据，回测时无此属性
    Balance: 1000,        // 账户余额
    FrozenBalance: 0,     // 账户冻结的余额
    Stocks: 1,            // 无效
    FrozenStocks: 0       // 无效
}
```

- Info 属性：同上。
- Balance 属性：可用资金数量，CTP 商品期货中，该属性为可用钱数。
- FrozenBalance 属性：冻结资金数量，如果下单后，订单未成交，则冻结该订单用于交易的资金，FrozenBalance 即为冻结的资金数量。
- Stocks 属性：无效。
- FrozenStocks 属性：无效。

3.5.11 策略参数

在策略代码中，在策略界面上设置的策略参数，是以全局变量形式体现，JavaScript 语言中可以直接访问策略界面上设置的参数数值或者修改，python 策略中修改全局变量时需要使用 global 关键字。策略参数一共分为五种：

变量	类型	默认值	说明
number	数字型	1	数字
string	字符串	Hello FMZ	输入无需加双引号，输出字符串
bool	布尔型	true	勾选为 true，不勾选为 false
combox	下拉框	1|2|3	通过下拉菜单方式选择参数
secreString	加密字符串	passWord	字符串被加密发送

注意：在策略代码中均以全局变量形式使用，“下拉框”类型的参数，参数值为选定的下拉框中项目的索引，索引从 0 开始。其它策略参数设置、功能，例如：策略参数分组，策略参数条件依赖、策略参数代码化保存在策略等。详见发明者量化交易平台 API 文档：https://www.fmz.com/api#%E7%AD%96%E7%95%A5%E5%8F%82%E6%95%B0

3.6 获取 Tick、深度、历史 K 线数据

在商品期货量化交易中需要用到很多不同类型的数据，如：交易所原始 Tick 数据、订单薄深度数据，以及最常用的 K 线数据。让我们看看如何用 API 函数获取这些数据，以及常用的商品期货策略框架。

3.6.1 exchange.GetTicker()

Tick 数据俗称交易快照，交易所内的数据就像河流一样，而交易快照就是这个河流某个截面，国内商品期货是每秒 2 个 Tick 数据。exchange.GetTicker()函数获取实时 tick 数据，返回 Ticker 结构。回测系统中该函数返回的 Ticker 数据中 High、Low 为模拟值，取自当时盘口的卖一、买一。实盘时则是交易所 Tick 接口定义的一定周期内的最高价、最低价。

```
def main():
    exchange.SetContractType("MA888")
    Log(exchange.GetTicker())
```

回测输出结果为：

```
{
    'Time': 1594170000000,
    'High': 1795.0,
    'Low': 1793.0,
    'Sell': 1795.0,
    'Buy': 1793.0,
    'Last': 1794.0,
    'Volume': 0.0,
    'OpenInterest': 1280130000.0
}
```

注意：在调用任何访问交易所接口的 API 函数时（如 exchange.GetTicker()、exchange.Buy(Price, Amount)、exchange.CancelOrder(Id)等）都有可能由于各种原因导致访问失败。所以要对这些函数的调用做容错处理。

例如：exchange.GetTicker()获取行情数据函数可能由于，交易所服务器问题，网络传输问题等，导致该函数返回值为 null，这时就需要做容错处理：

```
def main():
    exchange.SetContractType("MA888")
    ticker = exchange.GetTicker()
    if not ticker:
        ticker = exchange.GetTicker()
```

3.6.2 exchange.GetDepth()

exchange.GetDepth()函数获取交易所订单薄（深度数据），返回值:Depth 结构体。Depth 结构体包含两个结构体数组，分别是 Asks[]和 Bids[]，Asks 和 Bids 包含以下结构体变量:

数据类型	变量名	说明
number	Price	价格
number	Amount	数量

例如我想获取当前卖二价，可以这么写代码:

```
def main():
    exchange.SetContractType("MA888")
    depth = exchange.GetDepth()
    price = depth["Asks"][1]["Price"]
    Log("卖二价为:", price)
```

注意：商品期货涨停时，卖单卖一的价格是涨停价格，订单量是 0，跌停时，买单买一的价格是跌停价格，订单量是 0。通过判断买一、卖一的订单量数量，可以判断出是否涨跌停。

3.6.3 exchange.GetRecords()

exchange.GetRecords()函数可以获取历史 K 线数据，可以指定参数，表示要获取的历史 K 线数据的 K 线周期。如果不指定参数，获取以机器人、回测页面上设置的 K 线周期的数据。

```
def main():
    exchange.SetContractType("MA888")
    Log(exchange.GetRecords(60 * 2))        # 获取 2 分钟数据
    Log(exchange.GetRecords(PERIOD_M5))     # 获取 5 分钟数据
```

可以传入秒数作为要获取的 K 线数据的周期，也可以传入系统定义的值：PERIOD_M5。exchange.GetRecords()函数返回的数据为数组结构，数组中的元素为 Record 结构。

3.6.4 商品期货策略框架

商品期货策略需要检测与期货公司前置机连接状态，在获取行情之前需要订阅合约，这样才能获取订阅的行情。在发明者量化交易平台上，回测时模拟如同实盘一样的连接机制。

以上我们了解、学习的行情接口因为篇幅有限，为了容易理解，并没有写设置合约、检测与前置机连接状态等代码，那么一个完整的商品期货策略框架是什么样的呢？完整的框架：

```
def main():
    while True:
        if exchange.IO("status"):
            exchange.SetContractType("rb888")
```

```
        ticker = exchange.GetTicker()
        depth = exchange.GetDepth()
        trades = exchange.GetTrades()
        records = exchange.GetRecords()
        Log("rb888 ticker Last:", ticker["Last"])
        Log("rb888 depth:", depth)
        Log("rb888 trades:", trades)
        Log("rb888 records:", records)
        LogStatus(_D(), "已经连接 CTP ！")
    else:
        LogStatus(_D(), "未连接 CTP ！")
```

其中我们陌生的代码也只有：

- exchange.IO("status")
- exchange.SetContractType("rb888")
- LogStatus(_D(), "已经连接 CTP ！")

exchange.IO("status")函数可以判断当前是否和期货公司前置机连接，如果连接返回 1，如果非连接状态返回 0。

exchange.SetContractType("rb888")函数调用是把当前合约设置为 rb888 并订阅该合约，rb888 是螺纹钢主力合约。如果想使用指数合约，可以用 rb000。

LogStatus(_D(), "已经连接 CTP ！")函数的作用是在机器人状态栏上显示时间信息和文字，_D（）函数返回当前时间的字符串。该策略代码会不停循环执行打印行情数据，如下图所示：

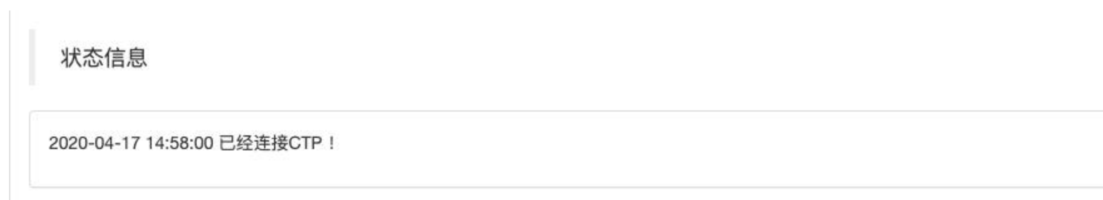

图 3.49　状态信息（1）

日志信息　　显示毫秒　下载日志

时间	平台	类型	价格	数量	信息
2020-04-17 14:58:00		信息			rb888 records: [{'Time': 1570982400000, 'Open': 3432.0, 'High': 3438.0, 'Low': 3351.0, 'Close': 3352.0, 'Volume': 2669290.0, 'OpenInterest': 3157694000.0}, {'Time': 1571068800000, 'Open': 3352.0, 'High': 3365.0, 'Low': 33
2020-04-17 14:58:00		信息			rb888 trades: []
2020-04-17 14:58:00		信息			rb888 depth: {'Asks': [{'Price': 3382.0, 'Amount': 1000.0}], 'Bids': [{'Price': 3380.0, 'Amount': 1000.0}]}
2020-04-17 14:58:00		信息			rb888 ticker Last: 3381.0
2020-04-17 14:54:00		信息			rb888 records: [{'Time': 1570982400000, 'Open': 3432.0, 'High': 3438.0, 'Low': 3351.0, 'Close': 3352.0, 'Volume': 2669290.0, 'OpenInterest': 3157694000.0}, {'Time': 1571068800000, 'Open': 3352.0, 'High': 3365.0, 'Low': 33
2020-04-17 14:54:00		信息			rb888 trades: []
2020-04-17 14:54:00		信息			rb888 depth: {'Asks': [{'Price': 3384.0, 'Amount': 1000.0}], 'Bids': [{'Price': 3382.0, 'Amount': 1000.0}]}
2020-04-17 14:54:00		信息			rb888 ticker Last: 3383.0

图 3.50　日志信息（1）

3.7 获取和取消订单、获取当前挂单

有时下单之后，可能会因为行情或者价格的原因，导致订单不能完全成交或者订单只成交了一部分。所以就需要下单之后了解订单状态，以及对未成交的订单撤单处理。

3.7.1 exchange.SetContractType(ContractType)

在商品期货中要想获取行情和下单交易，首先要先订阅合约代码，才能进行下一步操作。exchange.SetContractType(ContractType)函数用于设置合约类型，参数值：string 类型。如同我们上节课学习的，商品期货策略框架中设置合约，并且订阅该合约。我们一起来看一个新例子：

```
def main():
    while True:
        if exchange.IO("status"):
            ret = exchange.SetContractType("MA888")
            Log("订阅的合约的详细信息：", ret)
            break
        else:
            LogStatus(_D(), "未连接")
```

exchange.SetContractType("MA888")函数返回合约的详细信息，赋值给 ret 变量。输出 ret 变量结果：

```
{
    'CombinationType': 48,
    'CreateDate': 0,
    'DeliveryMonth': 4,
    'DeliveryYear': 0,
    'EndDelivDate': 0,
    'ExchangeID': 'CZCE',
    'ExchangeInstID': 'MA005',
    'ExpireDate': 0,
    'InstLifePhase': 49,
    'InstrumentID': 'MA005',
    'InstrumentName': 'MA连续',
    'IsTrading': 1,
    'LongMarginRatio': 0.07,
    'MaxLimitOrderVolume': 1000,
    'MaxMarginSideAlgorithm': 48,
    'MaxMarketOrderVolume': 1000,
    'MinLimitOrderVolume': 1,
    'MinMarketOrderVolume': 1,
    'OpenDate': 0,
    'OptionsType': 0,
    'PositionDateType': 50,
```

```
    'PositionType': 50,
    'PriceTick': 1,
    'ProductClass': 49,
    'ProductID': 'MA',
    'ShortMarginRatio': 0.07,
    'StartDelivDate': 0,
    'StrikePrice': 0,
    'UnderlyingInstrID': '',
    'UnderlyingMultiple': 1,
    'VolumeMultiple': 10
}
```

可以看到有不少信息在我们写策略时是可以用的，比如合约乘数即一手合约是多少商品，例如 MA 甲醇，一手是 10 吨。

注意：当前合约设置为 MA888 之后，就可以获取当前 MA 主力合约的行情，对当前主力合约下单等操作。我们在所有操作前首先要确保和期货公司前置机（服务器）连接，再者要明确当前操作哪个合约。

3.7.2 exchange.SetDirection(Direction)

SetDirection 函数可以设置期货下单方向，参数常用的有四种：buy，closebuy，sell，closesell。商品期货多出 closebuy_today，与 closesell_today，指平今仓，默认为 closebuy/closesell 为平昨仓。

exchange.SetDirection("buy")	买入开多仓
exchange.SetDirection("sell")	卖出开空仓
exchange.SetDirection("closebuy")	卖出平多仓
exchange.SetDirection("closesell")	买入平空仓
exchange.SetDirection("closebuy_today")	卖出平今日多仓
exchange.SetDirection("closesell_today")	买入平今日空仓

3.7.3 exchange.Buy(Price, Amount)

下买单函数，第一个参数为下单价格，第二个参数为下单量。下单成功后返回一个订单 ID。测试例子：

```
def main():
    while True:
        if exchange.IO("status"):
            ret = exchange.SetContractType("MA888")
            ticker = exchange.GetTicker()
            exchange.SetDirection("buy")
            id = exchange.Buy(ticker.Buy, 1)
            Log(id)
            break
```

```
        else:
            LogStatus(_D(), "未连接")
```

输出结果为：

日志信息

时间	平台	类型	价格	数量	信息
2020-01-22 10:00:00		信息	1		
2020-01-22 10:00:00	Futures_CTP	买入 开多	2241	1	

图 3.51　日志信息（2）

以上例子用当时的行情中的买一价作为下单价格，下单量 1 手，下了一个开多仓订单。我们可以看到如果开多仓，exchange.Buy(ticker.Buy, 1)是和 exchange.SetDirection("buy")配合使用的，那么下单函数和 exchange.SetDirection()函数都有那些组合呢？

下单函数	SetDirection 参数设置方向	备注
exchange.Buy	"buy"	买入开多仓
exchange.Buy	"closesell"	买入平空仓
exchange.Sell	"sell"	卖出开空仓
exchange.Sell	"closebuy"	卖出平多仓
exchange.Buy	"closesell_today"	买入平今日空仓
exchange.Sell	"closebuy_today"	卖出平今日多仓

3.7.4　exchange.Sell(Price, Amount)

下卖单，返回订单编号，可用于查询订单信息和取消订单。期货下单时必须注意交易方向是否设置正确。

```
def main():
    while True:
        if exchange.IO("status"):
            ret = exchange.SetContractType("MA888")
            ticker = exchange.GetTicker()
            exchange.SetDirection("sell")
            id = exchange.Sell(ticker.Sell, 1)
            Log("开空仓订单 ID: ", id)
            break
        else:
            LogStatus(_D(), "未连接")
```

下空头仓位的平仓单就是用 exchange.Buy 和 exchange.SetDirection("closesell")组合使用。下多头仓位的平仓单就是用 exchange.Sell 和 exchange.SetDirection("closebuy")组合使用。

注意：商品期货下单前，必须使用 SetDirection 函数设置下单方向和类型。

3.7.5 exchange.CancelOrder(orderId)

exchange.CancelOrder(orderId)函数可以根据订单 ID 取消订单。如下面的代码：首先订阅了 MA888 合约，然后获取 Tick 行情，接着设置下单方向，并使用 Sell 函数下单。变量 id 接收了下单后返回的订单编号，最后使用 CancelOrder 函数传入 id 参数来取消这个订单。在撤单后加 break 是为了让撤单后就跳出循环，否则会不停下单撤单。

```
def main():
    while True:
        if exchange.IO("status"):
            ret = exchange.SetContractType("MA888")
            ticker = exchange.GetTicker()
            exchange.SetDirection("sell")
            id = exchange.Sell(ticker.Sell, 1)
            Log("开空仓订单 ID: ", id)
            Sleep(5000)
            exchange.CancelOrder(id)
            break
        else:
            LogStatus(_D(), "未连接")
```

输出结果为：

日志信息

时间	平台	类型	价格	数量	信息
2020-01-22 10:00:05	Futures_CTP	撤销			
2020-01-22 10:00:00		信息	开空仓订单ID: 1		
2020-01-22 10:00:00	Futures_CTP	卖出 开空	2243	1	

图 3.52　日志信息（3）

3.7.6 exchange.GetOrders()

获取所有未完成的订单。返回值:Order 结构体数组。当交易所对象 exchange 代表的账户当前没有挂单时，调用 exchange.GetOrders()返回空数组，即：[]。

```
def main():
    contractTypeList = ["MA888", "rb888", "i888"]
    while True:
        if exchange.IO("status"):
            for i in range(len(contractTypeList)):
                ret = exchange.SetContractType(contractTypeList[i])
                ticker = exchange.GetTicker()
                exchange.SetDirection("sell")
                id = exchange.Sell(ticker.Sell, 1)
```

```
            Log(contractTypeList[i], "开空仓订单 ID: ", id)
        orders = exchange.GetOrders()
        for i in range(len(orders)):
            Log(orders[i])
        break
    else:
        LogStatus(_D(), "未连接")
```

三种合约逐个切换，下单。然后调用 exchange.GetOrders()函数获取当前挂单。然后逐个打印，回测运行，输出结果为：

日志信息

时间	平台	类型	价格	数量	信息
2020-01-22 10:00:01		信息			{'Id': 3, 'Price': 657.5, 'Amount': 1.0, 'DealAmount': 0.0, 'AvgPrice': 0.0, 'Type': 1, 'Offset': 0, 'Status': 0, 'ContractType': 'i888'}
2020-01-22 10:00:01		信息			{'Id': 2, 'Price': 3511.0, 'Amount': 1.0, 'DealAmount': 0.0, 'AvgPrice': 0.0, 'Type': 1, 'Offset': 0, 'Status': 0, 'ContractType': 'rb888'}
2020-01-22 10:00:01		信息			{'Id': 1, 'Price': 2243.0, 'Amount': 1.0, 'DealAmount': 0.0, 'AvgPrice': 0.0, 'Type': 1, 'Offset': 0, 'Status': 0, 'ContractType': 'MA888'}
2020-01-22 10:00:01		信息			i888 开空仓订单ID： 3
2020-01-22 10:00:01	Futures_CTP	卖出 开空	657.5	1	
2020-01-22 10:00:00		信息			rb888 开空仓订单ID： 2
2020-01-22 10:00:00	Futures_CTP	卖出 开空	3511	1	
2020-01-22 10:00:00		信息			MA888 开空仓订单ID： 1
2020-01-22 10:00:00	Futures_CTP	卖出 开空	2243	1	

图 3.53　日志信息（4）

从上面的例子中我们可以看出，GetOrders 函数的返回值是不区分当前设置的合约的。它返回的是所有未完成的订单。

3.7.7　exchange.GetOrder(orderId)

根据订单号获取订单详情，参数值:orderid 为要获取的订单号，string 类型或数值类型。返回值：Order 结构体。

```
def main():
    while True:
        if exchange.IO("status"):
            ret = exchange.SetContractType("MA888")
            ticker = exchange.GetTicker()
            exchange.SetDirection("sell")
            id = exchange.Sell(ticker.Sell, 1)
            Log("开空仓订单 ID: ", id)
            Sleep(5000)
            exchange.CancelOrder(id)
            Sleep(5000)
            Log(exchange.GetOrder(id))
            break
        else:
            LogStatus(_D(), "未连接")
```

上面的例子是下单之后获取订单的 id，然后使用 CancelOrder 取消这个 id 的订单，最后使用 GetOrder 函数获取这个 id 的当前订单状态，可以看到打印出的订单信息，其中 Status 属性为 2。输出结果为：

日志信息

时间	平台	类型	价格	数量	信息
2020-01-22 10:02:00		信息	{'Id': 1, 'Price': 2243.0, 'Amount': 1.0, 'DealAmount': 0.0, 'AvgPrice': 0.0, 'Type': 1, 'Offset': 0, 'Status': 2, 'ContractType': 'MA888'}		
2020-01-22 10:00:05	Futures_CTP	撤销			
2020-01-22 10:00:00		信息	开空仓订单ID：1		
2020-01-22 10:00:00	Futures_CTP	卖出 开空	2243	1	

图 3.54　日志信息（5）

3.8　IO 扩展函数

发明者量化 API 定义了常用的功能函数，但是对于一些交易之外的功能，需要请求交易所原始数据，因此发明者量化定义了 IO 扩展函数。接下来就看看 IO 函数可以做哪些事情。

3.8.1　IO 函数切换行情模式

商品期货行情是推送机制，在发明者量化交易平台，我们可以使用 exchange.IO()函数切换行情模式。一共有三种模式可以切换。

```
exchange.IO("mode", 0)
```

立即返回模式，如果当前还没有接收到交易所最新的行情数据推送，就立即返回旧的行情数据，如果有新的数据就返回新的数据，设置为该模式后，行情接口调用时会立即返回，用于非阻塞的策略架构设计，例如多品种策略。

```
exchange.IO("mode", 1)
```

缓存模式(默认模式)，如果当前还没有收到交易所最新的行情数据(同上一次接口获取的数据比较)，就等待接收然后再返回，如果调用该函数之前收到了最新的行情数据，就立即返回最新的数据。设置该模式后，在没有收到最新行情时，会阻塞在该函数。通常用于单品种的交易策略，因为只用处理一个合约的行情，处理最新行情即可，其他时间可以阻塞等待。

```
exchange.IO("mode", 2)
```

强制更新模式，进入等待一直到接收到交易所下一次的最新推送数据后返回。这种模式，只使用最新获取的行情数据，即强制等待下一次数据推送过来。

注意：一般情况下使用默认的缓存模式。

3.8.2　IO 函数判断与期货公司前置机连接状态

```
exchange.IO("status")
```

这个函数调用我们应该并不陌生了，在之前的课程中，我们已经使用过了，该函数非常简单，使用时判断其返回值即可，返回值为真，代表和期货公司前置机服务器连接成功，返回值为假，代表和期货公司前置机服务器未连接。例如：

```
def main():
    while not exchange.IO("status"):
        LogStatus("正在等待与交易服务器连接, " + _D())
```

状态栏输出结果：

状态信息

正在等待与交易服务器连接, 2020-01-23 05:00:00

图 3.55　状态信息（2）

3.8.3　IO 函数获取交易所所有合约

IO 函数的 instruments 参数可以获取交易所所有合约，返回交易所所有合约的列表，只支持实盘，完整的查询范例：

```
def main():
    while not exchange.IO("status"):
        LogStatus("正在等待与交易服务器连接, " + _D())
    Log("开始获取所有合约")
    instruments = _C(exchange.IO, "instruments")
    Log("合约列表获取成功")
    length = 0
    for i in range(len(instruments)):
        length += 1
    Log("合约列表长度为:", length)
```

类似的调用还有：

- exchange.IO("products")返回交易所所有产品的列表，只支持实盘。
- exchange.IO("subscribed")返回已订阅行情的合约，格式同上，只支持实盘。
- exchange.IO("settlement")结算单查询，只支持实盘。

3.8.4　exchange.IO("api", ...)

发明者量化的 CTP(商品期货)终端提供了完整的全 API 实现，当发明者平台的 API 满足不了你需要的功能时。可以用 exchange.IO 函数进行更深层的系统调用，完全兼容官方的

Api 名称。CTP 的 IO 直接扩展函数调用请求，将会在收到第一个 isLast 标记为 true 的响应包后返回。

注意：该方法不支持回测和模拟交易，只支持实盘交易。

- 查询投资者信息：

```
def main():
    while not exchange.IO("status"):
        LogStatus("正在等待与交易服务器连接，" + _D())
    Log(exchange.IO("api", "ReqQryInvestor"))
```

- 修改密码：

```
def main():
    Sleep(6000)
    exchange.IO("api",  "ReqUserPasswordUpdate",  {"BrokerID":  "9999",
                "UserID": "11111", "OldPassword": "oldpass", "NewPassword":
                "newpass"})
```

- 查询结算单：

```
def main():
    while not exchange.IO("status"):
        LogStatus("正在等待与交易服务器连接，" + _D())
    r  =  exchange.IO("api",  "ReqQrySettlementInfo",  {"TradingDay":
                        "20190506"})
    s = ''
    for i in range(len(r)):
        for ii in range(len(r[i])):
            if r[i][ii]["Name"] == "CThostFtdcSettlementInfoField":
                s += r[i][ii]["Value"]["Content"]
    Log(s)
```

3.8.5 exchange.IO("wait")

exchange.IO("wait")函数可以使程序在有最新事件时进行响应，执行程序逻辑，在没有新事件触发时，该函数会阻塞，可以实现回调机制的策略设计。当前交易所有任何品种更新行情信息或订单成交时才返回，返回 EventTick/OrderEvent 结构。

只支持商品期货实盘。在使用 exchange.IO("wait")时，必须至少已经订阅了一个当前处于交易状态的合约（已经交割的过期合约，不会再有行情数据），否则会阻塞在该函数（由于没有任何行情、订单更新）。一个简单实现回调机制的例子：

```
def on_tick(symbol, ticker):
    Log("symbol:", symbol, "update")
    # 数据结构：https://www.fmz.com/api#ticker
    Log("ticker:", ticker)

def on_order(order):
    Log("order update", order)
```

```
def main():
    # wait connect trade server
    while not exchange.IO("status"):
        Sleep(10)
    # switch push mode
    exchange.IO("mode", 0)
    # subscribe instrument
    _C(exchange.SetContractType, "MA001")
    while True:
        e = exchange.IO("wait")
        if e:
            if e.Event == "tick":
                on_tick(e['Symbol'], e['Ticker'])
            elif e.Event == "order":
                on_order(e['Order'])
```

以上我们通过一个例子来说明 exchange.IO("wait")的使用方法，该方法的特点是当前交易所有任何品种更新行情信息或订单成交时才返回，所以非常适合用于单品种、多品种的回调机制策略设计。

如果要设计多账户的回调机制策略，还可以使用 exchange.IO("wait_any")函数。有任何一个交易所对象收到最新事件时就返回。EventTick 里的 Index 指交易所对象索引。

注意：该方法不支持回测和模拟交易，只支持实盘交易。

3.9 账户 API 获取账户和持仓信息

账户信息和持仓信息关乎着策略逻辑，是策略逻辑的必须条件，在发明者量化平台中，可以使用 GetAccount 函数获取账户信息，用 GetPosition 函数获取持仓信息。

3.9.1 exchange.GetAccount()

返回交易所账户信息。返回值:Account 结构结构体。通常使用返回的数据中的 Balance 属性，即账户可用资金，数据中的 FrozenBalance 属性，即挂单冻结的资金。

如果需要使用其它数据计算，例如当前总权益、保证金等，这些数据保存在 Info 属性中，Info 属性内储存的数据为 CTP 接口返回的原始数据。Info 属性仅实盘有效，回测时无此属性。下面我们一起来看一个简单的例子：

```
def main():
    while True:
        if exchange.IO("status"):
            exchange.SetContractType("rb888")
            account = exchange.GetAccount()
            Log("挂单前↑")
            Log("账户可用资金，Balance", account["Balance"])
```

```
        Log("账户挂单冻结资金，FrozenBalance:", account["FrozenBalance"])
        ticker = exchange.GetTicker()
        exchange.SetDirection("buy")
        exchange.Buy(ticker.Buy - 10, 1)
        account = exchange.GetAccount()
        Log("挂单后↑")
        Log("账户可用资金，Balance", account["Balance"])
```

Log("账户挂单冻结资金，FrozenBalance:", account["FrozenBalance"])

```
        LogStatus(_D(), "已经连接 CTP ！")
        break
    else:
        LogStatus(_D(), "未连接 CTP ！")
```

例子中，设置当前操作的合约为 rb888 即螺纹钢主力合约，在下单前，获取一次账户资产信息，打印可用资金，打印挂单冻结资金。

然后获取行情，设置交易方向为开多仓，根据行情当前的买一价格，下单一手多单螺纹钢合约。然后再次获取当前账户资产信息并打印。然后为了方便观察，使用 break 语句跳出循环，策略程序执行完毕。

日志信息

时间	平台	类型	价格	数量	信息
2020-01-23 10:00:00		信息	账户挂单冻结资金，FrozenBalance: 2114.172		
2020-01-23 10:00:00		信息	账户可用资金，Balance 997885.827		
2020-01-23 10:00:00		信息	挂单后↑		
2020-01-23 10:00:00	Futures_CTP	买入 开多	3509	1	
2020-01-23 10:00:00		信息	账户挂单冻结资金，FrozenBalance: 0.0		
2020-01-23 10:00:00		信息	账户可用资金，Balance 1000000.0		
2020-01-23 10:00:00		信息	挂单前↑		

图 3.56　日志信息（6）

3.9.2　exchange.GetPosition()

获取当前持仓信息，返回值：position 结构体数组。position 结构体数组包括：交易所接口应答的原始数据、杠杆大小、持仓量、仓位冻结、持仓均价、持仓浮动盈亏、持仓方向、合约代码、仓位占用保证金等等。

注意：返回的数组中包含当前交易所对象绑定的账户所有的持仓，并非当前设置的合约的持仓数据。

我们一起来看一个例子：

```
def main():
```

```
ctList = ["rb888", "i888", "MA888", "pp888",]
while True:
    if exchange.IO("status"):
        for i in range(len(ctList)):
            ret = exchange.SetContractType(ctList[i])
            t = exchange.GetTicker()
            exchange.SetDirection("sell")
            exchange.Sell(t.Buy - 10, 1, "合约:", ctList[i], "->",
        ret["InstrumentID"])
        orders = exchange.GetOrders()
        Log("orders length:", len(orders), "orders:", orders)
        pos = exchange.GetPosition()
        for i in range(len(pos)):
            Log(pos[i])
        break
    else:
        LogStatus(_D(), "未连接 CTP ！")
```

我们把要操作的合约代码写在一个数组（列表）中，然后通过一个 for 循环去进行每一个合约的下单操作，下单价格为当前买一价格减去 10 元，下开空仓的订单，由于价格比当前买一还低 10 元，所以马上就成交了。

然后我们使用 exchange.GetOrders()函数，获取当前所有挂单，并且打印。用于观察订单是不是还处于未成交状态，如果 exchange.GetOrders()函数返回的是一个空数组即：[]，说明订单都已经成交了。然后调用 exchange.GetPosition()函数，获取当前所有持仓，并且遍历持仓数据数组，逐个打印持仓信息。

日志信息

时间	平台	类型	价格	数量	信息
2020-01-23 10:00:02		信息	{'MarginLevel': 16.0, 'Amount': 1.0, 'FrozenAmount': 0.0, 'Price': 3519.0, 'Profit': -10.0, 'Margin': 2111.4, 'Type': 1, 'ContractType': 'rb888'}		
2020-01-23 10:00:02		信息	{'MarginLevel': 16.0, 'Amount': 1.0, 'FrozenAmount': 0.0, 'Price': 7366.0, 'Profit': -5.0, 'Margin': 2209.8, 'Type': 1, 'ContractType': 'pp888'}		
2020-01-23 10:00:02		信息	{'MarginLevel': 14.0, 'Amount': 1.0, 'FrozenAmount': 0.0, 'Price': 665.5, 'Profit': -50.0, 'Margin': 4658.5, 'Type': 1, 'ContractType': 'i888'}		
2020-01-23 10:00:02		信息	{'MarginLevel': 14.0, 'Amount': 1.0, 'FrozenAmount': 0.0, 'Price': 2257.0, 'Profit': -10.0, 'Margin': 1579.9, 'Type': 1, 'ContractType': 'MA888'}		
2020-01-23 10:00:01		信息	orders length: 0 orders: []		
2020-01-23 10:00:01	Futures_CTP	卖出 开空	7356	1	合约： pp888 -> pp2005
2020-01-23 10:00:01	Futures_CTP	卖出 开空	2247	1	合约： MA888 -> MA005
2020-01-23 10:00:00	Futures_CTP	卖出 开空	655.5	1	合约： i888 -> i2005
2020-01-23 10:00:00	Futures_CTP	卖出 开空	3509	1	合约： rb888 -> rb2005

图 3.57　日志信息（7）

细心的同学可能发现，为何此处 exchange.Sell 函数传入了 6 个参数。这里我们讲解一下发明者量化交易平台可以输出日志的函数的特性，下单函数就是一个可以输出日志的函数，可以看到当前例子的运行截图中，有下单日志打印，所有可以产生日志的函数，都是可以在必要参数后增加一些附带参数，用于打印一些附带说明信息。

例如本例中，下单日志中附带了「合约： pp888 -> pp2005」这样的信息。就是要说明当前合约代码设置的为 pp888，实际映射到具体交易的合约为 pp2005，在回测时间

2020-01-23 时，pp2005 为主力合约。从例子代码中可以看到这个映射合约可以从 exchange.SetContractType("pp888")函数返回时，返回的数据中获取到。

注意：映射合约支持商品指数（pp000）映射到 pp2005，也支持商品主力连续（pp888）映射到 pp2005。

3.10 常用日志信息函数

日志可以记录量化交易中策略运行状态信息，同时还可以监控策略中指定的事件，还可以通过日志检查错误发生的原因。

3.10.1 Log(...)

Log()函数用于打印日志信息，参数可以传入多个，参数可以传入任意类型。支持使用十六进制颜色代码着色。支持消息推送。

```
def main():
    Log("发明者量化你好 !@")
    Sleep(1000 * 5)
    Log("微信你好, #ff0000@")
```

在 Log()函数最后一个参数写"@"即可实现该条日志信息推送，可以推送到微信、邮箱、Telegram、监听 WebHook 的服务程序。

注意：微信推送有频率限制，并且不能推送重复信息，重复信息会被过滤掉。

日志信息

时间	平台	类型	价格	数量	信息
2020-01-23 10:00:05		信息	微信你好,		
2020-01-23 10:00:00		信息	发明者量化你好！		

图 3.58　日志信息（8）

在编写策略时，Log()函数通常用于打印一些提示信息、输出数据。也可以用于策略程序调试，逻辑流程分析。Log 支持打印 base64 编码后的图片，以“`”开头，以“`”结尾。

```
def main():
    Log("`data:image/png;base64,AAAA`")
```

Log 支持直接打印 Python 的 matplotlib.pyplot 对象，只要对象包含 savefig 方法就可以打印。

```
import matplotlib.pyplot as plt
def main():
    plt.plot([3,6,2,4,7,1])
    Log(plt)
```

Log 函数支持语言切换，Log 函数输出文本，会根据平台页面上语言设置自动切换为对

应的语言，例如：

```
def main():
    Log("[trans]中文|abc[/trans]")
```

Log 函数相当于 Python 语言中的 print 函数，只不过发明者量化支持多种语言，所以对每种语言的打印函数用 Log 统一封装。在完整的商品期货框架中使用 Log()函数：

```
def main():
    ctList = ["rb888", "i888", "MA888", "pp888",]
    while True:
        if exchange.IO("status"):
            for i in range(len(ctList)):
                Log("遍历 ctList,当前合约为：", ctList[i])
            break
        else:
            LogStatus(_D(), "未连接 CTP ！")
```

3.10.2 LogProfit(Profit)

LogProfit 函数用来在系统日志打印一条收益信息，并且会自动在收益曲线图表上打印一个收益点。

日期	平台	类型	价格	数量	信息
2020-04-14 08:59:01	Futures_CTP	卖出 开空 - i2009	595	1	i2009 Bid {"Price":597.5,"Amount":50}
2020-04-14 08:59:00		信息	SPK 信号所在行数: 20 信号次数: 1		
2020-04-13 10:50:02		收益	1364.65		{"Info":{"Interest":0,"FrozenMargin":0,"CashIn":0,"Commission":8.9265,"Positic
2020-04-13 10:50:01	Futures_CTP	买入 平空 - i2009	604	1	i2009 平今 Ask {"Price":601.5,"Amount":2065}
2020-04-13 10:50:01		信息	BP 信号所在行数: 25 信号次数: 1		

图 3.59　日志信息（9）

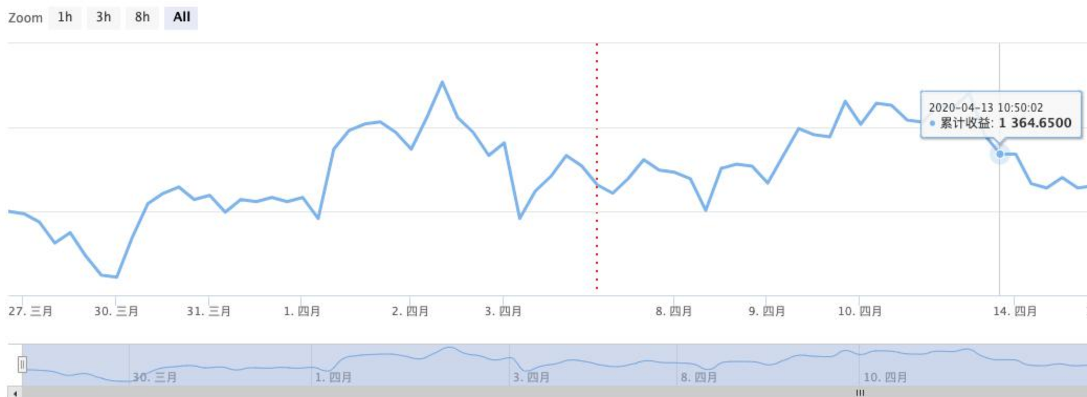

图 3.60　收益曲线（1）

该函数同样为可以打印日志的函数，可以在必要参数后传附带参数，用于显示一些打印收益时需要同时记录的信息，例如在打印收益时，同时输出当前账户资产信息，可以用于核对记录。因为打印的收益数据是需要在策略里面主动计算的，并非系统自动计算的。

所以需要注意的是如果你写的收益算法不对，打印的收益信息也就是没有意义的错误信息，此时附带一些当时的资产数据，方便核算。

3.10.3 LogStatus(Msg)

LogStatus(Msg)函数是在设计、编写策略时很重要的一个函数，用来控制策略机器人页面状态栏的显示。参数 Msg 不保存到日志列表里，只更新当前机器人的状态信息，在日志上方显示，可多次调用，更新状态。LogStatus 函数有非常多的功能，可以在状态栏上显示各种数据，显示表格，显示图片。比较常用的是显示当前时间，显示策略的相关数据信息。例如，在状态栏显示表格，写入一些信息数据：

```
import json
def main():
    while True:
        if exchange.IO("status"):
            tab1 = {
                "type": "table",
                "title": "行情数据",
                "cols": ["项目", "数据"],
                "rows": []
            }
            tab2 = {
                "type": "table",
                "title": "账户数据",
                "cols": ["项目", "数据"],
                "rows": []
            }
            tab3 = {
                "type": "table",
                "title": "持仓数据",
                "cols": ["项目", "数据"],
                "rows": []
            }

            exchange.SetContractType("rb888")

            t = exchange.GetTicker()
            a = exchange.GetAccount()
            p = exchange.GetPosition()
            tab1["rows"].append(["tick 数据", json.dumps(t)])
            tab2["rows"].append(["账户数据", json.dumps(a)])
            tab3["rows"].append(["持仓数据", json.dumps(p)])

            LogStatus(_D(), "\n`" + json.dumps(tab1) + "`\n" +
                "`" + json.dumps(tab2) + "`\n" +
```

```
            "`" + json.dumps(tab3) + "`")
    else :
        LogStatus(_D(), "未连接")
    Sleep(1000)
```

回测运行结果为：

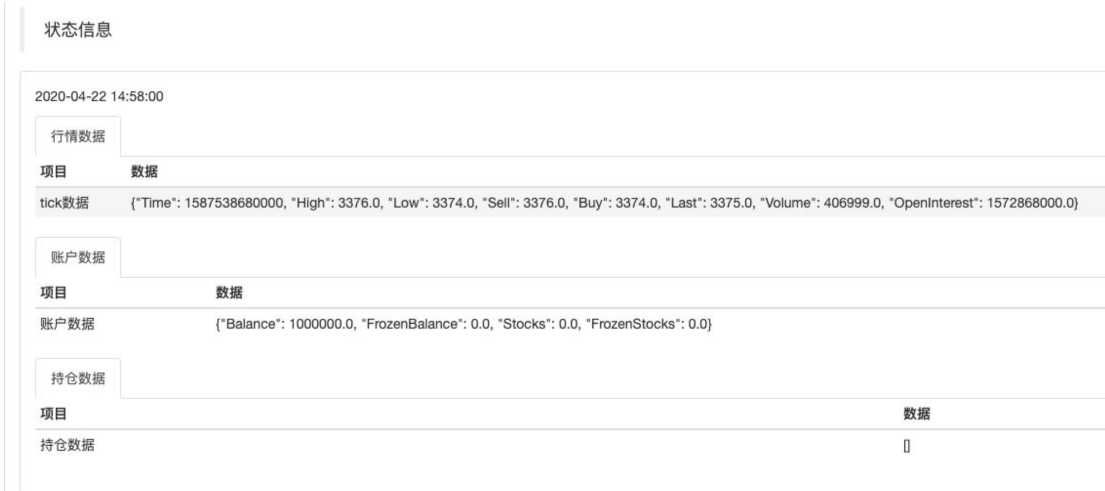

图 3.61　状态信息（3）

注意：编写设计策略时不用刻意追求 UI 显示方面要多么华丽，把需要显示的信息显示正确即可。更多的 LogStatus 函数用法、例子可以参考 API 文档。

3.10.4　Chart(…)

学习完用于状态栏显示信息的 LogStatus()函数，我们一起来学习另外一个和界面显示相关的函数，Chart()函数。Chart()函数用于策略机器人页面图表的各种操作，添加数据、更新图表配置等。可以显示各种各样的图表，使用的是 HighCharts 图表库。

发明者量化封装了原生的 HighCharts，使图表功能更好用，Chart 的参数是可以 JSON 序列化的 HighStocks 的 Highcharts.StockChart 参数，比原生的参数增加一个__isStock 属性，如果指定__isStock:false，则显示为普通图表。我们来看一个简单的例子：

```
ChartCfg = {
    '__isStock': True,
    'title': {
        'text': 'Python 画图'
    },
    'yAxis': [{
        'title': {'text': 'K 线'},
        'style': {'color': '#4572A7'},
        'opposite': False
    }, {
        'title': {'text': '指标轴'},
        'opposite': True
```

```
    }],
    'series': [{
        'type': 'candlestick',
        'name': '当前周期',
        'id': 'primary',
        'data': []
    }, {
        'type': 'line',
        'id': 'dif',
        'name': 'DIF',
        "yAxis" : 1,
        'data': []
    }, {
        'type': 'line',
        'id': 'dea',
        'name': 'DEA',
        "yAxis" : 1,
        'data': []
    }, {
        'type': 'line',
        'id': 'macd',
        'name': 'MACD',
        "yAxis" : 1,
        'data': []
    }]
}

def main():
    global ChartCfg
    preTime = 0
    chart = Chart(ChartCfg)
    chart.reset()
    while True:
        if exchange.IO("status"):
            exchange.SetContractType("rb888")
            while True:
                r = _C(exchange.GetRecords)
                if len(r) > 50:
                    break
            # 计算指标
            macd = TA.MACD(r)
            LogStatus(_D(), len(r))

            # 画图
            for i in range(len(r)):
                if r[i]["Time"] == preTime:
```

```
                chart.add(0, [r[i]["Time"], r[i]["Open"], r[i]["High"], 
        r[i]["Low"], r[i]["Close"]], -1)
                chart.add(1, [r[i]["Time"], macd[0][i]], -1)
                chart.add(2, [r[i]["Time"], macd[1][i]], -1)
                chart.add(3, [r[i]["Time"], macd[2][i]], -1)
            elif r[i]["Time"] > preTime:
                chart.add(0, [r[i]["Time"], r[i]["Open"], r[i]["High"], 
        r[i]["Low"], r[i]["Close"]])
                chart.add(1, [r[i]["Time"], macd[0][i]])
                chart.add(2, [r[i]["Time"], macd[1][i]])
                chart.add(3, [r[i]["Time"], macd[2][i]])
                preTime = r[i]["Time"]
    else :
        LogStatus(_D(), "未连接")
    Sleep(500)
```

该例子非常简单，ChartCfg 为一个图表配置字典，包含了图表配置信息，在调用发明者量化交易平台 API　Chart 函数时，传入 ChartCfg 作为参数，创建图表对象即 chart，然后就可以调用 chart 对象的 add 方法、update 方法、reset 方法等，操作图表了。

注意：当程序检测 exchange.IO("status")处于和期货公司前置机连接状态时，就订阅合约，获取 K 线数据，然后使用指标计算函数计算指标数据。将计算出的指标数据写入图表中。注意计算指标需要满足一定条件，比如 K 线数据要满足指标参数，有足够的 K 线 BAR。

图 3.62　K 线图表（1）

Chart 函数 API 文档：https://www.fmz.com/api#chart...

3.10.5　LogReset()

日志清除函数，该函数可以清除实盘或者模拟盘时机器人的日志数据。可以传入一个参数，指定保留最近多少条日志，清除其它日志。

```
def main():
    LogReset(10)
```

3.10.6 EnableLog(IsEnable)

打开或者关闭订单信息的日志记录。参数值：isEnable 为 bool 类型。IsEnable 设置为 false 则不打印订单日志，不写入机器人数据库。

```
def main():
    EnableLog(False)
```

3.11 常用内置函数

作为一个成熟的量化交易软件，发明者量化提供了很多方便易用的内置函数，节省了不少开发应用的时间，这里主要介绍常用内置函数及其用法。

3.11.1 Sleep(Millisecond)

Sleep()函数在商品期货策略中使用并不是很多，由于商品期货行情是推送机制，并不需要在策略程序中使用 Sleep()函数强制等待一定时间，只需在必要的地方使用，例如在检测和期货公司前置机连接状态时（调用 exchange.IO("status")函数）使用 Sleep()函数，避免循环中无耗时引起设备 CPU 占用过高。Sleep()函数的参数为毫秒数。例如我们之前在学习 exchange.IO 函数时，使用的例子：

```
def on_tick(symbol, ticker):
    Log("symbol:", symbol, "update")
    # 数据结构：https://www.fmz.com/api#ticker
    Log("ticker:", ticker)

def on_order(order):
    Log("order update", order)

def main():
    # wait connect trade server
    while not exchange.IO("status"):
        Sleep(10)
    # switch push mode
    exchange.IO("mode", 0)
    # subscribe instrument
    _C(exchange.SetContractType, "MA001")
    while True:
        e = exchange.IO("wait")
        if e:
            if e.Event == "tick":
                on_tick(e['Symbol'], e['Ticker'])
            elif e.Event == "order":
                on_order(e['Order'])
```

在 main()函数开始，检测和期货公司前置机连接时，就使用了 Sleep(10)。即每次执行 while 循环的循环体时，程序休眠 10 毫秒。

3.11.2　GetCommand()

获取策略交互界面发来的命令并清空，没有命令则返回 null，返回的命令格式为“按钮名称:参数”，如果没有参数，则命令就是按钮名称。

GetCommand()函数是一个非常重要的函数，策略的交互设计依赖于此函数。例如：我们给策略交互栏中添加两个交互控件（控件 a、控件 b）。

图 3.63　策略参数（3）

策略中我们按如下代码设计：

```
def main():
    while True:
        if exchange.IO("status"):
            LogStatus(_D(), "已经连接")
            cmd = GetCommand()
            if cmd :
                Log(cmd)
        else :
            LogStatus(_D(), "未连接")
        Sleep(1000)
```

运行显示结果为：

图 3.64　策略交互（1）

点击控件，即可触发打印信息。以上代码我们扩展一下就可以实现点击按钮平仓这样的半自动策略的需求。

3.11.3 IsVirtual()

IsVirtual()函数用来判断当前策略运行为机器人运行还是回测运行。回测系统运行返回true，实盘或者模拟盘机器人运行返回 false。

```
def main():
    while True:
        if exchange.IO("status"):
            exchange.SetContractType("MA009")
            exchange.GetTicker()
            LogStatus(_D(), "已经连接", "IsVirtual():", IsVirtual())
        else :
            LogStatus(_D(), "未连接", "IsVirtual():", IsVirtual())
        Sleep(1000)
```

可以看到策略回测时，状态栏上打印的 IsVirtual()函数返回值为 True。

状态信息

2020-04-23 14:58:00 已经连接 IsVirtual(): True

图 3.65　状态信息（4）

3.11.4 _G(K, V)

可保存的全局字典，回测和实盘均支持，回测结束后，保存的数据被清除。KV 表，永久保存在本地文件，每个机器人单独一个数据库，重启或者托管者退出后一直存在，K 必须为字符串，不区分大小写，V 可以为任何可以 JSON 序列化的内容。

```
def main():
    tradeCount = _G("addCount")
    if tradeCount:
        Log("恢复加仓次数数据：", tradeCount)
    else :
        tradeCount = 0
        Log("初始运行，加仓次数：", tradeCount)

    while True:
        if exchange.IO("status"):
            exchange.SetContractType("MA009")
            exchange.GetTicker()
            LogStatus(_D(), "已经连接", "IsVirtual():", IsVirtual())
```

```
            cmd = GetCommand()
            if cmd:
                tradeCount += 1
                Log("加仓次数：", tradeCount)
                Log("保存加仓次数", tradeCount)
                _G("addCount", tradeCount)
        else :
            LogStatus(_D(), "未连接", "IsVirtual():", IsVirtual())
        Sleep(1000)
```

使用交互控件模拟策略运行时触发的加仓操作，然后停止策略，保存加仓次数。当策略再次启动时，恢复加仓次数这个数据。

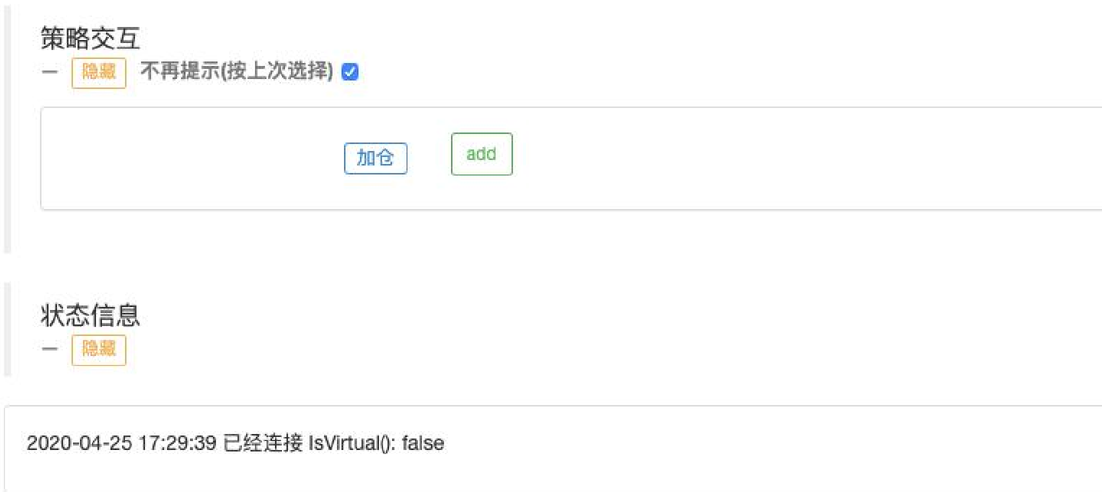

图 3.66 策略交互（2）

运行结果为：

图 3.67 策略交互（3）

可以看到，策略初始第一次运行时，加仓次数为 0，运行中我们点击了三次加仓按钮，触发加仓三次，代码 tradeCount += 1 执行了三次，然后停止策略，重新启动，可以看到 tradeCount 变量恢复为 3，再次点击加仓，继续累加 1，tradeCount 更新为 4。这样就可以实现某个数据持久化保存。

3.11.5 _D(Timestamp, Fmt)

_D(Timestamp,Fmt)函数在策略编写时也是经常用到的，例如打印当前时间字符串，就可以直接写作：Log(_D())。参数值：Timestamp 为数值类型。Fmt 为 string 类型，Fmt 默认为：yyyy-MM-dd hh:mm:ss，返回值：string 类型。

注意：不传任何参数就返回当前时间，例如：_D()，传入参数_D(1478570053)，返回指定时间戳的字符串，默认格式为 yyyy-MM-dd hh:mm:ss。

```
def main():
    strTime = _D()
    Log(strTime)
```

日志信息

时间	平台	类型	价格	数量	信息
2020-01-28 00:00:00		信息	2020-01-28 00:00:00		

图 3.68　日志信息（10）

可以看到，打印的时间和回测系统运行这行代码当时的时间记录时一致的。在使用 Python 编写策略时需要注意，Timestamp 参数为秒级别时间戳。其它语言调用该函数时，Timestamp 参数为毫秒级别时间戳。

3.11.6 _N(Num, Precision)

_N(Num, Precision)，格式化一个浮点数。参数 Num 为 number 类型，Precision 为整型 number。返回值：number 类型。例如：_N(3.1415, 2)返回 3.14。

```
def main():
    i = 3.1415
    Log(i)
    ii = _N(i, 2)
    Log(ii)
```

日志信息

时间	平台	类型	价格	数量	信息
2020-01-28 00:00:00		信息	3.14		
2020-01-28 00:00:00		信息	3.1415		

图 3.69　日志信息（11）

如果需要将小数点左边的 N 个位数都变为 0，可以这么写:

```
def main():
    i = 1300
    Log(i)
    ii = _N(i, -3)
    Log(ii)
```

日志信息

时间	平台	类型	价格	数量	信息
2020-01-28 00:00:00		信息	1000.0		
2020-01-28 00:00:00		信息	1300		

图 3.70　日志信息（12）

3.11.7　_C(function, args…)

该函数会一直调用指定函数到成功返回，比如_C(exchange.GetTicker)，默认重试间隔为 3 秒，可以调用_CDelay(...)函数来控制重试间隔，比如_CDelay(1000)，指改变_C 函数重试间隔为 1 秒。

注意：_C 函数可以对下面的函数进行容错调用。

- exchange.GetTicker()
- exchange.GetDepth()
- exchange.GetTrade()
- exchange.GetRecords()
- exchange.GetAccount()
- exchange.GetOrders()
- exchange.GetOrder()

```
def main():
    ticker = _C(exchange.GetTicker)
    _CDelay(2000)
```

```
    depth = _C(exchange.GetDepth)
    Log(ticker)
    Log(depth)
```

对于有参数的函数，使用_C(...)容错时：

```
def main():
    records = _C(exchange.GetRecords, PERIOD_D1)
    Log(records)
```

3.11.8 _Cross(Arr1, Arr2)

_Cross 函数可以判断 2 个数组的交叉状态，返回数组 arr1 与 arr2 的交叉周期数。正数为上穿周期，负数表示下穿的周期，0 指当前价格一样。它需要传入两个参数，并且这个两个参数都必须是数组。在实际应用中，这个函数可以很方便判断两条均线是否金叉死叉。我们来看一个略微复杂点的例子：

```
def main():
    while True:
        if exchange.IO("status"):
            exchange.SetContractType("MA888")
            r = _C(exchange.GetRecords)
            if len(r) < 10 :
                continue
            ma1 = TA.MA(r, 5)
            ma2 = TA.MA(r, 10)
            ext.PlotRecords(r, "MA888")
            ext.PlotLine("ma1", ma1[-2], r[-2]["Time"])
            ext.PlotLine("ma2", ma2[-2], r[-2]["Time"])
            ret = _Cross(ma1,ma2)
            LogStatus(_D(), "连接状态", "ret:", ret)
        else :
            LogStatus(_D(), "未连接状态")
        Sleep(1000)
```

回测显示结果为：

图 3.71　K 线图表（2）

状态信息

2020-04-24 14:58:00 连接状态 ret: -4

图 3. 72　状态信息（5）

该例子中我们用到了几个以前没学过的函数调用。

第一个：ma1 = TA.MA(r, 5)，MA 函数用来计算均线指标数据，第一个参数为 K 线数据，第二个参数为均线周期。我们回测时传入的 K 线数据为日 K 线，均线周期为 5。那么计算出来的均线即为 5 日均线。返回值赋值给 ma1，ma1 为一个数组，代表一条均线。同样方式计算 ma2。

第二个：ext.PlotLine("ma1", ma1[-2], r[-2]["Time"])，带有“ext.”前缀的函数为某个模板的接口函数。模板的概念就是一个封装好的可复用的代码库。

注意：需要在策略编辑页面勾选，才可以使用模板的接口函数。

图 3. 73　模板引用

ext.PlotLine 函数就是 Python 版画线类库的画线函数，用来在图表上画出一条曲线。ext.PlotRecords 函数是画出 K 线。画线类库我们不做过多介绍，知道其功能即可，详细可以在策略广场上找到这个模板代码，可以阅读源码学习。

第三个：_Cross(ma1,ma2)，每次使用_Cross 函数计算交叉周期。在调用 LogStatus 函数时写在状态栏上。可以看到状态信息显示的最后一次交叉值为-4。从画出的图表上可以观察到，快线下穿慢线已经经历了 4 个 BAR，即 4 个 K 线周期。

3.12　常用指标函数以及图表绘制

技术指标是以原始数据（开盘价、最高价、最低价、最高价、成交量等）为基础，通过一定的数学计算得出的结果。发明者量化把常用的技术指标封装成一个个函数，编写策略时无须重新计算，从而提高策略开发效率。

3.12.1 内置的 TA 指标库

发明者量化 TA 指标库，优化了常用指标算法，支持 JavaScript、Python、C++。源码地址：https://www.fmz.com/bbs-topic/409。TA 库指标函数使用非常简单，以 TA.KDJ 指标为例：

```
def main():
    r = exchange.GetRecords(PERIOD_M15)
    kdj = TA.KDJ(r, 9, 3, 3)
    Log("k:", kdj[0], "d:", kdj[1], "j:", kdj[2])
```

参数为 K 线数据，指标参数。返回一个指标数据，返回的指标数据根据不同指标有所不同。MACD，KDJ 这类多线组成的指标，指标数据是一个二维数组。RSI，ATR，MA 这类指标是一条线，返回的是一维数组。我们就以 KDJ 为例，结合之前学习的例子，画出 KDJ 指标。

```
ChartCfg = {
    '__isStock': True,
    'title': {
        'text': 'Python 画图'
    },
    'yAxis': [{
        'title': {'text': 'K 线'},
        'style': {'color': '#4572A7'},
        'opposite': False
    }, {
        'title': {'text': '指标轴'},
        'opposite': True
    }],
    'series': [{
        'type': 'candlestick',
        'name': '当前周期',
        'id': 'primary',
        'data': []
    }, {
        'type': 'line',
        'id': 'k',
        'name': 'K',
        "yAxis" : 1,
        'data': []
    }, {
        'type': 'line',
        'id': 'd',
        'name': 'D',
        "yAxis" : 1,
        'data': []
    }, {
```

```
        'type': 'line',
        'id': 'j',
        'name': 'J',
        "yAxis" : 1,
        'data': []
    }]
}

def main():
    global ChartCfg
    preTime = 0
    chart = Chart(ChartCfg)
    chart.reset()
    while True:
        if exchange.IO("status"):
            exchange.SetContractType("rb888")
            while True:
                r = _C(exchange.GetRecords)
                if len(r) > 50:
                    break
            # 计算指标
            kdj = TA.KDJ(r)
            LogStatus(_D(), len(r))

            # 画图
            for i in range(len(r)):
                if r[i]["Time"] == preTime:
                    chart.add(0, [r[i]["Time"], r[i]["Open"], r[i]["High"],
            r[i]["Low"], r[i]["Close"]], -1)
                    chart.add(1, [r[i]["Time"], kdj[0][i]], -1)
                    chart.add(2, [r[i]["Time"], kdj[1][i]], -1)
                    chart.add(3, [r[i]["Time"], kdj[2][i]], -1)
                elif r[i]["Time"] > preTime:
                    chart.add(0, [r[i]["Time"], r[i]["Open"], r[i]["High"],
            r[i]["Low"], r[i]["Close"]])
                    chart.add(1, [r[i]["Time"], kdj[0][i]])
                    chart.add(2, [r[i]["Time"], kdj[1][i]])
                    chart.add(3, [r[i]["Time"], kdj[2][i]])
                    preTime = r[i]["Time"]
        else :
            LogStatus(_D(), "未连接")
        Sleep(500)
```

回测运行结果为：

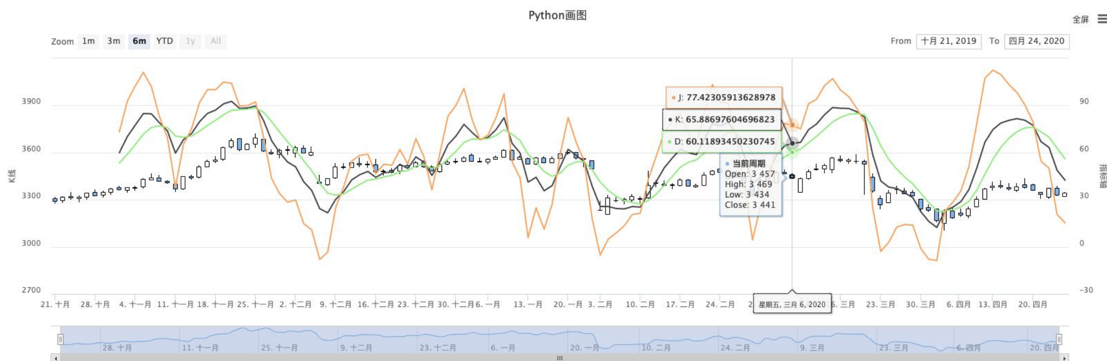

图 3.74　K 线图表（3）

注意：可以点击图片右上方“全屏”显示大图。

3.12.1　绘制图表

关于本例中图表绘制，需要注意的是指标中数据的大小范围。简单说就是当画 TA.MA 指标时，由于均线指标的数据大小和 K 线图表的价格数据基本差别不大。所以在画图时均线指标和 K 线可以共用一个 Y 轴。但是当画图 KDJ、MACD 这类指标时，由于指标值和 K 线数据差别太大，会导致图表压缩，图表展示效果就非常不好。例如图表配置中：

```
ChartCfg = {
    '__isStock': True,                    # 设置为 True 代表使用的是 Highstocks
    'title': {                            # 图表标题
        'text': 'Python 画图'
    },
    'yAxis': [{                           # Y 轴设置，这里包含 2 个 Y 轴
        'title': {'text': 'K 线'},        # 第一个 Y 轴名称
        'style': {'color': '#4572A7'},
        'opposite': False
    }, {
        'title': {'text': '指标轴'},      # 第二个 Y 轴名称，用于指标数据
        'opposite': True
    }],
    'series': [{                          # 数据系列
        'type': 'candlestick',            # K 线类型数据
        'name': '当前周期',
        'id': 'primary',
        'data': []
    }, {
        'type': 'line',                   # 曲线类型数据
        'id': 'k',
        'name': 'K',                      # 图表上显示名称 K，指标 KDJ 中的 K
        "yAxis" : 0,
        'data': []
    }, {
```

```
        'type': 'line',
        'id': 'd',
        'name': 'D',                        # 图表上显示名称 D，指标 KDJ 中的 D
        "yAxis" : 0,
        'data': []
    }, {
        'type': 'line',
        'id': 'j',
        'name': 'J',                        # 图表上显示名称 J，指标 KDJ 中的 J
        "yAxis" : 0,
        'data': []
    }]
}
```

注意：如果每个数据系列中设置："yAxis" : 0，让所有显示的数据都使用索引为 0 的指标轴，也就是都按照一个 Y 轴画图，这样显然不合理。

输出的图表为：

图 3.75　K 线图表（4）

图表配置有个大概了解后，我们继续看策略中如何把指标数据加载到图表中。首先，要计算出指标数据，需要有 K 线数据。计算指标数据使用函数：kdj = TA.KDJ(r)，传入 r 这个 K 线数据进行计算。

KDJ 指标参数不写即为默认 9，3，3。计算得出指标数据 kdj 这个二维数组。拿到指标数据，就可以开始向图表中添加数据，画指标线、K 线了。

```
56            # 画图
57            for i in range(len(r)):
58                if r[i]["Time"] == preTime:
59                    chart.add(0, [r[i]["Time"], r[i]["Open"], r[i]["High"], r[i]["Low"], r[i]["Close"]], -1)
60                    chart.add(1, [r[i]["Time"], kdj[0][i]], -1)
61                    chart.add(2, [r[i]["Time"], kdj[1][i]], -1)
62                    chart.add(3, [r[i]["Time"], kdj[2][i]], -1)
63                elif r[i]["Time"] > preTime:
64                    chart.add(0, [r[i]["Time"], r[i]["Open"], r[i]["High"], r[i]["Low"], r[i]["Close"]])
65                    chart.add(1, [r[i]["Time"], kdj[0][i]])
66                    chart.add(2, [r[i]["Time"], kdj[1][i]])
67                    chart.add(3, [r[i]["Time"], kdj[2][i]])
68                    preTime = r[i]["Time"]
```

向图表中添加数据使用图表对象 chart 的成员 add 函数，add 函数第一个参数为数据系列的索引，设置 0 即代表向 ChartCfg 配置中的 series 数据系列中第一个数据系列写入数据。

注意：ChartCfg 配置中 series 数据系列的第一个为 K 线数据，第二个为 KDJ 指标中的 K 指标线，以此类推。

add 的第二个参数为写入的数据内容。可以看到数据内容的格式和要写入的数据类型有关，例如 add 函数第一个参数，即索引为 0 时，写入 K 线数据，数据形式为[r[i]["Time"], r[i]["Open"], r[i]["High"], r[i]["Low"], r[i]["Close"]]，即一个数组。

其中元素顺序是时间戳、开盘价、最高价、最低价、收盘价。当写入 KDJ 指标线时，例如写入 D 线时，数据为[r[i]["Time"], kdj[1][i]]。即一个数组，其中元素为时间戳、KDJ 中 D 线指标值。add 的第三个参数不传时，此次 add 添加数据为添加一个新数据点，如果传-1 代表用传入的参数修改最后一个数据点。

```
for i in range(len(r)):
if r[i]["Time"] == preTime:
    # 更新数据
    elif r[i]["Time"] > preTime:
        # 写入新数据点
        preTime = r[i]["Time"]
```

用以上的遍历程序结构，就可以在首次运行写入数据后，每次检查 K 线数据最后 BAR 是否更新，更新时（r[i]["Time"] > preTime 时）写入新数据点，K 线 BAR 未出新 BAR 时（r[i]["Time"] == preTime 时）只更新数据。

可以动手试下修改程序，画出布林线指标（因为 BOLL 线指标也是三条线组成和 KDJ 形式一样）。再来看一个例子，我们这次画一个带成交量的 K 线图表。

```
cfg = {
    "rangeSelector": {
        "selected": 0
    },
    "title": {
        "text": '带量柱的 K 线图表'
    },
    "yAxis": [{
        "labels": {
            "align": 'right',
            "x": -3
        },
        "title": {
            "text": 'OHLC'
        },
        "height": '60%',
        "lineWidth": 2,
        "resize": {
            "enabled": true
        }
    }, {
        "labels": {
            "align": 'right',
```

```
            "x": -3
        },
        "title": {
            "text": 'Volume'
        },
        "top": '65%',
        "height": '35%',
        "offset": 0,
        "lineWidth": 2
    }],
    "tooltip": {
        "split": true
    },
    "series": [{
        "type": 'candlestick',
        "name": 'AAPL',
        "data": []
    }, {
        "type": 'column',
        "name": 'Volume',
        "data": [],
        "yAxis": 1
    }]
}

def main():
    chart = Chart(cfg)
    chart.reset()
    preBarTime = 0
    while True:
        if exchange.IO("status"):
            LogStatus(_D(), "已经连接")
            exchange.SetContractType("MA888")
            r = _C(exchange.GetRecords)
            for i in range(len(r) - 1):
                if r[i]["Time"] > preBarTime:
                    chart.add(0, [r[i]["Time"], r[i]["Open"], r[i]["High"],
            r[i]["Low"], r[i]["Close"]])
                    chart.add(1, [r[i]["Time"], r[i]["Volume"]])
                    preBarTime = r[i]["Time"]
        else:
            LogStatus(_D(), "未连接")
        Sleep(1000)
```

可以看到图表配置 cfg 变量，其中有两个 Y 轴配置（yAxis），并且有两个数据系列（series 中有两个元素）。在回测系统中运行：

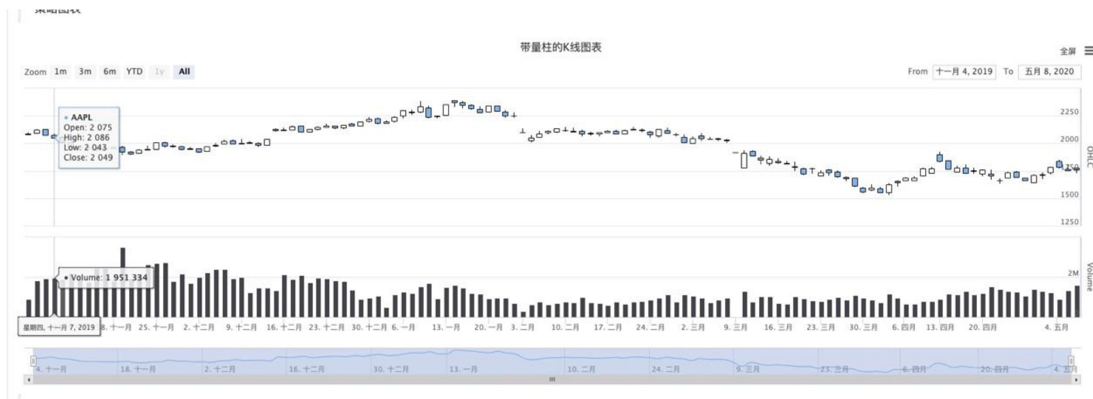

图 3.76 K 线图表（5）

上图就是我们使用 chart 绘制的一张图表并把它打印出来，这个图表包含 2 个子图，一个是 K 线图，一个是成交量。你可以根据上面的代码，改成任意你需要的数据，比如：MACD、KDJ 等等。

Python 的 highcharts 库支持更多图形，包括：

- K 线图
- 折线图和曲线图
- 面积图
- 柱形图和饼形图

3.13 策略参数及交互

在量化交易中，参数的大小，往往决定着策略最后的绩效结果。通过优化外部参数，可以使策略及时适应当前的市场行情。另外策略交互也可以通过手动的方式，给机器人发出各种指令，方便策略维护。

3.13.1 策略参数

参数实际上就是变量，如果将变量固定的写到策略代码中，这样每次调试策略的时候，就需要在代码中修改这个变量，这样看起来非常不灵活。并且没有办法对这个变量进行优化处理。

那么外部参数就很好的解决了这个问题，在发明者量化平台中，策略参数是以全局变量形式使用。Python 策略中使用策略参数需要用 global 关键字声明。策略参数有以下几种，在策略编辑页面设置的不同种类的的参数：

图 3.77　策略参数（4）

注意：界面参数，在策略编辑页面代码编辑区下方策略参数区设置。界面参数在策略代码中是以全局变量形式存在的，也就是说，可以在代码中修改界面参数。

- ❑ 描述选项：界面参数在策略界面上的名字。
- ❑ 备注选项：界面参数的详细描述。
- ❑ 类型选项：该界面参数的类型。
- ❑ 默认值选项：该界面参数的默认值。

另外，也可以设置一个参数，让另一个参数基于该参数的选择，实现显示与隐藏。比如我们设置参数 numberA，是一个数值类型。我们让 numberA 基于一个参数：isShowA(布尔类型)的真假决定 numberA 显示与隐藏。需要把 numberA 变量在界面参数上设置为：numberA@isShowA。

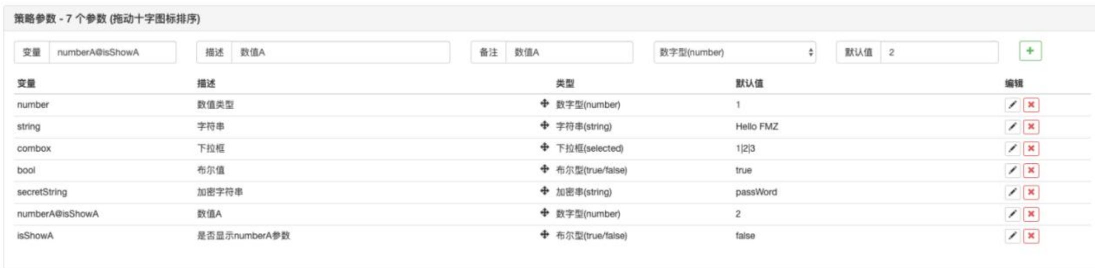

图 3.78　策略参数（5）

这样，不勾选 isShowA 参数，numberA 参数就隐藏了。

策略界面参数、交互控件、模板上的参数分组功能，只用在开始分组的参数的描述开头加上 (?第一组)即可。参数分组多品种或多周期等投资组合中，可以很方便的针对不同的品种使用不同的参数组，并且无需在原有的参数上再次修改。如下图所示：

图 3.79　策略参数（6）

图 3.80　策略参数（7）

3.13.2　策略交互

策略编辑页面可以设置策略的界面交互控件，这些控件会向运行时的机器人策略程序发送命令。这些命令由发明者量化交易平台 API 函数中的 GetCommand()函数捕获。然后策略可以根据这个命令做对应的操作（预先设计好的执行代码）。

底层系统有一个变量记录交互命令，当 GetCommand()函数被调用时，会取出交互命令，并且把底层系统的这个变量内容清空。当没有调用 GetCommand()时，底层系统中新的交互命令会覆盖旧的交互命令。例如当策略程序正在处理一个交互触发的策略代码段时，这个时候如果有多个交互命令从策略界面发送过来，此时如果策略还在执行某段代码，当策略再次 GetCommand()时，只能获取到最后一个交互命令。交互控件分为以下几种：

- 数字型
- 布尔型
- 字符串
- 下拉框
- 按钮

图 3.81　策略参数（8）

按钮类型的交互控件是不携带数据的一种控件，点击按钮即可触发。剩下的几种控件均为携带数据的控件。我们可以用一个测试策略测试交互功能：

```
def main():
    while True:
        LogStatus(_D())
        cmd = GetCommand()
        if cmd:
            Log("cmd:", cmd)
            arr = cmd.split(":")
            if arr[0] == "buy":
                Log("买入，该控件不带数量")
            elif arr[0] == "sell":
                Log("卖出，该控件带数量：", arr[1])
            else:
                Log("其它控件触发：", arr)
        Sleep(1000)
```

状态信息
— 隐藏

依次点击

2020-05-12 11:36:27

日志信息
— 共: 464 条, 24 页
— 隐藏错误日志 刷新 ▢显示毫秒 ▢声音提醒

日期	平台	类型	价格	数量	信息
2020-05-12 11:36:24		信息	买入，该控件不带数量		
2020-05-12 11:36:24		信息	cmd: buy		
2020-05-12 11:36:21		信息	卖出，该控件带数量： 1		
2020-05-12 11:36:21		信息	cmd: sell:1		
2020-05-12 11:36:14		信息	其它控件触发： [other 2]		
2020-05-12 11:36:14		信息	cmd: other:2		
2020-05-12 11:36:08		信息	其它控件触发： [isOpen true]		
2020-05-12 11:36:08		信息	cmd: isOpen:true		
2020-05-12 11:35:58		信息	其它控件触发： [ct i888]		
2020-05-12 11:35:58		信息	cmd: ct:i888		
2020-05-12 11:35:49		重启			

图 3.82　策略交互（4）

注意：这个交互例子只是一个简单的教学，实际编写策略时，要针对每个不同的控件指令做一系列的响应处理，携带的数据（GetCommand（）函数返回的为字符串）参与计算则需要转换成数值类型。

3.14　内置模板类库及经典策略架构

俗话说站在巨人肩膀上，才能看得更远。有时候你想费尽精力实现一个功能，可能早已有极好的解决方法，如果可以用现成的，那么节省下来的时间是不是可以用在策略逻辑

上呢？内置模板类库提供了大量的可复用的代码模块。

3.14.1　模板类库

发明者量化交易平台支持把一些常用的、可复用的代码模块封装成独立的库（在发明者量化交易平台上叫做模板类库）。这样可以提高策略开发速度，不用编写重复的代码，降低策略交易部分和策略逻辑部分的耦合度，便于策略维护、优化、扩展。在发明者量化交易平台上创建一个模板类库：

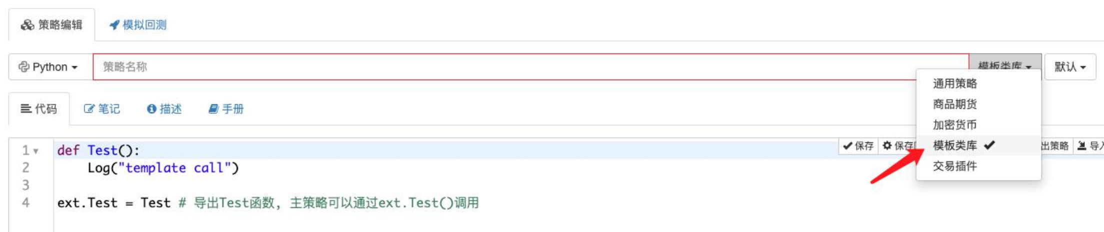

图 3.83　模板类库

创建一个模板类库和创建一个策略操作相同，在「策略库」中点击「新建策略」跳转到创建策略页面，区别是策略类型选择「模板类库」。然后给模板起个名字，保存即可。

模板类库设计主要有两个方面：

第一是需要设计导出函数，导出函数为模板类库的接口函数，即这个模板类库提供哪些功能，通过调用导出函数去使用这个模板类库提供的功能。第二是需要设计这个模板类库的参数，和普通策略一样，模板类库也可以设置参数，用于使用时动态设置一些数值等参数数据。

Python 版商品期货交易类库，这个模板类库移植自发明者量化交易平台 JavaScript 版商品期货交易类库。主要功能是对于商品期货交易开仓平仓的操作。可以学习该模板的设计思路，模板地址：https://www.fmz.com/strategy/24288

3.14.2　经典策略架构

发明者量化交易平台策略执行时是 onTick 轮询机制，整个策略逻辑可以设计为一个函数，该函数可以以一定时间间隔（防止轮询频率过快）执行。每次执行时可以进行获取当前实时行情，下单交易等操作，策略程序是每时每刻都在执行策略逻辑。

区别于 onBar 机制，onBar 机制为只有新 K 线 BAR 生成的时候，才去执行策略逻辑。如下，商品期货策略常用的 onTick 策略架构：

```
def onTick():
    exchange.SetContractType("rb888")
    ticker = exchange.GetTicker()
Log("rb888 ticker:", ticker)
# 可以写具体的策略逻辑，检测价格，触发开仓，下单等
```

```
def main():
    while True:
        if exchange.IO("status"):
            onTick()
            LogStatus(_D(), "已经连接 CTP ！")
        else:
            LogStatus(_D(), "未连接 CTP ！")
```

当然，如果需要 onBar 的策略架构，在发明者量化交易平台上同样是可以实现的，例如以下策略架构：

```
preBarTime = 0

def onTick():
    global preBarTime
    exchange.SetContractType("rb888")
    r = _C(exchange.GetRecords)
    if r[-1]["Time"] != preBarTime:
        preBarTime = r[-1]["Time"]
        Log("K 线 bar 更新了，执行交易逻辑！")
        # 可以写具体的策略逻辑，检测价格，触发开仓，下单等

def main():
    while True:
        if exchange.IO("status"):
            onTick()
            LogStatus(_D(), "已经连接 CTP ！")
        else:
            LogStatus(_D(), "未连接 CTP ！")
```

可以看到，只是加了几行代码，就把策略从 onTick 机制变为 onBar 机制了。

2020-02-17 09:00:00	信息	K线bar更新了，执行交易逻辑！
2020-02-14 09:00:00	信息	K线bar更新了，执行交易逻辑！
2020-02-13 09:00:00	信息	K线bar更新了，执行交易逻辑！
2020-02-12 09:00:00	信息	K线bar更新了，执行交易逻辑！
2020-02-12 00:00:00	信息	K线bar更新了，执行交易逻辑！

图 3.84　日志信息（13）

设置 K 线周期为日 K 线，可以看到，每次当 K 线更新时，才执行了 Log()函数打印日志信息。

3.15　温故知新

学完本章内容，读者需要回答：

1．试着配置交易所和托管者。
2．模拟级别回测和实盘级别回测有什么不同？
3．如何获取 K 线数据？
4．如何获取账户和持仓数据？
5．试着绘制图表。
在下一章中，读者会了解到：
1．什么是 CTA 策略
2．一些经典的 CTA 策略
3．策略开发中的注意事项

第 4 章　CTA 之趋势跟踪策略

CTA 是一种多样性的投资方法，一般指商品期货和金融期货策略，它并不拘谨于是主观交易还是量化交易，只要其交易方法相对规则化、系统化都可以称为 CTA 策略。本章将结合不同的策略理论来开发 CTA 策略。

本章主要涉及到的知识点有：

- 认识 CTA：对 CTA 策略有大致的了解。
- 策略逻辑：学会把交易理论和经验组合成策略逻辑。
- 策略实现：学会策略代码编写，让想法变为交易策略。

4.1　什么是 CTA 策略

CTA 的英文全称是 Commodity Trading Advisor，直译为商品交易顾问，通常指专业的资金管理人或机构。CTA 最初活跃于商品市场，随着金融市场的发展，其投资领域逐渐拓展到股票、国债、外汇等等。

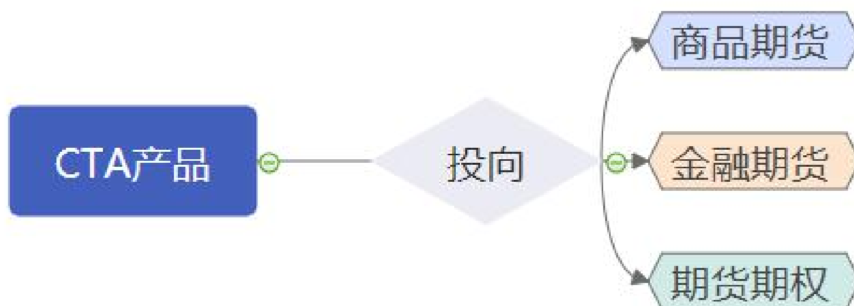

图 4.1　CTA 策略投资领域

4.1.1　CTA 策略的分类

CTA 策略的交易周期主要以分钟、小时和日线等数据为主，也有少部分使用 1 分钟以下周期的数据。CTA 策略有多种类型，以策略持仓周期可以分为中长线策略、短线策略、高频策略。以交易方法可以分为趋势策略和反转策略。

在量化 CTA 策略实际交易中，这两类策略并不是独立运行，有时候会根据市场状况和各自的优劣进行策略组合，相对于单一类型的策略而言，这种优化组合和分散部署策略的方法，经常能取到更好的效果。

注意：CTA 策略总体可以分为趋势策略和反转策略两大类。

4.1.2　趋势策略

用“守株待兔”这个词形容趋势策略再恰当不过了，这种策略的理念是顺势而为。大多时候不需要对未来价格进行预测，而是利用一些技术指标守在趋势发生的必经之路上，买入并持有直到趋势消失时退出。

趋势策略的特点是经常亏小钱，但一次就赚个大的。资金曲线呈现脉冲状态，一般回撤较大，所以该策略对资产管理、风险控制，以及交易者的心理素质要求有比较高，通常是叠加多个交易品种，或利用策略多样性来降低风险。

4.1.3　反转策略

反转策略其实就是低买高卖赚取差价，它的交易方法与趋势策略相对应，在价格低时买入，在价格高时卖出。其特点是经常赚小钱，但一旦遇到趋势行情就赔个大的，资金曲线呈现阶梯状。

由于市场大部分时间都在一个价格范围内上下波动，反转策略又非常适合这种行情，因此有很多人使用这种策略。反转策略既可以在两个相同品种的价差上套利交易，又可以利用价格网格进行低买高卖。

4.1.4　量化 CTA 策略

当然 CTA 策略绝不仅限于此，根据基本面、产业链调研、操盘经验等主观判断价格走势决定买卖也属于 CTA 策略。不过随着计算机科学的发展，涌现出很多量化 CTA 策略，主要是通过数据建模分析，发掘潜在交易机会。包括：自然语言处理、循环神经网络、随机森林模型等技术。

但是总体来看，量化 CTA 中用的比较多的策略是趋势策略，相对于其他策略而言，量化趋势策略的优点是：

1、入门简单，做过交易的人大概都知道什么是技术指标，这些技术指标不需要太多的编程技巧就可以轻易转换为量化趋势策略。

2、无惧牛熊，尤其是在期货这种双向可以交易的市场，趋势策略在价格上涨或下跌的行情中都能获利，特别是在牛市熊市快速转换或趋势明显的时候。

3、由于每个交易者对风险承受力都不一样，随着行情加剧变化，亏损会使反向交易者出现非理性的踩踏式平仓，这一点非常有利于量化趋势策略。

4.2 经典 MACD 交易策略

在第 1、2、3 章中我们学习了量化交易基础知识、Python 编程语言基础语法以及发明者量化平台使用方法。虽然内容很枯燥，但这是你实现交易策略的必备知识。接下来几章我们就用之前所学到的知识，趁热打铁从最简单的策略开始边学边用，一步一步帮助大家实现交易策略。

4.2.1 MACD 简介

相信做过交易的人对 MACD 都不陌生，这是一个非常古老的技术指标，它是由查拉尔·阿佩尔(Geral Appel)在上个世纪 70 年代发明的，全称指数平滑异同移动平均线。顾名思义这个指标是通过均线来对趋势进行判断。

如表 4.2 所示 MACD 主要由黄白线和中间的红绿柱组成。白线叫做 DIF，反映的是一段时间内价格的变化情况。黄线叫做 DEA，它是 DIF 的均线，所以相对要平缓一些。而红绿柱叫做 BAR 柱，反映的是 DIF 与 DEA 两线之间的距离。

注意：MACD 是一种中长线的趋势指标，在市场反复震荡时，可能会出现错误信号。

图 4.2　MACD 图表

4.2.2 MACD 原理

严格来讲 MACD 是均线的延伸，其意义与均线基本相同，只是它在计算时赋予了权重，时间越近赋予权重越大。同时它在图形上展示时非常直观，观察起来也一目了然。最重要的是其巧妙的利用短期指数移动平均线与长期指数移动平均线之间的聚合与分离状况，来判断市场状态。

具体来说，它是运用快速和慢速移动平均线之差，并加以双重平滑运算得来。这样不仅去除了一部分普通移动平均线频繁发出的假信号，而且还保留了移动平均线判断趋势行情的效果。因此 MACD 指标比均线更有趋势性和稳定性。具体来说就是：

第 1 步：先对杂乱的 K 线均值处理，即 EMA12 和 EMA26。EMA 其实是另一种复杂均线，与普通均线不同的是，其价格权重以指数形式逐渐缩小，随着时间的增加，价格的权重越大，更能及时反映近期价格波动情况。

第 2 步：为了解决信号滞后和频繁的无效信号问题，对两根均线差值处理，即 DIF。均线差值可以灵活反映两根均线的相互关系，DIF 上升往往意味着短期成本的涨速高于长期成本的涨速，市场短期内资金买入的意愿更强。

第 3 步：重复第 1 步，对 DIF 均值处理，即 DEA。

第 4 步：重复第 2 步，对 DIF、DEA 差值处理，即 MACD 直方图（Histogram），也就是我们常说的红绿柱子。

4.2.3 MACD 计算方法

第 1 步：计算 EMA12 和 EMA26

- EMA12 = XAverage(Close, 12)
- EMA26 = XAverage(Close, 26)

第 2 步：计算 DIF

- DIF = EMA12 - EMA26

第 3 步：计算 DEA

- DEA = XAverage(DIF, 9)

第 4 步：计算 MACD 直方图（Histogram）

- Histogram = DIF - DEA

4.2.4 MACD 使用方法

网上关于 MACD 的使用方法层出不穷，有利用 DIF 与 DEA 金叉死叉做趋势的、有结合价格看顶底背离做抄底的、也有辅助其他技术分析工具的等等，这些方法只在特定的时间有效。交易的关键不是用了哪个万能指标，而是制定一个正期望的交易策略，然后重复执行。

对于大部分交易者来说，做趋势策略要好过做震荡策略，因为趋势策略的容错率更低。

传统的使用方法是看 DIF 与零线的位置关系，或者是 DIF 与 DEA 的交叉状态，来判断价格走势。一般来说 DIF 大于 0 表示上涨，小于 0 表示下跌；或者当 DIF 向上突破 DEA 时，形成买入信号，当 DIF 向下突破 DEA 时，形成卖出信号。

但也有一部分人用红绿柱的高低，再结合价格走势判断 MACD 的顶背离和底背离，这是一种典型的反转交易方法。其理论是价格与 MACD 是趋于同向的，当价格上涨，MACD 也跟着上涨。如果价格逐步下跌，但 MACD 并没有跟着下跌，甚至逐步上升，与价格走势背道而驰，形成底背离，这将预示着价格可能即将上涨。同理，如果价格逐步上涨，但 MACD 并没有跟着上涨，甚至逐步下跌，与价格走势形成顶背离，这将预示着价格可能会下跌。

4.2.5　MACD 的有效性

值得注意的是，上面的几种方法，虽然逻辑上能站得住脚，但在实际使用时也有反复打脸的时候。为什么有时候会不灵呢？这里面有个悖论，可以试着想一下，如果 MACD 一直有效，那么大家都会来用，那么大家的买卖点位相似的。比如 DIF 向上突破 DEA 时是买入信号，大家都在买，谁会卖出呢？

所以，最终的结果将会是，MACD 越有效，用的人也就越多，当越来越多人使用它的时候，它就会慢慢失效，直到大部分人都放弃使用它，它又重新变的有效。因为人才是金融市场的最终参与者，这既不像物理定律，也不是数学公式，这里面没有整齐划一的规律，这是人与人买卖博弈的最终结果。

注意：市场是人与人之间的博弈，也是交易策略与交易策略之间的博弈，没有恒久有效的策略，也没有恒久无效的策略。

4.2.6　MACD 策略逻辑

所以，交易界有句俗语：大道至简，重剑无锋。意思就是越简单的东西，越没人信，用的人也就越少，其结果反而越有效，普适性越强。比如：少吃多动至今仍然是减肥的成功秘诀，但真正做的人很少，因为大部分人不相信或者半信半疑没有坚持下来。那么今天我们就用最简单的方法构建一个 MACD 策略。策略逻辑如下：

- 多头开仓：DIF 大于 DEA
- 空头开仓：DIF 小于 DEA
- 多头平仓：DIF 小于 DEA
- 空头平仓：DIF 大于 DEA

4.2.7　MACD 策略编写

根据上面的策略逻辑，用 Python 实现交易策略。首先注册并登录 fmz.com 网站，依次点击控制中心>策略库>新建策略>点击右上角下拉菜单选择 Python 语言，开始编写策略，注意看下面代码中的注释。

第 1 步：编写策略框架。策略开发就像盖房子一样，先把地基和框架搭建好，再往里面填充东西。策略框架包含 main 函数和 onTick 函数，main 函数是策略的入口函数，程序先从 main 函数开始逐行执行代码。在 main 函数中是 while 无限循环，重复执行 onTick 函数。一般情况下将策略核心代码写在 onTick 函数中。

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:                              # 进入无限循环模式
        onTick()                             # 执行策略主函数
        Sleep(1000)                          # 休眠 1 秒
```

第 2 步：定义虚拟持仓变量。虚拟持仓指的是理论持仓而非真实持仓，也就是在开平仓时假设订单完全成交。使用虚拟持仓的好处是编写简单，可以降低初学者编程门槛，快速迭代策略更新，一般用于回测环境中。

```
mp = 0                                       # 定义一个全局变量，用于控制虚拟持仓
```

虚拟持仓的原理很简单，策略运行之初默认是空仓 mp=0，当开多单后把虚拟持仓重置为 mp=1，当开空单后把虚拟持仓重置为，mp=-1，当平多单或空单后把虚拟持仓重置为 mp=0。这样我们在编写策略逻辑时，只需要判断 mp 的值就可以了。

第 3 步：计算 MACD。在发明者量化平台中内置了很多常用的指标函数，直接调用指标函数传入参数就可以计算结果，不需要再重新计算。MACD 计算过程是：订阅期货合约>>>获取 K 线数组>>>调用内置 MACD 指标函数即可。

```
_C(exchange.SetContractType, "rb000")        # 订阅期货品种
bar = _C(exchange.GetRecords)                # 获取 K 线数组
if len(bar) < 100:                           # 如果 K 线数组长度太小就返回
    return
macd = TA.MACD(bar, 5, 50, 15)               # 计算 MACD 值
dif = macd[0][-2]                            # 获取 DIF 的值，返回一个数组
dea = macd[1][-2]                            # 获取 DEA 的值，返回一个数组
```

注意：在调用内置指标函数之前，需要先判断 K 线数组的长度，因为指标的计算依赖足够的 K 线数据，所以使用 if 语句进行判断，如果 K 线数组长度太小，不足以计算指标，就直接返回。另外由于在计算指标时使用了收盘价数据，在 K 线还没有走完的时候，所计算的结果也会跟着来回变化，直接使用会造成信号闪烁。所以为了解决这个问题，折中的方法是在开平仓条件成立后，在下一根 K 线下单交易。

第 4 步：获取最新价。获取最新价的目的是下单交易，在下单函数 exchange.Buy()和 exchange.Sell()中，需要有 2 个参数，第 1 个是下单价格，也就是说在开平仓时必须指定固定的价格，通过获取 K 线数组最后一个元素中的'Close'就可以获取最新的价格（卖一价）。

```
last_close = bar_arr[-1]['Close']            # 获取最新价格（卖价）
```

第 5 步：下单交易。在开平仓条件中，首先判断当前的持仓状态；然后再判断 DIF 与零轴的位置或 DIF 与 DEA 的交叉状态；接着如果条件成立，就设置交易方向和类型，即：

开多、开空、平多、平空；最后使用 Buy 和 Sell 函数下单，下单之后重置虚拟持仓的状态。

```
global mp                                    # 全局变量，用于控制虚拟持仓
if mp == 1 and dif < dea:
    exchange.SetDirection("closebuy")        # 设置交易方向和类型
    exchange.Sell(last_close - 1, 1)         # 平多单
    mp = 0                                   # 设置虚拟持仓的值，即空仓
if mp == -1 and dif > dea:
    exchange.SetDirection("closesell")       # 设置交易方向和类型
    exchange.Buy(last_close, 1)              # 平空单
    mp = 0                                   # 设置虚拟持仓的值，即空仓
if mp == 0 and dif > dea:
    exchange.SetDirection("buy")             # 设置交易方向和类型
    exchange.Buy(last_close, 1)              # 开多单
    mp = 1                                   # 设置虚拟持仓的值，即有多单
if mp == 0 and dif < dea:
    exchange.SetDirection("sell")            # 设置交易方向和类型
    exchange.Sell(last_close - 1, 1)         # 开空单
    mp = -1                                  # 设置虚拟持仓的值，即有空单
```

4.2.8 完整策略代码

```
mp = 0                                       # 定义一个全局变量，用于控制虚拟持仓

# 程序主函数
def onTick():
    _C(exchange.SetContractType, "rb000")    # 订阅期货品种
    bar = _C(exchange.GetRecords)            # 获取 K 线数组
    if len(bar) < 100:                       # 如果 K 线数组长度太小就返回
        return
    macd = TA.MACD(bar, 5, 50, 15)           # 计算 MACD 值
    dif = macd[0][-2]                        # 获取 DIF 的值，返回一个数组
    dea = macd[1][-2]                        # 获取 DEA 的值，返回一个数组
    last_close = bar[-1]['Close']            # 获取最新价格（卖价）
    global mp                                # 全局变量，用于控制虚拟持仓
    if mp == 1 and dif < dea:
        exchange.SetDirection("closebuy")    # 设置交易方向和类型
        exchange.Sell(last_close - 1, 1)     # 平多单
        mp = 0                               # 设置虚拟持仓的值，即空仓
    if mp == -1 and dif > dea:
        exchange.SetDirection("closesell")   # 设置交易方向和类型
        exchange.Buy(last_close, 1)          # 平空单
        mp = 0                               # 设置虚拟持仓的值，即空仓
    if mp == 0 and dif > dea:
        exchange.SetDirection("buy")         # 设置交易方向和类型
        exchange.Buy(last_close, 1)          # 开多单
        mp = 1                               # 设置虚拟持仓的值，即有多单
```

```
    if mp == 0 and dif < dea:
        exchange.SetDirection("sell")         # 设置交易方向和类型
        exchange.Sell(last_close - 1, 1)      # 开空单
        mp = -1                               # 设置虚拟持仓的值，即有空单

def main():
    while True:
        onTick()
        Sleep(1000)
```

通过本节的学习，相信你已经对 MACD 原理及计算方法有了一定的了解，你可以参照本节中的代码，试着把策略临摹下来进行测试，也可以对策略加以升级改进。

4.3　利用平均趋向指数辅助 MACD 策略

“趋势是你的朋友”这是每一个交易者都耳熟能详的箴言。但做过交易的朋友可能会有体会，趋势总是在毫无预警地开始并突然结束。那么在 CTA 策略中，如何抓住趋势并过滤震荡行情，是许多主观和量化交易者孜孜不倦的追求。在本节课程中，我们将以平均趋向指数(ADX)为滤网，分析在它量化交易中的应用。

4.3.1　什么是平均趋向指数

平均趋向指数是衡量趋势的技术工具，简称 ADX(average directional indicator)，它是由韦尔斯·怀尔德在 1978 年提出。

注意：与其他技术分析工具不同的是，ADX 并不能判断多空方向，更不能提示精确的买卖点位，它只是衡量当前趋势的强弱。

图 4.3　平均趋向指数

ADX 的默认周期参数是 14，通常在 K 线图的副图中显示。它的值是在 0~100 之间，数值越大说明上涨或下跌趋势越强力，通常当 ADX 的值大于 40 时，说明趋势强力，此时使用趋势交易才具有最大的回报潜力；当 ADX 的值小于 20 时，说明趋势疲软，并警告交易者不要使用趋势跟踪交易策略。

4.3.2 ADX 的计算方式

ADX 的计算方式比较复杂，它涉及到了：价格正向移动距离(+DM)、价格负向移动距离(-DM)、真是波动幅度(TR)、正向方向性指数(+DI)，负向方向性指数(-DI)等很多中间变量：

计算动向变化

- up：今天的最高价 – 昨天的最高价
- down：昨天的最低价 – 今天的最低价
- +DM：如果 up 大于 max(down, 0)，则+DM 等于 up，否则等于零
- -DM：如果 down 大于 max(up, 0)，则-DM 等于 down，否则等于零

计算真实波幅

- TR：max(今天最高价与今天最低价的差值，今天最高价与昨天收盘价差值的绝对值，今天最低价与昨天收盘价差值的绝对值)

计算动向指数

- +DI(14)：+DM(14)/TR(14)*100
- -DI(14)：-DM(14)/TR(14)*100

计算 ADX

- DX：((+DI14)- (-DI14)/(+DI14)+(-DI14))*100
- ADX：MA(DX，14)

虽然 ADX 的计算比较复杂，但其逻辑还是比较清晰的：up 和 down 分别代表了价格正向和负向移动距离；+DI 和-DI 分别代表用波动率修正后上涨和下跌趋势。不管趋势是上涨还是下跌，只要存在明显的趋势行情，那么+DI 和-DI 中总有一个是较大的，因此 DX 的值会随着趋势的强弱指示在 0~100 之间；最后 ADX 则是 DX 的 14 天平均线。

当+ DI 高于-DI 时，表明价格处于上升趋势。 相反，当-DI 高于+ DI 时，价格处于下降趋势。 交易者可以通过检查同一时间点的 ADX 值来确定上升趋势或下降趋势的强度。

4.3.3 策略逻辑

在之前章节中，我们使用 MACD 指标创建了一个简单的策略，虽然该策略在趋势行情中表现还可以，但是在震荡行情常常入不敷出，甚至在长期的震荡行情中资金回撤比较大。为了降低策略在震荡时期的试错成本，因此我们将在本节中将之前的 MACD 策略加入 ADX 滤网，我们来看下效果到底如何？

原策略逻辑

- 多头开仓：DIF 大于 DEA

- ❑ 空头开仓：DIF 小于 DEA
- ❑ 多头平仓：DIF 小于 DEA
- ❑ 空头平仓：DIF 大于 DEA

改进后的策略逻辑

- ❑ 多头开仓：DIF 大于 DEA，并且 ADX 上升
- ❑ 空头开仓：DIF 小于 DEA，并且 ADX 上升
- ❑ 多头平仓：DIF 小于 DEA，或者 ADX 下降
- ❑ 空头平仓：DIF 大于 DEA，或者 ADX 下降

我们在原策略逻辑基础之上，对开仓和平仓分别加入 ADX 滤网，控制在行情进入震荡时期的开仓次数。在开仓的时候 ADX 的数值必须是上升的，当开仓之后一旦 ADX 下降就平仓出局。

注意：ADX 的加入，使整个策略逻辑就设计成一个严进宽出的模式，以此来控制震荡时期的回撤幅度。

4.3.4 策略编写

根据上面更改的策略逻辑，我们可以直接在原始策略的基础之上把 ADX 滤网加入进去，虽然 ADX 的计算方法比较复杂，但可以借助 talib 库只需要几行代码就可以把 ADX 的值计算出来。因为计算 ADX 需要用带 talib，而计算 talib 库又需要用到 numpy.array 数据类型，所以我们需要在代码开头导入 talib 库和 numpy 库。

```
import talib
import numpy as np
```

在使用 talib 库计算 ADX 的时候，一共需要 4 个参数：最高价、最低价、收盘价、周期参数。所以我们还需要写一个 get_data 函数，这个函数的目的是从 K 线数组中提取出最高价、最低价、收盘价。

```
# 把 K 线数组转换成最高价、最低价、收盘价数组
# 用于转换为 numpy.array 类型数据
def get_data(bars):
    arr = [[], [], []]
    for i in bars:
        arr[0].append(i['High'])
        arr[1].append(i['Low'])
        arr[2].append(i['Close'])
    return arr
```

然后我们使用 numpy 库把普通的数组转换为 numpy.array 类型数据，最后使用 talib 库就可以计算出 ADX 的值，具体的写法可以看下面代码中的注释：

```
np_arr = np.array(get_data(bar))           # 把列表转换为numpy.array类型数据
adx_arr = talib.ADX(np_arr[0], np_arr[1], np_arr[2], 14)  # 计算 ADX 的值
```

在策略逻辑中，需要判断 ADX 的大小和是否上升下降。判断大小很简单，只需要把 ADX 具体某一天的值提取出来就可以了，在判断时候上升下降则需要只取倒数第二根和第三根 K 线的 ADX 值。

```
adx1 = adx_arr[-2]                           # 倒数第二根 K 线的 ADX 值
adx2 = adx_arr[-3]                           # 倒数第三根 K 线的 ADX 值
```

最后修改下单逻辑：

```
if mp == 1 and (dif < dea or adx1 < adx2):
    exchange.SetDirection("closebuy")        # 设置交易方向和类型
    exchange.Sell(last_close - 1, 1)         # 平多单
    mp = 0                                   # 设置虚拟持仓的值，即空仓
if mp == -1 and (dif > dea or adx1 < adx2):
    exchange.SetDirection("closesell")       # 设置交易方向和类型
    exchange.Buy(last_close, 1)              # 平空单
    mp = 0                                   # 设置虚拟持仓的值，即空仓
if mp == 0 and dif > dea and adx1 > adx2:
    exchange.SetDirection("buy")             # 设置交易方向和类型
    exchange.Buy(last_close, 1)              # 开多单
    mp = 1                                   # 设置虚拟持仓的值，即有多单
if mp == 0 and dif < dea and adx1 > adx2:
    exchange.SetDirection("sell")            # 设置交易方向和类型
    exchange.Sell(last_close - 1, 1)         # 开空单
    mp = -1                                  # 设置虚拟持仓的值，即有空单
```

4.3.5 完整策略代码

```
# 导入库
import talib
import numpy as np

mp = 0  # 定义一个全局变量，用于控制虚拟持仓

# 把 K 线数组转换成最高价、最低价、收盘价数组
# 用于转换为 numpy.array 类型数据
def get_data(bars):
    arr = [[], [], []]
    for i in bars:
        arr[0].append(i['High'])
        arr[1].append(i['Low'])
        arr[2].append(i['Close'])
    return arr

# 程序主函数
def onTick():
    _C(exchange.SetContractType, "rb000")    # 订阅期货品种
    bar = _C(exchange.GetRecords)            # 获取 K 线数组
    if len(bar) < 100:                       # 如果 K 线数组长度太小就返回
        return
    macd = TA.MACD(bar, 5, 50, 15)           # 计算 MACD 值
    dif = macd[0][-2]                        # 获取 DIF 的值，返回一个数组
```

```
    dea = macd[1][-2]                          # 获取DEA的值，返回一个数组
    np_arr = np.array(get_data(bar))           # 把列表转换为numpy.array类型数据
    adx_arr = talib.ADX(np_arr[0], np_arr[1], np_arr[2], 14)  # 计算ADX的值
    adx1 = adx_arr[-2]                         # 倒数第二根K线的ADX值
    adx2 = adx_arr[-3]                         # 倒数第三根K线的ADX值
    last_close = bar[-1]['Close']              # 获取最新价格（卖价）
    global mp                                  # 全局变量，用于控制虚拟持仓
    if mp == 1 and (dif < dea or adx1 < adx2):
        exchange.SetDirection("closebuy")      # 设置交易方向和类型
        exchange.Sell(last_close - 1, 1)       # 平多单
        mp = 0                                 # 设置虚拟持仓的值，即空仓
    if mp == -1 and (dif > dea or adx1 < adx2):
        exchange.SetDirection("closesell")     # 设置交易方向和类型
        exchange.Buy(last_close, 1)            # 平空单
        mp = 0                                 # 设置虚拟持仓的值，即空仓
    if mp == 0 and dif > dea and adx1 > adx2:
        exchange.SetDirection("buy")           # 设置交易方向和类型
        exchange.Buy(last_close, 1)            # 开多单
        mp = 1                                 # 设置虚拟持仓的值，即有多单
    if mp == 0 and dif < dea and adx1 > adx2:
        exchange.SetDirection("sell")          # 设置交易方向和类型
        exchange.Sell(last_close - 1, 1)       # 开空单
        mp = -1                                # 设置虚拟持仓的值，即有空单

def main():
    while True:
        onTick()
        Sleep(1000)
```

之前流行过一段话：站在风口猪都会飞，我们做交易也是一样。在大趋势面前，再笨的策略也能分一杯羹，所以我们要做的就是抓住大趋势并在震荡时期控制回撤。ADX与MACD配合使用，可以帮助交易者确认差异，从而提高交易的精度。

注意：MACD是一种中长线的趋势指标，在市场反复震荡时，可能会出现错误信号。增加滤网的好处是既可以在震荡行情中降低风险，又可以在趋势行情中增加你的盈利潜力，以达到最大限度地降低风险并最大化利润的目的。一句话：要想赚大钱，就一定不要与趋势为敌！

4.4 自适应动态双均线策略

对于初学者来说，策略开发最好从临摹开始，本节我们将重温经典技术分析工具考夫曼均线，并根据其常用的使用方法来构建策略，深度解析每一个计算步骤，以及如何用Python和talib库去实现它。

4.4.1　传统均线弊端

我们知道价格变化的速度本身就在变化，传统简单均线受困于固定周期参数，这使得不论市场的走势如何，短期均线灵敏度高，更贴近价格走势，但在市场震荡时期反复转向，造成频繁发出错误开平仓信号；长期均线在趋势判断上更加可靠，但在市场加速上涨或下跌时反应迟钝，造成错过最佳的买卖点。

因此虽然传统简单均线可以在一定程度适应行情，但是却很难根据市场变化去进行调整，进而更好的把握趋势。特别在长期震荡行情中，不仅得不到正收益而且付出高额的交易成本，为了解决这个问题，我们引入考夫曼创立的自适应均线。

4.4.2　考夫曼均线原理

图 4.4　各种均线对比

在《精明交易者》中，作者考夫曼（Kaufman）提出了“自适应移动平均线”，简称 AMA。该均线考虑到了市场价格变化速率，在普通均线的基础上增加了平滑系数，并自适应动态调整均线的灵敏度，可以在慢速趋势和快速趋势之间自我调整。当市场出现盘整、趋势不明显时期，AMA 倾向于慢速移动平均线。当市场波动较大，趋势明显，价格沿一个方向快速移动时，AMA 倾向于快速移动平均线。

考夫曼均线本质上是根据一段时间内的价格波动率进行调整，计算出了合适的入场阈值提供了最佳的买卖点位。也就是说，它分为两部分主逻辑，第二部分逻辑在波动率层面做了又一次自适应。从而反应市场真实的趋势，便于快速抓住趋势性上涨和下跌的时机，同时规避市场来回震荡的影响。

4.4.3　考夫曼均线计算

有经验的交易者都习惯于在趋势展开的行情中使用快速均线，在震荡较多的行情中使

用慢速均线。但如何把这个方法数量化，让程序来区分这两种行情？这里就需要引入“效率”的概念。

如果价格一致朝一个方向运行，每天收盘价的变化贡献于总的运行幅度，那么就被称为高效率；如果价格涨涨跌跌，很多次收盘价的变化相互抵消，那么就被称为低效率。这类似于物理学中的位移，如果价格在 10 天内上涨了 100 个点，我们可称为高效率，如果价格在 10 天内上涨了 10 个点，我们可以称为低效率。

第 1 步：计算价格效率

价格效率是建立在市场移动的速度和方向以及市场中噪声量的基础之上的，假设价格效率是在 0~1 之间，0 表示市场没有移动，只有噪声；1 表示市场只有移动，没有噪声。如果价格在 10 天内上涨了 100 个点，每天移动 10 个点，其价格效率就是：100 / (10 * 10) = 1；如果价格在 10 天内上涨了 10 个点，但每天震荡 10 个点，其价格效率就是：10 / (10 * 10) = 0.1。

其计算公式是：首先计算价格变动值，即当根 K 线价格与前 N 根 K 线的价格差的绝对值；然后计算价格波动值，即 N 根 K 线内，所有价格变动绝对值的总和；最后计算效率系数，即价格变动值除以价格波动值。

- 价格变动值 = abs(价格 - n 日前价格)
- 价格波动值 = sum(abs(价格 - 上一个交易日价格)，n)
- 效率系数 = 价格变动值 / 价格波动值

注意：在价格变动值一定条件下，市场波动越大，效率系数越小，此时使用慢速移动平均线更能把握整体趋势走向，因为慢速平均线不易被市场短期波动改变方向；反之，价格变动值一定条件下，市场波动越小，效率系数越大，此时应该使用快速（短期）移动平均线。

第 2 步：计算平滑系数

考夫曼用一系列的移动平均速度来描述平滑系数，其计算方式与 EMA 类似，根据价格所占权重，重新定义快速和慢速趋势速度系数，比如可以将 2 天的平均称为快速，30 天的平均称为慢速。其中：

- 快速趋势系数是：2 /(2 + 1)= 2 / 3 = 0.66667；
- 慢速趋势系数是：2 /(30 + 1) = 2 / 31 = 0.06452。它们的差值是：0.60215。
- 快速趋势系数 = 2 / (n1 + 1)
- 慢速趋势系数 = 2 / (n2 + 1)

上面公式中的 n1 和 n2 是交易周期数，并且 n1 小于 n2。默认 n1 为 2，n2 为 30。最后利用效率比率计算平滑系数，也就是：效率系数 * 0.60215 + 0.06452。

- 平滑系数 = 效率系数 * (快速 - 慢速) + 慢速

可见，当市场波动越大，趋势明显时，平滑系数更加趋向于选择快速趋势系数快速趋势系数，反之，在市场震荡盘整，趋势不明显时期，平滑系数更趋向于选择慢速趋势系数慢速趋势系数。

第 3 步：计算 AMA 值

因为在效率系数太低时，可能会取消交易，所以卡夫曼建议在计算 AMA 值之前，对最

后的平滑系数再次乘方。

- ❑ 系数 = 平滑系数 * 平滑系数
- ❑ AMA = 上一个交易日的 AMA + 系数 * (价格 - 上一个交易日的 AMA)

假设昨天的 AMA 值是 40，当前的价格是 47，它们之间有 7 个点的差值。那么在一个高效市场，其 AMA 值提高将近 3.1 个点，这几乎是差值的一半。在一个低效市场，这个差值几乎不会对 AMA 值产生影响。

4.4.4 策略逻辑

根据考夫曼的观点，AMA 相当于平滑指数，如果其方向改变就应该立刻交易。换句话说就是 AMA 上升时应该买进，AMA 下降时应该卖出。不过如果贸然以此做交易信号，可能会造成大量的无效信号，因此就需要增加一个合适的滤网，即增加另一根 AMA 均线，以双均线交叉的形式发出买卖信号。

- ❑ 多头开仓：AMA1 和 AMA2 均为向上，并且 AMA1 大于 AMA2。
- ❑ 空头开仓：AMA1 和 AMA2 均为向下，并且 AMA1 小于 AMA2。
- ❑ 多头平仓：AMA1 和 AMA2 均为向下，或者 AMA1 小于 AMA2。
- ❑ 空头平仓：AMA1 和 AMA2 均为向上，或者 AMA1 大于 AMA2。

4.4.5 策略编写

按照以上策略逻辑，开始用代码实现出来。依次打开：控制中心>策略库>新建策略>点击右上角下拉菜单选择 Python 语言，开始编写策略，注意看下面代码中的注释。

第 1 步：抱着不重复造轮子的精神，我们在计算 AMA 的值时，直接使用之前介绍过的 talib 库。因为在使用 talib 计算 AMA 的时候需要用到 numpy.array 数据，所以这里也要导入 numpy 库。

```
# 导入库
import talib
import numpy as np
```

第 2 步：编写策略框架，这个在之前的章节已经学习过，一个是 onTick 函数，另一个是 main 函数，其中在 main 函数中无限循环执行 onTick 函数，如下：

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:         # 进入无限循环模式
        onTick()        # 执行策略主函数
        Sleep(1000)     # 休眠 1 秒
```

第 3 步：定义虚拟持仓变量，量化交易中判断持仓分为两种，一种是真实的账户持仓，

另一种就是虚拟持仓，还有一种是真实持仓和虚拟持仓联合判断。实盘时我们只使用真实持仓就足够了，但这里为了简化策略，作为演示使用虚拟持仓。

```
mp = 0                  # 定义一个全局变量，用于控制虚拟持仓
```

使用虚拟持仓的原理很简单，策略运行之初默认是空仓 mp=0，当开多单后把虚拟持仓重置为 mp=1，当开空单后把虚拟持仓重置为，mp=-1，当平多单或空单后把虚拟持仓重置为 mp=0。这样我们在判断构建逻辑获取仓位时，只需要判断 mp 的值就可以了。

第 4 步：计算 AMA，因为我们是使用 talib 计算 AMA 的值，所以需要用到收盘价的 numpy.array 数据，那么其流程是：订阅期货数据>>>获取 K 线数组>>>把 K 线数组转换为收盘价数组>>>把收盘价数组转换为 numpy.array 数据>>>使用 talib 计算 AMA 值。

```
# 把 K 线数组转换成收盘价数组，用于计算 AMA 的值
def get_close(r):
    arr = []
    for i in r:
        arr.append(i['Close'])
    return arr

_C(exchange.SetContractType, "rb000")           # 订阅期货品种
bar_arr = _C(exchange.GetRecords)               # 获取 K 线数组
if len(bar_arr) < 100:                          # 如果 K 线数组长度太小就返回
    return
close_arr = get_close(bar_arr)                  # 把 K 线数组转换成收盘价数组
np_close_arr = np.array(close_arr)          # 把列表转换为 numpy.array
ama1 = talib.KAMA(np_close_arr, 10).tolist()    # 计算短期 AMA
ama2 = talib.KAMA(np_close_arr, 100).tolist()  # 计算长期 AMA
```

请看上面代码所示，第 1~6 行是 get_close 函数，这个函数的作用是把 K 线数组转换成收盘价数组，主要用于计算 AMA。第 8 行~第 15 行是按照流程计算 AMA 值。

注意：计算 AMA 需要一个周期参数，如果 K 线长度小于这个周期参数，就不能计算其 AMA 值。所以在第 10 行和第 11 行，我们加了一个判断 K 线数组的长度，也就是说如果 K 线数据不足以计算 AMA 值时直接跳过。

第 5 步：开平仓，首先获取当前最新价格，因为在使用下单接口函数时，必须指定交易价格。K 线数组最后一个数据的收盘价就是最新价格。然后指定交易的方向类型，即：开多、开空、平多、平空。调用 exchange.SetDirection()函数，分别传入："buy"、"sell"、"closebuy"、"closesell"。最后下单之后重置持仓状态 mp 的值。

```
last_close = close_arr[-1]                      # 获取最新价格
global mp                                       # 全局变量，用于控制虚拟持仓
if mp == 1 and is_cross(ama2, ama1):
    exchange.SetDirection("closebuy")           # 设置交易方向和类型
    exchange.Sell(last_close - 1, 1)            # 平多单
    mp = 0                                      # 设置虚拟持仓的值，即空仓
if mp == -1 and is_cross(ama1, ama2):
    exchange.SetDirection("closesell")          # 设置交易方向和类型
    exchange.Buy(last_close, 1)                 # 平空单
    mp = 0                                      # 设置虚拟持仓的值，即空仓
```

```
if mp == 0 and is_cross(ama1, ama2):
    exchange.SetDirection("buy")                  # 设置交易方向和类型
    exchange.Buy(last_close, 1)                   # 开多单
    mp = 1                                        # 设置虚拟持仓的值，即有多单
if mp == 0 and is_cross(ama2, ama1):
    exchange.SetDirection("sell")                 # 设置交易方向和类型
    exchange.Sell(last_close - 1, 1)              # 开空单
    mp = -1                                       # 设置虚拟持仓的值，即有空单
```

4.4.6 完整策略代码

```
# 导入库
import talib
import numpy as np

mp = 0  # 定义一个全局变量，用于控制虚拟持仓

# 把K线数组转换成收盘价数组，用于计算AMA的值
def get_close(r):
    arr = []
    for i in r:
        arr.append(i['Close'])
    return arr

# 判断两根AMA交叉
def is_cross(arr1, arr2):
    if arr1[-2] < arr2[-2] and arr1[-1] > arr2[-1]:
        return True

# 程序主函数
def onTick():
    _C(exchange.SetContractType, "rb000")   # 订阅期货品种
    bar_arr = _C(exchange.GetRecords)       # 获取K线数组
    if len(bar_arr) < 100:                  # 如果K线数组长度过小于就直接返回
        return
    close_arr = get_close(bar_arr)          # 把K线数组转换成收盘价数组
    np_close_arr = np.array(close_arr)      # 把列表转换为numpy.array
    ama1 = talib.KAMA(np_close_arr, 10).tolist()    # 计算短期AMA
    ama2 = talib.KAMA(np_close_arr, 100).tolist()   # 计算长期AMA
    last_close = close_arr[-1]              # 获取最新价格
    global mp                               # 全局变量，用于控制虚拟持仓
    if mp == 1 and is_cross(ama2, ama1):
        exchange.SetDirection("closebuy")   # 设置交易方向和类型
        exchange.Sell(last_close - 1, 1)    # 平多单
        mp = 0                              # 设置虚拟持仓的值，即空仓
    if mp == -1 and is_cross(ama1, ama2):
```

```
        exchange.SetDirection("closesell")  # 设置交易方向和类型
        exchange.Buy(last_close, 1)         # 平空单
        mp = 0                              # 设置虚拟持仓的值，即空仓
    if mp == 0 and is_cross(ama1, ama2):
        exchange.SetDirection("buy")        # 设置交易方向和类型
        exchange.Buy(last_close, 1)         # 开多单
        mp = 1                              # 设置虚拟持仓的值，即有多单
    if mp == 0 and is_cross(ama2, ama1):
        exchange.SetDirection("sell")       # 设置交易方向和类型
        exchange.Sell(last_close - 1, 1)    # 开空单
        mp = -1                             # 设置虚拟持仓的值，即有空单

def main():
    while True:
        onTick()
        Sleep(1000)
```

上面的代码构建了自适应双均线，并从细节上逐行阐述其中的原理和算法，最后又以双自适应均线创建一个简单的 CTA 策略。整体来看自适应双均线比普通均线更加稳定又不失灵活性。

注意：AMA 本意上是用来替代普通均线，以更好地拟合市场价格走势，单独一根 AMA 并没有质的提升，所以需要额外配置一个过滤器，这个过滤器的选择基于市场波动状况来选择。

4.5 日内高低点突破策略

之前听过一句话：要想赚大钱必须学会长线持仓，但如果要赚快钱就要学会日内交易。如今的量化交易范围之广令人惊叹，各种交易策略层出不穷，其中最为流行的就是日内交易策略。

日内交易是一种快进快出的交易方式，由于可以控制隔夜风险的特点，得到了很多交易者的推崇和接受。为了帮助大家了解日内交易，丰富策略仓库，本节我们将深入了解商品期货中最为流行的日内策略之一日内高低点突破策略。

4.5.1 什么是日内交易

日内交易的目的是以更小的损失，来获取当天市场微小的价格波动所带来的利润。它是指开仓和平仓在同一天内或同一交易时间段内完成的交易方式，开仓和平仓可以是单次，也可以是多次，只要是开平仓在同一个交易日前结束就行。

理论上日内交易不承担隔夜的跳空风险，相对来说是一种较完美的低风险交易策略，但实际上并非如此，虽然日内交易回避了跳空所带来的风险，同时也错失了跳空所带来的利润。但如果以正确的方式交易，通过配合不同的交易规则，日内交易往往也能产生丰厚

的回报。

4.5.2 策略逻辑

我们知道判断上涨趋势最简单的方法是，当前低点比前一个低点更高，当前高点也比前一个高点更高；同理下跌趋势最简单的方法是，当前低点比前一个低点更低，当前高点也比前一个高点更低。但如果仅仅以高低点的比较去判断趋势的涨跌，这未免太过简陋，因为价格可能在一个点上来回跳动几十次甚至上百次，从而导致交易过于频繁。

所以我们需要设定一个价格区间来过滤这些日常杂波，来对简单的高低点突破策略进行完善。我们可以根据历史行情所出现的最高价和最低价，组成一个包含上轨和下轨的通道。根据顺势交易的原则，当价格突破上轨时多头开仓，当价格突破下轨时空头开仓。

- 多头开仓：当前无持仓，时间是在开盘与收盘前 10 分钟之间，并且价格大于上轨
- 空头开仓：当前无持仓，时间是在开盘与收盘前 10 分钟之间，并且价格小于下轨
- 多头平仓：当前持多单，价格小于下轨，或者时间大于 14：50
- 空头平仓：当前持空单，价格大于上轨，或者时间大于 14：50

有人统计过，大部分的窄幅止损都是无效的，小空间的止损会频繁打脸，所以我们要做的就是设计一个宽幅止损：

- 如果多头开仓后，价格不升反跌，我们所要做的不是立即止损，而是等待观望，直到价格跌破下轨才止损出局；
- 空头开仓后也是如此，当价格不跌反升，继续等待价格是否会自我修正，直到跌破上轨才止损出局。

4.5.3 策略编写

第 1 步：导入 time 库

```
import time
```

因为日内策略在编写的时候，要判断当前的时间来控制开平仓逻辑，这个策略在设计的时候是：只能在 9 点 30 分之 14 点 50 分之间开仓，14 点 50 分之后全部平仓，其余的时间都过滤掉了。所以就需要引入 time 时间库。

第 2 步：编写策略框架

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:         # 进入无限循环模式
        onTick()        # 执行策略主函数
        Sleep(1000)     # 休眠 1 秒
```

编写策略就像盖房子一样，先把地基和框架搭建好，再往里面填充东西。我们这里用

了两个函数，一个是 main 主函数，另一个是 onTick 函数，程序会先从 main 函数执行代码，在 main 函数中，我们用了一个无限循环模式，重复执行 onTick 函数。

第 3 步：设置全局变量

```
mp = on_line = under_line = 0
```

在全局变量中，mp 主要用于控制虚拟持仓，判断持仓一般分为两种，一种是真实的账户持仓，另一种就是虚拟持仓，还有一种是真实持仓和虚拟持仓联合判断。实盘时我们只使用真实持仓就足够了，但这里为了简化策略，作为演示使用虚拟持仓。on_line 和 under_line 分别记录上轨和下轨。

第 4 步：处理时间

```
def can_time(hour, minute):
    hour = str(hour)
    minute = str(minute)
    if len(minute) == 1:
        minute = "0" + minute
    return int(hour + minute)

_C(exchange.SetContractType, "MA888")                  # 订阅期货品种
bar_arr = _C(exchange.GetRecords)                      # 获取 K 线数组
if len(bar_arr) < 10:
    return
time_new = bar_arr[-1]['Time']                         # 获取当根 K 线的时间戳
time_local_new = time.localtime(time_new / 1000)       # 处理时间戳
hour_new = int(time.strftime("%H", time_local_new))    # 获取小时
minute_new = int(time.strftime("%M", time_local_new))  # 获取分钟
day_new = int(time.strftime("%d", time_local_new))     # 当前 K 线日期
time_previous = bar_arr[-2]['Time']                    # 获取上根 K 线的时间戳
previous = time.localtime(time_previous / 1000)        # 处理时间戳
day_previous = int(time.strftime("%d", previous))      # 上根 K 线日期
```

注意：处理时间一共用于两个地方：一个是判断当前时间是否在我们规定的交易时间内，如果当前是在这个时间之内，并且已经达到了开仓条件就开仓，如果不是在这个时间之内，并且当前有持仓就平掉所有持仓，达到收盘前平仓的目的。

另一个是判断当前 K 线是不是最新交易日的 K 线，因为我们的策略逻辑是每当新的一天 K 线出现时，就重置上下轨。通过对比两个 K 线的时间戳来重置 on_line 和 under_line 的值，也就是说上下轨通道是在不断变化的。

第 5 步：计算高低点上下轨

```
global mp, on_line, under_line                   # 引入全局变量
high = bar_arr[-2]['High']                       # 获取上根 K 线的最高价
low = bar_arr[-2]['Low']                         # 获取上根 K 线的最低价
if day_new != day_previous:                      # 如果是最新一根 K 线
    on_line = high * up                          # 重置上轨
    under_line = low * down                      # 重置下轨
can_trade = can_time(hour_new, minute_new)
if can_trade < 930:                              # 如果不是在规定交易的时间内
```

```
    if high > on_line:                                  # 如果上根 K 线最高价大于上轨
        on_line = high * up                             # 重置上轨
    if low < under_line:                                # 如果上根 K 线最低价小于下轨
        under_line = low * down                         # 重置上轨
if on_line - under_line < 10:                           # 如果上轨与下轨的差小于 10
    return
```

计算高低点上下轨的逻辑其实非常简单：如果当前是第一根 K 线，那么 on_line 和 under_line 的值分别是最高价和最低价，如果当前 K 线是最新交易日的 K 线，就重置 on_line 和 under_line 的值为最高价和最低价；

一旦在规定的交易时间内，on_line 和 under_line 的值就固定不变了，除非在这个时间之外并且如果上根 K 线最高价大于 on_line 就重置为最新的最高价；如果上根 K 线最低价小于 under_line 就重置为最新的最低价。

第 5 步：下单交易

在下单交易之前，我们先获取当前最新价格，因为在下单时需要在函数中传入下单价格。然后使用 if 语句，根据之前设计的交易逻辑，先是判断当前的持仓状态，然后再判断当前时间状态，以及最新价格与上下轨的相互位置关系，最后下单交易并重置虚拟持仓状态。

```
close_new = bar_arr[-1]['Close']                # 获取最新价格（卖价），用于开平仓
# 如果持多单，并且价格小于下轨或者非规定的交易时间
if mp > 0 and (close_new < under_line or can_trade > 1450):
    exchange.SetDirection("closebuy")           # 设置交易方向和类型
    exchange.Sell(close_new - 1, 1)             # 平多单
    mp = 0                                      # 设置虚拟持仓的值，即空仓
# 如果持空单，并且价格大于上轨或者非规定的交易时间
if mp < 0 and (close_new > on_line or can_trade > 1450):
    exchange.SetDirection("closesell")          # 设置交易方向和类型
    exchange.Buy(close_new, 1)                  # 平空单
    mp = 0                                      # 设置虚拟持仓的值，即空仓
if mp == 0 and 930 < can_trade < 1450:          # 如果当前无持仓且在交易时间内
    if close_new > on_line:                     # 如果价格大于上轨
        exchange.SetDirection("buy")            # 设置交易方向和类型
        exchange.Buy(close_new, 1)              # 开多单
        mp = 1                                  # 设置虚拟持仓的值，即有多单
    elif close_new < under_line:                # 如果价格小于下轨
        exchange.SetDirection("sell")           # 设置交易方向和类型
        exchange.Sell(close_new - 1, 1)         # 开空单
        mp = -1                                 # 设置虚拟持仓的值，即有空单
```

预测今天下午的天气是很容易的，但是要想预测这个月内的天气却很难。日内交易不需要较长的持仓周期，所承受的市场波动风险较低，尽管这种交易方式不符合每个人的风格，但对于那些风险较为敏感的交易者来说，日内交易还是相当值得深入研究。

4.5.4　完整策略代码

```
# 导入库
import time

# 定义全局变量：虚拟持仓、上轨、下轨
mp = on_line = under_line = 0

# 处理时间函数
def can_time(hour, minute):
    hour = str(hour)
    minute = str(minute)
    if len(minute) == 1:
        minute = "0" + minute
    return int(hour + minute)

def onTick():
    _C(exchange.SetContractType, "MA888")                  # 订阅期货品种
    bar_arr = _C(exchange.GetRecords)                      # 获取 K 线数组
    if len(bar_arr) < 10:
        return
    time_new = bar_arr[-1]['Time']                         # 获取当根K线的时间戳
    time_local_new = time.localtime(time_new / 1000)           # 处理时间戳
    hour_new = int(time.strftime("%H", time_local_new))        # 获取小时
    minute_new = int(time.strftime("%M", time_local_new))      # 获取分钟
    day_new = int(time.strftime("%d", time_local_new)) # 当前 K 线日期
    time_previous = bar_arr[-2]['Time']                    # 获取上根K线的时间戳
    previous = time.localtime(time_previous / 1000)    # 处理时间戳
    day_previous = int(time.strftime("%d", previous))  # 上根 K 线日期
    global mp, on_line, under_line          # 引入全局变量
    high = bar_arr[-2]['High']              # 获取上根 K 线的最高价
    low = bar_arr[-2]['Low']                # 获取上根 K 线的最低价
    if day_new != day_previous:             # 如果是最新一根 K 线
        on_line = high * up                 # 重置上轨
        under_line = low * down             # 重置下轨
    can_trade = can_time(hour_new, minute_new)
    if can_trade < 930:                     # 如果不是在规定交易的时间内
        if high > on_line:                  # 如果上根 K 线最高价大于上轨
            on_line = high * up             # 重置上轨
        if low < under_line:                # 如果上根 K 线最低价小于下轨
            under_line = low * down         # 重置上轨
    if on_line - under_line < 10:           # 如果上轨与下轨的差小于 10
        return
    close_new = bar_arr[-1]['Close']        # 获取最新价格（卖价），用于开平仓
    # 如果持多单，并且价格小于下轨或者非规定的交易时间
    if mp > 0 and (close_new < under_line or can_trade > 1450):
```

```
        exchange.SetDirection("closebuy")    # 设置交易方向和类型
        exchange.Sell(close_new - 1, 1)      # 平多单
        mp = 0                               # 设置虚拟持仓的值，即空仓
    # 如果持空单，并且价格大于上轨或者非规定的交易时间
    if mp < 0 and (close_new > on_line or can_trade > 1450):
        exchange.SetDirection("closesell")   # 设置交易方向和类型
        exchange.Buy(close_new, 1)           # 平空单
        mp = 0                               # 设置虚拟持仓的值，即空仓
    if mp == 0 and 930 < can_trade < 1450:  # 如果当前无持仓且在交易时间内
        if close_new > on_line:              # 如果价格大于上轨
            exchange.SetDirection("buy")     # 设置交易方向和类型
            exchange.Buy(close_new, 1)       # 开多单
            mp = 1                           # 设置虚拟持仓的值，即有多单
        elif close_new < under_line:         # 如果价格小于下轨
            exchange.SetDirection("sell")    # 设置交易方向和类型
            exchange.Sell(close_new - 1, 1)  # 开空单
            mp = -1                          # 设置虚拟持仓的值，即有空单

def main():
    while True:
        onTick()
        Sleep(1000)
```

4.6　增强版唐奇安通道策略

提起唐奇安通道，很多人都会联想到海龟交易法则，这也许是有史以来最成功的交易员培训课程。海龟们用神奇的交易系统赚了成百上千万美元，直到 1983 年海龟交易法则解密，人们才发现这个神奇的交易系统用的是修正版的唐奇安通道。

图 4.5　唐奇安通道

4.6.1 唐奇安通道简介

原始的唐奇安通道（Donchianchannel）规则其实很简单，它先设置一条阻力线和一条支撑线，阻力线由过去 N 天的最高价的最大值形成；支撑线由过去 N 天的最低价的最小值形成。

- 唐奇安上阻力线：由过去 N 天的当日最高价的最大值
- 唐奇安下支撑线：由过去 N 天的当日最低价的最小值

注意：图中唐奇安通道阻力线和支撑线，在外观上与布林带比较相像，只不过布林带的波动比较灵敏，而唐奇安通道则是直上直下。唐奇安通道可以衡量市场的波动性，一般来说通道宽度越宽，市场的波动就越大，通道宽度越窄，市场的波动性也就越小。

4.6.2 原始策略逻辑

除了具有衡量市场波动率这个功能外，它的主要作用是帮助交易者确定买入和卖出时机。因为唐奇安通道是根据最高价和最低价计算出来的，通道的宽窄又随着价格的变化自动调整，所以大多数时候价格是在通道之内运行，很少突破其上下轨道的。

也就是说，价格并不会随意突破阻力线和支撑线，但如果有效突破，那就预示着大行情可能将会出现。此时交易者可以根据支撑和阻力线，确定买进或卖出的具体时机。比如：当价格向上突破阻力线就买入，当价格跌破支撑线就卖出。

之所以原始策略逻辑在早期的金融市场大行其道，是因为最初的市场和市场参与者不太成熟。现如今散户都已经用上了量化交易，策略的同质性，导致策略低效，也就是说如果一个策略使用的人越多，在市场上的效率就越低。所以我们有必要对原始策略逻辑加以改进，让策略更加与众不同。

4.6.3 改进后的策略逻辑

我们分别从优化开仓方式和止盈止损这两个方面加以改进。首先是开仓方式，做过突破策略的交易者可能会有体会，行情突破阻力线，本来我们是要做多的，结果刚一入场，价格却急转直下，本来看着是一个很好的机会，最后弄了一个措手不及。

大家想一想假突破究竟是怎么来的，怎么总是那么巧合的发生，就好像庄家顶着自己的账号操纵市场一样。其实这是策略同质化的原因，因为前期高低点是固定的，大家都有目共睹，结果大家都等着价格向上突破时买进，该买的都已经买了，此时买力消失，价格自然而然下跌。另外大户也在盯着这个关键点，他也知道散户会在突破时买进，等散户买完不就可以做空割韭菜了么。

所以为了解决这个问题，我们在支撑线和阻力线分别增加一个系数，这样避免与大多数策略参数一致，造成的同质化现象，导致策略低效。另外我们知道，中国的期货市场总是涨的时候涨的缓，跌的时候跌的急，那么可以对支撑线和阻力线设置不同的系数，让策略更合理的适应当前市场环境。

- 唐奇安上轨：由过去 N 天的最高价的最大值*上涨系数
- 唐奇安下轨：由过去 N 天的最低价的最小值*下跌系数
- 唐奇安中轨：(唐奇安上轨 + 唐奇安下轨) / 2

然后是改进止盈止损的方式，原始的唐奇安通道规则是，价格突破阻力线开多单，把止盈止损放在支撑线这个位置；价格跌破支撑线开空单，把止盈止损放在阻力线这个位置。但是这里面有一个问题，假如市场波动率比较大，唐奇安通道上轨与下轨的距离就会加宽，此时就会增加止损的成本和损失一部分浮盈。

- 开多：如果当前无持仓，并且价格突破唐奇安上轨
- 开空：如果当前无持仓，并且价格跌破唐奇安下轨
- 平多：如果当前持多单，并且价格跌破唐奇安中轨
- 平空：如果当前持空单，并且价格突破唐奇安中轨

那么折中的办法是，可以根据唐奇安通道的上轨和下轨，再计算出一条中轨，这样把止盈止损放在中轨的位置，无论是持有多单还是空单，只要价格反向突破中轨及时止盈止损，这样不仅可以减少止损时所付出的成本，同时保护你未平仓的利润免受重大不利价格波动的影响。

4.6.4 策略编写

到目前为止，你应该很好地理解了原始唐奇安通道规则，以及我们将要改进它的方法。现在我们就用代码编写这个交易策略吧。

第 1 步：编写策略框架

策略框架其实就是两个函数，其中 main 函数是整个程序的入口函数，也就是说策略开始执行的时候，会先执行 main 函数；另外一个是 onTick 函数，onTick 只是一个函数的名字，当然你也可以自由命名，onTick 函数里面主要编写策略逻辑。整个框架其实就是在 main 函数中重复执行 onTick 函数。

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:        # 进入无限循环模式
```

```
    onTick()         # 执行策略主函数
    Sleep(1000)      # 休眠 1 秒
```

第 2 步：定义全局变量

我们这个策略只需要一个控制虚拟持仓的全局变量，所谓的虚拟持仓指的是理论持仓而非真实持仓，无论开仓还是平仓，我们都假设订单已经完全成交。这么做的目的是简化初学者的入门门槛。

```
# 定义全局变量
mp = 0                  # 用于控制虚拟持仓
```

第 3 步：处理 K 线数据

我们在前面已经定义过，上轨是过去 N 天的最高价的最大值，下轨是过去 N 天的最低价的最小值。要想计算这两个值，首先要先获取基础 K 线数据。但是在使用 GetRecords 方法获取完基础 K 线数据之后，先不要慌着计算上轨和下轨，而是先把数据处理一下。

注意：因为我们在计算上轨和下轨的时候需要 N 个 K 线，如果 K 线数量太少就不能计算了，所以要加一个 if 条件，判断当前 K 线是否满足我们所需要的数量，如果不满足就直接返回，等待下一次循环。另外我们还需要从 K 线数组中提取当前最新价格和上根 K 线的收盘价，最新价格主要用于开平仓，上根 K 线收盘价主要用于判断开平仓信号。

有的朋友可能会问，为什么不直接使用最新的价格来判断开平仓信号呢？这是因为如果使用最新价格来判断，就可能出现信号反复的问题，同时也为了规避未来函数和偷价这些常见的量化交易问题，所以我们的策略在设计上是：当前 K 线出信号，下根 K 线发单。

```
_C(exchange.SetContractType, "rb000")              # 订阅期货品种
bar_arr = _C(exchange.GetRecords)                  # 获取 K 线数组
if len(bar_arr) < 60:
   return
close_new = bar_arr[-1]['Close']                   # 获取最新价格（卖价）
close_last = bar_arr[-2]['Close']                  # 上根 K 线收盘价
```

第 4 步：计算上轨、下轨、中轨

在发明者量化交易平台中，已经内置了 talib 库中的 Highest 函数和 Lowest 函数，所以我们直接调用这两个函数就可以计算上轨和下轨的值。但因为我们是使用上根 K 线收盘价为基准，来判断它与上轨、下轨、中轨的位置关系来开平仓，所以在计算上轨和下轨之前需要先删除 K 线数组中的最后一个元素。

```
bar_arr.pop()                                      # 删除数组最后一个数据
on_line = TA.Highest(bar_arr, 55, 'High') * 0.999  # 计算唐奇安上轨
under_line = TA.Lowest(bar_arr, 55, 'Low') * 1.001 # 计算唐奇安下轨
middle_line = (on_line + under_line) / 2           # 计算唐奇安中轨
```

第 5 步：下单交易

要想在函数内使用外部的全局变量，需要在使用这个变量之前，先用 global 关键字把变量引入。注意下面代码中的注释，整个代码流程是使用 if 语句，然后根据我们之前定义的策略逻辑来编写。有两个地方需要注意，一个是在下单之前需要先设置下单的类型方向，也就是先调用 SetDirection 函数。另一个是在下单之后，要把虚拟持仓变量 mp 重新赋值。

```
global mp                                          # 引入全局变量
# 如果持多单，并且价格小于下轨
```

```
if mp > 0 and close_last < middle_line:
    exchange.SetDirection("closebuy")                   # 设置交易方向和类型
    exchange.Sell(close_new - 1, 1)                     # 平多单
    mp = 0                                              # 设置虚拟持仓的值，即空仓
# 如果持空单，并且价格大于上轨
if mp < 0 and close_last > middle_line:
    exchange.SetDirection("closesell")                  # 设置交易方向和类型
    exchange.Buy(close_new, 1)                          # 平空单
    mp = 0                                          # 设置虚拟持仓的值，即空仓
if mp == 0:                                         # 如果当前无持仓
    if close_last > on_line:                        # 如果价格大于上轨
        exchange.SetDirection("buy")                # 设置交易方向和类型
        exchange.Buy(close_new, 1)                  # 开多单
        mp = 1                                      # 设置虚拟持仓的值，即有多单
    elif close_last < under_line:                   # 如果价格小于下轨
        exchange.SetDirection("sell")               # 设置交易方向和类型
        exchange.Sell(close_new - 1, 1)             # 开空单
        mp = -1                                     # 设置虚拟持仓的值，即有空单
```

4.6.5 完整策略代码

```
mp = 0                                              # 定义全局变量,用于控制虚拟持仓

def onTick():
    _C(exchange.SetContractType, "rb000")           # 订阅期货品种
    bar_arr = _C(exchange.GetRecords)               # 获取 K 线数组
    if len(bar_arr) < 60:
        return
    close_new = bar_arr[-1]['Close']                # 获取最新价格（卖价）
    close_last = bar_arr[-2]['Close']               # 上根 K 线收盘价
    bar_arr.pop()                                   # 删除数组最后一个数据
    on_line = TA.Highest(bar_arr, 55, 'High') * 0.999   # 计算唐奇安上轨
    under_line = TA.Lowest(bar_arr, 55, 'Low') * 1.001  # 计算唐奇安下轨
    middle_line = (on_line + under_line) / 2        # 计算唐奇安中轨
    global mp                                       # 引入全局变量
    if mp > 0 and close_last < middle_line:         # 如果持多单，并且价格小于下轨
        exchange.SetDirection("closebuy")           # 设置交易方向和类型
        exchange.Sell(close_new - 1, 1)             # 平多单
        mp = 0                                      # 设置虚拟持仓的值，即空仓
    if mp < 0 and close_last > middle_line:         # 如果持空单，并且价格大于上轨
        exchange.SetDirection("closesell")          # 设置交易方向和类型
        exchange.Buy(close_new, 1)                  # 平空单
        mp = 0                                      # 设置虚拟持仓的值，即空仓
    if mp == 0:                                     # 如果当前无持仓
        if close_last > on_line:                    # 如果价格大于上轨
            exchange.SetDirection("buy")            # 设置交易方向和类型
```

```
            exchange.Buy(close_new, 1)             # 开多单
            mp = 1                                 # 设置虚拟持仓的值，即有多单
        elif close_last < under_line:              # 如果价格小于下轨
            exchange.SetDirection("sell")          # 设置交易方向和类型
            exchange.Sell(close_new - 1, 1)        # 开空单
            mp = -1                                # 设置虚拟持仓的值，即有空单

# 程序入口
def main():
    while True:                                    # 进入无线循环模式
        onTick()                                   # 执行策略主函数
        Sleep(1000)                                # 休眠 1 秒
```

唐奇安通道之所以流传至今，一定有它独特的道理。但随着市场的转变，我们也要与时俱进而不是贸然使用。随着你对交易认知的提升，你会发现改进的方法非常多。这就是交易的魅力所在，每一位交易者都应该是一位探险者，大胆探索小心求证，长此以往就一定能有适合自己的交易方法。最后配合合理的风险管理，方能成为以为成功的交易者。

4.7 hans123 日内突破策略

HANS123 策略最早主要应用于外汇市场，其交易方式比较简单，属于趋势突破系统，这种交易方式可以在趋势形成的第一时间入场，因此得到很多交易员的青睐，迄今为止 HANS123 已经扩展了许多版本，今天我们就一起来认识 HANS123 策略。

4.7.1 策略原理

有的人认为，上午开盘是市场分歧最大的时候，很容易出现价格走势方向不明，宽幅震荡的情况。经过 30 分钟左右的时间，市场充分消化完各种隔夜信息，价格走势将趋于理性回归正常。也就是说：前 30 分钟左右的行情走势，基本上构成了今天整体的交易格局。

- 上轨：开盘 30 分钟内最高价
- 下轨：开盘 30 分钟内最低价
- 中轨：(上轨 + 下轨) / 2

开盘后 30 分钟内产生的相对高低点，就形成道氏理论中所说的有效高低点，HANS123 策略就是以此建立的交易逻辑。在国内期货市场上，上午 09:00 开盘，09:30 就可以大致判断今天究竟是做多还是做空。当价格向上突破高点时，价格很容易继续上升；价格向下突破低点时，价格很容易继续下跌。

- 多头开仓：当前无持仓，并且价格向上突破上轨
- 空头开仓：当前无持仓，并且价格向下突破下轨

注意：虽然突破策略能在趋势形成时，第一时间入场。但这个优势也是把双刃剑，入场灵敏导致的结果就是，价格突破失败。所以设置止损是非常有必要的。同时为了达到赢冲输缩的策略逻辑，也要设置止盈。

- 多头止损：当前持多单，达到亏损金额
- 空头止损：当前持空单，达到亏损金额
- 多头止盈：当前持多单，达到盈利金额
- 空头止盈：当前持空单，达到盈利金额

4.7.2 策略编写

第 1 步：编写策略框架

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:                                          # 进入无限循环模式
        onTick()                                         # 执行策略主函数
        Sleep(1000)                                      # 休眠 1 秒
```

编写策略框架，这个在之前的章节已经学习过，一个是 onTick 函数，另一个是 main 函数，其中在 main 函数中无限循环执行 onTick 函数。

第 2 步：定义全局变量

```
up_line = down_line = trade_count = 0
```

因为上轨和下轨只是在 09:30 这个时间点才统计计算，其余时间不做统计，所以我们需要把这两个变量写到循环的外面。另外为了在日内交易中限制交易次数，也把 trade_count 变量写到循环外面。在 onTick 策略主函数内使用这两个全局变量之前，需要使用 global 关键字引用。

第 3 步：获取数据

```
_C(exchange.SetContractType, "rb000")                    # 订阅期货品种
bar_arr = _C(exchange.GetRecords, PERIOD_M1)             # 获取 1 分钟 K 线数组
current_close = bar_arr[-1]['Close']                     # 获取最新价格
```

要想获取数据，首先要使用发明者量化 API 中的 SetContractType 函数订阅期货品种，然后使用 GetRecords 函数获取 K 线数组，也可以在使用 GetRecords 函数时传入指定 PERIOD_M11 分钟的 K 线数组。

紧接着就是获取最新的价格，用来判断当前价格与上下轨之间的位置关系，同时在下单交易使用 Buy 或 Sell 函数时，需要传入指定的价格。另外别忘了过滤 K 线数量，因为如果 K 线过少，就会出现无法计算指标报错。

第 4 步：处理时间函数

```
def current_time(bar_arr):
    current = bar_arr[-1]['Time']                        # 获取当前 K 线时间戳
    time_local = time.localtime(current / 1000)          # 处理时间戳
    hour = time.strftime("%H", time_local)               # 格式时间戳获取小时
```

```
    minute = time.strftime("%M", time_local)          # 格式时间戳获取分钟
    if len(minute) == 1:
        minute = "0" + minute
    return int(hour + minute)
```

在计算上下轨和下单交易时，需要判断当前的时间是否符合我们指定的交易时间，所以为了方便判断，我们需要处理一下当前 K 线的具体小时数和分钟数。

第 5 步：计算上下轨

```
global up_line, down_line, trade_count                # 引入全局变量
current_time = current_time(bar_arr)                  # 处理时间
if current_time == 930:                               # 如果时间是 09:30
    bar_arr = _C(exchange.GetRecords, PERIOD_D1)      # 获取日 K 线数组
    up_line = bar_arr[-1]['High']                     # 前 30 根 K 线最高价
    down_line = bar_arr[-1]['Low']                    # 前 30 根 K 线最低价
    trade_count = 0                                   # 重置交易次数为 0
```

要计算上下轨，首先要引入我们之前定义的全局变量，使用 global 关键字即可。想象一下我们需要计算的是 09:00~09:30 之间的最高价和最低价。因为我们是使用 1 分钟的 K 线数据，也就是说当时间为 09:30 的时候，刚好有 30 根 K 线，那么我们直接计算这 30 根 K 线的最高价和最低价就可以了。

第 6 步：获取持仓

```
position_arr = _C(exchange.GetPosition)               # 获取持仓数组
if len(position_arr) > 0:                             # 如果持仓数组长度大于 0
    position_arr = position_arr[0]                    # 获取持仓字典数据
    if position_arr['ContractType'][:2] == 'rb':      # 如果持仓品种等于订阅品种
        if position_arr['Type'] % 2 == 0:             # 如果是多单
            position = position_arr['Amount']         # 赋值持仓数量为正数
        else:
            position = -position_arr['Amount']        # 赋值持仓数量为负数
        profit = position_arr['Profit']               # 获取持仓盈亏
else:
    position = 0                                      # 赋值持仓数量为 0
    profit = 0                                        # 赋值持仓盈亏为 0
```

注意：持仓状态牵涉到策略逻辑，在真实的交易环境中最好是使用 GetPosition 函数，获取真实的持仓信息，包括：持仓方向、持仓盈亏、持仓数量等等。

第 7 步：下单交易

为了避免策略出现逻辑错误，最好是将平仓逻辑写到开仓逻辑的前面。在这个策略中，开仓时首先要判断当前的持仓状态、是否在指定的交易时间内，然后再判断当前价格与上下轨之间的关系。平仓则是先判断是否接近尾盘，或者是否达到止盈止损条件。

```
# 如果临近收盘或者达到止盈止损
if current > 1450 or profit > 100 * 3 or profit < -100:
    if position > 0:                                  # 如果持多单
        exchange.SetDirection("closebuy")             # 设置交易方向和类型
        exchange.Sell(current_close - 1, 1)           # 平多单
    if position < 0:                                  # 如果持空单
```

```
        exchange.SetDirection("closesell")          # 设置交易方向和类型
        exchange.Buy(current_close + 1, 1)          # 平空单
# 如果当前无持仓，并且小于指定交易次数，并且在指定交易时间内
if position == 0 and trade_count < 3 and 930 < current < 1450:
    if current_close > up_line:                     # 如果价格大于上轨
        exchange.SetDirection("buy")                # 设置交易方向和类型
        exchange.Buy(current_close + 1, 1)          # 开多单
        trade_count = trade_count + 1               # 交易次数加一次
    if current_close < down_line:                   # 如果价格小于下轨
        exchange.SetDirection("sell")               # 设置交易方向和类型
        exchange.Sell(current_close - 1, 1)         # 开空单
        trade_count = trade_count + 1               # 交易次数加一次
```

4.7.3 完整策略代码

```
# 定义全局变量：上轨、下轨、当天交易次数
up_line = down_line = trade_count = 0

def current_time(bar_arr):
    current = bar_arr[-1]['Time']                   # 获取当前 K 线时间戳
    time_local = time.localtime(current / 1000)     # 处理时间戳
    hour = time.strftime("%H", time_local)          # 格式化时间戳，并获取小时
    minute = time.strftime("%M", time_local)        # 格式化时间戳，并获取分钟
    if len(minute) == 1:
        minute = "0" + minute
    return int(hour + minute)

def onTick():
    _C(exchange.SetContractType, "rb000")           # 订阅期货品种
    bar_arr = _C(exchange.GetRecords, PERIOD_M1)    # 获取 1 分钟 K 线数组
    current_close = bar_arr[-1]['Close']            # 获取最新价格
    global up_line, down_line, trade_count          # 引入全局变量
    current = current_time(bar_arr)                 # 处理时间
    if current == 930:                              # 如果 K 线时间是 09:30
        bar_arr = _C(exchange.GetRecords, PERIOD_D1)# 获取日 K 线数组
        up_line = bar_arr[-1]['High']               # 前 30 根 K 线最高价
        down_line = bar_arr[-1]['Low']              # 前 30 根 K 线最低价
        trade_count = 0                             # 重置交易次数为 0
    position_arr = _C(exchange.GetPosition)         # 获取持仓数组
    if len(position_arr) > 0:                       # 如果持仓数组长度大于 0
        position_arr = position_arr[0]              # 获取持仓字典数据
        if position_arr['ContractType'][:2] == 'rb':# 如果持仓品种等于 rb
            if position_arr['Type'] % 2 == 0:       # 如果是多单
                position = position_arr['Amount']   # 赋值持仓数量为正数
            else:
                position = -position_arr['Amount'] # 赋值持仓数量为负数
```

```
        profit = position_arr['Profit']          # 获取持仓盈亏
    else:
        position = 0                             # 赋值持仓数量为 0
        profit = 0                               # 赋值持仓盈亏为 0
    # 如果临近收盘或者达到止盈止损
    if current > 1450 or profit > 100 * 3 or profit < -100:
        if position > 0:                         # 如果持多单
            exchange.SetDirection("closebuy")    # 设置交易方向和类型
            exchange.Sell(current_close - 1, 1)  # 平多单
        if position < 0:                         # 如果持空单
            exchange.SetDirection("closesell")   # 设置交易方向和类型
            exchange.Buy(current_close + 1, 1)   # 平空单
    # 如果当前无持仓，并且小于指定交易次数，并且在指定交易时间内
    if position == 0 and trade_count < 3 and 930 < current < 1450:
        if current_close > up_line:              # 如果价格大于上轨
            exchange.SetDirection("buy")         # 设置交易方向和类型
            exchange.Buy(current_close + 1, 1)   # 开多单
            trade_count = trade_count + 1        # 交易次数加一次
        if current_close < down_line:            # 如果价格小于下轨
            exchange.SetDirection("sell")        # 设置交易方向和类型
            exchange.Sell(current_close - 1, 1)  # 开空单
            trade_count = trade_count + 1        # 交易次数加一次

# 策略入口函数
def main():
    while True:                                  # 无限循环
        onTick()                                 # 执行策略主函数
        Sleep(1000)                              # 休眠 1 秒
```

以上就是 HANS123 策略原理和代码解析，实际上 HANS123 策略提供了一个较好的入场时机，出场时机，这也是一种入场较早的交易模式，配合适当过滤技术，或可提高其胜算。你也可以根据自己对市场的认知和对交易的理解加以改进，或者也可以根据品种波动率来优化止盈止损等参数，以达到更好的效果。

4.8　菲阿里四价策略

在期货市场，价格呈现一切。几乎所有的技术分析，如均线、布林线、MACD、KDJ 等等，这些都是以价格为基础，通过特定的方法计算。包括基本面分析也是如此，通过分析近期和远期价差、期货和现货升贴水、上下游库存等等数据，计算当前价格是否合理，并预估未来的价格。既然如此，为什么不直接研究价格呢？今天我们讲的菲阿里四价策略就是完全根据价格来做出卖决定。

4.8.1 菲阿里简介

菲阿里是一位日本的交易者，主要偏向于商品期货日内主观交易。其大名远扬是在2001年的罗宾斯(ROBBINS-TAICOM)期货冠军大赛中，以1098%的成绩获得冠军。并且在之后的两年里再以709%、1131%的成绩夺冠。从成绩就知道，菲阿里是一个非常优秀的交易者。

幸运的是，菲阿里在《1000%的男人》这本书中详尽叙述了他的交易方法，菲阿里四价策略正是后人总结他的交易方法。虽然只是从外在形式加上自己的理解模仿了一部分，并不代表菲阿里全部交易精髓，但至少可以帮助我们在构建策略时拓展思路。

4.8.2 策略逻辑

菲阿里四价策略是一种比较简单的趋势性日内交易策略，四价分别是指：昨天高点、昨天低点、昨日收盘价、今天开盘价。从书中的交易笔记来看，菲阿里不使用任何分析工具，而是大量应用阻溢线的概念，也就是通常我们所说的阻力线和支撑线。

- ❑ 阻力线 = 昨日最高价
- ❑ 支撑线 = 昨日最低价

注意：对于阻力线和支撑线的定义，他使用的是昨日最高价和昨日最低价，可以视为昨天价格的波动范围，这也意味着多头或空头只有足够的力量时，才会有效突破阻力线和支撑线。并且一旦有突破这个波动范围，则说明价格背后的动能较大，后续走势可能会沿最小阻力线运动的概率较大。

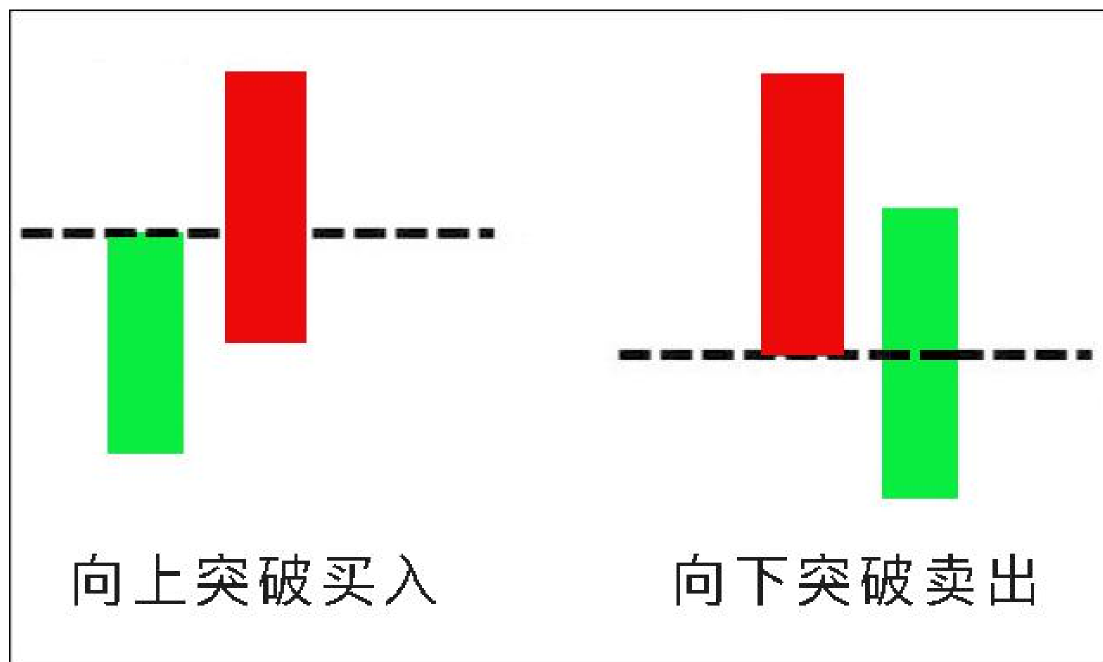

图 4.6　突破买入与卖出

- ❑ 多头开仓：价格突破阻力线

- ❑ 空头开仓：价格突破支撑线

如果开盘价处于阻力线和支撑线之间，当价格向上突破阻力线就建立多头，当价格向下突破支撑线就建立空头。如果一切顺利，则一直持仓到收盘。这样做的好处是，符合了充分非必要条件，即突破不一定上涨/下跌，但上涨/下跌一定会突破，也就是始终守在行情发生的必经之路伺机而动，因为较大行情的上涨和下跌一旦出现，势必要突破阻力线和支撑线的。

当然这也是出错率最高的方法，因为很多时候价格只是暂时性的突破了关键位置，如果贸然开仓可能会面临价格随时反向运动的风险。这时就需要设置一些过滤条件，限制假突破造成的来回开平仓问题。另外在交易周期上也尽量避免波动过于混乱的 5 分钟周期以下 K 线。

但是开仓后，盈利了还好，如果遇到亏损，总不能从小亏损一直积累到大亏损，才在收盘时平仓吧，这样显然不合理。所以我们对于平仓有两种处理方式：收盘平仓和止损平仓。如果 K 线上破高点或下破低点后又回到原来的区间内，就要考虑止损了。

- ❑ 多头平仓：收盘前 5 分钟或达到多头止损线
- ❑ 空头平仓：收盘前 5 分钟或达到空头止损线

实际上菲阿里在主观交易中，还有很多交易方法，包括：开盘后先涨后跌，跌破开盘价做空，止损设在之前上涨的最高点；开盘后先跌后涨，突破开盘价做多，止损设在之前下跌的最低点。动手能力强的朋友可以在自己的策略中增改。

到这里你会发现，对于一天的价格走势来说，收盘价相对于开盘价的涨跌，其概率接近于 50%。菲阿里的交易方法在胜率上就利于不败之地，再加上行情顺利的时候一直持仓到收盘，在行情不符合自己的预期时及时止损。形成了截断亏损，让利润奔跑的正向交易方式，这也是长期交易下来积累利润的原因。

4.8.3 策略编写

第 1 步：编写策略框架

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:                                     # 进入无限循环模式
        onTick()                                    # 执行策略主函数
        Sleep(1000)                                 # 休眠 1 秒
```

定义一个 main 函数和一个 onTick 函数，然后再 main 函数中，写入 while 循环，重复执行 onTick 函数。

第 2 步：导入库

```
import time                                         # 用于转换时间格式
```

因为我们这个是日内策略，到时候需要判断 K 线时间戳，如果有持仓，并且临近收盘

时平仓出局。那么我们就直接 import time 就可以了。

第 3 步：获取基础数据

```
_C(exchange.SetContractType, "rb888")           # 订阅期货品种
bar_arr = _C(exchange.GetRecords, PERIOD_D1)    # 获取日线数组
if len(bar_arr) < 2:                            # 如果小于 2 根 K 线
    return                                      # 返回继续等待数据
yh = bar_arr[-2]['High']                        # 昨日最高价
yl = bar_arr[-2]['Low']                         # 昨日最低价
today_open = bar_arr[-1]['Open']                # 当日开盘价
```

菲阿里四价需要用到四个数据：昨日最高价、昨日最低价、昨日收盘价、当日开盘价。因为这些都是日线级别的数据，所以我们在使用 GetRecords 的时候，可以直接传入 PERIOD_D1 参数，表明我们要获取的是日 K，这样无论你的策略加载的是哪个周期的数据，它始终获取的都是日线级别的数据。

另外，细心的朋友可能已经发现，为什么这一次在调用 GetRecords 的时候，代码的写法跟以前不一样？这次我们使用的是发明者量化平台内置的重试函数_C()。使用这个函数的好处是，该函数会一直调用指定函数到成功返回，这样可以避免直接使用 GetRecords 函数时，没有获取到数据导致报错的情况。

第 4 步：处理时间和获取最新价格

```
bar_arr = _C(exchange.GetRecords)               # 获取当前设置周期 K 线数组
current = bar_arr[-1]['Time']                   # 获取当前 K 线时间戳
local = time.localtime(current / 1000)          # 处理时间戳
hour = int(time.strftime("%H", local))          # 格式化时间戳，并获取小时
minute = int(time.strftime("%M", local))        # 格式化时间戳，并获取分钟
price = bar_arr[-1]['Close']                    # 获取最新价格
```

既然是获取当前的时间，那么肯定是使用获取当前设置周期的 K 线数据更为合适，所以需要重新使用一次 GetRecords，这次我们同样也是使用_C()重试函数，在不传入参数的情况下，就是默认获取当前设置周期的 K 线数组。另外，获取最新价格的目的是，计算交易逻辑和下单。

第 5 步：处理时间

```
def trade_time(hour, minute):
    minute = str(minute)
    if len(minute) == 1:
        minute = "0" + minute
    return int(str(hour) + minute)
```

注意：之所以创建这个函数，是因为我们在开仓之前，需要判断当前时间，是否在我们规定的交易时间之内，以及有持仓的时候，当前时间是否临近收盘。在第 4 步中，我们已经获取到了当前 K 线小时和分钟，为了方便比较，我们采用小时加分钟的方法，比如：

- 如果 K 线时间是 9：05，那么 trade_time 返回的结果就是 905
- 如果 K 线时间是 14：30，那么 trade_time 返回的结果就是 1430

第 6 步：设置虚拟持仓

```
mp = 0
```

第 7 步：设置止损

```
 # 设置多头止损
if today_open / yh > 1.005:                     # 如果当天开盘价大于昨天最高价
    long_stop = yh                              # 设置多头止损价为昨天最高价
elif today_open / yh < 0.995:                   # 如果当天开盘价小于昨天最高价
    long_stop = today_open                      # 设置多头止损价为当天开盘价
else:                                           # 如果当天开盘价接近昨天最高价
    long_stop = (yh + yl) / 2                   # 设置多头止损为昨天中间价
# 设置空头止损
if today_open / yl < 0.995:                     # 如果当天开盘价小于昨天最低价
    short_stop = yl                             # 设置空头止损价为昨天最低价
elif today_open / yl > 1.005:                   # 如果当天开盘价大于昨天最低价
    short_stop = today_open                     # 设置空头止损价为当天开盘价
else:                                           # 如果当天开盘价接近昨天最低价
    short_stop = (yh + yl) / 2                  # 设置空头止损为昨天中间价
```

在大多数情况下，价格突破阻力线和支撑线，就把止损设置到当前开盘价这个位置。但是这里面有个问题：如果当天开盘价大于阻力线，而价格往下走；或者当天开盘价小于支撑线，而价格往上走，会造成逻辑错误，导致频繁开平仓。为了解决这个问题，我们需要根据当天开盘与阻力线和支撑线的位置关系，分别设置不同的止损价格。如果当天开盘价大于昨天最高价 0.5%，那么就把多头的止损设置在昨天的最高价；

如果当天开盘价在阻力线和支撑线之间，那么多头的止损价格还是当天的开盘价；如果当天开盘价接近于昨天最高价，那么就把多头的止损设置在昨天的中间价。设置空头的止损也是根据这个道理。

第 8 步：下单交易

```
trading = trade_time(hour, minute)
if mp > 0:                                      # 如果当前持有多单
# 如果当前价格小于多头止损线，或者超过规定的交易时间
    if price < long_stop or trading > 1450:
        exchange.SetDirection("closebuy")       # 设置交易方向和类型
        exchange.Sell(price - 1, 1)             # 平多单
        mp = 0                                  # 重置虚拟持仓
if mp < 0:                                      # 如果当前持有空单
# 如果当前价格大于空头止损线，或者超过规定的交易时间
    if price > short_stop or trading > 1450:
        exchange.SetDirection("closesell")      # 设置交易方向和类型
        exchange.Buy(price, 1)                  # 平空单
        mp = 0                                  # 重置虚拟持仓
# 如果当前无持仓，并且在规定的交易时间内
if mp == 0 and 930 < trading < 1450:
    if price > yh:                              # 如果当前价格大于昨天最高价
        exchange.SetDirection("buy")            # 设置交易方向和类型
        exchange.Buy(price, 1)                  # 开多单
        mp = 1                                  # 重置虚拟持仓
    elif price < yl:                            # 如果价格小于昨天最低价
        exchange.SetDirection("sell")           # 设置交易方向和类型
        exchange.Sell(price - 1, 1)             # 开空单
```

```
        mp = -1                                         # 重置虚拟持仓
```

注意：为了避免逻辑错误，最好是把平仓逻辑写到开仓逻辑的前面。

4.8.4 完整策略代码

```
import time                                             # 导入库，用于转换时间格式

mp = 0                                                  # 虚拟持仓

def trade_time(hour, minute):
    minute = str(minute)
    if len(minute) == 1:
        minute = "0" + minute
    return int(str(hour) + minute)

def onTick():
    _C(exchange.SetContractType, "rb888")                   # 订阅期货品种
    bar_arr = _C(exchange.GetRecords, PERIOD_D1)            # 获取日线数组
    if len(bar_arr) < 2:                                    # 如果小于 2 根 K 线
        return                                              # 返回继续等待数据
    yh = bar_arr[-2]['High']                                # 昨日最高价
    yl = bar_arr[-2]['Low']                                 # 昨日最低价
    today_open = bar_arr[-1]['Open']                        # 当日开盘价
    bar_arr = _C(exchange.GetRecords)                       # 获取当前设置周期 K 线数组
    current = bar_arr[-1]['Time']                           # 获取当前 K 线时间戳
    local = time.localtime(current / 1000)                  # 处理时间戳
    hour = int(time.strftime("%H", local))                  # 格式化时间戳，并获取小时
    minute = int(time.strftime("%M", local))                # 格式化时间戳，并获取分钟
    price = bar_arr[-1]['Close']                            # 获取最新价格
    global mp
    # 设置多头止损
    if today_open / yh > 1.005:                         # 如果当天开盘价大于昨天最高价
        long_stop = yh                                  # 设置多头止损价为昨天最高价
    elif today_open / yh < 0.995:                       # 如果当天开盘价小于昨天最高价
        long_stop = today_open                          # 设置多头止损价为当天开盘价
    else:                                               # 如果当天开盘价接近昨天最高价
        long_stop = (yh + yl) / 2                       # 设置多头止损为昨天中间价
    # 设置空头止损
    if today_open / yl < 0.995:                         # 如果当天开盘价小于昨天最低价
        short_stop = yl                                 # 设置空头止损价为昨天最低价
    elif today_open / yl > 1.005:                       # 如果当天开盘价大于昨天最低价
        short_stop = today_open                         # 设置空头止损价为当天开盘价
    else:                                               # 如果当天开盘价接近昨天最低价
        short_stop = (yh + yl) / 2                      # 设置多头止损为昨天中间价
    # 下单交易
    trading = trade_time(hour, minute)
```

```
    if mp > 0:                                  # 如果当前持有多单
        # 如果当前价格小于多头止损线，或者超过规定的交易时间
        if price < long_stop or trading > 1450:
            exchange.SetDirection("closebuy")   # 设置交易方向和类型
            exchange.Sell(price - 1, 1)         # 平多单
            mp = 0                              # 重置虚拟持仓
    if mp < 0:                                  # 如果当前持有空单
        # 如果当前价格大于空头止损线，或者超过规定的交易时间
        if price > short_stop or trading > 1450:
            exchange.SetDirection("closesell")  # 设置交易方向和类型
            exchange.Buy(price, 1)              # 平空单
            mp = 0                              # 重置虚拟持仓
    # 如果当前无持仓，并且在规定的交易时间内
    if mp == 0 and 930 < trading < 1450:
        if price > yh:                          # 如果当前价格大于昨天最高价
            exchange.SetDirection("buy")        # 设置交易方向和类型
            exchange.Buy(price, 1)              # 开多单
            mp = 1                              # 重置虚拟持仓
        elif price < yl:                        # 如果价格小于昨天最低价
            exchange.SetDirection("sell")       # 设置交易方向和类型
            exchange.Sell(price - 1, 1)         # 开空单
            mp = -1                             # 重置虚拟持仓

def main():
    while True:                                 # 无限循环
        onTick()                                # 执行策略主函数
        Sleep(1000)                             # 休眠 1 秒
```

虽然距离比赛结束已经有近 20 年之久了，但以今天的眼光看那时的交易，毫无过时感。但需要注意的是，策略仅仅作为思路拓展，不能直接用于实盘。对于菲阿里策略来说，它提供了一个很好的入场参考工具，我们可以根据自己对市场的认知做更深的开发。

4.9 阿隆指标 AROON 策略

阿隆（Aroon）是一个很独特的技术指标，“Aroon”一词来自梵文，寓意为“黎明曙光”。它不像 MA、MACD、KDJ 那样广为人所熟悉，它推出的时间更晚，直到 1995 年才被图莎尔·钱德（Tushar Chande）发明出来，作者还发明了钱德动量摆动指标（CMO）和日内动量指数（IMI）。如果说一个技术指标知道的人越多，使用的人也越多，那么其赚钱能力也越低，那么相对新颖的阿隆指标则恰恰相反，站在这个角度看这是一个不错的选择。

4.9.1 阿隆指标简介

阿隆指标通过计算当前 K 线距离前最高价和最低价之间的 K 线数量，来帮助交易者预

测价格走势与趋势区域的相对位置关系变化。它有两部分组成，即：阿隆上线（AroonUp）和阿隆下线（AroonDown），这两条线在 0~100 之间上下移动，虽然命名为上线和下线，但从图表上看并不像 BOLL 指标那样是真正意义上的上线和下线。如下图就是阿隆指标：

图 4.7 阿隆指标

4.9.2 阿隆指标的计算方法

阿隆指标要求首先要设置一个时间周期参数，就像设置均线周期参数一样，在传统行情软件中，这个周期数是 14，当然这个周期参数并不是固定的，你还可以设置为 10 或者 50 等等。为了方便理解，暂且把这个时间周期参数定义为：N。确定 N 之后，我们就可以计算出阿隆上线（AroonUp）和阿隆下线（AroonDown），具体的计算公式如下：

- 阿隆上线= [(设置的周期参数 - 最高价后的周期数) / 计算的周期数] * 100
- 阿隆下线= [(设置的周期参数 - 最低价后的周期数) / 计算的周期数] * 100

注意：从这个公式中，我们就能大致看出，阿隆指标的思想。那就是：有多少个周期，价格在近期高 / 低点之下，辅助预测当前趋势是否会延续，同时衡量当前趋势的强弱。如果我们把这个指标归类的话，很明显它是属于趋势跟踪类型。但是与其他趋势跟踪型指标不同的是，它更重视时间而不是价格。

4.9.3 如何使用阿隆指标

阿隆上线（AroonUp）和阿隆下线（AroonDown）反映的是当前时间与之前最高价或最低价的远近，如果时间越近值就越大，如果时间越远值就越小。并且当两条线发生交叉就预示着价格方向可能会发生改变，如果 AroonUp 在 AroonDown 之上说明价格处于上涨趋势，

未来价格可能会进一步上涨；如果 AroonDown 在 AroonUp 之上说明价格处于下跌趋势，未来价格可能会进一步下跌。

同时我们还可以设置几个固定的值，来精确入场时机。我们知道阿隆指标是一直在 0~100 之间上下运行，那么在市场处于上涨趋势，也就是 AroonUp 在 AroonDown 之上时，当 AroonUp 大于 50，说明市场上涨的趋势已经形成，未来价格可能会继续上涨；当 AroonUp 下穿 50 时，说明价格上涨的动力正在减弱，未来价格可能会震荡和下跌。

反之在市场处于下跌趋势，也就是 AroonDown 在 AroonUp 之上时，当 AroonDown 大于 50，说明市场下跌趋势已经形成，未来价格可能会继续下跌；当 AroonDown 下穿 50 时，说明价格下跌的动力正在减弱，未来价格可能会震荡和上涨。那么根据上面两段理论，我们可以把买卖条件罗列为：

- 当 AroonUp 大于 AroonDown，并且 AroonUp 大于 50，多头开仓；
- 当 AroonUp 小于 AroonDown，或者 AroonUp 小于 50，多头平仓；
- 当 AroonDown 大于 AroonUp，并且 AroonDown 大于 50，空头开仓；
- 当 AroonDown 小于 AroonUp，或者 AroonDown 小于 50，空头平仓；

4.9.4　基于阿隆指标构建交易策略

理清交易逻辑后，我们就可以用代码去实现了，依次打开：fmz.com > 登录 > 控制中心 > 策略库 > 新建策略 > 点击右上角下拉菜单选择 Python 语言，开始编写策略，注意看下面代码中的注释。

第 1 步：编写策略框架

我们知道在量化交易中，程序是不断获取数据、处理数据、下单交易这样的循环过程，所以我们继续使用之前讲过的 main 函数和 onTick 函数，其中在 main 函数中无限循环执行 onTick 函数。如下：

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:         # 进入无限循环模式
        onTick()        # 执行策略主函数
        Sleep(1000)     # 休眠 1 秒
```

第 2 步：导入库

另外，在计算 AROON 时，需要用到 talib 库，我们直接用 import 一行代码导入。因为在使用 talib 计算时，必须先把数据处理成 numpy.array 类型，所以也到导入 numpy 库。

```
import talib
import numpy as np
```

第 3 步：定义虚拟持仓变量

量化交易中判断持仓分为两种，一种是真实的账户持仓，另一种就是虚拟持仓，还有

一种是真实持仓和虚拟持仓联合判断。实盘时我们只使用真实持仓就足够了，但这里为了简化策略，作为演示使用虚拟持仓。

```
mp = 0  # 用于控制虚拟持仓
```

使用虚拟持仓的原理很简单，策略运行之初默认是空仓 mp=0，当开多单后把虚拟持仓重置为 mp=1，当开空单后把虚拟持仓重置为，mp=-1，当平多单或空单后把虚拟持仓重置为 mp=0。这样我们在判断构建逻辑获取仓位时，只需要判断 mp 的值就可以了。

第 4 步：计算阿隆指标

计算阿隆指标，首先要获取基础数据，但前提是先要订阅数据，也就是订阅具体的合约代码，你可以订阅指数或者主力连续，甚至还可以订阅具体交割月份的合约代码。然后是获取 K 线数组，K 线数组是一个包含开高低收、成交量和时间的序列数据，同时也是计算大部分指标的基础数据。

在获取 K 线数组之后，紧接着就需要判断一下 K 线数组的长度，因为如果 K 线数组太短，不足以计算指标时就会出现异常。所以我们在这里使用 if 语句，判断如果 K 线数组小于指标参数时，就直接返回。

在使用 talib 计算指标时，它所传入的参数是 numpy.array 类型数据，所以还要把 K 线数组中的必要数据提取出来，并转换成 numpy.array 类型数据。这里我们自定义一个 get_data 函数，先别必要的数据提取出来。

```
# 把 K 线数组转换成最高价和最低价数组，用于转换为 numpy.array 类型数据
def get_data(bars):
    arr = [[], []]
    for i in bars:
        arr[0].append(i['High'])
        arr[1].append(i['Low'])
    return arr

exchange.SetContractType("ZC000")                  # 订阅期货品种
bars = exchange.GetRecords()                       # 获取 K 线数组
if len(bars) < cycle_length + 1:                   # 如果 K 线数量过少就直接返回
    return
# 把列表转换为 numpy.array 类型数据，用于计算 AROON 的值
np_arr = np.array(get_data(bars))
aroon = talib.AROON(np_arr[0], np_arr[1], 20)  # 计算阿隆指标
aroon_up = aroon[1][len(aroon[1]) - 2]         # 阿隆指标上线倒数第 2 根数据
aroon_down = aroon[0][len(aroon[0]) - 2]       # 阿隆指标下线倒数第 2 根数据
```

talib 在计算阿隆指标时，需要三个参数，依次是：最高价 numpy.array 类型数据、最低价 numpy.array 类型数据、时间周期。所以我们在自定义 get_data 函数中只需要把 K 线数组中的最高价和最低价提取出来就可以了，并把它们都转换成 numpy.array 类型数据。

紧接着，就可以计算阿隆指标了，直接调用 talib.AROON 方法并传入参数。计算后的阿隆指标是一个二维数组，所以我们分别把阿隆指标上线和下线分别提取出来，以便于判断开平仓逻辑。

第 5 步：下单交易

在下单交易之前，我们要先获取当前最新价格，因为在下单时需要在函数中传入下单价格。还需要引入全局变量 mp，该变量主要用于控制虚拟仓位。剩下的就是根据之前的策略逻辑，使用 if 语句编写下单交易。

```
close0 = bars[len(bars) - 1].Close      # 获取当根 K 线收盘价
global mp                               # 全局变量，用于控制虚拟仓位
# 如果当前空仓，并且阿隆上线大于下线，并且阿隆上线大于 50
if mp == 0 and  aroon_up > aroon_down and aroon_up > 50:
    exchange.SetDirection("buy")        # 设置交易方向和类型
    exchange.Buy(close0, 1)             # 开多单
    mp = 1                              # 设置虚拟持仓的值，即有多单
# 如果当前空仓，并且阿隆下线大于上线，并且阿隆下线小于 50
if mp == 0 and aroon_down > aroon_up and aroon_down > 50:
    exchange.SetDirection("sell")       # 设置交易方向和类型
    exchange.Sell(close0 - 1, 1)        # 开空单
    mp = -1                             # 设置虚拟持仓的值，即有空单
# 如果当前持有多单，并且阿隆上线小于下线或者阿隆上线小于 50
if mp > 0 and  (aroon_up < aroon_down or aroon_up < 50):
    exchange.SetDirection("closebuy")   # 设置交易方向和类型
    exchange.Sell(close0 - 1, 1)        # 平多单
    mp = 0                              # 设置虚拟持仓的值，即空仓
# 如果当前持有空单，并且阿隆下线小于上线或者阿隆下线小于 50
if mp < 0 and (aroon_down < aroon_up or aroon_down < 50):
    exchange.SetDirection("closesell")  # 设置交易方向和类型
    exchange.Buy(close0, 1)             # 平空单
    mp = 0                              # 设置虚拟持仓的值，即空仓
```

万事俱备之后，就可以判断策略逻辑并开平仓下单交易了。在判断策略逻辑时肯定是使用 if 语句，先判断 mp 的持仓状态，然后再判断阿隆上线和下线的相互位置关系。注意看上图中的代码注释。

注意：在期货交易下单之前，先指定交易的方向类型，即：开多、开空、平多、平空。调用 exchange.SetDirection()函数，分别传入：“buy”、“sell”、“closebuy”、“closesell”。最后下单之后重置持仓状态 mp 的值。

4.9.5 完整策略

```
import talib
import numpy as np

# 外部参数
# cycle_length = 100

# 定义全局变量
mp = 0  # 用于控制虚拟持仓

# 把 K 线数组转换成最高价和最低价数组，用于转换为 numpy.array 类型数据
```

```
def get_data(bars):
    arr = [[], []]
    for i in bars:
        arr[0].append(i['High'])
        arr[1].append(i['Low'])
    return arr


# 策略主函数
def onTick():
    exchange.SetContractType("ZC000")    # 订阅期货品种
    bars = exchange.GetRecords()         # 获取 K 线数组
    if len(bars) < cycle_length + 1:     # 如果 K 线数组的长度太小，所以直接返回
        return
    np_arr = np.array(get_data(bars))    # 把列表转换为 numpy.array 类型数据
    aroon = talib.AROON(np_arr[0], np_arr[1], 20)  # 计算阿隆指标
    aroon_up = aroon[1][len(aroon[1]) - 2]         # 阿隆上线倒数第 2 根数据
    aroon_down = aroon[0][len(aroon[0]) - 2]       # 阿隆下线倒数第 2 根数据
    close0 = bars[len(bars) - 1].Close             # 获取当根 K 线收盘价
    global mp                                      # 全局变量，虚拟仓位
    # 如果当前空仓，并且阿隆上线大于下线，并且阿隆上线大于 50
    if mp == 0 and  aroon_up > aroon_down and aroon_up > 50:
        exchange.SetDirection("buy")               # 设置交易方向和类型
        exchange.Buy(close0, 1)                    # 开多单
        mp = 1                                     # 设置虚拟持仓的值为有多单
    # 如果当前空仓，并且阿隆下线大于上线，并且阿隆下线小于 50
    if mp == 0 and aroon_down > aroon_up and aroon_down > 50:
        exchange.SetDirection("sell")              # 设置交易方向和类型
        exchange.Sell(close0 - 1, 1)               # 开空单
        mp = -1                                    # 设置虚拟持仓的值为有空单
    # 如果当前持有多单，并且阿隆上线小于下线或者阿隆上线小于 50
    if mp > 0 and  (aroon_up < aroon_down or aroon_up < 50):
        exchange.SetDirection("closebuy")          # 设置交易方向和类型
        exchange.Sell(close0 - 1, 1)               # 平多单
        mp = 0                                     # 设置虚拟持仓的值，即空仓
    # 如果当前持有空单，并且阿隆下线小于上线或者阿隆下线小于 50
    if mp < 0 and (aroon_down < aroon_up or aroon_down < 50):
        exchange.SetDirection("closesell")         # 设置交易方向和类型
        exchange.Buy(close0, 1)                    # 平空单
        mp = 0                                     # 设置虚拟持仓的值，即空仓

# 程序入口
def main():
    while True:                                    # 进入无限循环模式
        onTick()                                   # 执行策略主函数
        Sleep(1000)                                # 休眠 1 秒
```

在策略中我们固定了一部分参数，如：aroonUp 或 aroonDown 大于小于 50，造成策略的滞后性，很多情况下是行情上涨或下跌一段时间才开平仓买卖。这样虽然提高了胜率，减少了最大回撤率，但也错过了很多收益，这也印证了盈亏同源的道理。

4.10 简易波动 EMV 策略

与其他技术指标不同，简易波动（Ease of Movement Value）反映的是价格、成交量、人气的变化，它是一种将价格与成交量变化相结合的技术，它通过衡量单位成交量的价格变动，形成一个价格波动指标。当市场人气聚集，交易活跃时提示买入信号；当成交量低迷，市场能量即将耗尽时提示卖出信号。

注意：简易波动 EMV 根据等量图和压缩图的原理设计而成，它的核心理念是：市场价格仅在发生趋势转折或即将转折时，才会消耗大量能量，外在表现就是成交量变大。当价格在上升的过程中，由于推波助澜的作用，不会消耗太多的能量。虽然这个理念与量价同升的观点相悖，但的确有其独特的地方。

4.10.1 EMV 计算公式

第 1 步：计算 mov_mid

$$movmid = \frac{TH + TL}{2} - \frac{YH + YL}{2}$$

其中 TH 代表当天最高价，TL 代表当天最低价，YH 代表前日最高价，YL 代表前日最低价。那么如果 MID > 0 意味着今天的平均价高于昨天的平均价。

第 2 步：计算 ratio

$$ratio = \frac{TVOL/10000}{TH - TL}$$

其中 TVOL 代表当天交易量，TH 代表当天最高价，TL 代表当天最低价。

第 3 步：计算 emv

$$emv = \frac{movmid}{ratio}$$

4.10.2 EMV 用法

EMV 的作者认为，巨量上涨伴随的是能量的快速枯竭，上涨往往不会持续太久；反而温和的成交量，能够保存一定的能量，往往使上涨持续更久。一旦上涨趋势形成，较少的成交量就能推动价格上涨，EMV 的数值就会升高。

一旦下跌趋势行情形成，往往伴随的是无量或少量下跌，EMV 的数值就会下降。如果价格处于震荡行情或者价格上涨和下跌都伴随较大成交量时，EMV 的数值也会接近于零。

因此你会发现，EMV 在大部分行情中都处于零轴下方，这也是这个指标的一大特色。站在另一个角度看，EMV 重视大趋势且能够产生足够利润的行情。

EMV 的用法相当简单，只要看 EMV 是否穿越零轴即可，当 EMV 在 0 以下时，代表市场弱市；当 EMV 在 0 以上时，代表市场强市。让 EMV 由负数转为正数时应该买进；当 EMV 由正数转为负数时应该卖出。其特点是不仅能较好的避免市场中的震荡行情，而且还能在趋势行情启动的时候及时入场。但由于 EMV 反映的是价格在变动时的成交量的变化情况，所以仅对中长期走势有作用。对于短线或交易周期比较小的行情 EMV 的效果很差。

4.10.3 策略实现

第 1 步：编写策略框架

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:       # 进入无限循环模式
        onTick()      # 执行策略主函数
        Sleep(1000)   # 休眠 1 秒
```

发明者量化(FMZ.COM)采用轮训模式，首先需要定义一个 main 函数和一个 onTick 函数，main 函数是策略的入口函数，程序会从 main 函数开始逐行执行代码。在 main 函数中，写入 while 循环，重复执行 onTick 函数，所有的策略核心代码都写在 onTick 函数中。

第 2 步：获取持仓数据

```
# 获取持仓数量
def get_position():
    position = 0                                    # 赋值持仓数量为 0
    position_arr = _C(exchange.GetPosition)         # 获取持仓数组
    if len(position_arr) > 0:                       # 如果持仓数组长度大于 0
        for i in position_arr:                      # 遍历持仓数组
            if i['ContractType'][:2] == 'IH':       # 如果持仓品种等于订阅品种
                if i['Type'] % 2 == 0:              # 如果是多单
                    position = i['Amount']          # 赋值持仓数量为正数
                else:
                    position = -i['Amount']         # 赋值持仓数量为负数
    return position                                 # 返回持仓量
```

因为在这个策略中，只使用了实时的持仓数量，为了方便维护，这里使用 get_position 封装了持仓量，如果当前持有多单就返回正数，如果当前持有空单就返回负数。

第 3 步：获取 K 线数据

```
# 获取数据
exchange.SetContractType('IH000')                   # 订阅期货品种
```

```
bars_arr = exchange.GetRecords()              # 获取 K 线数组
if len(bars_arr) < 10:                        # 如果 K 线数量小于 10 根
    return
```

在获取具体的 K 线数据之前，首先要先订阅具体的合约，使用发明者量化的 SetContractType 函数，并传入合约代码即可，如果想知道该合约的其他信息，也可以使用一个变量来接收这个数据。接着使用 GetRecords 函数就可以获取 K 线数据，因为返回的是一个数组，所以我们使用变量 bars_arr 来接受它。

第 4 步：计算 emv

```
# 计算 emv
bar1 = bars_arr[-2]                    # 获取上一根 K 线数据
bar2 = bars_arr[-3]                    # 获取前一根 K 线数据
# 计算 mov_mid 的值
mov_mid = (bar1['High'] + bar1['Low'])/2 - (bar2['High'] + bar2['Low'])/2
if bar1['High'] != bar1['Low']:        # 如果被除数不为 0
    # 计算 ratio 的值
    ratio = (bar1['Volume'] / 10000) / (bar1['High'] - bar1['Low'])
else:
    ratio = 0
# 如果 ratio 的值大于 0
if ratio > 0:
    emv = mov_mid / ratio
else:
    emv = 0
```

注意：在这里我们并没有使用最新的价格来计算 EMV 的值，而是采用相对滞后的当前 K 线出信号，下根 K 线发单的方法。这么做的目的是让回测更接近于实盘交易。

我们知道，尽管现在量化交易软件已经非常先进了，但还是很难做到完全模拟真实的实盘 Tick 环境，特别是面对回测 Bar 级超长数据时，所以就采用这个折中的方法。

第 5 步：下单交易

```
# 下单交易
current_price = bars_arr[-1]['Close']  # 最新价格
position = get_position()              # 获取最新持仓量
if position > 0:                                        # 如果持有多单
    if emv < 0:                                         # 如果价格小于牙齿
        exchange.SetDirection("closebuy")               # 设置交易方向和类型
        exchange.Sell(round(current_price - 0.2, 2), 1) # 平多单
if position < 0:                                        # 如果持有空单
    if emv > 0:                                         # 如果价格大于牙齿
        exchange.SetDirection("closesell")              # 设置交易方向和类型
        exchange.Buy(round(current_price + 0.2, 2), 1)  # 平空单
if position == 0:                                       # 如果无持仓
    if emv > 0:                                         # 如果价格大于上唇
        exchange.SetDirection("buy")                    # 设置交易方向和类型
        exchange.Buy(round(current_price + 0.2, 2), 1)  # 开多单
    if emv < 0:                                         # 如果价格小于下巴
```

```
        exchange.SetDirection("sell")                          # 设置交易方向和类型
        exchange.Sell(round(current_price - 0.2, 2), 1) # 开空单
```

在下单交易之前，我们需要先确定两个数据，一个是下单的价格，另一个是当前的持仓状态。下单的价格很简单，直接使用当前的收盘价加减品种的最小变动价位即可。由于我们之前已经使用 get_position 函数封装了持仓量，所以这里直接调用即可。最后就是根据 EMV 与零轴的位置关系开平仓了。

4.10.4 策略回测

回测配置

配置参数

时间	2019-01-01 00:00	2020-01-01 00:00	1 天	模拟级 Tick
选项	日志 8000	收益 8000	图表 3000	1 天
	滑点 0	容错 0.5	延迟 200	柱长 300
费用	Maker 0.025	Taker 0.025	最低 5 元	
数据	绘制行情图表	默认数据源		
精度	定价 3	数量 0	深度 20	需要分笔数据
平台	商品期货	FUTURES	余额 1000000	
	商品期货 / FUTURES x			
分发	云端公用回测服务集群			
	开始回测			

图 4.8 简易波动策略回测配置

时间	平台	类型	价格	数量	信息
2019-12-27 13:48:00	Futures_CTP	买入 开多	3063.4	1	
2019-12-27 09:00:00	Futures_CTP	买入 平空	3021.6	1	
2019-12-24 09:29:57	Futures_CTP	卖出 开空	2992.8	1	
2019-12-24 09:00:00	Futures_CTP	卖出 平多	2992.6	1	
2019-12-23 09:29:57	Futures_CTP	买入 开多	3026.6	1	
2019-12-23 09:00:00	Futures_CTP	买入 平空	3026.4	1	
2019-12-20 09:29:57	Futures_CTP	卖出 开空	3035	1	
2019-12-20 09:00:00	Futures_CTP	卖出 平多	3034.8	1	
2019-12-12 09:29:57	Futures_CTP	买入 开多	2953	1	

图 4.9 简易波动策略回测（1）

资金曲线

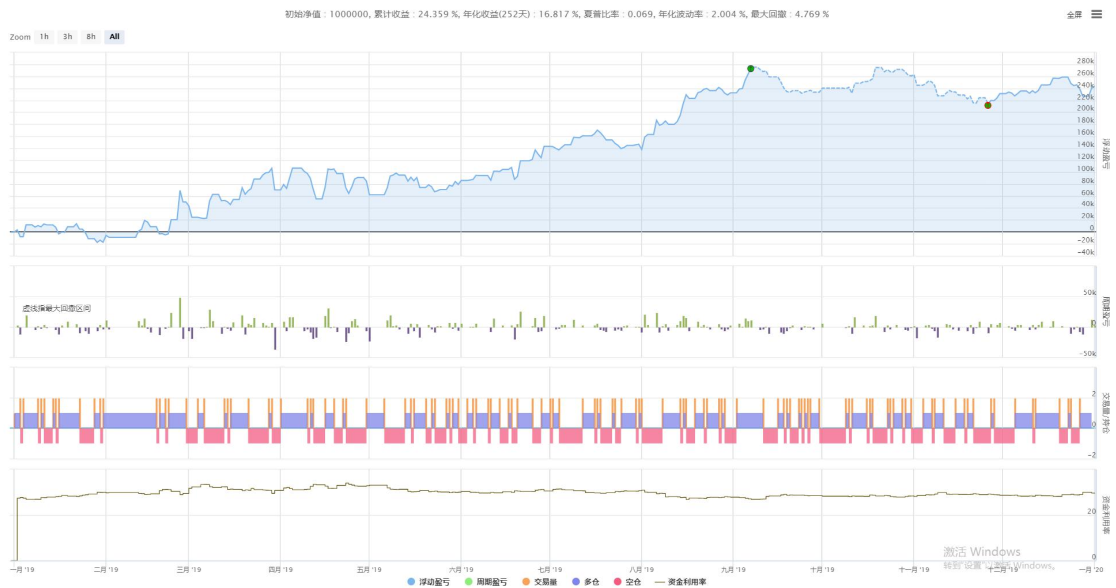

图 4.10　简易波动回测（2）

4.10.5　完整策略

```
# 获取持仓数量
def get_position():
    position = 0                                     # 赋值持仓数量为 0
    position_arr = _C(exchange.GetPosition)          # 获取持仓数组
    if len(position_arr) > 0:                        # 如果持仓数组长度大于 0
        for i in position_arr:                       # 遍历持仓数组
            if i['ContractType'][:2] == 'IH':        # 如果持仓品种等于订阅品种
                if i['Type'] % 2 == 0:               # 如果是多单
                    position = i['Amount']           # 赋值持仓数量为正数
                else:
                    position = -i['Amount']          # 赋值持仓数量为负数
    return position                                  # 返回持仓量

# 策略主函数
def onTick():
    exchange.SetContractType('IH000')                # 订阅期货品种
    bars_arr = exchange.GetRecords()                 # 获取 K 线数组
    if len(bars_arr) < 10:                           # 如果 K 线数量小于 10 根
        return
    bar1 = bars_arr[-2]                              # 获取上一根 K 线数据
    bar2 = bars_arr[-3]                              # 获取前一根 K 线数据
```

```
    mov_mid = (bar1['High'] + bar1['Low'])/2-(bar2['High'] + bar2['Low'])/2
    if bar1['High'] != bar1['Low']:             # 如果被除数不为0
        ratio = (bar1['Volume'] / 10000) / (bar1['High'] - bar1['Low'])
    else:
        ratio = 0
    if ratio > 0:                                # 如果ratio的值大于0
        emv = mov_mid / ratio
    else:
        emv = 0
    current_price = bars_arr[-1]['Close']        # 最新价格
    position = get_position()                    # 获取最新持仓量
    if position > 0:                             # 如果持有多单
        if emv < 0:                              # 如果当前价格小于牙齿
            exchange.SetDirection("closebuy")    # 设置交易方向和类型
            exchange.Sell(round(current_price - 0.2, 2), 1)  # 平多单
    if position < 0:                             # 如果持有空单
        if emv > 0:                              # 如果当前价格大于牙齿
            exchange.SetDirection("closesell")   # 设置交易方向和类型
            exchange.Buy(round(current_price + 0.2, 2), 1)   # 平空单
    if position == 0:                            # 如果无持仓
        if emv > 0:                              # 如果当前价格大于上唇
            exchange.SetDirection("buy")         # 设置交易方向和类型
            exchange.Buy(round(current_price + 0.2, 2), 1)   # 开多单
        if emv < 0:                              # 如果当前价格小于下巴
            exchange.SetDirection("sell")        # 设置交易方向和类型
            exchange.Sell(round(current_price - 0.2, 2), 1)  # 开空单

# 程序入口函数
def main():
    while True:      # 循环
        onTick()     # 执行策略主函数
        Sleep(1000)  # 休眠1秒
```

通过本节课程学习，可以看出EMV与普通交易者的看法相反，但却不无道理。由于EMV引入了成交量数据，因此比其他单纯用价格计算的技术指标，更能有效发现价格背后的东西。每一种策略都有着不同的特点，只有充分了解不同策略之间的优缺点，去其糟粕取其精华才能离成功更进一步。

4.11 动态阶梯突破策略

在前面几节课程中，我们学习了基于指标来构建简单的策略，其中在计算指标时用到了talib库，大大简化了策略编写难度。但有时候我们写策略可能会用到talib库中没有的计算方法，那么今天我们就通过动态阶梯突破策略，来学习下这种策略是如何实现的。

注意：talib 是一个 Python 金融技术指标库，包含了很多常用的技术指标算法。包括：MA、MACD、KDJ 等等。TaLib 官网：http://ta-lib.org/

4.11.1 什么是突破策略

我们知道，期货市场的价格以趋势和震荡交替的方式演变，如果我们只使用一种方法抓住趋势，就能赚到趋势行情的钱。那么，用什么方式来抓住趋势呢？比较简单的一种方法就是用突破策略。

根据一段时间内的历史价格数据，通过设置价格的上下轨，或者支撑位、压力位，当价格超过上轨，我们认为多头行情即将启动，开仓做多；当价格跌破下轨，我们认为空头行情，开仓做空。

市面上有很多不同种类的突破策略，大致可以分为：形态突破（包括：双肩型、头肩型、颈线、趋势线等等）、指标突破（均线、KDJ、ATR 等等）、通道突破（新高和新低、支撑线和阻力线）、量能突破（成交量、能量潮）。其中在量化交易中，比较常用的是：指标突破和通道突破。

4.11.2 突破策略理论

在逻辑学中，有一个"充分不必要条件"的概念，也就是说：如果有 A 不一定有 B，但如果有 B 就必定有 A。那么 B 就是 A 的充分而不必要的条件，即充分不必要条件，A 是 B 的必要不充分条件。所以站在结果的角度讲，价格突破关键点位后未必形成趋势，但趋势上涨或下跌必然会突破其间的关键价格位置。

另外，从突破的原因上讲，市场涨跌取决于买卖双方实力对比。当价位冲破上一时段的最高点时，在上一时段任何价位做空头的人都无一例外被套牢，它当中肯定有一部分要认赔平仓出局，反过来又给升势推波助澜。

反之亦然，当行情跌破上一时段的最低价时，在上一时段任何一点做多头的统统都出现浮动亏损，其中一定有部分交易者要平仓卖出止损，正好对跌势落井下石，加剧了行情继续下跌。

4.11.3 策略逻辑

阶梯策略，这是一个比较土的名字，因为其在图表上的外形类似台阶而得名，最初的灵感来自于阶梯止损。相信有过实盘经验的人应该深有体会：当市场进行横向整理或者摇摆不定的时候，对交叉类系统的打击很大，往往会买在高点，卖在低点。如果行情一直持续，则会出现连续亏损，连续的亏损信号将对交易者造成严重的心理负担和资金回撤压力。如下图：

注意：连续出现亏损信号，不仅限于交叉类系统中，归其原因不是指标的错误，而是对市场节奏的不适应。

图 4.11　均线图

利用通道技术则可以过滤或者减少价格反复缠绕，减少部分虚假信号，对于降低无效交易有巨大帮助。本策略并非传统的通道策略。而是根据前期最高价和最低价，反向建立自适应通道。这里提到的自适应是指回溯日期会根据我们的逻辑进行调整，具体来说本策略是由市场波动的变动率来变化。

与布林带通道不同的是阶梯策略的通道并不是以中轨得来，与之相反的是：先根据市场波动率计算出通道上下轨，然后再根据通道宽度算出中轨。另外，在判定最高点和最低点的 K 线条数取决于我们愿意给交易多少变化空间，我们用来越多的 K 线条数来确定上下轨，我们给予程序化交易的变化空间越大，相应的，在触发止损前盈利回撤的幅度也会越大。使用越近的高点或低点，止损被触发的速度也越快。想让通道窄一点就设定小一点，想让通道宽一点就设定大一点。

图 4.12　动态阶梯

设置通道

- 上轨：如果当根 K 线最低价小于上根 K 线最低价，上轨就等于前 N 根 K 线最高价
- 下轨：如果当根 K 线最高价大于上根 K 线最高价，下轨就等于前 N 根 K 线最低价
- 中轨：上轨和下轨的平均值

入场条件

- 多头入场：如果当前没有持仓，并且价格大于上轨，买入开仓。
- 空头入场：如果当前没有持仓，并且价格小于下轨，卖出开仓。

出场条件

- 多头出场：如果当前持有多单，并且价格小于中轨，卖出平仓。
- 空头出场：如果当前持有空单，兵器价格大于中轨，买入平仓。

为了避免过度拟合，在设计策略的时候，只给定了一个参数。虽然仅有一个参数，但却不失策略在市场中的灵活性。不仅如此，阶梯策略既能适应国内外商品期货，还可以应用于 A 股 ETF ，包括外盘 ETF 和反向杠杆 ETF，以及外汇市场。当然只是适应部分品种，从全品种统计来看，这是一个普适性比较强的策略。

4.11.4 策略编写

根据上面的策略逻辑，我们可以在发明者量化交易平台上实现交易策略。依次打开：fmz.com > 登录 > 控制中心 > 策略库 > 新建策略 > 点击右上角下拉菜单选择 Python 语言，开始编写策略，注意看下面代码中的注释。

第 1 步：编写策略框架

这个在之前的章节已经学习过，一个是 onTick 函数，另一个是 main 函数，其中在 main 函数中无限循环执行 onTick 函数，如下：

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:        # 进入无限循环模式
        onTick()       # 执行策略主函数
        Sleep(1000)    # 休眠 1 秒
```

第 2 步：定义全局变量

首先定义策略中的上轨、下轨，因为我们的策略中上轨、下轨时是根据一定的条件来计算的，也就是说：当前的最高价大于前面 K 线的最高价时，才重新计算下轨。这样做的意义是只参考最新的数据。

注意：当前的最低价小于前面 K 线的最低价时，才重新计算上轨。所以我们必须把上轨和下轨变量定义在 onTick 主函数外面。

```
# 定义全局变量
up_line = 0            # 上轨
```

```
under_line = 0        # 下轨
mp = 0                # 用于控制虚拟持仓
```

另外，我们还需要定义全局变量虚拟持仓 mp，策略运行之初默认是空仓 mp=0，当开多单后把虚拟持仓重置为 mp=1，当开空单后把虚拟持仓重置为，mp=-1，当平多单或空单后把虚拟持仓重置为 mp=0。这样我们在判断构建逻辑获取仓位时，只需要判断 mp 的值就可以了。

第 3 步：计算上轨、下轨、中轨

因为在计算这些数据之前，肯定要先获取历史的 K 线基础数据，这些基础数据的获取方式也很简单，首先订阅期货品种，然后调用发明者量化 API 中的 GetRecords 方法。接着因为在计算上轨和下轨的时候需要用到 talib 库中的 Highest 和 Lowest 方法。这两个方法都要传入周期参数，但如果 K 线数据不够的时候，就不能正常计算其值，所以在这里就要判断 K 线数据的长度，如果 K 线的长度不足以计算其值时，就直接返回跳过。

接着，我们分别获取当前 K 线和上根 K 线的最高价和最低价，通过对比当前 K 线最高价与上根 K 线最高价来定义下轨的值，如果当前的最高价大于前面 K 线的最高价时，就重新计算下轨；同理如果当前的最低价小于前面 K 线的最低价时，就重新计算上轨。最后上轨和下轨的平均值就是中轨。

```
exchange.SetContractType("rb000")  # 订阅期货品种
bars = exchange.GetRecords()        # 获取 K 线数组
if len(bars) < cycle_length + 1:    # 如果 K 线数组的长度太小，所以直接返回
    return
close0 = bars[len(bars) - 1].Close # 获取当根 K 线收盘价
high0 = bars[len(bars) - 1].High    # 获取当根 K 线最高价
high1 = bars[len(bars) - 2].High    # 获取上根 K 线最高价
low0 = bars[len(bars) - 1].Low      # 获取当根 K 线最低价
low1 = bars[len(bars) - 2].Low      # 获取上根 K 线最低价
# 获取前 cycle_length 根 K 线最高价的最高价
highs = TA.Highest(bars, cycle_length, 'High')
# 获取前 cycle_length 根 K 线最低价的最低价
lows = TA.Lowest(bars, cycle_length, 'Low')
global up_line, under_line, mp      # 使用全局变量
if high0 > high1:                   # 如果当根 K 线最高价大于上根 K 线最高价
    under_line = lows    # 把下轨重新赋值为：前 cycle_length 根 K 线最低价的最低价
if low0 < low1:                     # 如果当根 K 线最低价小于上根 K 线最低价
    up_line = highs      # 把上轨重新赋值为：前 cycle_length 根 K 线最高价的最高价
middle_line = (lows + highs) / 2    # 计算中轨的值
```

这里有一个地方需要注意，可能细心的朋友已经发现了，我们在计算上轨和下轨的时候，用到了 talib 库中的 Highest 和 Lowest 函数，因为在发明者量化平台中已经内置了这两个常用的函数，所以我们不需要像前几节那样在策略开头导入 talib 库，并且在使用内置函数的时候，其写法也略有不同，具体可以查看下方的代码。

第 4 步：下单交易

有了上轨、下轨、中轨的值，就可以配合当前的最新价格开平仓交易了，我们可以回过头再看下之前设计的交易逻辑：如果当前没有持仓，并且价格大于上轨 * 1.05，买入开

仓。

如果当前没有持仓，并且价格小于下轨 * 0.95，卖出开仓。如果当前持有多单，并且价格小于中轨，卖出平仓。如果当前持有空单，兵器价格大于中轨，买入平仓。

```
if mp == 0 and close0 > up_line:         # 如果当前空仓，并且最新价大于上轨
    exchange.SetDirection("buy")         # 设置交易方向和类型
    exchange.Buy(close0, 1)              # 开多单
    mp = 1                               # 设置虚拟持仓的值，即有多单
if mp == 0 and close0 < under_line:      # 如果当前空仓，并且最新价小于下轨
    exchange.SetDirection("sell")        # 设置交易方向和类型
    exchange.Sell(close0 - 1, 1)         # 开空单
    mp = -1                              # 设置虚拟持仓的值，即有空单
if mp > 0 and close0 < middle_line:      # 如果当前持有多单，并且最新价小于中轨
    exchange.SetDirection("closebuy")    # 设置交易方向和类型
    exchange.Sell(close0 - 1, 1)         # 平多单
    mp = 0                               # 设置虚拟持仓的值，即空仓
if mp < 0 and close0 > middle_line:      # 如果当前持有空单，并且最新价大于中轨
    exchange.SetDirection("closesell")   # 设置交易方向和类型
    exchange.Buy(close0, 1)              # 平空单
    mp = 0                               # 设置虚拟持仓的值，即空仓
```

下单交易使用 if 语句，如果条件为真，就先设置交易方向和类型，即：开多、开空、平多、平空。然后调用发明者量化平台中的 Buy 或 Sell 下单函数。

注意：最后下单之后重置虚拟持仓的状态。

4.11.5 完整策略

```
# 外部参数
cycle_length = 100

# 定义全局变量
up_line = 0                                  # 上轨
under_line = 0                               # 下轨
mp = 0                                       # 用于控制虚拟持仓

def onTick():
    exchange.SetContractType("rb000")        # 订阅期货品种
    bars = exchange.GetRecords()             # 获取 K 线数组
    if len(bars) < cycle_length + 1:         # 如果 K 线数组的长度太小就直接返回
        return
    close0 = bars[len(bars) - 1].Close       # 获取当根 K 线收盘价
    high0 = bars[len(bars) - 1].High         # 获取当根 K 线最高价
    high1 = bars[len(bars) - 2].High         # 获取上根 K 线最高价
    low0 = bars[len(bars) - 1].Low           # 获取当根 K 线最低价
    low1 = bars[len(bars) - 2].Low           # 获取上根 K 线最低价
```

```
    # 获取前 cycle_length 根 K 线最高价的最高价
    highs = TA.Highest(bars, cycle_length, 'High')
    # 获取前 cycle_length 根 K 线最低价的最低价
    lows = TA.Lowest(bars, cycle_length, 'Low')
    global up_line, under_line, mp          # 使用全局变量
    if high0 > high1:    # 如果当根 K 线最高价大于上根 K 线最高价
        under_line=lows  # 把下轨重新赋值为：前 cycle_length 根 K 线最低价的最低价
    if low0 < low1:      # 如果当根 K 线最低价小于上根 K 线最低价
        up_line = highs  # 把上轨重新赋值为：前 cycle_length 根 K 线最高价的最高价
    middle_line = (lows + highs) / 2        # 计算中轨的值

    if mp == 0 and close0 > up_line:        # 如果当前空仓，并且最新价大于上轨
        exchange.SetDirection("buy")        # 设置交易方向和类型
        exchange.Buy(close0, 1)             # 开多单
        mp = 1                              # 设置虚拟持仓的值，即有多单

    if mp == 0 and close0 < under_line:     # 如果当前空仓，并且最新价小于下轨
        exchange.SetDirection("sell")       # 设置交易方向和类型
        exchange.Sell(close0 - 1, 1)        # 开空单
        mp = -1                             # 设置虚拟持仓的值，即有空单

    if mp > 0 and close0 < middle_line:     # 如果当前持有多单且最新价小于中轨
        exchange.SetDirection("closebuy")   # 设置交易方向和类型
        exchange.Sell(close0 - 1, 1)        # 平多单
        mp = 0                              # 设置虚拟持仓的值，即空仓

    if mp < 0 and close0 > middle_line:     # 如果当前持有空单且最新价大于中轨
        exchange.SetDirection("closesell")  # 设置交易方向和类型
        exchange.Buy(close0, 1)             # 平空单
        mp = 0                              # 设置虚拟持仓的值，即空仓

def main():
    while True:                             # 进入无限循环模式
        onTick()                            # 执行策略主函数
        Sleep(1000)                         # 休眠 1 秒
```

在量化投资领域，突破策略是技术分析中很重要的一部份，它的观念在于市场价格穿透了之前的价格压力或支撑，继而形成一股新的趋势，我们的目标就是在突破发生时能够确认并建立部位以获取趋势的利润。通道策略有他的弹性，可顺可逆。把通道缩窄一点，就可以做顺势；或是把通道拉宽一点，就可以做逆势，或是再搭配其他指标写成其他的交易策略等。

可以发现其实通道无所不在，只是通道的取法各有不同罢了。换个角度来看，我们都只是在寻找两条线，支撑跟压力线来决定进场作多或作空，通道的上限与下限即是这种概念。总之，对量化交易而言，不管是什么策略，可以赚钱就是好策略。

4.12 Dual Thrust 日内交易策略

Dual Thrust 直译为“双重推力”，是上个世纪 80 年代由 Michael Chalek 开发的一个交易策略，曾经在期货市场风靡一时。由于策略本身思路简单，参数很少，因此可以适应于很多金融市场，正是因为简单易用和普适性高的特点，得到了广大交易者的认可流传至今。

4.12.1 Dual Thrust 简介

Dual Thrust 策略属于开盘区间突破策略，它以当天开盘价加减一定的范围来确定一个上下轨道，当价格突破上轨时做多，价格突破下轨时做空。不过与其他突破策略相比有两点不同：第一个是 Dual Thrust 策略在设置范围的时候，引入的是前 N 个交易日的开高低收这四个价格，这使得在一定时期内范围相对稳定，对于趋势跟踪策略来说是比较合理的。

第二个是 Dual Thrust 策略在多头和空头的触发条件上，考虑了非对称性，通过外部参数 Ks 和 Kx，可以针对多头和空头选择不同的周期，这一点比较符合期货市场涨缓跌急的特点。当 Ks 小于 Kx 时，多头相对容易被触发，当 Ks 大于 Kx 时，空头相对容易被触发。这样的好处是可以根据自己的交易经验，动态地调整 Ks 和 Kx 的值。

注意：可以根据历史数据测试的最优参数来使用策略。

4.12.2 Dual Thrust 上下轨

在 Dual Thrust 策略中，首先需要定义前 N 根 K 线的震荡区间，然后震荡区间乘以多头和空头系数计算出范围，接着以开盘价加减这个范围，形成上轨和下轨，最后根据价格与上下轨的相互位置关系开开平仓。

第 1 步：计算震荡区间。计算震荡区间首先需要获取四个价格，它们分别是：前 N 根 K 线中最高价(hh)、最高收盘价(hc)、最低价(ll)、最低收盘价(lc)。然后获取 hh 与 lc 的差和 hc 与 ll 的差，最后获取这两个差的最大值。公式为：

- Range = Max(hh-lc, hc-ll)

第 2 步：计算范围。在计算范围的时候，需要用到两个外部参数，分别是多头系数 Ks 和空头系数 Kx，它们的值可以根据交易者的经验自己设置。那么多头的范围就是 Rang 乘以 Ks；空头的范围是 Rang 乘以 Kx。公式为：

- long_range = Range * Ks
- short_range = Range * Kx

第 3 步：计算上轨下轨。有了多头范围和空头范围，就可以根据开盘价来计算上轨和下轨的值了，其中上轨的值是开盘价加上多头范围，下轨的值是开盘价减去空头范围。公式为：

- up_line = open + long_rang
- down_line = open - short_range

4.12.3 策略逻辑

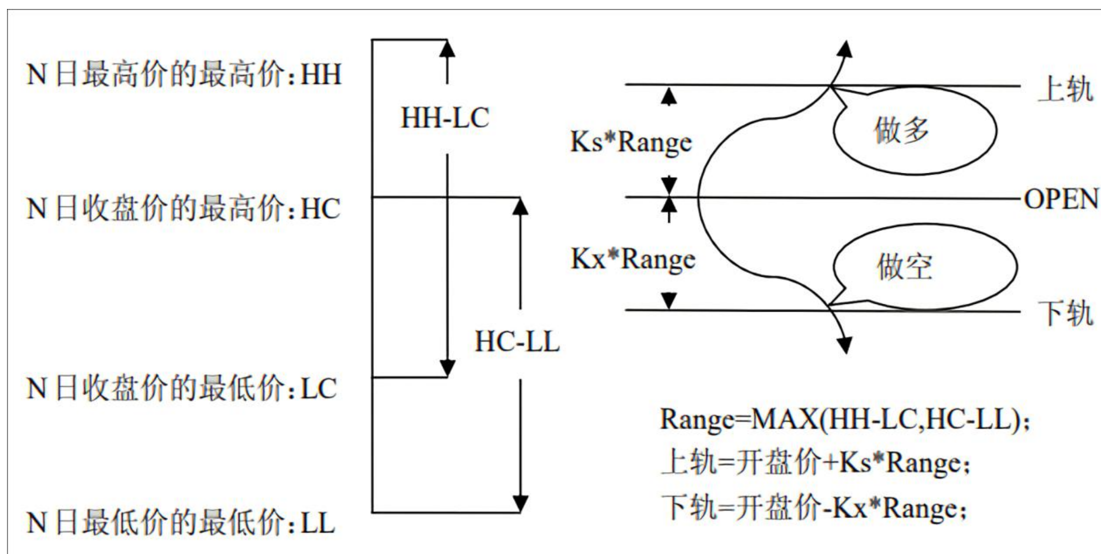

图 4.13 Dual Thrust 策略逻辑示意图

- 做多：价格向上突破上轨
- 做空：价格向下突破下轨

与其他突破策略一样，Dual Thrust 策略也是根据价格与上下轨的相对位置关系来开平仓，当价格向上突破上轨时开多单；当价格向下突破下轨时开空单。另外，Dual Thrust 策略没有止损止盈机制，也没有主动平仓机制。也就是说当持有多单时，如果价格向下突破下轨时直接反空为多；当持有空单时，如果价格向上突破上轨时直接反多为空。

4.12.4 策略编写

第 1 步：编写策略架构

还是我们熟悉的策略框架，包含一个 main 程序入口函数和一个 onTick 策略主函数，如下：

```
# 定义全局变量
mp = 0                  # 用于控制虚拟持仓
last_bar_time = 0       # 用于判断 K 线时间
up_line = 0             # 上轨
down_line = 0           # 下轨

# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:         # 进入无限循环模式
```

```
        onTick()        # 执行策略主函数
        Sleep(1000)     # 休眠 1 秒
```

第 2 步：定义全局变量

之所以定义全局变量是因为，在程序重复执行 onTick 函数中，如果变量定义在 onTick 函数中，那么这个变量的值会随着 onTick 的执行而改变。但有时候我们需要当达到某个条件的时候才改变这个变量，所以就需要我们把变量写到 onTick 函数的外面。

第 3 步：计算上下轨

仔细看下面代码中的注释，首先一次性引入所有的全局变量，然后订阅期货品种并获取 K 线数组，接着判断一下 K 线数组的状态是否符合我们的条件，如果没问题就从 K 线数组中获取最新的 K 线数据和最新的收盘价。

有了以上基础数据，就可以计算上下轨的值了。首先是获取四个价格：最高价、最高的收盘价、最低价、最低的收盘价，然后就可以计算范围，最后根据范围计算出上轨和下轨。大家可以根据以上的计算流程，熟悉下面的代码。

```
global mp, last_bar_time, up_line, down_line        # 引入全局变量
exchange.SetContractType(FuturesCode)               # 订阅期货品种
bar_arr = exchange.GetRecords()                     # 获取 K 线数组
# 如果没有获取到 K 线数据或者 K 线数据太短就返回
if not bar_arr or len(bar_arr) < Cycle:
   return
last_bar = bar_arr[len(bar_arr) - 1]                # 最新的 K 线
last_bar_close = last_bar['Close']                  # 最新 K 线的收盘价
if last_bar_time != last_bar['Time']:               # 如果产生了新的 K 线
   hh = TA.Highest(bar_arr, Cycle, 'High')          # 最高价
   hc = TA.Highest(bar_arr, Cycle, 'Close')         # 最高的收盘价
   ll = TA.Lowest(bar_arr, Cycle, 'Low')            # 最低价
   lc = TA.Lowest(bar_arr, Cycle, 'Close')          # 最低的收盘价
   Range = max(hh - lc, hc - ll)                    # 计算范围
   up_line = _N(last_bar['Open'] + Ks * Range)      # 计算上轨
   down_line = _N(last_bar['Open'] - Kx * Range)    # 计算下轨
   last_bar_time = last_bar['Time']                 # 更新最后时间戳
```

第 4 步：下单交易

下单交易很简单，使用 if 语句判断当前的持仓状态和价格与上下轨的相互位置关系来开平仓。同样的在下单交易之前也需要设置交易方向和类型，即：开多、开空、平多、平空。

注意：最后下单之后重置虚拟持仓的状态。

```
if mp == 0 and last_bar_close >= up_line:
   exchange.SetDirection("buy")                     # 设置交易方向和类型
   exchange.Buy(last_bar_close, 1)                  # 开多单
   mp = 1                                           # 设置虚拟持仓的值即有多单
if mp == 0 and last_bar_close <= down_line:
   exchange.SetDirection("sell")                    # 设置交易方向和类型
   exchange.Sell(last_bar_close - 1, 1)             # 开空单
   mp = -1                                          # 设置虚拟持仓的值即有空单
if mp == 1 and last_bar_close <= down_line:
```

```
    exchange.SetDirection("closebuy")          # 设置交易方向和类型
    exchange.Sell(last_bar_close - 1, 1)       # 平多单
    mp = 0                                     # 设置虚拟持仓的值，即空仓
if mp == -1 and last_bar_close >= up_line:
    exchange.SetDirection("closesell")         # 设置交易方向和类型
    exchange.Buy(last_bar_close, 1)            # 平空单
    mp = 0                                     # 设置虚拟持仓的值，即空仓
```

4.12.5 策略回测

图 4.14　Dual Thrust 策略回测

4.12.6 完整策略

```
# 定义全局变量
mp = 0                                              # 用于控制虚拟持仓
last_bar_time = 0                                   # 用于判断 K 线时间
up_line = 0                                         # 上轨
down_line = 0                                       # 下轨

'''
# 外部参数
Cycle = 5                                           # 周期
Ks = 1                                              # 多头系数
Kx = 2                                              # 空头系数
'''

# 策略主函数
def onTick():
    global mp, last_bar_time, up_line, down_line    # 引入全局变量
    exchange.SetContractType(FuturesCode)           # 订阅期货品种
    bar_arr = exchange.GetRecords()                 # 获取 K 线数组
    # 如果没有获取到 K 线数据或者 K 线数据太短就返回
    if not bar_arr or len(bar_arr) < Cycle:
        return
    last_bar = bar_arr[len(bar_arr) - 1]            # 最新的 K 线
    last_bar_close = last_bar['Close']              # 最新 K 线的收盘价
    if last_bar_time != last_bar['Time']:           # 如果产生了新的 K 线
        hh = TA.Highest(bar_arr, Cycle, 'High')         # 最高价
        hc = TA.Highest(bar_arr, Cycle, 'Close')        # 最高的收盘价
        ll = TA.Lowest(bar_arr, Cycle, 'Low')           # 最低价
        lc = TA.Lowest(bar_arr, Cycle, 'Close')         # 最低的收盘价
        Range = max(hh - lc, hc - ll)                   # 计算范围
        up_line = _N(last_bar['Open'] + Ks * Range)     # 计算上轨
        down_line = _N(last_bar['Open'] - Kx * Range)   # 计算下轨
        last_bar_time = last_bar['Time']                # 更新最后时间戳
    if mp == 0 and last_bar_close >= up_line:
        exchange.SetDirection("buy")                    # 设置交易方向和类型
        exchange.Buy(last_bar_close, 1)                 # 开多单
        mp = 1                                          # 设置虚拟持仓有多单
    if mp == 0 and last_bar_close <= down_line:
        exchange.SetDirection("sell")                   # 设置交易方向和类型
        exchange.Sell(last_bar_close - 1, 1)            # 开空单
        mp = -1                                         # 设置虚拟持仓有空单
    if mp == 1 and last_bar_close <= down_line:
        exchange.SetDirection("closebuy")               # 设置交易方向和类型
```

```
        exchange.Sell(last_bar_close - 1, 1)            # 平多单
        mp = 0                                          # 设置虚拟持仓空仓
    if mp == -1 and last_bar_close >= up_line:
        exchange.SetDirection("closesell")              # 设置交易方向和类型
        exchange.Buy(last_bar_close, 1)                 # 平空单
        mp = 0                                          # 设置虚拟持仓空仓

# 程序入口
def main():
    while True:                                         # 进入循环模式
        onTick()                                        # 执行策略主函数
        Sleep(1000)                                     # 休眠 1 秒
```

注意：尽可能折中选择 Ks 和 Kx 外部参数的值。如果值太小，可能会及时跟踪到趋势，但会有很多虚假的突破信号；如果值太大，可能会错过趋势开始的部分，或者刚入场不久，趋势就结束了。

4.13　经典恒温器策略

趋势行情不会永远持续下去，事实上市场大部分时间都处于震荡行情，所以才会有人希望能得到一种交易策略，既可以用在趋势行情，也可以用在震荡行情。本节我们就用发明者量化交易平台，构建一个趋势和震荡行情通用的经典恒温器策略。

4.13.1　策略简介

提到恒温器可能会有人想到汽车发动机与水箱之间的恒温器。当发动机温度低时，恒温器是关闭状态，此时发动机和水箱的水是不相通的，直到发动机温度升高，达到最佳机油润滑效果；当发动机温度升高到一定阈值时，节温器是开启状态，此时发动机和水箱的水形成循环，并流经风扇开启降温模式，直到达到发动机最佳工作温度。

那么恒温器策略也类似这个原理，并且延用了这个名字。它通过波动指数作为阈值，将市场分为趋势行情和震荡行情，自动对两种不同的行情使用对应的交易逻辑，有效弥补了趋势策略在震荡行情中的不适应。

4.13.2　市场波动指数

如何把市场划分为趋势行情和震荡行情，也就成了这个策略的关键，恒温器策略引入了市场波动指数（Choppy Market Index），简称 CMI。它是一个用来判断市场走势类型的技术指标。通过计算当前收盘价与 N 周期前收盘价的差值与这段时间内价格波动的范围的比值，来判断目前的价格走势是趋势还是震荡。

CMI 的计算公式为：

$$CMI = \frac{abs(Close - ref(close, (n-1))) * 100}{HHV(high, n) - LLV(low, n)}$$

其中，abs 是绝对值，n 是周期数。

4.13.3　策略逻辑

一般来说 CMI 的值在 0~100 区间，值越大，趋势越强。当 CMI 的值小于 20 时，策略认为市场处于震荡模式；当 CMI 的值大于等于 20 时，策略认为市场处于趋势模式。

整个策略逻辑，可以简化的写成下面这样：

- 如果 CMI < 20，执行震荡策略；
- 如果 CMI ≥ 20，执行趋势策略；

策略架构就是这么简单，剩下的就是把震荡策略的内容和趋势策略的内容，填充到这个框架里面。

4.13.4　策略编写

依次打开：fmz.com 网站 > 登录 > 控制中心 > 策略库 > 新建策略 > 点击右上角下拉菜单选择 Python 语言，开始编写策略，注意看下面代码中的注释。

第 1 步：编写策略框架

这个在之前的章节已经学习过，一个是 onTick 函数，另一个是 main 函数，其中在 main 函数中无限循环执行 onTick 函数，如下：

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:       # 进入无限循环模式
        onTick()      # 执行策略主函数
        Sleep(1000)   # 休眠 1 秒
```

第 2 步：定义虚拟持仓变量

```
mp = 0                # 定义一个全局变量，用于控制虚拟持仓
```

虚拟持仓的作用主要是用来控制策略仓位，当策略运行之初默认是空仓 mp=0，当开多单后把虚拟持仓重置为 mp=1，当开空单后把虚拟持仓重置为，mp=-1，当平多单或空单后把虚拟持仓重置为 mp=0。

注意：在判断构建逻辑获取仓位时，只需要判断 mp 的值就可以了。虚拟持仓的特点时编写简单，快速迭代策略更新，一般用于回测环境中，假设每一笔订单都完全成交，但在实际交易中常用的还是真实持仓。

第 3 步：获取基础数据

```
exchange.SetContractType("rb000")   # 订阅期货品种
```

```
bar_arr = exchange.GetRecords()     # 获取 K 线数组
if len(bar_arr) < 100:              # 如果 K 线少于 100 根
    return                          # 直接返回
close0 = bar_arr[-1]['Close']       # 获取最新价格（卖价），用于开平仓
bar_arr.pop()  # 删除 K 线数组最后一个元素，策略采用开平仓条件成立，下根 K 线交易模式
```

首先使用发明者量化 API 中的 SetContractType 方法订阅期货品种。接着使用 GetRecords 方法获取 K 线数组，因为有时候 K 线数量太少，导致无法计算一些数据，所以我们判断如果 K 线少于 100 根，就直接返回等待下一次新数据。

然后我们从 K 线数组中获取最新的卖一价，这个主要用于使用开平仓函数时传入价格参数。最后因为我们的策略采用当前 K 线开平仓条件成立，在下根 K 线交易的模式，所以需要删除 K 线数组最后一个元素。

因为策略回测是根据历史价格来计算各种收益绩效，历史价格是一种固定的数据，不可能完全跟实盘一样，所以这样做有 2 个好处：第 1 个可以使回测绩效更接近于实盘；第 2 个是避免未来函数和偷价这些常见的策略逻辑错误。

计算市场波动指数 CMI

```
# 计算 CMI 指标用以区分震荡市与趋势市
close1 = bar_arr[-1]['Close']                           # 最新收盘价
close30 = bar_arr[-30]['Close']                         # 前 30 根 K 线的收盘价
hh30 = TA.Highest(bar_arr, 30, 'High')                  # 最近 30 根 K 线最高价
ll30 = TA.Lowest(bar_arr, 30, 'Low')                    # 最近 30 根 K 线最低价
cmi = abs((close1 - close30) / (hh30 - ll30)) * 100    # 计算市场波动指数
```

根据 CMI 的计算公式，我们需要 4 个数据，分别是：最新收盘价、前 30 根 K 线的收盘价、最近 30 根 K 线的最高价、最近 30 根 K 线的最低价。前两个很简单，可以直接从 K 线数组中获取。

最后两个则需要调用发明者量化内置的 talib 指标库 TA.Highest 和 TA.Lowest，这两个指标函数需要传入三个参数，分别是：K 线数据、周期、属性。最后当前收盘价与前 30 根 K 线的收盘价的差值与这段时间内价格波动的范围的比值就是市场波动指数 CMI。

定义宜卖市和宜买市

```
# 震荡市中收盘价大于关键价格为宜卖市，否则为宜买市
high1 = bar_arr[-1]['High']         # 最新最高价
low1 = bar_arr[-1]['Low']           # 最新最低价
kod = (close1 + high1 + low1) / 3   # 计算关键价格
if close1 > kod:
    be = 1
    se = 0
else:
    be = 0
    se = 1
```

在震荡市场中，通常存在一种现象：如果今天价格上涨的话，那么明天的价格下跌的概率更大。而今天价格如果下跌的话，那么明天的价格上涨的概率更大，而这也正是震荡市场的特性。所以这里首先定义一个关键价格(最高价+最低价+收盘价的平均值)。这些数据都可以在 K 线数据中直接获取。如果当前价格大于关键价格，那么明天应该震荡看空。相

反的，如果当前价格小于关键价格，那么明天应该震荡看多。

计算震荡行情的进出场价格

```
# 计算 10 根 K 线 ATR 指标
atr10 = TA.ATR(bar_arr, 10)[-1]

# 定义最高价与最低价 3 日均线
high2 = bar_arr[-2]['High']                # 上根 K 线最高价
high3 = bar_arr[-3]['High']                # 前根 K 线最高价
low2 = bar_arr[-2]['Low']                  # 上根 K 线最低价
low3 = bar_arr[-3]['Low']                  # 前根 K 线最低价
avg3high = (high1 + high2 + high3) / 3     # 最近 3 根 K 线最高价的均值
avg3low = (low1 + low2 + low3) / 3         # 最近 3 根 K 线最低价的均值

# 计算震荡行情的进场价格
open1 = bar_arr[-1]['Open']                # 最新开盘价
if close1 > kod:                           # 如果收盘价大于关键价格
   lep = open1 + atr10 * 3
   sep = open1 - atr10 * 2
else:
   lep = open1 + atr10 * 2
   sep = open1 - atr10 * 3
lep1 = max(lep, avg3high)                  # 计算震荡市多头进场价格
sep1 = min(sep, avg3low)                   # 计算震荡市空头进场价格
```

首先计算 10 根 K 线 ATR 指标，同样也是直接调用发明者量化的内置 talib 库中的 TA.ATR 即可。为了防止假突破，导致策略来回止损，因此加入了一个最高价与最低价 3 日均线滤网来避免这种情形，分别从 K 线数组中获取最近 3 根 K 线的值求其平均就可以了。

有了以上计算步骤，最后就可以计算震荡行情中的进出场价格了，其原理是以开盘价为中心，上下加减最近 10 根 K 线的真实波动幅度，形成一个开多和开空的价格通道。为了使策略更加符合市场走势，在做多和做空时分别设置了不同的空间。

注意：在震荡行情中看多，只代表价格上涨的概率更大一些，并不是指价格一定就会上涨。所以把做多的阈值设置的比较低一点，把做空的阈值设置的比较高一点。同理在震荡行情中看空，只代表价格下跌的概率更大一些，并不是指价格一定就会下跌。所以把做空的阈值设置的比较低一点，把做多的阈值设置的比较高一点。

计算趋势行情的进场价格

```
# 计算趋势行情的进场价格
boll = TA.BOLL(bar_arr, 50, 2)      # 调用 BOLL 指标函数
up_line = boll[0][-1]               # 获取上轨
mid_line = boll[1][-1]              # 获取中轨
down_line = boll[2][-1]             # 获取下柜
```

在处理趋势行情的进出场价格上，沿用了布林带策略，当价格向上突破布林带上轨时多头开仓，当价格向下突破布林带下轨时空头开仓，平仓方式则是以当前价格与布林中轨的位置关系来判断。

4.13.5　策略回测

测试环境

- ❑ 交易品种：螺纹钢指数
- ❑ 时间：2016 年 01 月 01 日~2019 年 01 月 01 日
- ❑ 周期：一小时
- ❑ 滑点：开平仓各 2 跳
- ❑ 手续费：交易所 2 倍

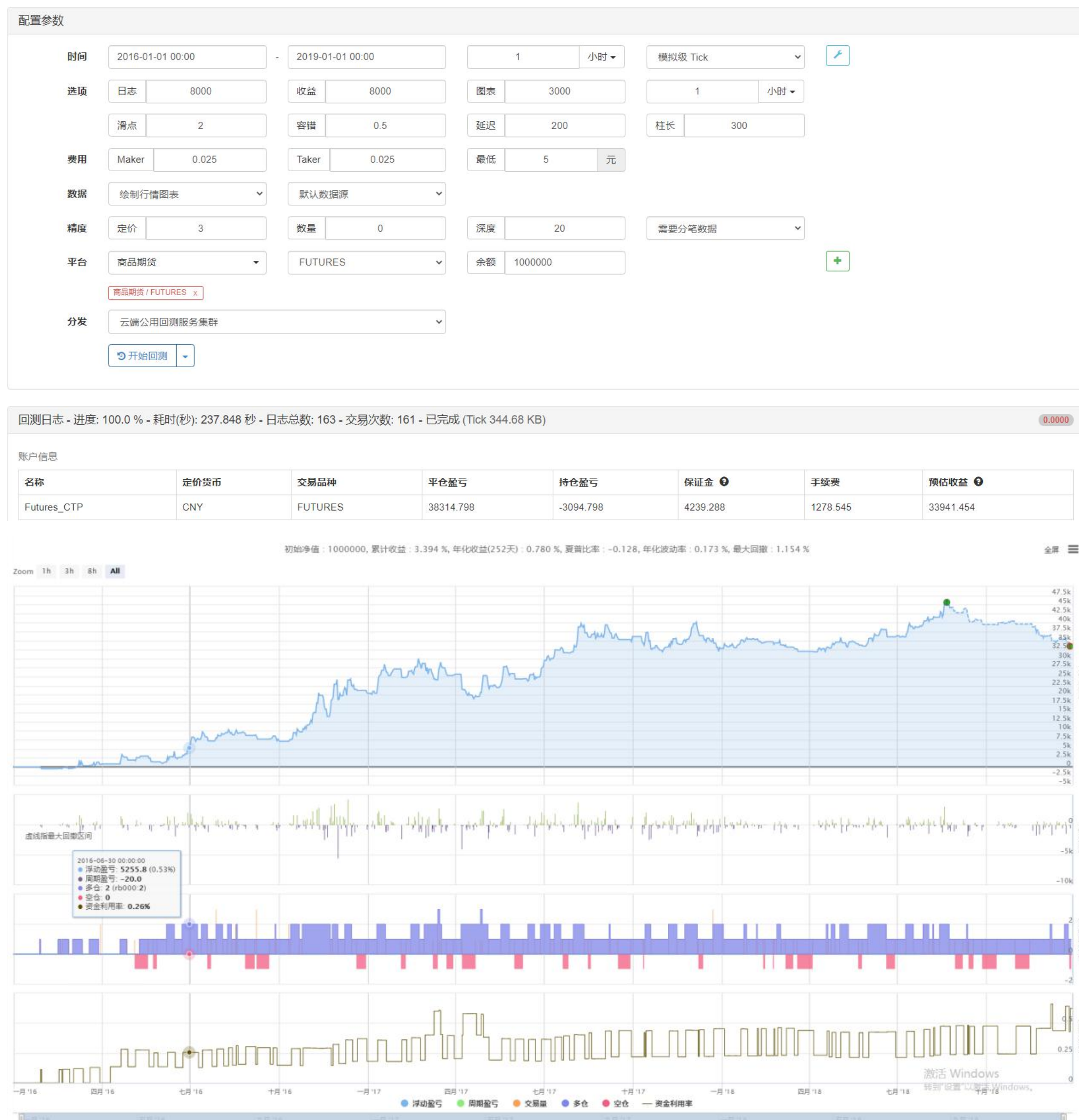

图 4.15　恒温器策略回测

为了将回测结果尽量接近实盘交易，这里把手续费设置为交易所的 2 倍，开仓和平仓

各加 2 跳的滑点，回测的数据品种为螺纹钢指数，交易品种为螺纹钢主力连续。固定 1 手开仓。以下是在 1 小时级别的初步回测绩效报告。

4.13.6 完整策略代码

```
mp = 0                                          # 定义一个全局变量，用于控制虚拟持仓
# 策略主函数
def onTick():
    exchange.SetContractType("rb000")           # 订阅期货品种
    bar_arr = exchange.GetRecords()             # 获取 K 线数组
    if len(bar_arr) < 100:                      # 如果 K 线少于 100 根
        return                                  # 直接返回
    close0 = bar_arr[-1]['Close']               # 获取最新价格（卖价），用于开平仓
    bar_arr.pop()                               # 删除 K 线数组最后一个元素
    # 计算 CMI 指标用以区分震荡市与趋势市
    close1 = bar_arr[-1]['Close']               # 最新收盘价
    close30 = bar_arr[-30]['Close']             # 前 30 根 K 线的收盘价
    hh30 = TA.Highest(bar_arr, 30, 'High')      # 最近 30 根 K 线的最高价
    ll30 = TA.Lowest(bar_arr, 30, 'Low')        # 最近 30 根 K 线的最低价
    cmi = abs((close1 - close30) / (hh30 - ll30)) * 100# 计算市场波动指数
    # 震荡市中收盘价大于关键价格为宜卖市，否则为宜买市
    high1 = bar_arr[-1]['High']                 # 最新最高价
    low1 = bar_arr[-1]['Low']                   # 最新最低价
    kod = (close1 + high1 + low1) / 3           # 计算关键价格
    if close1 > kod:
        be = 1
        se = 0
    else:
        be = 0
        se = 1
    # 计算 10 根 K 线 ATR 指标
    atr10 = TA.ATR(bar_arr, 10)[-1]
    # 定义最高价与最低价 3 日均线
    high2 = bar_arr[-2]['High']                     # 上根 K 线最高价
    high3 = bar_arr[-3]['High']                     # 前根 K 线最高价
    low2 = bar_arr[-2]['Low']                       # 上根 K 线最低价
    low3 = bar_arr[-3]['Low']                       # 前根 K 线最低价
    avg3high = (high1 + high2 + high3) / 3          # 最近 3 根 K 线最高价的均值
    avg3low = (low1 + low2 + low3) / 3              # 最近 3 根 K 线最低价的均值
    # 计算震荡行情的进场价格
    open1 = bar_arr[-1]['Open']                     # 最新开盘价
    if close1 > kod:                                # 如果收盘价大于关键价格
        lep = open1 + atr10 * 3
        sep = open1 - atr10 * 2
    else:
```

```
        lep = open1 + atr10 * 2
        sep = open1 - atr10 * 3
    lep1 = max(lep, avg3high)                        # 计算震荡市多头进场价格
    sep1 = min(sep, avg3low)                         # 计算震荡市空头进场价格
    # 计算趋势行情的进场价格
    boll = TA.BOLL(bar_arr, 50, 2)
    up_line = boll[0][-1]
    mid_line = boll[1][-1]
    down_line = boll[2][-1]
    global mp                                        # 引入全局变量
    if cmi < 20:                                     # 如果是震荡行情
        if mp == 0 and close1 >= lep1 and se:
            exchange.SetDirection("buy")             # 设置交易方向和类型
            exchange.Buy(close0, 1)                  # 开多单
            mp = 1                                   # 设置虚拟持仓的值，即有多单
        if mp == 0 and close1 <= sep1 and be:
            exchange.SetDirection("sell")            # 设置交易方向和类型
            exchange.Sell(close0 - 1, 1)             # 开空单
            mp = -1                                  # 设置虚拟持仓的值，即有空单
        if mp == 1 and (close1 >= avg3high or be):
            exchange.SetDirection("closebuy")        # 设置交易方向和类型
            exchange.Sell(close0 - 1, 1)             # 平多单
            mp = 0                                   # 设置虚拟持仓的值，即空仓
        if mp == -1 and (close1 <= avg3low or se):
            exchange.SetDirection("closesell")       # 设置交易方向和类型
            exchange.Buy(close0, 1)                  # 平空单
            mp = 0                                   # 设置虚拟持仓的值，即空仓
    else:                                            # 如果是趋势行情
        if mp == 0 and close1 >= up_line:
            exchange.SetDirection("buy")             # 设置交易方向和类型
            exchange.Buy(close0, 1)                  # 开多单
            mp = 1                                   # 设置虚拟持仓的值，即有多单
        if mp == 0 and close1 <= down_line:
            exchange.SetDirection("sell")            # 设置交易方向和类型
            exchange.Sell(close0 - 1, 1)             # 开空单
            mp = -1                                  # 设置虚拟持仓的值，即有空单
        if mp == 1 and close1 <= mid_line:
            exchange.SetDirection("closebuy")        # 设置交易方向和类型
            exchange.Sell(close0 - 1, 1)             # 平多单
            mp = 0                                   # 设置虚拟持仓的值，即空仓
        if mp == -1 and close1 >= mid_line:
            exchange.SetDirection("closesell")       # 设置交易方向和类型
            exchange.Buy(close0, 1)                  # 平空单
            mp = 0                                   # 设置虚拟持仓的值，即空仓

# 程序入口
```

```
def main():
    while True:                                    # 进入无限循环模式
        onTick()                                   # 执行策略主函数
        Sleep(1000)                                # 休眠 1 秒
```

从资金曲线和数据来看，该策略表现良好，在螺纹钢品种回测中，除了 2017 年下半年有较大回撤外，整体资金曲线是稳步向上的。恒温器策略的自动调节交易方式，为大家应对震荡行情提供了一定的思路。你也可以根据自己的理解适当修改，做进一步的深入研究。

4.14 R-breaker 策略

R-breaker 是一个交易频率比较高的日内交易策略，最初在 1993 年公开发布，并且连续 15 年跻身《Future Trust》杂志成为年度最赚钱的十大策略之一，至今仍然活跃在国内外交易市场。该策略的特点是结合了顺势和逆势两种交易方法，因此交易机会相对较多，比较适合 1 分钟至 15 分钟周期的 K 线数据。

4.14.1 策略原理

R-breaker 是一个纯粹的支撑力和阻力策略，它根据前一个交易日的收盘价、最高价、最低价，通过一定的算法，计算出三个支撑线和三个阻力线，分别是：突破买入价、观察卖出价、反转卖出价、反转买入价、观察买入价、突破卖出价。

支撑线和阻力线是常用的技术分析工具，支撑线是价格下方的某个区域，市场在这个区域消化空头价格止跌，多头力量开始在此处聚集价格反弹；同理阻力线是价格上方的某个区域，市场在这个区域消化多头，空头力量开始在此处聚集。

并且支撑线和阻力线经常相互转换，当价格向下跌破支撑后，原有的支撑可能转变为阻力；当价格向上突破阻力后，原有的阻力可能转变为支撑。所以 R-breaker 以此来形成当日的盘中交易触发条件，就好比作战地图一样为交易者指明了方向。

注意：在技术分析中支撑线和阻力线经常互换。

4.14.2 计算方式

简单的说，R-Breaker 策略就是一个支撑位和阻力位策略，它根据昨日的最高价、最低价和收盘价，计算出七个价格：一个中心价(pivot)、三个支撑位(s1、s2、s3)、三个阻力位(r1、r2、r3)。然后根据当前价格与这些支撑位和阻力位的位置关系，以形成买卖的触发条件，并且通过一定的算法调整，调节这七个价格之间的距离，进一步改变交易的触发值。

- 突破买入价(阻力位 r3) = 昨日最高价 + 2 *（中心价 - 昨日最低价）2
- 观察卖出价(阻力位 r2) = 中心价 +（昨日最高价-昨日最低价）
- 反转卖出价(阻力位 r1) = 2 * 中心价 - 昨日最低价
- 中心价(pivot) =（昨日最高价 + 昨日收盘价 + 昨日最低价）/ 3

- ❑ 反转买入价(支撑位 s1) = 2 * 中心价 - 昨日最高价
- ❑ 观察买入价(支撑位 s2) = 中心价 -（昨日最高价 - 昨日最低价）
- ❑ 突破卖出价(支撑位 s3) = 昨日最低价 - 2 *（昨日最高价 - 中心价）

由此我们可以看到，R-Breaker 策略是根据昨天的价格绘制了一个类似网格的价格线，并且每天更新一次这些价格线。在技术分析上支撑位和阻力位，并且两者的作用可以互相转换。当价格成功向上突破阻力位时，阻力位变成了支撑位；当价格成功向下突破支撑位时，支撑位变成了阻力位。

在实际交易中，这些支撑位和阻力位为交易者指出了开平仓方向和精确等买卖点位。具体的开平仓条件交易者可以根据盘中价格、中心价、阻力位、支撑位灵活定制，也可以根据这些网格价格线进行加减仓的头寸管理。

4.14.3 策略逻辑

R-breaker 的交易规则分为趋势和反转两个策略。在空仓的情况下，如果盘中价格超过突破买入价，则采取趋势策略，即在该点位开仓做多；如果盘中价格跌破突破卖出价，则采取趋势策略，即在该点位开仓做空。

如果持多单，当日内最高价超过观察卖出价后，盘中价格出现回落，且进一步跌破反转卖出价构成的支撑线时，采取反转策略，即在该点位反手做空；如果持空单，当日内最低价低于观察买入价后，盘中价格出现反弹，且进一步超过反转买入价构成的阻力线时，采取反转策略，即在该点位反手做多。

趋势策略：

- ❑ 如果当前无持仓，并且盘中价格超过突破买入价，则多头开仓。
- ❑ 如果当前无持仓，并且盘中价格跌破突破卖出价，则空头开仓。

反转策略：

- ❑ 当日最高价大于观察卖出价，且价格向下跌破反转卖出价，空头开仓或多单反手。
- ❑ 当日最低价小于观察买入价，且价格向上突破反转买入价，多头开仓或空单反手。

趋势和反转相结合，更有利把握日内的投资机会。R-Breaker 策略在日内无明显的涨跌趋势时拒绝开仓，可以滤除大部分小幅盘整和震荡。能较早地捕获市场上突然出现的机会，并能在机会丧失殆尽前及时撤走，全身而退。

4.14.4 策略编写

第 1 步：编写策略框架

我们知道在量化交易中，程序是不断获取数据、处理数据、下单交易这样的循环过程，所以我们继续使用之前讲过的 main 函数和 onTick 函数，其中在 main 函数中无限循环执行 onTick 函数。如下：

```
# 策略主函数
def onTick():
    pass
```

```
# 程序入口
def main():
    while True:                                       # 进入无限循环模式
        onTick()                                      # 执行策略主函数
        Sleep(1000)                                   # 休眠 1 秒
```

第 2 步：获取数据

```
exchange.SetContractType('rb000')                     # 订阅期货品种
bars_arr =exchange.GetRecords(PERIOD_D1)              # 获取日 K 线数组
if len(bars_arr) < 2:                                 # 如果 K 线数量小于 2 根
    return
yesterday_open = bars_arr[-2]['Open']                 # 昨日开盘价
yesterday_high = bars_arr[-2]['High']                 # 昨日最高价
yesterday_low = bars_arr[-2]['Low']                   # 昨日最低价
yesterday_close = bars_arr[-2]['Close']               # 昨日收盘价
```

首先使用 SetContractType 函数订阅期货品种代码，比如 MA000 是以甲醇指数为数据，以甲醇主力合约下单；MA888 是以甲醇主力连续为数据，以甲醇主力合约下单。当然也可以指定具体交易合约代码。

接着使用 GetRecords 函数获取 K 线数据，该函数返回的是一维数组。在使用这个函数时，也可以传入一个参数，来获取指定的 K 线数据，比如传入“PERIOD_D1”是获取日线级别的 K 线。最后使用变量接收昨日的开高低收价格。

第 3 步：计算阻力与支撑

```
pivot = (yesterday_high + yesterday_close + yesterday_low)/3*num  # 枢轴点
r1 = 2 * pivot - yesterday_low                        # 阻力位 1
r2 = pivot + (yesterday_high - yesterday_low)         # 阻力位 2
r3 = yesterday_high + 2 * (pivot - yesterday_low)     # 阻力位 3
s1 = 2 * pivot - yesterday_high                       # 支撑位 1
s2 = pivot - (yesterday_high - yesterday_low)         # 支撑位 2
s3 = yesterday_low - 2 * (yesterday_high - pivot)     # 支撑位 3
today_high = bars_arr[-1]['High']                     # 今日最高价
today_low = bars_arr[-1]['Low']                       # 今日最低价
current_price = _C(exchange.GetTicker).Last           # 当前价格
```

我们根据上面定义的计算方法，首先计算中心价，然后分别计算 3 个阻力位和支撑位。这里需要注意的是，Python 的加减乘除与我们数学的计算一样，先乘除后加减，必要时需要加上括号。最后从 K 线数组中获取今天的最高价和最低价，以及当前最新的价格。

第 4 步：获取持仓量

```
position_arr = _C(exchange.GetPosition)               # 获取持仓数组
if len(position_arr) > 0:                             # 如果持仓数组长度大于 0
    for i in position_arr:
        if i['ContractType'][:2] == 'rb':             # 如果持仓品种等于订阅品种
            if i['Type'] % 2 == 0:                    # 如果是多单
                position = i['Amount']                # 赋值持仓数量为正数
            else:
                position = -i['Amount']               # 赋值持仓数量为负数
```

```
        profit = i['Profit']                            # 获取持仓盈亏
else:
    position = 0                                        # 赋值持仓数量为 0
    profit = 0                                          # 赋值持仓盈亏为 0
```

使用 GetPosition 函数可以获取到当前账户所有的持仓数据，返回的是一个数组，保险起见可以是_C 重试函数，以防出错。如果当前的持仓数组大于 0，证明有持仓数据；通过遍历该数组，获取数据中的 Amount 字段持仓量，并根据 Type 判断是持有多单还是空单。

第 5 步：下单交易

```
if position == 0:                                       # 如果无持仓
    if current_price > r3:                              # 如果当前价格大于阻力位 3
        exchange.SetDirection("buy")                    # 设置交易方向和类型
        exchange.Buy(current_price + 1, 1)              # 开多单
    if current_price < s3:                              # 如果当前价格小于支撑位 3
        exchange.SetDirection("sell")                   # 设置交易方向和类型
        exchange.Sell(current_price - 1, 1)             # 开空单
if position > 0:                                        # 如果持有多单
    # 如果今日最高价大于阻力位 2，并且当前价格小于阻力位 1
    if today_high > r2 and current_price < r1 or current_price < s3:
        exchange.SetDirection("closebuy")               # 设置交易方向和类型
        exchange.Sell(current_price - 1, 1)             # 平多单
        exchange.SetDirection("sell")                   # 设置交易方向和类型
        exchange.Sell(current_price - 1, 1)             # 反手开空单
if position < 0:                                        # 如果持有空单
    # 如果今日最低价小于支撑位 2，并且当前价格大于支撑位 1
    if today_low < s2 and current_price > s1 or current_price > r3:
        exchange.SetDirection("closesell")              # 设置交易方向和类型
        exchange.Buy(current_price + 1, 1)              # 平空单
        exchange.SetDirection("buy")                    # 设置交易方向和类型
        exchange.Buy(current_price + 1, 1)                  # 反手开多单
```

交易之前首先判断当前的持仓状态，然后根据当前价格与支撑线和阻力线的位置关系，判断交易的方向，如果条件成立，则使用 SetDirection 函数确定交易方向，最后使用 Buy 和 Sell 传入下单价格和下单量这两个参数下单。

注意：在下单之前先设置交易方向。

4.14.5 完整策略

```
# 策略主函数
def onTick():
    # 获取数据
    exchange.SetContractType('rb000')                       # 订阅期货品种
    bars_arr =exchange.GetRecords(PERIOD_D1)                # 获取日 K 线数组
    if len(bars_arr) < 2:                                   # 如果 K 线数量小于 2 根
        return
    yesterday_open = bars_arr[-2]['Open']                   # 昨日开盘价
```

```
yesterday_high = bars_arr[-2]['High']                   # 昨日最高价
yesterday_low = bars_arr[-2]['Low']                     # 昨日最低价
yesterday_close = bars_arr[-2]['Close']                 # 昨日收盘价
# 计算
pivot = (yesterday_high+yesterday_close+yesterday_low) / 3*num # 枢轴点
r1 = 2 * pivot - yesterday_low                          # 阻力位 1
r2 = pivot + (yesterday_high - yesterday_low)           # 阻力位 2
r3 = yesterday_high + 2 * (pivot - yesterday_low)       # 阻力位 3
s1 = 2 * pivot - yesterday_high                         # 支撑位 1
s2 = pivot - (yesterday_high - yesterday_low)           # 支撑位 2
s3 = yesterday_low - 2 * (yesterday_high - pivot)       # 支撑位 3
today_high = bars_arr[-1]['High']                       # 今日最高价
today_low = bars_arr[-1]['Low']                         # 今日最低价
current_price = _C(exchange.GetTicker).Last             #当前价格
# 获取持仓
position_arr = _C(exchange.GetPosition)                 # 获取持仓数组
if len(position_arr) > 0:                               # 如果持仓数组大于 0
    for i in position_arr:
        if i['ContractType'][:2] == 'rb':               # 如果持仓品种等于 rb
            if i['Type'] % 2 == 0:                      # 如果是多单
                position = i['Amount']                  # 赋值持仓数量为正数
            else:
                position = -i['Amount']                 # 赋值持仓数量为负数
            profit = i['Profit']                        # 获取持仓盈亏
else:
    position = 0                                        # 赋值持仓数量为 0
    profit = 0                                          # 赋值持仓盈亏为 0
if position == 0:                                       # 如果无持仓
    if current_price > r3:                              # 如果价格大于阻力位 3
        exchange.SetDirection("buy")                    # 设置交易方向和类型
        exchange.Buy(current_price + 1, 1)              # 开多单
    if current_price < s3:                              # 如果价格小于支撑位 3
        exchange.SetDirection("sell")                   # 设置交易方向和类型
        exchange.Sell(current_price - 1, 1)             # 开空单
if position > 0:                                        # 如果持有多单
    # 如果今日最高价大于阻力位 2，并且当前价格小于阻力位 1
    if today_high > r2 and current_price < r1 or current_price < s3:
        exchange.SetDirection("closebuy")               # 设置交易方向和类型
        exchange.Sell(current_price - 1, 1)             # 平多单
        exchange.SetDirection("sell")                   # 设置交易方向和类型
        exchange.Sell(current_price - 1, 1)             # 反手开空单
if position < 0:  # 如果持有空单
    # 如果今日最低价小于支撑位 2，并且当前价格大于支撑位 1
    if today_low < s2 and current_price > s1 or current_price > r3:
        exchange.SetDirection("closesell")              # 设置交易方向和类型
        exchange.Buy(current_price + 1, 1)              # 平空单
```

```
        exchange.SetDirection("buy")                    # 设置交易方向和类型
        exchange.Buy(current_price + 1, 1)              # 反手开多单

# 程序主函数
def main():
    while True:                                         # 循环
        onTick()                                        # 执行策略主函数
        Sleep(1000)                                     # 休眠 1 秒
```

Breaker 作为一种轻量级复合型策略，趋势和反转相结合，更有利于把握日内交易机会，并且当市场转入震荡时期，无明显涨跌的行情中，可以过滤大部分无效的交易，以复合型策略同时赚取趋势的 alpha 和反转的 alpha 收益。

4.15 温故知新

学完本章内容，读者需要回答：

1．什么是虚拟持仓？

2．试着用 talib 技术指标库获取不同的技术指标。

3．如何避免使用未来函数？

4．如何设置交易方向和类型？

在下一章中，读者会了解到：

1．正确的设置止损

2．风险和资金管理

第 5 章　CTA 之回归策略

趋势跟踪策略的表现形式是追涨杀跌，与之相反就是低买高卖的回归策略，也称均值回归策略。商品期货普遍存在一种规律，震荡行情远多于趋势行情，回归策略通过不断的低买高卖实现利润增长。

5.1 布林带跨期套利策略

索罗斯在 1987 年撰写的《金融炼金术》一书中，曾经提出过一个重要的命题：I believe the market prices are always wrong in the sense that they present a biased view of the future.市场有效假说只是理论上的假设，实际上市场参与者并不总是理性的，并且在每一个时间点上，参与者不可能完全获取和客观解读所有的信息，再者就算是同样的信息，每个人的反馈都不尽相同。也就是说，价格本身就已经包含了市场参与者的错误预期，所以本质上市场价

格总错误的。这或许是套利者的利润来源。

5.1.1 策略原理

在一个非有效的期货市场中，不同时期交割合约之间受到市场影响也并不总是同步，其定价也并非完全有效的原因。那么，根据同一种交易标的的不同时期交割合约价格为基础，如果两个价格出现了较大的价差幅度，就可以同时买卖不同时期的期货合约，进行跨期套利。比如：螺纹钢 2010 合约和螺纹钢 2105 合约。

举个例子，假设螺纹钢 2010 和螺纹钢 2105 的价差长期维持在 5 左右。如果某一天价差达到 7，我们预计价差会在未来某段时间回归到 5。那么就可以卖出螺纹钢 2010，同时买入螺纹钢 2105，来做空这个价差。反之亦然。尽管这种价差是存在的，但是人工操作耗时、准确性差以及价格变化的影响，人工套利往往存在诸多不确定性。通过量化模型捕捉套利机会并制定套利交易策略，以及程序化算法自动向交易所下达交易订单，快速准确捕捉机会，高效稳定赚取收益，这就是量化套利的魅力所在。

5.1.2 策略逻辑

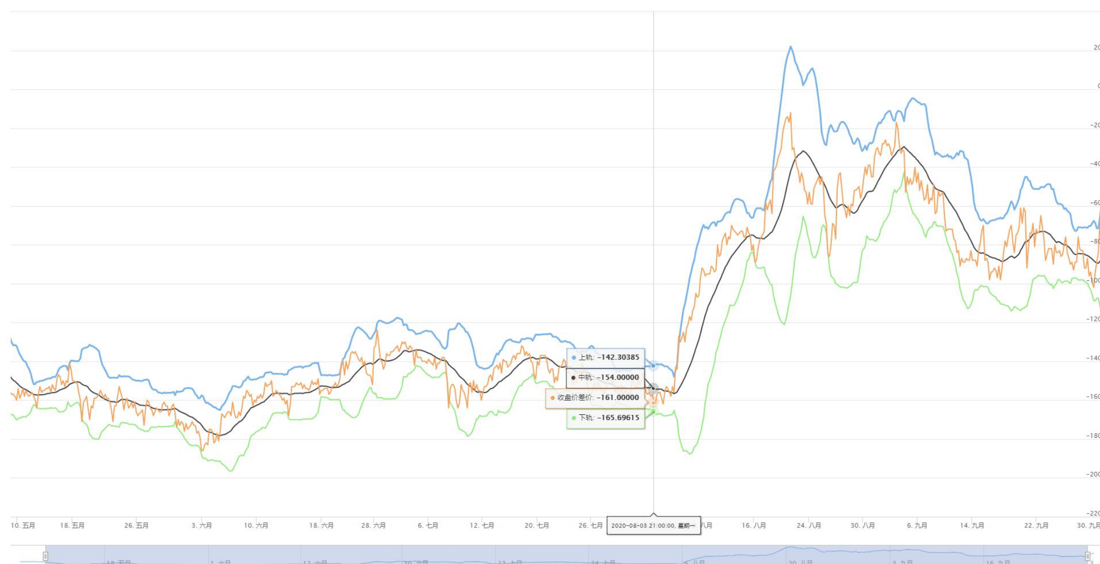

图 5.1 布林带图表

- 做多价差开仓条件：如果当前账户没有持仓，并且价差小于 boll 下轨，就做多价差。即：买开螺纹钢 2010，卖开螺纹钢 2105。
- 做空价差开仓条件：如果当前账户没有持仓，并且价差大于 boll 上轨，就做空价差。即：卖开螺纹钢 2010，买开螺纹钢 2105。
- 做多价差平仓条件：如果当前账户持有螺纹钢 2010 多单，并且持有螺纹钢 2105 空单，并且价差大于 boll 中轨，就平多价差。即：卖平螺纹钢 2010，买平螺纹钢 2105。

- ❑ 做空价差平仓条件：如果当前账户持有螺纹钢 2010 空单，并且持有螺纹钢 2105 多单，并且价差小于 boll 中轨，就平空价差。即：买平螺纹钢 2010，卖平螺纹钢 2105。

5.1.3 策略编写

```
# fmz@968ff1b01d4d2f125b3197281d743f12
class Hedge:
    def __init__(self, q, e, initAccount, symbolA, symbolB, maPeriod, atrRatio, opAmount):
        self.q = q
        self.initAccount = initAccount
        self.status = 0
        self.symbolA = symbolA
        self.symbolB = symbolB
        self.e = e
        self.isBusy = False
        self.maPeriod = maPeriod
        self.atrRatio = atrRatio
        self.opAmount = opAmount

    def poll(self):
        if (self.isBusy or not exchange.IO("status")) or not ext.IsTrading(self.symbolA):
            Sleep(1000)
            return
        exchange.SetContractType(self.symbolA)
        recordsA = exchange.GetRecords()
        exchange.SetContractType(self.symbolB)
        recordsB = exchange.GetRecords()
        if not recordsA or not recordsB:
            return
        if recordsA[-1]["Time"] != recordsB[-1]["Time"]:
            return
        minL, rA, rB = min(len(recordsA), len(recordsB)), recordsA.copy(), recordsB.copy()
        rA.reverse()
        rB.reverse()
        arrDiff = []
        for i in range(minL):
            arrDiff.append(rB[i]["Close"] - rA[i]["Close"])
        arrDiff.reverse()
        if len(arrDiff) < self.maPeriod:
            return
        boll = TA.BOLL(arrDiff, self.maPeriod, self.atrRatio)
```

```
        ext.PlotLine("上轨", boll[0][-2], recordsA[-2]["Time"])
        ext.PlotLine("中轨", boll[1][-2], recordsA[-2]["Time"])
        ext.PlotLine("下轨", boll[2][-2], recordsA[-2]["Time"])
        ext.PlotLine("收盘价差价", arrDiff[-2], recordsA[-2]["Time"])
        LogStatus(f"{_D()}\n 上轨：{boll[0][-1]}\n 中轨：{boll[1][-1]}\n 下轨：
{boll[2][-1]}\n 当前收盘差价：{arrDiff[-1]}")
        action = 0
        if self.status == 0:
            if arrDiff[-1] > boll[0][-1]:
                Log("开仓 A 买 B 卖", ", A 最新价格：", recordsA[-1]["Close"], ",
B 最新价格：", recordsB[-1]["Close"], "#FF0000")
                action = 2
            elif arrDiff[-1] < boll[2][-1]:
                Log("开仓 A 卖 B 买", ", A 最新价格：", recordsA[-1]["Close"], ",
B 最新价格：", recordsB[-1]["Close"], "#FF0000")
                action = 1
        elif self.status == 1 and arrDiff[-1] > boll[1][-1]:
            Log("平仓 A 买 B 卖", ", A 最新价格：", recordsA[-1]["Close"], ", B 最
新价格：", recordsB[-1]["Close"], "#FF0000")
            action = 2
        elif self.status == 2 and arrDiff[-1] < boll[1][-1]:
            Log("平仓 A 卖 B 买", ", A 最新价格：", recordsA[-1]["Close"], ", B 最
新价格：", recordsB[-1]["Close"], "#FF0000")
            action = 1
        if action == 0:
            return
        self.isBusy = True
        tasks = []
        if action == 1:
            tasks.append([self.symbolA, "sell" if self.status == 0 else
"closebuy"])
            tasks.append([self.symbolB, "buy" if self.status == 0 else
"closesell"])
        elif action == 2:
            tasks.append([self.symbolA, "buy" if self.status == 0 else
"closesell"])
            tasks.append([self.symbolB, "sell" if self.status == 0 else
"closebuy"])

        def callBack(task, ret):
            def callBack(task, ret):
                self.isBusy = False
                if task["action"] == "sell":
                    self.status = 2
                elif task["action"] == "buy":
                    self.status = 1
```

```
            else:
                self.status = 0
                account = _C(exchange.GetAccount)
                LogProfit(account["Balance"]                                -
self.initAccount["Balance"], account)
          self.q.pushTask(self.e, tasks[1][0], tasks[1][1], self.opAmount,
callBack)
       self.q.pushTask(self.e,  tasks[0][0],  tasks[0][1],  self.opAmount,
callBack)

def main():
    while not exchange.IO("status"):
        Sleep(1000)
    initAccount = _C(exchange.GetAccount)
    q = ext.NewTaskQueue()
    p = ext.NewPositionManager()
    if CoverAll:
        p.CoverAll()
    t = Hedge(q, exchange, initAccount, SA, SB, MAPeriod, ATRRatio, OpAmount)
    while True:
        q.poll()
        t.poll()
```

策略参数设置如下：

变量	描述	类型	默认值
SA	合约A	字符串（string）	rb2010
SB	合约B	字符串（string）	rb2101
OpAmount	开仓手数	数字型（number）	1
CoverAll	启动时平掉所有仓位	布尔型（true/false）	false
MAPeriod	布林周期参数	数字型（number）	20
ATRRatio	布林乘数参数	数字型（number）	2

图 5.2　布林带策略参数

如上面的代码，该策略首先订阅了 rb2010 和 rb2101 合约，并分别获取了它们的 K 线数据，然后计算两个合约的价差，接着以价差为数据计算布林带指标，最后实现当差价超过布林线上轨时正对冲，触碰下轨时反对冲。持仓时触碰布林中线平仓。

5.1.4　策略回测

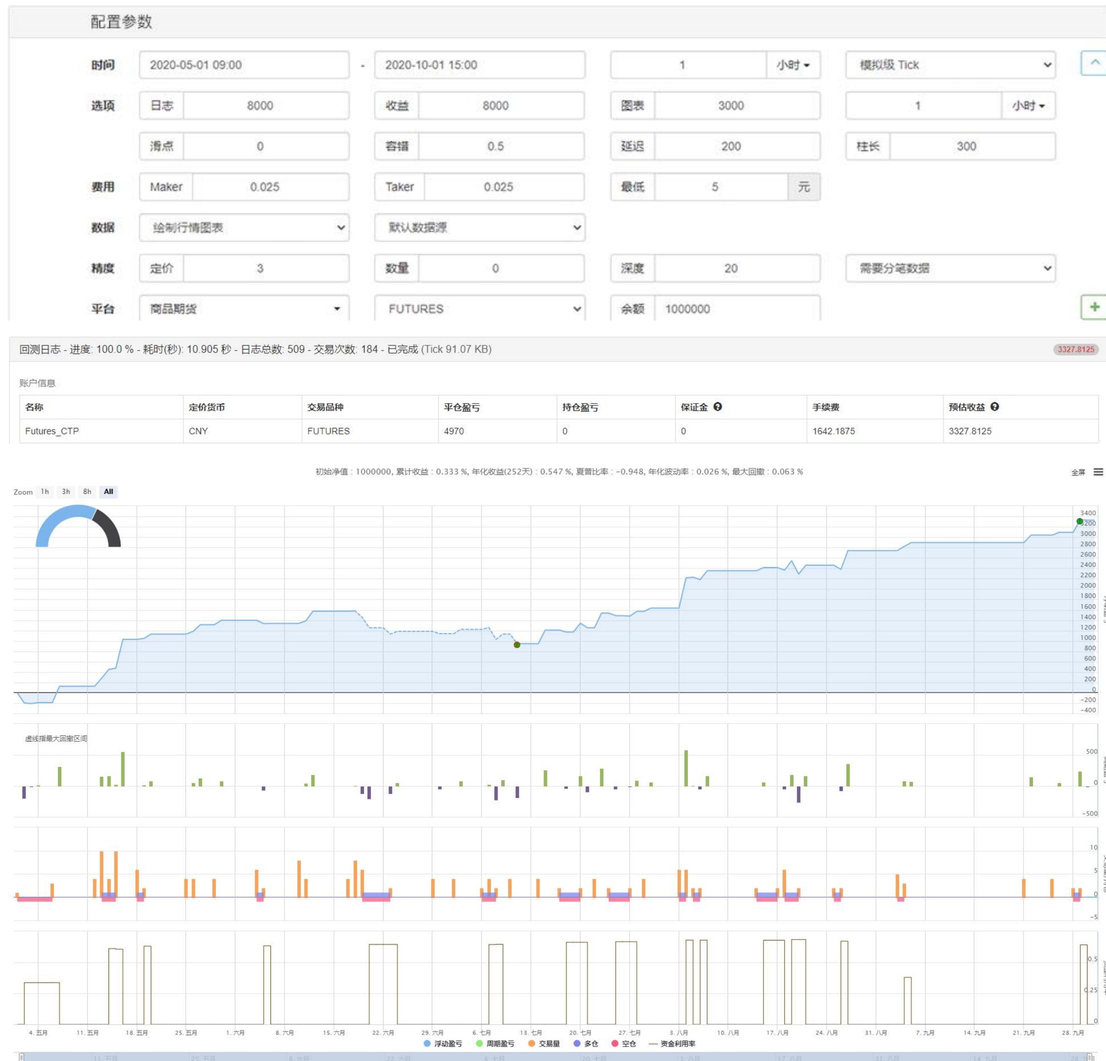

图 5.3　布林带策略回测

5.2　期现套利图表

“套利”在现实生活中却很常见。比如：便利店老板从批发市场以 0.5 元买入一瓶矿泉水，然后在店里以 1 元的价格出售，最后赚取 0.5 元的差价。这个过程其实就类似套利。金融市场上的套利跟这个道理差不多，只不过套利的形式有多种多样。

5.2.1 什么是套利

在商品期货市场中，理论上 5 月份交割的苹果合约价格减去 10 月交割的苹果合约价格，其结果应该接近于 0 或者稳定在一定的价格区间内。但事实上由于受到天气、市场供需等诸多因素的原因，近期和远期合约价格在一段时间内会分别受到不同程度的影响，价差也会出现较大幅度的波动。

但无论如何，价差通常最终会回归到一定的价格区间内，那么如果价差大于这个区间，就卖 5 月份合约，同时买 10 月份合约，做空价差赚取利润；如果价差小于这个区间，就买 5 月份合约，同时卖 10 月份合约，做多价差赚取利润。这就是通过买卖同一个品种但不同交割月份的跨期套利。

除了跨期套利外，还有买入出口国大豆同时卖出进口国大豆，或者卖出出口国大豆同时买入进口国大豆的跨市场套利；买入上游原材料铁矿石同时卖出下游成品螺纹钢，或者卖出上游原材料铁矿石同时买入下游成品螺纹钢的跨品种套利等等。

5.2.2 期现套利方法

但是上面这几种套利方法，虽然字面上是“套利”，并不属于纯粹意义上的套利，它们本质上还是属于有风险的投机，只不过这种投机的方式是做多或做空价差。虽然价差在大部分时间内趋于稳定，但也可能出现很长时间不回归的行情。

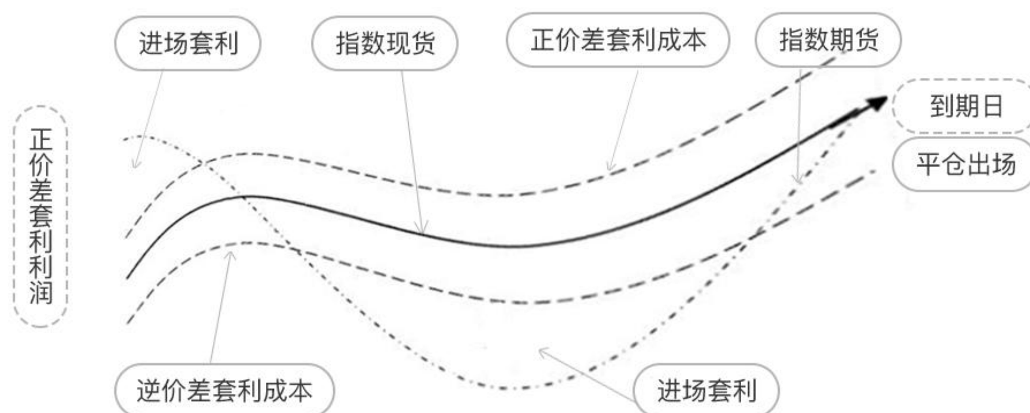

图 5.4 期现套利方法

期现套利的核心原理是，同一个商品在同一个时间点只能有一个价格，期货到了交割时间就变成现货，所以在临近交割时会强制回归。这个与跨期套利完全不同，跨期套利是两个不同交割月的合约，到期时也就是两个不同月份的现货，当然可以是两个价格。

- 基差 = 期货价格 - 现货价格

期现套利最大的特点是理论上无风险，主要是根据基差状态，计算利润区间。如果基差过大，就可以买入现货，同时做空期货，等待基差重新归零，就可以期货和现货双边平

仓，赚取基差的利润。主要有两种方法：双平套利和交割套利。

5.2.3 期现套利方法

说起来简单，最复杂的一个环节是商品现货交易，这其中涉及到仓单、税务等等一系列问题。首先需要一个与投资范围相关的公司，如果是交割套利期货账户必须是公司法人的，如果双平套利就需要一个可靠的销售渠道。网上有很多现货交易的网站，如：卓创网、上海有色网等等。

需要注意的是，现货交易通常有 17%~20%的增值税，所以如果是双平套利，在买入现货后需要做空 1.2~1.25 倍的期货。如果是交割套利，在买入现货后需要做空相同比例的期货，还需要考虑手续费、运输、仓库等成本。当然这一切的前提是期现的基差足够大，有足够的边界。

除此之外，由于上海黄金交易所黄金(T+D)的存在，使得黄金期现套利不仅可以正向套利，还可以无需黄金租赁进行反向套利操作。上海黄金交易所的现货黄金(T+D)延期交易不仅交易方便，而且成交量和持仓量大，流动性非常适合期现套利。

5.2.4 获取数据

现货和基差的数据网上有很多种，大部分是以表格的形式呈现，这显然不合适用来分析判断行情。发明者量化(FMZ.COM)已经内置了商品期货基本面数据，包括现货数据和基差数据。只需要调用一个函数就可以获取每个品种的现货和基差价格，并且支持 2016 年至今的历史数据。

名称	代码	名称	代码	名称	代码	名称	代码	名称	代码
基差	BASIS	企业景气及企业家信心指数	BCEEI	交易结算资金(银证转账)	BTSF	存款准备金率	CKZBJ	居民消费价格指数	CPI
外商直接投资数据	FDI	外汇和黄金储备	FEGR	国内生产总值	GDP	全国股票交易统计表	GPJYTJ	股票账户统计详细数据	GPZHSJ
货币供应量	HBGYL	海关进出口增减情况一览表	HGJCK	消费者信心指数	ICS	贷款市场报价利率	LRP	新房价指数	NREPI
旧房价指数	OREPI	采购经理人指数	PMI	工业品出厂价格指数	PPI	全国税收收入	QGSSSR	企业商品价格指数	QYSPJG
社会消费品零售总额	TRSCG	城镇固定资产投资	UIFA	期货标准仓单	WHR	本外币存款	WBCK	外汇贷款数据	WHXD
银行利率调整	YHLL	交易所会员成交量及持仓明细	DTP	交易所会员持仓盈亏	FCPP	财政收入	CZSR	工业增加值增长	GYZJZ
汽柴油价格	OILPRICE	现货价格	SPOTPRICE	新增信贷数据	XZXD				

图 5.5　fmz 基本面数据

代码如下：

```
# 回测配置
'''backtest
start: 2020-06-01 00:00:00
end: 2020-06-02 00:00:00
period: 1d
basePeriod: 1h
exchanges: [{"eid":"Futures_CTP","currency":"FUTURES"}]
'''
```

```
# 策略入口
def main():
    while True:
        ret = exchange.GetData("GDP")  # 调用国内生产总值数据
        Log(ret)  # 打印数据
        Sleep(1000 * 60 * 60 * 24 * 30)
```

输出结果为：

```
{
    "季度": "2006 年第 1 季度",
    "国内生产总值": {
        "绝对值(亿元)": 47078.9,
        "同比增长": 0.125
    },
    "第一产业": {
        "绝对值(亿元)": 3012.7,
        "同比增长": 0.044
    },
    "第三产业": {
        "绝对值(亿元)": 22647.4,
        "同比增长": 0.131
    },
    "第二产业": {
        "绝对值(亿元)": 21418.7,
        "同比增长": 0.131
    }
}
```

5.2.5 现货和基差图表

让我们用发明者量化，以图表的形式，把现货价格和基差价格实现出来。首先注册并登陆发明者量化官网(FMZ.COM)，点击控制中心，点击策略库+新建策略。在左上角下拉菜单中选择 Python，并填入策略的名字。

第一步：编写策略框架

```
# 策略主函数
def onTick():
    pass


# 策略入口
def main():
    while True:  # 进入循环模式
        onTick()  # 执行策略主函数
        Sleep(1000 * 60 * 60 * 24)  # 策略休眠一天
```

策略框架是两个函数，main 函数是策略的入口，主要功能是交易之前的预处理，程序

会先从 main 函数开始执行，然后进入无线循环模式，重复执行 onTick 函数，onTick 函数是策略的主函数，主要执行核心代码。

第二步：增加图表功能

```
# 全局变量
# 期现图表
cfgA = {
    "extension": {
        "layout": 'single',
        "col": 6,
        "height": "500px",
    },
    "title": {
        "text": "期现图表"
    },
    "xAxis": {
        "type": "datetime"
    },
    "series": [{
        "name": "期货价格",
        "data": [],
    }, {
        "name": "现货价格",
        "data": [],
    }
    ]
}
# 基差图表
cfgB = {
    "extension": {
        "layout": 'single',
        "col": 6,
        "height": "500px",
    },
    "title": {
        "text": "基差图表"
    },
    "xAxis": {
        "type": "datetime"
    },
    "series": [{
        "name": "基差价格",
        "data": [],
    }]
}
chart = Chart([cfgA, cfgB])  # 创建一个图表对象
```

```
# 策略主函数
def onTick():
    chart.add(0, [])  # 绘制图表
    chart.add(1, [])  # 绘制图表
    chart.add(2, [])  # 绘制图表
    chart.update([cfgA, cfgB])  # 更新图表


# 策略入口
def main():
    LogReset()  # 运行前先清空之前的 Log 日志信息
    chart.reset()  # 运行前先清空之前的图表信息
    while True:  # 进入循环模式
        onTick()  # 执行策略主函数
        Sleep(1000 * 60 * 60 * 24)  # 策略休眠一天
```

在这个策略中，一共创建了 2 个图表，并左右分布排列。其中左图 cfgA 是期现图表，包含期货价格和现货价格，右图 cfgB 是基差图表。然后调用发明者量化内置的 Python 版画线类库创建一个 chart 对象。最后是在 onTick 函数中实时更新图表中的数据。

第三步：获取数据

```
last_spot_price = 0  # 保存最后一个有效的现货价格
last_basis_price = 0  # 保存最后一个有效的基差价格

def onTick():
    global last_basis_price, last_spot_price  # 导入全局变量
    exchange.SetContractType("i888")  # 订阅期货品种
    futures = _C(exchange.GetRecords)[-1]  # 获取最新 K 线数据
    futures_ts = futures.Time  # 获取最新 K 线期货时间戳
    futures_price = futures.Close  # 获取最新 K 线收盘价

    spot = exchange.GetData("SPOTPRICE")  # 获取现货数据
    spot_ts = spot.Time  # 获取现货时间戳
    if '铁矿石' in spot.Data:
        spot_price = spot.Data['铁矿石']
        last_spot_price = spot_price
    else:
        spot_price = last_spot_price

    basis = exchange.GetData("BASIS")  # 获取基差数据
    basis_ts = basis.Time  # 获取基差时间戳
    if '铁矿石' in basis.Data:
        basis_price = basis.Data['铁矿石']
        last_basis_price = basis_price
    else:
        basis_price = last_basis_price
```

我们一共需要获取三种数据：期货价格、现货价格、基差价格。获取期货价格很简单，直接使用 SetContractType 函数订阅期货品种，再使用 GetRecords 函数就可以获取 K 线的收盘价。现货和基差的价格，可以使用之前介绍的方法，使用 GetData 函数调用基本面数据代码，返回的是包含时间戳的字典数据。

5.2.6　图表展示

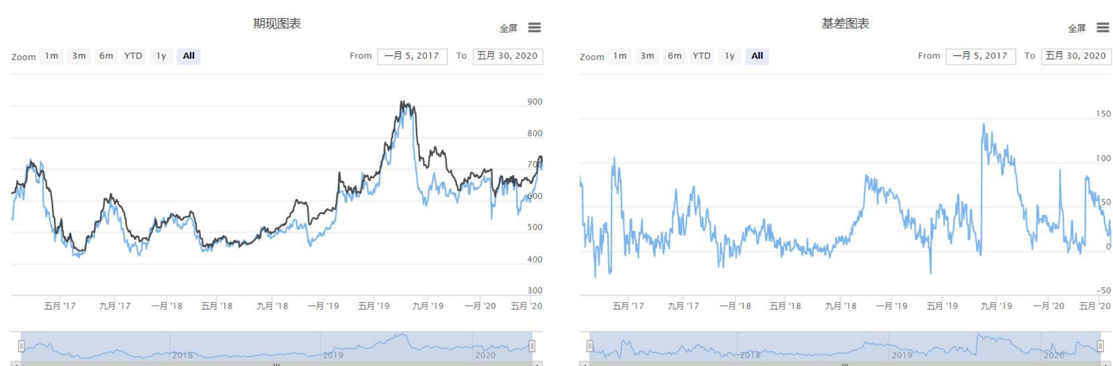

图 5.6　期货、现货、基差图表

套利的没有想象中那么复杂，它不需要太多的金融理论知识，也不需要太过复杂的数学或统计模型，套利本质上就是赚取价格从不合理到回归合理之间的利润而已。市场行情每年都在变化，对于交易者而言，最好不要把历史数据完全照搬到现在，而是结合当下数据研究基差是否合理。

5.3　乖离率 BIAS 策略

俗话说分久必合合久必分，在期货市场也有这种现象，没有只涨不跌的品种也没有只跌不涨的品种。但是什么时候分什么时候合，这就要看乖离率了。本节我们将使用乖离率构建一个简单的策略。

5.3.1　乖离率简介

乖离率 BIAS 是由移动平均线衍生出来的一种技术指标，它主要是以百分比的形式，衡量价格在波动中与移动平均线的偏离程度。如果说均线是交易者的平均成本，那么乖离率就是交易者的平均回报率。

注意：相对来说乖离率是一个比较冷门的技术指标，但在量化交易中使用乖离率可以增加策略的多样性。

图 5.7　乖离率指标

5.3.2　乖离率的原理

乖离率的理论基础是对交易者的心里分析，当价格大于市场平均成本太多时，表示多头交易者获利越丰厚，容易萌生赚钱就走的念头，进而会造成价格下跌。当价格小于市场平均成本太多时，表示空头交易者获利丰厚，容易萌生赚钱就走的念头，进而会造成价格上涨。

- 当价格向上偏离均线时，乖离率过大，未来价格有很大几率会下跌。
- 当价格向下偏离均线时，乖离率过小，未来价格有很大几率会上涨。

虽然移动平均线是由价格计算而来，但从外在形式上价格一定会向移动平均线靠拢，或者说价格总是围绕着移动平均线上下波动。如果价格偏离均线太远，不管价格是在均线之上还是之下，最后都可能趋向于均线，而乖离率正是表示价格偏离均线的百分比值。

5.3.3　乖离率计算公式

- 乖离率=[(当日收盘价-N 日平均价)/N 日平均价]*100%

其中，N 是移动均线参数，由于 N 的周期不同，乖离率的计算结果也不同。一般情况下 N 的取值是：6、12、24、36 等等。在实际使用中，也可以根据不同的品种动态调整。但参数的选择十分重要，如果参数过小，乖离率就会过于敏感，如果参数过大，乖离率就会过于迟钝。乖离率的计算结果有正负之分，正的乖离率越大，代表多头获利越大，价格回调的概率越大。负的乖离率越大，代表空头获利越大，价格反弹的概率越大。

5.3.4 策略逻辑

由于乖离率是另一种均线的表现形式，那么我们也可以根据双均线策略改编一个双乖离率策略。通过短期乖离率与长期乖离率的位置关系，判断当前的市场状态。如果长期乖离率大于短期乖离率实际代表着短期均线金叉长期均线，反之亦然。

- 多头开仓：如果当前无持仓，并且长期乖离率大于短期乖离率
- 空头开仓：如果当前无持仓，并且长期乖离率小于短期乖离率
- 多头平仓：如果当前持多单，并且长期乖离率小于短期乖离率
- 空头平仓：如果当前持空单，并且长期乖离率大于短期乖离率

5.3.5 策略编写

第 1 步：编写策略框架

```
# 策略主函数
def onTick():
    pass

# 程序入口
def main():
    while True:      # 进入无限循环模式
        onTick()     # 执行策略主函数
        Sleep(1000)  # 休眠 1 秒
```

发明者量化(FMZ.COM)采用轮训模式，首先需要定义一个 main 函数和一个 onTick 函数，main 函数是策略的入口函数，程序会从 main 函数开始逐行执行代码。在 main 函数中，写入 while 循环，重复执行 onTick 函数，所有的策略核心代码都写在 onTick 函数中。

第 2 步：定义虚拟持仓和外部变量

```
short = 10
long = 50
mp = 0
```

虚拟持仓的好处是编写简单，快速迭代策略更新，一般用于回测环境中，假设每一笔订单都完全成交，但在实际交易中常用的还是真实持仓。由于虚拟持仓是记录开平仓后的状态，所以需要定义成全局变量。

第 3 步：获取 K 线

```
exchange.SetContractType('rb000')          # 订阅期货品种
bars_arr = exchange.GetRecords()           # 获取 K 线数组
if len(bars_arr) < long + 1:               # 如果 K 线数量过小
    return
```

使用发明者量化的 SetContractType，传入“rb000"就可以订阅螺纹钢指数合约，但在回测和实盘中，是以螺纹钢指数为数据，使用具体的主力合约下单。接着使用 GetRecords 函数就可以获取螺纹钢指数的 K 线数据了。

注意：由于在计算乖离率时需要一定周期，所以为了避免程序出错，在没有足够 K 线的时候，使用 if 语句过滤。

第 4 步：计算乖离率

```
close = bars_arr[-2]['Close']                # 获取上一根 K 线收盘价
ma1 = TA.MA(bars_arr, short)[-2]             # 计算上一根 K 线短期均线值
bias1 = (close - ma1) / ma1 * 100            # 计算短期乖离率值
ma2 = TA.MA(bars_arr, long)[-2]              # 计算上一根 K 线长期均线值
bias2 = (close - ma2) / ma2 * 100            # 计算长期乖离率值
```

根据乖离率计算公式，首先获取收盘价，在这个策略中我们使用的是上一根 K 线收盘价，也就是当前 K 线信号成立，下根 K 线发单。接着使用发明者量化内置的 talib 库计算均线。

第 5 步：下单交易

```
global mp                                    # 全局变量
current_price = bars_arr[-1]['Close']        # 最新价格
if mp > 0:                                   # 如果持有多单
    if bias2 <= bias1:                       # 如果长期乖离率小于等于短期乖离率
        exchange.SetDirection("closebuy")    # 设置交易方向和类型
        exchange.Sell(current_price - 1, 1)  # 平多单
        mp = 0                               # 重置虚拟持仓
if mp < 0:                                   # 如果持有空单
    if bias2 >= bias1:                       # 如果长期乖离率大于等于短期乖离率
        exchange.SetDirection("closesell")   # 设置交易方向和类型
        exchange.Buy(current_price + 1, 1)   # 平空单
        mp = 0                               # 重置虚拟持仓
if mp == 0:                                  # 如果无持仓
    if bias2 > bias1:                        # 长期乖离率大于短期乖离率
        exchange.SetDirection("buy")         # 设置交易方向和类型
        exchange.Buy(current_price + 1, 1)   # 开多单
        mp = 1                               # 重置虚拟持仓
    if bias2 < bias1:                        # 长期乖离率小于短期乖离率
        exchange.SetDirection("sell")        # 设置交易方向和类型
        exchange.Sell(current_price - 1, 1)  # 开空单
        mp = -1                              # 重置虚拟持仓
```

由于我们在 while 循环外部定义了一个全局变量 mp，用于接收当前的持仓状态，所以在使用这个变量的时候，需要先用 global 引入这个全局变量。另外还需要获取当前的最新价格用于开平仓。

5.3.6　策略回测

回测日志 - 进度: 100.0 % - 耗时(秒): 24.339 秒 - 日志总数: 159 - 交易次数: 159 - 已完成 (Tick 230.71 KB)　0.0000

账户信息

名称	定价货币	交易品种	平仓盈亏	持仓盈亏	保证金	手续费	预估收益
Futures_CTP	CNY	FUTURES	12290	70	2132.4	1465.287	10894.712

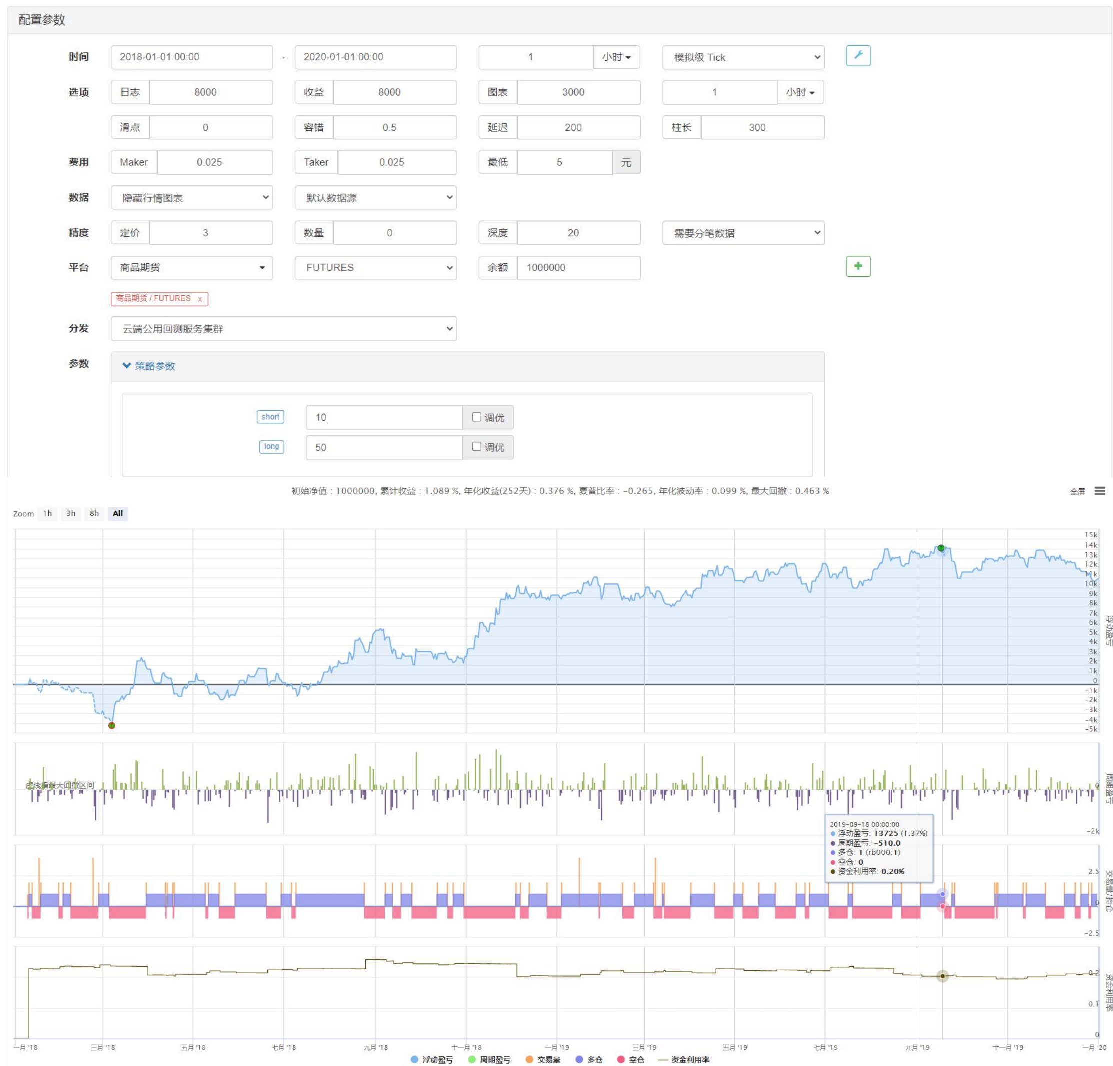

图 5.8　乖离率指标

5.3.7　完整策略

```
# 外部参数和全局变量
short = 10
long = 50
mp = 0

# 策略主函数
def onTick():
    # 获取数据
    exchange.SetContractType('rb000')            # 订阅期货品种
    bars_arr = exchange.GetRecords()             # 获取 K 线数组
    if len(bars_arr) < long + 1:                 # 如果 K 线数量过小
```

```
        return

    # 计算 BIAS
    close = bars_arr[-2]['Close']                # 获取上一根 K 线收盘价
    ma1 = TA.MA(bars_arr, short)[-2]             # 计算上一根 K 线短期均线值
    bias1 = (close - ma1) / ma1 * 100            # 计算短期乖离率值
    ma2 = TA.MA(bars_arr, long)[-2]              # 计算上一根 K 线长期均线值
    bias2 = (close - ma2) / ma2 * 100            # 计算长期乖离率值

    # 下单交易
    global mp                                    # 全局变量
    current_price = bars_arr[-1]['Close']        # 最新价格
    if mp > 0:                                   # 如果持有多单
        if bias2 <= bias1:                       # 如果长期乖离率<=短期乖离率
            exchange.SetDirection("closebuy")    # 设置交易方向和类型
            exchange.Sell(current_price - 1, 1)  # 平多单
            mp = 0                               # 重置虚拟持仓
    if mp < 0:                                   # 如果持有空单
        if bias2 >= bias1:                       # 如果长期乖离率>=短期乖离率
            exchange.SetDirection("closesell")   # 设置交易方向和类型
            exchange.Buy(current_price + 1, 1)   # 平空单
            mp = 0                               # 重置虚拟持仓
    if mp == 0:                                  # 如果无持仓
        if bias2 > bias1:                        # 长期乖离率大于短期乖离率
            exchange.SetDirection("buy")         # 设置交易方向和类型
            exchange.Buy(current_price + 1, 1)   # 开多单
            mp = 1                               # 重置虚拟持仓
        if bias2 < bias1:                        # 长期乖离率小于短期乖离率
            exchange.SetDirection("sell")        # 设置交易方向和类型
            exchange.Sell(current_price - 1, 1)  # 开空单
            mp = -1                              # 重置虚拟持仓


# 程序入口函数
def main():
    while True:                                  # 循环
        onTick()                                 # 执行策略主函数
        Sleep(1000)                              # 休眠 1 秒
```

本节我们学习了乖离率的原理，以及使用乖离率构建了一个简单的交易策略。在实际交易中乖离率是一种简单有效的交易工具，能为交易者提供有效的参考。

5.4 温故知新

学完本章内容，读者需要回答：

1．回归策略与趋势策略有什么不同？
2．期货套利的原理是什么？
在下一章中，读者会了解到：
1．回测数据和绩效报告
2．如何避免回测中的陷阱
3．如何使用科学的方法回测

第 6 章　用科学的方法回测策略

在第 4 章和第 5 章中详细介绍了如何开发一个交易策略，包括 CTA 趋势策略和震荡策略。一个新开发出来的交易策略，需要全方位检测才能应用于实战，同样一个优秀的策略也是在试错中不断改进得以产生。

本章主要涉及到的知识点有：

- 数据升级：使用 Tick 级别数据让回测更加精准。
- 解读回测报告：从回测报告中分析交易策略需要改进的方向。
- 规避回测陷阱：规避量化交易回测常见的陷阱。
- 策略优化：优化策略进出场条件使其更加可靠。

6.1　使用 Tick 数据让回测更精准

量化交易回测的目的是还原交易过程，进而验证策略的逻辑和可行性，所以回测的准确性尤为重要。CTA 策略有很多种风格，从频率上来讲：有高频策略、中低频策略。从另一个角度来讲：有日内策略，也有隔夜策略。通常不同类型策略对于回测时选用的数据也是不同的。

6.1.1　回测需要哪些数据

如何做到精准回测是很多量化交易者关心的问题，那么首先要弄清楚回测中都有哪些数据，因为数据的质量很大程度上已经决定了回测的质量。常见的数据有开盘价、最高价、最低价、收盘价、成交量等等，这些统称为 K 线数据。

另外还有一种原始的 Tick 快照，如果把交易所内的数据想象成一条河流，其中包含每个订单的详细数据，那么 Tick 快照就是这个数据流中的某个切片，频率是每秒 2 次，是当时某一时刻市场交易情况的再现。

事实上 K 线数据就是基于 Tick 数据合成的，按照时间周期 1 分钟的 K 线数据是由 1 分钟内的 Tick 数据组成，5 分钟的 K 线数据是由 5 分钟内的 Tick 数据组成，以此类推......形

成了各种分钟图、小时图、日线图等等。这就意味着一分钟的 K 线数据可能包含 120 个 Tick 数据。因此，回测的历史数据可以分为：K 线数据和 Tick 数据，并且在同一个周期内 Tick 的数据量要比 K 线数据量大很多，理论上 Tick 数据比 K 线数据回测更加准确。

6.1.2 基于 Bar 数据的回测

市面上量化交易软件几乎都支持 K 线数据的回测，由于数据量少，大大简化了回测引擎的工作量，所以这种回测通常都非常快，十年左右的数据几秒之内就能回测完，甚至叠加几十个期货品种回测也不会超过一分钟。但是 K 线数据回测有很多问题：

1、极端价格

做过交易的人都知道，在涨停中很难买入，在跌停中很难卖出，但是在回测中是可以成交的，一些做量化交易的新手，如果不在策略中对涨跌停价进行过滤，回测的结果会与实盘不一致。

2、价格真空

从跌停瞬间到涨停或者价格跳空上涨，在大周期 K 线图上看是一根大阳线，但是这根 K 线中间的实质挂单很少，如果是即时价成交的策略，在 Bar 数据上回测，是可以成交的。举个例子，当前 K 线一直在 5000 价格附近徘徊，临近收盘瞬间涨至 5100，并且中间几乎没有挂单和成交。如果策略信号在这根 K 线上是 5050 开仓，那么在 K 线数据回测中是可以成交的。

3、过去和未来的数据

理论上 K 线的形成可能是：开盘价>>>最低价>>>最高价>>>收盘价。但实际上它有可能先创新高，再创新低，再收盘；也有可能先创新低，再创新高，再收盘；甚至也可能一波三折先创新低，再创新高，再创新低，再拉高收盘；表面上看是一根有上影线和下影线的 K 线，中间的过程有很多种可能。

假如有一根 K 线是这样的：开盘价 4950、最低价 4900、最高价 5100、收盘价 5050，一根普通阳线。策略是：如果最新价超过前期高点 5000 就买入，买入后设置 1%的止损，也就是价格跌破 4950 就止损。

模拟回测：开盘 4950>>>价格超过前期高点 5000>>>信号成立买入开仓>>>收盘时赚了 1%；但真实的情况可能是这样的：开盘 4950>>>价格超过前期高点 5000>>>信号成立买入开仓>>>不久价格开始下跌>>>继续下跌至 4949>>>止损信号成立卖出平仓亏损 1%>>>价格上涨 5100>>>价格下跌至 5050 收盘。同样的策略，在 K 线数据回测和真实的交易中，会有两种截然不同的结果。

6.1.3 基于 Tick 数据的回测

如果能用 Tick 数据来做回测和分析，无疑具有很大的优势，但目前市面上很少能真正做到，有些量化交易软件只是使用了 Tick 价格，并没有使用 Tick 挂单量，可能造成见价成交的现象。比如当前的 Tick 数据是：卖价 5001、买价 5000，如果挂的买单时 5000，结果

肯定是买不到的，但事实并非如此。

在真实的交易环境中，订单是在交易所的 Tick 数据流中完成撮合的，交易所的撮合规则是：价格优先、时间优先。如果此时盘口订单只要不是太厚，那么所挂的 5000 买单，是有可能被动成交的。

6.1.4 盘口数据回测引擎原理

真正的 Tick 数据回测不仅根据 Tick 数据的价格优先来撮合订单，还根据价格相同时间优先，通过计算盘口挂单量，来判断当前挂单是否达到被动成交的条件实现见量成交，以此做到真正的模拟实盘环境。以下图为例：

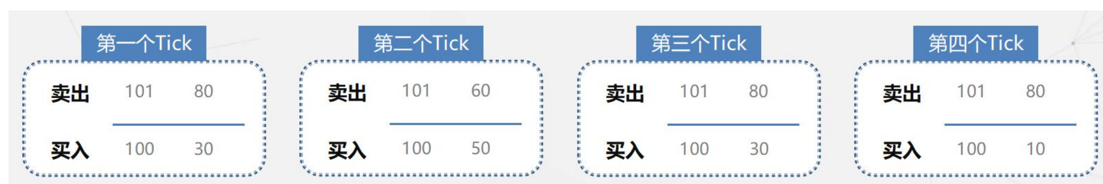

图 6.1 盘口数据

首先第 1 个 Tick 买价是 100，挂单量是 30 手；此时产生了买入信号，以 100 的价格买入 20 手等待被动成交；第 2 个 Tick 产生了，买价是 100，挂单量是 50 手，这里面有我们 20 手的挂单；第 3 个 Tick 产生了，买价是 100，挂单量是 30 手，这证明已经有 20 手买单被成交了或者撤单了，我们离成交又近了一步；第 4 个 Tick 产生了，买价是 100，挂单量是 10 手，是一个大卖家，一下子把我们买单全部成交。

通过上面的例子，我们可以发现，在 Tick 数据中，价格未变的前提下，可以通过盘口挂单量的变化，来推算自己的挂单有没有被动成交。利用的就是价格相同，时间优先的方法。这种回测引擎几乎最大程度的仿生了真实的实盘交易环境，杜绝了见价成交和虚假成交，让每一个盘口数据真实回放，使回测与实盘最大可能的一致。

6.1.5 如何选择最佳回测方式

通常中低频策略交易次数不多，滑点成本对策略的影响较小，对数据精度要求也不是太高，所以一般情况下使用 K 线数据回测，只需要在回测的时候加上几跳的滑点就可以，真正需要注意的是过度拟合的问题。

有些日内交易或者涉及到日内开平仓交易的策略，如果有必要，也可以在回测的配置参数页面上调整数据粒度，比如在 1 小时周期上回测，可以调整为更精细的 15 分钟数据力度。必要时也可以使用 Tick 级别的数据，来提高回测的精准度。

注意：如果是隔夜的中低频策略尽量以商品指数为数据，如果是日内的中低频策略尽量以商品主力连续为数据。

高频交易因为策略交易的次数足够多，单品种一天就能交易几十甚至上百次，所以只要撮合引擎是合理的，那么在大数定律的作用下，回测的结果基本靠谱，一般不存在过度

拟合的问题。但是由于高频交易的次数很多，因此就需要对回测数据有非常高的要求。

因为在高频交易回测中，交易频率越高，持仓的时间周期就越短，单笔的平均利润就越低，此时如果回测引擎设计的不合理，或者撮合方式与真实的交易环境不一样，那么就会出现“差之毫厘，谬以千里”的现象，所以对于高频交易来说，盘口级别的回测引擎是不二之选。

6.2 回测绩效报告详解

量化交易与主观交易最重要的区别之一是量化交易通过历史数据复盘，得出一系列绩效报告，交易者可以从报告中发现策略的缺点来优化和改进策略。当然优化和改进策略的前提是需要对策略有一个正确的评价，那么就需要对回测的绩效报告有一定的了解。只有正确解读回测绩效报告，才能知道策略需要改进的方向。

6.2.1 回测配置参数

图 6.2 回测配置详解

即使是相同的策略，回测配置参数不一样，也会影响最终的回测结果。在解读回测绩效报告之前，先来看下回测配置，上图是发明者量化软件的回测界面，每个参数的解释说明如下：

1、回测开始时间

2、回测终止时间

3、数据周期，可以选择：天、小时及分钟的任意数值

4、回测数据级别，可以选择：模拟级 Tick 和实盘级 Tick

5、底层 K 线周期，可以选择：天、小时及分钟的任意数值

6、滑点
7、最低手续费，手续费是双向收取
8、行情图表，可以选择：绘制行情图表和隐藏行情图表
9、回测起始资金
10、添加交易所
11、策略外部参数
12、开始回测

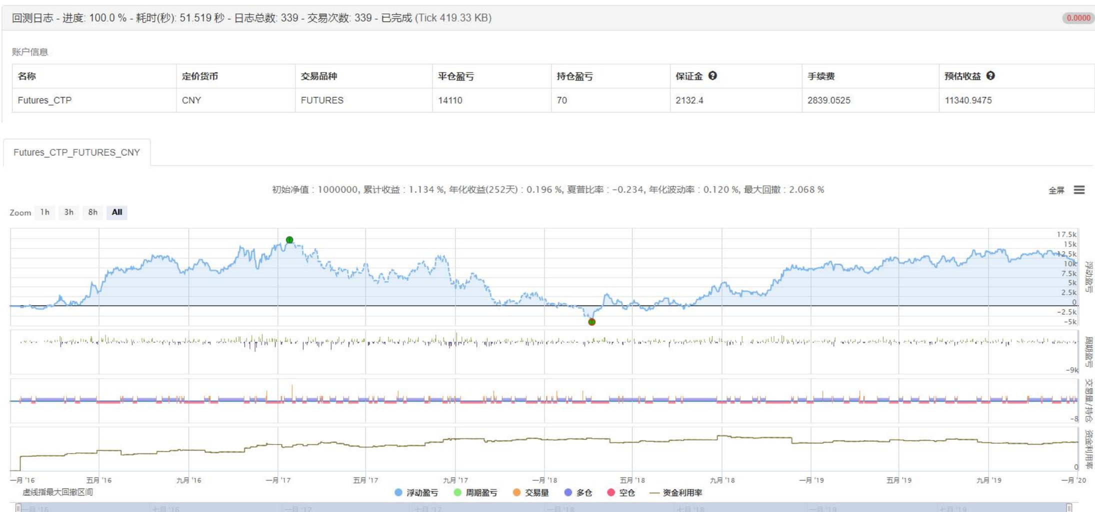

图 6.3　回测绩效

上图是某个回测绩效报告，很多人认为盈利率高、胜率高的模型就是一个好模型，真的是这样吗？让我们来看看衡量策略有哪些重要参考指标。

6.2.2　年化收益率

年化收益率表示投资期限为一年的理论收益率，日收益率、月收益率、季度收益率都可以换算成年收益率。如果一个策略回测的日收益率是 0.01%，那么年化收益率是 3.65%。其计算公式为：(收益 / 本金) / 投资天数 * 250 ×100%

注意：策略回测的年化收益率不是从开始开仓的时候算起，而是从数据开始日期算起。实际上年化收益率代表了策略的盈利效率。另外期货市场其有效的投资时间是一年的交易日，扣除节假日约等于 250 天。

6.2.3　年化波动率

波动率是衡量策略风险的指标之一，它是描述策略资金曲线的涨跌幅程度，是对策略稳健性的衡量，也反映了策略风险水平。其计算方式为：最高价减去最低价的值再除以最低价所得到的比率，年化波动率就是每日波动标准差的年化。

波动率越高，其资金曲线的波动越激烈，策略的稳健性就越低。波动率越低，其资金曲线的波动越平缓，策略的稳健性就越高。

6.2.4　最大回撤比率

除了波动率外，更能直观反映风险的绩效指标就是最大回撤率。它是统计资金曲线任意周期内最高点到最低点时的回撤幅度的最大值。它是描述策略可能出现的最糟糕情况。最大回撤是一个重要的风险指标，对于量化交易而言，该指标甚至比波动率还重要。

6.2.5　夏普比率

很多人喜欢用收益率衡量一个策略，这个无可厚非，因为从投资交易的角度讲，只要赚钱的策略都是好策略。但是请看下图：

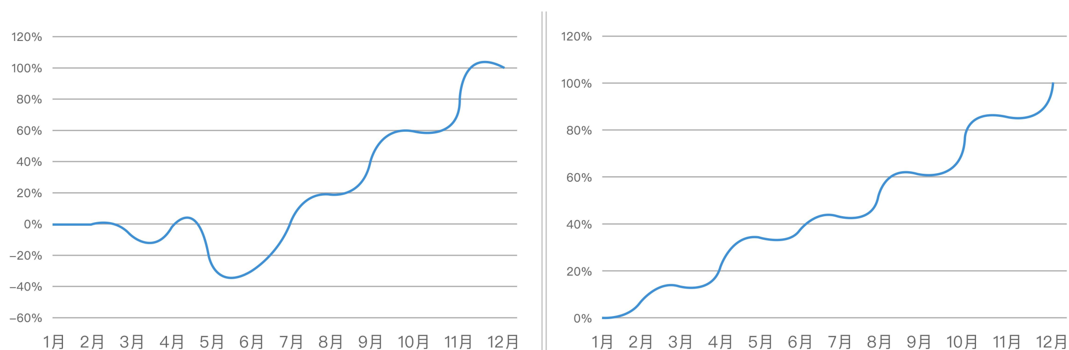

图 6.4　资金曲线（1）

左边的策略收益是 100%，右边的策略收益也是 100%，而左边的策略最大回撤是 50%，右边的策略几乎没有回撤，毫无疑问右边的策略要明显好于左边的策略。所以仅仅用收益率评价一个策略是不科学的。

回撤意味着风险，也意味着波动，正确的方式是将收益率和风险都考虑在内，也就是说不但要考虑收益率，更要考虑每承担每一单位风险所产生的超额收益。夏普比率就是一个对收益和风险综合考虑的指标。其公式为：(策略收益率-无风险利率)/策略收益率的标准差。

举个例子，假如十年期国债收益率是 3%，而策略回测的收益率是 15%，那么超额收益就是 15%-3%=12%，12%除以 6%的策略收益标准差等于 2。这就意味着交易者每承担 10%的风险，能得来 20%的超额收益。

每个策略回测都可以计算夏普比率，如果值为正数，则表示策略收益大于策略波动风险；如果值为负数，则表示策略波动风险大于策略收益。也就是说在设计策略时要考虑风险，尽量用最小的风险换取最大的回报。

6.3　如何避免回测陷阱

量化交易回测虽然可以快速验证策略在历史数据中是否有效，但很多时候回测并不代表未来能盈利，回测看起来非常好的策略，往往实盘表现不佳。因此需要在策略设计过程中规避回测的陷阱，才能让策略回测反映真实的结果。

6.3.1　未来函数

未来函数就是利用了未来的价格，交易策略如果包含未来函数，在实盘运行时回造成信号闪烁的问题。比如有一个策略逻辑时这样的：当收盘价大于开盘价就买入，当收盘价小于开盘价就卖出。这在回测时是没有问题的，因为收盘价是已经完成，固定不变的数据。

注意：在实盘交易中，收盘价只有在收盘的时候才能固定下来，所以程序会把当前的最新价格当作收盘价，这种利用未来价格的策略，会导致买卖信号频繁出现和消失。如果一个策略的买卖点不是固定的，回测的数据也是没有意义的。

如何避免使用未来函数？最简单的办法是使用滞后的价格，可以把这个策略条件修改为：当上根 K 线收盘价大于开盘价就买入，当上根 K 线收盘价小于开盘价就卖出。因为无论是在回测中还是在实盘中，上根 K 线始终是已经完成的，这样就可以保证回测与实盘保持一致。

6.3.2　偷价

相反偷价是利用了已经过去的价格，偷价并不会造成信号频繁出现和消失，但是会造成信号无效。比如有一个策略逻辑是：当收盘价大于开盘价就在开盘时买入，当收盘价小于开盘价就在开盘时卖出。

显然这个策略条件在实盘时是不能成交的，当收盘价出现时，开盘价早就过去了。但是在回测中，程序是会以开盘价买入卖出的，这相当于在原本的资金曲线上叠加了一条斜率为正的直线会造成一种非常夸张的回测资金曲线。

为避免这种情况发生，编写完策略首先要检查策略逻辑，如果策略回测的收益曲线非常平滑，回撤极小，就要警惕了。尤其是策略逻辑存在隐蔽性偷价行为，务必在实盘之前先用仿真交易测试一段时间。

6.3.3　成本冲击

实盘交易中为了保证订单能及时成交，通常需要用对手价或者市价下单，商品期货买一价和卖一价至少相差一个点差，如果是交易不活跃的期货合约就需要更多的点差成本。或者当自己的订单量超过市场现有的订单量时，就会造成自己的订单消耗了市场流动性，触动价格朝向不利于自己的方向移动，使交易成本进一步上升。

不仅如此，手续费、极端行情、软硬件系统、服务器响应、网络延迟都会增加实盘的

交易成本。尤其是交易频率比较高的策略对受市场冲击成本更大，为了让回测更接近实盘环境，折中的办法是在回测时加上固定 2 跳左右的滑点。

6.3.4 幸存者偏差

幸存者偏差是一种逻辑上谬误，意思是没有意识数据筛选的过程，忽略被筛选掉的数据，只通过筛选后的数据得出与实际偏离的结论。如下图中的例子：

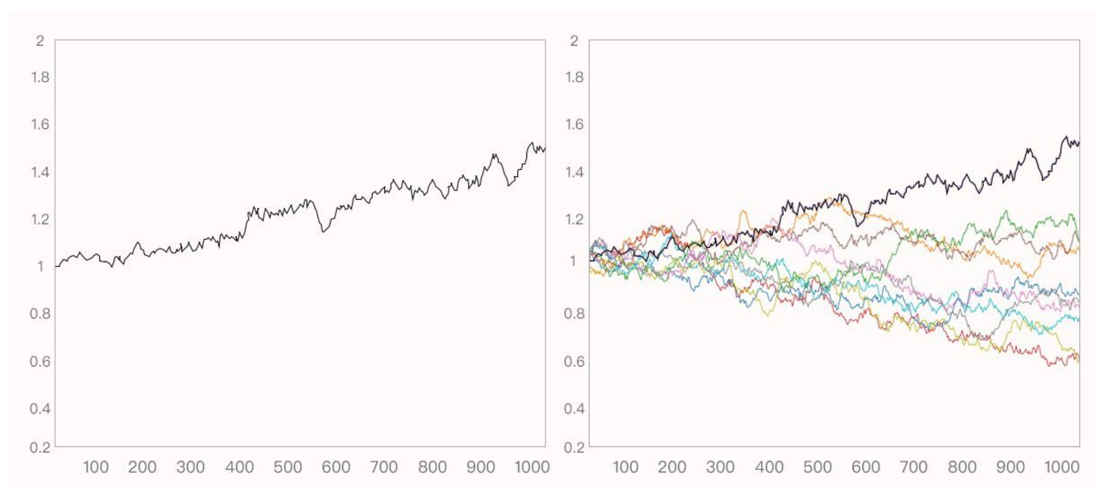

图 6.5 资金曲线（2）

左边的图是一个非常好的交易策略，资金曲线稳稳向上，没有最大回撤。再看看右边这张图，这个资金曲线只是 100 次随机交易回测中表现最好的一个。通过这个例子可以知道，回测也有运气的成分，有时候的回测结果可能这个策略刚好适应了历史数据，再换几个参数或者回测品种就不一定有这么好的结果了。

6.3.5 过拟合

过拟合是统计学中的术语，它是指过于精确地匹配数据特征，以至于无法在其他数据中良好地拟合，量化交易中的过拟合是一种回测时表现很好，实盘中表现较差的现象。如下图中的例子：

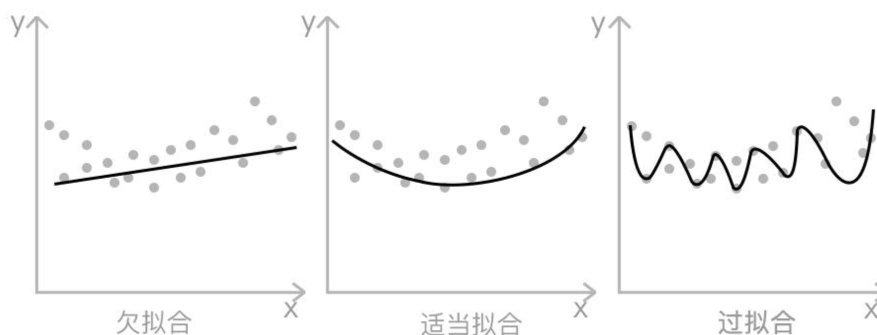

图 6. 6　数据拟合

上图分别是模型欠拟合、适当拟合、过拟合的素描，实际上策略回测应该像第 2 张图那样在数据中匹配普遍规律，而不是像第 3 张图那样试图匹配所有规律，这样才能在新的数据中更好地适应，否则将会导致策略泛化能力下降。

由于商品期货历史数据有限，所以过拟合问题就更加严重，尤其是对于中低频策略来说，几乎不可能完全避免过拟合，但可以利用下面几种方法来减少拟合：

1、减少核心参数

2、简化处理逻辑

3、增加数据样本

4、样本内外测试

如果策略核心参数过多或策略逻辑非常复杂就很容易过拟合历史数据，尤其是当数据样本过少时，不足以让策略获得整个全局特征，如果在样本过少的情况下企图验证策略是否有效，无异于坐井观天，可能会把回测数据自身的特性当成所有潜在样本的共性，这样一来策略再面对实盘时就不适应了，当样本数据足够时，就不会被局部特征所迷惑。

6.4　递进和交叉回测

巴菲特曾经说过：“投资市场里，后视镜永远比挡风玻璃让你看得更清楚。”他的投资哲学是，除非投资标的的“过去”和“未来预期”一样稳定可靠，否则绝不投资。量化交易回测就像是后视镜，如何通过回测判断策略预期，就得使用科学的回测方法。

6.4.1　样本内和样本外回测

期货交易往往当时很难理解，但事后分析起来很简单，那是因为复盘总是站在上帝视角。量化交易回测也同样存在这个问题，回测是站在数据的终点往数据的起点看，那么企图在有限的数据中发现规律，无异于坐井观天。

为了解决这个问题，通常是把数据分成样本内数据和样本外数据。样本内数据相当于上课学习的课本知识，样本外数据相当于课后作业和期末考试。而量化交易中是在样本内

数据上进行策略参数调优，在样本外数据上验证策略是否有效。

数据的划分并没有严格的要求，但至少有几个原则：在数据有限的情况下，通常样本内与样本外的比例是：6:4 或者 7:3；如果数据足够，则这比例可以更宽限一些，可以分为：5:5 甚至 2:8；尤其是逻辑简单的日内短线策略，可以减少样本内数据，更多的分配给样本外数据。

6.4.2 样本递进回测

但是仅凭一次测试就判定策略的好坏显然是不太合理的，因为回测的结果也有运气的成分，有可能保留了不好的策略，也有可能把好的策略筛选掉。那么样本递进回测将是一种更好的回测方式。

样本递进回测是将数据分为多个阶段，每个阶段又分为样本内数据和样本外数据，通过对样本内数据优化得到策略参数，再应用到样本外进行检验，并且不断以递进方式移动样本段，最后将所有样本外的测试报告组合成一个整体的回测绩效报告。

例如：螺纹钢指数大约有 10 年左右的历史数据，可以将数据分为 5 个阶段：

- 第 1 阶段：2010 年~2015 年为样本内数据，2016 年为样本外数据。
- 第 2 阶段：2011 年~2016 年为样本内数据，2017 年为样本外数据。
- 第 3 阶段：2012 年~2017 年为样本内数据，2018 年为样本外数据。
- 第 4 阶段：2013 年~2018 年为样本内数据，2019 年为样本外数据。
- 第 5 阶段：2014 年~2019 年为样本内数据，2020 年为样本外数据。

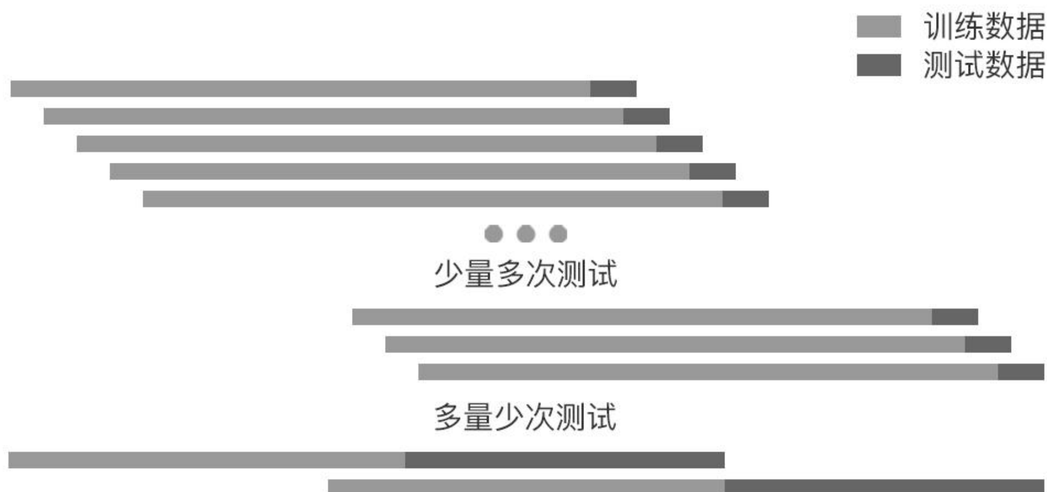

图 6.7 递进回测示例

如上图所示，分别展示了样本递进回测的两种方式：

第 1 种：每次检验时，测试数据比较短，测试次数较多。

第 2 中：每次检验时，测试数据比较长，测试次数较少。

注意：在实际应用中，可以通过改变测试数据的长度，进行多次测试，用来判断策略在应对非平稳数据的稳定性。

6.4.3 样本交叉回测

除了递进回测外，还有一种交叉回测方式。这也是一种动态的回测方式，将数据分为多个阶段，每个阶段又分为样本内数据和样本外数据，通过对样本内数据优化得到策略参数，再应用到样本外进行检验，只不过交叉回测的样本外数据是贯穿整个样本，最后将所有样本外的测试报告组合成一个整体的回测绩效报告。

例如：螺纹钢指数大约有 10 年左右的历史数据，可以将数据分为 5 个阶段：

- 第 1 阶段：2010 年~2011 年为样本外数据，其余为样本内数据。
- 第 2 阶段：2012 年~2013 年为样本外数据，其余为样本内数据。
- 第 3 阶段：2014 年~2015 年为样本外数据，其余为样本内数据。
- 第 4 阶段：2016 年~2017 年为样本外数据，其余为样本内数据。
- 第 5 阶段：2018 年~2019 年为样本外数据，其余为样本内数据。

如下图所示，展示了样本交叉回测原理：

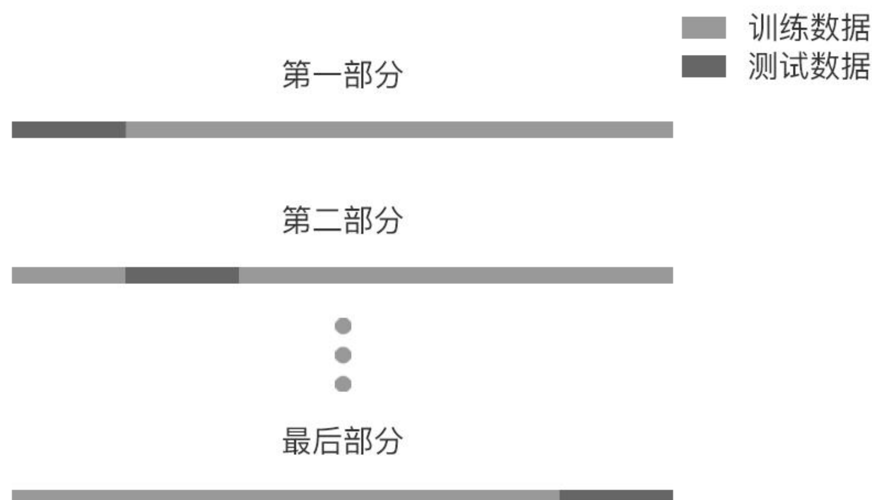

图 6.8 交叉回测示例

交叉回测最大的优点就是充分的利用有限的数据，每个样本内数据同样也是样本外数据。但交叉回测应用时也存在明显的缺点：

1、当价格数据非平稳时，模型的测试结果往往不可靠。例如，用 2008 年的数据做样本内，用 2005 年的数据做样本外。很有可能 2008 年的市场环境与 2005 年相比发生了很大的变化，所以策略测试的结果不可信。

2、与第一条类似，在交叉回测中，如果用最新的数据作为样本内优化策略，而用较老

的数据回测策略，这本身就不符合逻辑。

优秀的交易策略应该能够在未来具有获利性，很多时候直接在全部的历史数据上选择最优参数是非常危险的，如果先利用样本内数据进行参数优化，再利用样本外数据进行样本外测试，除了能客观检测交易策略外，更能有效率节省时间。

量化交易策略本质上就是一个从大量的貌似随机的数据中找寻局部非随机数据的过程，如果不借助统计学的知识，很容易数据陷阱中。如果发现样本外数据表现不好，又觉得丢掉策略太可惜或者不愿意承认这个策略不行，而对着样本外数据继续做优化，直到样本外数据上也表现得一样好，那最后受伤的一定是自己的真金白银。

但即便拥有庞大数据的历史，但面对浩瀚无尽且不可预测的未来，历史就显得极度匮乏。所以基于历史自下而上倒推出来的交易系统，终究会随着时间而沉没。因为历史不能穷尽未来。因此一个完整的正期望交易系统必须由其内在原理或逻辑所支撑。。

6.5 温故知新

学完本章内容，读者需要回答：

1．量化交易回测都有哪些数据？

2．如何避免回测陷阱？

3．递进和交叉回测有哪些区别？

在下一章中，读者会了解到：

1．期货市场三大风险

2．等价鞅和反等价鞅仓位管理

3．极端行情监控

第 7 章　风险管理与投资组合

量化交易最关键的常识就是风险，但对于大多数交易者来说，风险是一个令人不愉快的话题。虽然严格控制风险意味着与暴利绝缘，但对于优秀的量化交易者来说资金和风险管理绝对有必要。

本章主要涉及到的知识点有：

- 学会止损：正确的认识和设置止损。
- 认识风险：认识市场三大风险。
- 资金管理：承担合理风险的前提下获取最大利润。

7.1 认识期货中的风险

大部分初学者热衷于谈交易技术，这个无可厚非。因为技术是基础的东西，初入市场的人最直观可以学到的。技术就像你在撸代码中使用的编程语言、或者库、或者框架，合理的使用技术会初步积累你在市场中的优势。但资金管理也同样重要。

7.2.1 系统性风险

知己知彼百战不殆，资金管理就是管理风险，在讲正确的资金管理之前，请先认识市场的三大风险，这样更利于您对本节课程的理解。风险是我们交易者，最无奈、最不喜欢、最最不愿意提的两个字。

因为风险代表着亏损、亏损代表着失败，总之风险会引起很多不愉快的事情。首先就是系统性风险。要想完全阐述金融市场的风险，是非常复杂的。在这里仅仅举个例子，金融市场的系统性风险跟大海有点类似，大海是高风险环境，时而平静、时而疯狂。

无论是扑鱼还是运输，有一点可以肯定，那就是来获得利润。不下海就得不到我们想要的利润，可是如果下海就有可能被大海吞噬，并且无论是多么周密的计划，这个风险都始终存在，除非你不下海。

你能控制大海吗，还是能计算大海？赢亏同源的道理就是从这里来的。即使放在现实生活中也同样如此。既然注定无法逃避，也就不需要逃避。那么，我们除了在交易系统中，用合理的规则，包容市场的系统性风险，还有别的选择吗？没有，只能包容系统性风险。

7.2.2 人为主观风险

有一句老话，人才是最大风险源，最该小心的那个人就是自己。这话无不在向我们揭露另一种类型的风险，那就是人为主观性风险。人在极端环境下，很难做出客观理性的选择。比如在面对巨大亏损的时候，如果心态失去控制，很容易感性冲动，就像赌红眼的赌徒。

交易做的时间长，不可能不会碰到心态失去控制的情况，而老手一般可以在 10 分钟之内平复心情，因为他们知道，发脾气根本没有任何帮助。可以说一个交易素质好不好，主要看他是如何控制自己、控制认为主观性风险的。

注意：因为系统性风险是不可控制，不可避免的，所以这样的损失是恒定的，并且利润也是恒定的。但这个利润还要减去人为主观性风险所带来的损失。而这才是完美理论在实战中因为人的不同，最终结果严重偏差的根本原因。

7.2.3 策略风险

做交易的人一定知道期货市场是有主力的，主力会刻意误导散户，主力明明要启动行情却偏偏故意先来个大跌，导致我们看不清楚真实的方向，该买的时候卖，该卖的时候买。

这样的行为就是通过策略强加于我们的风险。

理论上赢亏同源，但与系统性风险不同的是，策略风险并没有办法在概率上被衡量，而系统性风险可以被概率统计与衡量。策略性风险并不存在交易多少次就一定会成功多少次，或失败多少次的统计模型，因为它严重的技巧化与因人而异化。

主力背后或许是一个精英团队，水准比散户高多了。并且无论在消息面、基本面分析、还是资金实力，都比广大散户有压倒性的优势，从承担风险类型的区别来看，散户面对主力就已经很吃亏了，所以散户的成功率那么低，老赚不到钱再自然不过了。

7.2.4 资金管理的意义

就如上面所述，期货市场的风险规模大，涉及面广，具有放大性、复杂性与可预防性等特征。期货的风险成因主要有价格频繁波动、保证金交易的杠杆效应、非理性投机及市场机制不健全等等。

这里需要注意的是，你需要对资金管理有个正确的态度。资金管理的功能是控制净值回撤和提高稳定性，不是盈利。通过总仓位比例控制亏损，保证交易能够无限重复。资金管理无法把一个本身是负期望的交易策略，从亏损状态扭转为盈利状态。但是可以避免把一个本身是正期望的交易策略，从盈利状态变成亏损状态。所以资金管理并不是万能的，但它是整个交易系统中必备的一部分。

7.2.5 资金管理的方法

第 1 种：固定百分比资金管理

固定百分比资金管理，是一种非常流行的方法，同时也比较稳健，它的优点是在回撤期，延缓本金的下降速度，而在行情来的时候，又可以加快本金的上涨速度。并且它的方法也非常简单，我建议初学者采用这种资金管理方式，它的计算公式如下：

$$\text{头寸量} = \frac{\text{剩余本金（美元）} \times N\%}{\text{止损距离（点）} \times \text{每一点代表的美元}}$$

其中，N 代表你最大能承受的亏损额度的百分比，止损距离就是开仓点减去止损点。

第 2 种：赢冲输缩

从风险控制的角度来说，“输冲”类的资金管理策略的风险是开放性的，会导致破产风险加速升高，而“输缩”则可以维持破产风险的稳定。

假设我们有 100 元，每次每次只拿总资金的 10%，第一次是 10 元，第二次是 9 元，不管第几次我们都还会有十次以上机会。所以赢冲输缩的策略就可以保证我们在市场活得更久些。

第 3 种：分散式交易

华尔街有句名言：不要把所有的鸡蛋放在同一个篮子里面。多元化资产配置，是唯一的免费午餐。单品种的稳定性和抗风险性极差。并且单个品种的市场容量是有限的，大资金出入会对市场造成很大的冲击，出入场都很难成交在理想价位。品种分散，还可以参与150多个全球市场，从商品，黄金，到货币和股票指数等。除了品种分散，还可以多策略、多参数、多周期组合。从而达到削峰填谷的目的。

第4种：凯利公式

凯利公式是一个特定赌局中，使得拥有正期望值之重复行为长期增长率最大化的公式。公式如下：

$$f^* = \frac{bp-q}{b} = \frac{p(b+1)-1}{b},$$

其中：

- f*为现有资金应进行下次投注的比例；
- b为投注可得的赔率（不含本金）；
- p为获胜率；
- q为落败率，即1 - p；

注意，这个公式的适用范围是反复多次下注的场景 。这是一切赌戏和投资最基本的道理，也就是『没有把握，决不下注』 - 不下注，就不会输。

暴力背后往往背负着风险，资金管理的重要性无论怎么强调都不为过，当然不同的交易者有不同的资金管理方式，如同没有策略圣杯一样，没有一种资金管理方案是放之四海而皆准的，没有一种资金管理是适合所有投资方式的，适合的才是最好的。

7.2 等价鞅资金管理

鞅是概率论中的一种随机过程，也就是说这个随机过程未来的期望值与当前时间点的值相同。在量化交易中鞅代表了收益期望为0，如果某个策略的交易过程是鞅，那么在理想情况下其未来的净值跟当前的净值一样。

7.2.1 什么是马丁格尔

马丁格尔是一种等价鞅式的资金管理方法，英文直译为：martegal，最初是指控制马车的马具。后来马丁格尔代表一种赌博策略。最早可以追溯到十八世纪，历经几百年经久不衰，直到现在还有很多马丁格尔或者类似的策略。

马丁格尔最初应用在轮盘赌博中，逐渐延伸到金融交易中，直到今天在股票、期货、外汇，还能看到马丁格尔的影子。之所以经久不衰，是因为从理论上讲，这是一种永不亏

钱的策略。

7.2.2　正向马丁格尔

这种永不亏钱的秘诀在于，在每次赔钱后，将赌注加倍，而在任何一次赢钱之后，将赌注回归到初始单位。无论在赢钱之前输了多少次，只要概率让下注者赢一次，不但能赢回之前所有的损失，还外加一次赌注的收益。马丁格尔在金融市场中，创造了很多盈利奇迹和滑铁卢的亏损。

以抛硬币为例，出现正反面的概率约等于 50%，连续出现正面或反面的次数，都是以 50%的概率开始递减，也就是说在任何一次抛硬币中，出现正面的概率是 50%，连续 2 次出现正面的概率是 25%，连续 3 次出现正面的概率是 12.5%等等以此类推。

如果最开始赌注是 1 元，连续输钱的赌注以 2 的倍数增加，也就是：1、2、4、8、16、32、64、128、256、512 等等，直到赢钱为止，一个回合才结束，所以每个回合都能赢得 1 元。虽然在纸面上，马丁格尔可以做到永不亏钱，但是随着一连串的损失发生，赌注的规模会呈几何倍的速度增加。

注意：为了避免资金雄厚的赌徒运用这个策略，几乎所有的赌场对每一次赌局有最高的下注限制。

7.2.3　正向马丁格尔代码

```
/*backtest
start: 2020-01-01 00:00:00
end: 2020-01-02 00:00:00
period: 1d
basePeriod: 1d
exchanges: [{"eid":"Futures_CTP","currency":"FUTURES"}]
*/

var chart = {
    __isStock: true,
    tooltip: {
        xDateFormat: '%Y-%m-%d %H:%M:%S, %A'
    },
    title: {
        text: '资金曲线'
    },
    rangeSelector: {
        buttons: [{
            type: 'hour',
            count: 1,
            text: '1h'
        }, {
```

```
            type: 'hour',
            count: 2,
            text: '3h'
        }, {
            type: 'hour',
            count: 8,
            text: '8h'
        }, {
            type: 'all',
            text: 'All'
        }],
        selected: 0,
        inputEnabled: false
    },
    xAxis: {
        type: ''
    },
    yAxis: {
        title: {
            text: ''
        },
        opposite: false,
    },
    series: [{
        name: "",
        id: "",
        data: []
    }]
}; // 画图对象


// 策略入口函数
function main() {
    var ObjChart = Chart(chart);  // 绘图对象
    ObjChart.reset();  // 启动前先清空绘图
    var now = 0  // 随机次数
    var bet = 1
    var maxBet = 0  // 记录最大倍数
    var lost = 0
    var maxLost = 0  // 最大连续亏损次数
    initialFunds = 10000  // 初始资金
    var funds = initialFunds  // 实时资金
    while (true) {
        if (Math.random() > 0.5) { // 胜率为50%
            funds = funds + bet  // 赚钱
            bet = 1 // 每次赚钱后，就把下注倍数重置为1
```

```
            lost = 0
        } else {
            funds = funds - bet // 赔钱
            bet = bet * 2 // 失败就把下注倍数翻倍
            lost++
        }
        if (bet > maxBet) {
            maxBet = bet  // 计算最大倍数
        }
        if (lost > maxLost) {
            maxLost = lost  // 计算连续亏损次数
        }
        ObjChart.add([0, [now, funds]])  // 添加画图数据
        ObjChart.update(chart)  // 画图
        now++  // 随机次数加 1
        if (funds < 0) {  // 如果破产结束程序
            Log("初始资金：" + initialFunds)
            Log("随机次数：" + now)
            Log("最大连续亏损次数：" + maxLost)
            Log("最大倍数：" + maxBet)
            Log("最终资金：" + funds)
            return
        }
    }
}
```

输出结果为：

```
初始资金：10000
随机次数：511660
最大连续亏损次数：19
最大倍数：524288
最终资金：-257889
```

图 7.1　正向马丁格尔曲线

7.2.4　反向马丁格尔

与正向马丁格尔相反，反向马丁格尔则是每次赢的时候，将赌注加倍，直到出现赔钱的时候将赌注回归到初始单位。这是马丁格尔策略的延伸，理论上这种策略更适合用在趋势行情中，因为顺势而为的操作有很高的成功率。成功率的提高伴随的是逐步加仓获取的超额收益。

7.2.5　反向马丁格尔代码

```
/*backtest
start: 2020-01-01 00:00:00
end: 2020-01-02 00:00:00
period: 1d
basePeriod: 1d
exchanges: [{"eid":"Futures_CTP","currency":"FUTURES"}]
*/

var chart = {
    __isStock: true,
    tooltip: {
        xDateFormat: '%Y-%m-%d %H:%M:%S, %A'
    },
    title: {
        text: '资金曲线'
```

```
    },
    rangeSelector: {
        buttons: [{
            type: 'hour',
            count: 1,
            text: '1h'
        }, {
            type: 'hour',
            count: 2,
            text: '3h'
        }, {
            type: 'hour',
            count: 8,
            text: '8h'
        }, {
            type: 'all',
            text: 'All'
        }],
        selected: 0,
        inputEnabled: false
    },
    xAxis: {
        type: ''
    },
    yAxis: {
        title: {
            text: ''
        },
        opposite: false,
    },
    series: [{
        name: "",
        id: "",
        data: []
    }]
}; // 画图对象


// 策略入口函数
function main() {
    var ObjChart = Chart(chart);  // 绘图对象
    ObjChart.reset();  // 启动前先清空绘图
    var now = 0  // 随机次数
    var bet = 1
    var maxBet = 0  // 记录最大倍数
    var lost = 0
```

```
    var maxLost = 0  // 最大连续亏损次数
    initialFunds = 10000  // 初始资金
    var funds = initialFunds  // 实时资金
    while (true) {
        if (Math.random() > 0.5) { // 胜率为 50%
            funds = funds + bet  // 赚钱
            bet = bet * 2 // 赚钱就把下注倍数翻倍
            lost = 0
        } else {
            funds = funds - bet // 赔钱
            bet = 1 // 每次赔钱后，就把下注倍数重置为 1
            lost++
        }
        if (bet > maxBet) {
            maxBet = bet  // 计算最大倍数
        }
        if (lost > maxLost) {
            maxLost = lost  // 计算连续亏损次数
        }
        ObjChart.add([0, [now, funds]])  // 添加画图数据
        ObjChart.update(chart)  // 画图
        now++  // 随机次数加 1
        if (funds < 0) {  // 如果破产结束程序
            Log("初始资金：" + initialFunds)
            Log("随机次数：" + now)
            Log("最大连续亏损次数：" + maxLost)
            Log("最大倍数：" + maxBet)
            Log("最终资金：" + funds)
            return
        }
    }
}
```

输出结果为：

```
初始资金：10000
随机次数：19984
最大连续亏损次数：13
最大倍数：2048
最终资金：-1
```

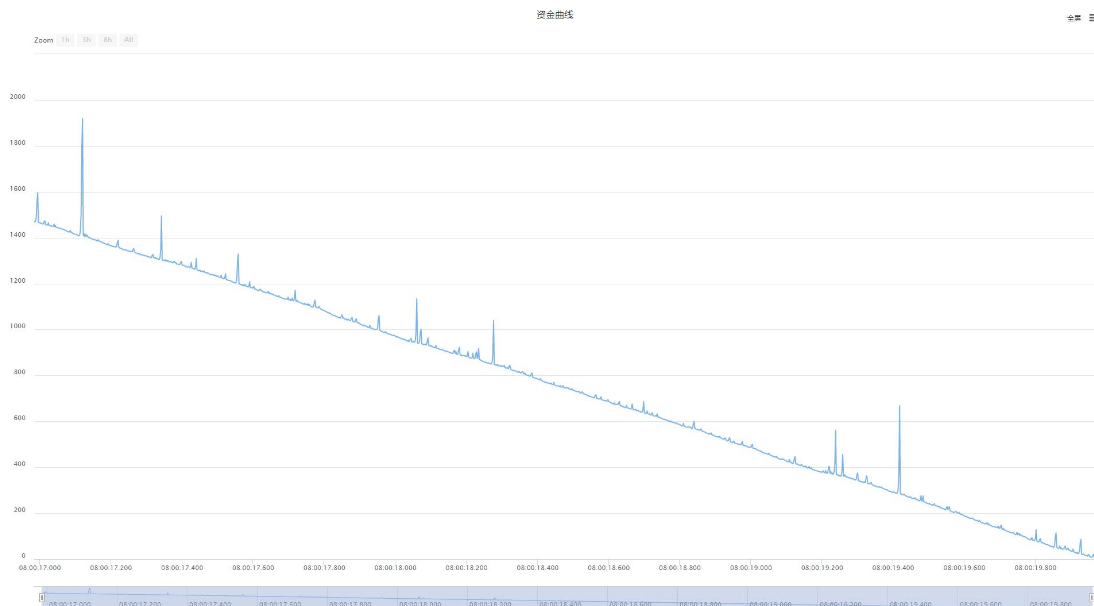

图 7.2 反向马丁格尔曲线

7.3.6 马丁格尔在期货市场上的应用

虽然在期货市场并没有最高下单量的限制，但与赌场不同的是，期货的涨跌并不是完全随机赌大小，真实的金融交易市场要比赌场更加复杂。如果将马丁格尔策略用在期货交易中，一旦市场按照反方向趋势行情运行，后面随着行情的发展，头寸翻倍增加会越来越大，风险也随之加大。那么对于想要使用马丁格尔策略用于期货市场的交易者来说，至少需要解决 3 个问题：

- ❑ 起始仓位
- ❑ 加仓倍数
- ❑ 加仓距离

注意：起始仓位需要根据你的资金量而定，也就是在交易之前计算好资金能承受的最大连续亏损次数。如果起始仓位过高，会导致每次翻倍加仓后投入过大的资金量。另外加仓倍数太高也会导致同样的问题，马丁格尔默认是双倍加仓，如果设置成 3 倍加仓，破产的速度会更快，但如果设置成 1.5 倍加仓，就会出现另一种结果。最后需要考虑的是加仓的距离，比如在 5000 点开多单，价格下跌 15 点加仓，和价格下跌 30 点加仓，也是不一样的。这点完全取决于交易者的风险承受能力和交易习惯偏好。

7.3 反等价鞅资金管理

等价鞅策略的特点是赚钱的时候赚一点，赔钱的时候赔很多，这种资金管理方式顺应了人性，人有厌恶风险，不愿承认错误的共性，在交易中则表现为，有一点盈利就获利为

安，在亏损时却紧握头寸不肯认亏。而反等价鞅的资金管理方法则是反人性的体现。

7.3.1 什么是凯利公式

反等价鞅典型的代表就是凯利公式，该公式最早应用于赌博中，在一个正期望的独立的重复赌局中，凯利公式可以计算出每次下注额度，实现本金最快速增长。如果是一个负期望的赌局，凯利公式则给出不赌即赢的结论。

假设总共有 100 元资金，有一个抛硬币的赌局，如果硬币出现正面得 2 元，如果出现反面赔 1 元，可以使用任意金额进行投注，可以每一次只押注 1 元，也可以一次将 100 元全部押注。

很明显这是一个正期望的赌局，硬币只有正反两面，出现正反两面的概率均为 50%，即胜率为 50%，但每一次赢钱的时候赢 2 元，赔钱的时候赔 1 元，即赔率为 2。虽然这是一个正期望的赌局，但每一次只押注 1 元，或者一次将 100 元全部押注肯定不是最优解。

如果一次将 100 元全部押注，则有 50%的概率赔钱的可能，这个赌局也就破产终止了。如果每一次只押注 1 元，虽然不会有破产的风险，但会错失很多机会和利润，导致资金增长效率很低。凯利公式的出现，神奇地解决了每次投注多少金额可以达到利润最大化。

7.3.2 凯利公式计算方式

凯利公式的计算方式非常简单，公式为：f=（bp-(1-p))/b，其中：

- f：最佳投注比例
- p：胜率
- b：赔率

将上面抛硬币的例子代入这个公式：f=(2*0.5-(1-0.5))/2=0.25，也就是说每次投注使用资金的 25%比例，可以达到利润最大化。接下来用 Python 写一个策略来验证一下。

7.3.3 用数据验证凯利公式

```
import random

 chart = {
    "title": {
        "text": '资金曲线'
    },
    "rangeSelector": {
        "buttons": [{
            "type": 'all',
            "text": 'All'
        }],
        "selected": 0,
        "inputEnabled": false
```

```
    },
    "yAxis": {
        "title": {
            "text": '资金曲线'
        },
        "opposite": false,
    },
    "series": [{
        "name": "10%头寸",
        "id": "",
        "data": []
    }, {
        "name": "25%头寸",
        "id": "",
        "data": []
    }, {
        "name": "50%头寸",
        "id": "",
        "data": []
    }]
};                                              # 画图对象

def main():
    now = 0                                     # 模拟下注次数
    ObjChart = Chart(chart);                    # 绘图对象
    ObjChart.reset();                           # 启动前先清空绘图
    funds1 = 100                                # 初始资金
    funds2 = 100                                # 初始资金
    funds3 = 100                                # 初始资金
    while True:
        betRatio1 = funds1 * 0.10               # 10%比例下注
        betRatio2 = funds2 * 0.25               # 25%比例下注
        betRatio3 = funds3 * 0.50               # 50%比例下注
        if random.random() > 0.5:               # 胜率为 50%
            funds1 = funds1 + betRatio1 * 2     # 赔率为 2
            funds2 = funds2 + betRatio2 * 2     # 赔率为 2
            funds3 = funds3 + betRatio3 * 2     # 赔率为 2
        else:
            funds1 = funds1 - betRatio1 * 1     # 赔率为 2
            funds2 = funds2 - betRatio2 * 1     # 赔率为 2
            funds3 = funds3 - betRatio3 * 1     # 赔率为 2
        ObjChart.add(0, [now, funds1])          # 添加画图数据
        ObjChart.add(1, [now, funds2])          # 添加画图数据
        ObjChart.add(2, [now, funds3])          # 添加画图数据
        ObjChart.update(chart)                  # 画图
        now = now + 1                           # 下注次数
```

```
        if now > 1000:
            return                                    # 模拟下注 1000 次
```

上面的代码使用 random 模块中的 random 随机方法，模拟了抛硬币的例子，当值大于 0.5 时表示硬币为正面，当值小于等于 0.5 时表示硬币为反面。在胜率为 50%，赔率为 2 的情况下，分别用 10%、25%、50%比例下注，观测它们的资金曲线。

输出结果为：

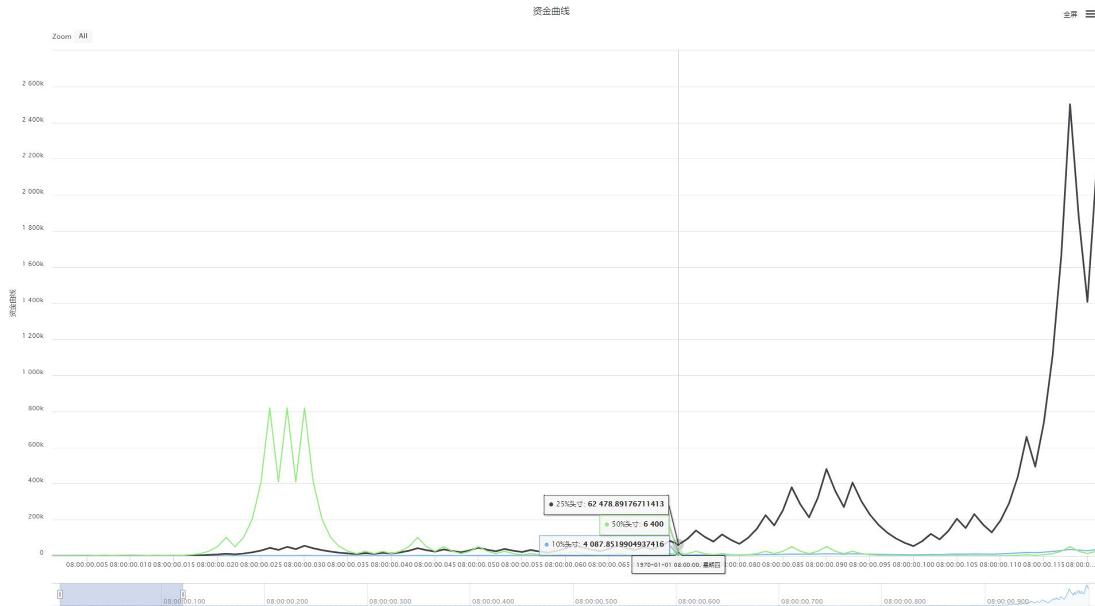

图 7.3　不同的下注比例结果

如上图所示，蓝色的线为 10%的头寸，黑色的线为 25%的头寸，绿色的线为 50%的头寸。经过 1000 次的下注，整体来看使用 25%的头寸盈利最佳。注意看图片左边，虽然 50%的头寸在连续正面时资金增长很快，但是也承担了更大的风险在连续反面时资金下跌也很快。

下面的表格列出了一部分在不同胜率和赔率组合下，最佳的头寸比例：

胜率	赔率	期望值	最佳下注比例
0.25	0.75	负期望	-0.5
0.5	0.5	零期望	0
0.5	0.5	正期望	0.25
0.6	0.4	正期望	0.2
0.4	0.6	正期望	0.25

从上面的表格中，可以看出：

- 只有期望值为正的时候，游戏才可以进行下去。
- 在胜率相同的情况下，赔率越大越好。
- 在赔率相同的情况下，胜率越大越好。

7.3.4 凯利公式在量化交易中的应用

在量化交易中，通过策略回测可以得出胜率和赔率的关键数据，可以计算出最佳的头寸比例。比如一个 CTA 趋势策略胜率为 38%，赔率（盈亏比）为 2.8，那么通过凯利公式计算：f=(0.38*2.8-(1-0.38))/2.8≈0.15，即每次使用 15%，最终可以获得最大化收益。

7.3.5 凯利公式的局限性

但是凯利公式运用前提是有顺序、无关联性的赌局。在抛硬币游戏中，每次结果都是一个完全独立的事件，当前的结果并不会受到上次结果的影响。对于赌博来说或许是一个好模型，但并不一定适应于投资领域。

在赌博中盈亏是随机的，但在金融市场中，盈亏并不是完全随机，价格不仅会受到其本身价值的认同度、外在宏观事件的影响，还会受到理性和非理性的影响。也就是说价格在这些因素影响下，上涨或下跌可能发生的概率有多大难以量化计算。

并且同时面对多个交易品种。每个品种的赢亏幅度也不同，出现的涨跌的概率也可能不同，每个品种的关联度也不同（当 A 上涨，B 也可能跟着上涨）。

注意：理论上凯利公式汲取“日取其半，万世不竭”的道理，在资金可以无限拆小的情况下，可以永远不会破产，达到效率最大化。但是在非线性的市场中，凯利公式的意义远不如赌博中那么大。更多时候通过对凯利公式的学习，反馈了资金管理的重要性。

7.4 构建投资组合和风险控制

有人曾经做过一个研究，长期来看，整个投资收益中来自择时部分的还不足 5%，剩下约 90% 的投资收益都是来自于成功的投资组合，这个人就是“全球资产配置之父”加里 •布林森。

7.4.1 投资分散与均衡

一个好的投资组合有助于家族财富的传承，在财富的更替洗牌中，那些历经百年依然屹立不倒的家族，绝非偶然。比如洛克菲勒家族、摩根家族、罗斯柴尔德家族等等。基本都是靠某个行业发家，而后代则是靠均衡的资产配置，使这些财富长期保持增长。

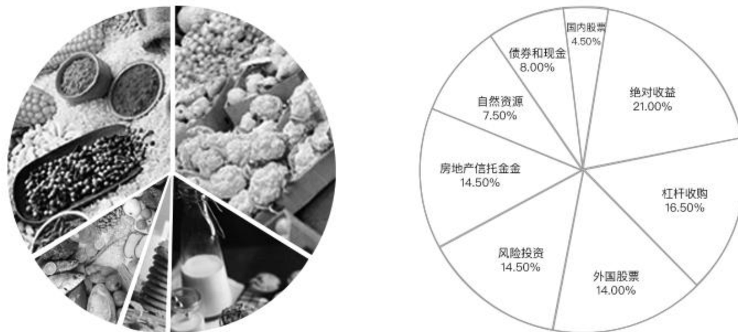

图 7.4　均衡资产配置

在量化交易中，一个好的投资组合可以在跌宕起伏的金融市场中起到“削峰填谷”的作用。均衡的资产配置就跟合理的搭配膳食一样。营养学界流传着这样一种说法：没有不好的食物，只有不合理的膳食。

但是在投资领域“资产配置”这个词一直饱受争议。有人说，不能把所有的鸡蛋放在同一个篮子里；也有人说，与其鸡蛋放在多个篮子里，不如放在一个篮子里，然后看好这个篮子。

特别是中国过去十几年快速发展，导致资产价格升值过快，在房价暴涨的背景下，只需要买房，就能轻易的跑赢通货膨胀，均衡的资产配置显的可有可无。但现在不一样了，国内经济增速明显下降，资产价格持续回落，加上全球货币宽松，资产变化轮动加快。特别是在高速通胀环境下，单一的资产配置的时代已经过去。

7.4.2　投资组合分类

商品期货只是量化交易投资组合中的一部分，其中还包括：基金、债券、股票等等，从收益的角度看：基金<债券<股票<期货。从风险的角度看：基金<债券<股票<期货。这些都可以用于投资组合。

除了多品种组合外，也可以将不同的策略组合到一起，比如：CTA 趋势策略、CTA 震荡策略、跨期套利策略等等。在实际应用中，不同周期的数据也可以组合到一块，比如：15 分钟、1 小时、2 小时、日线数据组合到一块。

7.4.3　构建投资组合

通过将上述这些不完全相关的资产加入到组合中，就可以有效降低系统性风险，同时收益最大化；也可以在总体回报率不变的情况，大概率降低亏损的概率，波动（风险）最小化。

- ❑　鸡蛋不要放在同一个篮子里（分散）
- ❑　篮子也不要放在同一个地方（分散）

- ❑ 不要一次性把鸡蛋都放进去（定投）
- ❑ 也不要在篮子里只是放鸡蛋（多元）

并且最大程度分散风险，稳健增值。即使在不利的市场环境中，也能具备足够的防御性。另外，投资是反人性的，科学的投资组合，可以稍微顺从一些人性，提高投资的容错率。

在投资组合中，一定要先解决保障性配置，也就是保险。它是整个资产配置的前提和基础保障。另外配置低风险、低收益的货币基金，兼顾日常消费和周转资金的流动性。其次配置一些用来跑赢 GDP 的投资品种，确保整个投资组合有好的收益基础。低风险的政府债券、企业债券、债券基金、黄金、房地产等是较好的选择。

如果想要跑赢“印钞机”的速度（M2），就需要配置一些高风险、高收益的资产，比如：个股、ETF、股票型基金、混合型基金；或者风险更高的美股、港股、商品期货、外汇现货等。最后，用很少的一部分钱，去博胜率小赔率大的机会。比如：项目股权、数字货币。

7.4.4 收益与风险

投资组合没有固定的模式和比例，一切取决于自己的风险偏好、收益预期、投资期限，将资金在不同资产类别之间合理分配。从收益风险比的角度可以划分为：保守型、稳健型、激进型。

注意：如果你对风险的承受能力较弱，就需要多配置些第 1 条和第 2 条；如果你对收益要求比较高，就需要多配置第 2 条和第 3 条。

但这并非一成不变，而是因人而异，并根据资本市场行情动态调整。通常股市与债券的相关系数大约为 0.6，如果股市比较低迷，那么债券市场相对比较火爆。比如，2008 年股市低迷的时候，债券市场风生水起。

客观的讲，摆在投资者面前的工具，并不只有投资组合，还有品种选择和市场择时，但对于大多数人而言，真正有用的也就只有投资组合，而且往往还能兵不血刃，出奇制胜。

但是人们往往醉心于在品种选择与市场择时中找圣杯，如果这个市场真的有规律，那么这些规律迟早会被智商很高、又很努力的人发现，他将赚到市场中绝大多数钱，直到垄断整个市场。可事实呢？华尔街迄今已经几百年了。

总之，投资是一种极大的不确定性行为，任何风险会在任何时候，都可能一触即发，所以投资组合绝对不能缺席，因为在崩盘的时候会让你稍微好过一些，而且在最应该买入的时候，也能产生足够的现金。

7.5 温故知新

学完本章内容，读者需要回答：

1. 期货市场有哪些风险？

2．正反马丁格尔策略有什么区别？
3．如何计算凯利公式？
4．如何构建投资组合？
在下一章中，读者会了解到：
1．正确的设置止损
2．量化交易与基本面
3．常用的数理知识

第 8 章　交易技巧及交易理念

华罗庚在谈到读书方法的时候，提到“读书是由薄到厚再由厚到薄的过程”。交易也类似，刚开始什么都不懂自然就很“薄”，等到入门发现需要学习内容太多就感到很“厚”，随着日积月累，理解了交易的核心，升华到交易理念时就变的很“薄”。

8.1　常用止盈止损方法

你有没有过这样的经历？当价格朝开仓的反方向移动时，你可能认为是小幅度回调。继续移动时，你可能认为调整这么多了，价格也该反转了吧。但是如果行情继续如此反复下去，一般人能够忍受这样快速且长时间的回撤。这时“ 止损 ”也许是多数人心中所想并去执行的。

8.1.1　止损的成本

止损是有成本的，然而，当你止损之后，接着就出现了一段非常暴力又迅速的行情。回头一看。止损止到了低谷。结果明明是赚钱的单子，却以止损提前离场，还白白损失了前几次的试错成本。

面对这种情况，不仅资金缩水，更会有一种被左右愚弄的痛苦。特别连续出现几次这样的情况，对自信心的打击是难以承受的。以至于怀疑自己，为了止损却错失行情而感到不值，久而久之就不再止损。

曾经有家机构做过统计：在止损后，80%多的情况下，价格会重返原点附近。换句话说80%的止损是错误的，不止损的话有 80%概率不仅不会亏损，甚至还会小赚。如果把周期拉的足够大，大多数股票、债券、商品、外汇，价格都在一个大的范围内上下波动，毕竟趋势行情的占比还不足三分之一。

那么，在行情中间，无论多，还是空，大部分时候，我们的单子即使被套，只要不止损，总能抗回来。甚至还能获得一些利润。嗯......表面上是这么个理，既然如此，为什么要

傻乎乎止损呢。

8.1.2　止损的意义

假如没有止损，却真的碰到反向的大行情。那资金遭到重创，回本岂不更加困难了？为此我做了一个图表：

亏损	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
盈利	11%	25%	43%	67%	100%	150%	233%	400%	900%	永别了

图 8.1　亏损与回本比例图

如上图，如果在必要的时候没有割肉止损，而是心存侥幸抱着，期待价格能回归。结果呢，20 年了还没回归。看完以上，你是不是明白了交易一定要：坚决止损。你下次回归到成本的难度，取决于你上次是否保护了好你的本金。而赚多少取决于市场，亏多少几乎全部取决于自己。

虽然 80%的止损都是错误的，但是为了避免 80%的错误，不去执行止损，则在剩余的 20%的行情中，巧合遇到单边大行情或大调整、大反弹，又是反向，就容易死亡。 所以，为了 20%的小概率的爆仓或大幅亏损的可能，我们却要去选择 80%的大概率的止损的错误，这是没有办法的事情。因此，在大部分时候，我们止损，并不是我们方向错了，而是出于控制损失的需要。

在市场里，鳄鱼法则就是：当你发现自己的交易背离了市场的方向，必须立即止损，不得有任何延误，不得存有任何侥幸。鳄鱼吃人听起来太残酷，但市场其实就是一个残酷的地方，每天都有人被它吞没或黯然消失。

8.1.3　如何止损

紧接着，就面一个问题：如何止损？亏多少才止损？是固定数额？还是固定百分比？还是动态止损？我认为止损是对于结果和演进路径都不确定事物的利用方法，帮助我们取得好的一面，舍弃坏的一面。正确的使用止损需要建立新的未来观。接下来看下各种止损的方法：

第 1 个：价格止损

```
if 现价 > (1 + X%) * 开仓价:
    平仓止盈
elif 现价 < (1 - Y%) * 开仓价:
    平仓止损
else:
    继续持仓
```

将买入价或持仓均价，设置为止损价， 一旦价格上涨大于 X% 或下跌大于 Y% 就卖出。这是也是一个固定止损/止盈价位的止损方案。但固定止损存在一些弊端，因为止损标准和行情本身没有太大关联，所以很有可能出现刚刚离场行情就出现反转的情况。

第 2 个：时间止损

```
if 持仓时间 > X天 and 涨幅 < Y%:
    平仓止损
else:
    继续持仓
```

时间也是有价值的，如果在一定的时间内的回报收益低于一个预设值就认为该交易低于预期，选择卖出。 这是一个非常简单的止损策略， 由于止损线是固定的， 所以不能很好的减少回撤。

第 3 个：移动止损

```
X = 允许的最大回撤
if 现价 < 持仓周期内最高价 * (1 - X%):
    平仓止损
else:
    继续持仓
```

相比较而言，移动止损更为客观。其考虑的是价格的回撤， 若价格回撤大于某预设值 X% 就将其卖出。移动止损的止损价会随着最高价的创新高而变化。这一系列变化着的止损方式就是跟踪止损了。

注意：因为止损标准和行情发展有着密切的关系和逻辑，同时也能覆盖止盈策略，所以也是很多老手常用的方式之一。但由于止损方式是结合行情来具体决定的，所以不像固定止损那样可以把每一笔交易的亏损控制在一定数额之内。这也就要求进场点的选择要风险更小、更为准确。

第 4 个：阶梯止损

```
X = fx(开仓后最高价，初始止损价，阶梯长度，阶梯变化率)
if 现价 < X:
    平仓止损
else:
    继续持仓
```

阶梯止损是另一种移动止损方法。止损价会根据持股周期内最高价的变化而变化。与上述移动止损不同的是，其止损价的计算方式略有不同。

第 5 个：时间+阶梯止损

```
X = fx(持仓周期，期望回报率)
if 现价 < X:
    平仓止损
else:
    继续持仓
```

时间+阶梯止损是将“ 时间有价值 ”和“ 移动止损 ”这两个思路结合在一起的策略。 止损价会随着持股周期的变化而变化，一旦跌破止损价，则卖出。由于这种止损方法兼容了一些主观的成分，所以最好可以和其他方法结合起来使用，这样才能保证止损上可以做到“ 软硬兼施 ”。

8.1.4 止损的本质

止损只是必要条件，而不是充分条件。它只是整个交易系统中的一个分支。前提是有

一个正期望的交易系统，否则止损只是让你死的慢些。交易中八成的止损都是因为杂乱无序的波段造成的。

人生和投机一样，大部分都是自己打败了自己。止损要止得惊天地，泣鬼神才是成功的止损，否则频繁的止损只会让人一点点步入失败的泥沼。可以这样说，止损本质上是对市场的敬畏、对不确定性的承认、对市场的尊重。

会不会买只是我们能赚多少的因素之一，而会不会止损却是我们能亏多少的全部因素。未来是不确定的，错误的持仓，未必就是错误的方向。止损虽不能决定市场，但是却能界定你面对怎样的市场。

如果把交易比作生活，市场中所有的价格都是合理的，就跟生活一样，存在即为合理。就如同我们无法在生活中决定什么应该存在一样无法决定市场的价格应该是什么。但我们依然可以决定我们能在生活中做什么，在市场中做什么。在生活中的底线，就如同在市场中的止损一样。

8.2　量化交易与基本面数据

在期货交易市场，争论技术分析和基本面分析谁好谁坏，从来就没有停止过。技术分析者认为价格已经包含了一切，相信未来价格还以趋势方式演变，只关心图表上价格行为本身的变化，判断它能卖多少钱。基本面分析者认为真正价值最终将会反映在价格上，并不需要关心短期价格走势，更多的是分析影响价格背后的因素，判断它值多少钱。

8.3.1　常用的基本面数据

影响期货价格的基本面因素非常多，其中包括：宏观因素、品种因素、其他因素。往细了分多达几十项，并且这些数据是在不停变化的。

- 宏观因素：宏观政策、产业政策、政治因素、外汇汇率、经济周期、货币政策
- 品种因素：升水贴水、供需关系、商品库存、产业利润
- 其他因素：季节因素、天气因素、新闻事件、市场情绪

想要综合分析这些因素合力后会产生的结果，首先获取与之关联的数据必须是全面的、准确的，否则只会得出片面的结果。但要对于个人交易者来说，要想准确获取这些庞大的数据，似乎是力所不及的事情，那么有没有合适散户的基本面分析法呢？

8.3.2　基本面分析铁三角

根据奥卡姆剃刀原则，期货的基本面分析只需要抓住核心要素，就能从错综复杂的信息中找出规律。宏观经济数据复杂多变，每天每时每刻，地球上有太多的经济数据公布，各国政界、央行、投行，官方的和非官方的。除了政治和经济危机外，宏观分析是聊天的好材料，实用性不大。另外在季节因素、天气因素、新闻事件、市场情绪中，很多都是突

发事件，本身就是无法分析预测的。所以散户只需要把精力放在品种上即可。

期货是现货未来的价格，理论上期货到期时的价格应该约等于现货价格，如果期货价格高于现货价格就称为期货升水，如果期货价格低于现货价格就称为期货贴水。因为期货和现货的价格都是公开的数据，可以计算出升水还是贴水，从而判断期货未来的大概价值。

注意：相同的品种，在现货市场与期货市场的价格差，叫做基差。无论是期货升水还是期货贴水，随着交割日期的临近，期货价格和现货价格都会趋于一致，有时候是现货向期货回归，有时候是期货向现货回归，更多时候是期货和现货双向回归。

影响商品现货价格的因素虽然有很多，但最终大都体现在供需关系上。如果买者多于卖者，价格就会上涨；如果卖者多于买者，价格就会下跌。国内的商品期货大致上可以分为：农产品和工业品。期货圈子中流传着这样一句话："农产品看供给，工业品看需求。"农产品是刚需，需求是相对稳定的，决定价格主要看供给；工业品是下游需求带动的，再者国内基本都产能过剩，决定价格主要看需求。

虽然，在实际操作中我们很难获取工业品的需求数据，也很难计算出农产品的供给数据。但是价格波动依存于供给与需求的相互作用，这种相互作用的结果就是库存。如果库存处于低位，说明市场供不应求，需求的力量大于供给的力量，未来价格看涨；如果库存处于高位，说明市场供大于求，供给的力量大于需求的力量，未来价格看跌。

所谓的仓单就是交易所的交割仓库入库现货后开具的标准仓单，它反映的是交易所公布的库存数量。当期货价格较高时，现货商就是注册仓单然后在市场上销售，所以根据这个原理我们可以反推出在期货中的交易方向。

- 期货多头：如果仓单大量减少，说明期货价格低于现货价格，应该做多。
- 期货空头：如果仓单大量增加，说明期货价格高于现货价格，应该做空。

另外，还可以利用仓单来判断库存。仓单既可以注册也可以注销，当期货主力想要价格上涨时，会把持有的注册仓单注销掉，改变交易所公布的库存数量，来达到交割货物不足的假象，进而影响期货价格上涨的预期。当期货主力想要价格下跌时，会注册仓单，造成交割货物增多的假象，使得被动影响期货价格下跌。

到这里，基本面分析三大因素：库存、基差、仓单就已经凑齐了，有些做基本面分析的朋友可能还会加上产业利润、技术分析等等，增加窥视市场的维度，理论上两者相加是大于二的，因为能知道越多信息，越多的角度去观察市场，才能做出更好的决策。那么基本面交易策略可以为以下条件：

- 多头：贴水 + 低库存 + 仓单减少
- 空头：升水 + 高库存 + 仓单增加

8.3.3 获取基本面数据

```
import requests
from bs4 import BeautifulSoup
import time
import datetime
import json
```

```
diff_data = reserve_data = receipt_data = 0

def to_timestamp(date_str):
    time_array = time.strptime(date_str + " 00:00:00", "%Y-%m-%d %H:%M:%S")
    return int(round(time.mktime(time_array) * 1000))

def date_arr(year, month, day):
    begin, end = datetime.date(year, month, day), datetime.date.today()
    arr = []
    for i in range((end - begin).days + 1):
        day = begin + datetime.timedelta(days=i)
        arr.append([str(day).replace('-', ''), str(day),
                    day.weekday() + 1, to_timestamp(str(day))])
    return arr

def spot_futures_diff_data(date, futures_name):
    global diff_data
    url = f"http://www.100ppi.com/sf2/day-{date}.html"
    try:
        url_text = requests.get(url).text
    except BaseException:
        return int(diff_data)
    soup = BeautifulSoup(url_text, "html5lib")
    if len(soup.select("#fdata")) > 0:
        results = soup.select("#fdata")[0]
        for i in results.find_all('tr'):
            if len(i.find_all('td', text=futures_name)) > 0:
                data = i.find_all('font')[0].text
                if data is not None:
                    diff_data = data
    return int(diff_data)

def spot_data(date, futures_name, url_type, types):
    global reserve_data, receipt_data
    data_type = reserve_data if types == 'WHSTOCKS' else receipt_data
    url = f'http://www.shfe.com.cn/data/dailydata/{date}{url_type}.dat'
    try:
        url_text = requests.get(url).text
    except BaseException:
        return data_type
    total = count = 0
    if url_text[0] == '{':
        for i in json.loads(url_text)['o_cursor']:
            if futures_name in i['VARNAME']:
                if '合计' not in i['WHABBRNAME']:
```

```
                if '总计' not in i['WHABBRNAME']:
                    try:
                        inventory = int(i[types])
                    except BaseException:
                        return data_type
                    if inventory > 0:
                        total, count = total + inventory, count + 1
        if count > 0:
            data_type = int(total / count)
    return data_type

def main():
    cfgA = {
        "extension": {"layout": 'single', "col": 4, "height": "500px"},
        "title": {"text": "基差图表"},
        "series": [{"name": "基差", "data": []}]
    }
    cfgB = {
        "extension": {"layout": 'single', "col": 4, "height": "500px"},
        "title": {"text": "库存图表"},
        "series": [{"name": "库存", "data": []}]
    }
    cfgC = {
        "extension": {"layout": 'single', "col": 4, "height": "500px"},
        "title": {"text": "仓单图表"},
        "series": [{"name": "仓单", "data": []}]
    }
    LogReset()
    chart = Chart([cfgA, cfgB, cfgC])
    chart.reset()
    for i in date_arr(2018, 1, 1):
        diff = spot_futures_diff_data(i[1], '天然橡胶')
        reserve = spot_data(i[0], '天然橡胶', 'weeklystock', 'WHSTOCKS')
        receipt = spot_data(i[0], '天然橡胶', 'dailystock', 'WRTWGHTS')
        if diff != 0 and reserve != 0 and receipt != 0:
            chart.add(0, [i[3], diff])
            chart.add(1, [i[3], reserve])
            chart.add(2, [i[3], receipt])
            chart.update([cfgA, cfgB, cfgC])
            time.sleep(1)
            Log(f'基差：{diff} 库存：{reserve} 仓单：{receipt} 日期：{i[1]}')
```

8.3.4 绘制基本面数据图表

上面的代码分别从生意社和上海期货交易所网站分别获取了：基差、库存、仓单等数

据，然后利用 Python 的 highcharts 库把这些数据绘制出来。输出结果如下：

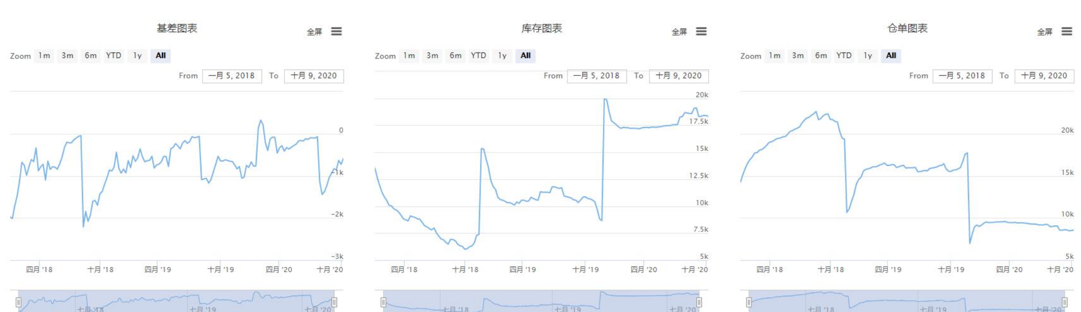

图 8.2　基本面数据图

基本面分析和技术分析并不存在孰优孰劣，它们探究的是同一个市场，只是站的角度不同。没有人可以仅凭一个角度分析，就能窥视市场全部。我相信两者相加是大于二的，因为能知道越多信息，越多的角度去观察市场，才能做出更好的决策。

8.3　交易中的常用的数理知识

交易是一门艺术，事关对经济的分析、政策的判断、人性的理解；又是一门严谨的科学，事关随机微积分、概率统计、优化理论。交易中的数理知识千差万别，难以一概而论。常见的有以均价为基准的 VWAP；通过固定时间间隔执行的 TWAP； 趋势跟随的 momentum trader 等等。如果你自己编一个根据 MACD，RSI 什么的产生指标的东西，也可以勉强称为算法交易。

8.4.1　VWAP 算法

VWAP 是把大额委托单在间隔时间段内拆分成小的委托单并分批交易的算法，来达到最终买入或卖出成交均价尽量接近这段时间内整个市场成交均价的目的。它是量化交易系统中常用的一个基准，其意义是减少自身大额订单对市场的冲击，最小化交易成本。计算公式为：

$$Vwap = \frac{\sum_{i=1}^{n} price_i * volume_i}{\sum_{i=1}^{n} volume_i}$$

VWAP 算法根据历史成交量预测出未来的成交量，并结合总成交量和拆单时间段等条件，把大额委托单分割成许多小的委托单，在指定的时间段逐步发送出去。VWAP 算法改善了订单的执行效率，提高了大额订单的隐蔽性。

8.4.2 TWAP 算法

与 VWAP 不同的是，TWAP 算法是把大额委托单平均地分配到交易时段上，并在每个时间段上提交拆分后的订单。比如将一天的交易时间分为 N 段，大额委托单也平均分配在 N 个时间段上，TWAP 算法会在每个时间段内执行委托交易。其公式为：

$$\text{Twap} = \frac{\sum_{i=1}^{n} \text{price}_i}{n}$$

TWAP 并不考虑成交量的因素，而是根据交易时段的平均价格，从而达到减小交易成本的目的。在分时成交量无法准确估计的情况下，该模型可以较好地实现算法交易的基本目的。

注意：当委托的订单过大时，即使采用 TWAP 算法将订单拆分到每个时间段内，也会出现下单量较大的可能，当市场流动性不足时，依然有可能对市场造成冲击。

8.4.3 布朗运动

布朗运动是一个连续的随机过程，很多科学领域都涉及到了布朗运动，尤其是普遍运用于不可预测随机游走的金融市场。布朗运动是一种非常简单的连续随机过程，它是描述期货价格随机运行的模型。

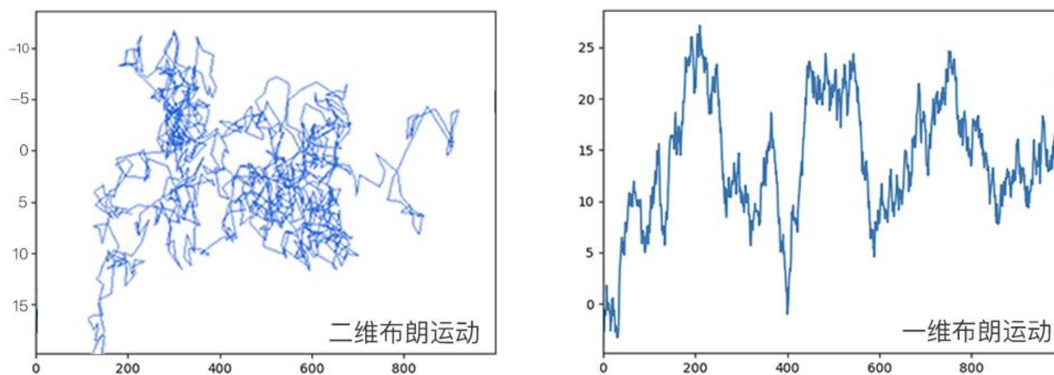

图 8.3　二维布朗运动和一维布朗运动

尽管影响股票价格涨跌的原因是无穷无尽的，但价格的运动并非是“ 完全随机游走 ”。而是每个因素的影响力通常被反馈力牵制（索罗斯的反身性），市场不但有正反馈机制，还有负反馈机制。

正因如此，很多情况下，价格会有各种正负反馈机制并存，导致正态分布建模的前提不再成立。所以说，价格是一个带着“ 漂移 ”的布朗运动。

8.4.4 维纳过程

在数学中，维纳过程是一种连续时间随机过程。又与物理学中的布朗运动有密切关系。金融数学中，维纳过程可以用于描述期权定价模型。

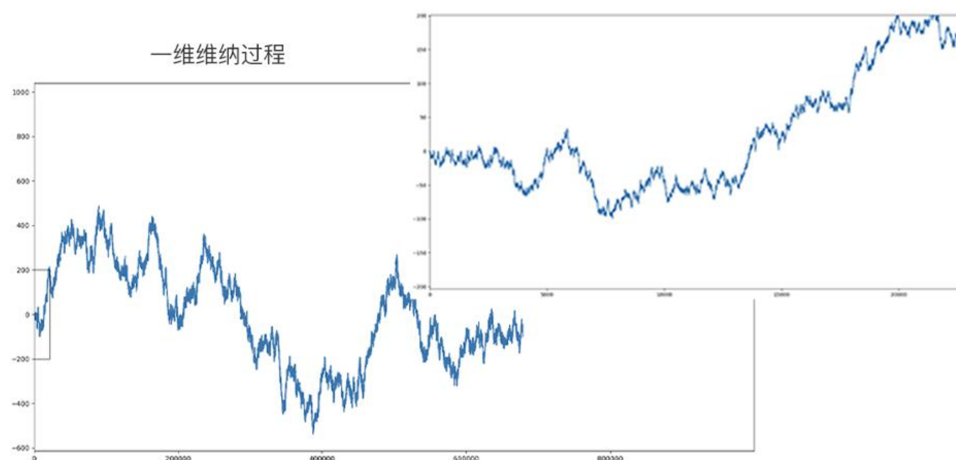

图 8.4 一维维纳过程

如果一个马尔可夫过程中，增量的概率分布服从于一个关于时间 t 的正态分布，我们就说这个过程是维纳过程，或者说布朗运动。表示成这个样子：

$$\Delta x \sim N(a\Delta t, b^2\Delta t)$$

维纳过程本质上是伊藤过程的一个特殊形式，它是包含在伊藤过程这个概念里面的。维纳过程可以用随机漫步或任意拥有平稳独立增量的离散随机过程的尺度极限来构造。这个构造方法基于 Donsker 定理。

8.4.5 伊藤引理

在金融学中，如果说布朗运动是描述价格随机波动的基本模型，那么伊藤引理则提供了随机过程函数的微分框架。尤其对于期权定价有非常重要的意义，通过伊藤引理可以得到期权价格的随机微分方程，从而得到期权价格模型。

以抛硬币为例，假如有一枚正面和反面一样重的硬币，正面朝上赢 1 元，反面朝上输 1 元。当抛了 N 次（次数足够多），截取第 N-1 次抛硬币所有情况的结果，就会发现结果总是符合正态分布。

抛硬币都是一个独立的事件，每次结果都跟上一次或者下一次以及其他任何一次的结果无关，也就是说从布朗运动得到了一个维纳过程。如果把价格分解为：预期收益和波动率两个部分。如果预期收益率和波动率是确定的，就可以用随机变化来表示价格的变化。

期权的 BS 公式就是一个很好的例子。

8.4.6 马尔科夫过程

在概率论及统计学中，马尔可夫过程是一个具备了马尔可夫性质的随机过程。马尔可夫过程是不具备记忆特质的。换言之，马尔可夫过程的条件概率仅仅与系统的当前状态相关，而与它的过去历史或未来状态，都是独立、不相关的。

注意：它的时点前和时点后的取值是相互独立的——也就是说，下一分钟发生的事情，完全不受历史时期的变动所控制，只和现在的状态值有关。

这样的一个无记忆性的过程给了我们一个事实上的优势——我们在做未来的预测的时候，完全可以不用去看历史价格，而只关注当前价格。

由于这样预测的数据具有不确定性，所以预测结果必然也就是一个概率分布的形式。假设豆粕在时间 n 的价格为 Sn，对于下一个时点 n+1 而言，其价格 Sn+1 的条件概率并不取决于时点 n 之前的历史价格，即：

$$P(S_{n+1} = x | S_1 = x_1, S_2 = x_2, \dots) = P(S_{n+1} = x | S_n)$$

这样 S1，S2，S3，...，Sn，... 是一个马尔科夫过程。其中 xi 是一个状态价格，其取值的范围叫做状态空间。当然连续的马尔科夫过程和连续随机变量一样：

$$P(S_{n+1} \leq x | S_1 = x_1, S_2 = x_2, \dots) = P(S_{n+1} \leq x | S_n)$$

8.4 建立概率思维，提升交易格局

期货有很多种交易方法，无论是价值投资、技术分析、事件热点、套利对冲等，表面上看起来逻辑严谨，但实际上往往相互矛盾。价值投资的优势是可以根据价值给价格波动划分一个安全边际，技术分析的优势是三大假设使交易具有一定的科学性。

但它们都有一个共同的特点，那就是对未来的价格分析，只能做到大概预测，而不能精准预测。即使将基本面分析与技术分析相结合，也不能解决提高“精准”的问题，所以自始至终交易都是一个概率游戏。

8.5.1 交易来自生活

不仅如此，人这一生，小到过马路，交什么样的朋友；大到从事什么样的事业，跟什么人结婚等等，都是评估风险与回报的概率游戏。因为我们没有未卜先知的能力，每做一件事即使再有把握，风险都始终存在，无法做到百分之百是对的。

许多人在交易中犯错的重要原因就是缺少概率思维，在做交易时过于感性而非理性。

感性其实就是我们的原始本能，在市场中，这些原始本能可以激发人的许多弱点，并且成倍放大。

8.5.2 概率思维

概率思维，是一个文绉绉的名字，说得通俗点就是——赌博思维。“赌博”可能是被人们误解最深的词汇之一了。如果你的策略是负期望，就是赌徒；如果你的策略是正期望，就是赌博。

如果把“赌博”中的贬义去掉，将之理解为承担一定风险而获得一定回报的活动，那么人生真的处处是赌博。上学选择哪个专业、买不买房、项目上不上马、打工还是创业等等。甚至把钱存到银行也是赌博，因为你不确定未来是否会通货膨胀，银行是否会破产（参考希腊债务危机）。总之从摇篮到坟墓，生命的每个过程都是在赌博。

8.5.3 久赌必赢

在研究久赌必赢策略之前，先来研究一下，那些久赌必赢策略的原理。除了印钞机，还有什么是能久赌必赢的呢？

赌场里面的：百家乐、轮盘赌、老虎机、21 点等等，不管怎样变换形式，都隐藏着一个赌场从来不说的秘密：就是在 1 比 1 的赔率下，庄家的胜率总是大于 50%；另外，还有一种赔率很高胜率很低的赌博产品。就这样，赌场老板利用「大数定律」，持续不断的参与这样的游戏，那么长期一定是盈利的。

举个例子，三个骰子，押大小，4-10 是小，11-17 是大，押对了就赢钱。而骰宝有一种围骰，就是三个骰子点数相同，赌场庄家通杀，围骰出现的概率是 2.8%。那么出现大和出现小的概率就各是 48.6%。赌场就是靠这 2.8%的概率，如果每个赌客每局都押 100 元，玩 100 局赌场就会赢 280 元。

- (0.486+0.028)*100*100-0.486*100*100=280

但是这个赌场策略是有漏洞的，万一一个大玩家心血来潮押个几百亿，恰好又赢了，赌场就一下子破产了。所以，赌场会设置一个下注上限，本轮超过这个上限就不能再下注了，并且一次一结账。这样就算赌客可能一时运气好赢钱了，长期下去，还是会输给概率，在无限多次的骰宝游戏中，赌客就会输掉 2.8%的钱。

赌场老板的优势仅仅比赌客多 2%，在单次赌博中，老板可能是亏损的，甚至也可能遇到连续亏损。但是赌场老板并不会被亏损吓坏，因为他知道，自己之所以能赚钱，正是「大数定律」在其中起作用，只要有人继续在赌，只需要 2%的微弱优势，就能长期稳定盈利下去。

类似的久赌必赢的例子还有：各种彩票。彩票的奖池资金，自上市以来是越积越多，这些钱当然来自于广大彩民。你知道双色球中 500 万的几率是多少吗？答案是 1770 万分之一，这是一种高赔率低胜率的赌博。

8.5.4　概率的变化

假设有一个正反面一样重的硬币，抛出字（背面）和花（正面）的概率都是 50%，而且每次抛硬币与前次结果无关。连续地抛 10000 次这个硬币，那么出现正面的概率约等于 50%。

但是如果只抛 10 次，则出现正面的概率就变了，这个概率就不一定是 50%了。所以赌场庄家必须保证触发这个正期望策略的次数足够多，这个正期望的策略才有效。这也是私募机构在开启量化交易策略时，除非特殊条件，不能停止策略的原因。

8.5.5　交易中的大数定律

理解了赌场必赚的原因后，就可以把这些经验学过来，在期货市场“经营”属于自己的“赌场”。

1、在技术面或基本面，只做有胜率优势的交易。

2、长期来看胜率很难超过 50%，所以策略的赔率越大越好。

3、赌场每次投注都有上限，做交易也要设置止损点。

4、赌场每次投注都是独立的事件，在交易中也一样，当次交易应该于上次交易无关。

5、交易次数一定足够多，才能发挥大数定律的优势，能多品种多策略最好。

注意：赌场不会因为某次或多次亏损而修改它的规则，量化交易也是如此。

如果交易策略是已经验证过的正期望策略，那么单笔的盈亏没有意义，甚至连续几次的交易盈亏的意义也不大，只要交易的次数足够多，最后的结果一定是赚钱的。大数定律对具有概率思维的交易者非常有帮助，因为知道赢钱是必然的结果，就不会对中间的浮盈浮亏所困扰，从而提高的策略的执行率。

8.5　温故知新

学完本章内容，读者需要回答：

1．常用的止损方法有哪些？

2．基本面分析有哪些核心数据？

3．如何在交易中久赌必赢？