Python与量化投资

从基础到实战

主编：王小川 等

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书主要讲解如何利用 Python 进行量化投资，包括对数据的获取、整理、分析挖掘、信号构建、策略构建、回测、策略分析等。本书也是利用 Python 进行数据分析的指南，有大量的关于数据处理分析的应用，并将重点介绍如何高效地利用 Python 解决投资策略问题。本书分为 Python 基础和量化投资两大部分：Python 基础部分主要讲解 Python 软件的基础、各个重要模块及如何解决常见的数据分析问题；量化投资部分在 Python 基础部分的基础上，讲解如何使用优矿（uqer.io）回测平台实现主流策略及高级定制策略等。

本书可作为专业金融从业者进行量化投资的工具书，也可作为金融领域的入门参考书。在本书中有大量的 Python 代码、Python 量化策略的实现代码等，尤其是对于量化策略的实现代码，读者可直接自行修改并获得策略的历史回测结果，甚至可将代码直接实盘应用，进行投资。

图书在版编目（CIP）数据

Python 与量化投资：从基础到实战 / 王小川等主编. —北京：电子工业出版社，2018.4
ISBN 978-7-121-33857-1

Ⅰ. ①P… Ⅱ. ①王… Ⅲ. ①软件工具－程序设计－应用－投资 Ⅳ. ①F830.59-39

中国版本图书馆 CIP 数据核字（2018）第 049565 号

策划编辑：张国霞
责任编辑：徐津平
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：26.5　字数：550 千字
版　　次：2018 年 4 月第 1 版
印　　次：2018 年 4 月第 1 次印刷
印　　数：4000 册　定价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

推荐序一

很荣幸收到王小川博士的邀请，为其新书《Python 与量化投资：从基础到实战》作序。王小川博士是华创证券研究所非常出色的分析师，在日常工作中非常乐于分享他的开发经验和心得。在本书出版之前，他已经出版了两本关于 MATLAB 的畅销书，我相信这一本介绍使用 Python 进行量化投资的新书，会推动相关领域的发展。

在过去的几年中，在很多领域内基于创新类算法的应用场景和相关产品不断涌现，IT 的推动作用已经从自动化延展到了智能化。在开源的大氛围下，算法的更新迭代速度不断加快，并在各个领域渗透和融合，专业化程度越来越高。

在金融领域的量化投资、智能投顾、信用评级、新闻监控、舆情分析等多个方向上，目前已经大量使用了相关技术和算法，并且融合的程度在不断加深。与其他领域相比，金融领域的算法应用有其自身的特点：一是信息的来源多、部分数据非结构化；二是在不同的应用场景甚至策略之间，所适用算法的差异较大，例如投资交易的量化策略、智能投顾中的用户画像、新闻处理中的自然语言处理和大数据，都涉及了不同大类的算法；三是投资中各个影响因素之间的逻辑关系复杂化和模糊化；此外，很多金融问题不是单目标优化的，也不是封闭的信息集。

展望未来，在金融科技的落地方向上，量化投资、大数据的 Quantamental、精准画像、自然语言处理等依然会是焦点，势必吸引越来越多的关注及资源。量化投资和 Python 这两个词是当下的焦点，王小川博士平时的工作正是其交汇点。正如书名《Python 与量化投资：从基础到实战》所表达的，本书包含了王小川博士在工作中的宝贵经验；在案例中描述的示例，正是本研究所金融工程的很多重要研究方向，例如常用的行业轮动、市场中性策略、多因子策略、CTA 策略、期权策略、时间序列等。所以，本书对于了解量化开发的运用现状及掌握必备的开发能力而言，是非常有益的。

考虑到众多读者可能没有 Python 基础，本书从零开始介绍 Python 语言，并且由浅入

深、循序渐进。值得一提的是，与目前市场上的量化投资类图书不同，本书的最大特点是接地气、实用性强，并开源了全部的策略代码，读者可以自行运行和修改。

本书还设置了读者互动网站，对于广大投资者提出的关于本书的疑问，可以在第一时间做出解答。本书可以帮助大家更好地了解量化、掌握方法及提升量化投资的能力，非常值得大家细读。

华中炜

华创证券执委会委员、副总经理兼研究所所长

推荐序二

互联网时代的量化投资：科技让量化投资和智能投资更普及

科技一直是推动投资行业变革的重要力量，它的发展和应用催生了量化投资的新模式。量化投资利用科学的方法认识市场波动，通过实证方法验证投资假设，通过组合优化生成 Alpha 交易，可有效地控制风险暴露，高效覆盖大量的投资机会，并提高投资的效率。

自量化投资的开山之作 *Beat the market：A Scientific Stock Market System* 出版以来，量化投资便在全球范围内快速发展，涌现出指数基金、对冲基金、SmartBeta 和 Fund of Funds（FOF）等量化创新产品。量化投资改变了全球资产管理格局，成为主流的投资方法，其管理规模也在快速增长。目前，全球最大的资产管理公司和对冲基金都是基于量化和指数投资的机构。

量化投资行业的蓬勃发展吸引了众多年轻人投入其中，但因其门槛高、专业性强，只有大型投资机构才有能力提供量化研究和投资平台，普通大众没有机会利用专业的量化平台进行研究和投资，也缺乏系统性的量化投资培训教材，这成为制约行业发展的主要问题。因此，在 2015 年，通联数据推出开放的量化投资平台——优矿，让普通大众也能够拥有华尔街专业机构的量化装备，让量化投资变得更加容易。借助 Python 科学计算的能力和海量的金融大数据，在优矿平台上可以快速进行统计推断、因子分析、信号研究、资产定价、事件研究、机器学习、深度学习等量化研究工作。优矿已成长为大型的专业量化平台，为行业的发展培养了很多优秀人才。

优矿的部分特色如下。

◎ 海量的金融大数据：提供各类资产和财务数据、因子、主题、宏观行业特色大数据和量化场景 PIT 数据，保障在量化过程中不引入未来数据。

- 多资产回测框架：提供股票、期货、指数、场内外基金等多资产多策略回测和丰富的衍生工具，保证多因子策略、事件驱动等的快速实现。
- 优矿的风险模型：接轨国际化风险模型算法，采用优质原始数据，提供 10 种风格因子和 28 种行业因子，全面揭示市场行业风险。
- 量化因子库：提供 400 多种量化因子库，除了提供了传统的投资因子，还提供了特色 Alpha 因子如分析师评级、分析师赢利预测等。

《Python 与量化投资：从基础到实战》是华创证券研究所量化团队联合通联数据优矿团队的力作，在很大程度上填补了量化投资培训教材的空白，在本书中循序渐进地讲解了量化投资的思想和策略，并借助 Python 语言帮助读者从零开始进行量化投资实战。

本书适用于有一定数理及编程基础的人员阅读，如果读者能够静下心来，踏踏实实地学习和思考，去理解量化投资的本质和逻辑，就会发现本书蕴藏的宝贵价值。

展望未来，科技的发展也将推动量化投资升级换代。在传统的量化投资中，交易策略是被事先编程的静态模型，其局限性在于策略在一个时期内的效果非常好，但在市场环境发生变化之后就可能效果不佳。机器学习等人工智能技术的应用推动了量化投资进入新时代，智能机器会在市场的发展和变化中观察到市场的异常，交易策略也会随着市场的变化而变化。

量化投资的另一个新趋势是与基本面投资相结合。我们可以用机器帮助我们学习、归纳和总结基本面投资的分析方法和经验，最后形成一套可重复的研究模型。这就是将量化和基本面结合起来，形成“量本投资”的新范式。在未来，无论是做量化还是做基本面的投资者，都应该向中间地带去跨界，去探索。也希望本书的读者们都能够将投资知识和前沿科技融会贯通、学以致用，共同推动中国量化投资行业的发展。

王政

通联数据创始人兼首席执行官

前　言

为什么写作本书

作为投资者，我们常听到的一句话是“不要把鸡蛋放入同一个篮子中”，可见分散投资可以降低风险，但如何选择不同的篮子、每个篮子放多少鸡蛋，便是见仁见智的事情了，量化投资就是解决这些问题的一种工具。

而 Python 在 1991 年诞生，目前已成为非常受欢迎的动态编程语言，由于拥有海量的库，所以 Python 在各个领域都有广泛应用，在量化投资界采用 Python 进行科学计算、量化投资的势头也越来越猛。目前各种在线策略编程平台都支持 Python 语言，例如优矿、米筐、聚宽等，这也是我们选择 Python 进行量化投资的原因。

目前市场上关于 Python 与量化投资的图书不少，但仔细研究后不难发现，很多图书都是顶着量化投资的噱头在讲 Python 的语言基础，其能提供的策略有限，并且大部分不提供回测平台，此类书籍中的策略往往为涨停股票可以买入、跌停股票可以卖出、停牌也可以交易，等等，这大大违背了 A 股市场的交易规则，难以获得准确的回测结果。

鉴于以上情形，为了更好地推动量化投资在中国的普及与发展，我们编写了《Python 与量化投资：从基础到实战》一书，本书兼顾了 Python 语言与量化策略的编写，既可以为不懂 Python 语言的读者提供零基础入门，也可以为有 Python 基础的读者提供量化策略建模参考。细心的读者不难发现，本书量化投资策略部分的介绍篇幅远大于 Python 语言的介绍篇幅，这也可看出我们出版本书的初心。

如何使用本书

如果您从未接触过 Python 或者任何其他编程语言，则建议您从第 1 章开始看起，对

Python 基础编程稍做了解；如果您已经是 Python 的忠实用户，则可以从第 4 章开始看起，直接使用优矿平台完成对策略的编写。关于 Python 基础部分的内容，您可自行安装、运行 Python 进行学习；关于量化投资部分的内容，您需要用到优矿在线量化平台，不安装 Python 也可以运行。

本书的配套代码可以在 http://books.hcquant.com 下载。

Python 基础部分的示例代码的后缀名为.ipynb，是 Jupyter Notebook 文件，可以直接用 Python 打开运行；量化投资部分的示例代码的后缀名为.nb，需要上传到优矿的 Notebook 运行。

本书讲了什么

本书分为两大部分，共有 7 章，前 3 章为 Python 基础部分，可以帮助读者快速上手 Python；后 4 章为量化投资部分，借助通联数据优矿平台进行数据处理与策略建立，将各种策略代码直接开源，并且对各种策略进行了介绍与点评，可谓本书的精华部分。

第 1 章为准备工作，主要介绍 Python 的安装与常用的库，尤其是在量化投资领域会使用到的数据分析库。

第 2 章介绍 Python 的基础操作，为后续讲解 Python 量化投资做准备，等于从零开始讲解，可在短时间内快速上手 Python 编程。

第 3 章讲解 Python 的进阶内容，在第 2 章的基础上详细介绍 NumPy、Pandas、SciPy、Seaborn、Scikit-Learn、SQLAlchemy 等经典库，是对前两章的升华和应用。

第 4 章讲解常用金融数据的获取与整理，包括数据整合、数据过滤、数据探索与清洗、数据转化，等等。

第 5 章介绍通联数据回测平台，内容涉及回测平台函数参数介绍、股票/期货模板实例讲解、回测结果分析、风险评价指标与回测细节的注意事项。

第 6 章讲解常见的量化策略及其实现，内容涉及行业轮动、市场中性 Alpha、大师类策略、CTA 策略、Smart Beta、技术指标类策略、资产配置、时间序列分析、组合优化器、期权策略等。代码全部公开，您可在短时间内使用我们的策略模板编写适合自己的策略。

第 7 章给出了 10 道自问自答题目，可便于您在短时间内更好地了解量化投资，希望对您做投资有所帮助。

读者支持及反馈

本书提供了在线“有问必答”服务，可以扫描下面的二维码填写相关信息，成为本书的认证读者，在优矿社区中发帖提问，我们会安排专门人员进行答疑，在第一时间解决关于本书的一切问题。

我们随时欢迎您反馈信息，请告诉我们您对本书的评价，以及您喜欢本书的地方和不喜欢本书的地方。您的反馈对我们来说非常重要，这会帮助我们在后续的修订中完善现有的内容。请将您的反馈意见通过邮件 service@hcquant.com 发送给我们，并在邮件的标题中注明“读者反馈”字样。同时，您可以在优矿社区进行反馈（https://uqer.io/community/）。

感谢

本书在写作过程中得到了华创证券与通联数据优矿团队的大力支持，同时感谢陈杰、卢威、刘昺轶、秦玄晋、苏博、徐晟刚、王镇平、符哲君、彭亮在写作中的大力支持，我们将与您砥砺前行，共同见证中国量化投资的成长。

勘误

虽然我们已经尽力确保本书内容的准确性，但是错误仍可能出现，如果您发现本书的任何错误（文字或者代码错误），则请及时告知我们，我们将非常感激，并在修订时列出您的名字以表示感谢。您可以通过以下三种途径将本书中的错误告知我们。

- 在 books.hcquant.com 网站上发布本书勘误。

◎ 在优矿社区中直接发帖。

◎ 发送邮件至 service@hcquant.com。

互联网上的盗版著作权侵害是在所有媒体中持续存在的问题，如果您在互联网上发现了对我们的著作有任何形式的非法复制，则请立刻告诉我们其地址或者网站的名字，以便我们进行补救。请通过 service@hcquant.com 联系我们，我们将非常感谢您的帮助。

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

◎ **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。

◎ **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

◎ **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/33857*

目　录

第 1 章

准备工作

1.1 Python 的安装与设置

Python 虽然好用，但是想用好它却不容易，其中，让人比较头疼的就是包管理及 Python 的不同版本的问题，特别是在使用 Windows 的时候。为了解决这些问题，出现了不少发行版的 Python，比如 WinPython、Anaconda 等，这些发行版将 Python 和许多常用的 Package 打包，以方便人们直接使用，还提供了 virtualenv、pyenv 等工具管理虚拟环境。

笔者尝试了很多类似的发行版，最终选择了 Anaconda。Anaconda 是一个用于科学计算的 Python 发行版，支持 Linux、Mac、Windows 系统，提供了强大的包管理与环境管理的功能，可以很方便地解决多版本的 Python 并存、切换及各种第三方包安装的问题。Anaconda 通过工具或命令 conda 进行包管理和环境管理，并且包含了与 Python 相关的配套工具。这里说一下 conda 的设计理念，conda 将几乎所有工具、第三方包都当作 Package，甚至包括 Python 和 conda 自身。因此，conda 打破了包管理与环境管理的约束，能非常方便地安装各种版本的 Python 及各种 Package，并方便地切换。

从 Anaconda 官网 https://www.continuum.io/downloads 下载 Anaconda，在下载后直接

按照说明安装即可。在安装时会发现有两个不同版本的 Anaconda，分别对应 Python 2 和 Python 3。其实，安装哪个版本并不重要，因为通过环境管理，我们可以很方便地切换运行时的 Python 版本。注意，尽量按照 Anaconda 默认的行为安装，即不使用 root 权限，仅为个人安装，将安装目录设置在个人主目录下（对于 Windows 就无所谓了），这样做的好处是：在同一台机器上的不同用户完全可以安装、配置自己的 Anaconda，不会互相影响。

本书附带的代码采用 Python 2 编写，但是读者可以使用 Python 3 运行，不会遇到代码方面的麻烦。

1.2 常见的 Python 库

本节为刚接触 Python 的读者简单介绍常见的 Python 库。

1. NumPy

NumPy 是 Python 的一种开源的数值计算扩展，可用来存储和处理大型矩阵，比 Python 自身的列表结构要高效得多。NumPy 底层使用 BLAS 作为向量，各种运算的速度也得到大幅提升。它主要包括：

- 强大的 N 维数组对象 Array；
- 比较成熟的（广播）函数库；
- 用于整合 C、C++和 Fortran 代码的工具包；
- 实用的线性代数、傅里叶变换和随机数生成函数，使 NumPy 和稀疏矩阵运算包 SciPy 的配合使用更加方便。

另外，NumPy 中的数据类型在 Pandas、Scikit-Learn、StatsModels 等库中被作为基本数据类型使用。

2. Pandas

Python 之所以能成为强力的数据分析工具，和 Pandas 库有很大的关系。Pandas 的主要应用环境如下：

（1）数据的导入与导出；

（2）数据清理；

（3）数据挖掘与探索；

（4）为分析做数据处理与准备；

（5）结合 Scikit-Learn、StatsModels 进行分析。

在本书中用得最多的 Pandas 对象是 DataFrame，它是一个两维数据表结构，包含多行多列，如表 1-1 所示。

表 1-1

	secID	ticker	secShortName	tradeDate	closePrice
0	000001.XSHE	000001	平安银行	2017-06-20	9.12
1	000002.XSHE	000002	万科A	2017-06-20	21.03
2	000004.XSHE	000004	国农科技	2017-06-20	27.03
3	000005.XSHE	000005	世纪星源	2017-06-20	5.45
4	000006.XSHE	000006	深振业A	2017-06-20	8.87
5	000007.XSHE	000007	全新好	2017-06-20	15.87

相对于 R 等统计软件，Pandas 借鉴了 R 的数据结构，因此拥有了 R 的很多方便的数据操作特性；在语法设计上，Pandas 比 R 和 Stata 更严谨且更简洁易用；基于 Python 自动管理内存的能力，以及在很多细节上的优化（比如在数据操作过程中的数据复制和引用），Pandas 拥有了更好的管理和计算大数据的能力。

Pandas 的底层基于 NumPy 搭建，因此 Pandas 拥有了 NumPy 的全部优点，比如 Pandas 定义的数据结构可以支持 NumPy 已经定义的计算，相当于拥有了 MATLAB 的矩阵计算能力；NumPy 原生的 C 接口也为扩展 Pandas 的计算性能带来了很大的方便。

对于金融用户来讲，Pandas 提供了一系列适用于金融数据的高性能时间序列与工具，例如 Panel、时间 Series 等。

3. Matplotlib

Matplotlib 是 Python 最著名的绘图库，提供了一整套和 MATLAB 相似的命令 API，十分适合进行交互式制图。我们也可以很方便地将它作为绘图控件，嵌入 GUI 应用程序中。

Matplotlib 的文档相当完备，而且在 Gallery 页面（http://matplotlib.org/gallery.html）中有上百幅缩略图，打开后都有源程序。因此，如果需要绘制某种类型的图，则只需在这

个页面中浏览、复制、粘贴，就基本可以搞定。

4. Seaborn

Seaborn 其实是在 Matplotlib 的基础上进行了更高级的 API 封装，从而使作图更容易，如图 1-1 所示。在大多数情况下使用 Seaborn 就能制作出非常有吸引力的图，而使用 Matplotlib 就能制作出具有更多特色的图。笔者认为：应该把 Seaborn 视为 Matplotlib 的补充，而不是替代物。

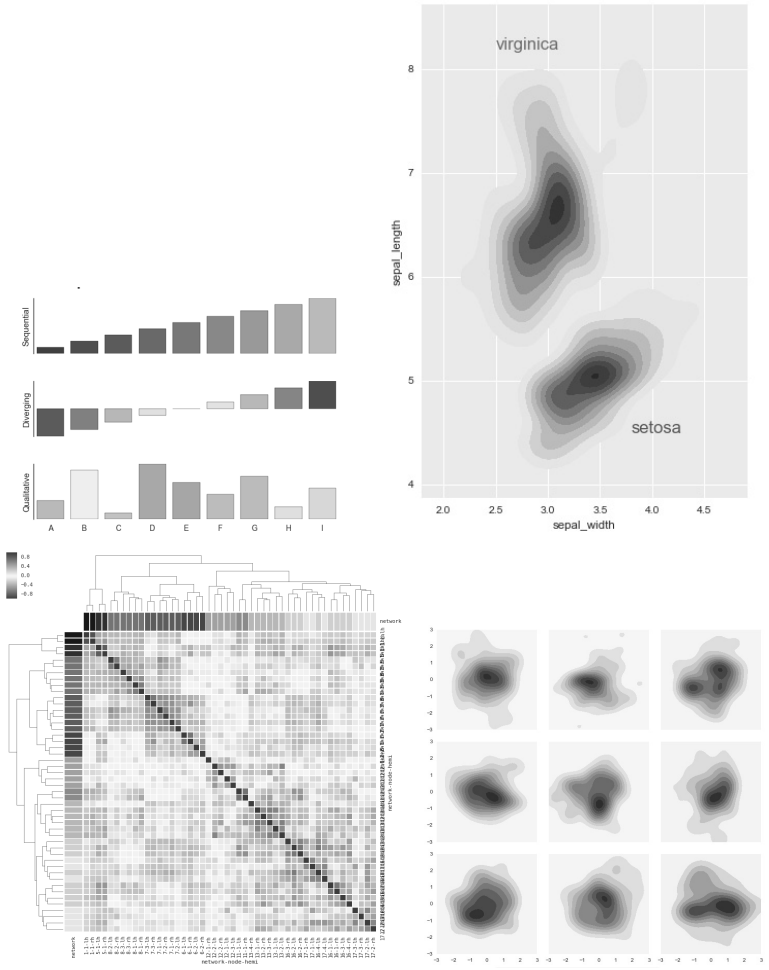


图 1-1

Seaborn 默认的浅灰色背景与白色网络线的灵感来源于 Matplotlib，却比 Matplotlib 的颜色更加柔和。我们发现，图对于传播信息很有用，几乎在所有情况下，人们喜欢图更甚于表。

5. SciPy

SciPy 包含致力于解决科学计算中常见问题的各个工具箱。它的不同子模块相当于不同的应用，例如插值、积分、优化、图像处理、特殊函数等。

SciPy 可以与其他标准科学计算程序库进行比较，比如 GSL（GNU C 或 C++科学计算库）或者 MATLAB 工具箱。SciPy 是 Python 中科学计算程序的核心包，用于有效地计算 NumPy 矩阵，让 NumPy 和 SciPy 协同工作。

6. Scikit-Learn

Scikit-Learn 是基于 Python 的机器学习模块，基于 BSD 开源许可证。Scikit-Learn 的基本功能主要被分为 6 部分：分类、回归、聚类、数据降维、模型选择和数据预处理，具体可以参考官方网站上的文档（http://scikit-learn.org/stable/）。

对具体的机器学习问题的解决，通常可以分为三步：数据准备与预处理；模型选择与训练；模型验证与参数调优。Scikit-Learn 封装了这些步骤，使建模的过程更方便、简单和快捷。

7. StatsModels

StatsModels 是 Python 的统计建模和计量经济学工具包，包括一些描述统计、统计模型估计和推断，例如线性回归模型、广义线型回归模型、方差分析模型、时间序列模型、非参检验、优化、绘图功能等。

8. Quartz

Quartz 是优矿的在线回测模块库，提供了跨资产多账户交易的量化策略框架，可以对策略进行专业的历史回测，得到详细的策略表现评估结果。它包括：多因子选股策略、事件驱动策略、CTA 策略、大宗商品期现预测策略、相对价值策略、宏观择时/对冲策略、分级基金轮动/套利策略、FOF 策略、期权波动率策略与资产配置和轮动策略等。最新版为 Quartz 3，其特点如下。

- 支持标准级的策略编写规范，删减了大量冗余的策略 API，让我们只需记忆少量的函数名，即可快速实现策略想法。
- 扩充了策略交易的资产范围，从原来只支持股票、基金这两种资产类型，扩充到支持股票、场内外基金、指数、期货这 4 种资产的全资产平台。也就是说，在一个策略中既可以交易股票，又可以交易基金，还可以交易期货，实现真正意义上的多空对冲策略。在不久的将来，还会支持期权、债券、港股等资产类型。
- 体现了策略信号生成和交易执行相分离的思想，可以灵活地进行各种资产的数据分析和交易撮合。
- 支持更多的策略特型，例如多因子裸多策略、市场中性策略、FOF 策略和 CTA 期现策略等。
- 会得到更充分的研究和支持，新的示例会逐步扩充在优矿知识库中，辅助我们更好地进行量化研究。
- 在性能方面进行了诸多优化，让我们可以用更快的运行速度进行策略研究。

9. CAL

在现阶段，中国市场缺乏为本土市场定制化的金融分析工具。中国的广大中小投资者在面对这个纷繁复杂的市场时往往束手无策：“这个债券定价合理吗？什么是隐含波动率？我的资产包要做优化，这个怎么处理？”这也是萦绕在所有投资者脑海中的问题。在过去，对这些问题的解决要通过购买昂贵的专业终端才能进行，是广大个人投资者可望而不可即的。

在如上所述的大环境下，CAL 应运而生。CAL 依托优矿量化实验室在线平台，基于C++开发高性能引擎，以 Python 作为接口语言向用户提供功能，它的宗旨是为广大投资者提供丰富、灵活的金融分析模块，降低投资者的分析平台搭建成本，帮助用户在金融大浪中顺利前行。可通过 https://uqer.io/help/faqCAL/查看 CAL 具体的帮助文档。

本书还会在第 3 章中更详细地介绍上述重要的库。

第 2 章

Python 基础介绍

2.1 Python 学习准备

如果已经安装了 Anaconda，则可以直接启动开始菜单 Anaconda 下的 Jupyter Notebook，或者直接在命令提示符中输入 jupyter notebook，就会弹出网页，开始 Notebook 之旅。如果还没有安装 Anaconda，则可以通过 pip install jupyter 进行安装。

Jupyter Notebook（此前被称为 IPython Notebook）是一个交互式笔记本，支持运行 40 多种编程语言。本节将介绍 Jupyter Notebook 的主要特性，以及为什么对于希望编写漂亮的交互式文档的人来说，Jupyter Notebook 是一个强大的工具。

在成功启动 Jupyter Notebook 后，可以在网页中看到当前路径下的文件，如图 2-1 所示。

默认的路径一般为“我的文档”，可以通过修改 Jupyter Notebook 的配置文件来变更 Jupyter Notebook 的默认路径。

如果想新建一个 Notebook，则只需单击右侧的 New 按钮，选择希望新建的 Notebook 类型，如图 2-2 所示。

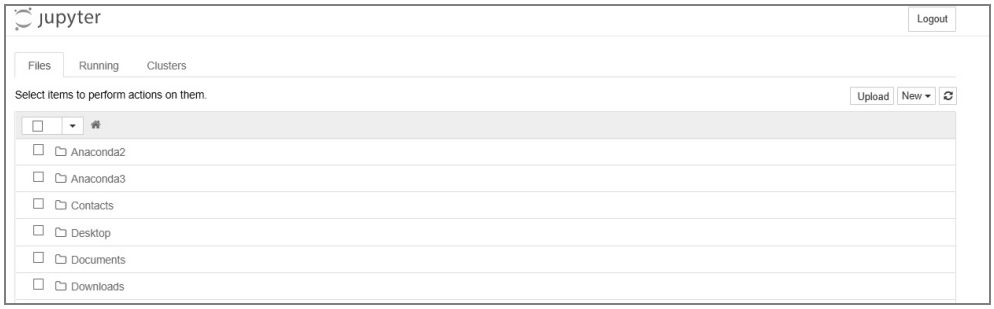


图 2-1

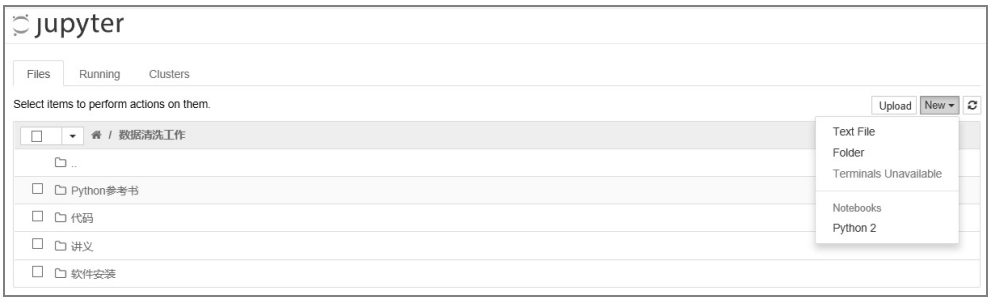


图 2-2

之后，我们发现 Jupyter Notebook 建立了一个空白的 Notebook 界面，如图 2-3 所示。

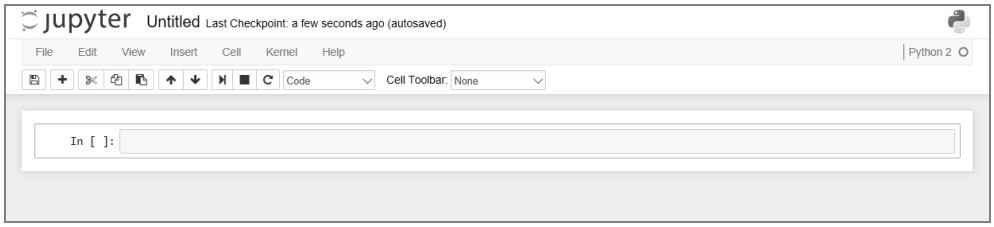


图 2-3

Notebook 界面由以下几部分组成。

◎ Notebook 的名称。

◎ 主工具栏，提供了保存、导出、重载 Notebook 及重启内核等选项。

◎ 快捷键。

◎ Notebook 的主要区域，包含了 Notebook 的内容编辑区。

我们之后会慢慢熟悉这些菜单和选项。如果想详细了解有关 Notebook 或一些库的具体使用方法，则可以使用菜单栏右侧的帮助菜单。

Notebook 界面下方的主要区域由被称为单元格的部分构成。每个 Notebook 由多个单元格构成，每个单元格有不同的用途。

我们从如图 2-4 所示的截图中看到的是一个灰色代码单元格（Code Cell），以[ ]为开头，在这种类型的单元格中可以输入任意代码并执行。例如，在输入 1 + 2 并按下 Shift + Enter 键后，单元格中的代码就会被计算，结果也就在 Notebook 中直接显示。

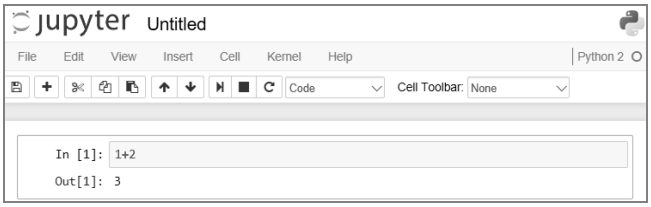


图 2-4

根据边框线，我们可以轻松地识别出当前工作的单元格。接下来，我们在第 2 个单元格中输入其他代码，如图 2-5 所示。

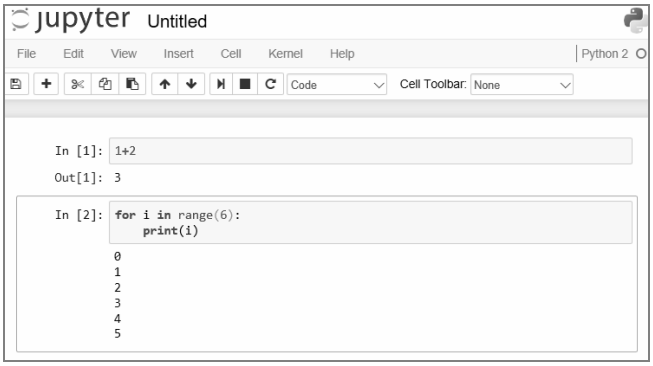


图 2-5

和前一个示例一样，在单元格中的代码被计算后会马上显示结果。Notebook 由于做了分块的编程，所以支持修改之前的单元格并对其重新计算。不过，也可以重新计算整个 Notebook，单击 Cell→Run all 即可完成。

现在我们已经知道了如何输入代码，那么为什么不试着让这个 Notebook 更漂亮、内容更丰富呢？为此，我们需要使用其他类型的单元格，即 Header 单元格和 Markdown 单元

格。首先，在顶部添加一个 Notebook 标题，选中第 1 个单元格；然后，单击 Insert→Insert above（在上方插入单元格），在文档的顶部会马上出现一个新的单元格，单击快捷键栏中的单元格类型，将其变成一个标记单元格（Markdown），如图 2-6 所示。

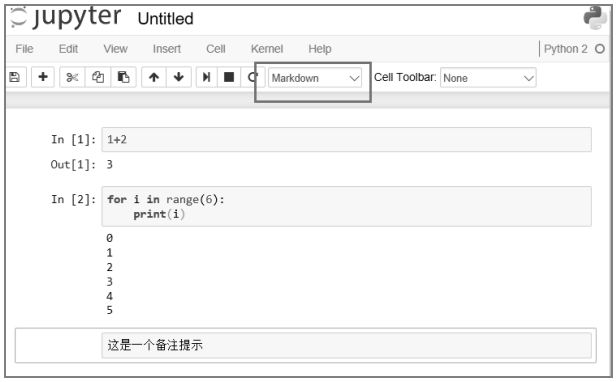


图 2-6

或者将快捷键栏中的单元格类型变成标题单元格（Heading），选中下拉选项中的 Heading，会弹出一个消息，告诉我们如何创建不同层级的标题，这样就有了一个标题类型的单元格，这个单元格以“#”为开头，这意味着它是一个一级标题。如果需要子标题，则可以使用以下标记表示（改变单元格类型时在弹出的消息中有解释），如图 2-7 所示。

图 2-7

最后，可以重命名该 Notebook，单击 File→Rename，然后输入新的名称。这样，新的名称将会出现在窗口的左上角，在 Jupyter 的标志旁边。

另外，Matplotlib 是一个用于创建漂亮图形的 Python 库，在结合 Jupyter Notebook 使用时体验更佳。如果需要在 Notebook 中使用 Matplotlib，则需要告诉 Jupyter 获取 Matplotlib 生成的所有图形，并将其嵌入 Notebook 中。为此，需要输入并运行：

```
%matplotlib inline
```

总之，Jupyter Notebook 是一款非常强大的工具，可以创建漂亮的交互式文档及制作教学材料，等等。建议马上使用 Jupyter Notebook，探索 Notebook 更多的强大功能。如果需要使用第 2 章中的代码，则可以直接将代码文件复制（Copy）到 Jupyter Notebook 打开的目录下，单击它即可打开。

2.2　Python 语法基础

2.2.1　常量与变量

在 Python 中程序运行时不会被更改的量叫作常量，比如数字 7 和字符串"abc"在运行时一直都是数字 7 与字符串"abc"，不会被更改成其他量，它们就是常量。除此之外，我们可以定义任意字符串为指定值的常量。常量有一个特点，即一旦被绑定，就不能更改。在 Python 中是不能像 C 等编程语言一样通过 const 来定义常量的，在 Python 中可以直接创建而不需要定义。在 Python 中随着程序的运行而更改的量叫作变量，比如我们可以定义一个变量 *i*，并将数字 5 赋给变量 *i*，再将数字 7 赋给变量 *i*，这时 *i* 的值就变成了 7。可见，*i* 的值是可以更改的，像 *i* 这样的可以更改值的量叫作变量。变量有一个特点：既可被赋值，也可被更改。

2.2.2　数与字符串

在 Python 中数的类型主要有 5 种，分别为有符号整数型（int）、长整型（long）、浮点型（float）、布尔型（bool）和复数型（complex）。具体如下所示：

◎　0、1、-1、1009、-290 等是 int 型；

◎ 87887l、-90990l、2345671 等是 long 型；

◎ 2.7788、3.277、8.88 等是 float 型；

◎ bool 型只有 True 和 False 这两种类型。

◎ 4+2j、-9+20j、56+7j 等是复数型。

示例如下：

```
In [1]: a=100
        b=4+2j
        print a,b,b.real
Out[1]: 100 (4+2j) 4.0
```

在 Python 中用引号引起来的字符集叫作字符串，例如'hello'、"my Python"、"2+3"等，示例如下：

```
In [2]: #单引号
        c1='Hello,Python'
        print c1
        c2='It is a "dog"!'
        print c2
Out[2]: Hello,Python
        It is a "dog"!
In [3]: c1.upper()
Out[3]: 'HELLO,PYTHON'
In [4]: c1[0]  #获取第 1 个字母
Out[4]: 'H'
In [5]: c1[0]='a'
   ---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-c0f11db241b6> in <module>()
----> 1 c1[0]='a'

   TypeError: 'str' object does not support item assignment
```

注意，在 Python 中字符串是不支持修改的。

在 Python 字符串中使用的引号可以是单引号、双引号或三引号，但它们的使用方法不同，示例如下：

```
In [6]: #双引号
        c1="Hello,Python"
```

```
        print c1
        c2="It is a dog!"
        print c2
        #三引号
        c3="""这
        是
        Python 量化投资"""
        print c3

Out[6]: Hello,Python
        It is a dog!
        这
        是
        Python 量化投资
```

如果想在 Python 里输出一个"It's a dog!"字符串，则应该如何编写 Python 程序呢？有两种方法，一种方法是通过如上所示的单双引号间插使用，即 print "It's a dog!"，用双引号来包含有单引号的字符串；另一种方法是使用转义符“\”，示例如下，其中“\n”代表换行。

```
In [7]: #转义符
        print 'It\'s a dog!'
        print "hello boy\nhello boy"
Out[7]: It's a dog!
        hello boy
        hello boy
```

在 Python 里，如果在字符串中包含转义符‘\’，但需要将它原样保留，不进行任何处理，则还可以使用自然字符串标识，告诉 Python 在字符串中的‘\’不是转义字符，即在字符串前加上 r，示例如下：

```
In [8]: #自然字符串
        print "hello boy\nhello boy"
        print r"hello boy\nhello boy"
Out[8]: hello boy
        hello boy
        hello boy\nhello boy
```

如果需要重复输出一个字符串，则除了可以手动输入，还可以使用字符串的重复运算符。比如，若要将 hello 重复输出 20 次，则可以通过"hello"*20 这种运算方法由计算机自动执行重复地输出指令，示例如下：

```
In [9]: #字符串的重复
        print "我爱 Python\n"*10
Out[9]: 我爱 Python
        我爱 Python
        我爱 Python
        我爱 Python
        我爱 Python
        我爱 Python
        我爱 Python
        我爱 Python
        我爱 Python
        我爱 Python
```

字符串"wangxiaochuan"中的"wang"、"xiao"等都是"wangxiaochuan"的子字符串。如果想将一个字符串中的子字符串取出来，就要进行子字符串运算。而子字符串的运算方法主要有两种，一种是索引运算法[]，另一种是切片运算法[:]，示例如下：

```
In [10]: #子字符串
         #索引运算符从 0 开始索引
         #切片运算符[a:b]是指从第 a 下标开始到第 b-1 下标。
         c1=u"wangxiaochuan"
         c2=c1[0]
         c3=c1[7]
         c4=c1[:2]
         c5=c1[2:]
         c6=c1[4:7]
         c7=c1[-1]
         c8=c1[-3:]
         cuts=[2,4,6]
         c9=c1+c2
         print c2
         print c3, c4, c5, c6,c7,c8,c9
Out[10]: w
         o wa ngxiaochuan xia n uan wangxiaochuanw
```

注意，Python 是以 0 作为开始来计算位数的，所以 c1[1]是“a”而不是“w”。“：”代表从第几位取到第几位，如果“：”的前后都是数字，那么其作用是“包前不包后”，c1[4:7]代表取 c1 的[4,5,6]位置的元素，不包括第 7 位的元素。

2.2.3　数据类

Python 的基本数据类型有 4 种：列表、元组、集合和字典。Python 中的各种库如 Pandas 也有自己的数据类型，但都是由最基本的列表、元组、字典等组合而成的。

1. 列表（List）

在 Python 中没有数组的概念，与数组最接近的概念就是列表和元组。列表是用来存储一连串元素的容器，用[]来表示。比如，在一个班里有 30 个学生，我们需要将这 30 个学生安排到一间教室里上课，如果把这 30 个学生分别比作元素，那么这个教室就是数组；这 30 个学生是按座位坐好且有序排列的，在数组中的元素也是有序排列的，示例如下：

```
In [11]: #列表
         students=["小明","小华","小李","小娟","小云",3]
         print students[len(students)-1]
         print students[-1]
         print students[2]
Out[11]: 3
         3
         小李
```

列表可以存放不同类型的数据，是最常用的 Python 数据类型。与字符串不同，列表元素支持改写，示例如下：

```
In [12]: #列表元素支持修改
         students=["小明","小华","小李","小娟","小云",3]
         print students[3]
         students[3]="小月"
         print students[3]
         students[5]="小楠"
         print students[5]
         students[5]=19978
         print students[5]
Out[12]: 小娟
         小月
         小楠
         19978
```

2. 元组（Tuple）

在 Python 中与数组类似的还有元组，元组中的元素也进行索引计算。列表和元组的区别在于：列表中的元素的值可以修改，而元组中的元素的值不可以修改，只可以读取；另外，列表的符号是[]，而元组的符号是()，示例如下：

```
In [13]: #元组
         students=("小明","小军","小强","小武","小龙")
         print students[1]
Out[13]: 小军
In [14]: #元组
         students=("小明","小军","小强","小武","小龙")

         students[1]="小云"
         print students[1]
   ---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-e8b2ee4ae754> in <module>()
     2 students=("小明","小军","小强","小武","小龙")
     3
----> 4 students[1]="小云"
     5 print (students[1])

   TypeError: 'tuple' object does not support item assignment
```

元组比列表更加安全，因为不能修改，所以可以安全地保存一个引用，而不用担心里面的内容被其他程序修改，从而导致逻辑错误。

3. 集合（Set）

在 Python 中集合主要有两个功能：一个功能是进行集合操作，另一个功能是消除重复的元素。集合的格式是 set（元素），示例如下：

```
In [15]: a=[1,2,3,4]
         set(a)
Out[15]: {1, 2, 3, 4}
In [16]: a=set("abcnmaaaaggsng")
         print a
         b=set("cdfm")
         #交集
         x=a&b
```

```
          print x
          print a
          #并集
          y=a|b
          print y
          #差集
          z=b-a
          print z
          #去除重复元素
          new=set(a)
          print new
Out[16]: set(['a', 'c', 'b', 'g', 'm', 'n', 's'])
          set(['c', 'm'])
          set(['a', 'c', 'b', 'g', 'm', 'n', 's'])
          set(['a', 'c', 'b', 'd', 'g', 'f', 'm', 'n', 's'])
          set(['d', 'f'])
          set(['a', 'c', 'b', 'g', 'm', 's', 'n'])
```

4. 字典（Dict）

在 Python 中的字典也叫作关联数组，可以理解为列表的升级版，用大括号{}括起来，格式为{key1:value1,key2:value2,….,keyn:valuen}，即：字典的每个键值（key=>value）对用冒号分割，每个对之间用逗号分隔，整个字典包括在花括号中。

如下所示，我们可以理解为字典 k 包含了“姓名”与“籍贯”方面的信息，也可以在字典中加入新的 key（爱好）：

```
In [17]: #字典
          k={"姓名":"王小川","籍贯":"山东"}
          print k["籍贯"]
          #添加字典里面的项目
          k["爱好"]="游泳"
          print k["姓名"]
          print k["爱好"]
Out[17]: 山东
          王小川
          游泳
```

通过 k.keys()可以取到字典的 key 字段。Python 字典包含了以下内置函数。

◎　cmp(dict1, dict2)：比较两个字典元素。

◎ len(dict)：计算字典元素的个数，即键的总数。

◎ str(dict)：输出字典可打印的字符串表示。

◎ type(variable)：返回输入的变量类型，比如，如果变量是字典，就返回字典类型。

Python 字典包含了以下内置方法。

◎ dict.clear()：删除字典内的所有元素。

◎ dict.copy()：返回一个字典的浅复制。

◎ dict.fromkeys()：创建一个新字典，将序列 seq 中的元素作为字典的键，将 val 作为字典的所有键对应的初始值。

◎ dict.get(key, default=None)：返回指定键的值，如果值不在字典中，则返回 default 值。

◎ dict.has_key(key)：如果键在字典 dict 中，则返回 true，否则返回 false。

◎ dict.items()：以列表形式返回可遍历的(键, 值)元组数组。

◎ dict.keys()：以列表形式返回一个字典的所有键。

◎ dict.setdefault(key, default=None)：和 get()类似，但如果键不存在于字典中，则将会添加键并将值设为 default。

◎ dict.update(dict2)：把字典 dict2 的键/值对更新到 dict 里。

◎ dict.values()：以列表形式返回字典中的所有值。

2.2.4 标识符

我们在 Python 中编程时使用的名称叫作标识符，变量和常量就是标识符的一种。在 Python 中标识符的命名是有规则的，按正确命名规则命名的能使用的标识符叫作有效标识符，不能使用的标识符叫作无效标识符。

有效标识符的命名遵循以下规范。

◎ 标识符的第 1 个字符必须是字母或下画线，不能出现数字或其他字符。

◎ 标识符除第 1 个字符外，其他部分可以是字母、下画线或数字。

◎　标识符是大小写敏感的，比如，name 与 Name 是不同的标识符。

Python 中的关键字是指系统自带的具备特定含义的标识符。常用的 Python 关键字主要有：and、elif、global、or、else、pass、break、continue、import、class、return、for 和 while。现在，查看具体的 Python 关键字：

```
    In [18]: #常用的关键字
             #查看一下关键字有哪些
             import keyword
             print keyword.kwlist
    Out[18]: ['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return',
'try', 'while', 'with', 'yield']
```

2.2.5　对象

Python 虽然是解释型语言，但从设计之初就已经是一门面向对象的语言，对于 Python 来说一切皆对象。正因为如此，在 Python 中创建一个类和对象是很容易的，当然，如果习惯面向过程或者函数的写法也是可以的，在 Python 中没有硬性的限制。

Python 的面向对象特征如下。

1. 封装

在面向对象程序设计中的术语对象（Object）基本上可以看作数据（特性）及由一系列可以存取、操作这些数据的方法所组成的集合。传统意义上的“程序=数据结构+算法”被封装“掩盖”，并简化为“程序=对象+消息”。对象是类的实例，类的抽象则需要经过封装。封装可以让调用者直接使用对象而不用关心对象是如何构建的。

2. 继承

我们的直觉是，继承是一种复用代码的行为。我们可以将继承理解为，它是以普通的类为基础建立的专门的类对象，子类和它继承的父类是 IS-A 的关系。

3. 多重继承

不同于 C#，Python 是支持多重类继承的（C#可继承自多个 Interface，但最多继承自

一个类）。多重继承机制有时很好用，但是它容易让事情变得复杂。

4. 多态

多态意味着可以对不同的对象使用同样的操作，但它们可能会以多种形态呈现结果。

在 Python 中一切皆对象，这意味着在 Python 中定义的一个数字或字符串都是对象，都可以直接使用其方法，示例如下：

```
In [19]: a='abc'
         print a.upper()
         print a.title()
Out[19]: ABC
         Abc
```

2.2.6 行与缩进

Python 中的逻辑行主要指一段代码在逻辑上的行数，而物理行指的是我们实际看到的行数，示例如下：

```
In [20]: #逻辑行与物理行
         #以下是 3 个物理行
         print "abc"
         print "789"
         print "777"
         #以下是 1 个物理行，3 个逻辑行
         print "abc";print "789";print "777"
         #以下是 1 个逻辑行，3 个物理行
         print '''我是帅哥
         王老师很帅
         这是 Python 教程! '''
Out[20]: abc
         789
         777
         abc
         789
         777
         我是帅哥
         王老师很帅
         这是 Python 教程!
```

在 Python 中一个物理行一般可以包含多个逻辑行，在一个物理行中编写多个逻辑行时，逻辑行与逻辑行之间用分号隔开。在每个逻辑行的后面必须有一个分号，但是我们在编写程序时，如果一个逻辑行占了一个物理行最后的位置，则这个逻辑行可以省略分号。

在 Python 中对逻辑行行首的空白是有规定的，逻辑行行首的空白不对，就会导致程序执行出错。与其他语言相比，这是一个很重要的不同点，示例如下：

```
In [21]: #什么是缩进
             print    '123'
         File "<ipython-input-21-ee15b08d83f6>", line 2
         print '123'
         ^
         IndentationError: unexpected indent
```

在 Python 中对缩进的空白是有要求的，示例如下：

```
In [22]: #如何缩进
         #一般情况下，行首应该不留空白
         import sys
         #缩进的方法有两种，可以按空格，也可以按 tab 键
         #if 语句的缩进方法
         a=7
         if a>0:
            print "hello"
         #while 语句的缩进方法
         a=0
         while a<7:
            print a  # 思考 a 为什么不包含 7
            a+=1
         a=100
Out[22]: hello
         0
         1
         2
         3
         4
         5
         6
```

在 Jupyter Notebook 中可以通过 Tab 键完成右缩进，通过 Shift+Tab 键完成左缩进。

2.2.7 注释

在 Python 中一般通过#进行注释，从#开始一直到一行（物理行）结束的部分都是注释。

2.3 Python 运算符与表达式

在 Python 中，有时我们需要对一个或多个数字，或者需要对一个或多个字符串，进行运算操作，比如，让字符串重复的“*”也是一种运算符；又如 2+3 中的“+”也是一种运算符。在 Python 中常见的运算符有：+、-、/、*、<、>、!=、//、%、&、|、^、~、>>、<<、<=、>=、==、not、and 和 or。

2.3.1 算数运算符

之前在介绍变量类型的时候，其实已经用过了很多算术符，比如+、-、*、/、* 等，这和任何其他编程语言都是类似的。除此之外，还有几个符号是之前没有提到的，例如%是用来返回除法余数的运算符号，*表示求幂运算，//表示求商的整数部分，等等。示例如下：

```
In [23]: #"+":两个对象相加
         #两个数字相加
         a=7+8
         print a
Out[23]: 15
In [24]: #两个字符串相加
         b="GOOD"+" JOB!"
         print b
Out[24]: GOOD JOB!
In [25]: #"-":取一个数字的相反数或者实现两个数字相减
         a=-7
         print a
         b=-(-8)
         print b
         c=19-1
         print c
Out[25]: -7
```

```
          8
          18
In [26]: #"*":两个数相乘或者字符串重复
          a=4*7
          print a
          b="hello"*3
          print b
          b="hello\n"*3
          print b
Out[26]: 28
          hellohellohello
          hello
          hello
          hello
In [27]: #"/":两个数字相除
          a=7/2
          print a
          b=7.0/2
          c=7/2.0
          print b
          print c
Out[27]: 3
          3.5
          3.5
In [28]: from __future__ import division
          a=7/2
          print a
Out[28]: 3.5
In [29]: #"**":求幂运算
          a=2**3   #相当于 2 的 3 次幂，就是 2*2*2
          print a
Out[29]: 8
In [30]: #"//":除法运算，然后返回其商的整数部分，舍掉余数
          a=10//3
          print a
Out[30]: 3
In [31]: #"%":除法运算，然后返回其商的余数部分，舍掉商
          a=10%3
          print a
          b=10%1  #在没有余数的时候返回什么？
          print b
```

```
Out[31]: 1
         0
```

__future__模块是包含 Python 未来特性的模块，如果使用的是 Python 2，就可以通过导入这个模块来使用 Python 3 的特性。

2.3.2 比较运算符

比较运算符可以用于比较两个值，所有的内建类型都支持比较运算。当用运算符比较两个值时，结果是一个逻辑值，不是 True，就是 False。虽然支持比较，但是有一点要注意：不同的类型的比较方式不一样，数字类型会根据数字的大小和正负进行比较，而字符串会根据字符串序列值进行比较，等等。如表 2-1 所示。

表 2-1

运 算 符	描　　述	示　　例
==	检查两个操作数的值是否相等，如果值相等，则条件变为真	(a==b)不为 true
! =	检查两个操作数的值是否相等，如果值不相等，则条件变为真	(a!=b)为 true
<>	检查两个操作数的值是否相等，如果值不相等，则条件变为真	(a<>b)结果为 true
>	检查左操作数的值是否大于右操作数的值，如果是，则条件成立	(a>b)为 true
<	检查左操作数的值是否小于右操作数的值，如果是，则条件成立	(a<b)为 true
>=	检查左操作数的值是否大于或等于右操作数的值，如果是，则条件成立	(a>=b)为 true
<=	检查左操作数的值是否小于或等于右操作数的值，如果是，则条件成立	(a<=b)为 true

示例如下：

```
In [32]: #"<": 小于符号，返回一个 bool 值
         a=3<7
         print a
         b=3<3
         print b
         type(a)
Out[32]: True
         False
         bool
In [33]: #">":大于符号，返回一个 bool 值
         a=3>7
         print a
```

```
          b=3>1
          print b
Out[33]: False
          True
In [34]: #"!=":不等于符号，同样返回一个 bool 值
          a=2!=3
          print a
          b=2!=2
          print b
Out[34]: True
          False
In [35]: #"<=":小于等于符号，比较运算，小于或等于，返回一个 bool 值
          a=3<=3
          print a
          b=4<=3
          print b
          #">="
          a=1>=3
          print a
          b=4>=3
          print b
Out[35]: True
          False
          False
          True
In [36]: #"==":比较两个对象是否相等
          a=12==13
          print a
          b="hello"=="hello"
          print b
Out[36]: False
          True
```

2.3.3　逻辑运算符

Python 语言支持逻辑运算符，其具体的逻辑表达式、描述及实例如表 2-2 所示（在以下实例中假设变量 a 为 10，b 为 20）。

表 2-2

运 算 符	逻辑表达式	描　　述	实　　例
and	x and y	布尔"与"，如果 x 为 False，则 x and y 返回 False，否则返回 y 的计算值	(a and b)返回 20
or	x or y	布尔"或"，如果 x 是非 0，则返回 x 的值，否则返回 y 的计算值	(a or b)返回 10
not	not x	布尔"非"，如果 x 为 True，则返回 False；如果 x 为 False，则返回 True	not(a and b)返回 False

示例如下：

```
In [37]: a=''
         not a
Out[37]: True
In [38]: #not:逻辑非
         a=True
         b=not a
         print b
         c=False
         print not c
Out[38]: False
         True
In [39]: #and:逻辑与
         '''
         True and True 等于 True
         True and False 等于 False
         False and True 等于 False
         '''
         print True and True
Out[39]: True
In [40]: #or:逻辑或
         '''
         True or True 等于 True
         True or False 等于 True
         False or False 等于 False
         '''
         print True or False
Out[40]: True
```

2.3.4　Python 中的优先级

在 Python 中程序或运算符的执行是有先后顺序的，比如，A 与 B 同时出现，如果 A 可以优先于 B 执行，就说明 A 的优先级比 B 的优先级高；其中，A 与 B 可以是运算符，也可以是程序。也就是说，Python 中的优先级分为两种，一种是程序之间的优先级，另一种是运算符之间的优先级。这里主要讨论运算符之间的优先级，在运算符优先级排行榜中，各运算符的排行名次如下。

- 第 1 名：函数调用、寻址、下标。
- 第 2 名：幂运算**。
- 第 3 名：翻转运算~。
- 第 4 名：正负号。
- 第 5 名：*、/、%。
- 第 6 名：+、-。
- 第 7 名：<<、>>。
- 第 8 名：按位&、^、|，其实这三个也是有优先级顺序的，但是它们处于同一级别，所里这里不再细分。
- 第 9 名：比较运算符。
- 第 10 名：逻辑的 not、and、or。
- 第 11 名：Lambda 表达式。

关于其具体讲解，读者可自行参考配套代码。

2.4　Python 中的控制流

在 Python 中程序代码总是按照一定顺序执行的，如果想改变代码的执行顺序，比如我们有时需要从上到下按顺序执行，有时需要跳转执行，有时需要选择不同的分支执行，有时甚至需要循环地执行，这时就要用到控制流。我们通常根据不同的需要来选择控制语句，以控制某些代码段的执行方式，这些不同功能的控制语句就叫作控制流。

2.4.1 控制流的功能

控制流的功能就是控制代码的执行方式，下面通过两种方式来实现同一功能。

方式一：

```
In [41]: i=1
         print i
         i=i+1
         print i

         i=1
         print i
         i=i+1
         print i
Out[41]:
         1
         2
         1
         2
```

方式二：

```
In [42]: for k in range(0,2):
             i=1
             print i
             i=i+1
             print i
Out[42]:
         1
         2
         1
         2
```

例如，若要实现一个分支判断功能：如果天气晴朗（sunny），就输出“basketball”，否则，输出“badminton”。按照正常的顺序执行方式，是无法实现该功能的，此时就可以选择使用 if 语句，我们会在后面详细介绍具体的使用方法：

```
In [43]: weather="rainy"
         if weather=="sunny":
            print "basketball"
         else:
            print "badminton"
```

```
Out[43]: badminton
```

2.4.2　Python 的三种控制流

与许多编程语言相同，在 Python 中程序通常是由上向下执行的，而有时我们为了改变程序的执行顺序，会使用控制流语句控制程序的执行方式。

在 Python 中有三种控制流类型：第 1 种是顺序结构，指按顺序执行的结构；第 2 种是分支结构，指选择执行的结构，常用的语句为 if；第 3 种是循环结构，指循环多次执行的结构，常用的语句为 for、while。下面分别通过代码实例认识这三种结构。

顺序结构的代码实例如下：

```
In [44]: a=1
         print a
         a=a-1
         print a
         a=a+2
         print a
Out[44]:
         1
         0
         2
```

分支结构的代码实例如下：

```
In [45]: a=0
         if a==1:
            print "She"
         else:
            print "He"
Out[45]: He
```

循环结构 for 的代码实例如下：

```
In [46]: for i in range(5):
            print "hello world"
            print i
Out[46]:
         hello world
         0
```

```
            hello world
            1
            hello world
            2
            hello world
            3
            hello world
            4
```

循环结构 while 的代码实例如下：

```
In [47]: i=5
         while i:
            print "hello world"
            print i
            i=i-1
Out[47]:
         hello world
         5
         hello world
         4
         hello world
         3
         hello world
         2
         hello world
         1
```

2.4.3 认识分支结构 if

if 语句在 Python 中表示选择性地执行，虽然形式简单，但功能相当强大。选择性执行是常规程序设计中的主要控制机制之一，是计算机能够执行复杂任务的关键组成部分。

if 语句的格式用法如下：

```
'''
if（是这样的）：
    执行该部分的语句
elif（或者是这样）：
    执行 elif 部分的语句
else（或者以上情况都不是）：
```

```
    执行该部分的语句
'''
```

按照上面的格式规定，我们通过实例来讲解 if 语句在几种选择情况下的使用：

```
In [48]: #在一种情况下的 if 用法
         a=8
         if a==8:                          #这里的等号注意是==，而非=
            print "hello world"
Out[48]: hello world

In [49]: #在两种选择情况下的 if 用法
         a=9
         if a==9:
            print "hello world"
         else:
            print "Null"
Out[49]: hello world

In [50]: #在三种选择情况下的 if 用法
         a=10
         if a==10:
            print "hello world"
         elif a>8:
            print "hello"
         elif a>6:
            print "world"
         else:
            print "Null"
Out[50]: hello world
```

注意，由于同时满足了条件 a==10、a>8 与 a>6，所以 Python 会优先执行最先满足的条件下的代码。

if 语句的使用要点如下。

◎　各分支尽量不重复，并且尽量包含全部可能性。

◎　在 if 语句里还可以再写 if 语句，形成 if 语句的嵌套。

◎　elif 语句和 else 语句永远只是可选的。

◎　在 Python 中没有 switch 语句。

2.4.4 认识循环结构 for…in

Python 中的 for…in 语句是指在一系列的对象上进行迭代，即逐一使用序列中的每个项目。

for…in 语句的格式用法如下：

```
'''
    for i in 集合：
        执行该部分
    else：
        执行该部分
'''
```

根据上面对 for…in 语句的格式规定，我们通过实例来讲解 for…in 语句的使用方法。

最基本的 for 语句为搭配列表使用：

```
In [51]: for i in [1,2,3,4,5]:
             print i
Out[51]:
          1
          2
          3
          4
          5
```

在 for 语句中使用 range 函数时，range 函数中的第 1 个元素代表起始数字，第 2 个元素代表终止数字，第 3 个元素代表步长，如果不指定第 3 个元素，则默认步长为 1：

```
In [52]: for i in range(1,10,3):
             print "hello"
             print i
Out[52]:
          hello
          1
          hello
          4
          hello
          7
```

在 Python 中，for…in 语句经常与“迭代器”组合使用，“迭代器”指任何可以进行迭代操作的对象，比如在前面提到的列表、字符串、元组、字典、文件，等等。

```
In [53]: char=u"你好世界"        #U 代表这是 Unicode 编码
         tem=("元组：你好","元组：世界")
         list=["列表：你好","列表：世界"]
         dict={"first":"字典：你好","second":"字典：世界"}
         for i in char:
            print i
         for i in tem:
            print i
         for i in list:
            print i
         for i in dict:
            print i
Out[53]:
         你
         好
         世
         界
         元组：你好
         元组：世界
         列表：你好
         列表：世界
         second
         first
```

for...in 语句的使用要点如下。

◎　适当运用 range、xrange 函数。

◎　在 for 中还可以再写 for，形成循环嵌套。

◎　合理地运用 lambda 代替 for 循环，可以提高运行速度。

2.4.5　认识循环结构 while

在 Python 中 while 语句也可以用来控制一段语句的重复执行，即在某个条件下循环执行某段程序，以处理需要重复处理的相同任务。

while 语句的格式用法如下：

```
while 条件为真：
    循环执行该部分语句
```

```
else:
    如果条件为假，则执行该部分语句。else 部分可以省略。
```

下面根据 while 语句的格式规定，通过实例讲解 while 语句的使用方法。

如下所示是最简单的没有 else 部分的例子，这是一个死循环，请不要轻易尝试：

```
In [54]: a=True
         while a:
            print "hello world"
Out[54]:
         hello world
         hello world
         …
         hello world
```

接下来是有 else 部分的例子：

```
In [55]: b=False
         while b:
            print "hello world"
         else:
            print "Null"
Out[55]: Null
```

我们再来看看更复杂一些的有嵌套的 while 语句：

```
In [56]: a=1
         while a<5:
            if a<=3:
               print a
            else:
               print "hello"
            a=a+1
Out[56]:
         1
         2
         3
         hello
```

while 语句的使用要点如下。

◎ 配合 if 分支语句使用。

◎ 要恰当地运用 break 和 continue 语句，以便适时结束循环；

◎ 在 while 语句中还可以再写 while 语句，从而形成循环嵌套。

2.4.6　break 语句与 continue 语句

在 Python 中用于终止循环的语句有 break 语句和 continue 语句，这两种语句跳出循环的层次不同，但各有用处。

1. break 语句

break 语句是用来终止程序的执行的。当在循环结构中出现 break 语句时，即使在循环条件中没有 false 条件，或者序列还没被完全递归完，也会终止循环语句。如果 break 语句出现在嵌套循环中，则应当终止执行最深层的循环，并开始执行下一行代码。

break 语句在 while 循环中的示例如下：

```
In [57]: a=1
         while a:
            print a
            print "hello world"
            a=a+1
            if a==3:
               break
Out[57]:
         1
         hello world
         2
         hello world
```

break 语句在 for 循环中的示例如下：

```
In [58]: for i in range(5,9):
            print i
            print "hello world"
            if i>6:
               break
Out[58]:
         5
         hello world
```

```
            6
            hello world
            7
            hello world
```

将 break 语句用在双层循环语句中，跳出最内层的循环的示例如下：

```
    In [59]: a=5
             while a<=8:
                a=a+1
                for i in range(1,3):
                   print a,i
                   if i==2:
                      break
    Out[59]:
             6 1
             6 2
             7 1
             7 2
             8 1
             8 2
             9 1
             9 2
```

2. continue 语句

continue 语句的功能是强制停止循环中的这一次执行，直接跳到下一次执行。即跳过当前循环的剩余语句，继续进行下一轮循环。

continue 语句在 while 循环中的示例如下：

```
    In [60]: n = 0
             while n < 10:
                n = n + 1
                if n % 2 == 0:  # 如果 n 是偶数，则执行 continue 语句
                   continue     # continue 语句会直接继续下一轮循环，后续的 print() 语句不会执行
                print n
    Out[60]:
             1
             3
             5
```

```
          7
          9
```

输出结果全部为奇数，因为当 n 为偶数时跳出本次循环。

continue 语句在 for 循环中的示例如下：

```
In [61]: for n in range(1,10):
             if n%2==0:
                continue
             print n
Out[61]:
          1
          3
          5
          7
          9
```

输出结果同样全为奇数，因为当 n 为偶数时跳出本次循环。

continue 语句在双层循环语句中的示例如下：

```
In [62]: a=1
         while a<7:
            a=a+1
            if a==4:
               continue
            for i in range(7,10):
               if i==9:
                  continue
               print i
Out[62]:
          7
          8
          7
          8
          7
          8
          7
          8
          7
          8
```

3. continue 语句与 break 语句的区别

continue 语句指的是结束执行本次循环中剩余的语句，然后继续下一轮的循环；而 break 语句指的是直接结束这个循环，包括结束执行该循环的剩余的所有次循环。

请区分以下两个程序。

程序 1 如下：

```
In [63]: for i in range(10,19):
             if i==15:
                continue
             print i
Out[63]:
          10
          11
          12
          13
          14
          16
          17
          18
```

程序 2 如下：

```
In [64]: for i in range(10,19):
             if i==15:
                break
             print i
Out[64]:
          10
          11
          12
          13
          14
```

在程序 2 中，i=15 及之后均没有输出，因为执行遇到 break，直接跳过了剩余的所有循环，而程序 1 仅跳过了 i=15 这一次循环。

2.5　Python 函数

2.5.1　认识函数

函数的英文是 function，所以通俗地来讲，函数就是功能的意思。函数是用来封装特定功能的，比如，在 Python 里，len()是一个函数，其实现的功能是返回一个字符串的长度，所以 len()函数的特定功能是返回长度；又如，我们可以自己定义一个函数，然后编写这个函数的功能，之后在要使用时再调用这个函数。所以，我们可以将函数分为两种类型，一种是系统自带的不用我们编写其功能的函数，比如 len()函数；另一种是我们自定义的需要我们编写其功能的函数，这种函数的自由度高，叫作自定义函数，需要在使用时直接调用该函数。

1. Python 中的函数的功能

在 Python 中有很多内建函数，当然，随着学习的深入，我们也可以创建对自己有用的函数。为了更清楚地理解函数的功能，这里先给出几个示例。

示例一，实现取字符串长度的功能：

```
In [65]: a="ace"
         print len(a)
Out[65]: 3
```

示例二，实现字符串的切割：

```
In [66]: a="student,my xxxx"
         b=a.split('t')
         print b
Out[66]: ['s', 'uden', ',my xxxx']
```

示例三，进行字符串的大写转化：

```
In [67]: a.upper()
Out[67]: 'STUDENT,MY XXXX'
```

2. 在 Python 中定义函数

要想在 Python 中使用自定义函数，就得先定义一个函数。对一个函数的定义包括两部分，一部分是声明这个指定的部分是函数，而不是其他对象；一部分是定义这个函数所包含的功能，也就是要编写这个函数的功能。

定义函数的格式如下：

```
def 函数名():
    函数内容;函数内容
    函数内容;函数内容
```

示例如下：

```
In [68]: def function1():
             a=9
             a+=8
             print a
In [69]: function1()
Out[69]: 17
```

这样就定义了一个 function1 函数，它没有参数，也没有返回值，仅仅打印出经过运算的 a 的值。

2.5.2 形参与实参

在 Python 中函数的功能是实现一项或多项功能，比如函数 len()的功能是取字符串的长度。但是，如果没有指明要取哪个字符串的长度，则仅有 len()是没有实际意义的。要让 len()有实际意义，就必须将某个字符串放进这个函数里。比如，我们要取字符串"abcdm"的长度，就要将"abcdm"放进 len()里，变成 len("abcdm")，这时 len()才有实际意义，括号里的"abcdm"就叫作函数的参数。

```
In [70]: a="abcdm"
         print len(a)
Out[70]: 5
```

我们可以发现，参数是函数在执行功能时要用到的数据。

1. 什么是形参

在 Python 中函数是有参数的，函数的参数有两种，一种是实参，另一种是形参。形参一般发生在函数定义的过程中，一般指参数的名称，而不代表函数的值，它只是形式上的函数，只表明在一个函数中的哪个位置有哪个名称的参数。下面通过示例来了解形参：

```
In [71]: def function(a,b):
             if a>b:
                print a
             else:
                print b
```

如上所示的函数的名称是 function，它有 a、b 两个形参。我们在调用这个函数时需要对其赋两个参数，分别赋值给 a 和 b，该函数的功能是打印出较大的值：

```
In [72]: function(4,5)
Out[72]: 5
```

2. 什么是实参

实参与形参刚好互补，一般在函数调用时出现，并且一般指参数的具体的值。下面通过示例了解什么是函数的实参，并总结函数的实参与形参的区别：

```
In [73]: def function1(a,b):
             if a>b:
                return 'a+b'
             else:
                return 'a-b'
          a=function1(1,3)
          print a
Out[73]: a-b
```

如上所示，函数的名称是 function1，在函数中有 a、b 两个形参，每次调用函数时都需要分别给 a、b 赋值，如果 a 大于 b，则返回字符串 a+b；如果 a 小于等于 b，则返回字符串 a-b。

在 a=function1(1,3)中，1 和 3 是实参，分别被赋给了 a、b，因为 1<3，所以输出的结果是 a-b。

3. 参数的传递

在 Python 中，函数在调用的过程中对参数的传递是有顺序的。下面讲解第 1 种参数的传递方式，也是最简单的传递方式，即按位置传递：

```
In [74]: def function(a=1,b=0):
             if a>b:
                return a
             else:
                return b
         max=function(7,8)
         print max
Out[74]: 8
In [75]: function()
Out[75]: 1
```

在函数 function 中有两个形参，并且两个形参的默认值为 a=1 及 b=0，在调用 function 时，需要给其赋两个值，分别给第 1 个位置（a）和第 2 个位置（b）。该函数的功能是输出两个位置中较大的值。

注意：因为 a、b 有默认的值，所以在不输入或者仅输入其中一个的参数时，函数的调用也不会报错。

下面讲解第 2 种参数的传递方式，即赋值传递:关键字参数。在 Python 的一个函数中出现多个参数时，我们可以通过参数的名字直接给我们的参数赋值，这些参数就叫作关键字参数。

如下所示，函数 function 有 3 个形参，分别是：add、b、c，它们的默认值都是 0。在调用 function 时需要给其赋 3 个值。function(a,c,b)表示将 a 的值赋给 add，将 b 的值赋给 c，将 c 的值赋给 b：

```
In [76]: a=9
         b=10
         c=1
         def function(add=0,b=0,c=0):
            print add
            print b
            print c
         function(a,c,b)
Out[76]: 9
         1
```

```
          10
```

在如下示例中，function(add=a,b=b,c=c)表示将 a 的值赋给 add，将 b 的值赋给 b，将 c 的值赋给 c：

```
In [77]: function(add=a,b=b,c=c)
Out[77]: 9
         10
         1
```

在如下示例中，function(b=7,add=8)表示将 8 赋给 add，将 7 赋给 b。虽然没有对 c 赋值，但是 c 的默认值为 0，因此也并没有报错：

```
In [78]: function(b=7,add=8)
Out[78]: 8
         7
         0
```

在如下示例中，function(c=2,b=3,add=5)表示将 5 赋给 add，将 3 赋给 b，将 2 赋给 c。在利用关键字参数进行赋值的过程中可以改变参数的位置：

```
In [79]: function(c=2,b=3,add=5)
Out[79]: 5
         3
         2
In [80]: function(b=4,c=2,add=1)
Out[80]: 1
         4
         2
In [81]: function(c=100)
Out[81]: 0
         0
         100
In [82]: '''但是要注意，在顺序被打乱后必须使用关键字参数'''
         function(b=2,c=3,2)
Out[82]:
  File "<ipython-input-13-327547dd23bd>", line 2
    function(b=2,c=3,2)
                   ^
SyntaxError: positional argument follows keyword argument
```

2.5.3 全局变量与局部变量

1. 什么是作用域

在 Python 中，一个变量是在一定的范围内起作用的，这个范围就叫作作用域。下面通过实例来讲解作用域：

```
In [83]: i=10                #这里定义一个全局变量 i
         print i
Out[83]: 10
In [84]: def func():
             i=8             #给局部变量 i 赋值
         func()
         print i
Out[84]: 10
```

可以看到输出的结果仍然是 10，这说明在 func 函数中对 i 的第 2 次赋值并没有改变全局中 i 的取值。

2. 局部变量

在 Python 中，作用域在一定范围内（而非全局内）起作用的变量叫作局部变量。在一个函数中，若没有对变量进行全局变量声明，则默认为一个局部变量。下面通过实例来认识局部变量：

```
In [85]: i=2                     #给全局变量 i 赋值
         def func2(a):
             i=7                 #给局部变量 i 赋值
             print i
         func2(1)
         print i
Out[85]: 7
         2
In [86]: func2(i)
Out[86]: 7
In [87]: print i
Out[87]: 2
```

如上所示的代码很形象地说明了全局变量和局部变量，在运行 func2 函数时会打印局部变量 i 的值 7，但是在打印全局变量 i 时输出的结果仍然是 2，这说明在函数内部对 i 的赋值并没有改变全局中 i 的值。

3. 全局变量

在 Python 中，如果想让某些变量的作用域为全局，也就是作用于程序的所有区域，就要对这个变量进行全局声明，在声明后这个变量就成了全局变量。下面通过实例来讲解全局变量：

```
In [88]: def func3():
             global i
             i=7
         a=func3()
         print a
         print i
Out[88]: None
         7
```

函数 func3 和 func2 的区别在于，在 func3 中多了 global i 这一行代码，这行代码的作用是在函数中给全局变量赋值。在以上代码中，在运行 func3 后再打印 i，输出的结果不再是 2，而是 7，这说明在运行 func3 后改变了全局变量 i 的值。

注：若要在函数中给全局变量赋值，就需要用 global 关键字声明。

2.5.4　对函数的调用与返回值

1. 对函数的调用

在程序运行时，若想执行某个函数，就需要调用这个函数。若想调用一个函数，则在定义函数之后，直接输入并运行一遍这个函数名即可；若想传递实参给这个函数，则直接在括号里输入实参即可。比如一个函数被定义为 def func3()，则在调用它时直接输入 func3(参数)即可，其中的参数可以省略：

```
In [89]: i=7
         def func3():
             i=1
         func3()                #调用函数
         print i
Out[89]: 7
```

注：因为定义的全局变量 i 的值为 7，所以最后打印的结果为 7。

2. 函数的返回值

在 Python 中有的函数是有返回值的，有的函数是没有返回值的。有返回值的函数可以返回一个值，也可以返回多个值，函数的返回值是通过 return 语句来实现的。

有一个返回值的情况如下：

```
In [90]: def test():
             i=7
             return i
         print test()
Out[90]: 7
```

有多个返回值的情况如下：

```
In [91]: def test2(i,j):
             k=i*j
             return (i,j,k)
         a=test2(4,5)
         print a
Out[91]: (4, 5, 20)
```

从如上所示的两个例子中可以看出，函数的返回值可以是一个或多个。

从如下所示的例子可以更清楚地看出，函数不但可以返回多个值，还可以将返回值直接赋给多个变量：

```
In [92]: def test2(i,j):
             k=i*j
             return i,j,k
         a,b,c = test2(4,5)
         print a,b,c
Out[92]: 4 5 20
```

2.5.5 文档字符串

在 Python 中可以定义很多函数，但是对于开发人员来说，如果遇到太多的函数而并不很了解各个函数的作用，就会感到很混乱。这时如果想了解各个函数的作用，就得从头分析，很浪费时间。针对这个问题，我们提供了两种解决方式：第 1 种是在开发时为每个函数写一个文档进行说明；第 2 种是在每个函数开头的地方加上一行说明性文字，我们将其称为文档字符串，这样，我们在编程时就可以很快明白这个函数的作用及对参数的设定

等，代码也会更加清晰。下面通过实例来讲解在 Python 中对文档字符串的使用：

```
In [93]: def d(i,j):
             '''这个函数实现一个乘法运算。
             函数会返回一个乘法运算的结果。'''
             k=i*j
             return k
         help(d)
Out[93]: Help on function d in module __main__:
d(i, j)
    这个函数实现一个乘法运算。
    函数会返回一个乘法运算的结果。
```

通过对文档字符串的说明，我们就可以明白这个函数的功能及所返回结果的属性。

2.6　Python 模块

2.6.1　认识 Python 模块

1. 什么是模块

函数是可以实现一项或多项功能的一段程序，模块便是对函数功能的扩展，是可以实现一项或多项功能的程序块。其实，函数是一段程序，模块是一项程序块，函数和模块都是用来实现功能的，但是模块的作用范围比函数要广，在模块里可以包含多个函数。

2. 如何导入模块

在 Python 的一段程序中要使用某个模块，就必须先导入指定的模块，只有导入了某个模块，才能使用该模块。下面通过实例来讲解如何导入模块。

（1）通过 import 语句导入整个模块：

```
import module1, module2....  # 建议一个 import 语句只导入一个模块
import module as module_mo # 别名（也就是自定义的模块名称空间）
```

代码示例如下：

```
In [94]: import pandas
In [95]: import pandas as pd
```

（2）通过 from-import 语句导入指定模块的部分属性或进行模糊导入，具体细节请参见 2.6.3 节：

```
from module import name1,name2....
```

代码示例如下：

```
In [96]: from pandas import Series
```

3. sys 模块

Python 自带一些模块，我们把这些模块叫作标准库模块。标准库模块是指一类模块，而不是特指某一种模块。在标准库模块中也有非常多的模块，有的与电子邮件相关，有的与操作系统相关，有的与系统功能相关，等等。我们把在标准库中与系统功能有关的这些模块叫作 sys 模块。下面通过实例来讲解 sys 模块的基本使用方法：

```
In [97]: import sys
In [98]: print sys.argv[0]
D:/anaconda3/Scripts/ipython-script.py
In [99]: sys.getdefaultencoding()
Out[99]: 'utf-8'
```

sys 模块的常见函数列表如表 2-3 所示。

表 2-3

函　数	功　能
sys.argv	实现从程序外部向程序传递参数
sys.exit([arg])	实现在程序的中间退出，arg=0 为正常退出
sys.getdefaultencoding()	用于获取系统当前的编码，一般默认为 ascii
sys.setdefaultencoding()	设置系统默认的编码，在执行 dir（sys）时不会看到这个方法。在解释器中执行不通过时，可以先执行 reload(sys)，再执行 setdefaultencoding('utf-8)，将系统的默认编码设置为 utf-8
sys.getfilesystemencoding()	用于获取文件系统使用的编码方式，在 Windows 下返回'mbcs'，在 Mac 下返回'utf-8'
sys.path	用于获取指定的模块搜索路径的字符串集合，可以将写好的模块放在所得到的某个路径下，这样就可以在程序中 import 时正确找到
sys.platform	用于获取当前的系统平台
sys.stdin、sys.stdout、sys.stderr	stdin、stdout 及 stderr 变量包含与标准 I/O 流对应的流对象。如果需要更好地控制输出，而 print 不能满足我们的要求，就可以使用这些变量

2.6.2　from…import 详解

1. 学会使用 from…import

若想在 Python 中导入一个模块，则可以使用 import，但是 import 导入了这个模块的全部功能，并没有指定具体模块的某个属性或方法。若不仅想导入一个模块，还想导入模块中对应的某个功能，则可以使用 from…import 语句。下面通过实例来了解 from…import 语句的用法：

```
    In [100]: from pandas import DataFrame
              df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
'data1':range(7)})
              df1.head(2)
    Out[100]:
       data1 key
    0      0   b
    1      1   b
```

2. 学会使用 from…import *

在 Python 中使用 from…import 语句时，只能一次导入一个模块的一个功能，若想一次性导入这个模块的所有功能，也就是导入所有属性与方法，则可以使用 from…import* 语句。下面通过实例来了解 from …import *语句的用法：

```
    In [101]: from pandas import *
              df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
'data1': range(7)})
              df1.head(2)
    Out[101]:
       data1 key
    0      0   b
    1      1   b
```

通过对比 import 与 from …import …这两种导入模块的方法，我们可以清楚地发现 import…as…比 from…import *更简洁、高效，因为 from…import *看似简单，但需要用户牢记模块中的具体函数，本书也不推荐使用这种方法。

2.6.3 认识__name__属性

1. 认识主模块

在 Python 函数中，如果一个函数调用其他函数完成一项功能，这个函数就叫作主函数；如果一个函数没有调用其他函数，这个函数就叫作非主函数。模块也是如此，如果一个模块是被直接使用的，没有被其他模块调用，这个模块就叫作主模块；如果一个模块被其他模块调用，这个模块就叫作非主模块。

2. 认识__name__属性

在 Python 中有主模块与非主模块之分，那么如何区分主模块与非主模块呢？答案是：如果一个模块的__name__属性的值是__main__，那么这个模块是主模块，否则就是非主模块。其实我们可以将__name__看作一个变量，这个变量是系统给出的，其功能是判断一个模块是否是主模块。

3. 对__name__属性的使用

首先，我们分别看看这个模块在不同场景下的__name__的值：

```
In [102]: print __name__
Out[102]: __main__
```

然后，看看__name__属性的常用情况：

```
In [103]: if __name__=="__main__":
              print "It's main"
          else:
              print "It's not main"
Out[103]: It's main
```

2.6.4 自定义模块

Python 在安装时就自带的模块叫作系统自带模块，这些模块不需要用户自己定义和编写。而有些模块刚好与这种模块不同，需要我们自己定义和编写，这些模块就叫作自定义模块。下面通过实例进行讲解：

```
    def add(i,j):
        k=i+j
        return k
```

将这段语句保存在 myadd.py 文件中，并保存在“….\Anaconda\Lib”目录下就成为模块。注意：“….”为我们安装时的 Anaconda 的地址。

下面调用自定义模块 myadd：

```
    In [104]: import myadd as md     #调用自定义模块 myadd
    In [105]: s = md.add(1,3)
    In [106]: s
    Out[106]: 4
```

2.6.5　dir()函数

1. 认识 dir()函数

在 Python 中有很多模块，有时我们会忘记一个模块有哪些功能，这时可以通过 dir()函数来查看指定模块的功能列表。下面通过实例进行讲解：

```
    In [107]: import pandas
    In [108]: dir(pandas)
    Out[108]:
        ['Categorical', 'CategoricalIndex', 'DataFrame', 'DateOffset',
'DatetimeIndex', 'ExcelFile', 'ExcelWriter', 'Expr', 'Float64Index', 'Grouper',
'HDFStore', 'Index', 'IndexSlice', 'Int64Index', 'MultiIndex', 'NaT', 'Panel',
'Panel4D', 'Period', 'PeriodIndex', 'RangeIndex', 'Series', 'SparseArray',
'SparseDataFrame', 'SparseList', 'SparseSeries', 'SparseTimeSeries', 'Term',
'TimeGrouper', 'TimeSeries', 'Timedelta', 'TimedeltaIndex', 'Timestamp',
'WidePanel', '__builtins__', '__cached__', '__doc__', '__docformat__',
'__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__',
'__version__', '_join', '_np_version_under1p10', '_np_version_under1p11',
'_np_version_under1p12', '_np_version_under1p8', '_np_version_under1p9', ……]
```

注意：鉴于 Pandas 的方法太多，这里并没有全部列出来。

2. dir()函数扩展详解

在 Python 中，通过 dir()函数不仅能查看模块的功能列表，还能查看任意指定对象的

功能列表。如下所示为定义 a 为一个列表，并通过 dir(a)查看 a 的功能列表：

```
In [109]: a = [1]
          dir(a)
Out[109]:
    ['__add__','__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

2.7 Python 异常处理与文件操作

2.7.1 Python 异常处理

异常指一个事件，该事件会在程序执行的过程中发生，从而影响程序的正常执行，我们将这种状态称为 Python 异常。在 Python 中一切皆对象，异常也是 Python 的对象，表示一个错误。当 Python 脚本发生异常时，我们需要捕获并处理它，否则程序会终止执行。我们通常会把异常的类型和位置打印出来，以便程序员处理：

```
In [110]: print 字符串需要被引用
Out[110]: File "<ipython-input-3-79d7652315a8>", line 1
          print 字符串需要被引用
                ^
          SyntaxError: invalid syntax
```

在上述实例中，由于 Python 打印字符串时需要引用内容，所以会报错，解决方法为在输出内容前加“”，代码如下：

```
In [111]: print "字符串需要被引用"
Out[111]: 字符串需要被引用
```

假如我们在编程时知道某些语句可能会导致某种错误的发生，如果不想在异常发生时结束程序，则只需在 try 里捕获它，然后使用 except 语句处理可能出错的部分。

对 Python 异常的处理如下：

```
'''：
    try…except…else 的语法格式为：
    try:
        #运行识别的代码
    except:
        #如果 try 部分引发了异常，则执行此处的代码
    else:
        #如果都没有发生异常
'''
```

下面通过实例来讲解 try…except…else 的用法：

```
In [112]: #i=1
          try:
              print i
          except :     #这里一定要指明异常的类型
              i=9
              i+=10
              print "刚才 i 没定义，处理了异常之后，i 的值为："+str(i)
Out[112]: 刚才 i 没定义，处理了异常之后，i 的值为：19
```

当 i=1 为备注状态时，会执行 except 中的代码：

```
In [113]: #处理多种异常
          i=10
          #j="a"
          try:
              print i+j
          except NameError:
              i=j=0
              print "刚刚 i 或 j 没有进行初始化数据，现在我们将其都初始化为 0，结果是："
              print i+j
          except TypeError:
          print "刚刚 i 与 j 类型对应不上，我们转换一下类型即可处理异常，处理后：结果
是："+str(i)+str(j)
Out[113]:
          刚刚 i 或 j 没有进行初始化数据，现在我们将其都初始化为 0，结果是：
          0
```

常见的 Python 异常如表 2-4 所示。

表 2-4

异常的名称	描　　述
ArithmeticError	所有数值计算错误的基类
AssertionError	断言语句失败
AttributeError	对象没有这个属性
BaseException	所有异常的基类
DeprecationWarning	关于被弃用的特征的警告
EnvironmentError	操作系统错误的基类
EOFError	没有内建输入，到达 EOF 标记
Exception	常规错误的基类
FloatingPointError	浮点计算错误
FutureWarning	关于构造将来语义会有改变的警告
GeneratorExit	生成器（generator）发生异常来通知退出
ImportError	导入模块/对象失败
IndentationError	缩进错误
IndexError	在序列中没有此索引（index）
IOError	输入/输出操作失败
KeyboardInterrupt	用户中断执行（通常是输入^C）
KeyError	在映射中没有这个键
LookupError	无效数据查询的基类
MemoryError	内存溢出错误
NameError	未声明/初始化对象（没有属性)
NotImplementedError	尚未实现的方法
OSError	操作系统错误
OverflowError	数值运算超出最大限制
OverflowWarning	旧的关于自动提升为长整型（long）的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
ReferenceError	弱引用（Weak reference）试图访问已被垃圾回收的对象
RuntimeError	一般的运行时错误
RuntimeWarning	关于可疑的运行时行为（runtime behavior）的警告
StandardError	所有的内建标准异常的基类

续表

异常的名称	描　　述
StopIteration	迭代器没有更多的值
SyntaxError	Python 语法错误
SyntaxWarning	关于可疑语法的警告
SystemError	一般的解释器系统错误
SystemExit	解释器请求退出
TabError	Tab 和空格混用
TypeError	对类型无效的操作
UnboundLocalError	访问未初始化的本地变量
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时的错误
UnicodeError	Unicode 相关的错误
UnicodeTranslateError	Unicode 转换时的错误
UserWarning	用户代码生成的警告
ValueError	传入无效的参数
Warning	警告的基类
WindowsError	系统调用失败
ZeroDivisionError	除（或取模）零（所有数据类型）

2.7.2　异常的发生

上面提到，某些程序若出现某个异常，则在执行时会自动将该异常的类型和发生的位置打印出来。但是，这些错误的类型是系统已经定义好的，除了系统自定义的异常，我们也可以自定义异常。

自定义异常是指在某种情况下才引发某种异常，比如变量未被定义就拿来使用，引发自定义的 A 异常，这个过程就被叫作异常的引发。自定义的异常应通过直接或间接的方式继承 Exception 类。在 Python 中，要想实现在某种情况下引发某种自定义异常的功能，就可以使用 raise 语句。下面通过实例进行讲解：

```
In [114]: i=8
          print i
```

```
            if i>7:
                print 9
                raise NameError
                print 10
Out[114]:
            8
            9
            ---------------------------------------------------------------
            NameError                          Traceback (most recent call last)
            <ipython-input-5-c740ebd6a5c9> in <module>()
                 5 if i>7:
                 6     print 9
            ----> 7     raise NameError
                 8     print 10

            NameError:
```

自定义一个异常类的实例如下：

```
In [115]: class SoSError(Exception):                    #按照命名规范，以 Error 结尾，
并且自定义异常需要继承 Exception 类
                def __init__(self):
                    Exception.__init__(self)
            try:
                i=8
                if i>7:
                    raise SoSError()
            except SoSError,a:
            print "SoSError:我错了"
Out[115]: SoSError:我错了
```

2.7.3 try…finally 的使用

无论是否发生异常，try…finally 语句都将执行最后的代码：

```
'''
    #try…finally 语句的语法格式为：
    try:
        #运行识别的代码
    finally:
```

```
        #退出时总会执行的语句
'''
```

下面通过实例讲解对 try…finally 语句的使用：

```
In [116]: #假如要实现不管中间是否发生异常，都要输出 hello world
          try:
             print i
          finally:
             print "不管上面是否异常，我必须输出 hello world! "
Out[116]:
          不管上面是否异常，我必须输出 hello world!
          ---------------------------------------------------------------
          NameError                 Traceback (most recent call last)
          <ipython-input-10-88d42e4abf8f> in <module>()
               2 #假如要实现不管中间是否发生异常，都要输出 hello world
               3 try:
          ----> 4     print i
               5 finally:
               6     print "不管上面是否异常，我必须输出 hello world! "

          NameError: name 'i' is not defined
```

2.7.4　文件操作

我们可以使用 Python 程序直接自动操作某文件，例如：创建文件、打开文件、关闭文件、将指定的内容写入文件、读取文件、关闭文件等。

下面通过实例来讲解在 Python 中对文件的操作：

```
In [117]: #创建某个文件
          import os
          os.mkdir(r"d:/newdir")
```

下面通过实例来说明如何对一个文件进行写入和关闭文件。

第 1 步，做好内容：

```
In [118]: content='''我是文件内容
          内容是
          如何做好 Python 量化投资
          '''
```

第 2 步，创建文件：

```
In [119]: file=open(r"d:\newdir\test.txt",'w')  #w 参数，直接打开一个文件，不存在则创建文件
```

第 3 步，写入内容及关闭：

```
In [120]: file.write(content)
          file.close()
```

读取文件，关键点为先打开文件，再进入 while 循环依次读取每行：

```
In [121]: fr=open(r"d:\newdir\py1.txt")
          while True:
              line=fr.readline()
              if len(line)==0:
                  break
              print line
          fr.close
Out[121]:
          我是文件的内容

          内容是

          如何做好 Python 量化投资
```

另一种读取方法如下：

```
In [122]: fr=open(r"d:\newdir\test.txt",'r')
          for u in fr:
              print u
          fr.close
Out[122]:
          我是文件的内容

          内容是

          如何做好 Python 量化投资
```

同时，Python 提供了很多文件处理包，os 模块就是很强大的文件处理包。例如，os.getcwd()用于获得当前的工作目录，os.remove()用于删除一个文件，等等。Os 模块有很多强大的功能，我们会在后续的 Python 进阶内容中介绍。

第 3 章

Python 进阶

3.1 NumPy 的使用

NumPy 是高性能科学计算和数据分析的基础包，是本章接下来要讲解的 Pandas、Scikit-Learn、StatModels 等库的构建基础。NumPy 的主要功能在第 1 章中已经有所介绍，对于一般的金融数据分析而言，其大部分功能可由更高级的 Pandas 实现。我们并不需要在学习 NumPy 上花费过多精力，但是回过头来深入学习 NumPy 可以帮助我们更好地理解面向数组编程的思维方式。

在本书中对 NumPy 的引入约定为：

```
In [1]: import numpy as np
```

因此，一旦在代码中看到 np，就是使用了 NumPy。

3.1.1 多维数组 ndarray

NumPy 的主要对象是 ndarray，该对象是一个快速、灵活的大数据容器。在此需要注

意，在 ndarray 与 Python 中内置的 list、tuple 并不相同：在 Python 中，元素的数据类型可以不同；而在 ndarray 中，所有元素的数据类型必须相同。

首先，我们利用 NumPy 中的 array 函数创建一个一维 ndarray：

```
In [2]:data = [1, 2, 3, 4]
In [3]:arr = np.array(data)
       arr
Out[3]:array([1, 2, 3, 4])
```

等长序列组成的列表将会被转换为一个多维数组：

```
In [4]:data1 = [data, data]
In [5]:arr1 = np.array(data1)
       arr1
Out[5]:array([[1, 2, 3, 4],
              [1, 2, 3, 4]])
```

除了 np.array 函数，还有其他函数可以快速创建数组：

```
In [6]:np.zeros((3, 3))  #创建全 0 的数组
Out[6]:array([[ 0.,  0.,  0.],
              [ 0.,  0.,  0.],
              [ 0.,  0.,  0.]])
In [7]:np.ones((3, 3))  #创建全 1 的数组
Out[7]:array([[ 1.,  1.,  1.],
              [ 1.,  1.,  1.],
              [ 1.,  1.,  1.]])
In [8]:np.arange(1, 10, 2)  #创建 1～10 且差为 2 的等差数列
Out[8]:array([1, 3, 5, 7, 9])
In [9]:np.linspace(1, 10, 4)  #创建 1～10 且长度为 4 的等差数列
Out[9]:array([  1.,   4.,   7.,  10.])
```

3.1.2 ndarray 的数据类型

dtype 代表 ndarray 中元素的数据类型，数据类型有很多，对于新手来说，要全部记住这些数据类型是十分困难的。不过没有关系，在大部分数据分析工作中，我们只需要知道 float（浮点数）、int（整数）、bool（布尔值）、string_（字符串）、unicode_就可以了。若想了解更多的数据类型，则可以查阅 NumPy 的文档。

接下来举几个例子，代码如下：

```
In [10]:arr2 = np.array([1, 2, 3, 4])
        arr2.dtype
Out[10]:dtype('int32')
In [11]:arr3 = np.array([1.1, 2, 3, 4])
        arr3.dtype
Out[11]:dtype('float64')
In [12]:arr4 = np.array(['量', '化', '分', '析'])  #长度为 1 的 Unicode
        arr4.dtype
Out[12]:dtype('<U1')
```

在不同的数据类型之间也能互相转换：

```
In [13]:float_arr2 = arr2.astype(np.float64)
        float_arr2
Out[13]:array([ 1.,  2.,  3.,  4.])
In [14]:int_arr3 = arr3.astype(np.int32)
        int_arr3
Out[14]:array([1, 2, 3, 4])
In [15]:u_arr3 = arr3.astype(np.unicode_)
        u_arr3
Out[15]:array(['1.1', '2.0', '3.0', '4.0'], dtype='<U32')
```

3.1.3　数组索引、切片和赋值

索引切片的操作类似于 Python 的内置函数 list，只不过是从一维拓展到多维而已。

首先，我们创建一个 3×3 的 ndarray：

```
In [16]:arr4 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
        arr4
Out[16]:array([[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]])
In [17]:arr4[1]
Out[17]:array([4, 5, 6])
In [18]:arr4[1, 1]
Out[18]:5
In [19]:arr4[:2]
Out[19]:array([[1, 2, 3],
               [4, 5, 6]])
In [20]:arr4[:2, :2]
```

```
Out[20]:array([[1, 2],
               [4, 5]])
```

需要注意的是，在将一个标量赋值给一个切片时，该值会被广播到整个切片视图上，并直接对源数组进行修改。如果我们想避免改变源数组，则可以使用 copy 函数：

```
In [21]:arr4[:2, :2] = 10
         arr4
Out[21]:array([[10, 10,  3],
               [10, 10,  6],
               [ 7,  8,  9]])
In [22]:arr4.copy()[:2, :2] = 5
         arr4
Out[22]:array([[10, 10,  3],
               [10, 10,  6],
               [ 7,  8,  9]])
```

3.1.4 基本的数组运算

ndarray 的一大特色就是可以将代码向量化。所谓向量化，就是对一个复杂的对象进行整体操作，而不是对其中的单个元素进行循环。NumPy 的大部分代码都是由 C 语言编写的，并且进行了高度优化，这大大加快了计算速度。

简单的数学运算可以直接在 ndarray 上运行，例如：

```
In [23]:arr5 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
         arr5
Out[23]:array([[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]])
In [24]:arr5 + arr5
Out[24]:array([[ 2,  4,  6],
               [ 8, 10, 12],
               [14, 16, 18]])
In [25]:arr5 * arr5
Out[25]:array([[ 1,  4,  9],
               [16, 25, 36],
               [49, 64, 81]])
In [26]:arr5 * 2
Out[26]:array([[ 2,  4,  6],
               [ 8, 10, 12],
               [14, 16, 18]])
```

```
In [27]:arr5 ** 0.5
Out[27]:array([[ 1.        ,  1.41421356,  1.73205081],
               [ 2.        ,  2.23606798,  2.44948974],
               [ 2.64575131,  2.82842712,  3.        ]])
```

在 NumPy 中还有一些内置的数学函数，可以帮助我们快速地对数据进行统计计算，比如 sum、max、mean、std 等。

```
In [28]:arr5.sum()
Out[28]:45
In [29]:arr5.std()
Out[29]:2.5819888974716112
In [30]:arr5.mean()
Out[30]:5.0
In [31]:arr5.max()
Out[31]:9
```

表 3-1 列出了基本的数组统计方法。

表 3-1　基本的数组统计方法

方　法	说　明
sum	对数组中的全部或某个抽向元素进行求和
mean	算术平均数
std、var	标准差、方差
min、max	最大值、最小值
argmin、argmax	最大值索引、最小值索引
cumsum、cumprod	所有元素的累计和、累计积

我们也可以自定义函数，再将其传入 ndarray 中：

```
In [32]:def f(x):
            return x ** 2
In [33]:f(arr5)
Out[33]:array([[ 1,  4,  9],
               [16, 25, 36],
               [49, 64, 81]])
```

3.1.5　随机数

NumPy 还可以用来生成伪随机数，负责这一功能的是其子库 numpy.random。

首先，导入 numpy.random。为了更直观地展示随机数的生成过程，这里同时导入了 Matplotlib，以将结果可视化：

```
In [34]:import numpy.random as npr
        import matplotlib.pyplot as plt
        %matplotlib inline
```

例如，npr.rand 函数可以用来生成[0,1)的随机多维数组：

```
In [35]:npr.rand(3, 2)
Out[35]:array([[ 0.75531867,  0.96373173],
               [ 0.35314904,  0.33969993],
               [ 0.58071311,  0.74715197]])
```

稍微开动一下大脑，就可以将随机区间转化为[2,4)：

```
In [36]:a = 2
        b = 4
        npr.rand(3, 2) * (b - a) + a
Out[36]:array([[ 2.97882682,  2.11641403],
               [ 2.17445139,  2.02036574],
               [ 3.05646641,  2.60940393]])
```

表 3-2 展示了生成简单随机数的函数及其参数和描述。

表 3-2

函　数	参　数	描　述
rand	d0, d1, ..., dn	生成半开区间[0, 1)内的多维随机数
randn	d0, d1, ..., dn	生成来自标准正态分布的多个样本
randint	low[, high, size]	生成半开区间[low, high)内的随机整数
choice	a[, size, replace, p]	生成在给定的一维数组中的随机样本
bytes	length	生成随机字节

然后，对表 3-2 中部分函数生成的随机数进行可视化，效果如图 3-1 所示。

```
In [37]:size = 1000
        rn1 = npr.rand(size, 2)
        rn2 = npr.randn(size)
        rn3 = npr.randint(0, 10, size)
        rang = [0, 10, 20, 30, 40]
        rn4 = npr.choice(rang, size = size)
In [38]:fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows = 2, ncols = 2,
```

```
figsize = (10, 10))
            ax1.hist(rn1, bins = 25, stacked = True)
            ax1.set_title('rand')
            ax1.set_ylabel('frequency')
            ax1.grid(True)
            ax2.hist(rn2, bins = 25)
            ax2.set_title('randn')
            ax2.grid(True)
            ax3.hist(rn3, bins = 25)
            ax3.set_title('randint')
            ax3.set_ylabel('frequency')
            ax3.grid(True)
            ax4.hist(rn4, bins = 25)
            ax4.set_title('choice')
            ax4.grid(True)
```

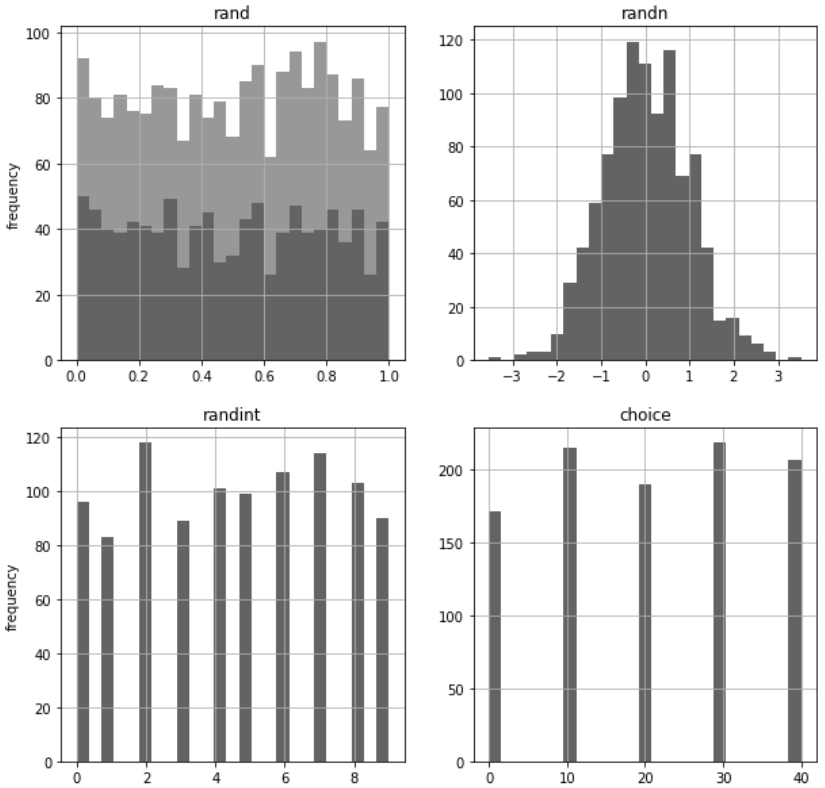


图 3-1

许多金融模型，例如 BSM 模型、跳跃扩散模型、平方根扩散模型等，都依赖于正态分布。我们可以通过生成相应的随机数，来将原本连续的金融模型离散化，从而进行近似模拟。在 npr.random 模块中内置了很多分布函数，这里限于篇幅不再全面展示，感兴趣的读者可以自行查阅 NumPy 的文档。

作为例子，我们将如下分布的随机数进行可视化。

（1）n=100、P=0.3 的二项分布。

（2）均值为 10、标准差为 2 的正态分布。

（3）自由度为 0.5 的卡方分布。

（4）λ 为 2 的泊松分布。

相应的代码如下：

```
    In [39]:rn5 = npr.binomial(100, 0.3, size)
            rn6 = npr.normal(10, 20, size)
            rn7 = npr.chisquare(0.5, size)
            rn8 = npr.poisson(2.0, size)
    In [40]:fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows = 2, ncols = 2,
figsize = (10, 10))
            ax1.hist(rn5, bins = 25)
            ax1.set_title('binomial')
            ax1.set_ylabel('frequency')
            ax1.grid(True)
            ax2.hist(rn6, bins = 25)
            ax2.set_title('normal')
            ax2.grid(True)
            ax3.hist(rn7, bins = 25)
            ax3.set_title('chisquare')
            ax3.set_ylabel('frequency')
            ax3.grid(True)
            ax4.hist(rn8, bins = 25)
            ax4.set_title('poisson')
            ax4.grid(True)
```

其可视化效果如图 3-2 所示。

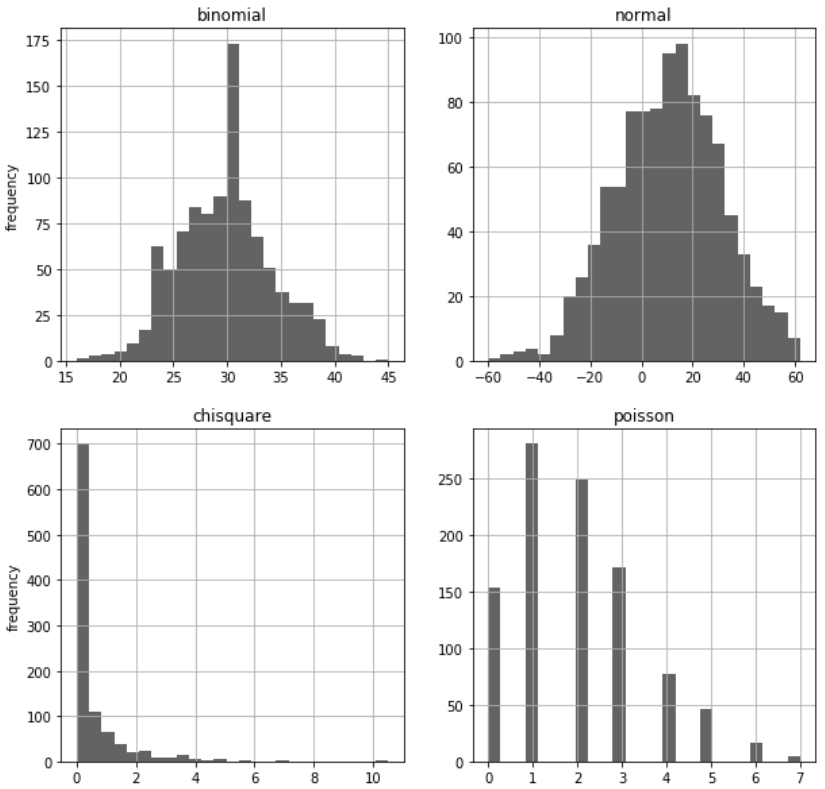


图 3-2

3.2　Pandas 的使用

Pandas 是基于 NumPy 衍生出的一种工具，用于解决数据分析问题，它纳入了大量的库和一些标准的数据模型，提供了可用于高效操作大型数据集的工具，是使 Python 成为强大而高效的数据分析工具的重要因素之一。

对金融数据的大部分处理是依赖 Pandas 实现的。在本书中对 Pandas 的引入约定为：

```
import pandas as pd
```

一旦在代码中看到 pd，就是使用了 Pandas。

3.2.1 Pandas 的数据结构

Pandas 的数据结构主要分为三种：Series（一维数组）、DataFrame（二维的表格型数据结构）和 Panel（三维数组）。

1. Series

Series 指一维数组，与 NumPy 中的一维 Array 类似。Series、Array 与 Python 基本的数据结构 List 也很相近，其区别是：在 List 中的元素可以是不同的数据类型，而在 Array 和 Series 中则只允许存储相同的数据类型，这样可以更有效地使用内存，提高运算效率。

Series 增加了对应的标签（label）以用于索引，可以包含 0 个或者多个任意数据类型的实体。其中，标签索引赋予了 Series 强大的存取元素功能。除通过位置外，Series 还允许通过索引标签进行元素存取：

```
In [41]: obj = pd.Series([40, 12, -3, 25])
In [42]: obj
Out[42]:
0    40
1    12
2    -3
3    25
```

Series 的字符串表现形式为：索引在左边，值在右边。因为在建立过程中没有指定索引，所以 Python 会自动为我们加入一个 0～n 的整数索引，我们可以通过数字获取具体位置上的元素：

```
In [43]: obj[0]
Out[43]: 40
```

可以通过 index 与 values 获取 Series 的索引与数据：

```
In [44]: obj.index
Out[44]: RangeIndex(start=0, stop=4, step=1)
In [45]: obj.values
Out[45]: array([40, 12, -3, 25], dtype=int64)
```

当然，也可以在 Series 建立时就指定索引：

```
In [46]: obj = pd.Series([40, 12, -3, 25],index=['a','b','c','d'])
```

```
In [47]: obj
Out[47]:
a    40
b    12
c    -3
d    25
```

可以直接通过索引获取数值：

```
In [48]: obj['c']
Out[48]: -3
```

对于 Series 的各种计算，其结果也会保留 index：

```
In [49]: obj[obj>15]
Out[49]:
a    40
d    25
```

在 Python 中一切皆对象，Series 也不例外。可通过 Tab 键补全查看一个 Series 的方法，即输入前几个字母，按下 Tab 键，代码如下：

```
In [50]: obj.describe()
Out[50]:
count     4.000000
mean     18.500000
std      18.339393
min      -3.000000
25%       8.250000
50%      18.500000
75%      28.750000
max      40.000000

In [51]: obj.mean()
Out[51]: 18.5
```

另外，Series 可以被转换为字典：

```
In [52]: obj.to_dict()
Out[52]: {'a': 40, 'b': 12, 'c': -3, 'd': 25}
```

2. DataFrame

DataFrame 指二维的表格型数据结构。在 DataFrame 有很多功能与 R 中的 data.frame

类似，我们可以将 DataFrame 理解为 Series 的容器，也就是说，在 DataFrame 中，多个 Series 共用了一个索引 index。

在以字典或 Series 的字典的结构构建 DataFrame 时，最外面的字典对应 DataFrame 的列，内嵌的字典及 Series 则是其中的每个值，例如：

```
    In [53]: d = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),'two' :
pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
    In [54]: df = pd.DataFrame(d)
    In [55]: df
    Out[55]:
       one  two
    a  1.0  1.0
    b  2.0  2.0
    c  3.0  3.0
    d  NaN  4.0
```

从字典的列表中构建 DataFrame 时，其中的每个字典代表的是每条记录（DataFrame 中的一行），字典中每个值对应的是这条记录的相关属性。

同时可以看到，当由多个 Series 组成 DataFrame 时，Pandas 会自动按照 index 对齐数据，如果某个 Series 的 index 缺失，则 Pandas 会将其自动填写为 np.nan。

3.2.2 Pandas 输出设置

在 Pandas 中可以通过 set_option 设置 Pandas 的输出格式，例如最多显示的行数、列数等：

```
    In [56]: import pandas as pd
    In [57]: pd.set_option("display.max_rows",1000)
    In [58]: pd.set_option("display.max_columns",20)
    In [59]: pd.set_option('precision',7)
    In [60]: pd.set_option('large_repr', 'truncate')
```

3.2.3 Pandas 数据读取与写入

Pandas 可以方便地读取本地文件如 csv、txt、xlsx 等，例如：

```
    In [61]: a=pd.read_csv('closeprice.csv',encoding='gbk',dtype={'ticker':
```

```
str})
    In [62]: a
    Out[62]:
```

其结果截图如图 3-3 所示。

	Unnamed: 0	ticker	secShortName	tradeDate	closePrice
0	0	000001	平安银行	2017-06-20	9.12
1	1	000002	万科A	2017-06-20	21.03
2	2	000004	国农科技	2017-06-20	27.03
3	3	000005	世纪星源	2017-06-20	5.45
4	4	000006	深振业A	2017-06-20	8.87
5	5	000007	全新好	2017-06-20	15.87

图 3-3

我们可自行输入 pd.read_，使用 Tab 自动完成功能查看 Pandas 可以读取的数据类型。

同理，可以使用 to_，将 DataFrame 输出到文件中：

```
    In [63]: a.to_excel('closeprice.xls')
```

3.2.4　数据集快速描述性统计分析

在读入数据后，我们需要对数据做基本的统计分析，代码如下：

```
    In [64]: import pandas as pd
    In [65]: data = pd.read_csv('closeprice.csv',encoding='gbk')
    In [66]: data.describe().T
    Out[66]:
               count  mean       std        min   25%     50%    75%    max
    Unnamed: 0 6.0    2.5000000  1.8708287  0.00  1.2500  2.500  3.75   5.00
    ticker     6.0    4.1666667  2.3166067  1.00  2.5000  4.500  5.75   7.00
    closePrice 6.0    14.5616667 8.2950550  5.45  8.9325  12.495 19.74  27.03
```

可以加入参数 include ='all'，这样 describe 就会展示全部列的统计结果。当然，我们在这里没必要对 ticker 进行均值统计。df.describe()会统计出各列的计数、平均数、方差、最小值、最大值及 quantile 数值。df.info()会展示数据类型、行列数和 DataFrame 占用的内存：

```
In [67]: data.info()
   <class 'pandas.core.frame.DataFrame'>
   RangeIndex: 6 entries, 0 to 5
   Data columns (total 5 columns):
   Unnamed: 0     6 non-null int64
   ticker         6 non-null int64
   secShortName    6 non-null object
   tradeDate      6 non-null object
   closePrice     6 non-null float64
   dtypes: float64(1), int64(2), object(2)
   memory usage: 312.0+ bytes
```

3.2.5 根据已有的列建立新列

本节将根据现有的代码列，加入该股票代码对应的行业列：

```
In [68]: a=pd.read_csv('closeprice.csv',encoding='gbk')
In [69]: a
Out[69]:
   Unnamed: 0       ticker      secShortName    tradeDate       closePrice
   0        0       1           平安银行        2017-06-20      9.12
   1        1       2           万科 A          2017-06-20      21.03
   2        2       4           国农科技        2017-06-20      27.03
   3        3       5           世纪星源        2017-06-20      5.45
   4        4       6           深振业 A        2017-06-20      8.87
   5        5       7           全新好          2017-06-20      15.87
```

使用 map 函数映射一个字典即可：

```
In [70]: b={1:'银行',2:'房地产',4:'医药生物',5:'房地产',6:'采掘',7:'休闲服务
',8:'机械设备'}
In [71]: a['ind']=a.ticker.map(b)
In [72]: a
Out[72]:
   Unnamed: 0   ticker  secShortName   tradeDate       closePrice  ind
   0        0   1       平安银行       2017-06-20      9.12        银行
   1        1   2       万科 A         2017-06-20      21.03       房地产
   2        2   4       国农科技       2017-06-20      27.03       医药生物
   3        3   5       世纪星源       2017-06-20      5.45        房地产
   4        4   6       深振业 A       2017-06-20      8.87        采掘
   5        5   7       全新好         2017-06-20      15.87       休闲服务
```

3.2.6　DataFrame 按多列排序

可以使用 list 给出需要排序的列，同时给出是升序还是降序：

```
    In [73]: import pandas as pd
             data = pd.DataFrame({'group': ['a', 'a', 'a', 'b','b', 'b', 'c',
'c','c'], 'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
             data.sort_values(by=['group','ounces'], ascending=[False, True],
inplace=True)
    In [74]: data
    Out[74]:
      group  ounces
    6     c     3.0
    7     c     5.0
    8     c     6.0
    3     b     6.0
    4     b     7.5
    5     b     8.0
    1     a     3.0
    0     a     4.0
    2     a    12.0
```

上述代码的作用就是先按照 group 降序排列，当 group 相同时再按照 ounces 升序排列。参数中的 inplace=True 直接将排序后的结果存在 data，即直接用排序好的数据覆盖原始数据。

3.2.7　DataFrame 去重

在大多数时候，在数据中会有重复的数据，在做分析前需要进行去重：

```
    In [75]: data = pd.DataFrame({'k1': ['one'] * 3 + ['two'] * 4, 'k2': [3, 2,
1, 3, 3, 4, 4]})
    In [76]: data
    Out[76]:
        k1  k2
    0  one   3
    1  one   2
    2  one   1
    3  two   3
    4  two   3
    5  two   4
    6  two   4
```

```
In [77]: data.drop_duplicates()
Out[77]:
   k1  k2
0  one   3
1  one   2
2  one   1
3  two   3
5  two   4
```

在不加任何参数时，Pandas 会将完全相同的行去重：

```
In [78]: data.drop_duplicates(subset=['k1'],keep='last')
Out[78]:
   k1  k2
2  one   1
6  two   4
```

当设置 subset 为 k1 时，只要 k1 重复，Pandas 就认为是重复的，可以通过 keep 参数确定需要保留哪个，一般在使用 keep 时先排序。

另外，如果需要查看重复的行，则可以进行如下操作：

```
In [79]: data[data.duplicated()]
Out[79]:
   k1  k2
4  two   3
6  two   4
```

3.2.8 删除已有的列

如果我们不再需要某一列，就可以对该列进行删除操作：

```
In [80]: a=pd.read_csv('closeprice.csv',encoding='gbk')
In [81]: a.drop(['Unnamed: 0'],axis=1)
Out[81]:
        Ticker      secShortName    tradeDate          closePrice
0       1           平安银行        2017-06-20         9.12
1       2           万科A           2017-06-20         21.03
2       4           国农科技        2017-06-20         27.03
3       5           世纪星源        2017-06-20         5.45
4       6           深振业A         2017-06-20         8.87
5       7           全新好          2017-06-20         15.87
```

或者：

```
In [82]: a.drop('Unnamed: 0', axis='columns')
Out[82]:
        ticker      secShortName    tradeDate           closePrice
0       1           平安银行         2017-06-20          9.12
1       2           万科 A           2017-06-20          21.03
2       4           国农科技         2017-06-20          27.03
3       5           世纪星源         2017-06-20          5.45
4       6           深振业 A         2017-06-20          8.87
5       7           全新好           2017-06-20          15.87
```

axis='columns'或者 axis=1 都是指按照列来处理。

3.2.9　Pandas 替换数据

如果想批量替换数据中的指定数值，则可以使用 replace：

```
In [83]: import numpy as np
In [84]: a.replace(1,np.nan)
Out[84]:
   Unnamed: 0       ticker      secShortName    tradeDate       closePrice
0          0.0      NaN         平安银行         2017-06-20      9.12
1          NaN      2.0         万科 A           2017-06-20      21.03
2          2.0      4.0         国农科技         2017-06-20      27.03
3          3.0      5.0         世纪星源         2017-06-20      5.45
4          4.0      6.0         深振业 A         2017-06-20      8.87
5          5.0      7.0         全新好           2017-06-20      15.87
```

在 Pandas 中对缺失数据使用 NumPy 的 np.nan。

3.2.10　DataFrame 重命名

也可以重命名某些列：

```
In [85]: a.rename(columns={'Unnamed: 0':'id'})
Out[85]:
    id      ticker      secShortName    tradeDate           closePrice
0   0       1           平安银行         2017-06-20          9.12
1   1       2           万科 A           2017-06-20          21.03
2   2       4           国农科技         2017-06-20          27.03
```

```
3   3       5           世纪星源          2017-06-20          5.45
4   4       6           深振业 A          2017-06-20          8.87
5   5       7           全新好            2017-06-20          15.87
```

3.2.11 DataFrame 切片与筛选

DataFrame 有三种切片方法，分别为 loc、iloc 和 ix。

df.loc 的第 1 个参数是行标签，第 2 个参数为列标签（为可选参数，默认为所有列标签），这两个参数既可以是列表，也可以是单个字符。如果这两个参数都为列表，则返回 DataFrame，否则返回 Series：

```
In [86]: a=pd.read_csv('closeprice.csv',encoding='gbk')
In [87]: a
Out[87]:
   Unnamed: 0       ticker      secShortName    tradeDate       closePrice
0           0       1           平安银行          2017-06-20      9.12
1           1       2           万科 A            2017-06-20      21.03
2           2       4           国农科技          2017-06-20      27.03
3           3       5           世纪星源          2017-06-20      5.45
4           4       6           深振业 A          2017-06-20      8.87
5           5       7           全新好            2017-06-20      15.87
In [88]: a.loc[:,['ticker','closePrice']]
Out[88]:
   ticker  closePrice
0       1        9.12
1       2       21.03
2       4       27.03
3       5        5.45
4       6        8.87
5       7       15.87
```

“a.loc[”中的“：”表示所有的行。

df.loc 的第 1 个参数是行的位置，第 2 个参数是列的位置（为可选参数，默认为所有列标签），这两个参数既可以是列表，也可以是单个字符。如果两个参数都是列表，则返回 DataFrame，否则返回 Series：

```
In [89]: a.iloc[:4,[1,4]]
Out[89]:
   ticker  closePrice
0       1        9.12
```

```
1       2       21.03
2       4       27.03
3       5        5.45
```

其中，loc 为 location 的缩写，iloc 为 integer location 的缩写。

更广义的切片方式是使用.ix，它会自动根据我们给出的索引类型判断是使用位置还是标签进行切片：

```
In [90]: a.ix[:4,['ticker','closePrice']]
Out[90]:
   ticker  closePrice
0       1        9.12
1       2       21.03
2       4       27.03
3       5        5.45
4       6        8.87
```

主要根据设置的条件进行筛选，通过逻辑指针进行数据切片，在 DataFrame 中寻找满足条件的记录，方法如下：

```
In [91]: a[a.closePrice>10]
Out[91]:
   Unnamed: 0       ticker      secShortName    tradeDate       closePrice
1           1       2           万科A           2017-06-20      21.03
2           2       4           国农科技        2017-06-20      27.03
5           5       7           全新好          2017-06-20      15.87
```

也可以同时设置多个条件：

```
In [92]: a[(a.closePrice>10) & (a.ticker>3)]
Out[92]:
   Unnamed: 0       ticker      secShortName    tradeDate       closePrice
2           2       4           国农科技        2017-06-20      27.03
5           5       7           全新好          2017-06-20      15.87
```

实际上，因为一个 bool 类型的数据乘以 1，可以变为 1,0，所以可以将上述条件写为：

```
In [93]: a[(a.closePrice>10)*1 + (a.ticker>3)*1==2]
Out[93]:
   Unnamed: 0       ticker      secShortName    tradeDate       closePrice
2           2       4           国农科技        2017-06-20      27.03
5           5       7           全新好          2017-06-20      15.87
```

上述做法可以用于条件筛选，比如在至少或至多满足几个条件时进行筛选。

3.2.12 连续型变量分组

有时我们把数值聚集在一起更有意义，例如，如果要为交通状况（路上的汽车数量）根据时间（分钟数据）建模，则具体的分钟可能不重要，而时段如上午、下午、傍晚、夜间、深夜能更好地表现预测结果，这样建模会更直观，也能避免过度拟合：

```
In [94]: cat=pd.cut(a.closePrice,bins)
         bins=[4,9,10,20,30]
In [95]: cat
Out[95]:
0     (9, 10]
1    (20, 30]
2    (20, 30]
3      (4, 9]
4      (4, 9]
5    (10, 20]
Name: closePrice, dtype: category
Categories (4, object): [(4, 9] < (9, 10] < (10, 20] < (20, 30]]
```

也可以对分组后的结果进行统计：

```
In [96]: pd.value_counts(cat)
Out[96]:
(20, 30]    2
(4, 9]      2
(10, 20]    1
(9, 10]     1
Name: closePrice, dtype: int64
```

也可以给定各个分组的标签并进行切分：

```
In [97]: group_names = ['low',  'Middle_1', 'Middle_2', 'high']
In [98]: pd.cut(a.closePrice, bins, labels=group_names)
Out[98]:
0    Middle_1
1        high
2        high
3         low
4         low
5    Middle_2
Name: closePrice, dtype: category
Categories (4, object): [low < Middle_1 < Middle_2 < high]
```

3.2.13　Pandas 分组技术

在数据分析过程中，我们往往需要先将数据拆分，然后在其对应的每个分组中进行运算，比如，计算各行业内的股票数量、平均市值，或者从中挑选市盈率最低的股票等。Pandas 中的 groupby 函数恰好为我们提供了一个高效的数据分组运算功能，让我们能以一种轻松、自然的方式对数据集进行切分及再聚合等操作，可将其称为“核心功能”。接下来看看 groupby 函数的具体操作流程。

我们可将 groupby 函数的具体操作流程分解为以下三部分。

（1）拆分数据。

（2）应用函数。

（3）汇总计算结果。

如果还是一头雾水的话，则不妨看看图 3-4，毕竟一图胜千言。

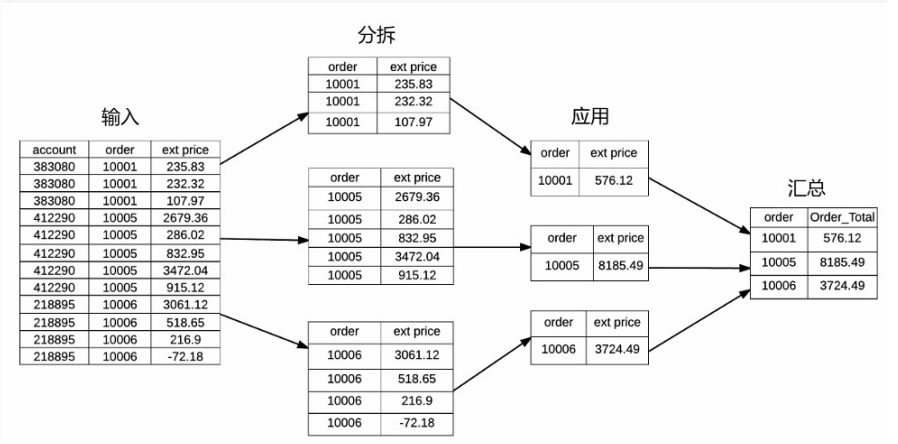


图 3-4

为了加深理解，我们将结合实例来讲解 groupby 函数的应用。

首先，载入数据集，该数据集由基金前 10 大重仓股所对应的市值及行业组成：

```
    In [99]:import pandas as pd
            df = pd.read_csv('20170930.csv', dtype={'ticker': str,
'holdingTicker': str,}, encoding='GBK')
            df = df[['ticker', 'holdingTicker', 'marketValue', 'industryName1']]
            df.head()
```

```
    Out[99]:
```

	ticker	holdingTicker	marketValue	industryName1
0	000001	002236	2.374015e+08	电子
1	000001	000568	2.162791e+08	食品饮料
2	000001	300156	1.603200e+08	公用事业
3	000001	603799	1.538800e+08	有色金属
4	000001	600056	1.519638e+08	医药生物

接下来，计算基金的持股支数，这里利用基金 ticker 对原数据集进行分组，再使用 Pandas 内置的 count()函数进行聚合运算：

```
    In [100]: df[['holdingTicker']].groupby(df['ticker']).count().tail()
    Out[100]:
```

	holdingTicker
ticker	
740101	10
750001	10
750005	10
762001	10
770001	10

可以发现该数据集的确为前十大重仓股，我们再根据行业对原数据集进行分组，计算每个行业对应的股票数量：

```
    In [101]: df[['holdingTicker']].groupby(df['industryName1']).count().
sort_values('holdingTicker', ascending=False).head()
    Out[101]:
```

	holdingTicker
industryName1	
银行	4666
电子	3850

食品饮料	3539
非银金融	3306
医药生物	2950

可以发现，公募基金的持仓风格一般都偏向于银行、电子、食饮和非银金融等行业。

接下来计算公募基金前十大重仓股的市值之和，这里用到了 sum()函数：

```
    In [102]: df[['marketValue']].groupby(df['ticker']).sum().sort_values
('marketValue', ascending=False).head()
    Out[102]:
```

	marketValue
ticker	
510050	1.818564e+10
001683	9.222674e+09
001772	7.983531e+09
150201	7.650096e+09
150200	7.650096e+09

可以发现，排名靠前的一般为 ETF 或老牌明星基金。

在这里还可以使用 mean()、std()、min()和 size()等聚合函数。我们不仅可以使用 Pandas 的内置函数进行聚合，还可以使用自定义函数进行聚合。先定义一个 t_range 函数，用于计算公募基金前十大重仓股中的最大市值与最小市值之差：

```
    In [103]: def t_range(arr):
                  return arr.max() - arr.min()
```

这里使用了 agg()函数，可以将我们之前自定义的函数传入其中：

```
    In [104]: df[['marketValue']].groupby(df['ticker']).agg(t_range).head()
    Out[104]:
```

	marketValue
ticker	

000001	1.349458e+08
000003	1.820900e+06
000004	1.820900e+06
000005	0.000000e+00
000007	7.454000e+05

agg()函数还支持以列表形式传入多个函数：

```
    In [105]: df[['marketValue']].groupby(df['ticker']).agg(['sum', 'max',
t_range]).head()
    Out[105]:
```

		marketValue	
	sum	max	t_range
ticker			
000001	1.517243e+09	2.374015e+08	1.349458e+08
000003	5.342707e+06	1.978000e+06	1.820900e+06
000004	5.342707e+06	1.978000e+06	1.820900e+06
000005	1.277144e+06	1.277144e+06	0.000000e+00
000007	8.266880e+06	1.287000e+06	7.454000e+05

在 agg()函数中不仅可以应用同一种函数，还可以通过不同的列应用不同的求解函数。下面需要分别对市值及行业应用不同的统计函数，并输出不同的列的数据，这时可以通过 agg 自定义函数实现，只需传入以列名为键值的字典：

```
    In [106]: df[['marketValue', 'industryName1']].groupby(df['ticker']).agg
({'marketValue':[t_range], 'industryName1':['count']}).head()
    Out[106]:
```

	marketValue	industryName1
	t_range	count
ticker		
000001	1.349458e+08	10

000003	1.820900e+06	4
000004	1.820900e+06	4
000005	0.000000e+00	1
000007	7.454000e+05	10

虽然 agg()的功能已经很强大了，但是仍然有局限。细心的读者可能已经发现了，agg()返回的只能是一个标量，如果我们想返回一个 DataFrame，使用 agg()就会束手无策。不过不用担心，我们可以使用对象更为广泛的分组运算方法——apply()函数。

这里传入了一个匿名函数，返回前 3 只股票的相关信息：

```
In [107]: df.groupby('ticker').apply(lambda x: x[:3]).head(6)
Out[107]:
```

		ticker	holdingTicker	marketValue	industryName1
ticker					
000001	0	000001	002236	2.374015e+08	电子
	1	000001	000568	2.162791e+08	食品饮料
	2	000001	300156	1.603200e+08	公用事业
000003	10	000003	600622	1.978000e+06	房地产
	11	000003	600884	1.930107e+06	电子
	12	000003	600036	1.277500e+06	银行

有了 agg 和 apply 后，我们就可以完成关于分组再聚合的所有操作和运算了。

3.3　SciPy 的初步使用

SciPy 包含致力于解决在科学计算中的常见问题的各个工具箱，它的不同子模块对应不同的应用，例如插值、积分、优化、图像处理和特殊函数等。

SciPy 可以与其他标准科学计算程序库如 GSL（GNU C 或 C++科学计算库）或者 MATLAB 工具箱进行比较，是 Python 中科学计算程序的核心包，用于有效地计算 NumPy 矩阵，以便于 NumPy 和 SciPy 协同工作。

由于 SciPy 涉及的领域众多，所以本章只选择部分与金融量化分析相关的子库进行介绍，并结合实例进行讲解。

3.3.1 回归分析

回归分析（Regression Analysis）是确定两种或两种以上变量之间相互依赖的定量关系的一种统计分析方法，运用十分广泛。回归分析按照涉及的变量的多少，可分为一元回归分析和多元回归分析；按照因变量的多少，可分为简单回归分析和多重回归分析；按照自变量和因变量之间的关系类型，可分为线性回归分析和非线性回归分析。

回归分析在金融领域的应用十分广泛，比如在 CAPM 模型中 beta 系数的计算，本质上就是一元线性回归，我们先从这个简单的例子入手。

按照惯例，这里先引入 NumPy、Pandas 及 Matplotlib 库，并更改 Matplotlib 库的字体以便其正常显示中文：

```
In [108]:import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
         import datetime as dt
         %matplotlib inline
In [109]:plt.rcParams['font.sans-serif']=['SimHei']  #用来正常显示中文标签
         plt.rcParams['axes.unicode_minus']=False  #用来正常显示负号
```

首先，获取近两年来沪深 300 指数及其最大成分股中国平安的每日涨跌幅数据：

```
In [110]:data = pd.read_csv('data.csv', index_col='Date')
         data.index = [dt.datetime.strptime(x, '%Y-%m-%d') for x in data.index]
In [110]:data.head()
Out[110]:
```

	沪深 300	中国平安
2015-06-23	0.03214	0.0496
2015-06-24	0.01965	0.0052
2015-06-25	-0.03557	-0.0287
2015-06-26	-0.07868	-0.0605
2015-06-29	-0.03336	-0.0119

```
In [111]:data.plot(figsize=(10, 6))
         plt.ylabel('涨跌幅')
```

其效果如图 3-5 所示。

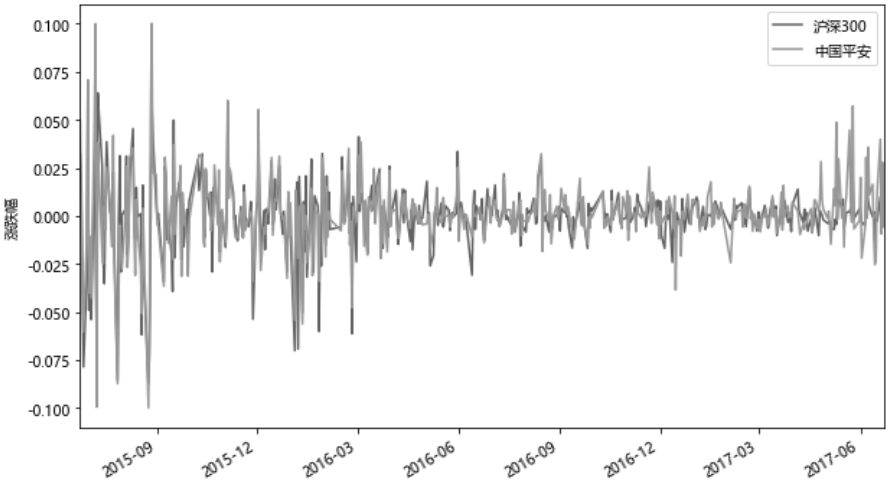


图 3-5

可以发现，中国平安与沪深 300 的相关性还是比较大的。

接下来要用到 StatsModels 模块。该模块曾经是 scipy.stats 下的子模块，后被重写并成为现在独立的 StatsModels 模块，在此利用其强大的 OLS（Ordinary Least Square）功能进行回归分析。

首先，引入该模块：

```
In [112]:import statsmodels.api as sm
In [113]:x = data['沪深 300'].values
         X = sm.add_constant(x)  #添加常数项
         y = data['中国平安'].values
```

接着，对二者的收益率进行线性回归，斜率即 beta 系数：

```
In [114]:model = sm.OLS(y, X)
         results = model.fit()
         results.params
Out[114]:array([ 0.00095063,  0.80946537])

In [115]:plt.figure(figsize=(10, 6))
```

```
            plt.plot(x, y, 'o', label='中国平安-沪深 300')
            plt.plot(x, results.fittedvalues, 'r--', label='ordinary least square')
            plt.legend()
            plt.xlabel('沪深 300')
            plt.ylabel('中国平安')
            plt.grid(True)
```

为了让结果更加直观，我们将源数据及拟合直线可视化，如图 3-6 所示。

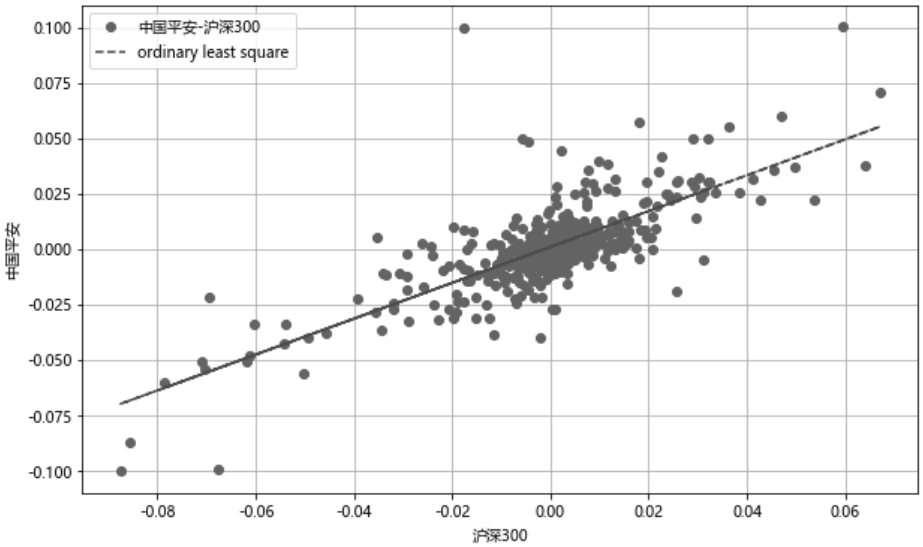


图 3-6

同样，归因分析对于投资组合管理十分重要。某一组合的收益与损失可以被分解为多个表示投资组合权重的因子。我们认为一个虚拟投资组合是由两个随机因子和事先分配的权重构成的，这一分析过程类似于多元线性回归：

```
In [116]:import numpy.random as npr
```

接下来分别构造三个随机因子，并对投资组合添加噪声，再对其进行多元线性回归。与一元线性回归类似，二者仅仅在基函数的数量上有所不同：

```
In [117]:factor = npr.rand(1000, 3)
          Factor = sm.add_constant(factor)  #添加常数项
          fac1 = factor[:, 0]  #因子 1
          fac2 = factor[:, 1]  #因子 2
          fac3 = factor[:, 2]  #因子 3
```

```
          e = npr.random(1000)  #噪声
          port = fac1*0.3+fac2*0.7+fac3*0.4+e #虚构投资组合及因子权重
In [118]:model1 = sm.OLS(port, Factor)
          results1 = model1.fit()
          results1.params
Out[118]:array([ 0.46389883,  0.28832045,  0.70877495,  0.45948944])
```

可以看到，由于对投资组合添加了一定的噪声，所以导致计算出的因子权重与原先设置的因子权重相比有了微小的偏差。

3.3.2　插值

插值是在离散数据的基础上补插连续函数，使这条连续曲线通过全部给定的离散数据点。插值是离散函数逼近的重要方法，可根据函数在有限个点处的取值状况，估算出函数在其他点处的近似值。

同样，大部分金融数据都是离散数据，所以插值在金融领域的应用极其广泛，我们先从一个实例入手。

首先，读取近一个月的沪深 300 指数数据：

```
In [119]:data1 = pd.read_csv('data1.csv')
In [120]:data1.head()
Out[120]:
```

	沪深 300
0	3424.1940
1	3424.1669
2	3485.6581
3	3480.4345
4	3492.8845

接下来，导入 scipy.interpolate 子库对数据进行样条插值：

```
In [121]:import scipy.interpolate as spi
In [122]:X = data1.index  #定义数据点
          Y = data1.values  #定义数据点
          x = np.arange(0, len(data1), 0.15)  #定义观测点
```

分别对源数据进行线性及三次样条插值，使用的函数为 splrep 和 splev，在表 3-3 中列出了 splrep 函数的主要参数及其描述，在表 3-4 中列出了 splev 函数的主要参数及其描述。

表 3-3

参　数	描　述
x, y	Y=f(x)上的数据点
w	应用到 *y* 坐标上的权重
xb, xe	拟合区间
s	平滑因子，权衡相似度与平滑度之间的关系
k	样条拟合阶数

表 3-4

参　数	描　述
x	一组插值点的 *x* 坐标
tck	Splrep 返回的长度为 3 的 tuple（节点，系数，阶数）
der	导数的阶
ext	如果 x 不在节点序列中，则 0 外推，1 返回 0，2 返回异常，3 返回边界值

代码如下：

```
In [123]:ipo1 = spi.splrep(X,Y,k=1) #k 样条拟合顺序（1<=k<=5）
         ipo3 = spi.splrep(X,Y,k=3)
In [124]:iy1 = spi.splev(x,ipo1)
         iy3 = spi.splev(x,ipo3)
```

然后，对源数据及插值点进行可视化，效果如图 3-7 所示。

```
In [125]:fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,12))
         ax1.plot(X, Y, label='沪深 300')
         ax1.plot(x, iy1, 'r.', label='插值点')
         ax1.set_ylim(Y.min() - 10, Y.max() + 10)
         ax1.set_ylabel('指数')
         ax1.set_title('线性插值')
         ax1.legend()
         ax2.plot(X, Y, label='沪深 300')
         ax2.plot(x, iy3, 'r.', label='插值点')
         ax2.set_ylim(Y.min() - 10, Y.max() + 10)
         ax2.set_ylabel('指数')
         ax2.set_title('三次样条插值')
```

```
ax2.legend()
```

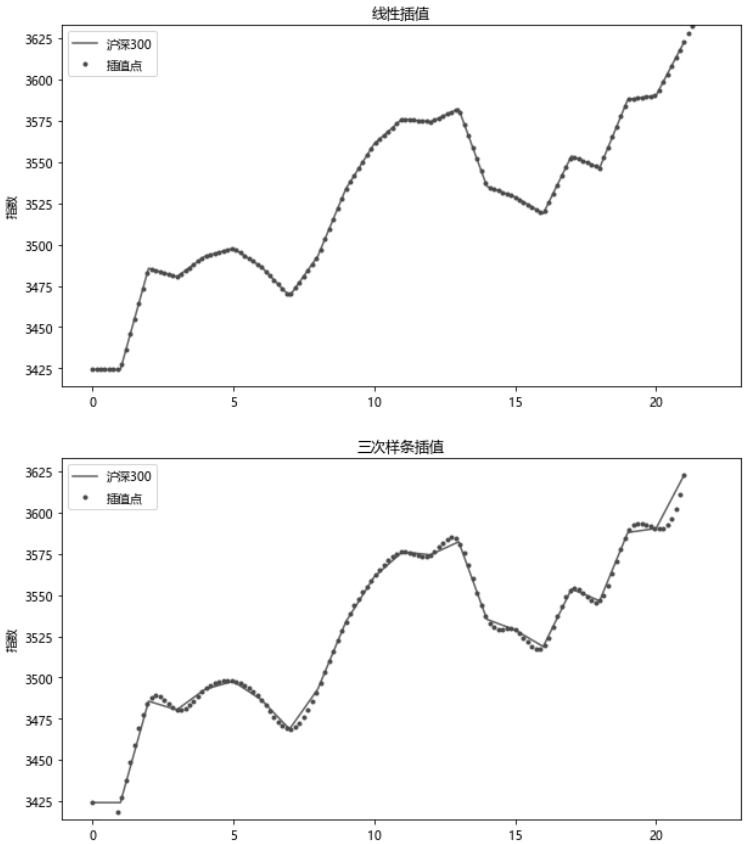


图 3-7

由于源数据本身就是以天为单位的离散数据点，所以在使用线性插值时，插值点与 Matplotlib 贴合。在使用三次样条插值时，插值点形成的曲线更为平滑、连续。

3.3.3　正态性检验

在 3.1.5 节中，我们提到许多金融模型都依赖于正态分布，这是金融理论的主要统计学基础之一。然而股票市场的收益是否真的符合正态分布？接下来我们利用 A 股数据检验一下。

首先，获取“平安银行”“万科 A”“格力电器”“华东医药”“东方雨虹”“宇通客车”

及“海螺水泥”从 2011 年至 2014 年每日的前复权收盘价格（这里的股票是随机选择的，不代表投资建议），代码如下：

```
In [126]:data2 = pd.read_csv('data2.csv', index_col='Date')
         data2.index=[dt.datetime.strptime(x,'%Y-%m-%d')for x in data2.index]
         data2.head()
Out[126]:
```

	平安银行	万科 A	格力电器	华东医药	东方雨虹	宇通客车
2011-01-04	5.491	7.418	7.004	15.301	9.705	5.859
2011-01-05	5.460	7.444	6.934	16.316	9.945	5.748
2011-01-06	5.419	7.452	6.957	16.175	10.017	6.224
2011-01-07	5.624	7.494	6.823	15.583	9.664	6.305
2011-01-10	5.477	7.368	6.726	15.037	9.462	6.299

我们将各股在起始日期的证券价格归一化，可视化价格如图 3-8 所示。

```
In [127]:(data2 / data2.iloc[0]*100).plot(figsize=(10,6))
         plt.xlabel('股价')
         plt.legend(loc='upper left')
         plt.grid(True)
```

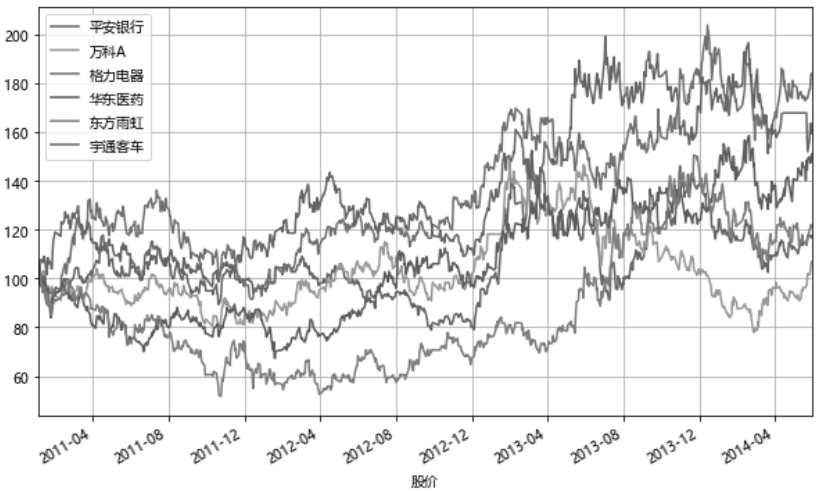


图 3-8

再分别计算各股的对数收益率：

```
In [128]:log_returns = np.log(data2.pct_change() + 1)
         log_returns.head()
Out[128]:
```

	平安银行	万科 A	格力电器	华东医药	东方雨虹	宇通客车
2011-01-04	NaN	NaN	NaN	NaN	NaN	NaN
2011-01-05	-0.005662	0.003499	-0.010045	0.064228	0.024429	-0.019127
2011-01-06	-0.007537	0.001074	0.003311	-0.008679	0.007214	0.079561
2011-01-07	0.037132	0.005620	-0.019449	-0.037286	-0.035876	0.012930
2011-01-10	-0.026486	-0.016956	-0.014319	-0.035667	-0.021124	-0.000952

利用 pandas.hist 将数据可视化：

```
In [129]:log_returns.hist(bins=50, figsize=(10,6), layout=(2, 3))
```

可以发现，至少从数据分布情况上来看，各股的对数收益率并不太符合正态分布，如图 3-9 所示。

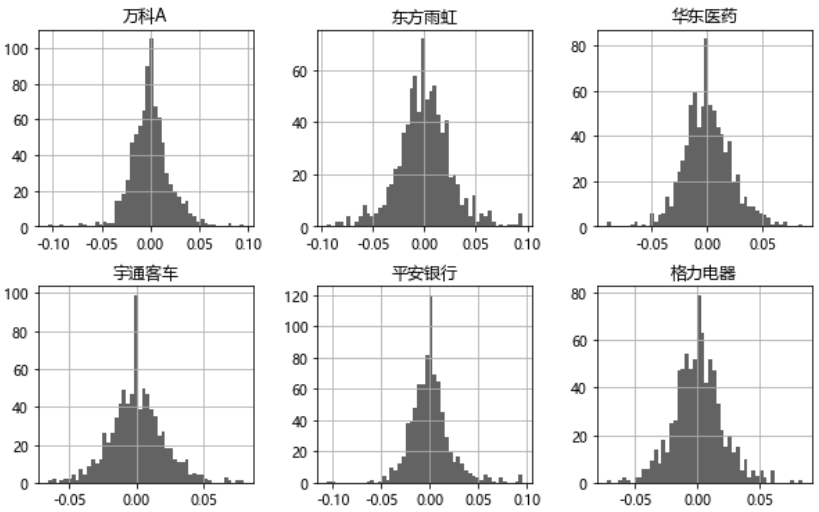


图 3-9

接下来我们通过分位数-分位数（qq）图来进一步判断各股的对数收益率是否符合正态分布。statsmodel.api 中的 qqplot 可以帮助我们完成这一工作，效果如图 3-10 所示。

```
    In [130]:fig, axes = plt.subplots(3, 2, figsize=(10, 12))
              for i in range(0, 3):
                  for j in range(0, 2):
                      sm.qqplot(log_returns.iloc[:, 2 * i + j].dropna(),
line='s',ax=axes[i, j])
                      axes[i, j].grid(True)
                      axes[i, j].set_title(log_returns.columns[2 * i + j])
                      axes[i, j].set_xlabel('理论分位数')
                      axes[i, j].set_ylabel('样本分位数')
              plt.subplots_adjust(wspace=0.3, hspace=0.4)
```

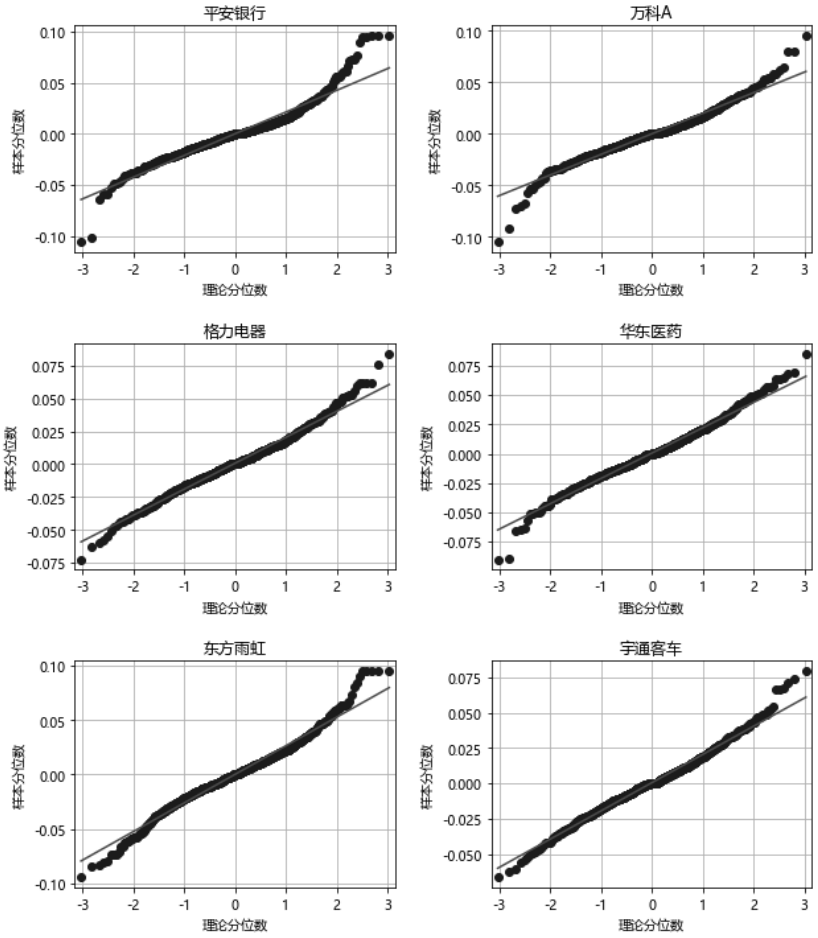


图 3-10

通过观察图 3-10 可以发现，样本分位数并不在一条直线上，这表明各股的对数收益率并不符合正态分布。在两侧分别有许多值远低于或远高于直线，这就是“肥尾效应”。

3.3.4　凸优化

“凸优化”是一种比较特殊的优化，指求取最小值的目标函数为凸函数的一类优化问题。在各行各业中都充斥着凸优化问题，其中以金融领域尤为明显，比如，利用市场数据校准齐全定价模型、效用函数优化、投资组合优化等。为了更加切合本书的主题，本节直接运用投资组合优化作为例子进行展示。

马柯维茨投资组合理论是美国经济学家 Markowitz（1952）在其发表的论文《资产组合的选择》中提出的，它标志着现代投资组合理论研究的开始。Markowitz 利用均值-方差模型分析得出通过投资组合可以有效降低风险的结论。这一理论的基本原理已经被写进诸多教材中，在此不作重复阐述，感兴趣的读者可以查阅相关资料。

首先，使用蒙特卡罗模拟生成一定数量的随机投资组合权重向量，并计算其相应的预期投资组合收益及方差：

```
In [131]:number = 10000  #随机数维度
         stock_num = len(log_returns.columns)
         weights = npr.rand(number, stock_num)
         weights/= np.sum(weights, axis=1).reshape(number, 1)
         prets = np.dot(weights, log_returns.mean()) * 252  #计算年化收益率
         pvols = np.diag(np.sqrt(np.dot(weights,
                         np.dot(log_returns.cov() * 252, weights.T)))) #
计算年化风险
```

接下来，对结果进行可视化，并将无风险短期利率定义为 0，以便计算夏普率。

```
In [132]:plt.figure(figsize=(10, 6))
         plt.scatter(pvols, prets, c=prets / pvols, marker='o')
         plt.grid(True)
         plt.xlabel('预期波动率')
         plt.ylabel('预期收益率')
         plt.colorbar(label='夏普率')
```

效果如图 3-11 所示。

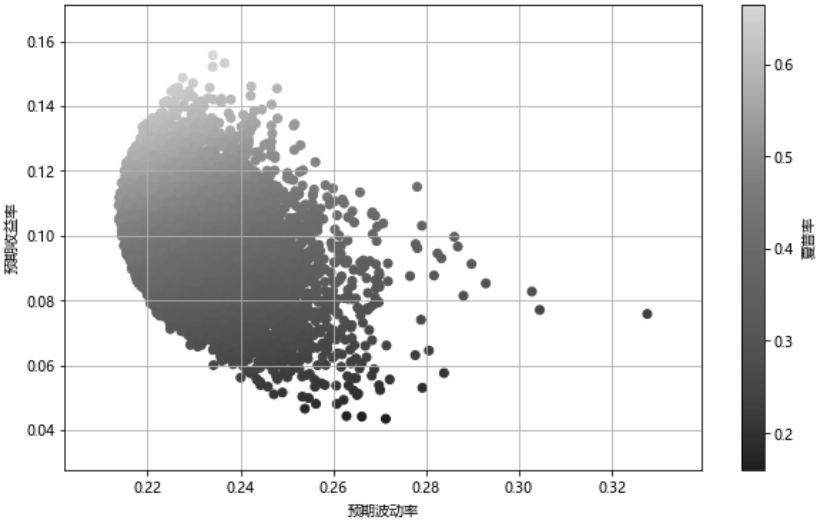


图 3-11

投资组合的优化关键在于对组合中各标的权重的优化，优化目标大致有以下三类。

（1）夏普最大化。

（2）收益最大化。

（3）风险最小化。

最优化投资组合的推导是一个约束最优化问题。我们可以使用 scipy.optimize 子库中的 minmize 函数求解这一问题。

首先，导入该模块：

```
In [133]:import scipy.optimize as sco
```

然后，建立一个函数将收益率、波动率、夏普率封装起来：

```
In [134]:def statistics(weights):
              weights = np.array(weights)
              pret = np.sum(log_returns.mean() * weights * 252)
              pvols = np.sqrt(np.dot(weights.T, np.dot(log_returns.cov() *
252, weights)))
              return np.array([pret, pvols, pret / pvols])
```

接着，建立一个目标函数：

```
In [135]:def min_sharpe(weights):
             return -statistics(weights)[2]
```

并将所有参数（权重）的总和约束为 1，将参数的权重限制在 0～1：

```
In [136]:cons = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
          bnds = tuple((0, 1) for x in range(stock_num))
```

最后，调用 minmize 函数进行最优化求解：

```
In [137]:opts = sco.minimize(min_sharpe, stock_num * [1 / stock_num],
                             method='SLSQP', bounds=bnds, constraints=cons)
          opts['x'].round(3)
Out[137]:array([ 0.   ,  0.   ,  0.521,  0.227,  0.   ,  0.251])
```

可以发现，在夏普率最大的投资组合中仅包括格力电器、华东医药及宇通客车。

此时该组合的统计数字为：

```
In [138]:statistics(opts['x'].round(3))
Out[138]:array([ 0.16181374,  0.23682582,  0.68326054])
```

即预期收益率为 16.2%，预期波动率为 23.7%，最优夏普率为 0.68。以此类推，我们还可以得到风险（波动率）最小化投资组合：

```
In [139]:def min_volatility(weights):
             return statistics(weights)[1]
In [140]:opts1 = sco.minimize(min_volatility, stock_num * [1 / stock_num],
                              method='SLSQP', bounds=bnds, constraints=cons)
          opts1['x'].round(3)
Out[140]:array([ 0.109,  0.199,  0.171,  0.255,  0.044,  0.221])
In [141]:statistics(opts1['x'].round(3))
Out[141]:array([ 0.10758601,  0.21346307,  0.50400293])
```

还可以得到收益最大化组合：

```
In [142]:def min_return(weights):
             return -statistics(weights)[0]
In [143]:opts2 = sco.minimize(min_return, stock_num * [1 / stock_num],
                              method='SLSQP', bounds=bnds, constraints=cons)
          opts2['x'].round(3)
Out[143]:array([ 0.,  0.,  1.,  0.,  0.,  0.])
In [144]:statistics(opts2['x'].round(3))
Out[144]:array([ 0.18677101,  0.31284775,  0.59700289])
```

接下来构建有效边界，即计算在给定收益的情况下风险最小的投资组合权重。与之前的三个最优化求解过程相比，这一步仅仅在约束条件上有所变化，需要确保预期收益率等于给定的收益率：

```
In [145]:trets=np.linspace(0.04, 0.18, 50)
          tvols=[]
          for tret in  trets:
              cons1=({'type': 'eq', 'fun': lambda x: statistics(x)[0] - tret},
                  {'type': 'eq', 'fun': lambda x: np.sum(x) - 1})
              res=sco.minimize(min_volatility, stock_num * [1 / stock_num],
                           method='SLSQP', bounds = bnds, constraints = cons1)
              tvols.append(res['fun'])
          tvols=np.array(tvols)
```

将这一结果可视化，叉号组成的曲线表示在给定的收益率下的最优投资组合，星号分别为最大夏普率、最小波动率及最大收益率组合，效果如图 3-12 所示。

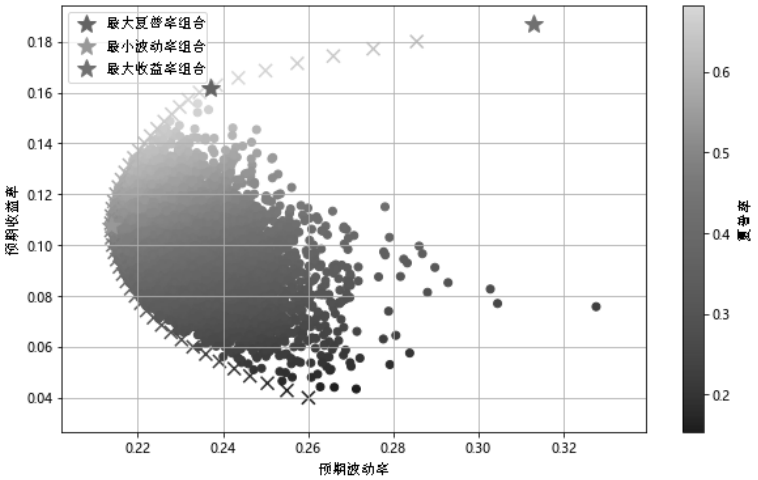


图 3-12

```
In [146]:plt.figure(figsize=(10,6))
          plt.scatter(pvols, prets,
                      c=prets / pvols, marker='o')
          plt.scatter(tvols, trets,
                      c=trets / tvols, marker='x', s=100)
          plt.plot(statistics(opts['x'])[1], statistics(opts['x'])[0],
                '*', markersize=15, label='最大夏普率组合')
          plt.plot(statistics(opts1['x'])[1], statistics(opts1['x'])[0],
```

```
        '*', markersize=15, label='最小波动率组合')
plt.plot(statistics(opts2['x'])[1], statistics(opts2['x'])[0],
        '*', markersize=15, label='最大收益率组合')
plt.grid(True)
plt.xlabel('预期波动率')
plt.ylabel('预期收益率')
plt.colorbar(label='夏普率')
plt.legend()
```

3.4　Matplotlib 的使用

Matplotlib 是 Python 绘图的始祖。现阶段的大部分高级绘图包如 Seaborn、holoviews、ggplot 及 Pandas 中的 plot()函数等都是基于 Matplotlib 定制的。换而言之，那些高级绘图包能做到的事，Matplotlib 也能做到，只是可能需要长时间的调试及较多的代码才能达到同一目的，这有点类似于 C 和 Python 的关系。

Matplotlib 是一个 Python 2D 绘图库，起初是为了模仿 MATLAB 中的绘图命令而开发的，现在可以在各种 Web 应用及像 Jupyter Notebook 这样的交互式环境中绘画图像。Matplotlib 拥有丰富且完善的说明文档，我们可以仅使用几行代码就生成散点图、直方图、柱形图及饼图等，也可以使用命令来控制图中的字体、标题、轴等属性。只要我们能想到的，Matplotlib 都能做到。

虽然对于科学家而言，绘图的美观性可能并不重要，只要能清晰地表达出数据的含义就足够了，然而在金融或者互联网领域，绘图的美观性就显得尤为重要了，此时如果依旧使用 Matplotlib，代码的数量就会变得愈发冗长。Seaborn 就是为了解决这一问题而出现的。

3.5　Seaborn 的使用

能画出引人注目的图表是非常重要的，例如在探索一个数据集时能绘制出美观的图表是件令人高兴的事；又如，我们在与观众交流观点时，也需要让图表吸引观众的注意力并让其印象深刻。Matplotlib 的自动化程度非常高，掌握如何设置系统以获得一个具有吸引力的图表并不容易。为了优化 Matplotlib 图表的外观，Seaborn 模块自带许多定制主题和高级接口。

Seaborn 模块被用于在 Python 中制作有吸引力的统计图形，它构建在 Matplotlib 之上，支持 NumPy 和 Pandas 的数据结构，以及来自 SciPy 和 StatsModels 的统计结果。该模块旨在用尽可能少的参数生成尽可能漂亮的图形，不过 Seaborn 是对 Matplotlib 的补充，而不是替代品。

Seaborn 的主要功能如下。

- 通过内置主题改善 Matplotlib 的外观。
- 有丰富的调色板工具，可更好地显示数据。
- 对变量分布进行可视化。
- 数据矩阵可视化，并使用聚类算法发现这些矩阵中的结构。
- 对自变量和因变量之间的线性回归结果进行可视化。
- 绘制统计时间序列，并将其不确定性可视化。
- 构建高级、抽象的网格图，可轻松地将复杂的问题可视化。

3.5.1 主题管理

Seaborn 的一个重要优点是内置了经过美化的主题，这类似于我们经常使用的 PPT 模板，用户无须在外观调整上花费多余的精力。我们通过一个简单的例子进行理解。

首先，导入 NumPy 及 Matplotlib 模块：

```
In [147]:import numpy as np
          import matplotlib as mpl
          import matplotlib.pyplot as plt
          %matplotlib inline
In [148]:plt.rcParams['font.sans-serif']=['SimHei']  #用来正常显示中文标签
          plt.rcParams['axes.unicode_minus']=False  #用来正常显示负号
```

然后，定义一个函数，用来绘画正弦函数，全部使用 Matplotlib 中的默认参数，不需要任何调整：

```
In [149]:def sinplot(flip=1):
              x = np.linspace(0, 14, 100)
              for i in range(1, 7):
                 plt.plot(x, np.sin(x + i * .5) * (7 - i) * flip)
In [150]:sinplot()
```

可以看到生成的图像没有网格线，颜色饱和度也较高，看上去不是那么柔和，如图 3-13 所示。

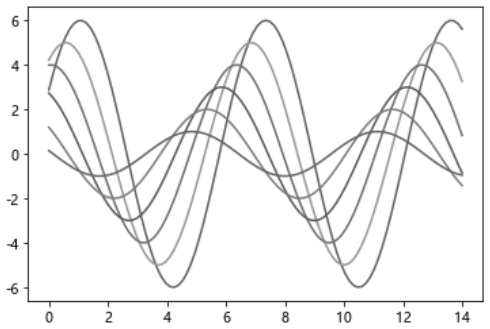


图 3-13

接下来，引入 Seaborn 模块，不经过任何设置，只需简单的 import 即可生效。可以看到，背景色变成了浅灰色，还生成了白色网格线，颜色饱和度也降低了，曲线与背景之间的对比变得不那么强烈，简而言之，看上去很舒服。如果我们亲自敲一遍代码，则还会发现图片的大小也发生了变化，变得更适合输出窗口：

```
    In [151]:import seaborn as sns
              sns.set_style({"font.sans-serif":['Microsoft YaHei', 'SimHei']})
#显示中文
    In [152]:sinplot()
```

其效果如图 3-14 所示。

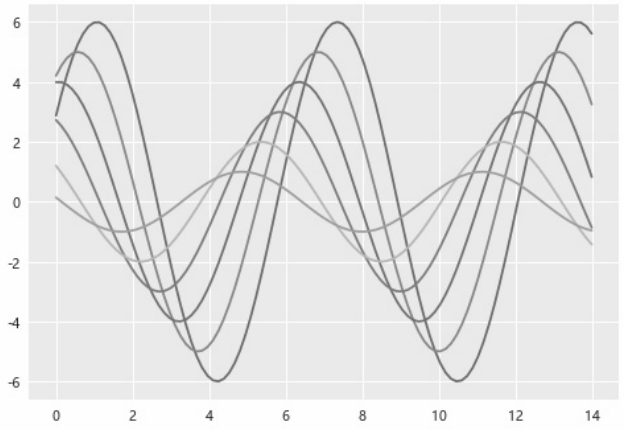


图 3-14

Seaborn 一共有 4 个内置主题，分别为 paper、talk、poster 和 notebook，对应不同的场合。可使用 set_context()函数设置：

```
In [153]:plt.figure(figsize=(12, 8))
          sns.set_context('paper')
          plt.subplot(221)
          sinplot()
          plt.title('paper')
          sns.set_context('talk')
          plt.subplot(222)
          sinplot()
          plt.title('talk')
          sns.set_context('poster')
          plt.subplot(223)
          sinplot()
          plt.title('poster')
          sns.set_context('notebook')
          plt.subplot(224)
          sinplot()
          plt.title('notebook')
```

各自的风格如图 3-15 所示。

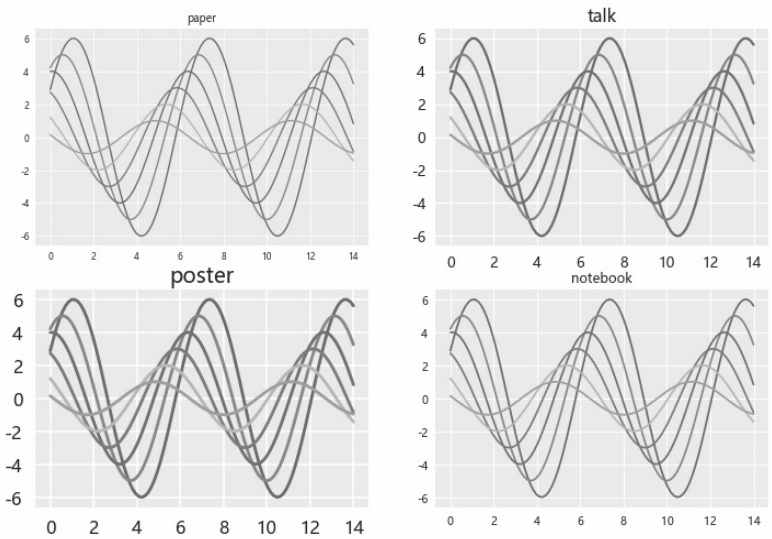


图 3-15

3.5.2　调色板

Seaborn 拥有极其丰富的调色板库，大致可分为循环、渐变、混合三类，利用 set_palette() 函数进行设置：

```
In [154]:plt.figure(figsize=(12, 8))
          sns.set_palette("muted")
          plt.subplot(221)
          sinplot()
          plt.title('循环')
          sns.set_palette("Blues_d")
          plt.subplot(222)
          sinplot()
          plt.title('渐变（深-浅）')
          sns.set_palette("Blues")
          plt.subplot(223)
          sinplot()
          plt.title('渐变（浅-深）')
          sns.set_palette("RdBu")
          plt.subplot(224)
          sinplot()
          plt.title('混合（红-蓝）')
```

将结果可视化的效果如图 3-16 所示。

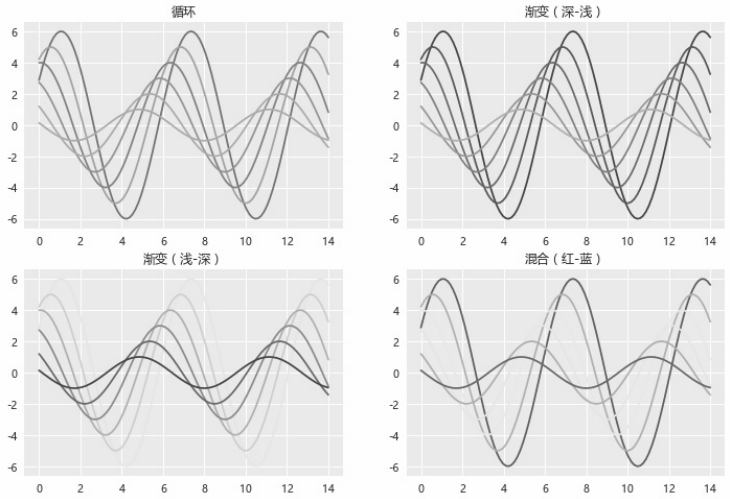


图 3-16

3.5.3 分布图

Seaborn 不仅在图形的外观上提供了有效的帮助，还提供了辅助图形，使我们更快捷地获得可视化结果。先从最简单的正态分布直方图入手：

```
In [155]:import numpy.random as npr
In [156]:size = 1000
         rn1 = npr.standard_normal(size)
```

Seaborn 中的 distplot 函数起到了类似于 Matplotlib 中的 hist 函数的功能，但更强大一些。我们通过 kde 及 rug 参数就可以选择是否显示核密度估计及边际毛毯图，效果如图 3-17 所示：

```
In [157]:sns.set_palette("muted")
         fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(10, 10))
         ax1.hist(rn1, bins=25)
         ax1.set_title('fig1')
         sns.distplot(rn1, bins=25, kde=False, ax=ax2)
         ax2.set_title('fig2')
         sns.distplot(rn1, bins=25, kde=True, ax=ax3)
         ax3.set_title('fig3')
         sns.distplot(rn1, bins=25, kde=True, rug=True, ax=ax4)
         ax4.set_title('fig4')
```

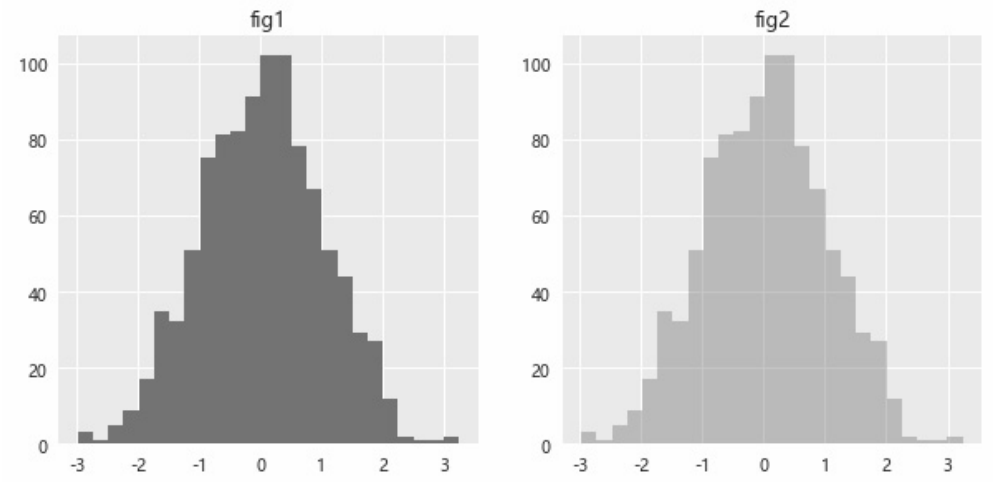


图 3-17

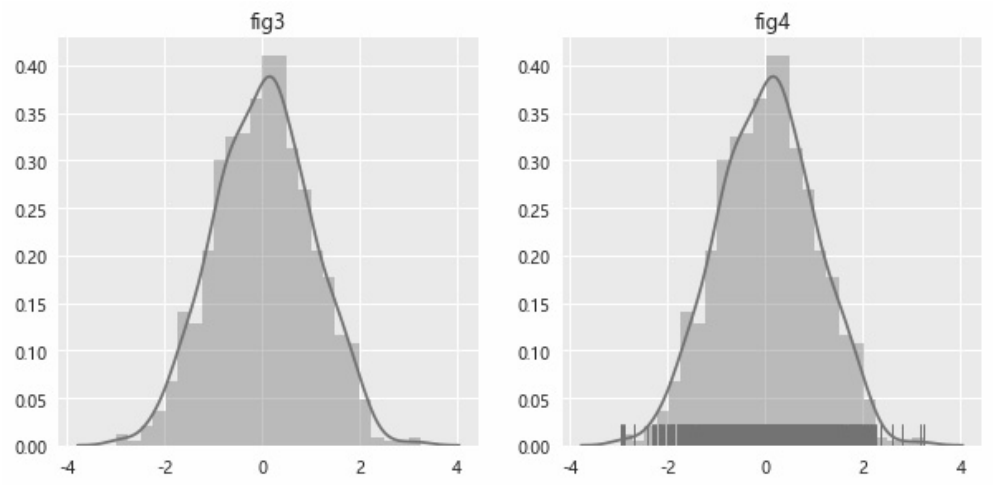


图 3-17（续）

图 3-17 很好地解释了 distplot 的运作原理。fig1 是用传统的 hist 函数做出的直方图，与关闭 kde 和 rug 的图 fig 类似，但是显然后者更美观一些。图 fig3、图 fig4 分别展示了 kde 及 rug 的作用。

接下来展示 Seaborn 中的 jointplot 函数，通过这个函数可以很方便地显示两个变量之间的相关性及各自的分布。

首先，导入 3.3 节中沪深 300 与中国平安的每日涨跌幅数据：

```
In [158]:import pandas as pd
In [159]:rn2 = pd.read_csv('data.csv', index_col='Date')
```

试想一下，如果要达到 jointplot 的可视化结果，则至少需要如下 4 步。

（1）计算两列数据的相关系数。

（2）画一个散点图。

（3）画两个直方图。

（4）拼合起来。

想想都很麻烦，但是如果使用 jointplot，则一切都变得简单起来，通过仅仅一行代码就能达到我们的目的，效果如图 3-18 所示：

```
In [160]:sns.jointplot(rn2['沪深 300'], rn2['中国平安'], size=8)
```

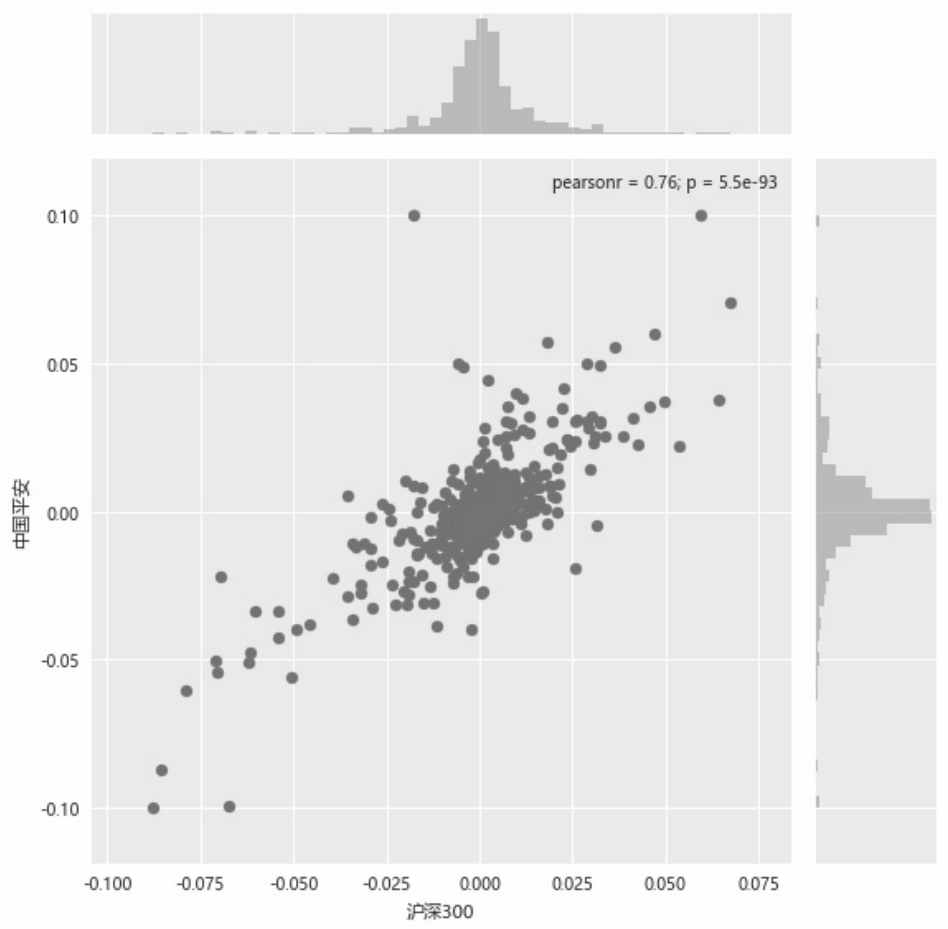


图 3-18

3.5.4 回归图

这里先回忆一下在 3.3 节中针对可视化线性回归图进行的操作：利用 StatsModels 模块计算线性回归系数，再通过 Matplotlib 进行可视化。如果利用 lmplot 函数，这个过程就会变得更简捷。lmplot 函数在作图时会根据设置调用 numpy.polyfit 或者 StatsModels 对数据进行处理，并直接将其可视化，效果如图 3-19 所示：

```
In [161]:sns.lmplot('沪深 300', '中国平安', data = rn2, size=8)
```

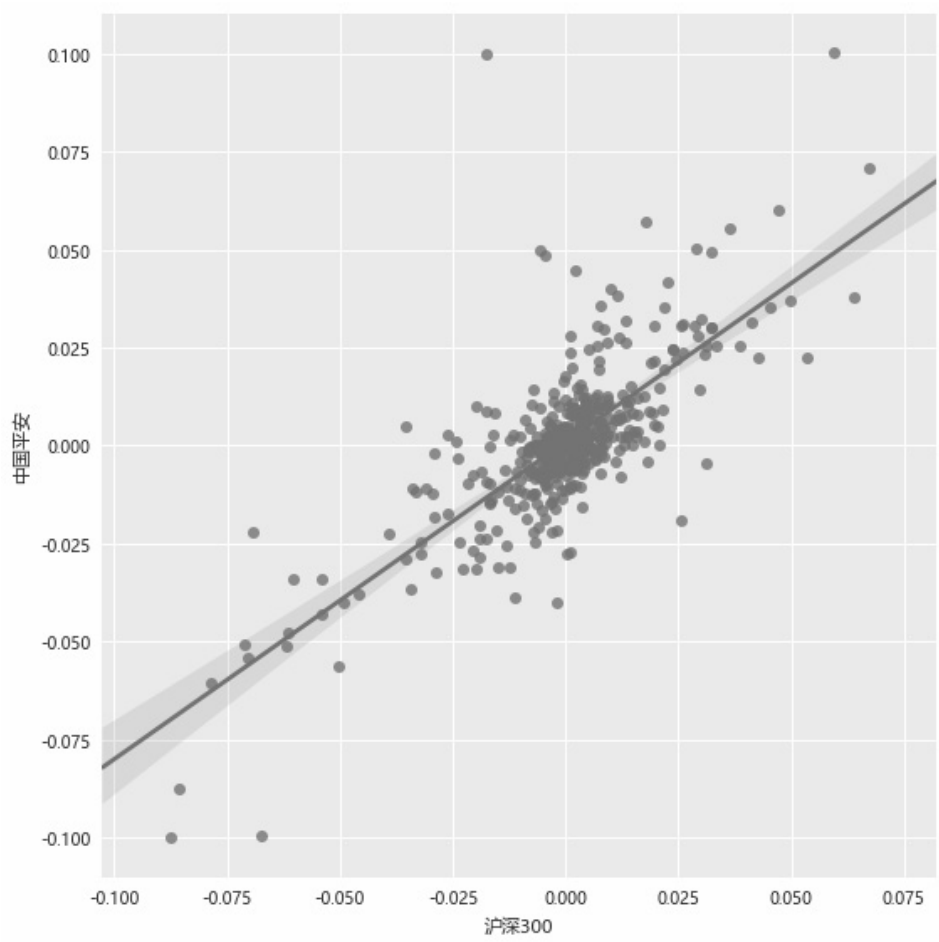


图 3-19

如果仅仅是线性回归的话，则对 jointplot 函数中的 kind 参数进行设置也能达到类似的效果，效果如图 3-20 所示：

```
In [162]:sns.jointplot(rn2['沪深 300'], rn2['中国平安'], kind='reg', size=8)
```

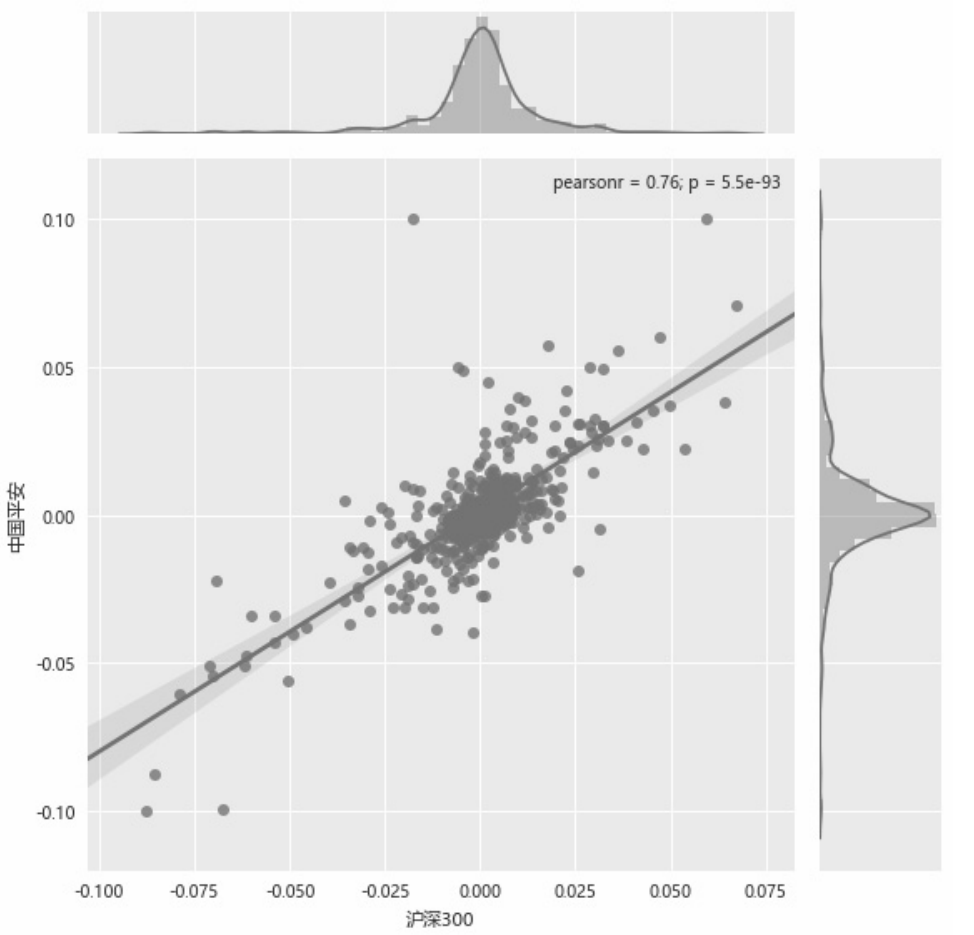


图 3-20

3.5.5 矩阵图

heatmap 是一种典型的矩阵图，通过颜色的深浅来表示数据的大小。在金融领域中，我们往往利用 heatmap 将各因素的历史表现可视化。用过 Matplotlib 的读者肯定知道绘制一个典型的 heatmap 是较为烦琐的，而 Seaborn 中的 heatmap 函数大大简化了这一过程。

首先，导入 3.3 节中的 data2 文件，对数据进行处理，得到各股在每个月的对数收益率之和：

```
In [163]:rn3 = pd.read_csv('data2.csv', index_col='Date')  #读取数据
         rn3_rets = np.log(rn3.pct_change() + 1)  #计算对数收益率
         rn3_rets.index = [str(x)[5 : 7] for x in rn3_rets.index]  #将 Date
转换为月份
         rn3_group = rn3_rets.groupby(rn3_rets.index).sum()  #数据聚合，得
到每个月的总收益率
         rn3_group.index.name = '月份'
         rn3_group.columns.name = '股票'
In [164]:rn3_group
Out[164]:
```

股票	平安银行	万科 A	格力电器	华东医药	东方雨虹	宇通客车
月份						
01	0.223209	0.040175	-0.005151	-0.101444	-0.195684	0.326332
02	0.129279	-0.020344	0.268781	0.034929	0.035299	0.184665
03	-0.242035	0.139447	0.078918	-0.006467	-0.332323	-0.196243
04	0.137138	0.057530	0.021891	-0.088239	0.160002	-0.015245
05	0.063384	0.195229	0.088088	0.197654	0.343360	0.015641
06	-0.334036	-0.158373	-0.064746	0.021863	0.023249	0.056561
07	-0.027345	-0.020519	0.128542	0.210819	-0.020651	-0.082094
08	0.010338	-0.146732	-0.160302	0.053729	0.085682	-0.049698
09	-0.013845	-0.104127	0.016198	0.005290	-0.072432	-0.057136
10	0.217846	0.095407	0.212994	0.089489	0.249686	-0.089340
11	-0.134181	-0.124814	-0.097552	-0.141878	0.043291	0.167522
12	0.124994	0.114998	0.122309	0.137034	-0.127758	0.205041

用红色表示正收益，用蓝色表示负收益，颜色越深，说明其绝对值越大，效果如图 3-21 所示：

```
In [165]:plt.figure(figsize=(10, 8))
         sns.heatmap(rn3_group, annot=True,  linewidths=.5)
```

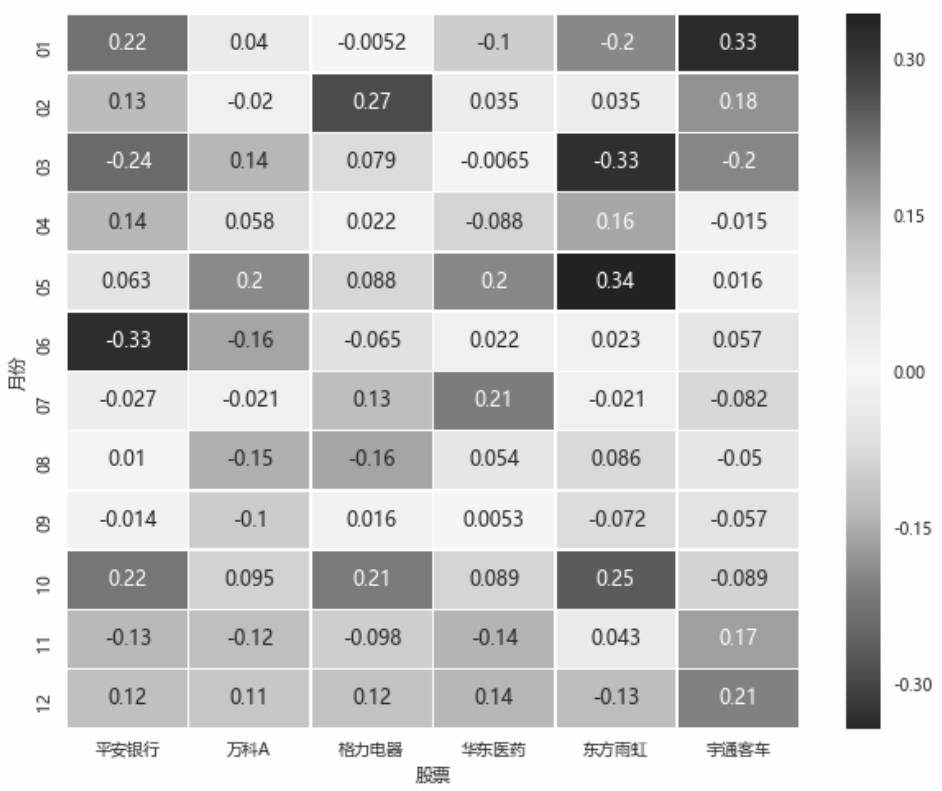


图 3-21

3.5.6 结构网格图

结构网格图可以帮助我们直观地获取各变量之间的关系及其自身的统计信息。如果使用 Matplotlib，则对一个嵌套循环的使用是无法避免的，但是利用 seaborn.pairplot()可以快速达到这一目的，仅需一行代码就可以解决问题。其效果如图 3-22 所示，其中，对角线上的是各股自身的对数收益率的分布图，其余为两两之间的分布关系：

```
In [166]:sns.pairplot(rn3_rets.dropna())
```

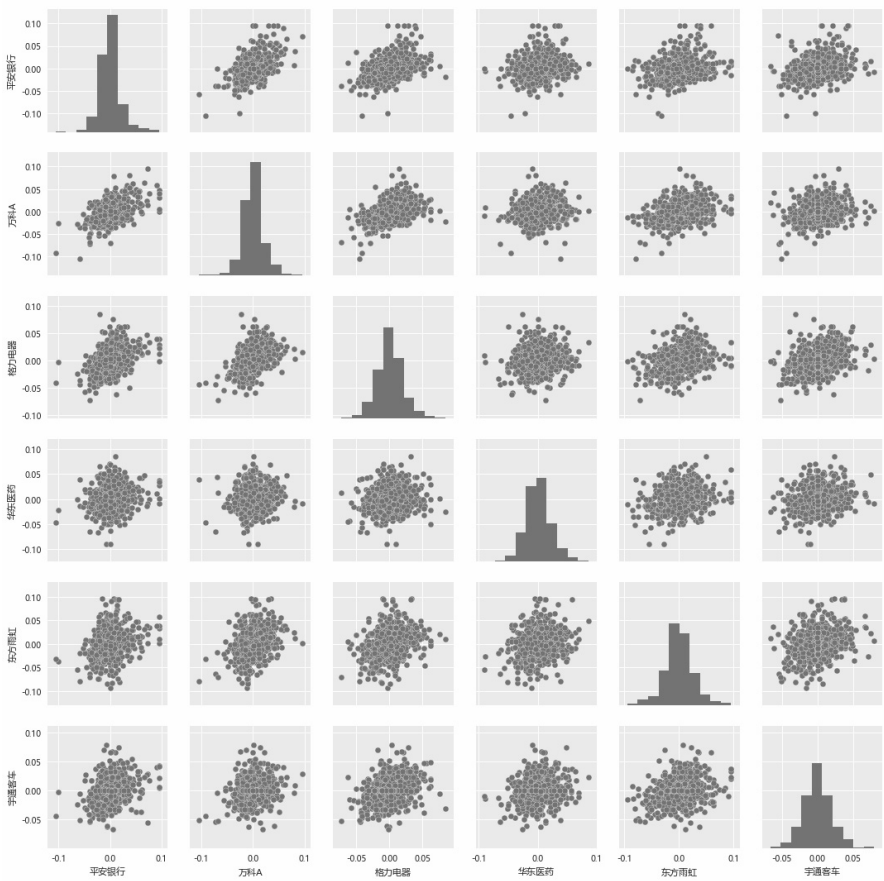


图 3-22

3.6　Scikit-Learn 的初步使用

Scikit-Learn 是利用 Python 进行机器学习常用的第三方模块，这个模块对一些常用的机器学习方法进行了封装，这样在需要进行机器学习时只需调用 Scikit-Learn 里的模块，就可以完成大多数机器学习任务。

机器学习的功能主要包括分类、回归、降维和聚类。主要的分类算法包括决策树（Decision Trees）、贝叶斯分类（Naïve Bayes）、支持向量机（Support Vector Machines）、随机森林（random forest）等，主要的回归算法有 SVR、Lasso 等。常见的降维方法有主

成分分析（PCA）、主题模型（LDA）等。常用的聚类算法有 K-Means、Gaussian 等。同时，Scikit-Learn 还包含了特征提取、数据处理和模型评估这三大模块。

3.6.1 Scikit-Learn 学习准备

1. 安装 Scikit-Learn

Scikit-Learn 目前的版本是 0.17.1，在安装 Scikit-Learn 之前需满足以下条件。

◎ 安装了 Python，版本在 2.6 或者 3.3 以上。

◎ 安装了 NumPy，版本在 1.6.1 以上。

◎ 安装了 SciPy，版本在 0.9 以上。

在满足上述条件后，即可使用 pip install –U scikit-learn 或在 Anaconda 科学计算环境下使用 conda install scikit-learn 命令来安装 Scikit-Learn。

在安装完成之后，可使用以下命令来检查版本：

```
In [167]:import sklearn
          sklearn.__version__
Out[167]:'0.18.1'
```

2. Scikit-Learn 的模块列表

（1）关于数据集的模块有 sklearn.datasets。

（2）关于特征预处理的模块有 sklearn.feture_extraction（特征抽取）、sklearn.feature_selection（特征选择）、sklearn.preprocessing（特征预处理）和 sklearn.random_projection（数据集合）。

（3）关于模型训练的模块有 sklearn.cluster、sklearn.cluster.bicluste、sklearn.linear_model、sklearn.naive_bayes、sklearn.naural_network、sklearn.svm 和 sklearn.tree。

（4）关于模型评估的模块有 sklearn.metrics 和 sklearn.cross_validation。

（5）关于其他功能的模块有 sklearn.covariance 和 sklearn.mixture 等。

3. 将 Scikit-Learn 应用于机器学习

传统的编程方式是人们自己进行数据收集，从而总结其中的经验，然后进行编译汇编，最后交由机器去执行。而机器学习是一种全新的编程方式，由人们提供大量的数据，计算机自己进行归纳总结，然后得到一个训练模型来代替人们进行判断。在 Scikit-Learn 中提供了封装好的算法模型，但在使用 Scikit-Learn 之前，我们需要通过一幅结构图来认识一下机器学习，如图 3-23 所示。

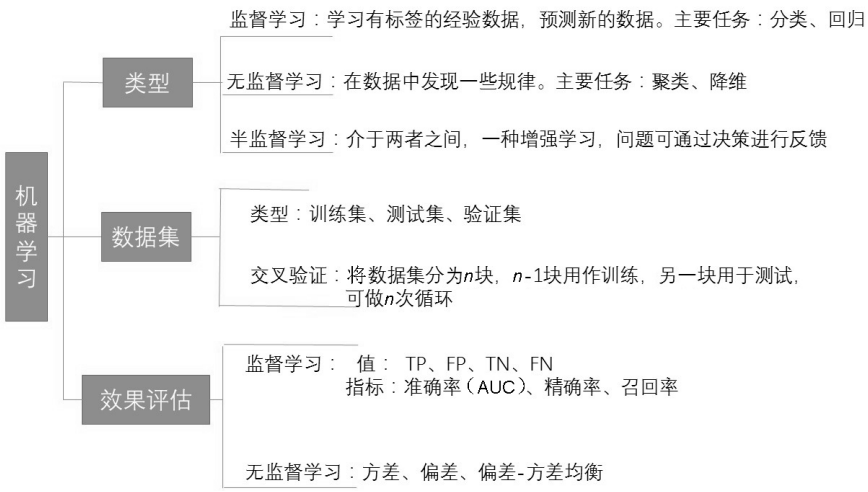


图 3-23

3.6.2　常见的机器学习模型

1. 决策树

决策树（Decision Tree）是最为常见的机器学习方法，通过树状结构判断模型进行决策。以二分类任务为例，我们尝试根据训练数据得到一个模型，从而对新的实例进行分类。例如，在对“是否可以为一个申请人贷款”这样的问题进行决策时，我们通常要进行一系列判断：先看其年龄段，如果是“青年”，则再看申请人目前是否有工作，如果是“有”，则再看申请人是否有房产，如果是“有”，则再看申请人的历史信贷情况如何，如果是“非常好”，则可得出结论：可以给申请人进行贷款。决策实例如图 3-24 所示。

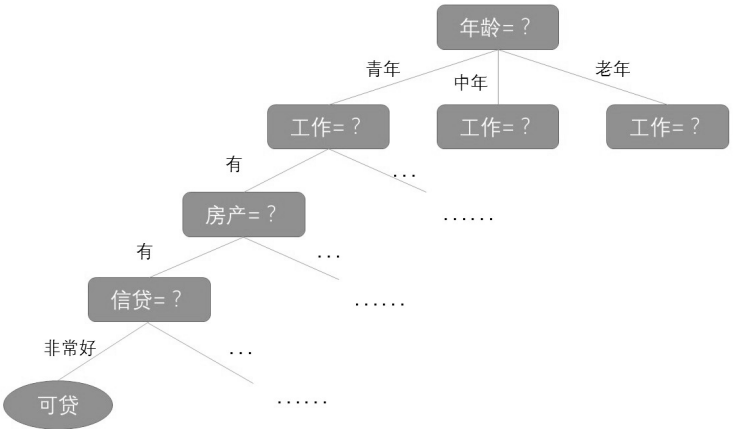


图 3-24

在一般情况下，一棵决策树包含一个根节点、若干内部节点及若干叶节点。根节点包含样本全集，叶节点对应决策结果。每个路径代表一个判定测试序列，对其多次训练以生成一棵泛化能力强、可处理未见示例的决策树。在 Scikit-Learn 中可调用 sklearn.tree 来进行机器学习操作。下面通过实例来了解决策树的工作流程。

假设我们获取了一个数据集，有特征与标签，如表 3-5 所示。

表 3-5

Num	Age	Job	House	Loan	Type
1	青年	是	否	好	是
2	青年	是	是	一般	是
3	中年	是	是	好	是
4	中年	否	是	非常好	是
5	中年	否	是	非常好	是
6	老年	否	是	非常好	是
7	老年	否	是	好	是
8	老年	是	否	好	是
9	老年	是	否	非常好	是
10	青年	否	否	一般	否
11	青年	否	否	好	否

续表

Num	Age	Job	House	Loan	Type
12	青年	否	否	一般	否
13	中年	否	否	一般	否
14	中年	否	否	好	否
15	老年	否	否	一般	否

首先，要进行数据预处理，将中文字符转化为数值型字符，将类别转化为“0”“1”两类，如表 3-6 所示。

表 3-6

Num	Age	Job	House	Loan	Type
1	1	0	1	2	0
2	1	0	0	1	0
3	2	0	0	2	0
4	2	1	0	3	0
5	2	1	0	3	0
6	3	1	0	3	0
7	3	1	0	2	0
8	3	0	1	2	0
9	3	0	1	3	0
10	1	1	1	1	1
11	1	1	1	2	1
12	1	1	1	1	1
13	2	1	1	1	1
14	2	1	1	2	1
15	3	1	1	1	1

接下来划分训练集和测试集，一般采用“留出法”，即直接将数据集划分为两个互斥的集合，将其中一个集合作为训练集 S，将另一个集合作为测试集 T。在划分训练集和测试集时要尽可能保持数据分布的一致性，避免因此引入意外的误差，从而对最终结果产生影响。

“交叉验证法”是划分训练集与测试集的另一种方法，即将数据集划分为 k 个大小相

似的互斥子集，每个子集都尽可能保持数据分布的一致性，然后每次用 k-1 个子集的并集作为训练集，将余下的那个子集作为测试集，这样就可以获得 k 组训练（测试）集，从而进行 k 次训练和测试，最终返回 k 个测试结果的均值。

下面通过调用 sklearn.tree 对训练集进行训练：

```
In [168]:import matplotlib.pyplot as plt
          import numpy as np
          import pandas as pd
          import sklearn
          import re
          from sklearn import cross_validation
          import sklearn.tree as tree
          data=pd.read_excel('loan.xlsx')
          target=data['Type']
          data.drop('Type',axis='columns',inplace=True)
          train_data,test_data,train_target,test_target=cross_validation.
train_test_split(data,target,test_size=0.4,train_size=0.6,random_state=12345)
          clf_1=tree.DecisionTreeClassifier(criterion='entropy')
          clf_1.fit(train_data,train_target)
          train_est=clf_1.predict(train_data)
          train_est_p=clf_1.predict_proba(train_data)[:,1]
```

将模型应用于测试集：

```
In [169]:test_est=clf_1.predict(test_data)
```

在模型对新的示例进行判断后，我们希望在对模型进行评估时分支节点所包含的样本尽可能属于同一个类，即节点的纯度越高越好，一般用信息熵、增益率和基尼指数来代表纯度。

我们进行机器学习的目标是使机器学得的模型很好地适用于“新样本”，而不是仅仅在训练样本上工作得好。我们将模型适用于新样本的能力称为“泛化能力”。我们通常希望一个模型的训练误差及泛化误差都很低，其中，泛化误差低是我们首先要考虑的因素：

```
In [170]:from sklearn import metrics
          print metrics.accuracy_score(test_target, test_est)
Out[170]:0.5
```

我们根据真实类别和测试集预测类别的组合划分出真正例、假正例、真反例、假反例这四种情形，分别用 TP、FP、TN、FN 来表示。若已知 TP+FP+TN+FN=样例总数，则可画出学习结果的混淆矩阵，如表 3-7 所示。

表 3-7

真 实 情 况	预 测 结 果	
	正　例	反　例
正例	TP（真正例）	FN（假反例）
反例	FP（假正例）	TN（真反例）

可以通过下面的代码查看混淆矩阵：

```
In [171]:print metrics.confusion_matrix(test_target, test_est)
Out[5]:
          [[2 3]
          [0 1]]
```

我们关心的两个指标为查全率与查准率，其中，查准率的计算方式如下：

$$P=TP/(TP+FP)$$

查全率的计算公式如下：

$$R=TP/(TP+FN)$$

在本例中，查准率为 100%，查全率为 40%。

2. 支持向量机

支持向量机（Support Vector Machine，SVM）是一个二类分类器，其中，Vector 作为点，是数据；Machine 是分类器。作为一种前馈神经网络类型，SVM 是近几十年来在机器学习中表现较好的算法。

给定一个训练样本，SVM 会建立一个超平面作为决策面，将正反例样本区分开，并使得隔离边界最大化。下面我们通过一个实例来了解 SVM 的原理。

假设桌上有红、蓝两色球，我们寻找一条曲线将其分开，如图 3-26 中的黑色曲线所示。

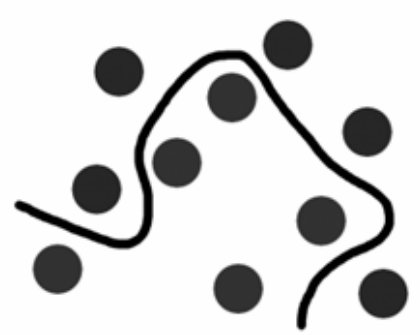


图 3-26

SVM 的原理是我们需要寻找一条最优的曲线，使其距离任意红球和蓝球的距离最大化，如图 3-27 所示。

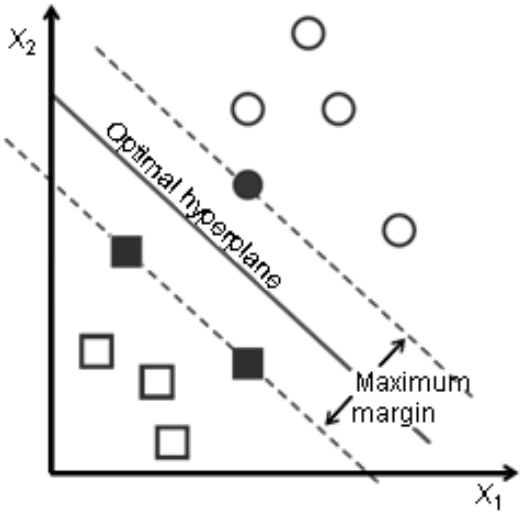


图 3-27

我们将这些球抛到空中时，便可用一个曲面将正反两例区分开，这个平面距离所有球的距离仍是最大的，这个平面就是我们寻找的超平面。距离这个曲面最近的红球和蓝球就是支持向量，如图 3-28 所示。

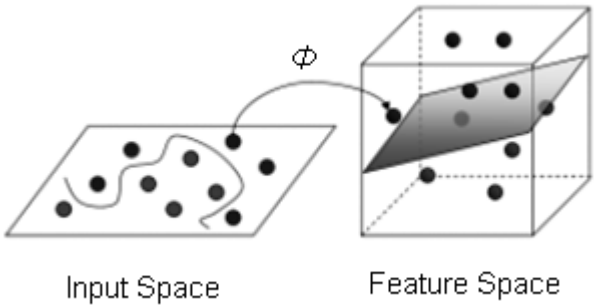


图 3-28

本书主要讲解如何将 Python 应用于量化投资，因此这里不对各类机器学习算法进行详细推理和展开，主要展示如何应用各类算法。基于上面的决策树是否进行放贷的实例，我们利用 SVM 进行学习。数据处理、数据集划分的方法同上，这里主要给出通过 SVM 进行模型训练及预测的代码：

```
In [172]:import sklearn.svm as svm
          clf_2=svm.SVC()
          clf_2.fit(train_data,train_target)
          train_est=clf_2.predict(train_data)
          test_est=clf_2.predict(test_data)
```

查看模型效果的代码如下：

```
In [173]:from sklearn import metrics
          print metrics.accuracy_score(test_target, test_est)
          print metrics.confusion_matrix(test_target, test_est)
Out[173]:
          0.666666666667
          [[3 2]
          [0 1]]
```

可以发现，模型预测的准确率为 0.67。根据混淆矩阵可以看出，相较于决策树模型，本次学习仅有两个被误判，正确率有所提高。

SVM 算法具有较高的准确率，可解决高维问题，能够处理非线性特征的相互作用。但在高维问题中比较消耗内存，调参也有一定难度，对于非线性问题没有通用的解决方案，有时很难找到合适的核函数。

3. 朴素贝叶斯分类器

贝叶斯决策论是在概率框架下实施决策的基本方法。在所有相关概率都已知的理想情况下，贝叶斯决策论考虑如何通过这些概率进行判断，从而选择最优的类别标记。朴素贝叶斯算法基于贝叶斯定理：

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}$$

对于待分类实例，朴素贝叶斯分类器可给出在此项出现的条件下哪个类别出现的概率最大，就将此待分类实例归属于该类别。具体过程如下。

（1）设 $X = \{c1, c2, \cdots cm\}$ 为一个待分类实例，每个 c 为 X 的一个特征属性。

（2）有类别集和 $T = \{y1, y2, \cdots yn\}$，$y$ 为每个具体类别。

（3）计算 $P(y1 \mid X), P(y2 \mid X), \cdots P(yn \mid X)$，如果 $\max\{P(y1 \mid X), P(y2 \mid X), \cdots P(yn \mid X)\} = P(yk \mid X)$，则 $X \in yk$。

同样，基于上面的决策树是否进行放贷的实例，我们利用朴素贝叶斯算法进行学习。数据处理、数据集划分的方法同上，在此主要给出通过朴素贝叶斯进行模型训练及预测的代码：

```
In [174]:from sklearn.naive_bayes import GaussianNB
          clf_3=GaussianNB()
          clf_3.fit(train_data,train_target)
          train_est=clf_3.predict(train_data)
          test_est=clf_3.predict(test_data)
In [175]:from sklearn import metrics
          print metrics.accuracy_score(test_target, test_est)
          print metrics.confusion_matrix(test_target, test_est)
Out[175]:
          0.5
          [[2 3]
          [0 1]]
```

在本例中，朴素贝叶斯算法与决策树算法所得的结果一致。朴素贝叶斯算法有其自身的优点，例如起源于古典数学理论，有着坚实的数学基础及稳定的分类效率，因而常被用于文本分类，而且在小规模数据处理上表现较好。同时，朴素贝叶斯也存在一些缺点，例如需要计算先验概率、对数据的输入性比较敏感等。

4. 神经网络

神经网络目前已成为一个相当大的多学科交叉的学科领域。人工神经网络模拟生物神经网络，用于解决分类和回归问题，是一类模式匹配算法。其中最基本的成分是神经元模型，人工神经网络由一系列简单的单元相互连接而成，每个单元都有一定数量的实值输入，经过激活函数的处理，可产生单一的实值输出。主要结构如图 3-29 所示。

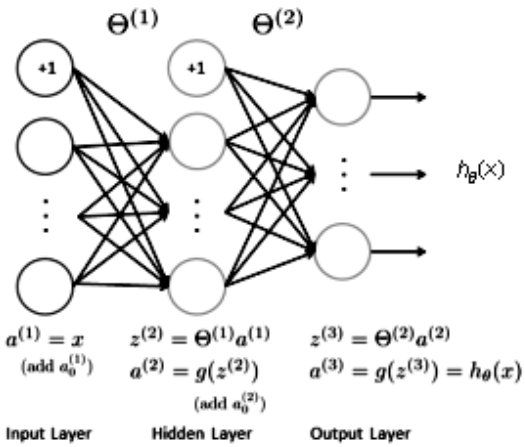


图 3-29

基于上面所示的决策树是否进行放贷的实例，我们利用人工神经网络进行学习。数据处理、数据集划分的方法同上，在此主要给出通过人工神经网络进行模型训练及预测的代码：

```
In [176]:from sklearn.neural_network import MLPClassifier
          clf_4=MLPClassifier()
          clf_4.fit(train_data,train_target)
          train_est=clf_4.predict(train_data)
          test_est=clf_4.predict(test_data)
In [177]:from sklearn import metrics
          print metrics.accuracy_score(test_target, test_est)
          print metrics.confusion_matrix(test_target, test_est)
Out[177]:
          1.0
          [[5 0]
          [0 1]]
```

我们惊讶地发现，人工神经网络算法的分类准确率达到 100%，但这只是针对个别例子。人工神经网络具有分类准确率高、可进行并行处理和分布式存储及学习能力强等众多优点，同时能充分逼近复杂的非线性网络。但神经网络也有一定的缺点，例如学习时间过长、参数较多、输出结果难以解释等。

3.6.3 模型评价方法——metric 模块

我们在对模型效果做评价时已经用到了 metrics 模块的 accuracy_score 和 confusion_matrix 方法，下面将详细介绍 metrics 模块的使用。

分类任务模型常用的评价指标包括分类准确率分数、混淆矩阵、召回率、ROC 曲线、AUC 值等。每个指标都有其含义，针对不同的问题，我们所关注的评价指标也有所不同。

1. accuracy_score（分类准确率）

分类准确率分数指所有分类正确数目的百分比。

使用方法如下：

```
metrics.accuracy_score(y_true, y_pred, normalize=True, sample_weight=None)
```

其中，y_true 为测试集的真实类别，y_pred 为测试集的预测类别。normalize 的默认值为 True，返回正确分类的比例；如果为 False，则返回正确分类的样本数。

示例如下：

```
In [178]:import numpy as np
          from sklearn.metrics import accuracy_score
          y_pred = [0, 2, 1, 3]
          y_true = [0, 1, 2, 3]
          print(accuracy_score(y_true, y_pred))
          print(accuracy_score(y_true, y_pred, normalize=False))
Out[178]:
          0.5
          2
```

但是 accuracy 并不会告诉我们错误的类型，因此我们需要借助其他指标进行分析。

2. confusion_matrix（混淆矩阵）

我们在上面提到过，针对由 TP、FP、FN、TN 组成的预测值（真实值），矩阵可以帮助我们分辨被正确分类及错误分类的样本数。

使用方法如下：

```
metrics.confusion_matrix(y_true, y_pred, labels=None, sample_weight=None)
```

其中，y_true 为测试集的真实类别，y_pred 为测试集的预测类别。Labels 表示混淆矩阵的索引，若未赋值，则按照 y_true、y_pred 中出现过的值进行排序。示例如下：

```
In [179]:from sklearn.metrics import confusion_matrix
         y_true = [2, 0, 2, 2, 0, 1]
         y_pred = [0, 0, 2, 2, 0, 2]
         confusion_matrix(y_true, y_pred)
Out[179]:
         array([[2, 0, 0],
                [0, 0, 1],
                [1, 0, 2]])
```

对角线代表被正确分类的样本数，其余位置均代表被错误分类的样本数：

```
In [180]:y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
         y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
         confusion_matrix(y_true, y_pred, labels=["ant", "bird", "cat"])
Out[180]:
         array([[2, 0, 0],
                [0, 0, 1],
                [1, 0, 2]])
```

在本例中，0 代表 ant，1 代表 bird，2 代表 cat。混淆矩阵结果表明：两只 ant 均被正确识别，在 3 只 cat 中有两只被正确识别，有一只被识别为 ant，而唯一的 bird 被识别为 cat。

3. roc_curve（ROC 曲线）

ROC 曲线指接收者操作特征曲线（Receiver Operating Characteristic Curve），根据一系列不同的二分类方式，以真阳性率（灵敏度）为纵坐标，以假阳性率（1-特异度）为横坐标绘制曲线，可体现敏感性与特异性的相互关系。其中，纵坐标（真阳性率 TPR 或灵敏

度）的计算公式如下：

$$TPR=TP/(TP+FN)$$

横坐标（假阳性率 FPR）或特异度的计算公式如下：

$$FPR=FP/(FP+TN)$$

另外，ROC 曲线下的面积 AUC（Area Under Curve）是对模型准确率的度量。

ROC 曲线的使用方法如下：

```
    metrics.roc_curve(y_true, y_score, pos_label=None, sample_weight=None,
drop_intermediate=True)
```

其中，y_true 为测试集的真实类别，y_score 为测试集属于正值样本的概率。该函数有三个返回变量：fpr、tpr 和阈值 thresholds。thresholds 为一阈值，当 score 大于或等于它时，我们认为它为正样本，否则为负样本。每取一个不同的 thresholds，便可得到一对互相对应的 FPR 和 TPR，即 ROC 曲线上的一个点，将这些点连起来便构成了 ROC 曲线，效果如图 3-25 所示：

```
    In [181]:import numpy as np
             from sklearn import metrics
             y = np.array([1, 1, 2, 2])
             scores = np.array([0.1, 0.4, 0.35, 0.8])
             fpr, tpr, thresholds = metrics.roc_curve(y, scores, pos_label=2)
             print(fpr)
             print(tpr)
             print(thresholds)
    Out[181]:
             array([ 0. ,  0.5,  0.5,  1. ])
             array([ 0.5,  0.5,  1. ,  1. ])
             array([ 0.8 ,  0.4 ,  0.35,  0.1 ])
    In [182]:%matplotlib inline
             plt.plot(fpr,tpr)
    Out[182]:
```

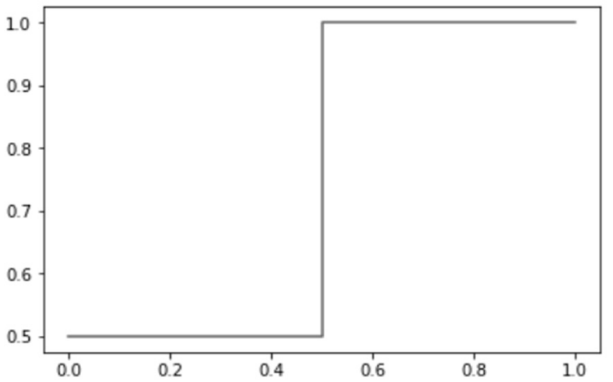


图 3-25

4. recall_score（召回率）

召回率为将提取出正确的信息条数除以样本总信息条数的计算结果，即有多少样本被正确识别。其使用方法如下：

```
    metrics.recall_score(y_true, y_pred, labels=None, pos_label=1,
average='binary', sample_weight=None)
```

其中，y_true 为测试集的真实类别，y_pred 为测试集的预测类别。average 参数适用于多分类问题，可取值 macro、micro、samples、wighted 和 None。macro 为每个分类给出相同的权重，从而计算二分类 metrics 的均值；micro 给定每个样本类权重来计算整个份额；samples 通过计算真实类和预测类的差异的 metrics 来求平均值；weighted 对不均衡数量的类计算二分类 metrics 的平均值；None 将返回一个数组，包含每个类的得分。

示例如下：

```
    In [183]:from sklearn.metrics import recall_score
             y_true = [0, 1, 2, 0, 1, 2]
             y_pred = [0, 2, 1, 0, 0, 1]
             recall_score(y_true, y_pred, average='macro')
    Out[183]:
             0.33...
    In [184]:recall_score(y_true, y_pred, average='micro')
    Out[184]:
             0.33...
    In [185]:recall_score(y_true, y_pred, average='weighted')
```

```
Out[185]:
          0.33...
In [186]:recall_score(y_true, y_pred, average=None)
Out[186]:
          array([ 1.,  0.,  0.])
```

决策树算法易于理解，计算简单，能够处理有缺失属性的样本及不相关的特征，在短时间内可对较大的数据做出可行且效果良好的判断。但决策树也存在一些问题，比如忽略了数据之间的相关性、容易出现过拟合等。

3.6.4 深度学习

提高模型复杂度容易导致训练效率降低、过拟合等。随着大数据、云计算时代的到来，越来越多的问题的数据量级迅速增大，计算机处理能力也随之大幅提升。深度学习算法是对人工神经网络的进一步发展。建立庞大、复杂的神经网络的一个简单办法就是增加隐藏层的数量，从而增加了拥有激活函数的神经元数量和激活函数嵌套的层数。

深度学习可分为有监督学习和无监督学习。不同的模型适用于不同的学习框架。卷积神经网络（Convolutional neural networks，CNN）是在有监督学习下的常见模型，深度置信网络（Deep Belief Nets，DBNs）则是在无监督学习下的常用模型。深度学习目前已在计算机视觉、语音识别、自然语言处理等领域得到较好的应用，在量化投资方面也有很大的开发潜力。

3.7 SQLAlchemy 与常用数据库的连接

在生产环境中我们不可能总是通过 Excel 或者 CSV 文件读取或存储数据，因为对于大量的数据来说，这种方法十分低效，更好的方法是通过连接数据库来进行相关操作，本节将着重介绍这种方法。

SQLAlchemy 是 Python 下的一款开源库，提供了 SQL 工具包及对象关系映射（ORM）工具，采用了简单的 Python 语言，为高效和高性能的数据库访问设计实现了完整的企业级持久模型。我们可以使用 SQLAlchemy 进行传统的数据库操作“增删改查”，使得 Python 也具有了 Web 开发的能力。对于量化投资而言，我们的需求也是类似的，在很多时候我们需要通过连接 wind 底层数据库来获取股票数据。在做多因子策略时，我们也会对某一

子集合成的因子进行存储以方便下次直接调用。本节先介绍如何使用 SQLAlchemy 进行数据库连接。

3.7.1　连接数据库

SQLAlchemy 支持大部分主流数据库，例如 SQLite、MySQL、PostgreSQL、Oracle、MS-SQL 及 Firebird 等。为了连接数据库，我们首先需要安装数据库驱动。

表 3-8 罗列了不同的数据库所对应的驱动及连接的写法，可以通过 pip insall xxxx 来安装驱动，如果遇到安装问题，Windows 用户则可以尝试前往 https://www.lfd.uci.edu/~gohlke/pythonlibs/下载对应的二进制安装文件。

表 3-8

数 据 库	驱　　动	连接的写法
SQLite	Python 3 自带	sqlite:///path/to/database.db（相对路径）
		sqlite: ////path/to/database.db（绝对路径）
		sqlite:///C:\\path\\to\\database.db（Windows 的绝对路径）
MySQL	mysqlclient	mysql://user:password@host:port/dbname
MS-SQL	pymssql	mssql+pymssql://user:password@ host:port/dbname
PostgreSQL	psycopg2	postgresql+psycopg2://user:password@host:port/dbname
Oracle	cx-Oracle	oracle+cx_oracle://user:password@host:port/dbname
Firebird	fdb	firebird+fdb://user:password@host:port/path/to/db

在安装完数据库驱动之后，我们就可以开始连接数据库了，这里使用本地的 MySQL 数据库作为范例：

```
    In [187]:from sqlalchemy import create_engine
              from sqlalchemy.ext.automap import automap_base
              import pandas as pd
              import numpy as np
    In [188]:engine = create_engine('mysql://root:123@127.0.0.1:3306/ test?
charset=utf8')
```

其中，root 为用户名，123 为密码，127.0.0.1:3306 为 IP 地址及端口号，test 为数据库名。

3.7.2 读取数据

在连接数据库之后，我们可以结合 Pandas 中的 read_sql 方法查询数据库中的数据并直接返回 DataFrame，在方法的参数中可以传入 SQL 语句：

```
In [189]: pd.read_sql('select * from data', engine)
Out[189]:
```

	ID	stockname	price
0	1	格力电器	10
1	2	中国平安	34
2	3	浦发银行	16
3	4	万科 A	25

也可以直接传入表名：

```
In [190]:pd.read_sql('data',engine)
Out[190]:
```

	ID	stockname	price
0	1	格力电器	10
1	2	中国平安	34
2	3	浦发银行	16
3	4	万科 A	25

3.7.3 存储数据

Pandas 中的 to_sql 方法支持将 DataFrame 类型的数据方便、快速地存储到数据库中。

首先，构造待插入的数据：

```
In [191]:df = pd.DataFrame([[5, '永辉超市', 11], [6, '华夏幸福', 34]],
                columns=['ID', 'stockname', 'price'],
                index=range(2))
          df
Out[191]:
```

	ID	stockname	price
0	5	永辉超市	11
1	6	华夏幸福	34

然后，将该数据插入 data 表中，这里要注意 if_exists=…参数的传入，可选的选项分别为'fail'（如果原表存在，则不执行操作）、'append'（如果原表存在，则插入数据；如果原表不存在，则新建表并插入）、'replace'（如果原表存在，则清空该表并重新插入数据）：

```
In [192]: df.to_sql('data', engine, index=False, if_exists='append')
In [193]: pd.read_sql('data',engine)
Out[193]:
```

	ID	stockname	price
0	1	格力电器	10
1	2	中国平安	34
2	3	浦发银行	16
3	4	万科 A	25
4	5	永辉超市	11
5	6	华夏幸福	34

最后，插入一张新表：

```
In [194]: df.to_sql('t_data', engine, index=False, if_exists='append')
In [195]: pd.read_sql('t_data',engine)
Out[195]:
```

	ID	stockname	price
0	5	永辉超市	11
1	6	华夏幸福	34

Pandas 内置的 to_sql 方法十分便捷，但是也有局限性，只能针对 Dataframe 类型的数据进行存储，无法存储应用更为普遍的 dict 类型的数据，幸好 SQLAlchemy 为我们提供了解决方法。

首先，构造一个 DataFrame 及其对应的字典：

```
    In [196]:df1 = pd.DataFrame(np.arange(20000).reshape(10000, 2), index=
range(10000), columns=['key', 'value'])
              r = df1.to_dict('records')
```

同样，利用上面介绍的 to_sql 方法将其存入 f_data 表中，没有任何问题：

```
    In [197]:df1.to_sql('f_data', engine, index=False, if_exists='append')
              pd.read_sql('f_data', engine).tail()
    Out[197]:
```

	key	value
9995	19990	19991
9996	19992	19993
9997	19994	19995
9998	19996	19997
9999	19998	19999

然后，使用 SQLAlchemy 的方法，先将 f_data 表实例化为 SQLAlchemy 支持的类型：

```
    In [198]:# 下面这两句话就完成了 ORM 映射，Base.classes.XXXX 就是映射的类
              # Base.metadata.tables['XXX']就是相应的表
              Base = automap_base()
              Base.prepare(engine, reflect = True)
              f_data = Base.metadata.tables['f_data']
```

再对实例化后的 f_data 执行 insert 方法，也能快速、批量地插入数据，注意这里的 r 为 dict 类型：

```
    In [199]:engine.execute(f_data.insert(), r)
              pd.read_sql('f_data', engine).tail()
    Out[199]:
```

	key	value
19995	19990	19991
19996	19992	19993
19997	19994	19995
19998	19996	19997
19999	19998	19999

第4章

常用数据的获取与整理

4.1 金融数据类型

数据可以说是量化投资的根本，一切投资策略都是建立在数据基础上的。总的来说，证券投资有很多类型的数据，如下所述。

- 股票：指沪深交易所的股票的基本信息及股票行情数据。
- 财务报表：指上市公司披露的所有财务报表数据，包含三大报表和财报附注等细节。
- 公司行为：指上市公司的业绩预告、业绩快报、IPO、配股、分红、拆股、股改等信息。
- 基金：指场内外各类基金的基本信息、场内基金行情、场外基金净值，以及基金资产配置、收益情况、净值调整等信息。
- 期货：指国内四大期货交易所期货合约的基本信息、期货行情及国债期货的转换因子等信息。
- 指数：指国内外指数的基本信息、指数行情、指数成分构成情况及指数成分股权

重情况等信息。

- 港股：指香港交易所股票的基本信息及股票行情。
- 大宗商品：指国内各个品种（包括期货合约可交割品种）的大宗商品现货价格行情、产销量、库存等信息。
- 债券：指债券或者回购基本信息、日级别的债券或者回购行情，以及发行上市、付息、利率、评级和评级变动、债券发行人评级及变动、担保人评级及变动等信息。
- 期权：指上交所期权合约的基本信息、期权行情及每日盘前静态数据等信息。
- 宏观产业：指中国及全球各国的宏观指标、行业经济指标等数据。

以上是传统的结构化的投资数据。随着社交网络的发展，更多的非结构化数据进入了我们的投资视野，例如：

- 雪球、股吧等社交媒体数据；
- 主流媒体新闻文本和结构化数据；
- 主流渠道公告文本和结构化数据；
- 淘宝、天猫等电商数据。

同时，我们很少在策略中直接使用以上原始数据，而是建立衍生数据，使之成为量化因子。

常见的量化因子如下。

- 常用的技术指标类：指主流的技术指标、使用前的复权价格计算，反映了股票的量价信息，例如 hurst 等指标。
- 每股指标：指每股的相关财务指标，例如 EPS、每股企业自由现金流等。
- 价值类：体现市场对公司的估值大小，例如 PB、PE 等。
- 质量类：体现公司的赢利能力与收益质量，例如营业利润率，反映了公司的营业利润（经过 TTM 调整）占营收的比例。
- 动量类：衡量股票价格的“惯性效应”。
- 成长类：反映公司的成长性，例如净利润增长率。

◎ 情绪类：指基于量价关系来衡量市场参与者的情绪。

◎ 收益与风险类因子：指与收益风险相关的因子，例如股票的 60 日 alpha 和 beta。

◎ 基础科目与衍生科目：是财报数据的衍生，例如净利润的 TTM 数值、息税前净利润（EBIT）等。

◎ 分析师预期：根据分析师的一致预期数据计算，体现了分析师对个股趋势的预测。

4.2　金融数据的获取

我们可以通过多种途径获取金融数据，业内的许多公司会购买 Wind、恒生聚源等数据提供商的数据库，若尚未入行，则也可以通过非常多的第三方策略平台获取免费数据，例如优矿、聚宽、米筐等。优矿依托通联数据，提供了丰富的数据信息，这里主要介绍如何在优矿中调用获取金融数据。

首先，可以在优矿官网（https://uqer.io）注册一个账号，然后单击“研究数据”模块，如图 4-1 所示。

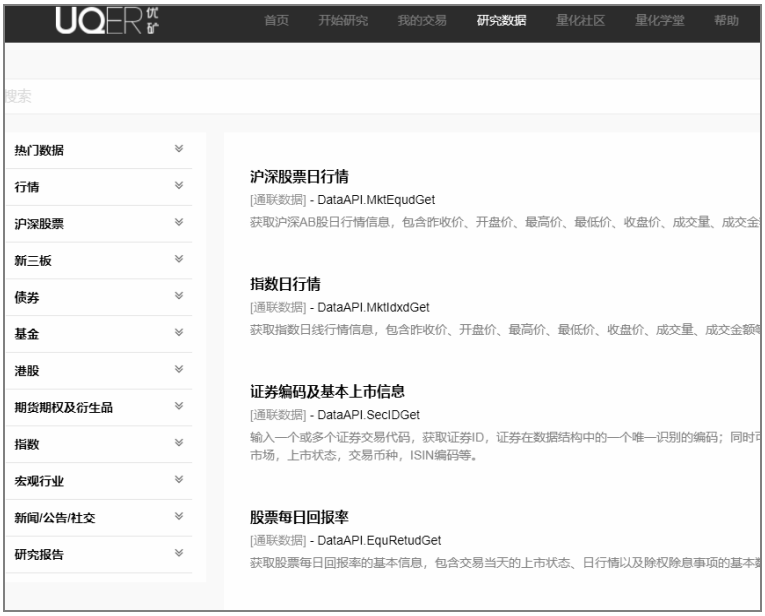


图 4-1

可以看到，这些数据基本涵盖了大部分金融数据，而且绝大部分是免费的。我们以一些最常用的金融数据来展示如何对它们进行调用。

单击“开始研究”模块，在左侧找到新建按钮，新建一个 Notebook，再单击对应的 Notebook，便进入 Python 代码的编辑环境。如果使用过 Jupyter Notebook 或者 IPython Notebook，则可以很快发现优矿中的 Notebook 编译环境与之非常相似。实际上，优矿中的 Notebook 就是基于 IPython Notebook 开发的，基本操作基本一致，可以很快上手。

我们将 NoteBook 左上角的模式设置为代码模式，开始调用数据。首先，打开一个浏览器新窗口并回到优矿的“研究数据”模块，寻找我们需要的数据。假定需要股票日行情数据，则我们既可以通过左边的一级选项一步一步地往下点来找到数据，也可以直接通过关键字在上方的搜索框中进行搜索。然后，我们找到了沪深股票的日行情数据，单击“展开详情”，结果如图 4-2 所示。

沪深股票日行情

[通联数据] - DataAPI.MktEqudGet

```
DataAPI.MktEqudGet(tradeDate=u"20150513",secID=u"",ticker=u"",beginDate=u"",endDate=u"",isOpen="",field=u"",pandas="1")
```

获取沪深AB股日行情信息，包含昨收价、开盘价、最高价、最低价、收盘价、成交量、成交金额等字段，每日16:00更新

参数

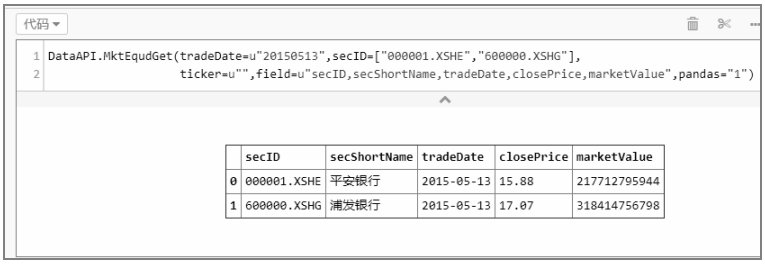
名称	类型	描述
tradeDate	str	输入一个日期，不输入其他请求参数，可获取到一天全部沪深股票日行情数据，输入格式"YYYYMMDD",tra 选择一个
secID	list	通联编制的证券编码，格式是"交易代码.证券市场代码"，如000001.XSHE。 可传入证券交易代码使用Data 可以是列表,tradeDate、secID、ticker至少选择一个
ticker	list	股票交易代码，如'000001'（可多值输入），可以是列表,tradeDate、secID、ticker至少选择一个
beginDate	str	起始日期，输入格式"YYYYMMDD",可空
endDate	str	截止日期，输入格式"YYYYMMDD",可空
isOpen	int	股票今日是否开盘标记位：0-今日未开盘，1-今日有开盘,可空
field	list	所需字段,可以是列表,可空
pandas	str	1表示返回 pandas data frame，0表示返回csv,可空

图 4-2

在详情中展示了各个参数的含义，例如，在输入时我们需要给出要调用哪些股票及在什么时间段的行情。而在返回值中，我们可选的行情不仅包括基本的高开低收、成交量、涨跌幅，也包括流通市值、VWAP 等数据，应将所有选择返回的项传给 field 参数。在 Notebook 中的代码编写如图 4-3 所示。

如图 4-3 所示，我们调取了平安银行与浦发银行在 2015 年 5 月 13 日的收盘价与总市值，返回的是一个 DataFrame。需要注意的是，这里传入的股票代码是通联内部的编码，要在原股票代码的基础上加上后缀，例如对深市的股票加上后缀 XSHE，对沪市的股票则

加上后缀 XSHG。当然，也支持直接传入股票代码，可以将原始股票的代码传入 ticker 参数，如图 4-4 所示。



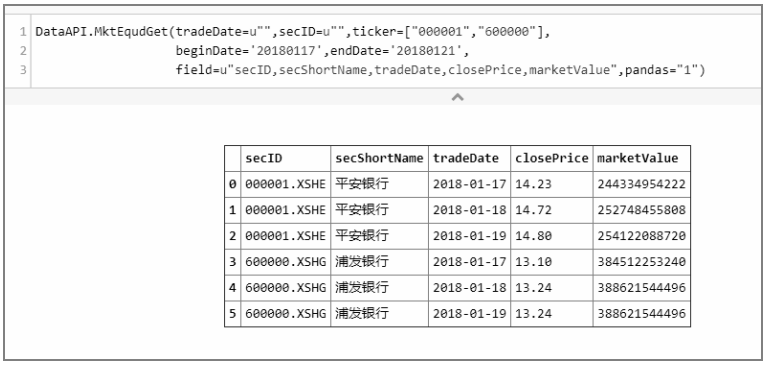
	secID	secShortName	tradeDate	closePrice	marketValue
0	000001.XSHE	平安银行	2015-05-13	15.88	217712795944
1	600000.XSHG	浦发银行	2015-05-13	17.07	318414756798

图 4-3

	secID	secShortName	tradeDate	closePrice	marketValue
0	000001.XSHE	平安银行	2015-05-13	15.88	217712795944
1	600000.XSHG	浦发银行	2015-05-13	17.07	318414756798

图 4-4

除了某一天的截面数据，DataAPI 也支持获取过去一时间段的数据，通过 beginDate 与 endDate 参数即可方便地获取。如图 4-5 所示，我们调取了平安银行与浦发银行从 2018 年 1 月 17 日至 2018 年 1 月 19 日的收盘价与总市值数据。



	secID	secShortName	tradeDate	closePrice	marketValue
0	000001.XSHE	平安银行	2018-01-17	14.23	244334954222
1	000001.XSHE	平安银行	2018-01-18	14.72	252748455808
2	000001.XSHE	平安银行	2018-01-19	14.80	254122088720
3	600000.XSHG	浦发银行	2018-01-17	13.10	384512253240
4	600000.XSHG	浦发银行	2018-01-18	13.24	388621544496
5	600000.XSHG	浦发银行	2018-01-19	13.24	388621544496

图 4-5

股票日行情 DataAPI 支持提取多只股票在某一时间段的数据，我们再来看看量化投资常用的因子数据，如图 4-6 所示。

图 4-6

在搜索因子中排名前两位的就是我们需要的 DataAPI。可以看到有两个因子数据接口：一个用于获取多只股票在某一天的因子数据，另一个用于获取某只股票在历史上某一时间段的因子数据。可能是出于对数据量的考虑，并没有一个因子 DataAPI 可以直接调用多只股票在某一时间段的数据，所以我们在使用优矿的因子 DataAPI 时，应当考虑哪个 DataAPI 会更适合我们的需求。

如果只是想调用某一天的多只股票的因子数据，则应该使用如下 DataAPI，如图 4-7 所示。

```
DataAPI.MktStockFactorsOneDayGet(tradeDate=u"20150227",secID=["000001.XSHE","600000.XSHG"],ticker=u"",
                                 field=u"secID,tradeDate,PS,PB,NetProfitGrowRate",pandas="1")
```

	secID	tradeDate	PS	PB	NetProfitGrowRate
0	000001.XSHE	2015-02-27	2.2999	1.2612	0.2938
1	600000.XSHG	2015-02-27	2.3104	1.1716	0.2176

图 4-7

而如果想调用一只股票在某一时间段的因子数据，则应该使用另一个 DataAPI，如图 4-8 所示。

```
DataAPI.MktStockFactorsDateRangeGet(secID=u"",ticker=u"600000",beginDate=u"20170612",endDate=u"20170616",
                                    field=u"secID,tradeDate,PS,PB,NetProfitGrowRate",pandas="1")
```

	secID	tradeDate	PS	PB	NetProfitGrowRate
0	600000.XSHG	2017-06-12	2.2314	1.0258	0.0106
1	600000.XSHG	2017-06-13	2.1982	1.0105	0.0106
2	600000.XSHG	2017-06-14	2.1755	1.0001	0.0106
3	600000.XSHG	2017-06-15	2.1615	0.9937	0.0106
4	600000.XSHG	2017-06-16	2.1563	0.9913	0.0106

图 4-8

如果我们还是想获取多只股票在某一时间段的因子数据，则可以写循环来多次调用 DataAPI，将它们全部取出来，如图 4-9 所示。

```
import pandas as pd
df = pd.DataFrame()
for ticker in ["600000",'000001']:
    tmp_df = DataAPI.MktStockFactorsDateRangeGet(secID=u"",ticker=ticker,beginDate=u"20170612",endDate=u"20170616",
                                                 field=u"secID,tradeDate,PS,PB,NetProfitGrowRate",pandas="1")
    df = pd.concat([df,tmp_df],axis=0)
df.reset_index(drop=True)
```

	secID	tradeDate	PS	PB	NetProfitGrowRate
0	600000.XSHG	2017-06-12	2.2314	1.0258	0.0106
1	600000.XSHG	2017-06-13	2.1982	1.0105	0.0106
2	600000.XSHG	2017-06-14	2.1755	1.0001	0.0106
3	600000.XSHG	2017-06-15	2.1615	0.9937	0.0106
4	600000.XSHG	2017-06-16	2.1563	0.9913	0.0106
5	000001.XSHE	2017-06-12	1.4498	0.8330	0.0181
6	000001.XSHE	2017-06-13	1.4514	0.8339	0.0181
7	000001.XSHE	2017-06-14	1.4450	0.8302	0.0181
8	000001.XSHE	2017-06-15	1.4386	0.8266	0.0181
9	000001.XSHE	2017-06-16	1.4354	0.8248	0.0181

图 4-9

除了这些常用的金融数据，研究数据模块还包括财报数据、事件数据、期货数据等，我们可以通过搜索或者分类选项找到它们，在详情中对相应的参数有详细的解释。

4.3　数据整理

仅仅知道如何获取数据是不够的，我们还需要将原始数据整理成正确的、便于我们进一步使用的数据。下面展示一些常用的数据整理理念及 Python 的实现方法。

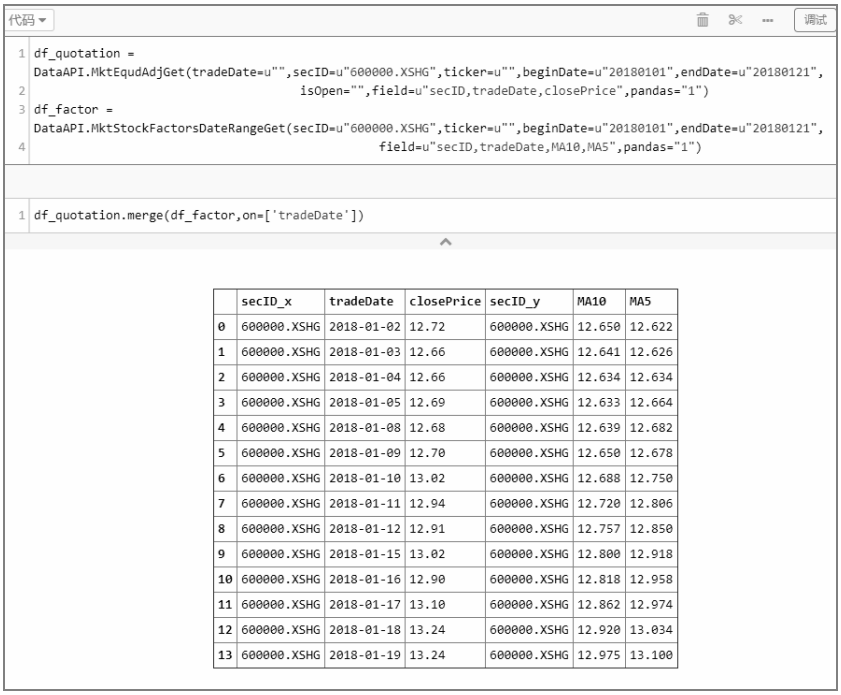
4.3.1　数据整合

数据整合指将不同数据源的数据进行汇总，形成可用于综合分析的表。

1. 合并、追加

指向表中添加其他表中的字段或记录。例如，如果要分析一只股票站上其均线的情况，则需要知道其收盘价格及均线价格。当然，均线价格可以通过收盘价计算出来，但实际上在优矿因子库中已经有了均线因子，可以直接使用。现在的问题就变成了，如何将我们通

过行情 DataAPI 与因子 DataAPI 调出来的数据合并？这个问题在 Python 中通过一两行代码即可解决，如图 4-10 所示。



```
df_quotation =
DataAPI.MktEqudAdjGet(tradeDate=u"",secID=u"600000.XSHG",ticker=u"",beginDate=u"20180101",endDate=u"20180121",
                                    isOpen="",field=u"secID,tradeDate,closePrice",pandas="1")
df_factor =
DataAPI.MktStockFactorsDateRangeGet(secID=u"600000.XSHG",ticker=u"",beginDate=u"20180101",endDate=u"20180121",
                                            field=u"secID,tradeDate,MA10,MA5",pandas="1")
```

```
df_quotation.merge(df_factor,on=['tradeDate'])
```

	secID_x	tradeDate	closePrice	secID_y	MA10	MA5
0	600000.XSHG	2018-01-02	12.72	600000.XSHG	12.650	12.622
1	600000.XSHG	2018-01-03	12.66	600000.XSHG	12.641	12.626
2	600000.XSHG	2018-01-04	12.66	600000.XSHG	12.634	12.634
3	600000.XSHG	2018-01-05	12.69	600000.XSHG	12.633	12.664
4	600000.XSHG	2018-01-08	12.68	600000.XSHG	12.639	12.682
5	600000.XSHG	2018-01-09	12.70	600000.XSHG	12.650	12.678
6	600000.XSHG	2018-01-10	13.02	600000.XSHG	12.688	12.750
7	600000.XSHG	2018-01-11	12.94	600000.XSHG	12.720	12.806
8	600000.XSHG	2018-01-12	12.91	600000.XSHG	12.757	12.850
9	600000.XSHG	2018-01-15	13.02	600000.XSHG	12.800	12.918
10	600000.XSHG	2018-01-16	12.90	600000.XSHG	12.818	12.958
11	600000.XSHG	2018-01-17	13.10	600000.XSHG	12.862	12.974
12	600000.XSHG	2018-01-18	13.24	600000.XSHG	12.920	13.034
13	600000.XSHG	2018-01-19	13.24	600000.XSHG	12.975	13.100

图 4-10

如图 4-10 所示，我们通过 merge 函数便把均线价格添加到行情表上了，再在这个表上判断当日是否站上均线就十分方便了。

2. 数据透视

指将长表转换为宽表，将作业型表转换为分析型表。假设我们有一个包含多只股票在某一时间段的总市值数据的长表，是优矿行情 DataAPI 的返回结果类型，那么如何方便地求出这些股票每一天的市值之和呢？这时可以使用 Python 的数据透视表方法，将长表转换为宽表，之后运用 DataFrame 的 sum 方法即可很简单地解决这个问题，如图 4-11 所示。

图 4-11

如上所示，在 DataAPI 的原始返回数据结构中计算每天的市值之和是比较困难的，然而在转换数据格式后，分析和计算起来就十分方便了。

4.3.2　数据过滤

我们经常要对原始数据按某个指标进行过滤，例如在获取到了所有股票的 PE 后，仅想看 PE 大于 0 的股票（对于 PE 小于 0 的股票，该指标没有意义），这是非常常见的数据整理需求。

如图 4-12 所示，我们调取了所有 A 股在 2018 年 1 月 19 日的 PE 值，然后在 DataFrame 属性框中写筛选逻辑即可完成过滤。

```
df = DataAPI.MktStockFactorsOneDayGet(tradeDate=u"20180119",secID=set_universe("A"),ticker=u"",
                                      field=u"secID,tradeDate,PE",pandas="1")
```

```
df[df.PE>0].head(10)
```

	secID	tradeDate	PE
0	000001.XSHE	2018-01-19	11.0330
1	000002.XSHE	2018-01-19	18.2449
2	000004.XSHE	2018-01-19	94.2883
4	000006.XSHE	2018-01-19	13.2546
5	000007.XSHE	2018-01-19	81.3401
6	000008.XSHE	2018-01-19	54.4002
7	000009.XSHE	2018-01-19	71.6105
8	000010.XSHE	2018-01-19	143.7984
9	000011.XSHE	2018-01-19	11.3323
10	000012.XSHE	2018-01-19	28.3515

图 4-12

4.3.3 数据探索与数据清洗

通常我们会认为 ROE 的值约为 10%～30%，但实际上在调用所有股票的 ROE 因子进行分析时，我们发现并不是这么一回事。这里以 ROE 为例，介绍常用的数据探索方法及数据清洗方式。

我们在进行数据探索时，通常会先通过做一个简单的直方图来看看数据的分布。如图 4-13 所示，真实的 ROE 分布可能与我们想象的并不一样，存在许多负值，并且大约存在-3～3 的极端值。

可以再做一下 boxplot 图来看看结果，如图 4-14 所示。

通过如图 4-14 所示的 boxplot 图也可以看出，在数据中存在很多异常值。当然这些异常值按照 ROE 的传统算法，可能并不算是错误的值。但如果要把 ROE 当作一个指标，进一步分析其对股票的未来收益或者其他方面的影响，则在建立回归或者其他模型时，就必须考虑到对异常值的处理，因为它对模型的影响可能很大。

```
df = DataAPI.MktStockFactorsOneDayGet(tradeDate=u"20180119",secID=set_universe("A"),ticker=u"",
                                      field=u"secID,tradeDate,ROE",pandas="1")
df.plot.hist(bins=100)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x100100d0>
```

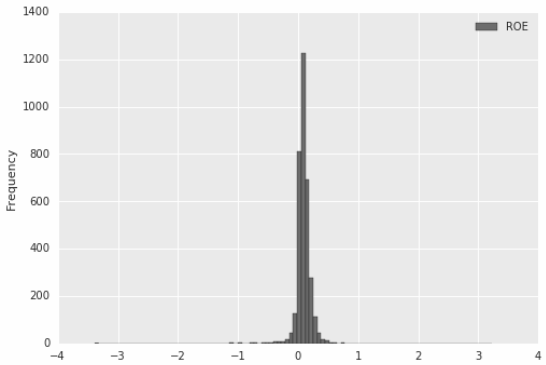


图 4-13

```
_ = df.boxplot(sym='rs')
```

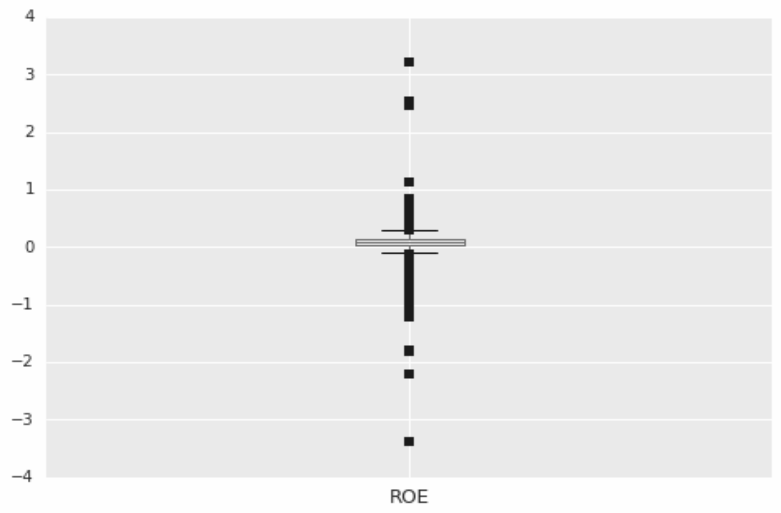


图 4-14

这里我们以最常用的 3 倍标准差法为例，将超过 3 倍标准差的数据调整为 3 倍标准差，如图 4-15 所示。

```
df_ = df.copy()
```

代码

```
df_.loc[df.ROE-df.ROE.mean()<-3*df.ROE.std(),'ROE'] = df.ROE.mean()-3*df.ROE.std()
df_.loc[df.ROE-df.ROE.mean()>3*df.ROE.std(),'ROE'] = df.ROE.mean()+3*df.ROE.std()
```

```
df_.plot.hist(bins=100)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x89cbb50>
```

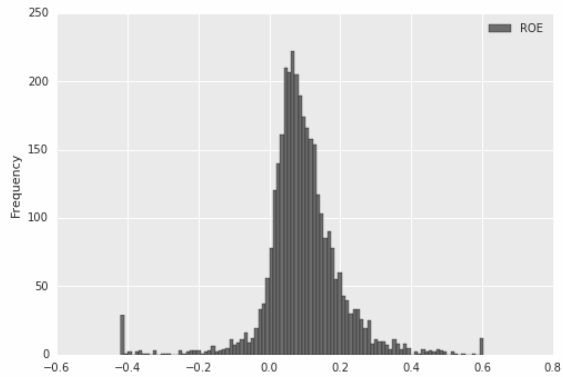


图 4-15

可以看到，经过去极值处理后的数据全部在原始数据的 3 倍标准差内，分布不再有极端值，已处理后的因子建模将更加稳定，这也是数据挖掘中常常提及的“盖帽法”。

4.3.4 数据转化

除了数据清洗，为了建立模型，我们有时还需要进行数据转化。这里介绍两个常用的数据转化方法：数据标准化及设置哑变量。

1. 数据标准化

数据标准化可以消除量纲及变量自身变异程度的影响，对很多模型来说都是十分重要的。常用的数据标准化有 Min-max 标准化和 z-score 标准化。

Min-max 标准化的计算公式为：

$$v' = \frac{v - Min}{Max - Min}$$

z-score 标准化的计算公式为：

$$z' = \frac{v - u}{\sigma}$$

Min-max 标准化可以将标准化后的值统一到 0～1；z-score 标准化可以使标准化后的数据分布均值为 0，方差为 1。我们将去极化后的数据再进行一次标准化，如图 4-16 及图 4-17 所示。

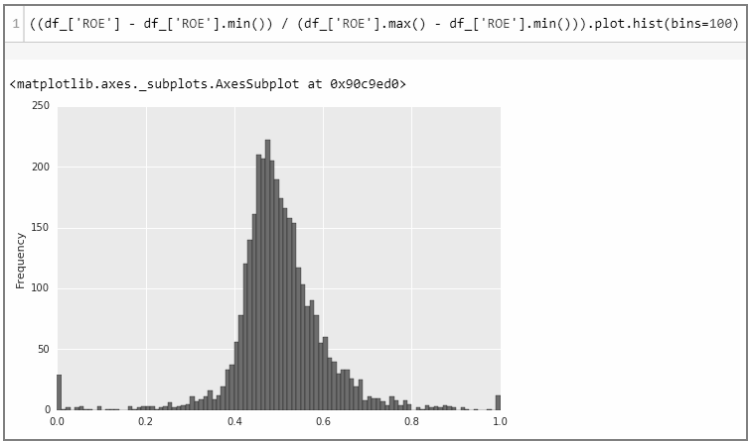


图 4-16

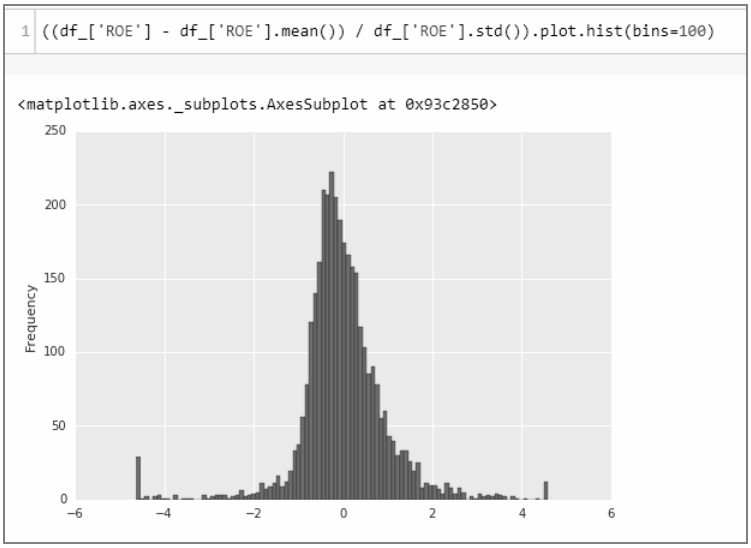


图 4-17

标准化永远不会改变数据原始的序数，它做的只是对数值大小的调整。若不看数值的话，则数据的分布并没有变化。

2. 哑变量

除了标准化，我们在进行金融建模时的另一个常用的数据转化方法就是设置哑变量。例如，在将股票的行业信息加入建模分析时，依据原始分类是无法进行处理的，必须把它转化为 0 或 1 的变量，如图 4-18 所示。

```
import pandas as pd
```

代码

```
df_industry = DataAPI.EquIndustryGet(industryVersionCD=u"010303",industry=u"",

secID=df_['secID'].tolist(),ticker=u"",intoDate=u"20180101",field=u"secID,industryName1",pandas="1")
industry_list = df_industry['industryName1'].drop_duplicates().tolist()
```

```
def get(x):
    ind_s = pd.Series([0]*len(industry_list),index = industry_list)
    if len(df_industry[df_industry['secID']==x]) > 0:
        ind = df_industry[df_industry['secID']==x]['industryName1'].values[0]
        ind_s.loc[ind] = 1
    # print ind_s
    return ind_s
```

```
df_[industry_list] = df_['secID'].apply(lambda x: get(x))
```

```
df_.head()
```

	secID	tradeDate	ROE	银行	房地产	医药生物	公用事业	综合	机械设备	建筑装饰	...	商业贸易	化工	有色金属
0	000001.XSHE	2018-01-19	0.1124	1	0	0	0	0	0	0	...	0	0	0
1	000002.XSHE	2018-01-19	0.2029	0	1	0	0	0	0	0	...	0	0	0
2	000004.XSHE	2018-01-19	0.0631	0	0	1	0	0	0	0	...	0	0	0
3	000005.XSHE	2018-01-19	-0.0267	0	0	0	1	0	0	0	...	0	0	0
4	000006.XSHE	2018-01-19	0.2078	0	1	0	0	0	0	0	...	0	0	0

图 4-18

如图 4-18 所示，我们首先调用通联数据的股票行业分类 DataAPI，获取各股票的一级行业分类名字，然后通过一系列数据处理，生成每个行业的 0 或 1 的变量，这样才能把行业作为变量加入模型中进行分析。

当然，Pandas 本身也有 get_dummies 函数，也可以瞬间对分类变量进行哑变量化，读者可自行查阅帮助文档学习。

第 5 章

通联数据回测平台介绍

在第 4 章中已经介绍了通联数据的 DataAPI，以及它在优矿“开始研究”环境下的使用方法，本章将介绍通联数据回测分析模块的使用方法及参数说明。首先，进入优矿的“开始研究”环境，新建一个 Notebook，在左上角的模式中选择“策略”，如图 5-1 所示。

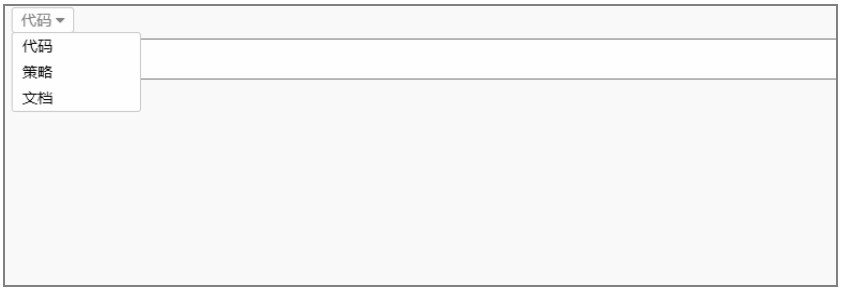


图 5-1

然后，我们就会发现生成了一个策略模板。

可以发现，策略模板包含了对一些回测参数的设置及 initialize、handle_data 这两个函数，下面逐一解释其含义，如图 5-2 所示。

```
start = '2017-01-01'                        # 回测起始时间
end = '2018-01-01'                          # 回测结束时间
universe = DynamicUniverse('HS300')         # 证券池，支持股票、基金、期货、指数四种资产
benchmark = 'HS300'                         # 策略参考标准
freq = 'd'                                  # 策略类型，'d'表示日间策略使用日线回测，'m'表示日内策略使用分钟线回测
refresh_rate = 1                            # 调仓频率，表示执行handle_data的时间间隔，若freq = 'd'时间间隔的单位为交易日，若freq = 'm'时间间隔为分钟

# 配置账户信息，支持多资产多账户
accounts = {
    'fantasy_account': AccountConfig(account_type='security', capital_base=10000000)
}

def initialize(context):
    pass

# 每个单位时间(如果按天回测,则每天调用一次,如果按分钟,则每分钟调用一次)调用一次
def handle_data(context):
    previous_date = context.previous_date.strftime('%Y-%m-%d')

    # 获取因子PE的的历史数据集，截止到前一个交易日
    hist = context.history(symbol=context.get_universe(exclude_halt=True), attribute='PE', time_range=1, style='tas')[previous_date]

    # 将因子值从小到大排序，并取前100支股票作为目标持仓
    signal = hist['PE'].order(ascending=True)
    target_position = signal[:100].index

    # 获取当前账户信息
    account = context.get_account('fantasy_account')
    current_position = account.get_positions(exclude_halt=True)

    # 卖出当前持有，但目标持仓没有的部分
    for stock in set(current_position).difference(target_position):
        account.order_to(stock, 0)

    # 根据目标持仓权重，逐一委托下单
    for stock in target_position:
        account.order(stock, 10000)
```

图 5-2

5.1 回测平台函数与参数介绍

5.1.1 设置回测参数

我们可以通过设置回测参数来设置策略回测的时间区间、股票池、基准、调仓频率，设置交易账户的账户类型、初始资金，还可以设置交易税费、滑点等。其相应的参数及设置如下。

1. start 与 end（设置回测区间）

start 指回测的起始日期，start = '2017-01-01'表明策略是从 2017 年 1 月 1 日开始回测的；end 指回测的结束日期，end = '2018-01-01'表明策略是在 2018 年 1 月 1 日结束回测的。

其类型为字符串或 datetime。

2. universe（证券池）

universe 指策略回测的证券池，即策略逻辑作用的域，下单与历史数据获取都只限于 universe 中的证券。universe 支持全部 A 股及全部可在二级市场交易的 ETF 与 LOF；还支持以三种方式获取证券池：DynamicUniverse、set_universe 和 StockScreener；也可以使用固定的股票列表，例如['000001.XSHE', '600000.XSHG']。对这个参数的设置，主要是为了让回测平台能够提前预加载相关数据，以加快回测速度。

其类型为 list。

示例如下：

```
universe = ['000001.XSHE', '600000.XSHG']                # 静态证券池
universe = set_universe('HS300', date='2016-03-01')      # 静态证券池
universe = DynamicUniverse('HS300')                      # 动态证券池
universe = StockScreener(Factor.PE.nlarge(10))           # 动态证券池
```

3. set_universe（静态证券池）

用于返回预设的证券代码列表，支持行业成分股、指数成分股。当资产池指定为一个特定的列表时，策略框架仅返回相应列表中的内容。示例代码如下：

```
set_universe(symbol, date)
```

set_universe(symbol, date)中的参数 symbol 指证券代码列表的名字，为 str 或预设的行业与指数实例类型，用法如下。

（1）当参数为 str 类型时，支持 7 个预设值，其中，SH50 表示上证 50；SH180 表示上证 180；HS300 表示沪深 300；ZZ500 表示中证 500；CYB 表示创业板；ZXB 表示中小板；A 表示全 A。对于该类型的参数，可以用 DataAPI.IdxGet()函数获取所有指数的 secID 值。示例如下：

```
set_universe('HS300')   # 表示沪深 300 的字符串
```

（2）预设的行业实例和指数实例指具体支持的行业和指数，在 Notebook 中也有自动代码提示功能，可帮助我们查找行业和指数的名称。

行业实例如下：

```
set_universe(IndSW.YinHangL2)   # 行业分类实例 IndSW，申万行业；YinHangL2，银行二级行业分类
```

指数实例如下：

```
set_universe(IdxCN.IdxShangZhengZongZhi)    # 指数成分实例
```

set_universe(symbol, date)中的参数 date 指定证券池列表的日期，默认为现实的最近一个交易日。该参数的类型为 str 或者 datetime，字符串只支持 YYYY-MM-DD 和 YYYYMMDD 这两种格式。

set_universe(symbol, date)返回 list 类型，为证券代码列表。示例如下：

```
universe = set_universe('SH50', date='2016-03-01') # 获取 2016 年 3 月 1 日的上证 50 列表
universe = set_universe('SH50')     # 获取上一个交易日的上证 50 列表
```

静态证券池也可以指定固定的个别资产或资产列表，示例如下：

```
universe = ['000001.XSHE', 'IFM0'] # 指定平安银行和股指期货
```

4. DynamicUniverse（动态证券池）

用于返回动态证券池实例。在使用板块成分股、指数成分股或行业成分股作为策略的交易对象时，策略框架会根据实际情况调整当天股票池的内容。其用法为：

```
DynamicUniverse(<板块代码或行业、指数实例>)。
```

在策略回测中，我们推荐使用 DynamicUniverse 来替代 set_universe，以避免出现幸存者偏差，即因提前使用了未来的板块成分而导致的策略效果偏好的错误。

DynamicUniverse 支持 7 个预设板块，为 str 类型，如表 5-1 所示。

表 5-1

板块代码	含　义
SH50	上证 50
SH180	上证 180
HS300	沪深 300
ZZ500	中证 500
CYB	创业板
ZXB	中小板
A	全 A 股

可以使用 DynamicUniverse('HS300')调用，表示沪深 300 的当期成分股。同时支持预设行业和指数实例，指具体支持的行业和指数，在 Notebook 中也有自动代码提示功能，可帮助我们查找行业和指数名称。示例如下：

◎ DynamicUniverse(IndSW.YinHangL2)# 行业分类实例，IndSW，申万行业；YinHangL2，银行二级行业分类

◎ DynamicUniverse(IdxCN.IdxShangZhengZongZhi) # 指数成分实例

◎ DynamicUniverse('HS300',IndSW.YinHangL2,IdxCN.IdxShangZhengZongZhi) # 混合使用

还支持动态证券池和普通列表取并集，示例如下：

◎ universe = DynamicUniverse('HS300') + ['000001.XSHE']# 包含沪深 300 成分股动态证券池和平安银行

5. apply_filter

用于将筛选条件作用于动态证券池的每个交易日的证券池上，进一步缩小策略标的范围。当前支持使用优矿因子库中的所有因子对证券池进行筛选。示例如下：

```
DynamicUniverse(<板块代码或行业、指数实例>).apply_filter(<因子筛选条件表达式>)
```

其参数为<因子筛选条件表达式>，表达式写法如下：

```
Factor.<factor_name>.<筛选方法>
```

其中，factor_name 是因子名，可以通过优矿因子库查看所有支持筛选的因子。目前提供了 5 种筛选方法，如表 5-2 所示。

表 5-2

条件约束	含　义	举　例	描　述
value_range	按值筛选	Factor.PE.value_range(70,100)	PE 值在 70 和 100 之间的股票
pct_range	按百分比筛选	Factor.PE.pct_range(0.95,1)	PE 值在 95%和 100%之间的股票，升序排列
num_range	按序号筛选	Factor.PE.num_range(1,100)	PE 值在第 1 到第 100 之间的股票，升序排列
nlarge	取最大	Factor.PE.nlarge(10)	PE 最大的 10 只股票
nsmall	取最小	Factor.PE.nsmall(10)	PE 最小的 10 只股票

运算规则如下。

- 支持单个筛选条件，也支持多个筛选条件的表达式，最多支持 5 个筛选条件。
- 在表达式中可以使用两种二元运算：一种为交（&），同时满足两个筛选条件；一种为并（|），满足任意一个筛选条件，运算方向为从左到右。
- 当筛选条件多于两个时，可以通过括号嵌套来确定运算顺序，例如（<条件 1> & <条件 2>) | (<条件 3> & <条件 4>)。

运算示例如下：

```
    (Factor.PE.nlarge(100) | Factor.PB.pct_range(0.95, 1)) &amp;amp;
Factor.RSI.value_range(70, 100)
    # 筛选出 PE 值最大的 100 只股票或者 PB 值排名在 95%到 100%的股票，以及 RSI 值大小在 70 至
100 的股票池。
```

它返回 DynamicUniverse 类型的实例，示例如下：

```
    universe = DynamicUniverse('HS300').apply_filter(Factor.PE.nsmall(100))
    # 获得沪深 300 成分股中 PE 最小的 100 只股票列表
```

6. benchmark（参考基准）

为策略参照标准，即该量化策略回测结果的比较标准，通过比较可以大致看出策略的好坏，为 str 类型。策略的一些风险指标如 alpha、beta 等也要通过 benchmark 计算。

benchmark 支持如下三种赋值方式。

（1）5 个常用指数：SHCI（上证综指）、SH50（上证 50）、SH180（上证 180）、HS300（沪深 300）和 ZZ500（中证 500），可以通过直接命名“HS300”来定义，例如：

```
    benchmark = 'HS300'  # 策略参考标准为沪深 300
```

（2）也可以通过 DataAPI.IdxGet()这个 API 拿到其他指数的 secID，例如：

```
    benchmark = '399006.ZICN'  # 策略参考标准为创业板指
```

（3）个股的 secID，例如：

```
    benchmark = '000001.XSHE'  # 策略参考标准为平安银行
```

7. freq 和 refresh_rate（策略运行频率）

策略回测在本质上是使用历史行情和其他依赖数据对策略的逻辑进行历史回放。freq 和 refresh_rate 这两个函数共同决定了回测使用的数据和调仓频率，其中，freq 表示使用的数据为日线行情数据或者分钟线行情数据，为 str 类型；refresh_rate 表示调仓间隔的时间，为每次触发 handle_data 间隔的时间，为 int 或(a, b)结构。

优矿支持日线策略和分钟线策略这两种模式。

1）日线策略

在采用日线策略时，每天会执行一次 handle_data。执行时间为开盘前，此时仅可获得当天的盘前信息，以及截至前一天的行情、因子等数据，不会获得当天的盘中行情等数据。其中：

- freq 取值为 d，d 表示在策略中使用的数据为日线级别数据，只能进行日线级别的调仓；
- refresh_rate 使用整数 1，表示在每一个交易日进行调仓的策略。

示例如下：

```
    start = '2017-01-01'  # 在 2017 年 1 月 1 日开始回测
    end = '2017-02-01'    # 在 2017 年 2 月 1 日结束回测
    universe = DynamicUniverse('HS300')         # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                         # 策略参考基准
    freq = 'd'
    refresh_rate = 1

    accounts = {
        'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000)
    }

    def initialize(context):                    # 初始化策略运行环境
        pass

    def handle_data(context):                   # 核心策略逻辑
        account = context.get_account('fantasy_account')

        print context.current_date
```

如下代码展示了部分调仓日期，在每个交易日都有调仓：

```
2017-01-03 00:00:00
2017-01-04 00:00:00
2017-01-05 00:00:00
...
2017-01-25 00:00:00
2017-01-26 00:00:00
```

在使用 Weekly(1)时，表示在每周的第 1 个交易日进行调仓：

```
start = '2017-01-01'    # 在 2017 年 1 月 1 日开始回测
end = '2017-02-01'      # 在 2017 年 2 月 1 日结束回测
universe = DynamicUniverse('HS300')     # 证券池，支持股票、基金和期货
benchmark = 'HS300'                     # 策略参考基准
freq = 'd'
refresh_rate = Weekly(1)

accounts = {
    'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000)
}

def initialize(context):                # 初始化策略运行环境
    pass

def handle_data(context):               # 核心策略逻辑
    account = context.get_account('fantasy_account')

    print context.current_date
```

如下代码展示了调仓日期，在每周的第 1 个交易日进行调仓：

```
2017-01-03 00:00:00
2017-01-09 00:00:00
2017-01-16 00:00:00
2017-01-23 00:00:00
```

在使用 Monthly(1, -1)时，表示在每个月的第 1 个和最后 1 个交易日进行调仓：

```
start = '2017-01-01'    # 在 2017 年 1 月 1 日开始回测
end = '2017-03-01'      # 在 2017 年 3 月 1 日结束回测
universe = DynamicUniverse('HS300')     # 证券池，支持股票、基金和期货
benchmark = 'HS300'                     # 策略参考基准
```

```
    freq = 'd'
    refresh_rate = Monthly(1, -1)

    accounts = {
        'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000)
    }

    def initialize(context):                # 初始化策略运行环境
        pass

    def handle_data(context):               # 核心策略逻辑
        account = context.get_account('fantasy_account')

        print context.current_date
```

如下代码展示了调仓日期，在每个月的第 1 个和最后 1 个交易日进行调仓：

```
    2017-01-03 00:00:00
    2017-01-26 00:00:00
    2017-02-03 00:00:00
    2017-02-28 00:00:00
    2017-03-01 00:00:00
```

2）分钟线策略

在采用分钟线策略时，首先会在开盘前执行一次 handle_data，然后在盘中调仓所对应时间的分钟结束后执行一次 handle_data（不包含收盘时刻），其中：

◎ freq 取值为 m，m 表示在策略中使用的数据为分钟线数据，可以进行分钟线级别的调仓；

◎ refresh_rate 使用(1, 2)，表示在每个交易日每隔两分钟进行调仓。

示例代码如下：

```
    start = '2017-01-04'    # 在 2017 年 1 月 4 日开始回测
    end = '2017-01-04'      # 在 2017 年 1 月 4 日结束回测
    universe = DynamicUniverse('HS300')     # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'm'
    refresh_rate = (1, 2)
```

```
    accounts = {
        'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000)
    }

    def initialize(context):                # 初始化策略运行环境
        pass

    def handle_data(context):               # 核心策略逻辑
        account = context.get_account('fantasy_account')

        print context.now
```

如下代码展示了部分调仓时间，在每个交易日每隔两分钟进行调仓：

```
    2017-01-04 09:30:00
    2017-01-04 09:32:00
    2017-01-04 09:34:00
    ...
    2017-01-04 14:56:00
    2017-01-04 14:58:00
```

使用(2, ['10:30', '14:30'])，表示在每两个交易日的特定时间使用分钟线数据进行调仓：

```
    start = '2017-01-04'    # 在 2017 年 1 月 4 日开始回测
    end = '2017-01-10'      # 在 2017 年 1 月 10 日结束回测
    universe = DynamicUniverse('HS300')     # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'm'
    refresh_rate = (2, ['10:30', '14:30'])

    accounts = {
        'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000)
    }

    def initialize(context):                # 初始化策略运行环境
        pass

    def handle_data(context):               # 核心策略逻辑
        account = context.get_account('fantasy_account')

        print context.now
```

如下代码展示了全部调仓时间，在每 2 个交易日的 10:30、14:30 进行调仓：

```
    2017-01-04 10:30:00
    2017-01-04 14:30:00
    2017-01-06 10:30:00
    2017-01-06 14:30:00
    2017-01-10 10:30:00
    2017-01-10 14:30:00
```

在使用(Weekly(1, -1), ['10:30', '14:30'])时，表示在每周的第 1 个和最后 1 个交易日的特定时间使用分钟线数据进行调仓：

```
    start = '2017-01-01'    # 在 2017 年 1 月 1 日开始回测
    end = '2017-01-10'      # 在 2017 年 1 月 10 日结束回测
    universe = DynamicUniverse('HS300')     # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'm'
    refresh_rate = (2, ['10:30', '14:30'])

    accounts = {
        'fantasy_account':AccountConfig(account_type='security',capital_base
=10000000)
    }

    def initialize(context):                # 初始化策略运行环境
        pass

    def handle_data(context):               # 核心策略逻辑
        account = context.get_account('fantasy_account')

        print context.now
```

如下代码展示了全部调仓时间，指定在每周第 1 个和最后 1 个交易日的 10:30、14:30 进行调仓：

```
    2017-01-06 10:30:00
    2017-01-06 14:30:00
    2017-01-09 10:30:00
    2017-01-09 14:30:00
```

在使用(Monthly(1), 120)时，表示在每月第 1 个交易日每隔 120 分钟使用分钟线数据进行调仓：

```
    start = '2017-01-01'    # 在 2017 年 1 月 1 日开始回测
    end = '2017-03-01'      # 在 2017 年 3 月 1 日结束回测
    universe = DynamicUniverse('HS300')     # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'm'
    refresh_rate = (Monthly(1), 120)

    accounts = {
        'fantasy_account':AccountConfig(account_type='security',capital_base
=10000000)
    }

    def initialize(context):                # 初始化策略运行环境
        pass

    def handle_data(context):               # 核心策略逻辑
        account = context.get_account('fantasy_account')

        print context.now
```

如下代码展示了全部调仓时间，表示在每月第 1 个交易日每隔 120 分钟使用分钟线数据进行调仓：

```
    2017-01-03 09:30:00
    2017-01-03 11:30:00
    2017-02-03 09:30:00
    2017-02-03 11:30:00
    2017-03-01 09:30:00
    2017-03-01 11:30:00
```

5.1.2 accounts 账户配置

为优矿回测框架的交易账户配置函数，优矿最新框架支持多种交易品种，可对多个交易账户同时进行回测。示例如下：

```
    accounts = {
        'security_account': AccountConfig(account_type='security', capital_
base=10000000, position_base = {}, cost_base = {}, commission = Commission
(buycost=0.001, sellcost=0.002, unit='perValue'), slippage = Slippage(value=0.0,
unit='perValue'))
    }   # 股票（包含场内基金）账户配置
```

```
    accounts = {
        'futures_account':AccountConfig(account_type='futures',capital_base=
10000000,commission=Commission(buycost=0.001,sellcost=0.002,unit='perValue'),
slippage = Slippage(value=0.0, unit='perValue'), margin_rate = 0.1)
    }   # 期货账户配置

    accounts = {
        'otcfund_account': AccountConfig(account_type='otc_fund',
capital_base=10000000, commission = Commission(buycost=0.001, sellcost=0.002,
unit='perValue'), slippage = Slippage(value=0.0, unit='perValue'),
dividend_method = 'cash_dividend')
    }   # 场外基金（不包含货币基金）账户配置
```

配置单个账户的示例如下：

```
    start = '2016-01-01'        # 回测起始时间
    end = '2017-01-01'          # 回测结束时间
    universe = DynamicUniverse('HS300')        # 证券池，支持股票、基金和期货
    benchmark = 'HS300'         # 策略参考基准
    freq = 'd'                  # 'd'表示使用日频率回测，'m'表示使用分钟频率回测
    refresh_rate = 1            # 执行 handle_data 的时间间隔

    accounts = {
        'security_account': AccountConfig(account_type='security',
capital_base=10000000, position_base = {'600000.XSHG':1000}, cost_base =
{'600000.XSHG':10.05}, commission = Commission(buycost=0.001, sellcost=0.002,
unit='perValue'), slippage = Slippage(value=0.0, unit='perValue'))
    }

    def initialize(context):    # 初始化策略运行环境
        pass

    def handle_data(context):   # 核心策略逻辑
        account = context.get_account('security_account')
```

在配置多个账户时，可以通过 commission、slippage 对多个账户进行全局配置，示例如下：

```
    start = '2016-01-01'        # 回测起始时间
    end = '2017-01-01'          # 回测结束时间
    universe = DynamicUniverse('HS300') + ['IFM0']      # 证券池，支持股票、基金
```

```
和期货
    benchmark = 'HS300'                          # 策略参考基准
    freq = 'd'                                   # 'd'表示使用日频率回测，'m'表示使用分
钟频率回测
    refresh_rate = 1                             # 执行 handle_data 的时间间隔

    # 对两个账户进行全局的交易费用和滑点设置
    commission = Commission(buycost=0.001, sellcost=0.002, unit='perValue')
    slippage = Slippage(value=0.0, unit='perValue')

    accounts = {
        'security_account1': AccountConfig(account_type='security',
capital_base=10000000, position_base = {'600000.XSHG':1000}, cost_base =
{'600000.XSHG':10.05}, commission = commission, slippage = slippage),

        'security_account2': AccountConfig(account_type='security',
capital_base=20000000, position_base = {'600000.XSHG':2000}, cost_base =
{'600000.XSHG':10.05}, commission = commission, slippage = slippage)
    }

    def initialize(context):                     # 初始化策略运行环境
        pass

    def handle_data(context):                    # 核心策略逻辑
        account1 = context.get_account('security_account1')
        account2 = context.get_account('security_account2')
```

配置股票期货混合账户的示例如下：

```
    start = '2016-01-01'                         # 回测起始时间
    end = '2017-01-01'                           # 回测结束时间
    universe = DynamicUniverse('HS300') + ['IFM0']       # 证券池，支持股票、基金
和期货
    benchmark = 'HS300'                          # 策略参考基准
    freq = 'd'                                   # 'd'表示使用日频率回测，'m'表示使用分
钟频率回测
    refresh_rate = 1                             # 执行 handle_data 的时间间隔

    accounts = {
        'security_account': AccountConfig(account_type='security',
capital_base=10000000, position_base = {'600000.XSHG':1000}, cost_base =
{'600000.XSHG':10.05}, commission = Commission(buycost=0.001, sellcost=0.002,
```

```
unit='perValue'), slippage = Slippage(value=0.0, unit='perValue')),

        'futures_account': AccountConfig(account_type='futures',
capital_base=10000000, commission = Commission(buycost=0.001, sellcost=0.002,
unit='perValue'), slippage = Slippage(value=0.0, unit='perValue'), margin_rate
= 0.1)
    }

    def initialize(context):                # 初始化策略运行环境
        pass

    def handle_data(context):               # 核心策略逻辑
        account1 = context.get_account('security_account')
        account2 = context.get_account('futures_account')
```

1. AccountConfig（账户配置）

用于配置单个交易账户，在策略初始化时会根据账户的配置创建对应的交易账户。其参数如下。

- 必选参数：account_type capital_base。
- 可选参数：position_base cost_base commission slippage。
- 期货专用参数：margin_rate。
- 场外基金专用参数：dividend_method。

2. account_type（账户类型设置）

用于设置交易账户类型，为 str 类型，支持 security 股票和场内基金、futures 期货、otc_fund 场外基金（不含货币基金）、index 指数。

```
    account_type = 'security'
```

3. capital_base（初始资金设置）

用于设置交易账户初始资金，为 float 或 int 类型。示例如下：

```
    capital_base = 100000
```

4. position_bas（初始持仓设置）

用于设置交易账户初始持仓（仅适用配置股票账号），为字典类型，键为股票代码，值为数量。示例如下：

```
    position_base = {'000001.XSHE':1000, '600000.XSHG':2000}      # 初始持仓为 1000 股平安银行，2000 股浦发银行
```

5. cost_base（初始成本设置）

用于设置交易账户初始成本（仅适用配置股票账号），为字典类型，键为股票代码，值为成本。示例如下：

```
    cost_base = {'000001.XSHE':11.01, '600000.XSHG':5.00}     # 初始持仓成本平安银行为 11.01 元，浦发银行为 5.00 元
```

6. commission（手续费设置）

用于设置交易手续费。对其参数说明如下。

◎ 为 buycost 时，指买入成本，为 float 类型。

◎ 为 sellcost 时，指卖出成本，为 float 类型。

◎ 为 unit 时，指手续费单位，为 str 类型。

支持如下两个值。

◎ perValue：按成交金额的百分比收取手续费。

◎ perShare：按成交的股数收取手续费（仅用于期货账号配置）。

示例如下：

```
    commission = Commission(buycost = 0.0003, sellcost = 0.002, unit = 'perValue')
    # 按成交金额的百分比收取买入 0.3‰，卖出 2‰的费用
```

7. slippage（滑点设置）

用于设置交易滑点标准，可处理市场冲击问题。对其参数说明如下。

◎ 为 value 时，指策略进行交易时的滑点值，为 float 类型。

◎ 为 unit 时，指计算滑点的方式，分为固定滑点和百分比滑点这两种计算方式，为 str 类型。

其用法如下。

◎ perValue：百分比滑点，按股价百分比进行滑点调整；在设置滑点后，买入价格调整为股价×(1+滑点值)，卖出价格调整为股价×(1-滑点值)。

◎ perShare：固定滑点，按每股股价进行滑点调整，最小单位是 0.01，表示 0.01 元；在设置滑点后，买入价格调整为股价+滑点值，卖出价格调整为股价-滑点值。

示例如下：

```
    slippage = Slippage(value=0.001, unit='perValue')  # 滑点设置成百分比滑点
0.001
```

8. margin_rate（保证金比率设置）

用于设置保证金比例，仅用于期货策略，为 float 或者 dict 类型。在创建交易账户时进行保证金比例的设置。

◎ 全局设置方法：所有品种都使用同一个保证金率（通常在只回测一个品种时，这种写法比较常见）。

◎ 根据品种设置：为某个或某些品种设置特定的保证金率。

示例如下：

```
    margin_rate = 0.1

    margin_rate = {'IF': 0.16, 'RB': 0.1}
```

9. dividend_method（基金分红方式设置）

用于设置基金分红方式，仅用于场外基金策略，为 str 类型，可使用 cash_dividend 现金分红及 reinvestment 红利再投这两种方式。示例如下：

```
    dividend_method='cash_dividend'
```

5.1.3 initialize（策略初始化环境）

initialize()为策略初始化函数，用于配置策略运行环境 context 对象的属性或自定义各种变量，在策略运行周期中只执行一次。我们可以通过 context 添加新的属性或自定义各种变量。示例如下：

```
def initialize(context)
```

context 在策略运行（回测或模拟交易）启动时被创建，持续出现在整个策略的生命周期中。策略在运行时可以读取 context 的已有系统属性或者自定义属性。

5.1.4 handle_data（策略运行逻辑）

handle_data()为策略算法函数，在策略运行时（回测或模拟交易）会根据初始化配置的策略算法运行频率调用该函数。策略算法可以通过 context 获取运行时的行情数据、K 线图、因子数据、订单簿等数据，并根据分析的结果通过交易账户进行订单委托。

策略框架会根据实时的市场情况进行订单撮合执行，在每日结束时还会完成清算的操作。

示例如下：

```
def handle_data(context)
```

关于 handle_data 的设计与编写，请参见 5.2 节。

5.1.5 context（策略运行环境）

context 表示策略运行环境，包含运行时间、行情数据等内容，还可用于在存储策略中生成的临时数据的存储。

策略框架会在启动时创建 context 的对象示例，并以参数形式传递给 initialize(context) 和 handle_data(context)，用于策略调度。

在回测时，context 包含运行时间、回测参数、回测运行时等数据；在模拟交易时，context 包含运行时间、模拟交易参数、实时运行数据等数据。

1. now（策略运行时的当前时刻）

用于获取策略运行的当前时刻，返回 datetime，只能在 handle_data 方法中使用，并且不允许修改。示例如下：

```
    context.now
```

通过如下代码可获取分钟线策略运行时的当前时刻：

```
    start = '2017-01-04'    # 在 2017 年 1 月 4 日开始回测
    end = '2017-01-04'      # 在 2017 年 1 月 4 日结束回测
    freq = 'm'
    refresh_rate = (1, 2)

    accounts = {
        'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000)
    }

    def initialize(context):                # 初始化策略运行环境
        pass

    def handle_data(context):               # 核心策略逻辑
        account = context.get_account('fantasy_account')

        print context.now
```

如下展示了部分调仓时间，在每个交易日每 2 分钟都进行调仓：

```
    2017-01-04 09:30:00
    2017-01-04 09:32:00
    2017-01-04 09:34:00
    ...
    2017-01-04 14:56:00
    2017-01-04 14:58:00
```

2. current_date（当前回测日期）

用于获取策略运行的当前日期。返回 datetime，只能在 handle_data 方法中使用，并且不允许修改。

context.current_date 和 context.now 的不同之处为：两者对日线频率的回测结果完全一

致，在分钟频率上回测，但 context.now 包含小时、分钟等日内时间信息，context.current_date 不包含这些信息。

示例如下：

```
context.current_date
```

如下代码用于获取策略运行时的当前日期：

```
start = '2017-01-01'    # 在 2017 年 1 月 1 日开始回测
end = '2017-02-01'      # 在 2017 年 2 月 1 日结束回测
freq = 'd'
refresh_rate = 1

accounts = {
    'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000)
}

def initialize(context):                # 初始化策略运行环境
    pass

def handle_data(context):               # 核心策略逻辑
    account = context.get_account('fantasy_account')

    print context.current_date
```

如下展示了部分调仓日期：

```
2017-01-03 00:00:00
2017-01-04 00:00:00
2017-01-05 00:00:00
...
2017-01-25 00:00:00
2017-01-26 00:00:00
```

3. previous_date（回测日期的前一交易日）

用于获取当前回测日期的前一交易日，返回 datetime，只能在 handle_data 方法中使用，并且不允许修改。用法同 current_date 一致，示例如下：

```
context.previous_date
```

4. current_minute（当前运行的分钟值）

用于获取当前运行时的时间分钟值，在分钟线策略中使用，建议使用 context.now 替换。返回时间值，例如 10:00，只能在 handle_data 方法中使用，并且不允许修改。

示例如下：

```
context.current_minute
```

输出结果如下：

```
09:30
```

5. current_price（获取成交价）

用于获取当前参考价格，即最后成交价。在开盘前运行时，获得的是昨天的收盘价；在盘中运行时，获得的是最后一次成交价。参数为 symbol，指想要获取的证券的价格，需要在之前定义的 universe 中存在，为 str 类型。返回浮点数，指最后一刻的价格。

示例如下：

```
context.current_price(symbol)
```

以下代码用于获取平安银行昨天的收盘价：

```
start = '2017-01-01'                    # 回测起始时间
end = '2017-01-04'                      # 回测结束时间
universe = ['000001.XSHE']              # 证券池，支持股票、基金和期货
benchmark = 'HS300'                     # 策略参考基准
freq = 'd'                              # 'd'表示使用日频率回测，'m'表示使用分
钟频率回测
refresh_rate = 1                        # 执行 handle_data 的时间间隔

accounts = {
    'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000)
}

def initialize(context):                # 初始化策略运行环境
    pass

def handle_data(context):               # 核心策略逻辑
    account = context.get_account('fantasy_account')
```

```
        print context.current_price('000001.XSHE')
```

输出结果如下：

```
    8.967
    9.026
```

6. get_account（获取交易账户）

用于获取账户名称为 account_name 的交易账户。参数为 account_name，指在策略初始化时设置的账户名称，为 str 类型，返回交易账户对象。示例如下：

```
    context.get_account(account_name)
```

以下代码用于获取交易账户：

```
    start = '2017-01-01'                    # 回测起始时间
    end = '2017-01-04'                      # 回测结束时间
    universe = ['000001.XSHE']              # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'd'                              # 'd'表示使用日频率回测，'m'表示使用分
钟频率回测
    refresh_rate = 1                        # 执行 handle_data 的时间间隔

    accounts = {
        'account_name': AccountConfig(account_type='security',
capital_base=10000000)
    }

    def initialize(context):                # 初始化策略运行环境
        pass

    def handle_data(context):               # 核心策略逻辑
        account = context.get_account('account_name')
```

7. get_universe（获取当前交易日证券池）

用于获取当前交易日的证券池，是策略初始化参数中 universe 的子集。支持多种资产类型。只要资产在策略运行当天处于上市状态，就可通过 context.get_universe()获得。

注意：在 context.get_universe()返回的列表中，可能有部分资产处于停盘等不可交易状态。

示例如下：

```
context.get_universe(asset_type, exclude_halt=False)
```

对其参数说明如下。

（1）为 asset_type 时，表示资产类型。

（2）为 str 时，支持如下几种类型。

◎ stock：股票列表。

◎ index：指数成分股列表。

◎ exchange_fund：场内基金列表。

◎ otc_fund：场外基金列表。

◎ futures：期货合约列表。

◎ base_futures：普通期货合约列表。

◎ continuous_futures：连续期货合约列表。

（3）为 exclude_halt 时，指去除资产池中的停牌股票，仅适用于股票，为布尔型。

返回符合筛选条件当天上市状态的证券池。

如下代码用于获取上证 50 股票剔除停牌股票后的股票池：

```
    start = '2017-01-01'                    # 回测起始时间
    end = '2017-01-03'                      # 回测结束时间
    universe = DynamicUniverse('SH50')      # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'd'                       # 'd'表示使用日频率回测，'m'表示使用分钟频率回测
    refresh_rate = 1                        # 执行 handle_data 的时间间隔

    accounts = {
        'stock_account': AccountConfig(account_type='security',
capital_base=10000000)
    }

    def initialize(context):                # 初始化策略运行环境
        pass
```

```
    def handle_data(context):                # 核心策略逻辑
        account = context.get_account('stock_account')
        print context.get_universe('stock', exclude_halt=True)
```

部分输出结果为：

```
    ['600000.XSHG', '600016.XSHG', '600028.XSHG', '600029.XSHG',
'600030.XSHG'...]
```

如下代码用于获取沪铜 1610、沪深 300 期货当月对应的期货合约列表、普通期货合约列表和连续期货合约列表：

```
    start = '2016-01-01'                     # 回测起始时间
    end = '2017-01-01'                       # 回测结束时间
    universe = ['CU1610', 'IFL0']            # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                      # 策略参考基准
    freq = 'd'                               # 'd'表示使用日频率回测，'m'表示使用分
钟频率回测
    refresh_rate = 1                         # 执行 handle_data 的时间间隔

    accounts = {
        'futures_account': AccountConfig(account_type='futures',
capital_base=10000000)
    }

    def initialize(context):                 # 初始化策略运行环境
        pass

    def handle_data(context):                # 核心策略逻辑
        account = context.get_account('futures_account')
        print context.get_universe('futures')
        print context.get_universe('base_futures')
        print context.get_universe('continuous_futures')
```

输出结果为：

```
    ['CU1610', 'IFL0']
    ['CU1610']
    ['IFL0']
    ...
```

8. transfer_cash（账户间资金划转）

用于账户间的资金划转。对其参数说明如下。

◎　为 origin 时，指资金流出的账户名称，为 str 类型。

◎　为 target 时，指资金流入的账户名称，为 str 类型。

◎　为 amount 时，指划转的资金量，为 float 类型。

示例如下：

```
    context.transfer_cash(origin, target, amount)
```

如下代码用于获取账户间的资金划转：

```
    start = '2017-01-01'                        # 回测起始时间
    end = '2017-01-10'                          # 回测结束时间
    universe = ['000001.XSHE', 'IFM0']          # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                         # 策略参考基准
    freq = 'd'                                  # 'd'表示使用日频率回测，'m'表示使用分
钟频率回测
    refresh_rate = 1                            # 执行 handle_data 的时间间隔

    accounts = {
        'stock_account': AccountConfig('security', capital_base=1e6),
        'futures_account': AccountConfig('futures', capital_base=1e6)
    }

    def initialize(context):                    # 初始化策略运行环境
        pass

    def handle_data(context):
        stock_account = context.get_account('stock_account')
        futures_account = context.get_account('futures_account')

        if context.current_date.strftime('%Y-%m-%d') == '2017-01-05':
            context.transfer_cash(origin=stock_account, target=futures_account,
amount=1e5)
            assert stock_account.cash, futures_account.cash == (900000, 1100000)
            assert stock_account.portfolio_value +
futures_account.portfolio_value == 2000000
        if context.current_date.strftime('%Y-%m-%d') == '2017-01-06':
```

```
            context.transfer_cash(origin=futures_account, target=stock_account,
amount=2e5)
            assert stock_account.cash, futures_account.cash == (1100000, 900000)
            assert  stock_account.portfolio_value +
futures_account.portfolio_value == 2000000
```

5.2 股票模板实例

上面介绍了编写策略回测的基本模板，下面以一个完整的股票策略为例进行讲解。策略截图如图 5-3、图 5-4 所示。

```
import pandas as pd
start = '2014-11-01'
end   = '2018-01-01'
benchmark = 'HS300'
universe = DynamicUniverse('HS300')   # 股票池为沪深300
refresh_rate = 60
max_history_window = 60
accounts = {
    'fantasy_account': AccountConfig(account_type='security', capital_base=10000000,
                                     commission=Commission(buycost=0.001, sellcost=0.002,
unit='perValue'),slippage = Slippage(value=0.0, unit='perValue'))
}
def initialize(context):
    pass
def handle_data(context):

    account = context.get_account('fantasy_account')
    universe = context.get_universe(exclude_halt=True)
    history = context.history(universe,'closePrice', 60)
    momentum = {'symbol':[], 'c_ret':[]}
    for stk in history.keys():
        momentum['symbol'].append(stk)
        momentum['c_ret'].append(history[stk]['closePrice'][-1]/history[stk]['closePrice'][0])
    # 按照过去60日收益率排序，并且选择前60只的股票作为买入候选
    momentum = pd.DataFrame(momentum).sort(columns='c_ret',ascending=False).reset_index()
    # print momentum
    momentum = momentum[:60]   # 选择
    buylist = momentum['symbol'].tolist()

    for stk in account.get_positions():
        if stk not in buylist:
            order_to(stk, 0)

    # 等权重买入所选股票
    portfolio_value = account.portfolio_value
    for stk in buylist:
            order_pct_to(stk, 1.0/len(buylist))
```

年化收益率	基准年化收益率	阿尔法	贝塔	夏普比率	收益波动率	信息比率	回测详情	开始交易
18.7%	16.5%	1.5%	1.05	0.50	30.5%	0.26		

图 5-3

图 5-4

图 5-3 和图 5-4 实现的是一个动量策略，其策略思想非常简单，为选取过去 60 个交易日累积收益率最高的 60 只股票买入，换仓频率为 60 天。可以看出，该策略最后的表现结果一般，比指数略好，不过本章的重点不在于评价策略，而是深入学习如何编写策略。

首先，我们看到策略的第 1 句引入了 Pandas 模块“import pandas as pd”，这是由于我们在编写策略时在后面用到了这个模块。我们应当引入所有可能需要的外部模块，通常将其写在策略代码的开头。

对接下来的几个设置我们已经有所了解，回测区间是 2014 年 11 月 1 日至 2018 年 1 月 1 日，股票池是动态沪深 300 成分股，比较基准是沪深 300 指数，调仓频率是每 60 个交易日调仓一次，此处没给出频率参数，默认是“d”（天）。

注意，这里出现了之前没有讲到过的 max_history_window，之所以需要设置它，是因为我们在 handle_data 中调用了 context 的 history 方法。max_history_window 默认对日线数据支持 30 个交易日，对分钟线支持 240 条 K 线数据，回溯长度在超出范围时需要手动指定。当在 history 中调用的历史数据窗口大于 max_history_window 的默认值时，会报出类似如下所示的错误：

```
    ValueError: Exception in "Context.history":history overflow.Your current max
daily history window is 30. Please use a shorter parameter, or change
max_history_window in your initial
```

所以，如果我们想用 context 自带的 history 方法，就需要确保此时的 max_history_window 大于调用历史数据窗口。

接着，我们又看到了对 accounts 账户的设置，account_type 及 capital_base 和我们之前看到的并没有区别。因为我们采用的是股票策略，所以 account_type='security'，capital_base（初始资金）还是 1000 万元。但是这里还需要设置另外两个参数：一个是 commission，另一个是 slippage。commission 参数用于设置手续费，而 slippage 参数用于设置滑点，它们接收的是特定的 Commission 和 Slippage 实例。

通过对上面讲解的手续费及滑点设置的内容，我们可以知道在买入时收取的是买入金额千分之一的手续费，卖出时收取的是卖出金额千分之二的手续费；而对于滑点，我们将其直接设置为 0。可以根据实际需要修改这些设置。

然后，我们可以看到策略最重要的逻辑都是在 handle_data 函数里实现的。首先通过 account = context.get_account('fantasy_account')获取我们在前面设定好的股票账户。context 自带了一些日期属性及价量信息、股票池处理等功能，此处我们看到了 context 的 get_universe 方法，并且它的参数 exclude_halt 被设置成了 True，意思是获取我们当前的股票池并且剔除停牌的股票。因为我们只能交易非停牌的股票，所以要把停牌的股票去掉。

我们还看到了 context 的另一个方法 history，在讲解 max_history_window 参数时也提到过它：

```
    context.history(symbol,attribute,time_range, freq='1d', style='sat',
rtype='frame')
```

history 方法用于获取 K 线图等时间序列数据。对其参数说明如下。

（1）symbol：指需要获取的证券列表，支持单个证券或证券列表。

（2）attribute：指需要获取的属性，支持单个值或属性列表。可选的范围如下。

◎ openPrice：前复权开盘价。

◎ highPrice：前复权最高价。

◎ lowPrice：前复权最低价。

◎ closePrice：前复权收盘价。

◎ preClosePrice：前复权前收盘价。

◎ turnoverVol：前复权成交量。

◎ turnoverValue：前复权成交额。

（3）time_range：指所需回溯的历史 K 线图条数，和 freq 属性相对应。

（4）freq：指 K 线图周期，支持'1d'、'1m'、'5m'、'15m'、'30m'、'60m'等周期。'1d'表示日线，'1m'表示一分钟线。分钟 K 线图仅可在分钟频率回测时使用。

（5）style：指数据返回的类型，可以选择'ast'、'sat'或者'tas'这三种类型，其中'a'表示'attribute'，'s'表示 symbol，'t'表示时间，三种选择分别对应这三个维度呈现的顺序，例如'ast'表示在返回的字典中的键是 attribute，其值为列为 symbol、行为 time 的 DataFrame，依此类推。

（6）rtype：指返回值的数据类型。可以选择'frame'、'array'这两种类型。

由上述可知，通过 context.history(universe,'closePrice', 60)，我们获取了各只股票在历史上 60 个交易日的前复权收盘价，格式是一个字典，键为股票代码，值是一个索引为时间、列为前复权收盘价的 DataFrame。

接下来我们把一个字典初始化，然后对每只股票进行循环，用最近的前复权价格除以 60 个交易日之前的前复权价格，得到累积净值。然后对股票按累计净值进行排序，选取前 60 只股票作为待买股票池。

接下来，我们就要进行股票的下单买卖了。这里出现了 account 的方法 get_positions()，用于获取策略当前的所有持仓，具体返回的是一个字典，键为股票代码，值为持仓股票数量。我们对我们持仓的股票进行遍历，若它不在我们的待买股票池里，就做一个 order_to(stk,0)操作，意思是将该股票清仓。

除了 order_to 方法，还有其他下单方法，下面一一进行解释。

1. order(symbol, amount, price=0., otype='market')

指根据指定的参数，进行订单委托。订单类型支持市价单或限价单，在限价单中需将 otype 设置为"limit"，并设置下单价格。

对其参数说明如下。

◎ 为 symbol 时，指需要交易的证券代码，必须包含后缀，其中上证证券为.XSHG，深证证券为.XSHE。

◎ 为 amount 时，指需要交易的证券代码为 symbol 的证券数量，为正则买入，为负则卖出；程序会自动对 amount 向下取整到最近的整百。

◎ 为 price 时，指下限价单时的下单价格（仅日内策略可用）。

◎ 为 otype 时，指可选'market'（市价单）和'limit'（限价单）这两个值，表示交易指令类型（为 limit 时仅日内策略可用）。

2. order_to (symbol, amount, price=0., otype='market')

指通过下单，将某只股票的持仓调整到持有多少手。策略框架会自动计算当前持仓和目标持仓的差额，并进行下单。在每次 handle_data 时调用，最多只允许一次 order_to 函数调用，否则可能造成下单量计算错误。

对其参数说明如下。

◎ 为 symbol 时，指需要交易的证券代码，必须包含后缀，其中上证证券为.XSHG，深证证券为.XSHE。

◎ 为 amount 时，指需要交易的证券代码为 symbol 的证券数量，为正则买入，为负则卖出。程序会自动对 amount 向下取整到最近的整百。

◎ 为 price 时，指下限价单时的下单价格（仅日内策略可用）。

◎ 为 otype 时，指可选'market'（市价单）和'limit'（限价单）这两个值，表示交易指令类型（为 limit 时仅日内策略可用）。

3. order_pct (symbol, pct)

指根据当前的账户权益按一定比例下单。比如当前账户有 100 000 元，下单 20%，就会使用 20000 元计算最大可委托手数并下单。

对其参数说明如下。

◎ 为 symbol 时，指需要交易的证券代码，必须包含后缀，其中上证证券为.XSHG，深证证券为.XSHE。

◎ 为 pct 时，指需要交易的证券代码为 symbol 的证券占虚拟账户当前总价值的百分比，范围为 0～1，为正则买入，为负则卖出。程序会自动对计算出的下单数量向下取整到最近的整百。

4. order_pct_to(symbol, pct)

指根据当前的账户权益按一定比例下单。策略框架会自动计算当前持仓和目标持仓的差额，并进行下单。比如当前账户有 100 000 元，下单 20%，就会使用 20000 元计算最大可委托手数，为 500 股，当前持有 300 股，则下单 200 股。

对其参数说明如下。

- 为 symbol 时，指需要交易的证券代码，必须包含后缀，其中上证证券为.XSHG，深证证券为.XSHE。
- 为 pct 时，指需要交易的证券代码为 symbol 的证券占虚拟账户当前总价值的百分比，范围为 0～1，为正则买入，为负则卖出。程序会自动对计算出的下单数量向下取整到最近的整百。

看了上面的解释，相信你已经了解各种下单方法的含义。我们首先把在持仓里但不在待买股票池里的股票卖出，接着也是通过 account 的方法 account. portfolio_value 获取组合的市值，然后运用 order_pct_to 方法将所有待买股票买入或卖出到等权。需要注意的是，通常我们都会将卖出操作写在买入操作的前面，因为只有卖出股票腾出现金，我们才可以买入其他股票。

5.3　期货模板实例

在介绍完如何编写股票策略后，我们再来看看如何在优矿策略模块编写期货策略。

首先，了解一下期货专用的 API。

1. get_symbol（获得主力连续合约的映射合约）

我们在进行信号生成时，可以使用主力合约时间序列，但在下单时要使用具体的合约。若想得到当天的主力合约映射的具体合约符号，则可以使用 context.get_symbol 方法获取：

```
context.get_symbol(symbol)
```

context.get_symbol 方法用于获取某个主力连续合约的映射合约，返回主力连续合约的映射合约，仅适用于期货品种。其参数为 smybol，指主力合约的符号，为 str 类型。

其示例如下：

```
    start = '2017-01-01'                    # 回测起始时间
    end = '2017-01-03'                      # 回测结束时间
    universe = ['IFM0']                     # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'd'                              # 'd'表示使用日频率回测，'m'表示使用分
钟频率回测
    refresh_rate = 1                        # 执行 handle_data 的时间间隔

    accounts = {
        'fantasy_account': AccountConfig(account_type='futures',
capital_base=10000000)
    }

    def initialize(context):                # 初始化策略运行环境
        pass

    def handle_data(context):               # 核心策略逻辑
        account = context.get_account('fantasy_account')
        print context.get_symbol('IFM0')
```

输出结果如下：

```
    IF1701
```

2. mapping_changed（判断是否切换主力连续合约的映射合约）

用于判断是否切换某个主力连续合约的映射合约，返回布尔值，仅适用于期货品种。其参数为 smybol，指主力合约符号，为 str 类型。

示例如下：

```
    bool context.mapping_changed(symbol)
```

3. get_rolling_tuple（获取主力连续合约映射关系变化前后的具体合约）

用于获取主力连续合约映射关系变化前后的具体合约，仅适用于期货品种。其参数为 smybol，指主力合约符号，为 str 类型。

其示例如下：

```
    context.get_rolling_tuple(symbol)
```

其返回如下，指主力切换前后的对应具体合约，为字符串对类型：

```
    (symbol_before, symbol_after)
```

以下代码用于获取主力连续合约映射关系变化前后的具体合约：

```
    start = '2016-01-15'                    # 回测起始时间
    end = '2016-06-01'                      # 回测结束时间
    universe = ['IFM0', "SRM0", "RBM0", "TFM0"]  # 证券池，支持股票、基金和期货
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'd'                              # 'd'表示使用日频率回测，'m'表示使用分
钟频率回测
    refresh_rate = 1                        # 执行 handle_data 的时间间隔

    accounts = {
        'futures_account': AccountConfig(account_type='futures',
capital_base=10000000)
    }

    def initialize(context):    # 初始化策略运行环境。
        pass

    def handle_data(context):   # 回测调仓逻辑，每个调仓周期运行一次，可在此函数内实现
信号生产，生成调仓指令
        futures_account = context.get_account("futures_account")

        for symbol in universe:
            symbol_before, symbol_after = context.get_rolling_tuple(symbol)
            if symbol_before != symbol_after:
                print "{0}: {1}->{2}".format(context.current_date,
symbol_before, symbol_after)
```

4. switch_position（移仓操作）

指移仓操作，下达 symbol_before 至 symbol_after 相同数量的平仓及开仓指令（含多空持仓），仅适用于期货品种。其参数为 symbol_before 时指移仓前合约，为 str 类型；其参数为 symbol_after 时指移仓后合约，为 str 类型。

示例如下：

```
    futures_account.switch_position(symbol_before, symbol_after)
```

以下代码用于获取移仓操作：

```
    start = '2016-12-15'                    # 回测开始时间
    end = '2016-12-19'                      # 回测结束时间
    universe = ['IFM0']                     # 策略期货合约
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'd'                              # 调仓频率
    refresh_rate = 1                        # 调仓周期

    accounts = {
        'futures_account': AccountConfig(account_type='futures',
capital_base=1e9)
    }

    def initialize(context):    # 初始化虚拟期货账户。
        pass

    def handle_data(context):   # 回测调仓逻辑，每个调仓周期运行一次
        print context.current_date
        futures_account = context.get_account('futures_account')
        current_date = context.current_date.strftime('%Y-%m-%d')

        if current_date == '2016-12-15':
            symbol_before, symbol_after = context.get_rolling_tuple('IFM0')
            print symbol_before, symbol_after
            futures_account.order(symbol_after, 3, 'open')
            print futures_account.position
        elif current_date == '2016-12-16':
            symbol_before, symbol_after = context.get_rolling_tuple('IFM0')
            print symbol_before, symbol_after
            futures_account.switch_position(symbol_before, symbol_after)
            print futures_account.position
        elif current_date == '2016-12-19':
            symbol_before, symbol_after = context.get_rolling_tuple('IFM0')
            print symbol_before, symbol_after
            print futures_account.position
```

同介绍股票一样，这里还是以一个简单的例子来讲解期货策略编写的细节。策略的整体结构和之前的股票策略类似，这里会着重解释期货策略与众不同的地方，需要注意的细节如图 5-5、图 5-6 所示。

```
universe = ['IFM0']  # 策略期货合约
start = '2015-01-01'                        # 回测开始时间
end  = '2018-01-11'                         # 回测结束时间
capital_base = 600000                       # 初试可用资金
refresh_rate = 1                            # 调仓周期
freq = 'm'                                  # 调仓频率: m-> 分钟; d-> 日

commission = Commission(buycost=0.0, sellcost=0.0, unit='perValue') #设置股指期货交易手续费
slippage = Slippage(value=0, unit='perValue') #设置价格滑点百分比

accounts = {
    'fantasy_account': AccountConfig(account_type='futures', capital_base=capital_base, commission=commission, slippage=slippage)
} #设置虚拟期货账户

def initialize(context):           # 初始化虚拟期货账户，一般用于设置计数器，回测辅助变量等。
    pass

def handle_data(context):          # 回测调仓逻辑，每个调仓周期运行一次，可在此函数内实现信号生产，生成调仓指令。
    futures_account = context.get_account('fantasy_account')
    symbol = context.get_symbol(universe[0])
    if context.current_minute == '09:30':
        futures_account.order(symbol, 1, 'open')
    # 平仓
    elif context.current_minute == '09:36':
        long_position = futures_account.get_positions().get(symbol, dict()).long_amount
        if long_position > 0:
            futures_account.order(symbol, -long_position, 'close')
    # 止损
    elif context.current_minute < '09:36':
        if len(futures_account.get_positions()) > 0 and futures_account.get_positions().get(symbol).today_profit<-5000:
            long_position = futures_account.get_positions().get(symbol, dict()).long_amount
            if long_position > 0:
                futures_account.order(symbol, -long_position, 'close')
```

年化收益率	基准年化收益率	阿尔法	贝塔	夏普比率	收益波动率	信息比率	最大回撤	换手率		
14.8%	6.1%	11.1%	0.07	1.41	8.0%	0.18	5.8%	0.00	回测详情	开始交易

图 5-5

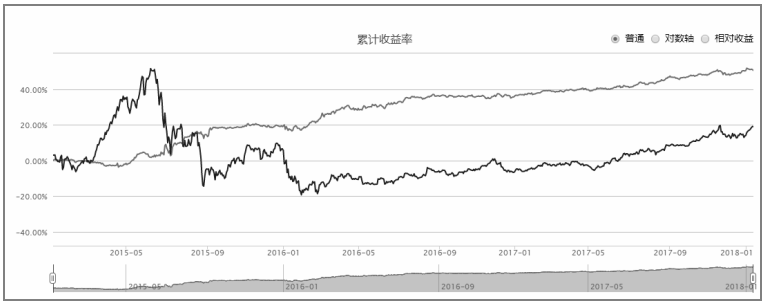


图 5-6

以上所示是一个很简单的期货策略，主要用于学习编写期货策略。其策略思想是在每天 9 点半开盘时买入 1 手沪深 300 期货，在 9 点 36 分时卖出、清仓，在此期间做一个亏损 5000 元的止损。

我们来看看以上期货策略是如何具体编写的。首先将 universe 设置为['IFM0']，指沪深 300 期货主力。因为一只期货的存续时间较短，所以在进行长期回测时会涉及期货品种的切换，而在优矿的期货回测内部已经帮大家做好了这个。将 universe 设置成期货主力后，

在后面的 handle_data 中会根据日期自动切换品种。

接下来的参数和股票策略的参数基本一致，我们的回测时间是 2015 年 1 月 1 日至 2018 年 1 月 11 日，初始资金是 60 万元，调仓频率是每分钟调仓一次。需要注意的是，要将‘account_type’调整为‘futures’，因为我们的账户是期货账户。这里为了简单展示，把交易费用与滑点都设置为了 0。

和股票策略一样，我们通过 context.get_account('fantasy_account')拿到了我们的期货账户，并且通过 context.get_symbol()拿到了我们目前的期货主力对应的期货品种。

现在，开始编写我们的策略逻辑。运用 context 的 current_minute 获取当前的分钟时刻，判断其是否是 9 点半，如果是，则运用期货账户的 order 方法买入一手沪深 300 期货。需要注意的是，期货账户的下单方法要比股票多一个参数，那就是方向。因为期货既可以做多也可以做空，所以我们在这里详细地分析期货账户的 order 方法。

5. futures_account.order(symbol,amount,offset_flag, order_type='market', price=0)

该函数为期货下单函数。对其参数说明如下。

◎ 为 symbol 时，指需要交易的期货代码。

◎ 为 offset_flag 时，指开仓或平仓参数，可选 open 和 close 这两个参数，open 为开仓，close 为平仓。

◎ 为 amount 时，指需要交易的期货代码为 symbol 的证券数量，其正负含义需与 offset_flag 结合起来看。若 offset_flag 为 open，则 amount 为正就是买入开仓，为负就是卖出开仓；若 offset_flag 为 close，则 amount 为正就是买入平仓，为负就是卖出平仓。

◎ 为 price 时，指下限价单时的下单价格（仅日内策略可用）。

◎ 为 otype 时，可选'market'（市价单）和'limit'（限价单）这两个值，表示交易指令类型（为 limit 时仅日内策略可用）。

接下来，我们在 9 点 36 分时首先通过 futures_account.get_positions()获取期货账户的持仓情况，获取期货的多头持仓数量 long_amount。get_positions()方法返回的虽然也是一个字典，它的键是对应的期货代码，但它的值是包含持仓信息的一个实例。其实例包含的属性如表 5-3 所示。

表 5-3

属　性	类　型	注　释
long_amount	int	多头持仓数量
short_amount	int	空头持仓数量
long_margin	float	多头保证金
short_margin	float	空头保证金
long_cost	float	多头平均开仓成本
short_cost	float	空头平均开仓成本
profit	float	持仓浮动盈亏（随市场价格实时变动）
today_profit	float	当日浮动盈亏（以逐日盯市方式计算）

所以，我们在 9 点 36 分做的就是把持有的仓位给平了。而对于止损是在 9 点半至 9 点 36 分之间做的，我们首先判断目前是否有持仓，接着判断持仓是否已经亏了 5000 元，如果是，则提前卖出它。

5.4　策略回测详情

单击策略净值右上方的“回测详情”按钮，回测平台会给出详细的调仓持仓及风险收益指标等数据。如图 5-7 所示是我们的股票策略的回测详情。

图 5-7

同时，优矿还提供了策略结果展示函数，如下所述。

1. bt

指回测报告，格式为 pandas.DataFrame，包括日期、现金头寸、证券头寸、投资组合价值、参考指数收益率、交易指令明细表等 6 列，以及用户在 observe 中定义的其他列。

时间从开始日期及需要获取的最长历史窗口后开始。

其用法为在运行完成策略后，在 code 单元中输入 bt 并运行，然后查看结果。

2. bt_by_account

指回测报告，格式为 pandas.DataFrame。包括日期、现金头寸、证券头寸、投资组合价值、参考指数收益率、交易指令明细表等 6 列，以及用户在 observe 中定义的其他列。

时间从开始日期及需要获取的最长历史窗口后开始。

其用法为在运行完成策略后，在 code 单元中输入 bt_by_account 并运行，然后查看结果。

3. perf

指根据回测记录计算各项风险收益指标，格式为字典，键为指标名称，值为指标的值，有些为 float，有些为 list。如表 5-4 所示。

表 5-4

属　性	类　型	注　释
returns	list	策略日收益率
cumulative_reurns	list	策略累计收益率
cumulatie_values	list	策略累计价值
benchmark_returns	list	参考标准日收益率
benchmark_cumulatie_returns	list	参考标准累计收益率
benchmark_cumulatie_values	list	参考标准累计价值（初始值为 1）
benchmark_annualized_return	list	参考标准年化收益率
annualized_return	list	策略年化收益率
treasury_retiurn	list	同期无风险收益率
excess_returns	list	策略相对无风险收益率的超额收益

续表

属　性	类　型	注　释
Alpha	list	策略 CAPM 阿尔法
beta	list	策略 CAPM 贝塔
sharpe	list	策略年化夏普率
volatility	list	策略年化波动率
max_drawdown	list	策略最大回撤
information_coefficient	list	信息系数
information_ratio	list	信息比率
turnover_rate	list	换手率

其用法为在运行完策略后，在 code 单元中输入 perf 并运行，然后查看结果。

5.5　策略的风险评价指标

通过利用策略的风险评价指标，我们可以在各个维度都对策略有客观、全面的认识。优矿回测框架提供了如下风险指标。

1. 年化收益率（Annualized Returns）

表示投资期限为一年的预期收益率，计算公式为

$$p_r = \left(\frac{p_{end}}{p_{start}} \right)^{(250/n)} - 1$$

其中，p_{end} 指策略最终总资产，p_{start} 指策略初始总资产，n 指回测交易日数量。

2. 基准年化收益率（Benchmark Returns）

表示参考标准年化收益率，计算公式为

$$B_r = \left(\frac{B_{end}}{B_{start}} \right)^{(250/n)} - 1$$

其中，B_{end} 指基准最终值，B_{star} 指基准初始值，n 指回测交易日数量。

3. 阿尔法（Alpha）

表示在投资中面临的非系统性风险。Alpha 是投资者获得的与市场波动无关的回报，一般用来度量投资者的投资技能。比如投资者获得了 12%的回报，其基准获得了 10%的回报，那么 Alpha 或者价值增值的部分就是 2%。其计算公式为

$$\alpha = p_r - r_f - \beta(B_r - r_f)$$

其中，P_r 为策略年化收益率，r_f 为无风险收益率，B_r 为基准年化收益率。

相应的 Alpha 值及释义如表 5-5 所示。

表 5-5

Alpha 值	释　义
$a>0$	策略相对于风险获得了超额收益
$a=0$	策略相对于风险获得了适当收益
$a<0$	策略相对于风险获得了较少收益

4. 贝塔（Beta）

表示在投资中面临的系统性风险，反映了策略对大盘变化的敏感性。例如，一个策略的 Beta 为 1.3，则在大盘涨 1%时，策略就可能涨 1.3%，反之亦然；如果一个策略的 Beta 为-1.3，则说明在大盘涨 1%时，策略可能跌 1.3%，反之亦然。其计算公式为

$$\beta = \frac{Cov(p_n, B_n)}{\sigma_m^2}$$

其中，pn 指策略每日收益率，Bn 指基准每日收益率，σ_m^2 指基准每日收益方差，Cov(pn,Bn)指策略每日收益率和基准每日收益率的协方差。

5. 夏普比率（Sharpe Ratio）

表示每承受一单位总风险，会产生多少超额报酬，可以同时对策略的收益与风险进行综合考虑。其计算公式为

$$SharpRatio = \frac{p_r - r_f}{\sigma_p}$$

其中，p_r 指策略年化收益率，r_f 指无风险收益率，σ_p 指策略收益波动率。

6. 收益波动率（Volatility）

用于测量资产的风险性，波动越大，代表策略风险越高。其计算公式为

$$\sigma_p = \sqrt{\frac{250}{n-1}\sum_{i=1}^{n}(p_t - \overline{p_t})^2}$$

其中，n 指回测交易日数量，p_t 指策略每日收益率，$\overline{p_t}$ 指策略每日平均收益率。$\overline{p_t}$ 的计算公式为

$$\overline{p_t} = \frac{1}{n}\sum_{i=1}^{n} p_n \text{。}$$

7. 信息比率（Information Ratio）

用于衡量单位超额风险带来的超额收益。信息比率越大，则说明该策略单位跟踪误差所获得的超额收益越高。因此，信息比率较大的策略的表现要优于信息比率较低的基金。合理的投资目标应该是在承担适度风险的情况下，尽可能地追求高信息比率。其计算公式为

$$InformationRatio = \frac{p_r - B_r}{\sigma_t}$$

其中，p_r 指策略年化收益率，B_r 指基准年化收益率，σ_t 指策略每日收益差值与基准每日收益差值的年化标准差。

8. 最大回撤（Max Drawdown）

用于描述策略可能出现的最糟糕的情况，计算公式为

$$MaxDrawDown = max(1 - \frac{p_x}{p_y})$$

其中，p_x、p_y 指策略某日总资产（证券持仓和现金之和），$y>x$。

9. 换手率（Turnover Rate）

用于描述策略变化的频率及持有某只股票平均时间的长短，计算公式为

$$TurnOverRate = \frac{p_x}{p_{avg}}$$

其中，p_x 指买入总价值和卖出总价值中的较小者，p_{avg} 指虚拟账户的平均价值。

5.6 策略交易细节

1. 回测交易撮合机制和订单委托

在回测时，订单撮合过程发生在运行 handle_data 函数结束之后，是以历史实时行情进行的虚拟撮合。由于是对真实场景的模拟，所以订单并不会立刻以某个价格成交，而是通过和实时行情的具体价格和具体成交量进行比对，来断定成交价格和成交时间。

由于订单撮合是以一篮子订单形式处理的，所以在处理订单时会有细微的调整。具体的交易细节和订单撮合机制如下所述。

1）日线级别策略回测

撮合机制遵循“先卖后买，开盘价撮合”原则，即先处理卖出订单，后处理买入订单。卖出订单产生的现金会参与买入订单的交易。由于下单是在当天开盘前进行的，所以订单撮合会与当天的开盘价进行比较，如果满足条件，就会撮合成交；如果不满足条件，则继续挂单，等待下一次撮合尝试。

2）分钟线级别策略回测

撮合机制遵循“先下单先处理，开盘价撮合”原则，即对先下单的订单先进行撮合尝试。订单撮合会与下一分钟 K 线的开盘价进行比较，如果满足条件，就会撮合成交；否则继续挂单，等待下一次撮合尝试。

对于不同的订单类型，成交条件如下。

（1）市价单。会以下一根 K 线的开盘价（openPrice）撮合成交，成交量不超过下一根 K 线的总成交量，如果超过，则剩余订单申报量等待下一次撮合成交。

（2）限价单。会以下一根 K 线的开盘价（openPrice）判断是否成交，如果买单申报价小于 openPrice 或卖单申报价大于 openPrice，则不成交，等待下一次撮合成交。成交量不超过下一根 K 线的总成交量，如果超过，则剩余订单申报量等待下一次撮合成交。

在当日收盘后，所有未成交的订单都将被系统自动撤销。

不满足订单委托限制的订单会被拒绝，变为废单。具体原因可能如下。

◎ 买入时，账户里的可用现金不够。

◎ 卖出时，账户里的可用持仓不够。

◎ 限价单委托价格超过涨跌停价格。

◎ 下单数量不足一手（股票为 100 股）。

◎ 股票当日停牌。

◎ 股票当日未上市或已退市。

订单未成交或部分成交的原因可能如下。

◎ 买入时，所需买进的数量超过当天或当分钟的成交量。

◎ 卖出时，所需卖出的数量超过当天或当分钟的成交量。

◎ 股票涨停（不能买入）或跌停（不能卖出）。

2. 滑点

在真实的证券成交环境下，下单的点位和最终成交的点位往往有一定的偏差，在订单委托到市场后，会对市场的走向造成一定的影响，比如买单会提高市场价格，卖单会降低市场价格。这类冲击成本会造成滑点的出现。优矿为了更真实地模拟策略在真实市场的表现，在回测时提供了滑点模式，用于评估由于市场冲击和交易价格变化引起的交易成本变化。我们可以通过设置 slippage 关键字来设置具体的滑点信息，默认的滑点为 0。

3. 交易税费

交易税费主要包含券商手续费和印花税。优矿回测默认采用买入千分之一、卖出千分之二的税费。如果需要进行更加细微的调整，则可以通过设置回测初始化参数 commission 关键字来自定义税费信息。

◎ 券商手续费：券商征收的下单手续费，中国 A 股市场目前为双边收费，每个券商的手续费不同。

◎ 印花税：是国家强制征收的印花税，目前对卖方单边征收，对买方不再征收，为0.1%。

4. 停牌退市

如果委托的订单在当天有停牌退市等不可以交易的情况，则订单会被拒绝，变为废单。

5. 涨跌停

如果委托的订单在当天一直涨停或者跌停，无法买入或卖出，则订单会一直处于挂单状态，直到收盘。

6. 期货连续合约跳空处理

连续合约是由不同的合约拼合而成的，前后两个合约存在价差，所以连续合约有明显的价格跳空现象。在前后合约的价差比较大时，会引起策略出现假信号，造成信号失真。优矿提供了价差平移法和前复权法，对连续合约时间序列进行处理，减少假信号造成的影响，同时提供了合约指数，可以直接根据合约指数进行信号生成。

至此，我们已经了解了整个优矿回测框架。其中的函数和参数看似很多，难以记忆与使用，但是对真实策略的建立并非需要对全部参数进行设置，对大部分参数使用默认值即可。

第 6 章

常用的量化策略及其实现

在前面章节的基础上，本章将展示什么是量化投资，以及如何利用优矿量化平台编写各种策略，其中的某些策略可作为大家学习与构建策略时的参考，有些可以直接用来建立股票自选池甚至可以直接实盘，请大家仔细阅读和研习。

6.1 量化投资概述

6.1.1 量化投资简介

量化投资是借助量化金融分析方法进行资产管理的一种投资方法，而量化金融分析方法是一种结合了金融数据、个人经验、数学模型及计算机技术的复杂金融建模及分析方法，其中的每个方面都有自己的细分领域，如下所述。

- 金融数据：行情数据、高频数据、因子数据、新闻数据、地理信息数据、社交数据。
- 个人经验：基金经理的个人投资经验。

◎ 数学模型：统计分析、机器学习、深度学习。

◎ 计算机技术：简单计算、并行计算。

6.1.2 量化投资策略的类型

量化投资策略有很多种分类方式，比如按照标的分类、按照技术分类、按照组合构建方式分类等。每种分类方式之间都不是互相独立的，每家投资机构内部的分类方式也各有特点。在此我们按照比较常见的按照标的分类的方式大概做一个梳理。

按照标的分类，基本上都是围绕股票、基金、期货、期权、债券、海外资产来进行分类的。需要特别说明的是，人们一般不会单纯地投资某种资产，在大多数情况下对基金都会投资一种资产以上；而且，在每一种资产下都会有很多细致的分类。本节只列出如下常见的资产类别。

（1）股票策略

（2）基金策略

◎ 指数基金

◎ 分级基金

◎ 股票型、债券型、货币型、衍生证券型基金

（3）期货策略

◎ 股指期货

◎ 商品期货

（4）期权策略

◎ 指数期权

◎ 个股期权（目前在国内尚未开放个股期权）

（5）债券策略

◎ 国家债券

◎ 政府债券

◎　上市公司债券

◎　未上市公司债券

◎　其他资产债券、组合债券（比如美国次贷危机中的主角：次级债券）

（6）海外资产策略

◎　直接投资海外市场证券

◎　通过国内跟踪海外市场的基金来进行间接投资

6.1.3　量化研究的流程

量化研究的流程比较系统和复杂，一般来说，一个策略从无到有都会经历很多步骤，不同的公司、机构对这整个流程也会有各自的分类。我们在内部交流后，推出了一个自认为比较合理的常见的量化策略从无到有的完整流程，如图 6-1 所示，其中的每个步骤都有很多细节需要处理。

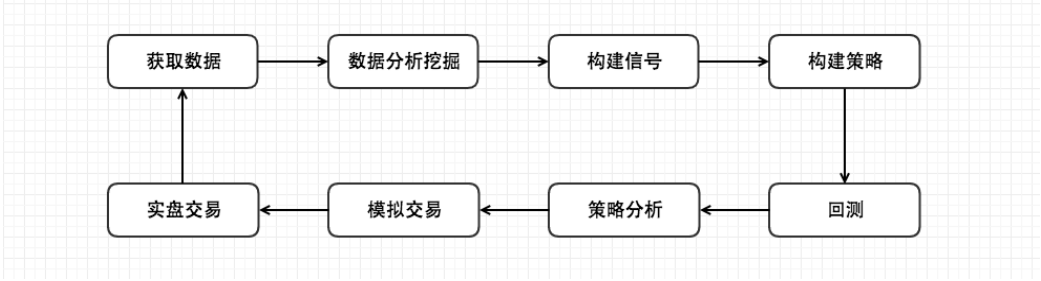


图 6-1

下面简单列出在如图 6-1 所示的步骤中需要处理的常见细节。

（1）获取数据

◎　公司财务数据

◎　公司新闻数据

◎　公司关联数据，以及产业上下游、主营业务、所属行业主题等数据

◎　基本行情数据

◎ 高频数据、股票 Level-1 数据、股票 Level-2 数据、期货 Level-1 数据

（2）数据分析挖掘

◎ 传统分析方法

◎ 新兴大数据、机器学习、数据挖掘方法

（3）构建信号

◎ 在构建信号前进行数据处理、标准化、去极值、中性化

◎ 基础信号的研究、分组回测、ic、ir、衰减、行业分布

◎ 将基础信号合成复杂信号

（4）构建策略

◎ 策略模版，兼容不同标的的策略，适用于股票、基金、期货等金融资产

◎ 兼容日线、分钟线甚至 Tick 级别的策略

◎ 方便好用的策略函数，获取历史行情、历史持仓信息、调仓记录等

◎ 支持各种订单类型：止盈止损单、限价单、市价单

（5）回测

◎ 完美符合历史的真实行情

◎ 股票分红送转、除权除息处理

◎ 股票涨跌停处理

◎ 股票停复牌处理

◎ 市场冲击、交易滑点、手续费、期货保证金交易

◎ 大单分笔成交处理等

（6）策略分析

◎ 策略归因、风险归因、实时监控

◎ 订单分析、成交分析、持仓分析、交易行为分析

◎　多策略分析，方便构建 FOF、MOM 类型的策略

（7）模拟交易

◎　接入实时行情、实时获取成交回报

◎　篮子交易、算法交易

◎　支持撤单处理

◎　实时监控、实时归因分析

（8）实盘交易

◎　接入真实券商账户

◎　极速行情、实时下单、实时获取订单回报

历史一直在变，唯一不变的是历史一直在发展。三次工业革命的每一次都推动人类历史上了一个新台阶，回顾世界经济、国外资产管理、量化投资的发展，我们同样发现，几乎每一次金融变革都是科学技术升级换代的产物；或者说每当金融危机发生之后，金融界总会寻求技术、规则上的改变，每一次由金融危机导致的金融变革，无一不在当时世界先进的科学技术上寻求支撑。

我们认为，不论是世界的发展还是金融的进步，都离不开技术和科技。这是一个科技变革、技术驱动的社会，先进的科学技术就像一层层厚实的钢筋混凝土，而金融的发展是在这些坚实的基础上林立而起的高楼大厦。

在 2010 年之前，我们只听过 IT（Internet Technology）时代的说法，当时流行的是 Technology Will Shape the Future 的理念；如今，从最近几年开始，特别是在 2014 年后，人们渐渐发现 IT 只是一种工具或一种载体，真正核心的东西是数据。后来，马云还提出了 DT（Data Technology）的理念，而现在都在宣扬 Data Will Shape the Future。相比之前，人们现在更关注这个世界上的核心元素，而非计算、分析这些核心元素所需要的工具。

在量化投资领域，数据是最大的核心，甚至是唯一的核心，基本上每个操作及决策都离不开大量复杂的数据分析和计算。回顾资产管理、量化投资的发展，再反观国内金融政策的变革、金融市场的进步，我们认为，在量化投资领域，国外已日渐成熟，而国内还处于起步发展阶段。虽然说国内起步较晚，资源较少，市场不够成熟，但是有国外这么多成熟市场的先前经验，有已经发展成熟的技术工具，我们相信，现在的国际市场环境及科学

技术环境，对于国内量化投资发展来说是一个完美的孵化器。

6.2 行业轮动理论及其投资策略

6.2.1 行业轮动理论简介

行业轮动理论自 2004 年美林证券提出投资时钟理论以来，逐渐成为一个重要的研究热点。行业轮动指在股市中行业指数收益率的有规律的此起彼伏的现象。这里所说的行业是指一组主营业务类似的公司的集合，这些公司的股票在市场上的表现通常比较接近。行业轮动说明了三点：首先，在近期刚刚热起来的行业将会继续在一段时间内表现较好；然后，这些表现较好的行业终将不会再表现得那么好，但是其他行业开始表现得更好；最后，这些行业此起彼伏的表现是可能被预测的，并且和商业周期非常相关。

6.2.2 行业轮动的原因

对行业轮动现象的解释有很多，但大体以实体经济和行为金融为主。从实体经济出发，不同的行业存在着不同的赢利周期，而且经济大环境也存在着周期性的变化。股票市场的行业轮动是实体经济变化的一个映射，但是这个映射并不仅仅反映现在的情况，更主要反映将来的情况；同时，这个将来的情况也只是投资者主观预期的情况，未必就是将来的实际情况。从行为金融学出发，投资者存在认知与情感的缺陷，其中认知的缺陷可以通过不断学习来克服，而情感上的缺陷却很难克服，所以行业轮动也在很大程度上受到投资的行为模式的影响。

1. 从产业链的角度来看行业轮动

在传统上，我们把产业链划分为上中下游：上游一般指资源品，例如煤、石油、铁矿石和有色金属；中游行业指制造业，例如化工、电机制造、钢铁等；下游行业主要指与消费直接相关的行业。

收入复苏时间从早到晚排列为：下游、中游和上游。收入弹性由大到小排列为：上游、中游和下游。在经济复苏初期，在中上游还在下降的时候，下游就开始恢复了；其次是中游的复苏；上游的复苏较晚，但复苏的力度很大。在经济逐渐繁荣的过程中，中上游表现

出较大的弹性与力度，而下游的增长是平稳上升的。

在成本方面，中下游的成本和收入同步波动，而上游的成本在复苏初期平稳上升。据此推测，上游的毛利率在复苏初期快速上升。在经济的中期调整阶段，上游的成本并不会与收入出现同等幅度的下降，下降幅度小于收入下降的幅度，因此这段时间的毛利率会出现暂时的下降。中下游的成本和收入的同步性较高，因此其毛利率不会大幅波动。

毛利率弹性由大到小排列为：上游、下游、中游。在复苏初期，上游的毛利率即出现大幅的上升；此后，其毛利率会随收入出现一定的波动。在中期调整的后期，上游的毛利率下降明显，中下游的毛利率只呈现小幅下降。

净利率的弹性由大到小排列为：上游、中游、下游；净利率水平由高到低排列为：上游、下游、中游。毛利率和净利率在弹性方面的不同充分反映了中游的“传导者”角色；在大小方面的不同则说明了中游的“规模化”特征，即管理费用率相对较低，且受产销量影响较大。

利润同比增速由大到小排列为：下游、中游、上游，复苏的时间顺序与收入类似；利润增长的弹性与收入有所差别，由大到小排列为：上游、中游、下游。中游对需求的波动反应较大，利润同比增速的高点往往在经济周期繁荣顶点之前到来，而下游则明显滞后，在繁荣顶点之后才会到达高点。下游与经济周期正相关，且波动相对于中上游较小。

利润占比变化传递的信息与利润同比增速相似。在经济复苏初期，下游的利润占比明显偏高，这是因为中上游的利润弹性较大，经济衰退使中上游的利润大幅下降；在经济复苏至繁荣阶段时，中游利润占比膨胀，因为中游的利润复苏快于上游；在经济繁荣至衰退阶段时，由于上游的滞后性，其利润占比在这一时期快速扩大，在经济达到谷底时，下游的利润占比大幅下降，而下游又再次短期增加。

可以看出，处于不同上下游的行业有着明显不同的赢利周期，更重要的是赢利的弹性不同，导致每个行业的表现存在自己的特性，也就形成了行业轮动的条件。这也说明：处于中上游的行业由于利润弹性大，导致股市估值预期变化大，更能体现行业轮动的特征。

2. 从行为金融学的角度来看行业轮动

在国外也有一些关于行为金融学结合行业轮动的研究，Moskowitz 与 Grinblatt（1999）得到了这样的结论：行业指数的惯性（动量）效应解释了市场上的大部分惯性效应；如果控制其他市值、估值因素等，则行业指数仍呈现出明显的惯性效应；如果控制行业因素，

则全市场的惯性效应将不会再显著。

国内也有许多研究者从投资者行为模式的角度来解释板块（行业）联动：何诚颖（2001）运用现代资本市场理论和行为金融学理论对板块现象的分析表明，板块现象是一种市场投机，其形成与中国股市投资者的行为特征密切相关。陈梦根与曹凤岐（2005）认为在转轨经济和新兴市场中，投资者容易受政策预期的影响，并且决策行为趋同，这就强化了股价冲击的传导机制，使股市呈现出齐涨同跌的现象。陈鹏与郑翼村（2006）研究了我国股市板块联动现象的表现形式，探讨了板块联动现象的成因，将板块联动现象划分为 4 种基本类型：基本面变化引致型、概念驱动型、庄家操纵型和无风起浪型，认为噪声投资者的非理性行为是构成板块联动现象的最主要原因。刘博和皮天雷（2007）认为在我国股市中投资者普遍认同的一个概念就是“补涨”，没涨的股票要无条件补涨，没跌的股票应当无条件补跌，于是形成了股市的各大板块“齐涨共跌”的局面。

6.2.3 行业轮动投资策略

基于行业轮动的原因及表现出来的特性，研究者开发出多种行业轮动策略，不过主要还是从上文提到的两个角度进行构建的。

在开发一个行业轮动策略时，首先要面临的问题就是对行业分类的选择。对行业分类的选择对后面的策略影响是很大的。

国内常用的行业分类标准有申万、证监会、中证、中信等。

国际上常用的行业分类标准有联合国制定的 ISIC（全称是 International Standard Industrial Classification of All Economic Activities）、欧盟制定的 NACE（全称是 Statistical Classification of Economic Activities in the European Community）、美国制定的 SIC（Standard Industrial Classification）和标准普尔与摩根士丹利共同制定的 GICS（Global Industry Classification Standard）等。

实际上，我们也可以根据自己的需求，利用聚类算法来构建一套符合自身需求的行业分类体系。

1. 策略介绍

上面已经对行业轮动策略的概念和原因进行了详细介绍，这里不再赘述。本策略从行业历史收益率的角度来看行业轮动，希望从行业指数的历史数据中找出在统计上具有显著

相关性的行业。

按照行业轮动理论的思想，本策略选取的行业标准和策略思想如下。

◎ 行业：申万一级行业（28 个，包括综合行业）。

◎ 数据：从 2015 年 1 月 1 日至今的历史上的所有申万一级行业的每日行情数据。

◎ 思想：首先，利用行业的每日行情数据计算出各行业的月度行情，根据月度行情数据计算出月度收益率数据；其次，根据月度收益率数据对各行业的每月收益进行排序，得到排序矩阵；最后，将所有的行业月度收益率排序数据与行业的一阶滞后月度收益率排序做相关性分析，并选择出在统计上有显著相关性的行业。

2. 计算月度行业收益率

计算月度行业收益率的代码如下：

```
    import numpy as np
    import pandas as pd
    import itertools
    from datetime import datetime
    from dateutil.parser import parse
    from scipy import stats as ss
    import matplotlib.pyplot as plt
    import seaborn as sn
    sn.set_style('white')
    def get_sw_ind_quotation():
        """
        返回申万一级行业指数的所有历史行情
        Args:
            opt(bool):选择是否剔除综合行业，默认不剔除
        Returns:
            DataFrame: 申万一级行业指数日线行情
        Examples:
            >> df_daily_industry_unstack = get_sw_ind_quotation()
        """
        # 拿取申万一级行业指数代码，一共有 28 个
        index_symbol = DataAPI.IndustryGet(industryVersion=u"SW",industry
VersionCD=u"",industryLevel=u"1",isNew=u"1",field=u"",pandas="1")['indexSymb
ol'].tolist()
        index_symbol = [str(item) + '.ZICN' for item in index_symbol]
```

```
    # 加上后缀，使得下面的行情 API 可以调用
        symbol_history_list = [ ]
        for symbol in index_symbol: # 该 API 每次只能调用一个 indexID
            df_daily_industry_symbol = DataAPI.MktIdxdGet(beginDate=
'2015-01-01', endDate='2018-01-23', ticker=symbol[:6], field=u"ticker,
tradeDate, closeIndex")
            symbol_history_list.append(df_daily_industry_symbol)
        df_daily_industry_symbol = pd.concat(symbol_history_list,axis=0)
    # 将获取的行业数据汇总
        df_daily_industry_unstack = df_daily_industry_symbol.set_index
(['tradeDate','ticker']).unstack()['closeIndex']# 将所有行业行情数据转为一张二维表
    return df_daily_industry_unstack
    df_daily_industry_unstack = get_sw_ind_quotation()
    df_daily_industry_unstack.head() # 如上我们便得到了申万一级行业至今的行情数据
```

接下来，将日线的行情数据转化为周线的行情数据，因为在第 1 天的行情中，有的行业存在缺失值，所以我们将第 1 天的行情剔除。定义月行情为每月的第 1 个交易日到下个月的第 1 个交易日：

```
    df_daily_industry_unstack = df_daily_industry_unstack.iloc[1:]
    # 去掉第 1 期基期
    df_daily_industry_unstack['tradeDate'] = df_daily_industry_unstack.index
# 加入 tradeDate 列，方便做 map
    def getMonthlyIndex(df_index):  # 得到月线行情
    # 将 tradeDate 列转化为时间格式
        df_index['tradeDate'] = df_index['tradeDate'].map(lambda x:parse(x))
        df_index['year_month']=df_index['tradeDate'].map(lambda
x:(x.year,x.month)) # 得到（年，月）用于筛选
        return df_index.groupby(['year_month']).head(1)
    # 返回每个月第 1 个交易日的行情
    df_monthly_industry_unstack = getMonthlyIndex(df_daily_industry_unstack
[:])
    df_monthly_industry_unstack = df_monthly_industry_unstack.sort_values
(['tradeDate']) # 按 tradeDate 列排序
```

这样，我们便得到了申万一级行业每个月第 1 个交易日的行情数据。

3. 计算月度行业收益率排序矩阵

接下来计算月度收益率及其排名，通过调用 Series 的内置函数 pct_change()就可以很方便地计算收益率；通过调用 rank()函数按行排序就可以计算排名，默认为从小到大：

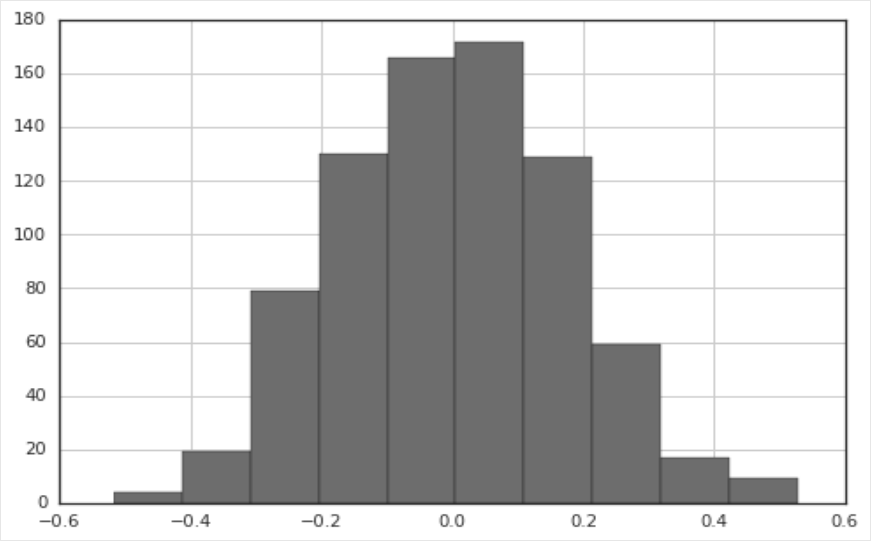
```
    del df_monthly_industry_unstack['tradeDate']
    # 删除 tradeDate 列和 year_month 列，方便后面调用后缀函数计算行业收益率
    del df_monthly_industry_unstack['year_month']
    df_monthly_industry_return = df_monthly_industry_unstack.pct_change
(axis=0) # 调用 pct_change()计算收益率
    df_monthly_industry_return = df_monthly_industry_return.dropna(how='all')
# 删除全是缺失值的第 1 行
    df_monthly_industry_return_rank = df_monthly_industry_return.rank(axis=1)
# 按行做排序
```

这样，我们便得到了申万一级行业每期收益率的排序情况。

4. 行业月度收益率相关性分析

获取申万一级行业指数代码，一共有 28 个：

```
    index_symbol = DataAPI.IndustryGet(industryVersion=u"SW",
industryVersionCD=u"",industryLevel=u"1",isNew=u"1",field=u"",pandas="1")['i
ndexSymbol'].tolist()
    def get_corr(ind1, ind2, df_ind):
        """
        返回行业 1 与行业 2 的 1 阶滞后的相关系数
        Args:
            ind1(str):行业指数代码
            ind2(str): 行业指数代码
            df_ind(DataFrame): 所有行业指数的月度收益表
        Returns:
            numpy.float64: 行业 1 与行业 2 的 1 阶滞后的相关系数
        Examples:
            >>ind_corr=get_corr('801760','801150',df_monthly_industry_
return_rank)
        """
        x = df_ind[ind1].iloc[0:-1].values
        y = df_ind[ind2].iloc[1:].values
        return np.corrcoef(x,y)[0][1]   #返回两组数据的皮尔森相关系数
    predict_corr = { }
    for item in itertools.product(index_symbol,repeat=2):
    #列表中的每个元素和剩余所有的元素进行配对
        predict_corr[item]=get_corr(item[0],item[1],df_monthly_industry_
return_rank) # 得到行业收益率排名的相关系数字典
    predict_corr = pd.Series(predict_corr) # 将字典转化为序列
    predict_corr.hist() # 做出相关系数的频数直方图
```



可以看出，行业与行业的一阶滞后的相关性并不高，大部分维持在 0 附近，我们要找出那些相关系数在统计上显著的两个行业。根据统计公式，相关系数不等于 0 的显著性 t 的统计量为：

$$t = \frac{r\sqrt{n-2}}{1-r^3}, df = n-2$$

查阅 t 的分布表，$t_{(34)01}$=2.728，反推出|r|=0.43，即当|r|>0.43 时，在 99%置信水平上是统计显著的。所以我们筛选出大于 0.43 或小于-0.43 时的行业相关性：

```
filter_corr = predict_corr[(predict_corr>0. 43) | (predict_corr<-0. 43)]
# 筛选相关系数大于 0.43 或小于-0.43 时的情况
filter_corr.sort_values(ascending=0)    #按照相关行业间的相关系数进行排序
```

5. 策略结果及分析

相关性显著的行业如表 6-1 所示。

表 6-1

行　业　1	行　业　2	corr
传媒	轻工制造	0.527449
电子	家用电器	0.523264
国防军工	轻工制造	0.460587

续表

行　业　1	行　业　2	corr
国防军工	综合	0.436172
农林牧渔	国防军工	0.431786
电气设备	建筑材料	-0.44004
商业贸易	食品饮料	-0.46866
食品饮料	纺织服装	-0.51695

如表 6-1 所示，我们可以得到所有统计显著的行业 1 与行业 2 一阶滞后的相关系数。从总体来看，相关系数比较大，行业相关性较强，但因为我们选取的是 2015 年 1 月之后的月度数据，数据样本量不够大，因此结果中的相关系数也较大。我们在进行实际操作的过程中可以根据自己的需求选择较长的数据区间进行筛选。虽然样本区间比较短，但我们可以将这个策略作为参考来理解行业轮动策略。

从表 6-1 中可以看到，传媒行业和轻工制造的一阶滞后的相关性最高，其次是电子行业和家用电器，它们的相关性都超过了 0.5。因此，如果传媒行业在这期涨得比较好，那么下一期我们应该超配轻工制造行业；如果电子行业在这期涨得比较好，那么下一期我们应该超配家用电器行业。另外，食品饮料和纺织服装的一阶滞后的相关性超过了-0.5，有较强的负相关性。

6. 行业轮动策略小结

本节从行业收益率排名的相关性出发，试图找出具有预测意义的各个行业之间的关系。通过分析行业与行业的一阶滞后的相关性，我们得到 8 个 99%置信水平是统计显著的关系。其中传媒行业和轻工制造、电子行业和家用电器的相关性最高均超过了 0.5，即这期传媒业的收益率排名与下一期轻工制造行业的收益率排名显著正相关；这期电子行业的收益率排名与下一期家用电器行业的收益率排名显著正相关。

6.3　市场中性 Alpha 策略

6.3.1　市场中性 Alpha 策略介绍

市场中性策略是指同时构建多头和空头头寸以对冲市场风险，无论市场处于上涨势还

是下跌势的环境下，均能获得稳定收益的一种投资策略。也就是说，市场中性 Alpha 策略是一类收益与市场涨跌无关，致力于获取绝对收益的低风险量化投资策略，其主要通过同时持有股票多头和期货空头，来获取多头组合超越期货所对应的基准指数的收益。

6.3.2 市场中性 Alpha 策略的思想和方法

Alpha 策略最初是由 William Sharpe 在 1964 年的著作《投资组合理论与资本市场》中首次提出的，并指出投资者在市场中交易面临系统性风险和非系统性风险，用公式表达如下：

$$E(R_p) = R_f + \beta \times (R_m - R_f)$$

其中，$\beta = Cov(R_i, R_m)/Var(R_m)$，$E(R_p)$表示投资组合的期望收益率，$R_f$ 表示无风险报酬率，R_m 表示市场组合收益率，β 为某一组合的系统风险系数。CAPM 模型主要表示单个证券或投资组合同系统风险收益率之间的关系，也就是说，单个投资组合的收益率等于无风险收益率与风险溢价之和。

Alpha 策略的思想是通过衍生品来对冲投资组合的系统风险 β，锁定超额收益 Alpha。因此首先需要寻找稳定的 Alpha，构建 Alpha 组合，进而计算组合的 β 来对冲风险。Alpha 策略成功的关键就是寻找到一个超越基准（具有股指期货等做空工具的基准）的策略。比如，可以构造指数增强组合+沪深 300 指数期货空头策略。这种策略隐含的投资逻辑是择时比较困难，不想承受市场风险。

该策略的方法如图 6-2 所示。

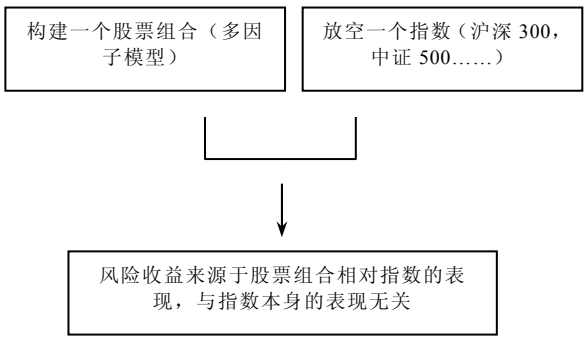


图 6-2

市场中性 Alpha 策略是一套科学且成熟的投资研究方法，在实际应用中，我们通常采用多因子模型驱动、期货对冲来获取长期稳定的 Alpha。同时，使用多因子模型可有效地

结合基本面和技术面，使策略更可靠、更稳健。另外，该策略的投资标的广、市场容量大，基本没有产品规模限制。

6.3.3　实例展示

通过上面的介绍，我们对市场中性 Alpha 策略有了较清晰的认识，这里利用多因子模型构建一个股票组合，同时选取基准指数对应的期货进行对冲，就可以构建一个完整的市场中性 Alpha 策略。

下面是一个具体的市场中性策略的实例。

（1）策略配置如下。

◎　回测区间：2015 年 1 月 1 日至 2015 年 8 月 26 日。

◎　基准：沪深 300 指数。

◎　股票池：沪深 300 动态成分股，在每个交易日都进行调仓。

（2）对因子的选取和处理如下。

◎　对因子的选取：净利润增长率、净资产收益率、相对强弱指数（RSI）。

◎　对因子的处理：用到了去极值（Winsorize）、标准化（Standardize）、中性化（Neutralize）处理。

◎　组合构建：等权配置。

（3）空头期货对冲为：空头使用期货 IF 当月合约做空来进行对冲。如果距离合约到期日有 3 天，则移仓换月。IFL0 表示当月合约，IFL1 表示次月合约。

代码展示如下：

```
from CAL.PyCAL import *
import numpy as np
from pandas import DataFrame
start = '2015-01-01'                                    # 回测起始时间
end = '2015-08-26'                                      # 回测结束时间
benchmark = 'HS300'                                     # 策略参考标准
universe = DynamicUniverse('HS300') + ['IFL0', 'IFL1']
# 证券池，支持股票和基金
capital_base = 10000000                                 # 起始资金
```

```
    freq = 'd'
    # 策略类型，'d'表示日间策略使用日线回测，'m'表示日内策略使用分钟线回测
    refresh_rate = 1
    cal = Calendar('China.SSE')
    period = Period('-1B')

    # 账户初始化配置
    stock_commission = Commission(buycost=0.0, sellcost=0.0, unit='perValue')
    futures_commission=Commission(buycost=0.0, sellcost=0.0, unit='perValue')
    slippage = Slippage(value=0, unit='perValue')

    accounts = {
        'stock_account':AccountConfig(account_type='security',capital_base=
capital_base, commission=stock_commission, slippage=slippage),
        'futures_account':AccountConfig(account_type='futures',capital_base=
capital_base, commission=futures_commission, slippage=slippage)
    }

    #策略算法
    def initialize(context):
        context.signal_generator = SignalGenerator(Signal('NetProfitGrowRate'),
Signal('ROE'), Signal('RSI'))
        context.need_to_switch_position = False
        context.contract_holding = ''

    def handle_data(context):
        # log.info(context.current_date)
        universe = context.get_universe(exclude_halt=True)

        yesterday = context.previous_date
        signal_composite = DataFrame()

        # 净利润增长率
        NetProfitGrowRate = context.signal_result['NetProfitGrowRate']
        signal_NetProfitGrowRate=standardize(neutralize(winsorize
(NetProfitGrowRate),yesterday.strftime('%Y%m%d')))
        signal_composite['NetProfitGrowRate'] = signal_NetProfitGrowRate

        # 权益收益率
        ROE = context.signal_result['ROE']
        signal_ROE=standardize(neutralize(winsorize(ROE),yesterday.strftime
```

```
('%Y%m%d')))
        signal_composite['ROE'] = signal_ROE

        # RSI
        RSI = context.signal_result['RSI']
        signal_RSI=standardize(neutralize(winsorize(RSI),yesterday.strftime
('%Y%m%d')))
        signal_composite['RSI'] = signal_RSI

        #  信号合成，各因子权重
        weight = np.array([0.6, 0.3, 0.1])
        signal_composite['total_score'] = np.dot(signal_composite, weight)

        # 组合构建
        total_score = signal_composite['total_score'].to_dict()
        wts = simple_long_only(total_score, yesterday.strftime('%Y%m%d'))
        handle_stock_orders(context, wts)
        handle_futures_orders(context)
        # handle_futures_position_switch(context)

        # 订单委托
        def handle_stock_orders(context, wts):
        account = context.get_account('stock_account')

        # 先卖出
        sell_list = account.get_positions()
        for stk in sell_list:
            account.order_to(stk, 0)

        # 再买入
        buy_list = wts.keys()
        total_money = account.portfolio_value
        prices = account.reference_price
        for stk in buy_list:
            if np.isnan(prices[stk]) or prices[stk] == 0:  # 停牌或者还没有上市等
原因不能交易
                continue
            account.order(stk, int(total_money * wts[stk] / prices[stk]
/100)*100)
    def handle_futures_orders(context):
        stock_account = context.get_account('stock_account')
```

```
        future_account = context.get_account("futures_account")

        # 将主力连续合约映射为实际合约
        contract_current_month = context.get_symbol('IFL0')

        # 判断是否需要移仓换月
        contract_holding = context.contract_holding
        if not contract_holding:
            contract_holding = contract_current_month

        if contract_holding:
            last_trade_date = get_asset(contract_holding).last_trade_date

            # 当月合约离交割日只有 3 天
            days_to_expire = (last_trade_date- context.current_date).days
            if days_to_expire == 3:
                log.info(u'距离%s 到期，还有%s 天' % (contract_holding, days_to_
expire))
                contract_next_month = context.get_symbol('IFL1')
                futures_position=future_account.get_position(contract_holding)
                if futures_position:
                    current_holding = futures_position.short_amount
                    log.info(u'移仓换月。[平仓旧合约:%s,开仓新合约:%s,手数:%s]' %
(contract_holding, contract_next_month, int(current_holding)))
                    if current_holding == 0:
                        return
                    future_account.order(contract_holding,current_holding,
"close")
                    future_account.order(contract_next_month,-1*current_
holding,"open")
                    context.contract_holding = contract_next_month
                    return
        stock_position = stock_account.get_positions()

        # 有多头股票仓位，使用期货进行空头对冲
        if stock_position:
            stock_positions_value=stock_account.portfolio_value-stock_account.
cash
            # print u'当前股票多头市值', stock_positions_value
            futures_position = future_account.get_position(contract_holding)
```

```
            # 没有空头持仓，则建仓进行对冲
            if not futures_position:
                contract_current_month = context.get_symbol('IFL0')
                multiplier = get_asset(contract_current_month).multiplier
                futures_price = context.current_price(contract_current_month)
                total_hedging_amount = int(stock_positions_value / futures_price
/ multiplier)
                log.info(u'%s 没有持仓，准备建仓。空头开仓%s 手' % (contract_current_
month, contract_holding))
                future_account.order(contract_current_month,-1*total_hedging_
amount,"open")
                context.contract_holding = contract_current_month

            # 已经有空头持仓，则判断是否需要调仓
            else:
                contract_holding = context.contract_holding
                contract_current_month = context.get_symbol('IFL0')
                futures_price = context.current_price(contract_current_month)
                multiplier = get_asset(contract_holding).multiplier
                # 计算当前对冲需要的期货手数
                total_hedging_amount = int(stock_positions_value / futures_price
/ multiplier)
                hedging_amount_diff=total_hedging_amount-futures_position.
short_amount
                # 调仓阈值，可以适当放大，防止反复调仓
                threshold = 2
                if hedging_amount_diff >= threshold:
                    log.info(u'空头调仓。[合约名:%s,当前空头手数:%s,目标空头手数:%s]'
%(contract_holding,int(futures_position.short_amount),total_hedging_amount))
                    # 多开空仓
                    future_account.order(contract_holding,-1*int(hedging_
amount_diff), "open")
                elif hedging_amount_diff <= -threshold:
                    log.info(u'空头调仓。[合约名:%s,当前空头手数:%s,目标空头手数:%s]'
%(contract_holding,int(futures_position.short_amount),total_hedging_amount))
                    # 平掉部分空仓
                    future_account.order(contract_holding,int(abs(hedging_
amount_diff)), "close")
```

回测的结果如图 6-3 所示。

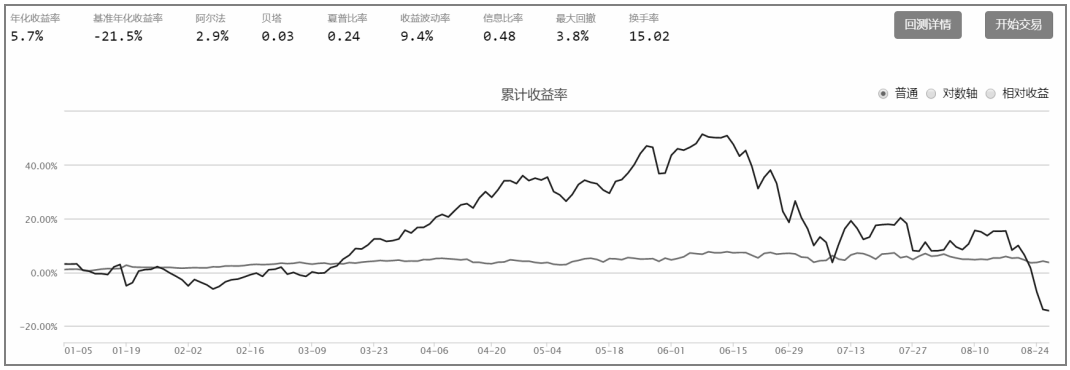


图 6-3

通过对回测的结果进行分析可以发现，从整体上看，策略的收益曲线非常平稳。从 2015 年 1 月 1 日到 2015 年 08 月 26 日，策略的年化收益达到 5.7%，而基准的年化为-21.5%，策略年化超额收益为 27.2%。还可以发现，策略的贝塔值仅为 0.03，表明对冲效果非常好，基本上对冲了市场的风险，实现了穿越牛熊。

6.4 大师策略

本节将用金融工程手段与方法，对国内外投资大师的经典投资理论进行介绍、实现、验证与分析。所谓大师策略，其实和基本面量化有异曲同工之妙，更偏向于主动投资管理，通过使用者对公司财务指标、市场方向等的顶层逻辑去建立相应的量化模型，再通过数量化手段去验证自己的逻辑。本节的大师策略不仅适用于量化投资者，也可为主动投资者和技术面投资者提供参考，希望读者能通过本节学习到大师策略量化的相关流程及实现方法。

在编写大师策略时，我们更关注于大师投资的逻辑与思路，尽量以量化指标还原大师的投资逻辑。但因为国内外市场的环境不同，所以我们偶尔会用更适合中国市场的指标对原有的大师策略进行替代与优化。使用大师策略时切忌“刻舟求剑”，也不推荐直接使用大师策略进行投资，因为从策略验证效果上看，许多大师策略在国内的复制效果并不理想。

6.4.1　麦克·欧希金斯绩优成分股投资法

提起麦克·欧希金斯，你可能并没有什么印象，然而提起“狗股理论”，你可能就略知一二了。不错，麦克·欧希金斯正是畅销书《战胜道琼斯》（*Beating the Dow*）及《用债券战胜道琼斯》（*Beating the Dow with Bonds*）的作者，也是狗股理论的发明人。

麦克·欧希金斯出生于委内瑞拉，由于他的父亲身处石油行业，所以他从小就辗转于南美、东南亚、北非和欧洲等多个国家生活及学习，其丰富的经历为他将来的投资生涯奠定了坚实的基础。

在获得锡耶纳学院的经济学学士学位后，麦克·欧希金斯曾在宝洁工作过一段时间，随后于 1971 年进入投资行业，担任 Spencer Trask＆Company 及 White,Weld＆Company 的证券经纪人，并于 1978 年年初开办了自己的资产管理公司。

麦克·欧希金斯一直是《时代周刊《巴伦周刊》《福布斯》《商业周刊》《金融世界》《华尔街日报《纽约时报》《今日美国》《金融时报》及 CNBC、PBS 的常客。他是典型的价值投资者，持股周期往往高于一年，是长期投资的典范。他的投资方法简单有效，即使对于没有精力深入基本面研究的投资者，也非常实用，他希望使用简单的策略来达到长期击败市场的目的。

1. 策略说明

麦克·欧希金斯所使用的选股方法极为简单，他将道琼斯指数的成分股作为股票池，挑选股息率最高的 10 家公司进行投资组合，每年进行一次调仓。以下是麦克·欧希金斯的基本投资逻辑。

（1）选取道琼斯工业指数 30 支成分股。

（2）选出股息率最高的 10 家公司，如果第 10 家及第 11 家股息率相同，则选择收盘价较低的个股。

（3）等权构建投资组合。

（4）每年调整一次投资组合。

2. 具体策略与量化实现

由于国内市场与美国市场有若干差异，因此我们针对 A 股市场，对麦克·欧希金斯的

选股标准进行了一定程度的本土化改良。

选股标准如下。

（1）沪深 300 成分股。

（2）选出股息率最高的 10 家公司作为投资组合。

我们按照如下方法进行回测。

◎ 数据：2007 年 5 月 8 日至 2017 年 12 月 31 日的历史上的所有 A 股，包括后来退市的 A 股。

◎ 每年进行一次调仓，调仓日为 5 月的第 1 个交易日，考虑停牌、涨跌停对交易的影响。

◎ 资金在选出股票上平均分配，交易费用为 1.3‰。对比指数为沪深 300。

测试代码如下。

首先，导入相关模块：

```
from __future__ import division
import pandas as pd
import numpy as np
import datetime
from dateutil.parser import parse
from CAL.PyCAL import *
import os
cal = Calendar('China.SSE')
```

然后，编写时间截面选股函数：

```
def MichaelHiggins(universe,date):
    """
    给定股票列表和日期，
    Args:
        universe (list of str): 股票列表（有后缀）
        date (str or datetime): 常见的日期格式支持
    Returns:
        list: PEG 最小的 30%，资产负债率最小的 50%，PCF 最小的 30%的股票的交集

    Examples:
        >> universe = set_universe('HS300')
```

```
            >> buy_list = PeterLynchSelect(universe,'20160909')
        """
        trade_date = date if isinstance(date,datetime.datetime) else parse(date)
        trade_date = trade_date.strftime('%Y%m%d')
        #获取股息率
        df_factor = DataAPI.MktStockFactorsOneDayGet(secID=universe,tradeDate=
trade_date,

field=['secID','CTOP'],pandas="1")
        df_factor.sort_values('CTOP', ascending=False, inplace=True)
        # 选取最高的 10 家
        df_factor_select = df_factor.head(10)
        sec_list = df_factor_select['secID'].tolist()
        return sec_list
```

其次，获取调仓日：

```
    tradedaylist=DataAPI.TradeCalGet(exchangeCD=u"XSHG,XSHE",beginDate=u"",e
ndDate=u"",field=u"",pandas="1")
    tradedaylist=tradedaylist[tradedaylist['isOpen']==1]
    tradedaylist=tradedaylist[tradedaylist.calendarDate>'2007-01-01']
    tradedaylist['mon']=tradedaylist.calendarDate.apply(lambda x:x[5:7])
    tradedaylist['year']=tradedaylist.calendarDate.apply(lambda x:x[:4])
    tradedaylist=tradedaylist.drop_duplicates(subset=['mon','year'],keep='fi
rst')
    t_date = tradedaylist.ix[tradedaylist.mon.isin(['05']),:]['calendarDate'].
values
    t_date = [datetime.datetime.strptime(x, "%Y-%m-%d") for x in t_date ]
```

最后，编写调仓逻辑：

```
    start = '2007-05-08'                    # 回测起始时间
    end = '2017-12-31'                      # 回测结束时间
    universe = DynamicUniverse('HS300')     # 证券池，支持股票和基金
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'd'                       # 'd'表示使用日频率回测，'m'表示使用分钟频率回测
    refresh_rate = 1                        # 执行 handle_data 的时间间隔
    commission = Commission(0.0013,0.0013)
    accounts = {
        'fantasy_account': AccountConfig(account_type='security', capital_
base=1000000)
    }
```

```
    def initialize(context):                    # 初始化策略运行环境
        pass

    def handle_data(context):                   # 核心策略逻辑
        account = context.get_account('fantasy_account')
        if context.current_date in t_date:
            position = account.get_positions()
            buy_list =MichaelHiggins(context.get_universe(exclude_halt=True),
context.previous_date) #exclude_halt=True
            # 判断持仓是否为空
            if len(position) > 0:
                # 获取停牌 secid
                notopen = DataAPI.MktEqudGet(tradeDate=context.now,
secID=position.keys(),isOpen="0",field=u"secID",pandas="1")
                sum_ = 0
                # 计算停牌 secID 的权益
                for sec in notopen.secID:
                    tmp = account.get_position(sec).value
                    sum_ += tmp
                buyweight = 1.0 - sum_/account.portfolio_value
            else:
                buyweight = 1.0
            for stk in position:
                # 先卖
                if stk not in buy_list:
                    account.order_to(stk, 0)
            if len(buy_list) > 0:
                weight = buyweight/len(buy_list)
            else:
                weight = 0
            for stk in buy_list:
                if stk  in account.get_positions():
                    account.order_pct_to(stk,weight)
            for stk in buy_list:
                if stk not in account.get_positions():
                    account.order_pct_to(stk,weight)
```

等待几分钟就可以得到回测结果了，如图 6-4 所示。

图 6-4

是不是很方便呢？我们还可以通过最上方的各项回测指标判断策略的优劣。如果想了解更多的回测结果分析，则可以关注华创证券金融工程组的大师系列研报专题（http://master.hcquant.com）。

6.4.2　杰拉尔丁·维斯蓝筹股投资法

杰拉尔丁·维斯（Geraldine Weiss）是美国著名的投资大师，于 1966 年创刊《投资质量趋势》（*Investment Quality Trend*），拥有长达 40 年的投资经验，也是投资咨询界的先驱者。更令人惊讶的是，维斯女士是第 1 位成为有执照的投资顾问的女性，这在以往以男性为主的投资顾问行业中开创了历史的先河。

维斯女士是伯克利加利福尼亚大学的毕业生，是一位证券分析师、作家、讲师，并长期担任金融广播和电视的嘉宾和贡献者。她的分析及报道遍布主要的金融期刊和报纸，例如《巴伦周刊》《华尔街日报》《财富》《福布斯》及《纽约时报》等，更是 CNBC、Wall Street Week 等访谈节目的常客。

维斯女士的主要投资观念为“股息效应”，并著有《股息如何创造价值》（*The Dividends Connection-How Dividends Create Value in the Stock Market*）及《股息从不说谎》（*Dividends*

Don't Lie）等畅销书籍，更被洛杉矶时报誉为“股息圣母”。

1. 策略说明

杰拉尔丁·维斯蓝筹股投资法不仅强调高股息，还看重公司的成长性及股票的流通性，这是策略在缺乏行业多样性的前提下依旧表现稳健的主要原因。以下是杰拉尔丁·维斯的主要基本投资逻辑。

（1）在过去 12 年内股息必须成长 5 倍。

（2）在标准普尔的评估体系中必须达到 A-或以上的评级。

（3）在外流通股数至少为 500 万股，以确保有充分的市场流通性。

（4）至少有 80 家机构持有该股。

（5）必须至少在 25 年内不间断地发放股息。

（6）在过去 12 年内公司的盈余必须至少有 7 年成长。

2. 具体策略与量化实现

由于国内市场与美国市场有若干差异，因此除借鉴杰拉尔丁·维斯的选股标准外，也可以整合以下客观、可量化的指标作为选股标准。

选股标准如下。

（1）股本数量大于市场平均值。

（2）选择沪深 300 成分股作为股票池。

（3）至少有 10 家以上的股票型基金持有该股。

（4）在 5 年内至少有 3 年净利润成长率大于 0。

（5）在过去 3 年内不间断地发放股息。

（6）过去 3 年的股息必须成长 0.5 倍以上。

（7）股息率≥4%

我们按照如下方法进行回测。

◎ 数据：2010 年 9 月 1 日至 2017 年 12 月 31 日的历史上的所有 A 股，包括后来退市的 A 股。

◎ 每年 3 次调仓，调仓日为 5 月、9 月、11 月的第 1 个交易日，考虑停牌、涨跌停对交易的影响。

◎ 资金在选出的股票上平均分配，交易费用为 1.3‰，对比指数为沪深 300。

测试代码如下。

首先，导入相关模块：

```
from __future__ import division
import pandas as pd
import numpy as np
import datetime
from dateutil.parser import parse
from CAL.PyCAL import *
import os
cal = Calendar('China.SSE')
```

然后，编写之后要用到的 group 函数：

```
def divgrowth(x):
    tmp = x['perCashDiv']
    if len(tmp) !=3:
        return 0.0
    else:
        return tmp.iloc[-1] / tmp.iloc[0]
```

其次，获取股票型基金列表：

```
fundlist = DataAPI.FundGet(category=u"E",field=u"ticker,secShortName,
establishDate,",pandas="1")
```

接着，编写时间截面选股函数：

```
def GeraldineWeiss(universe,date):
    """
    给定股票列表和日期，
    Args:
        universe (list of str): 股票列表（有后缀）
        date (str or datetime): 常见的日期格式支持
```

```
        Returns:
            list: PEG最小的30%，资产负债率最小的50%，PCF最小的30%的股票的交集

        Examples:
            >> universe = set_universe('HS300')
            >> buy_list = PeterLynchSelect(universe,'20160909')
        """
        trade_date = date if isinstance(date,datetime.datetime) else parse(date)
        trade_date = trade_date.strftime('%Y%m%d')
        start_date1 = cal.advanceDate(trade_date, Period('-1Y')).strftime
("%Y%m%d")
        start_date2 = cal.advanceDate(trade_date, Period('-2Y')).strftime
("%Y%m%d")
        start_date3 = cal.advanceDate(trade_date, Period('-3Y')).strftime
("%Y%m%d")
        start_date4 = cal.advanceDate(trade_date, Period('-4Y')).strftime
("%Y%m%d")
        #净利润增长、5年哑变量相加、现金流市值比（股息率）
        df_factor = DataAPI.MktStockFactorsOneDayGet(secID=universe, tradeDate
=trade_date,field=['secID','ticker', 'NetProfitGrowRate','CTOP'], pandas="1")
        df_factor['NetProfitGrowRate'] = np.where(df_factor
['NetProfitGrowRate'] > 0, 1, 0)
        df_factor.rename(columns={'NetProfitGrowRate': 'N'}, inplace=True)

        df_factor_1 = DataAPI.MktStockFactorsOneDayGet(secID=universe,
tradeDate=trade_date,field=['secID','NetProfitGrowRate'],pandas="1")
        df_factor_1['NetProfitGrowRate'] = np.where(df_factor_1
['NetProfitGrowRate'] > 0, 1, 0)
        df_factor_1.rename(columns={'NetProfitGrowRate': 'N1'}, inplace=True)
        df_factor = df_factor.merge(df_factor_1, on='secID')

        df_factor_2 = DataAPI.MktStockFactorsOneDayGet(secID=universe,
tradeDate=trade_date,field=['secID','NetProfitGrowRate'],pandas="1")
        df_factor_2['NetProfitGrowRate'] = np.where(df_factor_2
['NetProfitGrowRate'] > 0, 1, 0)
        df_factor_2.rename(columns={'NetProfitGrowRate': 'N2'}, inplace=True)
        df_factor = df_factor.merge(df_factor_2, on='secID')

        df_factor_3 = DataAPI.MktStockFactorsOneDayGet(secID=universe,
tradeDate=trade_date,field=['secID','NetProfitGrowRate'],pandas="1")
```

```
        df_factor_3['NetProfitGrowRate'] = np.where(df_factor_3
['NetProfitGrowRate'] > 0, 1, 0)
        df_factor_3.rename(columns={'NetProfitGrowRate': 'N3'}, inplace=True)
        df_factor = df_factor.merge(df_factor_3, on='secID')

        df_factor_4 = DataAPI.MktStockFactorsOneDayGet(secID=universe,
tradeDate=trade_date,field=['secID','NetProfitGrowRate'],pandas="1")
        df_factor_4['NetProfitGrowRate'] = np.where(df_factor_4
['NetProfitGrowRate'] > 0, 1, 0)
        df_factor_4.rename(columns={'NetProfitGrowRate': 'N4'}, inplace=True)
        df_factor = df_factor.merge(df_factor_4, on='secID')
        df_factor['N_all'] = df_factor['N'] + df_factor['N1'] + df_factor['N2']
+ df_factor['N3'] + df_factor['N4']

        # 获取股票型基金列表
        tmp_fundlist = fundlist[fundlist['establishDate'] <= trade_date]
        # 获取报告期
        if trade_date[5:7] == '04':
            reportDate = '{}-12-31'.format(str(int(trade_date[:4]) - 1))
        elif trade_date[5:7] == '08':
            reportDate = '{}-06-30'.format(trade_date[:4])
        else:
            reportDate = '{}-09-30'.format(trade_date[:4])
        fund_hold = DataAPI.FundHoldingsGet(reportDate=reportDate,secID=u"",
ticker=tmp_fundlist.ticker,secType="E",field=u"ticker,holdingTicker",pandas=
"1")
        fund_hold = fund_hold.groupby('holdingTicker').count()
        fund_hold.reset_index(inplace=True)

        # 获取总持仓基金个数
        fund_hold.rename(columns={'holdingTicker':'ticker','ticker':
'holdnumber'}, inplace=True)
        df_factor = df_factor.merge(fund_hold, on='ticker')

        # 获取总市值及收盘价
        df_factor1=DataAPI.MktEqudGet(secID=universe,tradeDate=trade_date,
field=u"secID,marketValue,closePrice",pandas="1")
        # 计算总股本
        df_factor1['totalshare']=df_factor1['marketValue']/df_factor1
['closePrice']
```

```
        df_factor = df_factor.merge(df_factor1, on='secID')

        # 获取三年股息率
        div = DataAPI.EquDivGet(eventProcessCD = '6',secID=df_factor.secID,
beginDate = start_date3,endDate=trade_date,field=['secID','perCashDiv',
'publishDate'],pandas="1")
        div.dropna(inplace=True)
        div = div[div['perCashDiv'] > 0]
        div['publishDate'] = [x[:4] for x in div['publishDate']]
        # 考虑一年多次发放的情况
        div.drop_duplicates(['publishDate', 'secID'], inplace=True)
        div_year = div.groupby('secID').count().reset_index()
        div_year.rename(columns={'perCashDiv': 'divyear'}, inplace=True)
        df_factor  = df_factor.merge(div_year, on='secID')
        # 计算股息率成长率
        div_growth = div.groupby('secID').apply(divgrowth).reset_index()
        div_growth.rename(columns={0:'divgrowth'}, inplace=True)
        df_factor = df_factor.merge(div_growth, on='secID')

        df_factor = df_factor.dropna()

        df_factor_select=df_factor[(df_factor['holdnumber']>=10)&(df_factor
['CTOP']>= 0.04)&(df_factor['totalshare'] >= df_factor['totalshare'].median())
&(df_factor['divyear']>=3)&(df_factor['N_all'] >= 3)  & (df_factor['divgrowth']
>= 1.5)]
        sec_list = df_factor_select['secID'].tolist()
        return sec_list
```

然后，获取调仓日：

```
    tradedaylist=DataAPI.TradeCalGet(exchangeCD=u"XSHG,XSHE",beginDate=u"",e
ndDate=u"",field=u"",pandas="1")
    tradedaylist=tradedaylist[tradedaylist['isOpen']==1]
    tradedaylist=tradedaylist[tradedaylist.calendarDate>'2007-01-01']
    tradedaylist['mon']=tradedaylist.calendarDate.apply(lambda x:x[5:7])
    tradedaylist['year']=tradedaylist.calendarDate.apply(lambda x:x[:4])
    tradedaylist=tradedaylist.drop_duplicates(subset=['mon','year'],keep='fi
rst')
    t_date = tradedaylist.ix[tradedaylist.mon.isin(['05']),:]['calendarDate'].
values
```

```
    t_date = [datetime.datetime.strptime(x, "%Y-%m-%d") for x in t_date ]
```

最后，编写调仓逻辑：

```
    start = '2007-05-08'                    # 回测起始时间
    end = '2017-12-31'                      # 回测结束时间
    universe = DynamicUniverse('HS300')     # 证券池，支持股票和基金
    benchmark = 'HS300'                     # 策略参考基准
    freq = 'd'                  # 'd'表示使用日频率回测，'m'表示使用分钟频率回测
    refresh_rate = 1                        # 执行 handle_data 的时间间隔
    commission = Commission(0.0013,0.0013)
    accounts = {
        'fantasy_account': AccountConfig(account_type='security', capital_
base=1000000)
    }

    def initialize(context):                # 初始化策略运行环境
        pass

    def handle_data(context):               # 核心策略逻辑
        account = context.get_account('fantasy_account')
        if context.current_date in t_date:
            position = account.get_positions()
            buy_list = GeraldineWeiss (context.get_universe(exclude_halt=True),
context.previous_date) #exclude_halt=True
            # 判断持仓是否为空
            if len(position) > 0:
                # 获取停牌 secid
                notopen = DataAPI.MktEqudGet(tradeDate=context.now,secID=
position.keys(),isOpen="0",field=u"secID",pandas="1")
                sum_ = 0
                # 计算停牌 secID 的权益
                for sec in notopen.secID:
                    tmp = account.get_position(sec).value
                    sum_ += tmp
                buyweight = 1.0 - sum_/account.portfolio_value
            else:
                buyweight = 1.0
            for stk in position:
                # 先卖
```

```
            if stk not in buy_list:
                account.order_to(stk, 0)
        if len(buy_list) > 0:
            weight = buyweight/len(buy_list)
        else:
            weight = 0
        for stk in buy_list:
            if stk  in account.get_positions():
                account.order_pct_to(stk,weight)
        for stk in buy_list:
            if stk not in account.get_positions():
                account.order_pct_to(stk,weight)
```

在等待几分钟后，我们就可以得到回测结果了，如图 6-5 所示。

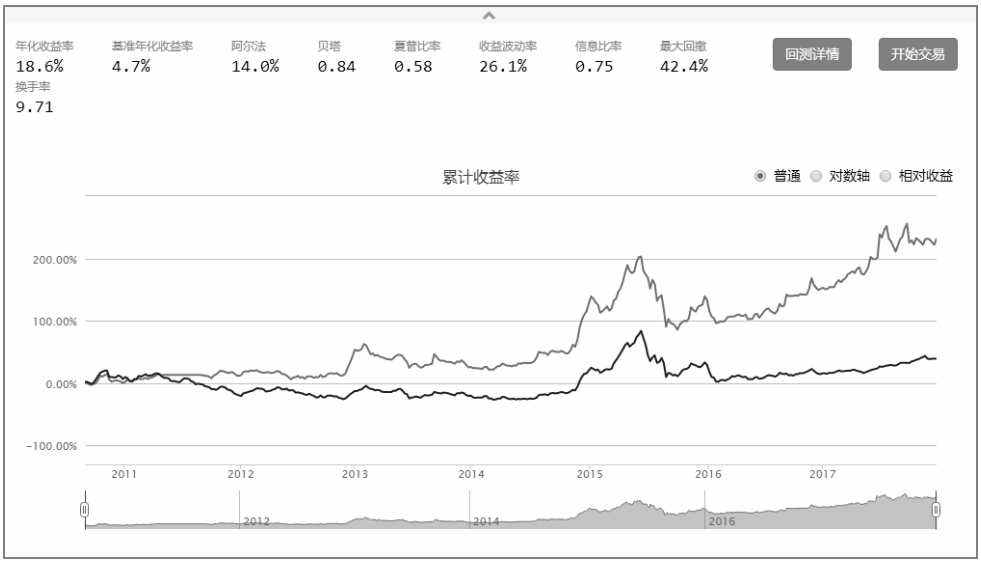


图 6-5

如果想了解更多的回测结果分析，则可以关注华创证券金融工程组的大师系列研报专题（网站：master.hcquant.com）。

本节的两个策略在 2016 年年初至 2017 年年底的收益十分不错，策略的顶层逻辑恰好契合了当下市场的价值投资氛围。随着市场的逐渐成熟，投资者逐渐理性，A 股逐渐港股化、美股化，大师类策略会有更广阔的发展空间。

6.5　CTA 策略

部分读者可能对 CTA 策略不太了解，本节会简单介绍 CTA 策略的概念。CTA 是 Commodity Trading Advisor Strategy 的缩写，从字面意思来看，是指商品交易顾问，也就是指投资于期货的资产管理产品。

从海外来说，CTA 基金的管理规模已经超过 3000 亿美元，业绩稳定，与其余市场策略的相关性较低，受到很多成熟投资者的青睐。但是对于国内来说，CTA 的起步较晚，规模较小，其余产品也多以自有资金为主，不接受外来资金，所以 CTA 并不为国内的投资者所熟知。

CTA 基金也分为两大阵营：主观 CTA 和量化 CTA。前者依靠基金管理人基于基本面调研或以往的交易经验来判断后续走势，决定何时买进卖出，做空做多，但是这种做法受市场及基金管理人的主观影响很大，所以其份额逐渐被量化 CTA 所蚕食，而后者正是本节的重点所在；量化 CTA 策略通过分析建立数量化的交易模型，并根据交易模型所产生的买卖信号进行投资决策。

量化 CTA 策略也分为两种：趋势跟随策略和均值回复策略。

- 趋势跟随策略利用大量的模型寻找当前的市场趋势，判断多空，只要存在大幅度的上涨和下跌，则都能获利，但是在市场波动较小或震荡市期间，这种策略可能因为不停地止损而出现回撤。
- 均值回复策略则主要应用在跨期、跨品种配对交易中，根据价差反转套利，和股票市场中的高抛低吸类似，核心思想是利用历史走势基本一致地对资产进行套利。

6.5.1　趋势跟随策略

在投资市场中，可将趋势形态简单地分为三种：上升趋势、下降趋势和震荡趋势。趋势跟随是一种基于价量分析的投资方式，其基本策略是在趋势开始形成时选择趋势方向买入，等待趋势结束后卖出。趋势跟随通常用作中长线策略或者周期较长的短线策略（三五天左右），很少应用在交易频率较高的日内交易中。鉴于趋势跟随赚取的是市场大涨或大跌状态下的钱，所以在长期盘整的震荡趋势中不适合配置趋势跟随策略。趋势跟随策略历

史久远，有很多著名的趋势跟随系统为聪明的投资者带来了巨大的财富，例如海龟交易法则、 Dual Thrust 交易法则、各类突破交易法则等。

期货市场是零和博弈，有盈有亏，投机商其实是在参与一场有技术含量的赌博。在赌场中，通常优势在庄家一边，因为庄家可以掌握更多的资源，所以可以更有力地影响市场。因此老练的玩家们在大多数时间都玩得很小，而是静待时机，等到确保大概率情况下的赢利势头后再加大赌注，从市场中获利。在期货市场中，集体的力量会构成趋势，在通常情况下的大笔赢利也都来源于这种趋势。趋势交易者从概率的角度去看待问题，从历史数据中判断在概率角度上有利的操作方式，遵从最大概率原则并顺势而为，避免预测未来，这样才能在长远的交易中取胜。当评判 CTA 策略的收益表现时，可以粗略地将某策略的收益拆解为两部分，如下式所示：

$$R = k \times r_{wl} r_{wr}$$

其中， R 代表策略收益； r_{wl} 代表盈亏比，为在一个交易段中赢利数额和亏损数额的比值； r_{w_r} 代表胜率，为在交易中赢利次数的占比， k 代表正则系数。

对于类型不同但表现相当的策略，其收益来源可能会有很大的差别。有的策略重在高盈亏比 r_{wl} ，在其交易中可能有的交易是亏钱的，但是亏小赚大，总体期望是赚钱的，通常趋势策略属于这一类；有的策略重在高胜率 r_{w_r} ，在交易次数较多的情况下，总体是能赚钱的，有不少反转策略属于这一类。

趋势交易者的信仰如下。

◎ 还不错的胜率：使用“还不错”来形容，旨在表明趋势策略通常追求盈亏比而非胜率。胜率在 40%以上的趋势策略，只要盈亏比够高，则在长期期望下也是赚钱的。

◎ 较高的盈亏比：趋势策略赚市场大涨大跌的钱，通常追求较高的盈亏比来达到赢利的目的。在趋势形成时通过持仓或加仓来扩大收益，在趋势结束时或方向判断错误时及时止损，可获得较高的盈亏比。

本节展示较为简单的双均线策略。

1. 双均线突破——无止盈止损

利用短期均线 MA_S 和长期均线 MA_L 生成开平仓信号。

◎　MA_S 上穿 MA_L，形成做多信号，买入开仓。

◎　MA_S 下穿 MA_L，形成做空信号，卖出开仓。

策略代码如下：

```
    #*********************************************#
    # 双均线突破策略 —— 无止盈止损
    # 短周期 MA 上穿长周期 MA，形成做多信号，买入开仓
    # 短周期 MA 下穿长周期 MA，形成做空信号，卖出开仓
    #*********************************************#
    import talib
    import pandas as pd
    import numpy as np

    ### 参数初始化
    universe = ['RBM0']              # 策略证券池
    start = '2013-07-01'
    end   = '2017-12-31'
    refresh_rate = 1                 # 调仓周期
    freq = 'd'                       # 调仓频率：s-> 秒；m-> 分钟；d-> 日；

    ## 自动生成保证金比例： margin_rate
    margin_ratio = DataAPI.FutuGet(ticker = universe, field = ['ticker',
'tradeMarginRatio'], pandas = '1')
    margin_rate = dict(zip(margin_ratio.ticker.tolist(), [0.01*index for index
in margin_ratio.tradeMarginRatio.tolist()]))

    accounts = {
        'futures_account': AccountConfig(account_type='futures', capital_base=
10000, margin_rate=margin_rate)
    }

    ### 策略初始化函数，一般用于设置计数器、回测辅助变量等
    def initialize(context):
        pass

    ### 回测调仓逻辑，每个调仓周期运行一次，可在此函数内实现信号生产、生成调仓指令
    def handle_data(context):
        futures_account = context.get_account('futures_account')

        if main_contract_mapping_changed(context, futures_account):
```

```
            return

        symbol = context.get_symbol('RBM0')
        amount = 1

        current_long = futures_account.get_positions().get(symbol, dict()).get
('long_amount', 0)
        current_short= futures_account.get_positions().get(symbol, dict()).get
('short_amount', 0)

        history_data = context.history(symbol=symbol, attribute=['closePrice',
'openPrice', 'lowPrice', 'highPrice'], time_range=30, freq='1d')[symbol]

        MA_S = talib.MA(history_data['closePrice'].apply(float).values,
timeperiod = 5)
        MA_L = talib.MA(history_data['closePrice'].apply(float).values,
timeperiod = 10)
        if MA_S[-1] > MA_L[-1] and MA_S[-2] < MA_L[-2]:
            if current_short > 0:
                futures_account.order(symbol, current_short, 'close')
            if current_long < amount:
                futures_account.order(symbol, amount, 'open')

        if MA_S[-1] < MA_L[-1] and MA_S[-2] > MA_L[-2]:
            if current_long > 0:
                futures_account.order(symbol, -current_long, 'close')
            if current_short < amount:
                futures_account.order(symbol, -amount, 'open')


    def main_contract_mapping_changed(context, futures_account):
        '''
            处理移仓换月的情况
        '''
        if context.mapping_changed('RBM0'):
            symbol_before, symbol_after = context.get_rolling_tuple('RBM0')
            if futures_account.get_position(symbol_before):
                futures_account.switch_position(symbol_before, symbol_after)
                return True

    return False
```

回测结果如图 6-6 所示。

图 6-6

可以看到，收益曲线的波动非常大，高达 33.8%，最大回撤也高达 47.3%，这与 CTA 策略最主要的特征业绩稳定相违背，原因主要有：信号变化过于频繁；没有在策略中设置止盈止损。那么如何对此策略进行改良呢？首先，我们加入止盈止损条件，如下所述。

2. 双均线突破——附带止盈止损

利用短期均线 *MA_S* 和长期均线 *MA_L* 生成开平仓信号。

◎　*MA_S* 上穿 *MA_L*，形成做多信号，买入开仓。

◎　*MA_S* 上穿 *MA_L*，形成做空信号，卖出开仓。

基于 Bar 线的止盈：指浮动赢利/保证金>5%，计算为浮动赢利×保证金比例/保证金。

基于 Bar 线的止损：指浮动亏损/保证金<3%，计算为浮动亏损×保证金比例/保证金。

策略代码如下：

```
#**********************************************#
# 双均线突破策略
#**********************************************#
```

```
    import re
    import talib
    import pandas as pd
    import numpy as np

    ### 参数初始化
    universe = ['RBM0']              # 策略证券池
    start = '2013-07-01'
    end   = '2017-12-31'
    refresh_rate = 1                 # 调仓周期
    freq = 'd'                       # 调仓频率：s-> 秒；m-> 分钟；d-> 日；

    ## 自动生成保证金比例： margin_rate
    margin_ratio = DataAPI.FutuGet(ticker = universe, field = ['ticker',
'tradeMarginRatio'], pandas = '1')
    margin_rate = dict(zip(margin_ratio.ticker.tolist(), [0.01*index for index
in margin_ratio.tradeMarginRatio.tolist()]))

    accounts = {
        'futures_account': AccountConfig(account_type='futures', capital_base=
10000, margin_rate=margin_rate)
    }

    ### 策略初始化函数，一般用于设置计数器、回测辅助变量等
    def initialize(context):
        pass

    ### 回测调仓逻辑，在每个调仓周期运行一次，可在此函数内实现信号生产、生成调仓指令
    def handle_data(context):
        futures_account = context.get_account('futures_account')

        if main_contract_mapping_changed(context, futures_account):
            return

        symbol = context.get_symbol('RBM0')
        amount = 1

        symbol = context.get_symbol(universe[0])
        current_long = futures_account.get_positions().get(symbol, dict()).get
('long_amount', 0)
        current_short= futures_account.get_positions().get(symbol, dict()).get
```

```
('short_amount', 0)

        history_data = context.history(symbol=symbol, attribute=['closePrice',
'openPrice', 'lowPrice', 'highPrice'], time_range=30, freq='1d')[symbol]

        ## 计算 MA_S 和 MS_L
        MA_S = talib.MA(history_data['closePrice'].apply(float).values,
timeperiod = 5)
        MA_L = talib.MA(history_data['closePrice'].apply(float).values,
timeperiod = 10)

        if MA_S[-1] > MA_L[-1] and MA_S[-2] < MA_L[-2]:
            if current_short > 0:
                print context.current_date, '买入平仓'
                futures_account.order(symbol, current_short, 'close')
            if current_long < amount:
                print context.current_date, '买入开仓'
                futures_account.order(symbol, amount, 'open')

        if MA_S[-1] < MA_L[-1] and MA_S[-2] > MA_L[-2]:
            if current_long > 0:
                print context.current_date, '卖出平仓'
                futures_account.order(symbol, -current_long, 'close')
            if current_short < amount:
                print context.current_date, '卖出开仓'
                futures_account.order(symbol, -amount, 'open')


        profit=futures_account.get_positions().get(symbol,dict()).get
('profit', 0)
        margin=futures_account.get_positions().get(symbol,dict()).get('long_
margin',0)-futures_account.get_positions().get(symbol,dict()).get('short_mar
gin', 0)

        if margin and profit/margin < -0.03:
            if current_long > 0:
                futures_account.order(symbol, -current_long, 'close')
            if current_short > 0:
                futures_account.order(symbol, current_short, 'close')
            print context.current_date, '止损'
        if margin and profit/margin > 0.05:
```

```
        if current_long > 0:
            futures_account.order(symbol, -current_long, 'close')
        if current_short > 0:
            futures_account.order(symbol, current_short, 'close')
        print context.current_date, '止盈'


def main_contract_mapping_changed(context, futures_account):
    '''
        处理移仓换月的情况
    '''
    if context.mapping_changed('RBM0'):
        symbol_before, symbol_after = context.get_rolling_tuple('RBM0')
        if futures_account.get_position(symbol_before):
            futures_account.switch_position(symbol_before, symbol_after)
            return True

    return False
```

回测结果如图 6-7 所示。

图 6-7

虽然策略的整体收益率下降了，但是收益波动率及最大回撤均得到了明显改善，然而这离我们的理想状态还有一定差距，那么减少买卖信号是否能让策略表现得更为稳健呢？

我们不妨试试布林带突破策略，在两条均线之间设置一个缓冲区，在缓冲区内不触发信号，只有在穿过缓冲区后才会触发信号。

3. 布林带突破——无止盈止损

利用前收盘价格 $pre_{closeprice}$ 对于布林带上边带 $upper_{band}$ 和下边带 $lower_{band}$ 突破生成开平仓信号：

- $pre_{closeprice}$ 上穿 $upper_{band}$，形成做多信号，买入开仓；
- $pre_{closeprice}$ 下穿 $lower_{band}$，形成做空信号，卖出开仓。

策略代码如下：

```
    #*********************************************#
    # 布林带突破 —— 无止盈止损
    # 价格上穿布林上边带，形成做多信号，买入开仓
    # 价格下穿布林下边带，形成做空信号，卖出开仓
    #*********************************************#
    import talib
    import pandas as pd
    import numpy as np

    ### 参数初始化
    universe = ['RBM0']              # 策略证券池
    start = '2013-07-01'
    end   = '2017-12-31'
    refresh_rate = 1                 # 调仓周期
    freq = 'd'                       # 调仓频率：s-> 秒；m-> 分钟；d-> 日；

    ## 自动生成保证金比例： margin_rate
    margin_ratio = DataAPI.FutuGet(ticker = universe, field = ['ticker',
'tradeMarginRatio'], pandas = '1')
    margin_rate = dict(zip(margin_ratio.ticker.tolist(), [0.01*index for index
in margin_ratio.tradeMarginRatio.tolist()]))

    accounts = {
        'futures_account': AccountConfig(account_type='futures', capital_base=
1000000, margin_rate=0.2)
    }
```

```
    ### 策略初始化函数，一般用于设置计数器、回测辅助变量等
    def initialize(context):
        pass

    ### 回测调仓逻辑，在每个调仓周期运行一次，可在此函数内实现信号生产、生成调仓指令
    def handle_data(context):
        futures_account = context.get_account('futures_account')

        if main_contract_mapping_changed(context, futures_account):
            return

        symbol = context.get_symbol('RBM0')
        amount = 100

        history_data = context.history(symbol=symbol, attribute=['closePrice',
'openPrice', 'lowPrice', 'highPrice'], time_range=30, freq='1d')[symbol]
        upper_band,middle_band,lower_band=talib.BBANDS(history_data
['closePrice'].apply(float).values, timeperiod = 10, nbdevup = 0.5, nbdevdn =
0.5)
        pre_close_price = history_data['closePrice'][-1]


        current_long=futures_account.get_positions().get(symbol,dict()).get
('long_amount', 0)
        current_short=futures_account.get_positions().get(symbol,dict()).get
('short_amount', 0)

        if pre_close_price > upper_band[-1]:
            if current_short > 0:
                print context.current_date, '买入平仓'
                futures_account.order(symbol, current_short, 'close')
            if current_long < amount:
                print context.current_date, '买入开仓'
                futures_account.order(symbol, amount, 'open')

        if pre_close_price < lower_band[-1]:
            if current_long > 0:
                print context.current_date, '卖出平仓'
                futures_account.order(symbol, -current_long, 'close')
            if current_short < amount:
```

```
            print context.current_date, '卖出开仓'
            futures_account.order(symbol, -amount, 'open')

def main_contract_mapping_changed(context, futures_account):
    '''
        处理移仓换月的情况
    '''
    if context.mapping_changed('RBM0'):
        symbol_before, symbol_after = context.get_rolling_tuple('RBM0')
        if futures_account.get_position(symbol_before):
            futures_account.switch_position(symbol_before, symbol_after)
            return True

return False
```

回测结果如图 6-8 所示。

图 6-8

我们可以发现，该策略在 2017 年之前的表现可谓穿越牛熊，但是在 2017 年出现了较大的回撤，这一点也可以理解。随着监管力度的不断加大，策略标的的波动幅度越来越小，这也使得 CTA 策略难以获得收益，在来回震荡中不断止损回撤。

4. 高低点突破策略

高低点突破策略是指利用价格对某动态高点或低点的突破产生交易信号。以峰值高低点突破策略为例，什么是峰值的高低点呢，定义如下。

◎ 峰值高点：以一段时间序列的正中间为最高值，该最高值为有效的峰值高点。

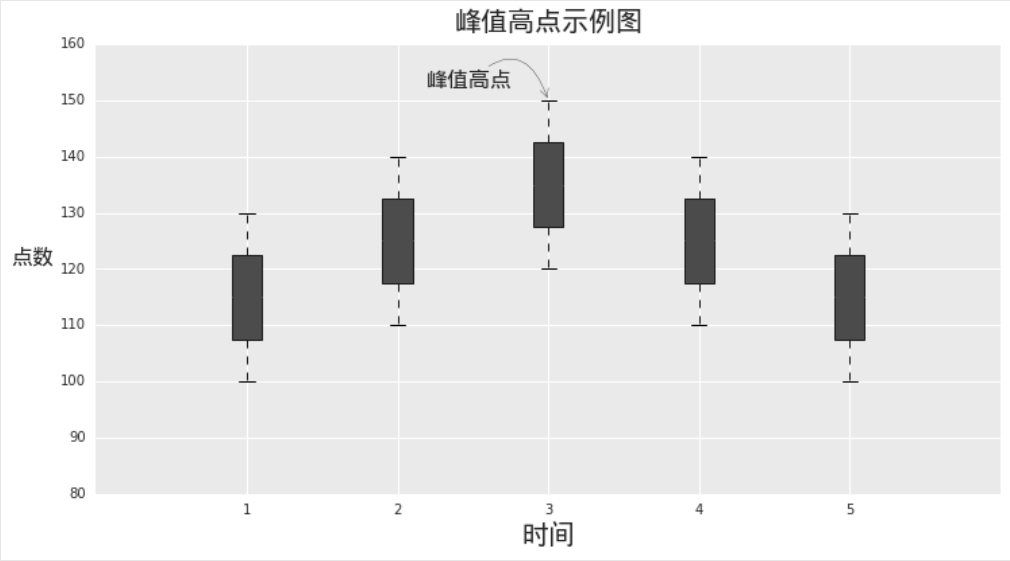
◎ 峰值低点：以一段时间序列的正中间为最低值，该最低值为有效的峰值低点。

代码如下：

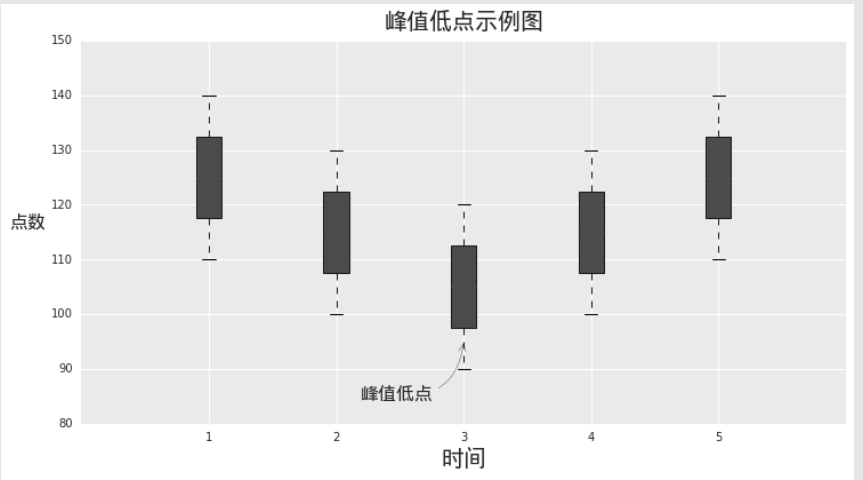
```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdate
from matplotlib.font_manager import FontProperties
from matplotlib.ticker import MultipleLocator,FormatStrFormatter
from CAL.PyCAL import *
cal = Calendar('China.SSE')

def box_plot(boxes, **args):
    '''
    Display self-definited box plot.

    Return a matplotlib figure exhibiting box plot of appointed boxes.

    Parameters
    -------
    boxes: dict-like
      {
          'boxes': array of instance of array-like.
          'color': array of basestring instance.
      }

    xlim: tuple-like, optional
      Set the limitations of x-axis.
    ylim: tuple-like, optional
      Set the limitations of y-axis.
    title: string, optional
      Set the title of the plot.
    xlabel: string, optional
      Set the label of x-axis.
```

```
        ylabel: string, optional
          Set the label of y-axis.

        Returns
        -------
        fig: matplotlib.figure.Figure
          Figure instance revelant on the portfolio value and benchmark

        '''
        fontsize, titlesize, labelsize, legendsize, linewidth = 20, 20, 16, 12,
5
        titleFont = font.copy()
        titleFont.set_size(titlesize)
        labelFont = font.copy()
        labelFont.set_size(labelsize)
        legendFont = font.copy()
        legendFont.set_size(legendsize)

        fig = plt.figure(figsize = (12,6))
        ax = fig.add_subplot(111)
        box_plot = ax.boxplot(boxes.get('boxes'), widths = 0.2,
patch_artist=True)

        plt.grid(True, axis = 'y')
        title = args.get('title') if args.get('title') else u'箱图示例'
        xlabel = args.get('xlabel') if args.get('xlabel') else u'x 轴'
        ylabel = args.get('ylabel') if args.get('ylabel') else u'y 轴'
        plt.title(title,fontsize = fontsize,verticalalignment =
'bottom',horizontalalignment = 'center',fontproperties = titleFont)
        plt.xlabel(xlabel,fontsize=fontsize,verticalalignment = 'top',
horizontalalignment = 'center',fontproperties = labelFont)
        plt.ylabel(ylabel, fontsize = fontsize, verticalalignment = 'bottom',
horizontalalignment = 'right',rotation=0, fontproperties = labelFont)
        if args.get('xlim'):
            plt.xlim(args.get('xlim')[0], args.get('xlim')[1])
        if args.get('ylim'):
            plt.ylim(args.get('ylim')[0], args.get('ylim')[1])
        for index,box in enumerate(box_plot['boxes']):
            box.set(facecolor = boxes.get('colors')[index])
            box_plot['medians'][index].set(color = boxes.get('colors')[index])
        return fig
```

```
    boxes = {
        'boxes':[[100,110,120,130],[110,120,130,140],[120,130,140,150],
        [110,120,130,140],[100,110,120,130]],
        'colors':['r']*3 + ['g']*2
    }
    box_plot(boxes, xlim = (0,6),ylim = (80,160), title = u'峰值高点示例图',xlabel
= u'时间', ylabel = u'点数')
    plt.annotate(u'峰值高点', xy=(3, 150),xycoords='data', xytext=(-90,
10),textcoords='offset points', fontsize=16,arrowprops=dict(arrowstyle="->",
connectionstyle="arc3,rad=-0.8"),fontproperties = font.copy())
    plt.show()
```




```
    boxes = {
    'boxes':[[110,120,130,140], [100,110,120,130], [90,100,110,120],
    [100,110,120,130],[110,120,130,140]],
    'colors':['g']*3 + ['r']*2
    }
    box_plot(boxes, xlim = (0,6),ylim = (80,150), title = u'峰值低点示例图',xlabel
= u'时间', ylabel = u'点数')
    plt.annotate(u'峰值低点', xy=(3, 95),xycoords='data', xytext=(-90,
-50),textcoords='offset points', fontsize=16,arrowprops=dict(arrowstyle="->",
connectionstyle="arc3,rad=.5"),fontproperties = font.copy())
    plt.show()
```



所以，高低点突破策略的具体逻辑如下。

◎　前收盘价格 $pre_{closeprice}$ 向上突破动态峰值高点，形成做多信号，买入开仓。

◎　前收盘价格 $pre_{closeprice}$ 向下突破动态峰值低点，形成做空信号，卖出开仓。

策略代码如下：

```
#**********************************************#
# 峰值高低点突破 —— 无止盈止损
# 价格上穿峰值高点，形成做多信号，买入开仓
# 价格下穿峰值高点，形成做空信号，卖出开仓
#**********************************************#
import talib
import pandas as pd
import numpy as np

### 参数初始化
universe = ['RB1610']                # 策略证券池
start = pd.datetime(2016, 6, 1)    # 回测开始时间
end   = pd.datetime(2016, 9, 1)    # 回测结束时间
capital_base = 1e5                  # 初试可用资金
refresh_rate = 1                  # 调仓周期
freq = 'd'                          # 调仓频率：s-> 秒；m-> 分钟；d-> 日；

## 自动生成保证金比例： margin_rate
margin_ratio = DataAPI.FutuGet(ticker = universe, field =
```

```
['ticker','tradeMarginRatio'], pandas = '1')
    margin_rate = dict(zip(margin_ratio.ticker.tolist(), [0.01*index for index
in margin_ratio.tradeMarginRatio.tolist()]))
    accounts = {
        'futures_account': AccountConfig(account_type='futures',
                                    capital_base=capital_base,
margin_rate=margin_rate)
    }

    ### 策略初始化函数，一般用于设置计数器、回测辅助变量等
    def initialize(context):
        context.high_point = None
        context.low_point = None
        pass

    ### 回测调仓逻辑，在每个调仓周期运行一次，可在此函数内实现信号生产、生成调仓指令
    def handle_data(context):
        futures_account = context.get_account('futures_account')
        symbol, amount = universe[0], 1
        history_data = context.history(symbol = symbol, attribute=['highPrice',
'lowPrice', 'closePrice'], time_range = 5)[symbol]
        pre_close_price = history_data['closePrice'].tolist()[-1]
        if history_data.highPrice.tolist().index(history_data.highPrice.max())
== int(len(history_data)/2):
            context.high_point = history_data.highPrice.max()
        if history_data.lowPrice.tolist().index(history_data.lowPrice.min())
== int(len(history_data)/2):
            context.low_point = history_data.highPrice.min()

        current_long = futures_account.get_positions().get(symbol, dict()).get
('long_amount', 0)
        current_short = futures_account.get_positions().get(symbol, dict()).
get('short_amount', 0)
        if pre_close_price > context.high_point:
            if current_short > 0:
                print context.current_date, '买入平仓'
                futures_account.order(symbol, current_short, 'close')
            if current_long < amount:
                print context.current_date, '买入开仓'
                futures_account.order(symbol, amount, 'open')
```

```
        if pre_close_price < context.low_point:
            if current_long > 0:
                print context.current_date, '卖出平仓'
                futures_account.order(symbol, -current_long, 'close')
            if current_short < amount:
                print context.current_date, '卖出开仓'
                futures_account.order(symbol, -amount, 'open')
```

回测结果如图 6-9 所示。

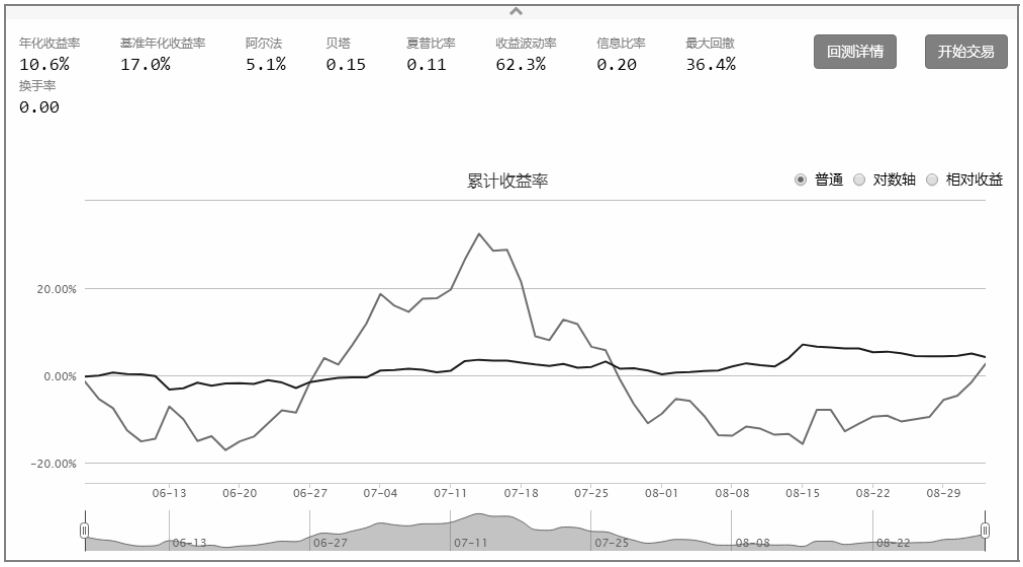


图 6-9

从结果上来看，该策略对标的趋势的抓取较为精准，但是收益的波动率过大，有一定的改善空间。

本节之前所阐述的策略都是日频 CTA 策略，接下来我们看看日内 CTA 策略。

R-Breaker 是一种短线日内交易策略，结合了趋势和反转这两种交易方式。该策略长期被 Future Thruth 杂志评为最赚钱的策略之一，尤其在标普 500 股指期货上效果最佳。该策略的主要特点如下。

（1）根据前一个交易日的收盘价、最高价和最低价数据，通过一定方式计算出 6 个价位，从大到小依次为突破买入价、观察卖出价、反转卖出价、反转买入价、观察买入价和突破卖出价，以此来形成在当前交易日盘中交易的触发条件。通过对计算方式的调整，可

以调节 6 个价格间的距离，进一步改变触发条件。

（2）根据盘中价格走势，实时判断触发条件，具体条件如下。

◎ 在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、开仓）做空。

◎ 在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、开仓）做多。

◎ 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。

◎ 在空仓的情况下，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。

（3）设定止损及止盈条件。

（4）设定过滤条件。

（5）在每日收盘前对所持合约进行平仓。

具体来看，这 6 个价位形成的阻力和支撑位的计算过程如下。

$$观察卖出价 = High + 0.35 \times (Close - Low)$$

$$观察买入价 = Low - 0.35 \times (High - Close)$$

$$反转卖出价 = 1.07 / 2 \times (High + Low) - 0.07 \times Low$$

$$反转买入价 = 1.07 / 2 \times (High + Low) - 0.07 \times High$$

$$突破买入价 = 观察卖出价 + 0.25 \times (观察卖出价 - 观察买入价)$$

$$突破卖出价 = 观察买入价 - 0.25 \times (观察卖出价 - 观察买入价)$$

其中，High、Close 和 Low 分别为昨日最高价、昨日收盘价和昨日最低价。这 6 个价位从大到小依次是：突破买入价、观察卖出价、反转卖出价、反转买入价、观察买入价和突破卖出价。

策略代码如下。首先，引入相关的库：

```
import DataAPI
import numpy as np
```

```
    import pandas as pd
    import talib as ta
    from collections import deque
    from itertools import product
    import datetime
    import matplotlib
    import matplotlib.pyplot as plt
    import matplotlib.cm as cm
    import seaborn as sns
    from CAL import *
    cal = Calendar('China.SSE')
```

策略相关的代码如下：

```
    ### 策略初始化函数
    universe = ['RBM0']              # 策略交易的期货合约，此处选择 IH1609
    start = "2017-01-02"             # 回测开始时间
    end   = "2017-12-31"             # 回测结束时间
    capital_base = 1e6               # 初始可用资金
    refresh_rate = (1, 5)            # 算法调用周期
    freq = 'm'                       # 算法调用频率：m-> 分钟；d-> 日；
    commission = {'RB': (0.000025, 'perValue')}
    slippage = Slippage(0, 'perValue')
    amount = 20
    accounts = {
        'futures_account': AccountConfig(account_type='futures',
capital_base=capital_base, commission=commission, slippage=slippage)
    }
    def initialize(context):
        context.pipe_length = 2
        context.refresh_rate = refresh_rate[1]
        context.keys = [u'openPrice', u'highPrice', u'lowPrice', u'closePrice',
u'turnoverVol', u'tradeDate']
        context.data = {key:deque([], maxlen=context.pipe_length) for key in
context.keys}
        context.high = np.NAN
        context.low = np.NAN
        context.close = np.NAN
        context.count1 = 0
        context.count2 = 0
    def handle_data(context):
        futures_account = context.get_account('futures_account')
```

```
        symbol = context.get_symbol(universe[0])
        long_position = futures_account.get_positions().get(symbol,
dict()).get('long_amount', 0)
        short_position = futures_account.get_positions().get(symbol,
dict()).get('short_amount', 0)
        if context.current_minute == '09:30':
            yester_data = DataAPI.MktFutdGet(tradeDate=context.previous_date,
ticker=symbol, field=[u'closePrice', u'highestPrice',u'lowestPrice'],
pandas="1")
            context.high = yester_data['highestPrice'].iat[0]
            context.low = yester_data['lowestPrice'].iat[0]
            context.close = yester_data['closePrice'].iat[0]

        if context.current_minute > '09:30' and context.current_minute <
'14:55':
            # ========================================
            if len(context.data['openPrice']) < context.pipe_length:
                hist = context.history(symbol=symbol, attribute=context.keys
[:-1], time_range=context.refresh_rate*context.pipe_length, freq='1m')[symbol]
            else:
                hist = context.history(symbol=symbol, attribute=context.
keys[:-1], time_range=context.refresh_rate, freq='1m')[symbol]
            current_data = {key:[] for key in context.keys}
            for i in range(len(hist)/context.refresh_rate):
                current_bar = hist[context.refresh_rate*i:context.refresh_rate*
(i+1)]
                current_bar['tradeDate'] = [i[:10] for i in current_bar.index]
                current_data['closePrice'].append(current_bar.ix[len
(current_bar)-1, 'closePrice'])
                current_data['openPrice'].append(current_bar.ix[0,
'openPrice'])
                current_data['highPrice'].append(current_bar['highPrice'].
max())
                current_data['lowPrice'].append(current_bar['lowPrice'].min())
                current_data['turnoverVol'].append(current_bar['turnoverVol'].
sum())
                current_data['tradeDate'].append(current_bar.ix[len
(current_bar)-1, 'tradeDate'])
            for i in context.keys:
                for j in current_data[i]:
                    context.data[i].append(j)
```

```
# ====================================
data = context.data
before_2_close = data['closePrice'][-2]
before_2_high = data['highPrice'][-2]
before_1_close = data['closePrice'][-1]
before_1_high = data['highPrice'][-1]
before_1_low = data['lowPrice'][-1]
before_1_open = data['openPrice'][-1]
# 观察卖出价
ssetup=context.high+0.35*(context.close-context.low)
# 观察买入价
bsetup=context.low-0.35*(context.high-context.close)
# 反转卖出价
senter=(1+0.07)/2*(context.high+context.low)-0.07*context.low
# 反转买入价
benter = (1+0.07)/2*(context.high+context.low)-0.07*context.high
# 突破买入价
bbreak = ssetup+0.25*(ssetup-bsetup)
# 突破卖出价
sbreak = bsetup-0.25*(ssetup-bsetup)

## 趋势
if before_2_close <= bbreak and before_1_close > bbreak:
    if long_position == 0:
        futures_account.order(symbol, amount, 'open')
    if short_position != 0:
        futures_account.order(symbol, short_position, 'close')
if before_2_close >= sbreak and before_1_close < sbreak:
    if short_position == 0:
        futures_account.order(symbol, -amount, 'open')
    if long_position != 0:
        futures_account.order(symbol, -long_position, 'close')
## 反转
### 多单反转
if before_1_high > ssetup and before_1_close > senter:
    context.count1 = 1
if context.count1 == 1 and before_1_close < senter:
    if long_position > 0:
        futures_account.order(symbol, -long_position, 'close')
        futures_account.order(symbol, -amount, 'open')
### 空单反转
```

```
            if before_1_low < bsetup:
                context.count2 = 1
            if context.count2 == 1 and before_1_close > benter:
                if short_position != 0:
                    futures_account.order(symbol, short_position, 'close')
                    futures_account.order(symbol, amount, 'open')

        elif context.current_minute >= '14:55':
            if short_position > 0:
                futures_account.order(symbol, short_position, 'close')
            if long_position > 0:
                futures_account.order(symbol, -long_position, 'close')
            context.high = np.NAN
            context.low = np.NAN
            context.close = np.NAN
            context.count1 = 0
            context.count2 = 0
```

这里，由于分钟级别的回测比较慢，所以我们选择对 2010 年这一整年进行回测，如图 6-10 所示。

图 6-10

可以看到，策略表现得十分稳健，最大回撤仅为 2.5%，beta 为-0.01，基本上和标的

的走势无关，即使如此，也取得了 7.5%的 Alpha。试想，如果在实盘中策略表现得依旧如此，则完全可以通过加大杠杆来获取更高的收益，的确是最赚钱的策略之一。

6.5.2　均值回复策略

前面已经介绍过均值回复策略，可将其主要分为跨品种套利及跨期套利。

1. 跨品种套利

跨品种套利所选择的品种从经济角度来说，可能处于同一产业链的上下游，比如玉米与淀粉；也有可能具有可替代或者互补的关系，比如豆油与菜油。一般来说，同一产业链的品种相关性较高，无论是从基本面还是从统计规律出发，均比较容易找到逻辑支撑。常见的产业链主要包括黑色产业链、豆类油脂产业链和化工产业链等。通过对同一产业链中相关性很强的品种的期货合约分别进行买入和卖出，通过品种间的强弱变化引发价差的收缩与扩大，可实现价差收益。“虚拟钢厂”的构建正是利用螺纹钢和其主要原材料铁矿石、焦炭及三种上下游品种之间的价格的不平衡变化而产生的价差收益，是一种跨品种套利的策略。

我们先来看看这三个品种的历史走势：

```
    import numpy as np
    import matplotlib.pyplot as plt
    import seaborn as sns
    import pandas as pd
    import talib
    from collections import deque
    from CAL.PyCAL import font
    import DataAPI
    import scipy.stats as st
    rbm0 = DataAPI.MktMFutdGet(mainCon=u"1", contractObject=u"RB",startDate=
u"2013-11-01",endDate=u"2016-06-01",field=[u'closePrice',
'tradeDate'],pandas="1")
    im0 = DataAPI.MktMFutdGet(mainCon=u"1", contractObject=u"I",startDate=
u"2013-11-01",endDate=u"2016-06-01",field=[u'closePrice',
'tradeDate'],pandas="1")
    jm0 = DataAPI.MktMFutdGet(mainCon=u"1", contractObject=u"J",startDate=
u"2013-11-01",endDate=u"2016-06-01",field=[u'closePrice',
'tradeDate'],pandas="1")
    y = np.array(rbm0['closePrice'])
```

```
    x1 = np.array(im0['closePrice'])
    x2 = np.array(jm0['closePrice'])

    fig = plt.figure(figsize=(15, 5))
    sns.set_style('whitegrid')
    ax = fig.add_subplot(111)
    ax.set_xlim(0, len(rbm0))
    ax.plot(y, color = '#000000', label='rb', lw=3)
    ax.plot(x1, 'r', label='i', lw=3)
    ax.plot(x2, 'g', label='j', lw=3)
    ax.set_xticks(range(0, len(rbm0), 60))
    ax.legend(loc=1)
    xlabel = ax.set_xticklabels([rbm0.ix[i, 'tradeDate'] for i in
ax.get_xticks()])
    title = ax.set_title(u'螺纹，铁矿石，焦炭价格历史走势图', loc=u'center',
fontproperties=font, fontsize=16)
```


```
    print '螺纹与铁矿石相关系数：', np.corrcoef(y, x1)[0][1]
    print '螺纹与焦炭相关系数：', np.corrcoef(y, x2)[0][1]
    print '铁矿石与焦炭相关系数：', np.corrcoef(x1, x2)[0][1]
    print '-----------------------------------------------'
    print '螺纹与铁矿石卡方检验 p 值:', st.chisquare(y, x1)[1]
    print '螺纹与焦炭卡方检验 p 值:',st.chisquare(y, x2)[1]
    print '铁矿石与焦炭卡方检验 p 值:',st.chisquare(x1, x2)[1]
    螺纹与铁矿石相关系数： 0.977895286856
    螺纹与焦炭相关系数： 0.964336641768
    铁矿石与焦炭相关系数： 0.948966994897
    -----------------------------------------------
    螺纹与铁矿石卡方检验 p 值： 0.0
    螺纹与焦炭卡方检验 p 值： 0.0
    铁矿石与焦炭卡方检验 p 值： 0.0
```

通过对它们的价格序列进行卡方检验发现，p 值几乎为 0，可以认为这三者之间高度正相关。实际上螺纹和铁矿石的相关系数为 0.978，螺纹和焦炭的相关系数为 0.964，铁矿石和焦炭的相关系数为 0.949。这就为在三者之间进行跨品种套利奠定了基础。

钢材生产流程大致可分为三个阶段：炼铁、炼钢和轧钢，生产成本主要包括原料成本、能源成本、人工成本、折旧和财务成本等。从原材料来看，生产 1 吨螺纹钢大致需要 1.6 吨铁矿石和 0.5 吨焦炭。因此，我们可以通过期货市场来大致模拟钢厂的生产过程，通过结合相关品种在期货市场的利润及现货市场供需基本面数据，把握品种间的套利机会。通过成本估算，当前阶段的螺纹钢成本≈1.6×铁矿石+0.5×焦炭+加工成本；利润≈螺纹钢期货价格-螺纹钢成本；炼铁、炼钢和轧钢三个阶段的总加工费大概在 1100 元左右。

如下所示是 2013 年 10 月至 2016 年 7 月虚拟钢厂的利润走势，在-200～300 的范围内波动，利润中枢在 100 左右：

```
    fig = plt.figure(figsize=(15, 5))
    sns.set_style('whitegrid')
    ax = fig.add_subplot(111)
    ax.set_xlim(0, len(rbm0))
    profit = (y - 1.6*x1 - 0.5*x2 - 1100).tolist()
    ax.plot(profit, lw=3)
    ax.plot([300]*len(profit), '--r')
    ax.plot([-200]*len(profit), '--g')
    ax.set_xticks(range(0, len(profit), 60))
    xlabel = ax.set_xticklabels([rbm0.ix[i, 'tradeDate'] for i in
ax.get_xticks()])
    title = ax.set_title(u'虚拟钢厂利润历史走势图', loc=u'center', fontproperties=
font, fontsize=16)
```

从价格走势图来看，利润走势不可能维持在高位或低位。这是因为，当利润达到极高值时，一方面钢厂会加速生产，供给增加则价格降低；另一方面铁矿石、焦炭等原材料的价格会上涨，从而压缩利润空间。当利润达到极低值时，一方面钢厂会因持续亏损而减产，产量减少将支撑价格；另一方面上游企业的议价能力更弱，原材料的价格承压走低，从而使利润逐渐回升。当盘面利润超过 300 元时，可以做空利润，即卖螺纹，买铁矿石和焦炭，当盘面利润低于 200 元时，又可以做多利润，即买螺纹，卖铁矿石和焦炭。

首先，从产业结构计算产出比，比如螺纹:铁矿:焦炭=1 吨:1.6 吨:0.5 吨；其次，按照合约规定的实际数量的 1:1 来确定，当前螺纹:铁矿:焦炭=1:10:10（螺纹期货合约为 10 吨/手，铁矿和焦炭期货合约均为 100 吨/手）；最后，确定套利比例也需要考虑各品种的波动率的不同。综合上述因素，可大致确定螺纹、铁矿、焦炭三者的手数配比为 20:3:1。初始资金设为 15 万元。

然后，选取螺纹、铁矿和焦炭主力合约。

策略逻辑如下。

- ◎ 当盘面利润高于 300 元时，做空螺纹，做多铁矿石与焦炭；当盘面利润低于 100 元时，如果螺纹有空仓，则平掉所有仓位。
- ◎ 当盘面利润低于-200 元时，做多螺纹，做空铁矿石与焦炭；当盘面利润高于 100 元时，如果螺纹有多仓，则平掉所有仓位。
- ◎ 当主力切换时，平掉所有仓位。
- ◎ 初始资金：30 万元。
- ◎ 回测时间：自铁矿石品种上市以来。

策略代码如下：

```
import re
import talib
import pandas as pd
import numpy as np

universe = ['RBM0', 'IM0', 'JM0']       # 策略期货合约
start = '2013-11-01'                    # 回测开始时间
end  = '2017-12-31'                     # 回测结束时间
# capital_base = 300000                 # 初试可用资金
refresh_rate = 1                        # 调仓周期
```

```
    freq = 'd'                                  # 调仓频率：m-> 分钟；d-> 日
    # marg
    diff = deque([], maxlen=10)
    accounts = {
        'futures_account': AccountConfig(account_type='futures',
capital_base=300000)
    }
    def initialize(context):           # 初始化虚拟期货账户，一般用于设置计数器、回测
辅助变量等
        context.symbol0 = 'RB1405'
        context.symbol1 = 'I1405'
        context.symbol2 = 'J1405'

    def handle_data(context):          # 回测调仓逻辑，在每个调仓周期运行一次，可在此
函数内实现信号生产、生成调仓指令
        futures_account = context.get_account('futures_account')
        long_position_0 = futures_account.get_positions().get(context.symbol0,
dict()).get('long_ amount', 0)
        short_position_0 = futures_account.get_positions().get(context.symbol0,
dict()).get('short_ amount', 0)

        long_position_1 = futures_account.get_positions().get(context.symbol1,
dict()).get('long_ amount', 0)
        short_position_1 = futures_account.get_positions().get(context.symbol1,
dict()).get('short_ amount', 0)

        long_position_2 = futures_account.get_positions().get(context.symbol2,
dict()).get('long_amount', 0)
        short_position_2 = futures_account.get_positions().get(context.symbol2,
dict()).get('short_ amount', 0)
        if context.get_symbol(universe[0]) != context.symbol0 or
context.get_symbol(universe[1]) != context.symbol1 or
context.get_symbol(universe[2]) != context.symbol2:
            if long_position_0 != 0:
                # print futures_account.current_date, '主力更换， 平仓'
                futures_account.order(context.symbol0, -long_position_0,
'close')
                futures_account.order(context.symbol1, short_position_1,
'close')
                futures_account.order(context.symbol2, short_position_2,
'close')
```

```
            if short_position_0 != 0:
                # print futures_account.current_date, '主力更换， 平仓'
                futures_account.order(context.symbol0,short_position_0,
'close')
                futures_account.order(context.symbol1,-long_position_1,
'close')
                futures_account.order(context.symbol2,-long_position_2,
'close')

            context.symbol0 = context.get_symbol(universe[0])
            context.symbol1 = context.get_symbol(universe[1])
            context.symbol2 = context.get_symbol(universe[2])
        else:
            context.symbol0 = context.get_symbol(universe[0])
            context.symbol1 = context.get_symbol(universe[1])
            context.symbol2 = context.get_symbol(universe[2])
            close_hist0 = context.history(symbol=context.symbol0, field=
['closePrice'], time_range=1)
            close_hist1 = context.history(symbol=context.symbol1, field=
['closePrice'], time_range=1)
            close_hist2 = context.history(symbol=context.symbol2, field=
['closePrice'], time_range=1)

            close0 = np.array(close_hist0[context.symbol0]['closePrice'])[-1]
            close1 = np.array(close_hist1[context.symbol1]['closePrice'])[-1]
            close2 = np.array(close_hist2[context.symbol2]['closePrice'])[-1]

            if short_position_0  == 0 and close0 - 1.6 * close1 - 0.5 * close2
- 1100 > 300:
                # print futures_account.current_date,  '做空螺纹'
                # print close0 - 1.6 * close1 - 0.5 * close2 - 1100
                futures_account.order(context.symbol0, -20, 'open')
                futures_account.order(context.symbol1, 3, 'open')
                futures_account.order(context.symbol2, 1, 'open')

            if long_position_0  == 0 and close0 - 1.6 * close1 - 0.5 * close2 -
1100 < -250:
                # print futures_account.current_date,  '做多螺纹'
                # print close0 - 1.6 * close1 - 0.5 * close2 - 1100
                futures_account.order(context.symbol0, 20, 'open')
```

```
            futures_account.order(context.symbol1, -3, 'open')
            futures_account.order(context.symbol2, -1, 'open')

        # 平仓
        if close0 - 1.6 * close1 - 0.5 * close2 - 1100 < 100:
            if short_position_0 != 0:
                # print futures_account.current_date, '平仓'
                # print close0 - 1.6 * close1 - 0.5 * close2 - 1100
                futures_account.order(context.symbol0,short_position_0,
'close')
            if long_position_1 != 0:
                futures_account.order(context.symbol1,-long_position_1,
'close')
            if long_position_2 != 0:
                futures_account.order(context.symbol2,-long_position_2,
'close')
        if close0 - 1.6 * close1 - 0.5 * close2 - 1100 > 100:
            if long_position_0 != 0:
                # print futures_account.current_date, '平仓'
                # print close0 - 1.6 * close1 - 0.5 * close2 - 1100
                futures_account.order(context.symbol0,short_position_0,
'close')
            if short_position_1 != 0:
                futures_account.order(context.symbol1,-long_position_1,
'close')
            if short_position_2 != 0:
                futures_account.order(context.symbol2,-long_position_2,
'close')
```

回测结果如图 6-11 所示。

该策略还有以下特点。

（1）虚拟钢厂套利策略是基于基本面逻辑的套利策略，如果基本面发生重大变化，则应及时止损止盈。

（2）可以看到，策略的本质其实是一个固定了上下轨道的突破策略，其上下轨的确定是根据模拟钢厂生产的利润传导机制确定的。

（3）在当前低迷的市场情况下，钢企合理利用期货市场进行套期保值、管控风险，具有现实意义。

（4）交易次数较少。

同样，该策略在 2017 年遭遇了一定的回撤，在国内商品市场的四大板块中，贵金属、农产品、工业品均维持了近一年的无趋势行情，有色金属也仅有微弱的趋势。另一方面，我们的定价方程是基于基本面数据的，随着时间的推移，基本面数据发生改变，定价方程也应该随着改变，否则会造成大幅回撤。

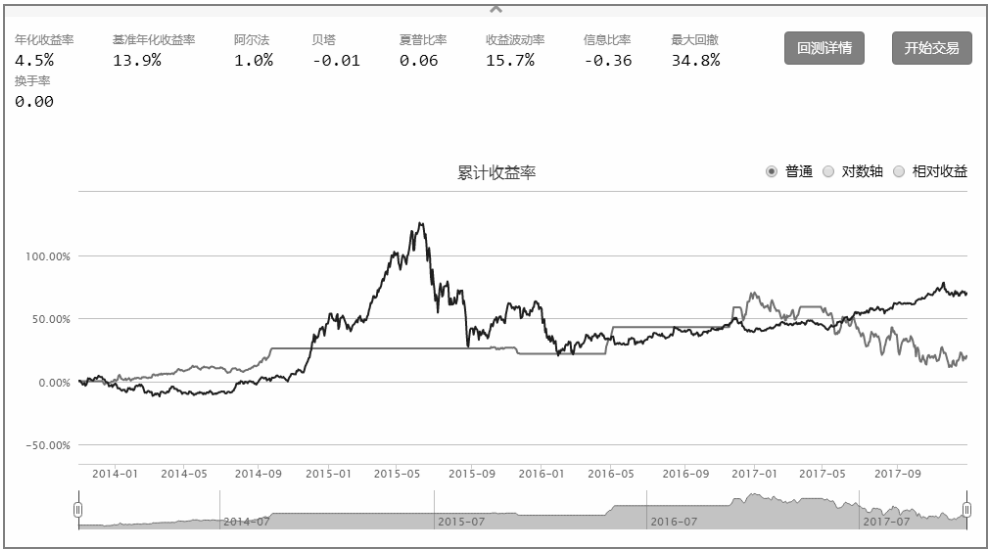


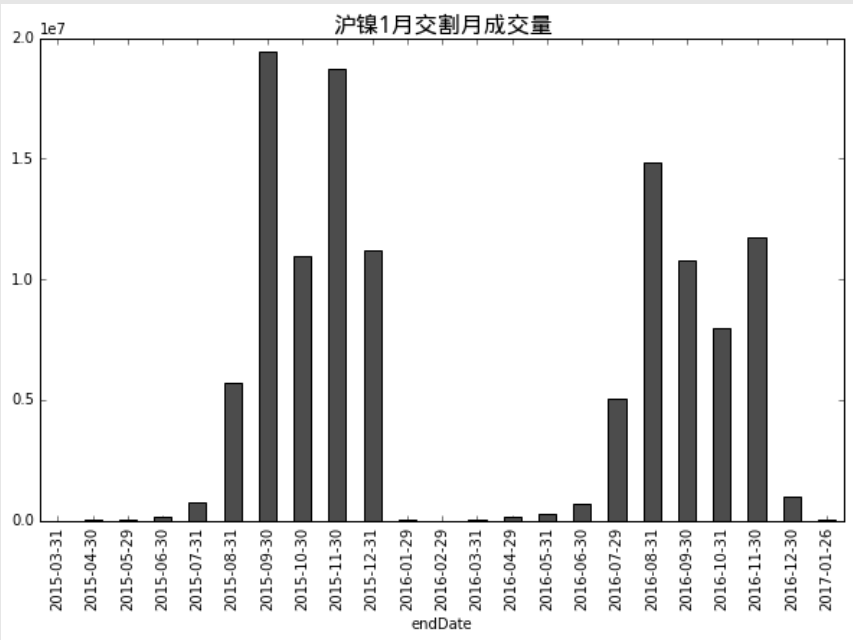
图 6-11

2. 跨期套利

与跨品种套利不同，跨期套利是在同一市场、同一品种、不同的到期月份的合约之间进行套利，其中涉及非主力合约的问题。非主力合约的成交量比较小，有的甚至低到几十元数百手，很容易被几手交易量将价格拉至异常。这种异常实际上较难捕捉，只会放大回测结果。我们以沪镍为例，其主力合约主要在 1、5、9 月交割，其余月份的成交量很低。而 1 月交割合约的活跃期为 7 月至 12 月，5 月交割合约的活跃期为 11 月至来年 4 月，所以在构建 1 月交割合约和 5 月交割合约的套利组合时，我们选择 11、12 月作为套利策略实施的时间进行测试，在 12 月底如果有持仓，则强行平仓。在这个时间段内，1 月交割合约为主力合约，5 月交割合约为次主力合约。

代码如下：

```
    from collections import deque
    import numpy as np
    import talib as ta
    import matplotlib.pyplot as plt
    from CAL.PyCAL import *
    NI1601 = DataAPI.MktFutmGet(ticker=u"NI1601",beginDate=u"",endDate=u"",
field=u"endDate,turnoverVol",pandas="1").sort_values(by='endDate')
    NI1701 = DataAPI.MktFutmGet(ticker=u"NI1701",beginDate=u"",endDate=u"",
field=u"endDate,turnoverVol",pandas="1").sort_values(by='endDate')
    NI = NI1601.append(NI1701)
    NI.drop_duplicates(subset='endDate', keep='first', inplace=True)
    NI = NI.set_index('endDate')
    fig = plt.figure(figsize=(10, 6))
    ax = fig.add_subplot(111)
    ax.set_axisbelow('True')
    ax = NI.plot(kind='bar', ax=ax, color='r', legend=False)
    s = ax.set_title(u'沪镍 1 月交割月成交量', fontproperties=font, fontsize=16)
```



在套利时，我们需要计算均衡价差，一般有两种算法。

◎ 模拟交割法：通过模拟交割过程中的各项费用，来计算套利过程中的各项成本，从而得出均衡价差。优点为容易计算，价差变化小；缺点为在实际情况下价差很

少恢复到这一水平。

◎ 历史滞后均值法：通过历史的价差选择时间窗口长度，得到均衡价差。优点为容易计算，价差是动态的；缺点为不能保证样本外价差恢复到这一水平。

我们以选定套利组合的开盘价差为对象，使用历史滞后均值法，取 *N* 日价差均值为中线，取中线加上 *N* 日价差的 *M* 倍均值为上轨，取中线减去 *N* 日价差的 *M* 倍均值为下轨来构造布林带通道，当价差突破上轨时，做空价差。我们选择 1、5、9 这三个月份交割合约，滚动地进行测试，其中 1 月至 5 月合约测试的时间为 11 月和 12 月，5 月至 9 月合约测试的时间为 3 月和 4 月，9 月至来年 1 月合约测试的时间为 7 月和 8 月，所以全年有半年不进行操作。

相关参数及设定如下。

◎ 成交价设置为当日开盘价。这里考虑到次主力合约的成交量可能较少，所以设置为 3 个滑点。

◎ 开仓信号为价差超过上轨或者突破下轨，信号触发当天就按所设定的成交价进行成交。如果平仓信号达到中线附近或者所设定的套利期限结束，则强行平仓。

◎ 手续费设为 6 元一手。

◎ 回测时间为 2015 年 5 月至 2017 年 12 月。

◎ 暂不考虑止损。

策略代码如下：

```
    universe = ['NIM0']                          # 策略期货合约
    start = '2015-05-18'                         # 回测开始时间
    end  = '2017-12-31'                          # 回测结束时间
    capital_base = 1000000                       # 初试可用资金
    refresh_rate = 1                             # 调仓周期
    freq = 'd'                                   # 调仓频率：m-> 分钟；d-> 日
    margin_rate = 0.09
    commission = {'NI': (6, 'perShare')}
    slippage = Slippage(3, 'perShare')
    accounts = {
        'futures_account':AccountConfig(account_type='futures',capital_base=
capital_base,margin_rate=margin_rate,commission=commission,slippage=slippage
)
```

```
    }
    months = ['07', '08', '11', '12', '03', '04']

    mid, up, low = [], [], []
    def initialize(context):          # 初始化虚拟期货账户，一般用于设置计数器、回测
辅助变量等
        context.mean = deque([], maxlen=10)
        context.std = deque([], maxlen=10)

        context.M0 = ''
        context.M1 = ''
    def handle_data(context):         # 回测调仓逻辑，在每个调仓周期运行一次，可在此
函数内实现信号生产、生成调仓指令
        futures_account = context.get_account('futures_account')
        today = str(context.current_date)
        if today[5:7] in ['11', '12']:
            M0 = universe[0][:2] + str(int(today[2:4])+1) + '01'
            M1 = universe[0][:2] + str(int(M0[2:4])+(int(M0[4:])+4)/12)+'0'+str
((int(M0[4:])+4)%12)
        elif today[5:7] in ['07', '08']:
            M0 = universe[0][:2] + str(int(today[2:4])) + '09'
            M1 = universe[0][:2] + str(int(M0[2:4])+(int(M0[4:])+4)/12)+'0'+str
((int(M0[4:])+4)%12)
        elif today[5:7] in ['03', '04']:
            M0 = universe[0][:2] + str(int(today[2:4])) + '05'
            M1 = universe[0][:2] + str(int(M0[2:4])+(int(M0[4:])+4)/12)+'0'+str
((int(M0[4:])+4)%12)

        long_position = futures_account.get_positions().get(context.M0,
dict()).get('long_amount', 0)
        short_position = futures_account.get_positions().get(context.M1,
dict()).get('short_aniount', 0)

        if today[5:7] not in months:
            if long_position != 0:
                futures_account.order(context.M0, -20, 'close')
                futures_account.order(context.M1, 20, 'close')
            return

        M0_open_price = context.history(symbol=M0, attribute='openPrice',
time_range=11)[M0]['openPrice']
```

```
        M1_open_price = context.history(symbol=M1, attribute='openPrice',
time_range=11)[M1]['openPrice']

        diff = np.array(M0_open_price - M1_open_price, dtype=float)
        context.mean.append(ta.MA(diff, 10)[-1])
        context.std.append(ta.STDDEV(diff, 10)[-1])

        if diff[-1] < context.mean[-1] - 1 * context.std[-1] and long_position
== 0:
            futures_account.order(M1, -20, 'open')
            futures_account.order(M0, 20, 'open')
        if diff[-1] > context.mean[-1] - 0.2 * context.std[-1]  and
long_position != 0:
            futures_account.order(M1, 20, 'close')
            futures_account.order(M0, -20, 'close')

        context.M0 = M0
        context.M1 = M1
```

回测结果如图 6-12 所示。

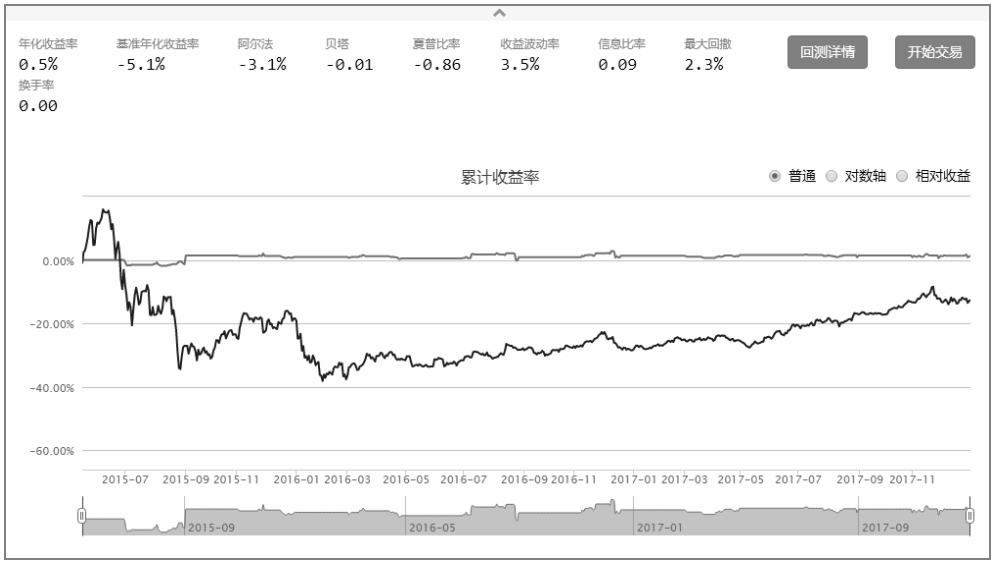


图 6-12

此处仅为一个简单的示例，通过回测发现，本策略并没有取得什么收益。一般来说，策略亏损的情况发生在如下两种情况下。

- 价差形成了长期趋势。虽然这时价差恢复到了 N 日平均线，但是与开仓点相比，其偏离程度很大。
- 价差偏差程度较小，这时均值回复的收益不足以弥补手续费和冲击成本造成的损失。

6.5.3　CTA 策略表现分析

在前两节讲到了基本的两类 CTA 策略：趋势跟随策略和均值回复策略。那么，如何评价这些策略的表现呢？这里可以运用优矿专业版提供的组合分析工具进行策略分析。

以简单的双均线突破策略为例，策略代码如下：

```
    import talib
    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt


    ### 参数初始化
    universe = ['RB1610']                # 策略证券池
    start = pd.datetime(2016, 6, 1)      # 回测开始时间
    end   = pd.datetime(2016, 9, 1)      # 回测结束时间
    capital_base = 1e4                   # 初试可用资金
    refresh_rate = 1                     # 调仓周期
    freq = 'd'                           # 调仓频率：s-> 秒；m-> 分钟；d-> 日；

    ## 自动生成保证金比例： margin_rate
    margin_ratio=DataAPI.FutuGet(ticker=universe,field=['ticker','tradeMargi
nRatio'], pandas = '1')
    margin_rate = dict(zip(margin_ratio.ticker.tolist(), [0.01*index for index
in margin_ratio.tradeMarginRatio.tolist()]))
    accounts = {
        'futures_account': AccountConfig(account_type='futures', capital_base=
capital_base, margin_rate=margin_rate)
    }

    ### 策略初始化函数，一般用于设置计数器、回测辅助变量等
    def initialize(context):
        pass
```

```
    ### 回测调仓逻辑，在每个调仓周期运行一次，可在此函数内实现信号生产、生成调仓指令
    def handle_data(context):
        futures_account = context.get_account('futures_account')
        symbol, amount = universe[0], 1
        history_data = context.history(symbol = symbol, time_range = 20)[symbol]
        MA_S=talib.MA(history_data['closePrice'].apply(float).values,
timeperiod=5)
        MA_L=talib.MA(history_data['closePrice'].apply(float).values,
timeperiod=10)

        current_long=futures_account.get_positions().get(symbol,dict()).get
('long_amount', 0)
        current_short=futures_account.get_positions().get(symbol,dict()).get
('short_amount', 0)
        if MA_S[-1] > MA_L[-1] and MA_S[-2] < MA_L[-2]:
            if current_short > 0:
                # print futures_account.current_date,futures_account.current_
time, '买入平仓'
                futures_account.order(symbol, current_short, 'close')
            if current_long < amount:
                # print futures_account.current_date, futures_account.current_
time, '买入开仓'
                futures_account.order(symbol, amount, 'open')

        if MA_S[-1] < MA_L[-1] and MA_S[-2] > MA_L[-2]:
            if current_long > 0:
                # print futures_account.current_date, futures_account.current_
time,'卖出平仓'
                futures_account.order(symbol, -current_long, 'close')
            if current_short < amount:
                # print futures_account.current_date, futures_account.current_
time, '卖出开仓'
                futures_account.order(symbol, -amount, 'open')
```

回测结果如图 6-13 所示。

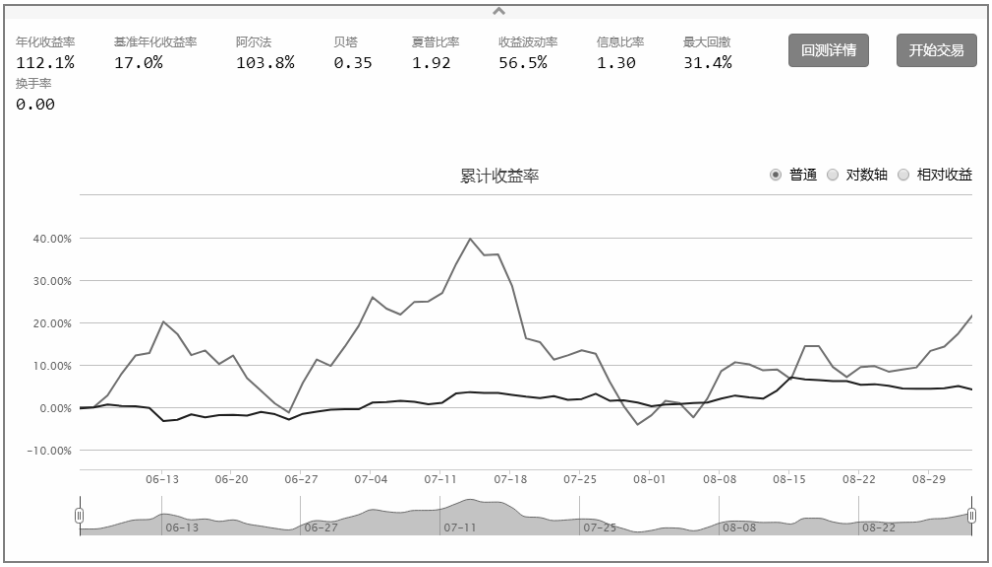


图 6-13

双均线策略的特点已经在 6.5.1 节中讲到了，在此不再重复阐述。

接下来对该策略的回撤期进行归纳，这里用到了 gen_drawdown_table 函数：

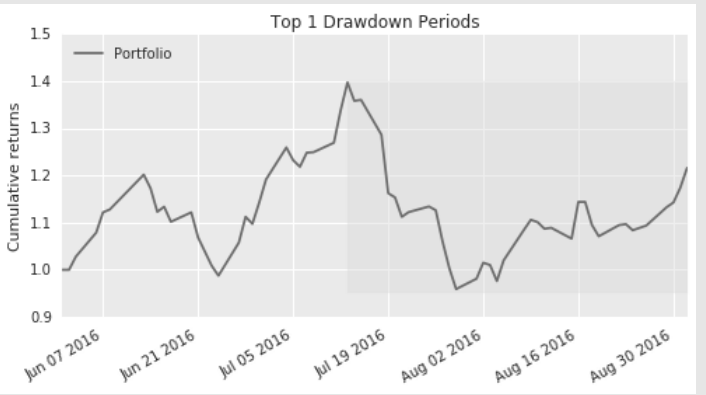
```
returns = perf['returns']
returns.index = pd.to_datetime(returns.index)
gen_drawdown_table(returns)
```

	net drawdown in %	peak date	valley date	recovery date	duration
0	31.3717	2016-07-13	2016-07-29	NaT	NaN
1	17.8365	2016-06-13	2016-06-24	2016-07-04	16
2	3.25517	2016-07-04	2016-07-06	2016-07-11	6
3	0	2016-06-01	2016-06-01	2016-06-01	1
4	0	2016-06-01	2016-06-01	2016-06-01	1
5	0	2016-06-01	2016-06-01	2016-06-01	1
6	0	2016-06-01	2016-06-01	2016-06-01	1
7	0	2016-06-01	2016-06-01	2016-06-01	1
8	0	2016-06-01	2016-06-01	2016-06-01	1
9	0	2016-06-01	2016-06-01	2016-06-01	1

在如上所示的表中给出了不同的回撤期，例如从 2016 年 7 月 4 日至 2016 年 7 月 6 日，策略发生了回撤，在 2016 年 7 月 6 日之后策略重新赢利，直到 2016 年 7 月 11 日，组合累计收益率恢复至回撤前的水平。

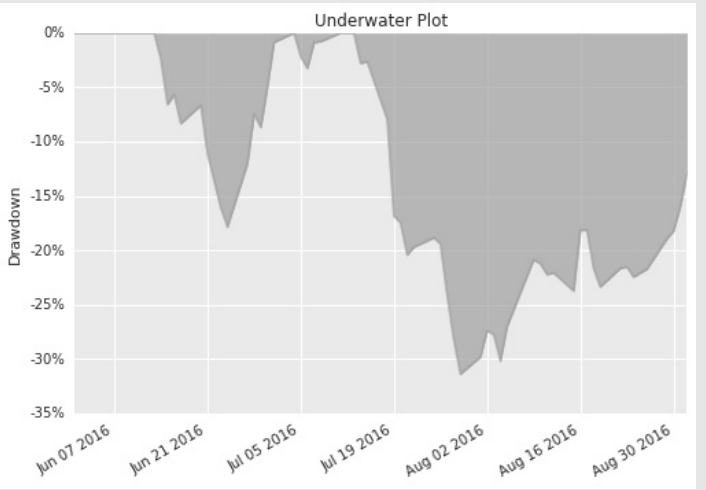
我们可以通过调用 plot_drawdown_periods 函数更加直观地对如上所示的表进行展示，其中，参数 returns 为累计收益率序列，top 用于指定需要绘制多少个回撤期（从大到小排列）：

```
fig = plt.figure(figsize=(8, 4))
ax = fig.add_subplot(111)
ax = plot_drawdown_periods(returns, top=1, ax=ax)
```



接下来通过 plot_drawdown_underwater 函数绘制当前时间节点下的策略的回撤：

```
ax = plot_drawdown_underwater(perf['returns'])
```

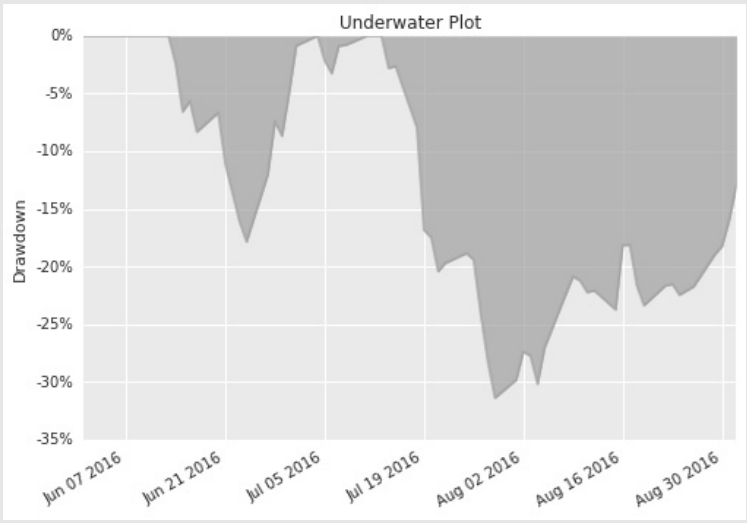


下面通过 plot_monthly_returns_heatmap 函数绘制策略每个月的收益。相对而言，对一些月度调仓的股票策略，使用该图进行分析更加有用：

```
ax = plot_monthly_returns_heatmap(returns)
```

下面通过 plot_monthly_returns_dist 函数绘制月度收益的分布图：

```
ax = plot_monthly_returns_dist(returns)
```



是不是很方便？我们还可以通过查询优矿专业版的帮助文档，来获得更多的功能。

6.6 Smart Beta

Smart Beta（聪明贝塔，又称智慧型投资策略）是指在传统的指数投资基础上，通过系统性的方法对指数中选股和权重进行优化，以跑赢传动指数投资的策略。与传统的市值加权指数相比，在传统的指数中市值越高的个股将占有越大的权重，往往将指数投资者暴露于被高估的股票及投资集中的风险中。而 Smart Beta 与传统的指数编制方法不同，它通过对传统指数选股及权重的优化，在指数化被动管理的同时，相对于传统指数也能够取得一定的超额收益。

下面分别从基于权重优化和基于风险因子的角度进行详细介绍。

6.6.1 基于权重优化的 Smart Beta

本节详细介绍从权重优化角度来看各种配置方法的理论与实践，包括等权组合、最小方差组合、风险平价组合和最大多元化组合。

1. Beta 简介

众所周知，Beta 在 CAPM 模型中衡量了相对于持有整个市场所带来的风险溢价（Risk Premium）的大小。我们通常用市场投资组合或市场指数基金来表示整个市场，市场指数通常都是市值加权（Market Capitalization Weighted），如果把市场指数换成按非市值加权的指数或投资组合，则得到的 Beta 就是 Smart Beta。下面分别从等权组合、最小方差组合、风险平价组合、最大多元化组合这 4 个角度进行权重优化。

2. 原理与算法

1）等权重

每个成分股的权重都一样，权重的计算公式如下，其中，N 为成分股的个数：

$$\boldsymbol{w}_i = \frac{1}{N}$$

2）最小方差

用于保证组合整体风险最小，可以用如下优化问题表示：

$$\begin{aligned} &\boldsymbol{w}^{*}=\min(\frac{1}{2}\omega'\boldsymbol{\Omega}\boldsymbol{\omega}) \\ &\text{s.t.}\quad \boldsymbol{w}'1=1 \\ &\boldsymbol{\omega}_i \geqslant 0 \end{aligned}$$

在上式中，$\boldsymbol{\Omega}$ 为各资产的协方差矩阵，$\boldsymbol{\omega}$ 为各资产的权重列向量。

3）风险平价

用于保证每个资产对组合整体风险的贡献都一样，可以用如下优化问题表示：

$$\begin{aligned} &\min f(\boldsymbol{\omega})=\sum_{i=1}^{N}\sum_{j=1}^{N}[\boldsymbol{\omega}_i(\boldsymbol{\Omega}\boldsymbol{\omega})_i-\boldsymbol{\omega}_j(\boldsymbol{\Omega}\boldsymbol{\omega})_j]^2 \\ &\text{s.t.}\quad \boldsymbol{w}'1=1 \\ &\boldsymbol{\omega}_i \geqslant 0 \end{aligned}$$

在上式中，$\boldsymbol{\Omega}$ 为各资产的协方差矩阵，$\boldsymbol{\omega}$ 为各资产的权重列向量。

4）最大多元化

首先假设组合投资没有风险的功能，那么组合波动率就可以像收益率一样加权求得，但在实际情况下组合投资可以降低组合风险，那么最大多元化（也就是最大分散度）可以被定义为加权波动率与真实波动率的比值，求解过程就可以转换为如下优化问题：

$$\begin{aligned} &\max D(\boldsymbol{\omega})=\frac{\boldsymbol{\omega}'\boldsymbol{\sigma}}{\sqrt{\boldsymbol{\omega}'\boldsymbol{\Omega}\boldsymbol{\omega}}} \\ &\text{s.t.}\quad \boldsymbol{w}'1=1 \\ &\boldsymbol{\omega}_i \geqslant 0 \end{aligned}$$

在上式中，$\boldsymbol{\Omega}$ 为各资产的协方差矩阵，$\boldsymbol{\omega}$ 为各资产的权重列向量，$\boldsymbol{\sigma}$ 为各资产的标准差列向量。

3. 回测

1）定义相关函数

用于定义在研究过程中使用的关于读取数据、预处理数据及上面介绍的几种投资组合权重最优化的相关函数。

定义在实测中读取相关数据的函数如下：

```
    import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt
    import seaborn as sns
    sns.set_style('white')
    from scipy.optimize import minimize
    from cvxopt import matrix, solvers
    from datetime import datetime as dt
    from CAL.PyCAL import font
    def get_covmat(tickers, date, periods):
        '''
        输入 tickers + 日期 + 过去天数，获得以此计算出来的年化协方差矩阵
        '''
        start_date = shift_date(date, periods)
        return_mat=DataAPI.MktEqudAdjGet(ticker=tickers, beginDate=start_date,
endDate=date, field=u"ticker,tradeDate,closePrice", pandas="1")
        return_mat = return_mat.pivot(index='tradeDate',columns='ticker',
values='closePrice').pct_change().fillna(0.0)
        return return_mat.cov()*250
```

定义输入协方差矩阵，得到不同优化方法下的权重配置的如下函数：

```
    def get_smart_weight(cov_mat, method='min variance', wts_adjusted=False):
        '''
        功能：输入协方差矩阵，以得到不同优化方法下的权重配置
        输入：
            cov_mat  pd.DataFrame，协方差矩阵，index 和 column 均为资产名称
            method  优化方法，可选的有 min variance、risk parity、max diversification、
equal weight
        输出：
            pd.Series  index 为资产名，values 为 weight
        PS:
            依赖 scipy package
        '''

        if not isinstance(cov_mat, pd.DataFrame):
            raise ValueError('cov_mat should be pandas DataFrame! ')

        omega = np.matrix(cov_mat.values)  # 协方差矩阵
```

```
        # 定义目标函数
        def fun1(x):
            return np.matrix(x) * omega * np.matrix(x).T

        def fun2(x):
            tmp = (omega * np.matrix(x).T).A1
            risk = x * tmp
            delta_risk = [sum((i - risk)**2) for i in risk]
            return sum(delta_risk)

        def fun3(x):
            den = x * omega.diagonal().T
            num = np.sqrt(np.matrix(x) * omega * np.matrix(x).T)
            return num/den

        # 初始值 + 约束条件
        x0 = np.ones(omega.shape[0]) / omega.shape[0]
        bnds = tuple((0,None) for x in x0)
        cons = ({'type':'eq', 'fun': lambda x: sum(x) - 1})
        options={'disp':False, 'maxiter':1000, 'ftol':1e-20}

        if method == 'min variance':
            res = minimize(fun1, x0, bounds=bnds, constraints=cons,
method='SLSQP', options=options)
        elif method == 'risk parity':
            res = minimize(fun2, x0, bounds=bnds, constraints=cons,
method='SLSQP', options=options)
        elif method == 'max diversification':
            res = minimize(fun3, x0, bounds=bnds, constraints=cons,
method='SLSQP', options=options)
        elif method == 'equal weight':
            return pd.Series(index=cov_mat.index, data=1.0 / cov_mat.shape[0])
        else:
            raise ValueError('method should be min variance/risk parity/max
diversification/equal weight!!! ')

        # 权重调整
        if res['success'] == False:
            # print res['message']
            pass
        wts = pd.Series(index=cov_mat.index, data=res['x'])
```

```
        if wts_adjusted == True:
            wts = wts[wts >= 0.0001]
            return wts / wts.sum() * 1.0
        elif wts_adjusted == False:
            return wts
        else:
            raise ValueError('wts_adjusted should be True/False! ')
```

定义获取成分股的函数如下：

```
    def get_idx_cons(idx, date):
        '''
        功能：获取指数在某一天的成分股列表
        输入：
            idx 指数，xxxxxx 型 string，000300 沪深 300，000016 上证 50，000905 中证 500，
000906 中证 800，000001 上证综指
                  可以为多个指数的组合，写法为['xxxxxx','xxxxxx']，此时返回的结果已经去
除重复的 ticker 了!!!
            date yyyymmdd 型 string
        输出：
            list of tickers
        依赖：
            DataAPI：IdxConsGet
        '''

        try:
            data = DataAPI.IdxConsGet(ticker=idx,intoDate=date,field='',
pandas="1")['consTickerSymbol']
        except:
            raise ValueError('DataAPI.IdxConsGet 出错了!!! ')

        return list(set(data))
```

定义一个获取调仓日期列表的函数如下：

```
    def get_dates(start_date, end_date, frequency='daily'):
        '''
        功能：输入起始日期和频率，即可获得日期列表（daily 包括起始日，其余的都是位于起始日
中间的）
        输入参数：
           start_date，开始日期，'xxxxxxxx'形式
           end_date，截止日期，'xxxxxxxx'形式
           frequency，频率，daily 为所有交易日，daily1 为所有自然日，weekly 为每周最后
```

```
一个交易日，weekly2 为每隔两周，monthly 为每月最后一个交易日，quarterly 为每季最后一个交易日
        输出参数：
           获得 list 型日期列表，以'xxxxxxxx'形式存储
        PS:
            要用到 DataAPI.TradeCalGet!!!
        '''
        data = DataAPI.TradeCalGet(exchangeCD=u"XSHG",beginDate=start_date,endDate=end_date,field=u"calendarDate,isOpen,isWeekEnd,isMonthEnd,isQuarterEnd",pandas="1")
        if frequency == 'daily':
            data = data[data['isOpen'] == 1]
        elif frequency == 'daily1':
            pass
        elif frequency == 'weekly':
            data = data[data['isWeekEnd'] == 1]
        elif frequency == 'weekly2':
            data = data[data['isWeekEnd'] == 1]
            data = data[0:data.shape[0]:2]
        elif frequency == 'monthly':
            data = data[data['isMonthEnd'] == 1]
        elif frequency == 'quarterly':
            data = data[data['isQuarterEnd'] == 1]
        else:
            raise ValueError('调仓频率必须为 daily/daily1/ weekly/weekly2 /monthly/quarterly!!! ')
        date_list = data['calendarDate'].values.tolist()
        return date_list
```

定义向前或向后移动 n 个交易日的函数如下：

```
    def shift_date(date, n, direction='back'):
        '''
        功能：给定 date，获取该日期前、后 n 个交易日对应的交易日
        输入：
            date  'yyyymmdd'类型字符串
            n  非负整数，取值区间为（0,720）
            direction  方向，取值为 back/forward
        PS:
            get_dates()
        '''
        last_two_year = str(int(date[:4])-3) + '0101'
```

```
        forward_two_year = str(int(date[:4])+3) + '1231'
        if direction == 'back':
            date_list = get_dates(last_two_year, date, 'daily')
            return date_list[len(date_list)-1-n]
        elif direction == 'forward':
            date_list = get_dates(date, forward_two_year, 'daily')
            return date_list[n]
        else:
            raise ValueError('direction should be back/forward!!! ')
```

定义获取股票日收益率的函数如下：

```
    def get_ticker_period_rtn(tickers, start_date, end_date):
        '''
        功能：输入 tickers + 起始日期，获取 tickers 在这期间的 daily return
        输入：
            ticker，list of ticker string
            start_date，yyyymmdd 日期
            end_date，yyyymmdd 日期
        输出：
            pd.DataFrame，index 为日期 yyyymmdd，columns 为 tickers string
        依赖：
            DataAPI：MktEqudAdjGet
            function：shift_date()
        '''
        begin_date = shift_date(start_date, 1)  # 向前推一天保证第一天也有 daily
return
        data = DataAPI.MktEqudAdjGet(ticker=tickers, beginDate=begin_date,
endDate=end_date, field='ticker,tradeDate,closePrice', pandas='1')
        daily_rtn = data.pivot(index='tradeDate', columns='ticker',
values='closePrice').pct_change().fillna(0.0)
        return daily_rtn
```

定义计算最大回撤的函数如下：

```
    def cal_maxdrawdown(data):
        '''
        功能：给定净值数据（list, np.array, pd.Series, pd.DataFrame），返回最大回撤
        输入：
            data, list/np.array/pd.Series/pd.DataFrame，净值曲线，初始金为 1
        输出：
            list/np.array/pd.Series 返回 float
            pd.DataFrame 返回 pd.DataFrame，index 为 DataFrame.columns
```

```
        '''
        if isinstance(data, list):
            data = np.array(data)
        if isinstance(data, pd.Series):
            data = data.values

        def get_mdd(values): # values 为 np.array 的净值曲线，初始资金为 1
            dd = [values[i:].min() / values[i] - 1 for i in range(len(values))]
            return abs(min(dd))

        if not isinstance(data, pd.DataFrame):
            return get_mdd(data)
        else:
            return data.apply(get_mdd)
```

定义计算策略评价的各项指标的函数如下：

```
    def cal_indicators(df_daily_return):
        '''
        功能：给定 daily return，计算各组合的评价指标，包括：年化收益率、年化标准差、夏普值、最大回撤
        输入：
            df_daily_return  pd.DataFrame，index 为升序排列的日期，columns 为各组合名称，value 为 daily_return
        '''
        df_cum_value = (df_daily_return + 1).cumprod()
        res = pd.DataFrame(index=['年化收益率','年化标准差','夏普值','最大回撤'], columns=df_daily_return.columns, data=0.0)
        res.loc['年化收益率'] = (df_daily_return.mean() * 250).apply(lambda x: '%.2f%%' % (x*100))
        res.loc['年化标准差'] = (df_daily_return.std() * np.sqrt(250)).apply(lambda x: '%.2f%%' % (x*100))
        res.loc['夏普值'] = (df_daily_return.mean() / df_daily_return.std() * np.sqrt(250)).apply(lambda x: np.round(x, 2))
        res.loc['最大回撤'] = cal_maxdrawdown(df_cum_value).apply(lambda x: '%.2f%%' % (x*100))
        return res
```

2）历史回测

回测区间为从 2007 年 1 月 1 日至今，选取上证 50 成分股为标的，调仓频率为每季度末换仓，用过去 500 个交易日的数据估计协方差矩阵。

代码展示如下：

```
    start_date = '2007-01-01'
    end_date = ('2018-01-23')
    idx_code = '000016'  # 上证 50
    cov_Ndays = 500  # 估计协方差矩阵所用历史天数
    # 获取上证 50 历史净值
    idx_data = DataAPI.MktIdxdGet(ticker=idx_code, beginDate=start_date,
endDate=end_date, field=u"tradeDate,closeIndex", pandas="1")
    # idx_data['tradeDate'] = idx_data.tradeDate.apply(lambda x: x.replace('-',
''))
    idx_data['values'] = idx_data.closeIndex / idx_data.closeIndex[0]
    idx_data = idx_data.set_index('tradeDate')
    result = pd.DataFrame(index=idx_data.index, data=0.0, columns=
['SH50','equal weight','min variance','risk parity','max diversification'])
    result['SH50'] = idx_data['values']
    # 回测
    date_list = sorted(get_dates(start_date, end_date, 'quarterly')+
[start_date, end_date])
    for i in range(len(date_list)-1):
        print date_list[i]
        current_period = date_list[i]
        next_period = date_list[i+1]
        tickers = get_idx_cons(idx_code, current_period)
        cov_mat = get_covmat(tickers, current_period, cov_Ndays)
        # 权重优化
        for j in {'min variance', 'risk parity', 'max diversification','equal
weight'}:
            wts = get_smart_weight(cov_mat, method=j, wts_adjusted=True)
            daily_rtn = get_ticker_period_rtn(wts.index.tolist(),
current_period, next_period)
            # print daily_rtn
            daily_rtn.ix[0] = 0.0
            cum_rtn = np.dot((daily_rtn + 1).cumprod(), wts)
            if i == 0:
                result.loc[daily_rtn.index[2:], j] = cum_rtn[2:] * 1.0
            else:
                result.loc[daily_rtn.index,j]=cum_rtn*result.loc[daily_rtn.
index [0] ,j]
    # 格式调整 + 指标计算
    indicators = cal_indicators(result[['SH50','equal weight','min
```

```
variance','risk parity','max diversification']].pct_change().dropna())
```

4 种组合的回测结果如表 6-2 所示。

表 6-2

	SH50	equal weight	min variance	risk parity	max diversification
年化收益率	9.41%	12.31%	10.00%	11.89%	10.71%
年化标准差	29.33%	29.37%	25.22%	28.09%	34.35%
夏普值	0.32	0.42	0.4	0.42	0.31
最大回撤	72.41%	70.96%	68.81%	71.03%	77.92%

从表 6-2 可以看出，这 4 种组合的年化收益率都要高于上证 50 指数，除最大多元化组合外的其他三个组合的最大回撤均小于上证 50 指数，并且其他三个组合的夏普值也均高于上证 50 指数。等权组合的结果从整体上看是最好的，因为该组合的年化收益率最高为 12.31%，夏普值 0.42 也是其中最大的，最大回撤也比较小，仅高于最小方差组合，同时年化标准差跟上证指数的年化标准差基本相同。从总体来说通过对权重的调整，我们实现了比指数表现更好的组合，无论是从收益还是风险的角度来看。

权重优化视角下的 Smart Beta 业绩展示如图 6-14 所示。

图 6-14

6.6.2 基于风险因子的 Smart Beta

在 6.6.1 节详细介绍了基于权重角度的 4 种优化组合的方法，本节从股票收益的来源出发，分析和验证 6 个常见风险因子的历史表现，包括价值、成长、质量、股息、规模和动量。

1. 简介

股票的收益是受很多因素共同决定的，从三因子模型到结构化风险模型，投资者对股票收益的来源了解得越来越清晰。尽管没有哪个风险因子能够长期跑赢市场，但从更长远的角度来看，不同的风险因子让投资者对组合的了解更加清晰。我们可以通过构建投资组合使其风险盯住某个风险因子，并分析组合长期回测的结果来看其表现是否优于基准。

2. 基于风险因子的 Smart Beta——价值

本策略的思想是：价值因子着眼于价值被低估的股票，认为投资低估值的股票与投资高估值的股票相比可以获取超额收益，因此选取常见的市盈率、市净率来构建价值因子，等权合成。

回测分析如下。

◎ 回测区间：2008 年 1 月 1 日至 2017 年 12 月 31 日，基准为沪深 300 指数。

◎ 股票池：沪深 300 与中证 500 动态成分股，每两周调仓一次，为 20%选股比例。

◎ 因子选取：市盈率 PE、市净率 PB。

◎ 因子处理：组合经过市场中性化处理。

代码展示如下：

```
    import numpy as np
    import pandas as pd
    start = '2008-01-01'
    end = '2017-12-31'   # 回测结束时间
    universe = DynamicUniverse('HS300') + DynamicUniverse('ZZ500')
    benchmark = 'HS300'                 # 策略参考标准
    freq = 'd'                          # 策略类型，'d'表示日间策略使用日线回测，'m'
表示日内策略使用分钟线回测
    refresh_rate = 10                   # 调仓频率
```

```
    accounts={'fantasy_account':AccountConfig(account_type='security',
capital_base=10000000)}
    def initialize(context):                    # 初始化虚拟账户状态
        context.signal_generator = SignalGenerator(Signal('PB'), Signal('PE'))
    def handle_data(context):                   # 每个交易日的买入卖出指令
        account = context.get_account('fantasy_account')
        current_universe = context.get_universe(exclude_halt=True)
        yesterday = context.previous_date.strftime('%Y-%m-%d')
        signals=context.history(current_universe, ['PE', 'PB'], 1, freq='1d',
rtype='frame',style='tas')[yesterday]
        signal1 = standardize((1.0 / winsorize(signals['PE'])).replace([np.inf,
-np.inf], 0.0))
        signal2 = standardize((1.0 / winsorize(signals['PB'])).replace([np.inf,
-np.inf], 0.0))
        signal = (0.5*signal1).add(0.5*signal2, fill_value=0.0)
        wts=long_only(signal.to_dict(),select_type=1, top_ratio=0.2,
weight_type=1, target_date=yesterday)
        # 交易部分
        positions = account.get_positions()
        sell_list = [stk for stk in positions if stk not in wts]
        for stk in sell_list:
            order_to(stk,0)
        c = account.portfolio_value
        change = {}
        for stock, w in wts.iteritems():
            p = context.current_price(stock)
            if not np.isnan(p) and p > 0:
                if stock in positions:
                    change[stock] = int(c * w / p) - positions[stock].amount
                else:
                    change[stock] = int(c * w / p) - 0
        for stock in sorted(change, key=change.get):
            account.order(stock, change[stock])
```

本策略的回测结果如图 6-15 所示。可以看出，从 2008 年 1 月 1 日到 2017 年 12 月 31 日，策略的年化收益达到 7.7%，基准的年化仅为-2.8%，策略年化超额收益为 10.5%。另外，策略的阿尔法为 10.6%，夏普比率为 0.14，最大回撤达到 67.6%。从整体上看，策略的表现比较稳定，尤其是在市场强调价值投资的 2017 年表现十分优异。

图 6-15

3. 基于风险因子的 Smart Beta——成长

本策略的思想为：成长因子专注于选取具有高成长性的公司，认为投资高成长性公司相对于投资低成长性的公司可以获取超额收益，因此选取常见的营业收入增长率、总资产增长率、归属于母公司所有者的净利润增长率来构建成长因子，等权合成。

回测分析如下。

◎ 回测区间：2008 年 1 月 1 日至 2017 年 12 月 31 日，基准为沪深 300 指数。

◎ 股票池：沪深 300 与中证 500 动态成分股，每两周调仓一次，为 20%选股比例。

◎ 因子选取：营业收入增长率、总资产增长率、归属于母公司所有者的净利润增长率。

◎ 因子处理：组合经过市场中性化处理。

代码展示如下：

```
import numpy as np
import pandas as pd
start = '2008-01-01'
end = '2017-12-31'   # 回测结束时间
universe = DynamicUniverse('HS300') + DynamicUniverse('ZZ500')
benchmark = 'HS300'              # 策略参考标准
freq = 'd'                       # 策略类型，'d'表示日间策略使用日线回测，'m'表示
```

```
日内策略使用分钟线回测
    refresh_rate = 10                          # 调仓频率
    accounts={'fantasy_account':AccountConfig(account_type='security',capita
l_base=10000000)}
    def initialize(context):                   # 初始化虚拟账户状态
        context.signal_generator=SignalGenerator(Signal('TotalAssetGrowRate
'),Signal('NPParentCompanyGrowRate'), Signal('OperatingRevenueGrowRate'))
    def handle_data(context):                  # 每个交易日的买入卖出指令
        account = context.get_account('fantasy_account')
        current_universe = context.get_universe(exclude_halt=True)
        yesterday = context.previous_date.strftime('%Y-%m-%d')
        signals=context.history(current_universe,['TotalAssetGrowRate',
'NPParentCompanyGrowRate','OperatingRevenueGrowRate'],1,freq='1d',rtype='fra
me',style='tas')[yesterday]
        signal1 = standardize(winsorize(signals['TotalAssetGrowRate']))
        signal2 = standardize(winsorize(signals['NPParentCompanyGrowRate']))
        signal3 = standardize(winsorize(signals['OperatingRevenueGrowRate']))
        signal=(0.33*signal1).add(0.34*signal2,fill_value=0.0).add(0.33*
signal3,fill_value=0.0)
        wts=long_only(signal.to_dict(),select_type=1,top_ratio=0.2,weight_
type=1,target_date=yesterday)
        # 交易部分
        positions = account.get_positions()
        sell_list = [stk for stk in positions if stk not in wts]
        for stk in sell_list:
            order_to(stk,0)
        c = account.portfolio_value
        change = {}
        for stock, w in wts.iteritems():
            p = account.reference_price[stock]
            if not np.isnan(p) and p > 0:
                if stock in positions:
                    change[stock] = int(c * w / p) - positions[stock].amount
                else:
                    change[stock] = int(c * w / p) - 0
        for stock in sorted(change, key=change.get):
            account.order(stock, change[stock])
```

本策略的回测结果如图 6-16 所示。可以看出，从 2008 年 1 月 1 日到 2017 年 12 月 31 日，策略的年化收益达到 2.2%，基准的年化仅为-2.8%，策略年化超额收益为 5%。另外，

策略的阿尔法为 5.4%，夏普比率为-0.04，最大回撤达到 72.1%。总体来说，以成长因子做 Smart Beta 策略长期来看仍比指数跑得略好一些。

图 6-16

4. 基于风险因子的 Smart Beta——质量

本策略的思想为：质量因子专注于选取具有高财务质量的公司，认为投资高财务质量的公司相对于投资低财务质量的公司可以获得超额收益，因此选取流动比率、营业利润率、权益收益率、总资产周转率来构建质量因子，等权合成。

回测分析如下。

◎ 回测区间：2008 年 1 月 1 日至 2017 年 12 月 31 日，基准为沪深 300 指数。

◎ 股票池：沪深 300 与中证 500 动态成分股，每两周调仓一次，为 20%选股比例。

◎ 因子选取：流动比率、营业利润率、权益收益率、总资产周转率。

◎ 因子处理：组合经过市场中性化处理。

代码展示如下：

```
import numpy as np
import pandas as pd
start = '2008-01-01'
end = '2017-12-31'   # 回测结束时间
```

```
    universe = DynamicUniverse('HS300') + DynamicUniverse('ZZ500')
    benchmark = 'HS300'                          # 策略参考标准
    freq = 'd'                    # 策略类型，'d'表示日间策略使用日线回测，'m'表示日内
策略使用分钟线回测
    refresh_rate = 10                            # 调仓频率
    accounts={'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000) }
    def initialize(context):                     # 初始化虚拟账户状态
        context.signal_generator=SignalGenerator(Signal('CurrentRatio'),
Signal('OperatingProfitRatio'), Signal('ROE'), Signal('TotalAssetsTRate'))
    def handle_data(context):                     # 每个交易日的买入卖出指令
        account = context.get_account('fantasy_account')
        current_universe = context.get_universe(exclude_halt=True)
        yesterday = context.previous_date.strftime('%Y-%m-%d')
        signals=context.history(current_universe,['CurrentRatio', 'Operating
ProfitRatio', 'ROE', 'TotalAssetsTRate'], 1, freq='1d', rtype='frame',
style='tas') [yesterday]
        signal1 = standardize(winsorize(signals['CurrentRatio'].dropna()))
        signal2 = standardize(winsorize(signals['OperatingProfitRatio'].
dropna()))
        signal3 = standardize(winsorize(signals['ROE'].dropna()))
        signal4 = standardize(winsorize(signals['TotalAssetsTRate'].dropna()))
        signal = (0.25*signal1).add(0.25*signal2, fill_value=0.0).add
(0.25*signal3, fill_value=0.0).add(0.25*signal4, fill_value=0.0)
        wts=long_only(signal.to_dict(),select_type=1, top_ratio=0.2,
weight_type=1, target_date=yesterday)
        # 交易部分
        positions = account.get_positions()
        sell_list = [stk for stk in positions if stk not in wts]
        for stk in sell_list:
            order_to(stk,0)
        c = account.portfolio_value
        change = {}
        for stock, w in wts.iteritems():
            p = account.reference_price[stock]
            if not np.isnan(p) and p > 0:
                if stock in positions:
                    change[stock] = int(c * w / p) - positions[stock].amount
                else:
                    change[stock] = int(c * w / p) - 0
        for stock in sorted(change, key=change.get):
```

```
            account.order(stock, change[stock])
```

本策略的回测结果如图 6-17 所示。可以看出，从 2008 年 1 月 1 日到 2017 年 12 月 31 日，策略的年化收益达到 1.8%，基准的年化仅为-2.8%，策略年化超额收益为 4.6%。另外，策略的阿尔法为 4.8%，夏普比率为-0.06，最大回撤达到 72.1%。此处选用的质量因子涵盖了赢利、偿债及营运等多方面的能力，致力于挑选财务较好的公司。最后策略的收益也略跑赢指数，总体表现不如价值因子，可见在 A 股市场单纯考虑质量好的公司是不够的，还需要考虑它们的估值水平。

图 6-17

5. 基于风险因子的 Smart Beta——股息

本策略的思想为：股息因子专注于选取高分红的公司，认为投资高分红公司相对于投资低分红公司可以获得超额收益，因此选取现金流市值比、5 年平均现金流市值比来构建股息因子，等权合成。

回测分析如下。

- ◎ 回测区间：2008 年 1 月 1 日至 2017 年 12 月 31 日，基准为沪深 300 指数。
- ◎ 股票池：沪深 300 与中证 500 动态成分股，每两周调仓一次，为 20%选股比例。
- ◎ 因子选取：现金流市值比、5 年平均现金流市值比。
- ◎ 因子处理：组合经过市场中性化处理。

代码展示如下：

```
    import numpy as np
    import pandas as pd
    start = '2008-01-01'
    end = '2017-12-31'   # 回测结束时间
    universe = DynamicUniverse('HS300') + DynamicUniverse('ZZ500')
    benchmark = 'HS300'                 # 策略参考标准
    freq = 'd'                          # 策略类型，'d'表示日间策略使用日线回测，'m'
表示日内策略使用分钟线回测
    refresh_rate = 10                   # 调仓频率
    accounts={'fantasy_account':AccountConfig(account_type='security',
capital_base=10000000) }
    def initialize(context):            # 初始化虚拟账户状态
        context.signal_generator = SignalGenerator(Signal('CTOP'),
Signal('CTP5'))
    def handle_data(context):           # 每个交易日的买入卖出指令
        account = context.get_account('fantasy_account')
        current_universe = context.get_universe(exclude_halt=True)
        yesterday = context.previous_date.strftime('%Y-%m-%d')
        signals = context.history(current_universe, ['CTOP', 'CTP5'], 1,
freq='1d', rtype='frame',style='tas')[yesterday]
        signal1 = standardize(winsorize(signals['CTOP'].dropna()))
        signal2 = standardize(winsorize(signals['CTP5'].dropna()))
        signal = (0.5*signal1).add(0.5*signal2, fill_value=0.0)
        wts=long_only(signal.to_dict(),select_type=1, top_ratio=0.2,
weight_type=1, target_date=yesterday)
        # 交易部分
        positions = account.get_positions()
        sell_list = [stk for stk in positions if stk not in wts]
        for stk in sell_list:
            order_to(stk,0)
        c = account.portfolio_value
        change = {}
        for stock, w in wts.iteritems():
            p = account.reference_price[stock]
            if not np.isnan(p) and p > 0:
                if stock in positions:
                    change[stock] = int(c * w / p) - positions[stock].amount
                else:
```

```
                change[stock] = int(c * w / p) - 0
        for stock in sorted(change, key=change.get):
            account.order(stock, change[stock])
```

本策略的回测结果如图 6-18 所示。可以看出，从 2008 年 1 月 1 日到 2017 年 12 月 31 日，本策略的年化收益达到 4.6%，基准的年化仅为-2.8%，策略年化超额收益为 7.4%。另外，策略的阿尔法为 7.4%，夏普比率为 0.04，最大回撤达到 68%。从整体上分析，策略的累积收益明显高于市场基准的收益，年化超额达到 7.4%。从总体上看，现金流市值比、5 年平均现金流市值比这两个因子比较理想；从长期来看，投资高分红的公司可以获得较明显高于市场指数的收益。尤其是在 2017 年的股票市场上，该策略的表现十分优异，这是因为当时高分红的公司特别受投资者青睐。

图 6-18

6. 基于风险因子的 Smart Beta——规模

本策略的思想为：规模因子专注于选取市值小的公司，认为投资小市值公司相对于投资大市值公司可以获得超额收益，因此选取对数总市值来构建规模因子。

回测分析如下。

◎ 回测区间：2008 年 1 月 1 日至 2017 年 12 月 31 日，基准为沪深 300 指数。

◎ 股票池：沪深 300 与中证 500 动态成分股，每两周调仓一次，为 20%选股比例。

◎ 因子选取：对数总市值。

◎　因子处理：组合经过市场中性化处理。

代码展示如下：

```
    import numpy as np
    import pandas as pd
    start = '2008-01-01'
    end = '2017-12-31'   # 回测结束时间
    universe = DynamicUniverse('HS300') + DynamicUniverse('ZZ500')
    benchmark = 'HS300'                          # 策略参考标准
    freq = 'd'                                   # 策略类型,'d'表示日间策略使用日线回测,
'm'表示日内策略使用分钟线回测
    refresh_rate = 10                            # 调仓频率
    accounts = { 'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000) }
    def initialize(context):                     # 初始化虚拟账户状态
        context.signal_generator = SignalGenerator(Signal('LCAP'))
    def handle_data(context):                    # 每个交易日的买入卖出指令
        account = context.get_account('fantasy_account')
        current_universe = context.get_universe(exclude_halt=True)
        yesterday = context.previous_date.strftime('%Y-%m-%d')
        signals = context.history(current_universe, 'LCAP', 1, freq='1d',
rtype='frame',style='tas')[yesterday]
        signal=standardize((1.0/winsorize(signals['LCAP'])).replace([np.inf,
-np.inf], 0.0))
        wts=long_only(signal.to_dict(),select_type=1, top_ratio=0.2,
weight_type=0, target_date=yesterday)
        # 交易部分
        positions = account.get_positions()
        sell_list = [stk for stk in positions if stk not in wts]
        for stk in sell_list:
            order_to(stk,0)
        c = account.portfolio_value
        change = {}
        for stock, w in wts.iteritems():
            p = account.reference_price[stock]
            if not np.isnan(p) and p > 0:
                if stock in positions:
                    change[stock] = int(c * w / p) - positions[stock].amount
                else:
                    change[stock] = int(c * w / p) - 0
        for stock in sorted(change, key=change.get):
```

```
            account.order(stock, change[stock])
```

本策略的回测结果如图 6-19 所示。可以看出，从 2008 年 1 月 1 日到 2017 年 12 月 31 日，策略的年化收益达到 7.3%，基准的年化仅为-2.8%，策略年化超额收益为 10.1%。另外，策略的阿尔法为 10.7%，夏普比率为 0.11，最大回撤达到 71.3%。从整体来看，策略的累积收益明显高于市场基准的收益，年化超额达到 10.1%；从规模效应来看，小市值公司会有更好的表现，在 2017 年之前总体来说都是成立的，并且虽然有较高的波动，但是小市值策略带来的 Alpha 是很高的；而从 2017 年开始，受到市场监管环境等影响，小市值策略开始失效，市场开始反转，大市值反而表现得更加良好。所以以市值构建的 Smart Beta 策略虽然长期表现得仍然不错，但是需要注意风险，在合适的市场环境下投资。

图 6-19

7. 基于风险因子的 Smart Beta——动量

本策略的思想为：动量因子专注于市场行情规律，通过实践发现，反转效应在 A 股市场更明显，所以这里又可以叫作反转因子，选取过去 60 日的收益和过去 120 日的收益来构建反转因子，等权合成。

回测分析如下。

- ◎ 回测区间：2008 年 1 月 1 日至 2017 年 12 月 31 日，基准为沪深 300 指数。
- ◎ 股票池：沪深 300 与中证 500 动态成分股，每两周调仓一次，为 20%选股比例。
- ◎ 因子选取：过去 60 日收益、过去 120 日的收益。
- ◎ 因子处理：组合经过市场中性化处理。

代码展示如下：

```
    import numpy as np
    import pandas as pd
    start = '2008-01-01'
    end = '2017-12-31'   # 回测结束时间
    universe = DynamicUniverse('HS300') + DynamicUniverse('ZZ500')
    benchmark = 'HS300'                          # 策略参考标准
    freq = 'd'                                   # 策略类型,'d'表示日间策略使用日线回测,
'm'表示日内策略使用分钟线回测
    refresh_rate = 10                            # 调仓频率
    accounts = { 'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000) }
    def initialize(context):                     # 初始化虚拟账户状态
        context.signal_generator=SignalGenerator(Signal('REVS60'),
Signal('REVS120'))
    def handle_data(context):                    # 每个交易日的买入卖出指令
        account = context.get_account('fantasy_account')
        current_universe = context.get_universe(exclude_halt=True)
        yesterday = context.previous_date.strftime('%Y-%m-%d')
        signals = context.history(current_universe, ['REVS60', 'REVS120'], 1,
freq='1d', rtype='frame',style='tas')[yesterday]
        signal1 = -standardize(winsorize(signals['REVS60']))
        signal2 = -standardize(winsorize(signals['REVS120']))
        signal = (0.5*signal1).add(0.5*signal2, fill_value=0.0)
        wts=long_only(signal.to_dict(),select_type=1, top_ratio=0.2,
weight_type=1, target_date=yesterday)
        # 交易部分
        positions = account.get_positions()
        sell_list = [stk for stk in positions if stk not in wts]
        for stk in sell_list:
            order_to(stk,0)
        c = account.portfolio_value
        change = {}
        for stock, w in wts.iteritems():
            p = account.reference_price[stock]
            if not np.isnan(p) and p > 0:
                if stock in positions:
                    change[stock] = int(c * w / p) - positions[stock].amount
```

```
            else:
                change[stock] = int(c * w / p) - 0
        for stock in sorted(change, key=change.get):
            account.order(stock, change[stock])
```

本策略的回测结果如图 6-20 所示。可以看出，从 2008 年 1 月 1 日到 2017 年 12 月 31 日，策略的年化收益达到 6%，基准的年化仅为-2.8%，策略年化超额收益为 8.8%。另外，策略的阿尔法为 9.3%，夏普比率为 0.08，最大回撤达到 70.8%。从整体上来看，策略的累积收益明显高于市场基准的收益，年化超额达到 8.8%，反转因子的回测结果比较理想；从长期来看，选取过去 60 日收益、过去 120 日的收益来构建反转因子组合可以获得明显高于市场指数的收益；但该因子相比于指数在 2017 年的表现并不稳定，指数表现为稳定的缓缓增长，策略则出现较大的回测。

图 6-20

8. 风险因子小结

6 类风险因子的回测结果如表 6-3 所示。可以看出，价值类的因子从整体来看优于其他因子，价值类因子的年化收益率最大，夏普比率也是其中最大的，最大回撤是其中最小的，收益波动率也比较小，仅仅大于股息因子的波动率。其次是市值因子，市值因子的年化收益率和夏普比率也相对较高，仅小于价值因子的结果，但市值因子的波动率最大。但从总体来看，基于风险因子的 Smart Beta 策略在很长一段时期内都要比以市值加权的指数的表现好。

表 6-3

	价　值	成　长	质　量	市　值	股　息	动　量
年化收益率	7.70%	2.20%	1.80%	7.30%	4.60%	6%
收益波动率	29.10%	30.90%	29.30%	34.20%	28.10%	31.80%
夏普值	0.14	-0.04	-0.06	0.11	0.04	0.08
最大回撤	67.60%	72.10%	72.10%	71.30%	68%	70.8

6.7　技术指标类策略

本节将继续介绍我们常用的量化策略之一：技术指标类策略。区别于基本面对于市场经济情况及公司经营管理情况的研究，技术指标是指根据投资者对某些特定方面市场行为的考虑，例如股价、成交量及衍生涨跌指数数据，结合数学上的计算方法，通过图表和技术指标的分析来研究市场行为和预测价格变动。

各项技术指标的具体数值及相互间的关系可反映股票市场的状态，对我们的市场操作具有一定的指导性。常见的技术指标有动量指标、随机指数、均线等，本节将结合通联数据平台介绍以下 6 种技术指标策略，在每一部分都将对指标的含义、计算方法、使用方法做出详细说明，并对每个技术指标策略的收益情况进行回测分析。

6.7.1　AROON 指标

1. 指标介绍

阿隆指标（AROON）由图莎尔 • 钱德（Tushar Chande）于 1995 年发明，通过计算价格从达到近期最高值和最低值以来所经过的周期数，帮助投资者预测证券价格从趋势到区域、区域或反转的变化。相较于关注相对于时间的价格的趋势指标，AROON 指标更关注相对价格的时间。

2. 计算公式

在计算阿隆指标时通常需要事先设定一个周期 T，一般取 25，然后通过统计距离最近

一段时间的最高、最低价格的时间来计算 Aroon-Up 和 Aroon-Down，具体计算公式为

$$Aroon\text{-}Up = ((T - Days\ Since\ T - day\ High)/T) \times 100$$

$$Aroon\text{-}Down = ((T - Days\ Since\ T - day\ Low)/T) \times 100$$

将上面的两个指标相减，可以进一步得出 Aroon-Osc 指标，即

$$Aroon\text{-}Osc = Aroon\text{-}Up - Aroon\text{-}Down$$

3. 指标使用说明

在对 Aroon 指标进行分析时，主要观察以下 4 种状态。

（1）Aroon 指标高于 70 时表示强势，低于 50 时表示弱势。以 Aroon-Up 线为例，当其达到 100 时，表示处于多头强势；维持在 70～100 时，表示处于多头上升趋势；维持在 0～30 时，表示多头处于较弱势；达到 0 时，表示多头处于极度弱势。Aroon-Down 线同理。如果两条线同时处于底部，则表示处于盘整时期，无明显趋势。

（2）平行运动：当两条线平行运动时，则表示原有趋势仍在继续，直到达到极值水平或者发生交叉穿行时才会改变趋势。

（3）交叉穿行：当上行线下穿下行线时，则表示原有趋势正逐渐减弱，预计将发生反转。

（4）Arron-Osc 线高于零表示处于上升趋势，低于零则表示处于下降趋势；与零线偏离越远，则表示走势越强。

4. 指标回测

下面构建基于 Aroon 指标的策略，具体的策略要求如下。

（1）如果 Aroon-Up 高于 70 且 Aroon-Down 低于 30，则买入；如果 Aroon-Down 高于 70 且 Aroon-Up 低于 30，则卖出。

（2）等权买入符合条件股票，如果可选的股票少于 10 只，则每只买入 0.1，出现重复信号时不重复买入。

（3）如果股票数量高于可用资金，则随机买入其中一部分，保证资金用完。

（4）回测时间为 2012 年 1 月 1 日至 2017 年 12 月 31 日，基准为 HS300，股票池为动态 HS300 成分股，每日调仓。

此处介绍基于通联数据平台的 AROON 指标策略，主要代码如下：

```
    import numpy as np
    import pandas as pd

    start = '2012-01-01'
    end = '2017-12-31'
    benchmark = 'HS300'
    universe = DynamicUniverse('HS300')
    capital_base = 1000000
    freq = 'd'
    refresh_rate = 1

    high_threshold = 70     # 强趋势
    low_threshold = 30      # 弱趋势

    accounts = {
        'fantasy_account': AccountConfig(account_type='security', capital_base
=10000000)
    }

    # 定义并注册 Signal 模块
    def initialize(context):
        # 优矿的 AroonUp、AroonDown 的参数都是 26
        aroon_up = Signal('AroonUp')
        aroon_down = Signal('AroonDown')
        context.signal_generator = SignalGenerator(aroon_up, aroon_down)

    def handle_data(context):
        current_universe = context.get_universe(exclude_halt=True)
        aroon_up, aroon_down = context.signal_result['AroonUp'], context.
signal_result['AroonDown']

        buylist = []
        account = context.get_account('fantasy_account')
        cash = account.cash

        current_positions = account.get_positions()
```

```
        for stock in current_universe:
            # 存在上涨强趋势，且无持仓
            if (aroon_up[stock] > high_threshold and aroon_down[stock] <
low_threshold) and stock not in current_positions:
                buylist.append(stock)

            # 存在下跌强趋势，且有持仓
            elif (aroon_up[stock] > high_threshold and aroon_down[stock] <
low_threshold) and stock in current_positions:
                order_to(stock, 0)  # 全部卖出
                cash += current_positions[stock].amount * context.current_price
(stock)  # 估计买入金额

        if len(buylist)==0:
            weight = 0
        else:
            weight = min(1.0/len(buylist), 0.1)
        for sec in buylist:
            order_pct_to(sec, weight)
```

AROON 指标策略的结果及其分析如图 6-21 所示。

图 6-21

根据上述 AROON 指标规则构建的策略结果如下，相对于在过去 6 年内基准取得 9.7% 的年化收益，我们的策略收益为 6.1%，策略结果差强人意。仔细观察历史收益走势可以发现，该策略的大部分时间与指数走势比较相似，读者可考虑在 AROON 指标的基础上进行一些修正来运用。

6.7.2　BOLL 指标

1. 指标介绍

布林线指标（BOLL）由约翰·布林先生创造，利用统计原理求出股价的移动平均、标准差及信赖区间，从而确定股价的波动高低价位，因此也被称为布林带。标准的布林带由三条线构成：上轨、中轨和下轨。上下轨的范围通常不固定，随股价的变化而变化，通常在股价涨跌幅度加大时，带状区会变宽；在涨跌幅较小时，带状区同样较窄。投资者可参考压力线和支撑线、指标开口大小来进行投资决策。

2. 计算公式

计算 BOLL 指标通常需要设置计算周期 N 和信赖区间倍数 k，一般取 N=20，k=2，其余三条轨道的具体计算方式如下。

（1）中轨：为 N 日的简单移动平均。

（2）上轨：为中轨与 k 倍的 N 日标准差的和。

（3）下轨：为中轨与 k 倍的 N 日标准差的差。

3. 指标使用说明

在运用 BOLL 指标时，通常将股价与三条轨线进行比较。在一般情况下，股价应始终在信道之内运行，而信道也会随着股价的变动而变动，如果股价脱离信道运行，则意味着行情处于比较极端的情况。其中，上下轨代表股价安全运行时的最高价位和最低价位。三条线均可起到支撑作用，而上轨线和中轨线可以起到压力的作用。当股价在布林线的中轨线上方运行时，则表示股价处于强势趋势；当股价在布林线的中轨线下方运行时，则表示处于弱势趋势。

4. 指标回测

下面构建基于 BOLL 指标的策略，具体的策略要求如下。

（1）如果收盘价上穿 BOLL 上轨，则买入；如果收盘价下穿 BOLL 下轨，则开盘卖掉。

（2）等权买入符合条件的股票，如果可选的股票少于 10 只，则每只买入 0.1，出现重复信号时不重复买入。

（3）如果股票数量高于可用资金，则随机买入其中一部分，保证资金用完。

（4）回测时间为 2014 年 1 月 1 日至 2017 年 12 月 31 日，基准为 HS300，股票池为动态 HS300 成分股，每日调仓。

通联平台的代码如下：

```
    import numpy as np
    import pandas as pd

    start = '2014-01-01'
    end = '2017-12-31'
    benchmark = 'HS300'
    universe = DynamicUniverse('HS300')
    capital_base = 1000000
    freq = 'd'
    refresh_rate = 1

    signalperiod = 9  # Signal 平滑周期

    accounts = {
        'fantasy_account': AccountConfig(account_type='security', capital_
base=10000000)
    }

    # 利用优矿的 Signal 框架来计算指标
    # 优矿的 BollUp 和 BollDown 的参数是 N=20，k=2，如果需要用到中轨，则直接使用 MA20 因
子
    def cross_situation(data, dependencies=['closePrice', 'BollUp', 'MA20',
'BollDown'], max_window=2):
        cross = {}
        for sec in data['closePrice'].columns:
            # 收盘价上穿 BOLL 上轨
            if data['closePrice'][sec][0] < data['BollUp'][sec][0] and
data['closePrice'][sec][1] < data['BollUp'][sec][1]:
                cross[sec] = 1

            # 收盘价下穿 BOLL 中轨
            elif data['closePrice'][sec][0] > data['MA20'][sec][0] and
data['closePrice'][sec][1] < data['MA20'][sec][1]:
                cross[sec] = -1
            else:
```

```
            cross[sec] = 0
        return pd.Series(cross)

    # 定义并注册 Signal 模块
    def initialize(context):
        up = Signal('BollUp')
        down = Signal('BollDown')
        cross = Signal('CRX', cross_situation)
        context.signal_generator = SignalGenerator(up, down, cross)

    def handle_data(context):
        current_universe = context.get_universe(exclude_halt=True)
        boll_up, boll_down, crx = context.signal_result['BollUp'],
context.signal_result['BollDown'], context.signal_result['CRX']

        buylist = []
        account = context.get_account('fantasy_account')
        cash = account.cash
        current_positions = account.get_positions()
        for stock in current_universe:
            # 收盘价上穿 BOLL 上轨，且无持仓
            if crx[stock] == 1 and stock not in current_positions:
                buylist.append(stock)

            # 收盘价下穿 BOLL 中轨，且有持仓
            elif crx[stock] == -1 and stock in current_positions:
                order_to(stock, 0)  # 全部卖出
                cash += current_positions[stock].amount * context.current_price
(stock)  # 估计买入金额

        if len(buylist)==0:
            weight = 0
        else:
            weight = min(1.0/len(buylist), 0.1)
        for sec in buylist:
            order_pct_to(sec, weight)
```

BOLL 指标策略的回测结果如图 6-22 所示。可以看出，BOLL 指标策略与指数走势十分相似，基本抓住了市场上的牛市机会，且大部分时间优于指数，但同时无法有效规避风险。投资者可考虑在长线投资中使用此指标，并结合其他指标来进行风险管控。

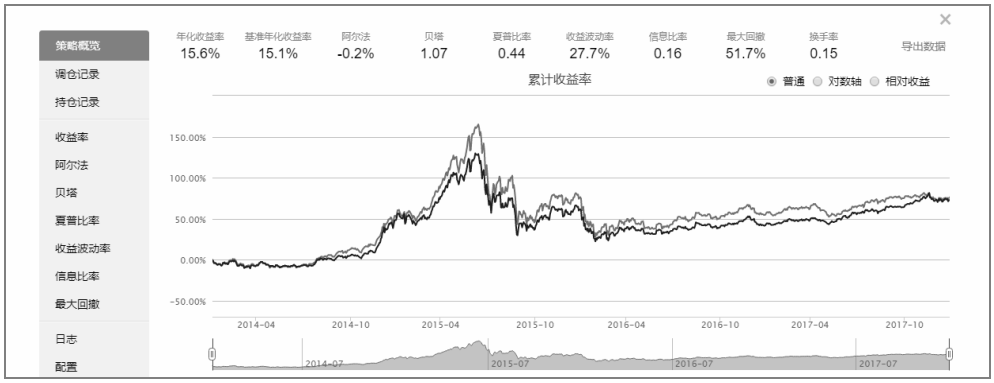


图 6-22

6.7.3 CCI 指标

1. 指标介绍

CCI（顺势指标）由唐纳德·蓝伯特先生提出，通过引用价格与固定区间的股价平均区间的偏离程度的概念，来判断是否处于超买或者超卖状态。CCI 指标区别于大多超买超卖型指标的 0～100 界限，波动于正无穷大和负无穷大之间，无须以 0 为中轴线，有利于投资者对行情进行研判，对于短期内的暴涨暴跌的非常态行情也同样适用。

2. 计算公式

在计算 CCI 指标时通常需要设置计算周期 n 和系数 c，一般取 n=20，c=0.015，数值的具体计算方式为：

$$CCI_n = \frac{Typical\ Price - SMA_n(TP)}{c \cdot \delta_n(TP)}$$

其中 $Typical\ Price$ =（最高价+最低价+收盘价）/ 3

3. 指标使用说明

在对 CCI 指标进行分析时，我们主要对非常态市场进行研判。

（1）当 CCI 曲线向上突破+100 线进入非常态区间时，则表明股价进入强势区间，投资者应及时买入。在进入非常态区间后，如果继续一直朝上运行，则表明股价持续强势。

如果在远离 100 线的地方开始掉头，则表明强势状态已难以持续；同时，如果前期涨幅过高，则投资者应考虑逢高卖出。如果出现持续下跌状态，则表明强势趋势已经结束。

（2）当 CCI 曲线向下突破-100 线进入非常态区间时，则表明股价进入弱势区间，投资者应考虑空仓并等待更高利润。如果在超卖区间运行一段时间后开始掉头，则表明底部已初步探明，投资者可适量建仓。当由下向上突破-100 线进入常态区间时，则表明市场探底基本结束，可能进入盘整阶段，投资者可考虑逢低建仓。

通过上面的分析我们可以看出，CCI 指标主要适用于股票市场的超买与超卖区间，在相对准确性方面，对于急涨急跌的行情更适合采用 CCI 指标。

6.8.3.4　指标回测

下面构建基于 CCI 指标的简单策略，具体的策略要求如下。

（1）如果 CCI 指标低于-100，则买入；如果 CCI 指标高于+100，则卖出。

（2）对每只股票买入 20000 元左右，出现重复信号时不重复买入。

（3）如果股票数量高于可用资金，则随机买入其中一部分，保证资金用完。

（4）回测时间为 2016 年 1 月 1 日至 2017 年 12 月 31 日，基准为 HS300，股票池为动态 HS300 成分股，每日调仓。

上面已经介绍了如何基于通联数据平台编写技术指标策略，这里将介绍另一种基于 Talib 的技术指标策略编写，并将之与通联数据平台进行比较。

Talib 的方法代码如下：

```
import talib
import numpy as np

start = '2016-01-01'
end = '2017-12-31'
benchmark = 'HS300'
universe = DynamicUniverse('HS300')
capital_base = 1000000
freq = 'd'
refresh_rate = 1

N = 20  # 移动窗口长度
```

```
    buy_threshold = -100  # 买入阈值
    sell_threshold = 100  # 卖出阈值

    accounts = {
        'fantasy_account': AccountConfig(account_type='security', capital_
base=10000000)
    }

    def initialize(context):
        return

    def handle_data(context):
        current_universe = context.get_universe(exclude_halt=True)

        # 拿过去 N 个交易日的最高价、最低价、收盘价来估算 CCI
        data = context.history(current_universe, ['highPrice', 'lowPrice',
'closePrice'], N+1, rtype='array')

        buylist = []
        account = context.get_account('fantasy_account')
        cash = account.cash
        current_positions = account.get_positions()
        for stock in current_universe:
            cci = talib.CCI(data[stock]['highPrice'], data[stock]['lowPrice'],
data[stock]['closePrice'], timeperiod=N)[-1]
            # CCI 处于超卖期，且无持仓
            if cci < buy_threshold and stock not in current_positions:
                buylist.append(stock)

            # CCI 处于超买期，且有持仓
            elif cci > sell_threshold and stock in current_positions:
                order_to(stock, 0)  # 全部卖出
                cash += current_positions[stock].amount * context.current_
price(stock)  # 估计买入金额
        d = min(len(buylist), int(cash) // 20000)
    # 可以买入的股票数量，如果资金不够，则只买入其中一部分
        for stock in buylist[:d]:
            order(stock, 20000 / context.current_price(stock))
```

Uqer 方法的代码如下：

```
    import numpy as np
```

```
    import pandas as pd

    start = '2016-01-01'
    end = '2017-12-31'
    benchmark = 'HS300'
    universe = DynamicUniverse('HS300')
    capital_base = 1000000
    freq = 'd'
    refresh_rate = 1

    buy_threshold = -100  # 买入阈值
    sell_threshold = 100  # 卖出阈值

    accounts = {
        'fantasy_account': AccountConfig(account_type='security', capital_
base=10000000)
    }

    # 定义并注册 Signal 模块
    def initialize(context):
        cci = Signal('CCI20')
        # 优矿提供了 CCI5、CCI10、CCI20、CCI88 等 4 种不同参数的预计算 CCI
        context.signal_generator = SignalGenerator(cci)

    def handle_data(context):
        current_universe = context.get_universe(exclude_halt=True)
        cci = context.signal_result['CCI20']

        buylist = []
        account = context.get_account('fantasy_account')
        cash = account.cash
        current_positions = account.get_positions()
        for stock in current_universe:
            # CCI 处于超卖期，且无持仓
            if cci[stock] < buy_threshold and stock not in current_positions:
                buylist.append(stock)

            # CCI 处于超买期，且有持仓
            elif cci[stock] > sell_threshold and stock in current_positions:
                order_to(stock, 0)  # 全部卖出
```

```
                cash += current_positions[stock].amount * context.current_
price(stock)  # 估计买入金额
        d = min(len(buylist), int(cash) // 20000)
    # 可以买入的股票数量，如果资金不够，则只买入其中一部分
        for stock in buylist[:d]:
            order(stock, 20000 / context.current_price(stock))
```

两种方法的运行结果如图 6-23 所示。

图 6-23

从上述结果可以看出，基于两种平台的技术指标策略结果基本一致。读者可根据自身的习惯尝试直接调用通联数据技术指标库，或者基于其他数据服务商进行技术指标计算，并在通联数据平台上进行策略调试。

6.7.4　CMO 指标

1. 指标介绍

钱德动量摆动指标（CMO）由图莎尔•钱德（Tushar S.Chande）提出，利用上涨日和下跌日数据寻找极度超买和极度超卖的条件。作为动量指标，CMO 指标用来衡量证券趋势强度，以±50 为区间界限来衡量市场状态。

2. 计算公式

计算 CMO 指标通常需要设置计算周期 n，一般取 n=20，数值的具体计算方式为

$$CMO_n = \frac{\sum_{i=1}^{n} up_i - \sum_{i=1}^{n} dn_i}{\sum_{i=1}^{n} up_i + \sum_{i=1}^{n} dn_i} \times 100$$

其中，计算周期内每一天的收盘价减去前收盘价的结果，如果结果为正，则将其赋给 up_i（dn_i 为 0）；如果结果为负，则将其绝对值赋给 dn_i（up_i 为 0）。

3. 指标使用说明

在对 CMO 指标进行分析时，其波动范围是-100～+100，因此我们通常有具体的定量规则对市场进行研判。

（1）CMO 指标的绝对值越高，趋势越强。趋近于 0 则表示标的证券趋向于在早水平方向波动。

（2）一般认为在 CMO 指标的值低于-50 时处于超卖状态，在高于+50 时处于超买状态，且两者的动量值为 3 倍之差。

6.8.4.4　指标回测

下面构建基于 CMO 指标的简单策略，具体的策略要求如下。

（1）如果 CMO 指标低于-50，则买入；如果 CMO 指标高于+50，则卖出。

（2）等权买入符合条件的股票，如果可选的股票少于 10 只，则每只买入 0.1，在出现重复的信号时不重复买入。

（3）如果股票的数量超过可用资金，则随机买入其中一部分，保证资金用完。

（4）回测时间为 2012 年 1 月 1 日至 2017 年 12 月 31 日，基准为 HS300，股票池为动态 HS300 成分股，每日调仓。

Uqer 方法的代码如下：

```
    import numpy as np
    import pandas as pd

    start = '2012-01-01'                    # 回测起始时间
    end = '2017-12-31'                      # 回测结束时间
    universe = DynamicUniverse('HS300')
    benchmark = 'HS300'
    capital_base = 1000000
    freq = 'd'
    refresh_rate = 1

    buy_threshold = -50
    sell_threshold = 50

    accounts = {
        'fantasy_account': AccountConfig(account_type='security', capital_
base=10000000)
    }

    # 定义并注册 Signal 模块
    def initialize(context):
        context.signal_generator = SignalGenerator(Signal('CMO'))

    def handle_data(context):
        account = context.get_account('fantasy_account')
        current_universe = context.get_universe(exclude_halt=True)
        current_positions = account.get_positions()
        buylist = []
        cash = account.cash
        for sec in current_universe:
            if context.signal_result['CMO'][sec] < buy_threshold and sec not in
current_positions:  # CMO 处于超卖期，且无持仓
                buylist.append(sec)
            elif context.signal_result['CMO'][sec] > sell_threshold and sec in
current_positions:  # CMO 处于超买期，且有持仓
```

```
                order_to(sec, 0)  # 全部卖出
                cash+=current_positions[sec].amount * context.current_price(sec)
# 估计买入金额

        if len(buylist)==0:
            weight = 0
        else:
            weight = min(1.0/len(buylist), 0.1)
        for sec in buylist:
            order_pct_to(sec, weight)
```

从运行结果可以看出，CMO 指标的整体表现略次于 HS300 指数，但在控制回撤方面不如指数，投资者可考虑修改离场条件，以期稳中求胜。

6.7.5　Chaikin Oscillator 指标

1. 指标介绍

佳庆指标（Chaikin Oscillator）由 Marc Chaikin 提出，是一种反映市场内在动能的动量指标。在前面提到的动量指标都是基于价格信息进行计算的，但成交量同样是一个不能忽略的重要信息来源，可以反映股价在本质上的强弱度。由于根据当日涨跌与否来将当日成交量全部看作多头或者空头的力量过于简单，所以 Chaikin Oscillator 指标针对这种方法进行了改进，计算出乘数因子和能量区间，进一步计算出 ADL 动量指标，再计算 Chaikin Oscillator 指标，并根据其符号的变化对涨跌趋势进行判断。

2. 计算公式

计算 Chaikin Oscillator 指标的方法略显复杂，每个标的证券在每一天都可计算出 Chaikin Oscillator 指标，主要公式为

$$\alpha=Money\ Flow\ Multiplier=[(Close-Low)-(High-Close)]/(High-Low)$$

$$V=Money\ Flow\ Volume=Money\ Flow\ Multiplier\times Volume\ for\ the\ Period$$

$$ADL=Previous\ ADL+Current\ Period's\ Money\ Flow\ Volume$$

$$Chaikin\ Oscillator=(3day\ EMA\ of\ ADL)-(10day\ EMA\ of\ ADL)$$

该公式看起来略显复杂，我们可以逐行解读。其中，α 为两倍收盘价与当日最高价和或者最低价和之差所占当日波动的比例，可以看作一个日内波动幅度系数。V 为改进后的日动量值，计算方法为 α 与当日总成交量的乘积。*ADL* 为前一日的 *ADL* 与当日 V 之和。最后便可求出 Chaikin Oscillator 指标，即 3 日 *ADL* 移动平均与 10 日移动平均之差。

3. 指标使用说明

在对 Chaikin Oscillator 指标进行分析时，由于其本身为短线与长线之差，因此我们可以通过其符号的变化对市场进行研判。

（1）Chaikin Oscillator 指标由负变正时，标的资产进入上涨趋势；Chaikin Oscillator 指标由正变负时，标的资产进入下跌趋势。

（2）Chaikin Oscillator 指标因个股的不同而不同，因此需要对历史指标值的变化情况进行观察，判断其规律性的超买超卖界限，从而判断趋势情况。

4. 指标回测

下面构建基于 Chaikin Oscillator 指标的简单策略，具体的策略要求如下。

（1）如果 Chaikin Oscillator 指标由负变正，则买入；如果 Chaikin Oscillator 指标由正变负，则卖出。

（2）等权买入符合条件的股票，如果可选的股票少于 10 只，则每只买入 0.1，在出现重复的信号时不重复买入。

（3）如果股票数量高于可用资金，则随机买入其中一部分，保证资金用完。

（4）回测时间为 2012 年 1 月 1 日至 2017 年 12 月 31 日，基准为 HS300，股票池为动态 HS300 成分股，每日调仓。

Uqer 方法的代码如下：

```
    import numpy as np
    import pandas as pd

    start = '2012-01-01'
    end = '2017-12-31'
    benchmark = 'HS300'
    universe = set_universe('HS300', start)  # 取 start 日的成分股，防止 survival
bias
    capital_base = 1000000
    freq = 'd'
    refresh_rate = 1

    def cross_situation(data, dependencies=['ChaikinOscillator'], max_window=
2):
        cross = {}
        for sec in data['ChaikinOscillator'].columns:
            if data['ChaikinOscillator'][sec][0] < 0 and
data['ChaikinOscillator'][sec][1] > 0:  # 上穿
                cross[sec] = 1
            elif data['ChaikinOscillator'][sec][0] > 0 and
data['ChaikinOscillator'][sec][1] < 0:  # 下穿
                cross[sec] = -1
            else:
                cross[sec] = 0
        return pd.Series(cross)

    accounts = {
        'fantasy_account': AccountConfig(account_type='security',
capital_base=10000000)
    }

    # 定义并注册 Signal 模块
    def initialize(context):
        cho = Signal('ChaikinOscillator')
        cross = Signal('CRX', cross_situation)
        context.signal_generator = SignalGenerator(cho, cross)
```

```
    def handle_data(context):
        current_universe = context.get_universe(exclude_halt=True)

        buylist = []
        account = context.get_account('fantasy_account')
        current_positions = account.get_positions()
        cash = account.cash

        for sec in current_universe:
            if context.signal_result['CRX'][sec] == 1 and sec not in
current_positions:  # Chaikin 上穿 0 线，且无持仓
                buylist.append(sec)
            elif context.signal_result['CRX'][sec] == -1 and sec in
current_positions:  # Chaikin 下穿 0 线，且有持仓
                order_to(sec, 0)  # 全部卖出
                cash += current_positions[sec].amount *
context.current_price(sec)  # 估计买入金额

        if len(buylist)==0:
            weight = 0
        else:
            weight = min(1.0/len(buylist), 0.1)
        for sec in buylist:
            order_pct_to(sec, weight)
```

可以看出，本技术指标的策略收益明显不如 HS300 指数，年化收益仅为-0.3%，无法有效跟上市场收益。此技术指标的运用仍需进一步改进或者优化。

6.7.6　DMI 指标

1. 指标介绍

动向指标（DMI）是由美国技术分析大师威尔斯·威尔德（Wells Wilder）设计的，用于衡量股票价格在涨跌过程中买卖双方的力量均衡点，从而对中长期股市趋势进行判断。相对于其他类似的指标，DMI 指标可将每日的具体振幅考虑在内，而非简单地考量日间的累计波动，这种统计方式更精确地度量了多空双方的力量变化。

2. 计算公式

在计算 DMI 指标时，通常涉及 4 个中间指标的计算：*+DI*、*-DI*（多空指标）、*ADX* 和 *ADXR*（趋向指标），各指标的具体计算方式如下。

首先，计算上升和下降动向：

$$\text{上升动向}+DM_t=t\text{ 日最高价}-(t-1)\text{日最高价}$$

$$\text{下降动向}-DM_t=(t-1)\text{日最低价}-t\text{ 日最低价}$$

然后，计算真实波幅 *TR*：

$TR_t=\max(|t\text{ 日最高}-t\text{ 日最低}|,|t\text{ 日最高}-(t-1)\text{日收盘}|,|t\text{ 日最低}-(t-1)\text{日收盘}|)$

其次，综合*+DM*、*-DM*、*TR* 来计算 *DI*：

$$\pm DI_t=\sum_{k=0}^{N-1}\pm DM_{t-k}\Big/\sum_{k=0}^{N-1}TR_{t-k}\times 100$$

最后，计算 *ADX* 与 *ADXR*：

$$ADX_t=MA\left(\frac{(+DI_t)-(-DI_t)}{(+DI_t)+(-DI_t)}\times 100,N\right)$$

$$ADXR=(ADX_t+ADX_{t-1})/2$$

3. 对指标的使用说明

在对 DMI 指标进行分析时，其本身包含多条指标线，因此我们可以从多空及趋势这两个方面对市场进行研判。

（1）当+DI 位于-DI 之上时，行情以上涨为主，且当+DI 上穿-DI 时，为买进信号；当+DI 位于-DI 之下时，行情以下跌为主，且当+DI 下穿-DI 时，为卖出信号。

（2）当+DI 从 20 之下升到 50 之上时，有一定的上涨行情；当-DI 从 20 之下升到 50 之上时，有一定下跌行情；当±DI 以 20 为基准线上下波动时，行情以整理为主。

（3）当 ADX 上穿 ADXR 为金叉触发点时，预示着上涨行情的开始；当 ADX 下穿 ADXR 为死叉触发点时，预示着上涨行情的结束。

（4）当 ADX 与 ADXR 在 20 左右波动时，行情以整理为主；当 ADX 在 50 之上反转时，行情有很大可能开始反转；当 ADX 与 ADXR 上行至 80 以上时，市场可能出现大行情。

4. 指标回测

下面构建基于 DMI 指标的简单策略，具体的策略要求如下。

（1）如果 ADX 指标高于 50 且+DI>-DI，则买入；如果 ADX 指标低于 20 或+DI<-DI，则卖出。

（2）等权买入符合条件的股票，如果可选的股票少于 10 只，则每只买入 0.1，在出现重复的信号时不重复买入。

（3）如果股票数量高于可用资金，则随机买入其中一部分，保证资金用完。

（4）回测时间为 2012 年 1 月 1 日至 2017 年 12 月 31 日，基准为 HS300，股票池为动态 HS300 成分股，每日调仓。

策略代码如下：

```
import numpy as np
import pandas as pd

start = '20120101'
end = '20171231'
benchmark = 'HS300'
universe = set_universe('HS300', start)  # 取start日的成分股防止survival bias
capital_base = 1000000
freq = 'd'
refresh_rate = 1
```

```
    buy_threshold = 50  # 买入阈值
    sell_threshold = 20  # 卖出阈值

    accounts = {
        'fantasy_account': AccountConfig(account_type='security', capital_
base=10000000)
    }

    # 定义并注册 Signal 模块
    def initialize(context):
        # 优矿的 PlusDI、MinusDI、ADX 的参数都是 14
        pdi = Signal('plusDI')
        mdi = Signal('minusDI')
        adx = Signal('ADX')
        context.signal_generator = SignalGenerator(pdi, mdi, adx)

    def handle_data(context):
        account = context.get_account('fantasy_account')
        buylist = []
        #cash = account.cash
        security_position = account.get_positions()
        for sec in context.universe:
            if (context.signal_result['ADX'][sec] > buy_threshold and
context.signal_result['plusDI'][sec] > context.signal_result['minusDI'][sec])
and sec not in context.security_position:  # 存在趋势同时收盘价上穿上轨
                buylist.append(sec)
            elif (context.signal_result['ADX'][sec] < sell_threshold or
context.signal_result['plusDI'][sec] < context.signal_result['minusDI'][sec])
and sec in context.security_position:  # 趋势消失或收盘价下穿下轨，且有持仓
                order_to(sec, 0)  # 全部卖出
                #cash += security_position[sec].amount * context.current_price
(sec)  # 估计买入金额
        if len(buylist)==0:
            weight = 0
        else:
            weight = min(1.0/len(buylist), 0.1)
        for sec in buylist:
            order_pct_to(sec, weight)
```

从策略的运行结果可以看出，该策略的表现不足以令人满意，年化收益仅为 3.9%，鉴于每期符合条件的股票数量过少，所以投资者在选择应用时可考虑适当修改条件。

6.7.7 优矿平台因子汇总

为了方便读者查找及使用所需的技术指标因子，我们在此列出通联数据优矿平台提供的所有因子，读者可直接进行调用或者进一步根据个人需求进行因子合成，从而实现在收益和风险等方面的掌控。

优矿提供的因子（共 424 个）如表 6-4 所示。

表 6-4

因子符号	类　别	名　称
DAVOL10	成交量性因子	10 日平均换手率与 120 日平均换手率
DAVOL20	成交量性因子	20 日平均换手率与 120 日平均换手率
DAVOL5	成交量性因子	5 日平均换手率与 120 日平均换手率
KlingerOscillator	成交量性因子	成交量摆动指标
MoneyFlow20	成交量性因子	20 日资金流量
OBV	成交量性因子	能量潮指标
OBV20	成交量性因子	20 日能量潮指标
OBV6	成交量性因子	6 日能量潮指标
STOA	成交量性因子	12 个月的月均换手率对数
STOM	成交量性因子	月度换手率对数

续表

因子符号	类　别	名　称
STOQ	成交量性因子	3 个月的月均换手率对数
TVMA20	成交量性因子	20 日成交金额的移动平均值
TVMA6	成交量性因子	6 日成交金额的移动平均值
TVSTD20	成交量性因子	20 日成交金额的标准差
TVSTD6	成交量性因子	6 日成交金额的标准差
VDEA	成交量性因子	VDEA
VDIFF	成交量性因子	VDIFF
VEMA10	成交量性因子	成交量的 10 日指数移动平均
VEMA12	成交量性因子	成交量的 12 日指数移动平均
VEMA26	成交量性因子	成交量的 26 日指数移动平均
VEMA5	成交量性因子	成交量的 5 日指数移动平均
VMACD	成交量性因子	成交量量指数平滑异同移动平均线
VOL10	成交量性因子	10 日平均换手率
VOL120	成交量性因子	120 日平均换手率
VOL20	成交量性因子	20 日平均换手率
VOL240	成交量性因子	240 日平均换手率
VOL5	成交量性因子	5 日平均换手率
VOL60	成交量性因子	60 日平均换手率
VOSC	成交量性因子	成交量震荡
VR	成交量性因子	成交量比率
VROC12	成交量性因子	12 日量变动速率指标
VROC6	成交量性因子	6 日量变动速率指标
VSTD10	成交量性因子	10 日成交量标准差
VSTD20	成交量性因子	20 日成交量标准差
ASSI	股指与市值类因子	对数总资产
CETOP	股指与市值类因子	现金收益滚动收益与市值比
ForwardPE	股指与市值类因子	动态 PE
LCAP	股指与市值类因子	对数市值
LFLO	股指与市值类因子	对数流通市值

续表

因子符号	类　别	名　称
MktValue	股指与市值类因子	总市值
NIAP	股指与市值类因子	归属于母公司所有者的净利润
NLSIZE	股指与市值类因子	非线性市值
NegMktValue	股指与市值类因子	流通市值
PB	股指与市值类因子	市净率
PBIndu	股指与市值类因子	PBIndu
PCF	股指与市值类因子	市现率
PCFIndu	股指与市值类因子	PCFIndu
PE	股指与市值类因子	市盈率
PEG3Y	股指与市值类因子	PEG3Y
PEG5Y	股指与市值类因子	PEG5Y
PEHist120	股指与市值类因子	PE/过去 6 个月 PE 的均值
PEHist20	股指与市值类因子	PE/过去 1 个月 PE 的均值
PEHist250	股指与市值类因子	PE/过去 1 年 PE 的均值
PEHist60	股指与市值类因子	PE/过去 3 个月 PE 的均值
PEIndu	股指与市值类因子	PEIndu
PS	股指与市值类因子	市销率
PSIndu	股指与市值类因子	PSIndu
SGRO	股指与市值类因子	5 年营业收入增长率
StaticPE	股指与市值类因子	静态 PE
TA2EV	股指与市值类因子	资产总计与企业价值之比
TEAP	股指与市值类因子	归属于母公司所有者权益
TotalAssets	股指与市值类因子	总资产
AD	趋势类因子	累积/派发线
AD20	趋势类因子	20 日累积/派发线
AD6	趋势类因子	6 日累积/派发线
ADX	趋势类因子	平均动向指数
ADXR	趋势类因子	相对平均动向指数
ASI	趋势类因子	累计振动升降指标

续表

因 子 符 号	类　　别	名　　称
Aroon	趋势类因子	阿隆指标
AroonDown	趋势类因子	下降阿隆
AroonUp	趋势类因子	上升阿隆
ChaikinOscillator	趋势类因子	佳庆指标
ChaikinVolatility	趋势类因子	佳庆离散指标
CoppockCurve	趋势类因子	估波指标
DDI	趋势类因子	方向标准离差指数
DEA	趋势类因子	DEA
DHILO	趋势类因子	波幅中位数
DIF	趋势类因子	DIF
DIFF	趋势类因子	DIFF
DIZ	趋势类因子	DIZ
EMV14	趋势类因子	14 天简易波动指标
EMV6	趋势类因子	6 天简易波动指标
Hurst	趋势类因子	赫斯特指数
MA10RegressCoeff12	趋势类因子	10 日价格平均线的 12 日线性回归系数
MA10RegressCoeff6	趋势类因子	10 日价格平均线的 6 日线性回归系数
MACD	趋势类因子	平滑异同移动平均线
MTM	趋势类因子	动量指标
MTMMA	趋势类因子	10 日 mtm 均值
PLRC12	趋势类因子	12 日价格线性回归系数
PLRC6	趋势类因子	6 日价格线性回归系数
PVT	趋势类因子	价量趋势指标
PVT12	趋势类因子	12 日 PVT 均值
PVT6	趋势类因子	6 日 PVT 均值
SwingIndex	趋势类因子	振动升降指标
TRIX10	趋势类因子	10 日三重指数平滑移动平均指标变化率
TRIX5	趋势类因子	5 日三重指数平滑移动平均指标变化率
UOS	趋势类因子	终极指标

续表

因子符号	类　别	名　称
Ulcer10	趋势类因子	10 日 Ulcer
Ulcer5	趋势类因子	5 日 Ulcer
minusDI	趋势类因子	下降指标
plusDI	趋势类因子	上升指标
BLEV	偿债能力资本结构因子	账面杠杆
BondsPayableToAsset	偿债能力资本结构因子	应付债券与总资产之比
CurrentAssetsRatio	偿债能力资本结构因子	流动资产比率
CurrentRatio	偿债能力资本结构因子	流动比率
DebtEquityRatio	偿债能力资本结构因子	产权比率
DebtTangibleEquityRatio	偿债能力资本结构因子	负债合计/有形净值，有形净值债务率
DebtsAssetRatio	偿债能力资本结构因子	债务总资产比
EquityFixedAssetRatio	偿债能力资本结构因子	股东权益与固定资产比率
EquityToAsset	偿债能力资本结构因子	股东权益比率
FixAssetRatio	偿债能力资本结构因子	固定资产比率
IntangibleAssetRatio	偿债能力资本结构因子	无形资产比率
InteBearDebtToTotalCapital	偿债能力资本结构因子	带息负债/全部投入资本
InterestCover	偿债能力资本结构因子	利息保障倍数
LongDebtToAsset	偿债能力资本结构因子	长期借款与资产总计之比
LongDebtToWorkingCapital	偿债能力资本结构因子	长期负债与营运资金比率
LongTermDebtToAsset	偿债能力资本结构因子	长期负债与资产总计之比
MLEV	偿债能力资本结构因子	市场杠杆
NOCFToInterestBearDebt	偿债能力资本结构因子	经营活动产生现金流量净额/带息负债
NOCFToNetDebt	偿债能力资本结构因子	经营活动产生现金流量净额/净债务
NOCFToTLiability	偿债能力资本结构因子	经营活动产生的现金流量净额/负债合计
NonCurrentAssetsRatio	偿债能力资本结构因子	非流动资产比率
QuickRatio	偿债能力资本结构因子	速动比率
SuperQuickRatio	偿债能力资本结构因子	超速动比率
TSEPToInterestBearDebt	偿债能力资本结构因子	归属母公司股东的权益/带息负债
TSEPToTotalCapital	偿债能力资本结构因子	归属于母公司所有者权益合计/全部投入资本

续表

因 子 符 号	类　　别	名　　称
TangibleAToInteBearDebt	偿债能力资本结构因子	有形净值/带息负债
TangibleAToNetDebt	偿债能力资本结构因子	有形净值/净债务
Alpha120	收益类因子	120 日 Alpha
Alpha20	收益类因子	20 日 Alpha
Alpha60	收益类因子	60 日 Alpha
Beta120	收益类因子	120 日 Beta
Beta20	收益类因子	20 日 Beta
Beta252	收益类因子	252 日 Beta
Beta60	收益类因子	60 日 Beta
CmraCNE5	收益类因子	12 月累计收益
DASTD	收益类因子	252 日超额收益标准差
GainLossVarianceRatio120	收益类因子	120 日收益损失方差比
GainLossVarianceRatio20	收益类因子	20 日收益损失方差比
GainLossVarianceRatio60	收益类因子	60 日收益损失方差比
GainVariance120	收益类因子	120 日正向收益方差
GainVariance20	收益类因子	20 日正向收益方差
GainVariance60	收益类因子	60 日正向收益方差
HsigmaCNE5	收益类因子	252 日残差收益波动率
InformationRatio120	收益类因子	120 日信息比率
InformationRatio20	收益类因子	20 日信息比率
InformationRatio60	收益类因子	60 日信息比率
Kurtosis120	收益类因子	120 日收益峰度
Kurtosis20	收益类因子	20 日收益峰度
Kurtosis60	收益类因子	60 日收益峰度
LossVariance120	收益类因子	120 日损失方差
LossVariance20	收益类因子	20 日损失方差
LossVariance60	收益类因子	60 日损失方差
RealizedVolatility	收益类因子	实际波动率，日内 5 分钟线的收益率标准差
SharpeRatio120	收益类因子	120 日夏普比率

续表

因子符号	类　别	名　称
SharpeRatio20	收益类因子	20 日夏普比率
SharpeRatio60	收益类因子	60 日夏普比率
TreynorRatio120	收益类因子	120 日特诺雷比率
TreynorRatio20	收益类因子	20 日特诺雷比率
TreynorRatio60	收益类因子	60 日特诺雷比率
Variance120	收益类因子	120 日收益方差
Variance20	收益类因子	20 日收益方差
Variance60	收益类因子	60 日收益方差
AdminiExpenseRate	赢利能力收益质量	管理费用与营业总收入之比
EBITToTOR	赢利能力收益质量	息税前利润与营业总收入之比
EGRO	赢利能力收益质量	5 年收益增长率
ETOP	赢利能力收益质量	收益市值比
ETP5	赢利能力收益质量	5 年平均收益市值比
FinancialExpenseRate	赢利能力收益质量	财务费用与营业总收入之比
GrossIncomeRatio	赢利能力收益质量	销售毛利率
InvestRAssociatesToTP	赢利能力收益质量	对联营和营公司投资收益/利润总额 TTM
InvestRAssociatesToTPLatest	赢利能力收益质量	对联营和营公司投资收益/利润总额（Latest）
NPCutToNP	赢利能力收益质量	扣除非经常损益后的净利润/净利润
NPToTOR	赢利能力收益质量	净利润与营业总收入之比
NetNonOIToTP	赢利能力收益质量	营业外收支净额占利润总额之比 TTM
NetNonOIToTPLatest	赢利能力收益质量	营业外收支净额占利润总额之比
NetProfitRatio	赢利能力收益质量	销售净利率
OperatingExpenseRate	赢利能力收益质量	营业费用与营业总收入之比
OperatingNIToTP	赢利能力收益质量	经营活动净收益/利润总额 TTM
OperatingNIToTPLatest	赢利能力收益质量	经营活动净收益/利润总额（Latest）
OperatingProfitRatio	赢利能力收益质量	营业利润率
OperatingProfitToTOR	赢利能力收益质量	营业利润与营业总收入之比
PeriodCostsRate	赢利能力收益质量	销售期间费用率
ROA	赢利能力收益质量	资产回报率

续表

因 子 符 号	类　　别	名　　称
ROA5	赢利能力收益质量	5 年资产回报率
ROAEBIT	赢利能力收益质量	总资产报酬率
ROAEBITTTM	赢利能力收益质量	总资产报酬率（TTM）
ROE	赢利能力收益质量	权益回报率
ROE5	赢利能力收益质量	5 年权益回报率
ROEAvg	赢利能力收益质量	净资产收益率（平均）
ROECut	赢利能力收益质量	净资产收益率（扣除摊薄）
ROECutWeighted	赢利能力收益质量	净资产收益率（扣除加权平均，公布值）
ROEDiluted	赢利能力收益质量	净资产收益率（摊薄）
ROEWeighted	赢利能力收益质量	净资产收益率（加权平均，公布值）
ROIC	赢利能力收益质量	投入资本回报率
SUE	赢利能力收益质量	未预期盈余
SUOI	赢利能力收益质量	未预期毛利
SalesCostRatio	赢利能力收益质量	销售成本率
TaxRatio	赢利能力收益质量	销售税金率
TotalProfitCostRatio	赢利能力收益质量	成本费用利润率
AR	能量型因子	人气指标
ARBR	能量型因子	人气指标
BR	能量型因子	意愿指标
BearPower	能量型因子	空头力道
BullPower	能量型因子	多头力道
CR20	能量型因子	CR20
Elder	能量型因子	艾达透视指标
JDQS20	能量型因子	阶段强势指标
MassIndex	能量型因子	梅斯线
NVI	能量型因子	负成交量指标
PSY	能量型因子	心理线指标
PVI	能量型因子	正成交量指标
RC12	能量型因子	12 日变化率指数

续表

因子符号	类　别	名　称
RC24	能量型因子	24 日变化率指数
RSTR12	能量型因子	12 月相对强势
RSTR24	能量型因子	24 月相对强势
RSTR504	能量型因子	504 天相对强势
TOBT	能量型因子	超额流动
CapitalSurplusFundPS	每股指标类因子	每股资本公积金
CashDividendCover	每股指标类因子	现金股利保障倍数
CashEquivalentPS	每股指标类因子	每股现金余额
CashFlowPS	每股指标类因子	每股现金流 TTM
DilutedEPS	每股指标类因子	稀释每股收益
DividendCover	每股指标类因子	股利保障倍数
DividendPS	每股指标类因子	每股股利
DividendPaidRatio	每股指标类因子	股利支付率
EPS	每股指标类因子	基本每股收益
EPSTTM	每股指标类因子	每股收益 TTM
EnterpriseFCFPS	每股指标类因子	每股企业自由现金流量
NetAssetPS	每股指标类因子	每股净资产
OperCashFlowPS	每股指标类因子	每股经营现金流 TTM
OperatingProfitPS	每股指标类因子	每股营业利润 TTM
OperatingProfitPSLatest	每股指标类因子	每股营业利润
OperatingRevenuePS	每股指标类因子	每股营业收入 TTM
OperatingRevenuePSLatest	每股指标类因子	每股营业收入
RetainedEarningRatio	每股指标类因子	留存盈余比率
RetainedEarningsPS	每股指标类因子	每股留存收益
ShareholderFCFPS	每股指标类因子	每股股东自由现金流量
SurplusReserveFundPS	每股指标类因子	每股盈余公积金
TORPS	每股指标类因子	每股营业总收入 TTM
TORPSLatest	每股指标类因子	每股营业总收入
UndividedProfitPS	每股指标类因子	每股未分配利润

续表

因 子 符 号	类　别	名　称
ADTM	超买超卖型因子	动态买卖气指标
ARC	超买超卖型因子	变化率指数均值
ATR14	超买超卖型因子	14 日均幅指标
ATR6	超买超卖型因子	6 日均幅指标
BIAS10	超买超卖型因子	10 日乖离率
BIAS20	超买超卖型因子	20 日乖离率
BIAS5	超买超卖型因子	5 日乖离率
BIAS60	超买超卖型因子	60 日乖离率
BackwardADJ	超买超卖型因子	股价向后复权因子
BollDown	超买超卖型因子	布林下轨线
BollUp	超买超卖型因子	布林上轨线
CCI10	超买超卖型因子	10 日顺势指标
CCI20	超买超卖型因子	20 日顺势指标
CCI5	超买超卖型因子	5 日顺势指标
CCI88	超买超卖型因子	88 日顺势指标
CMO	超买超卖型因子	钱德动量摆动指标
CMRA	超买超卖型因子	24 月累计收益
ChandeSD	超买超卖型因子	ChandeSD
ChandeSU	超买超卖型因子	ChandeSU
DBCD	超买超卖型因子	异同离差乖离率
DDNBT	超买超卖型因子	下跌贝塔
DDNCR	超买超卖型因子	下跌相关系数
DDNSR	超买超卖型因子	下跌波动
DVRAT	超买超卖型因子	收益相对波动
DownRVI	超买超卖型因子	DownRVI
FiftyTwoWeekHigh	超买超卖型因子	当前价格处于过去 1 年股价的位置
HBETA	超买超卖型因子	历史贝塔
HSIGMA	超买超卖型因子	历史波动
ILLIQUIDITY	超买超卖型因子	收益相对金额比

续表

因子符号	类　别	名　称
KDJ_D	超买超卖型因子	随机指标 D
KDJ_J	超买超卖型因子	随机指标 J
KDJ_K	超买超卖型因子	随机指标 K
MAWVAD	超买超卖型因子	因子 WVAD 的 6 日均值
MFI	超买超卖型因子	资金流量指标
Price1M	超买超卖型因子	Price1M
Price1Y	超买超卖型因子	Price1Y
Price3M	超买超卖型因子	Price3M
REVS10	超买超卖型因子	10 日价格动量
REVS120	超买超卖型因子	120 日价格动量
REVS20	超买超卖型因子	20 日价格动量
REVS20Indu1	超买超卖型因子	20 日收益与其行业均值之差
REVS250	超买超卖型因子	250 日价格动量
REVS5	超买超卖型因子	5 日价格动量
REVS5Indu1	超买超卖型因子	5 日收益与其行业均值之差
REVS5m20	超买超卖型因子	5 日和 20 日价格动量差
REVS5m60	超买超卖型因子	5 日和 60 日价格动量差
REVS60	超买超卖型因子	60 日价格动量
REVS750	超买超卖型因子	750 日价格动量
ROC20	超买超卖型因子	20 日变动速率
ROC6	超买超卖型因子	6 日变动速率
RSI	超买超卖型因子	相对强弱指标
RVI	超买超卖型因子	相对离散指数
Rank1M	超买超卖型因子	Rank1M
SBM	超买超卖型因子	SBM
SRMI	超买超卖型因子	修正动量指标
STM	超买超卖型因子	STM
Skewness	超买超卖型因子	股价偏度
UpRVI	超买超卖型因子	UpRVI

续表

因子符号	类　别	名　称
Volatility	超买超卖型因子	换手率相对波动率
Volumn1M	超买超卖型因子	Volumn1M
Volumn3M	超买超卖型因子	Volumn3M
WVAD	超买超卖型因子	威廉变异离散量
ARTDays	运营能力因子	应收账款周转天数
ARTRate	运营能力因子	应收账款周转率
AccountsPayablesTDays	运营能力因子	应付账款周转天数
AccountsPayablesTRate	运营能力因子	应付账款周转率
CashConversionCycle	运营能力因子	现金转换周期
CurrentAssetsTRate	运营能力因子	流动资产周转率
EquityTRate	运营能力因子	股东权益周转率
FixedAssetsTRate	运营能力因子	固定资产周转率
InventoryTDays	运营能力因子	存货周转天数
InventoryTRate	运营能力因子	存货周转率
OperatingCycle	运营能力因子	营业周期
TotalAssetsTRate	运营能力因子	总资产周转率
ACD20	均线型因子	20 日收集派发指标
ACD6	均线型因子	6 日收集派发指标
APBMA	均线型因子	绝对偏差移动平均
BBI	均线型因子	多空指数
BBIC	均线型因子	BBIC
EMA10	均线型因子	10 日指数移动均线
EMA12	均线型因子	12 日指数移动均线
EMA120	均线型因子	120 日指数移动均线
EMA20	均线型因子	20 日指数移动均线
EMA26	均线型因子	26 日指数移动均线
EMA5	均线型因子	5 日指数移动均线
EMA60	均线型因子	60 日指数移动均线
MA10	均线型因子	10 日移动均线

续表

因子符号	类　别	名　称
MA10Close	均线型因子	均线价格比
MA120	均线型因子	120 日移动均线
MA20	均线型因子	20 日移动均线
MA5	均线型因子	5 日移动均线
MA60	均线型因子	60 日移动均线
TEMA10	均线型因子	10 日三重指数移动平均线
TEMA5	均线型因子	5 日三重指数移动平均线
DEGM	成长能力类因子	毛利率增长
EARNMOM	成长能力类因子	八季度净利润变化趋势
NPParentCompanyCutYOY	成长能力类因子	归属母公司股东的净利润（扣除非经常损益）同比增长（%）
NPParentCompanyGrowRate	成长能力类因子	归属母公司股东的净利润增长率
NetAssetGrowRate	成长能力类因子	净资产增长率
NetCashFlowGrowRate	成长能力类因子	净现金流量增长率
NetProfitGrowRate	成长能力类因子	净利润增长率
NetProfitGrowRate3Y	成长能力类因子	净利润 3 年复合增长率
NetProfitGrowRate5Y	成长能力类因子	净利润 5 年复合增长率
OperatingProfitGrowRate	成长能力类因子	营业利润增长率
OperatingRevenueGrowRate	成长能力类因子	营业收入增长率
OperatingRevenueGrowRate3Y	成长能力类因子	营业收入 3 年复合增长率
OperatingRevenueGrowRate5Y	成长能力类因子	营业收入 5 年复合增长率
TotalAssetGrowRate	成长能力类因子	总资产增长率
TotalProfitGrowRate	成长能力类因子	利润总额增长率
AdminExpenseTTM	基础科目与衍生类因子	管理费用，TTM 值
AssetImpairLossTTM	基础科目与衍生类因子	资产减值损失，TTM 值
CostTTM	基础科目与衍生类因子	营业成本，TTM 值
DA	基础科目与衍生类因子	折旧和摊销
EBIAT	基础科目与衍生类因子	息前税后利润
EBIT	基础科目与衍生类因子	息税前利润
EBITDA	基础科目与衍生类因子	息税折旧摊销前利润

续表

因 子 符 号	类　　别	名　　称
FCFE	基础科目与衍生类因子	股权自由现金流量
FCFF	基础科目与衍生类因子	企业自由现金流量
FinanExpenseTTM	基础科目与衍生类因子	财务费用，TTM 值
GrossProfit	基础科目与衍生类因子	毛利
GrossProfitTTM	基础科目与衍生类因子	毛利，TTM 值
IntCL	基础科目与衍生类因子	带息流动负债
IntDebt	基础科目与衍生类因子	带息债务
IntFreeCL	基础科目与衍生类因子	无息流动负债
IntFreeNCL	基础科目与衍生类因子	无息非流动负债
NIAPCut	基础科目与衍生类因子	扣除非经常性损益后的归属于母公司所有者权益净利润
NPFromOperatingTTM	基础科目与衍生类因子	经营活动净收益，TTM 值
NPFromValueChgTTM	基础科目与衍生类因子	价值变动净收益，TTM 值
NRProfitLoss	基础科目与衍生类因子	非经常性损益
NetDebt	基础科目与衍生类因子	净债务
NetFinanceCFTTM	基础科目与衍生类因子	筹资活动现金流量净额，TTM 值
NetIntExpense	基础科目与衍生类因子	净利息费用
NetInvestCFTTM	基础科目与衍生类因子	投资活动现金流量净额，TTM 值
NetOperateCFTTM	基础科目与衍生类因子	经营活动现金流量净额，TTM 值
NetProfitAPTTM	基础科目与衍生类因子	归属于母公司股东的净利润，TTM 值
NetProfitTTM	基础科目与衍生类因子	净利润，TTM 值
NetTangibleAssets	基础科目与衍生类因子	有形净资产/有形资产净值
NetWorkingCapital	基础科目与衍生类因子	净运营资本
NonOperatingNPTTM	基础科目与衍生类因子	营业外收支净额（营业外收入–营业外支出），TTM 值
OperateNetIncome	基础科目与衍生类因子	经营活动净收益
OperateProfitTTM	基础科目与衍生类因子	营业利润，TTM 值
RetainedEarnings	基础科目与衍生类因子	留存收益
RevenueTTM	基础科目与衍生类因子	营业收入，TTM 值
SaleServiceRenderCashTTM	基础科目与衍生类因子	销售商品提供劳务收到的现金，TTM 值
SalesExpenseTTM	基础科目与衍生类因子	销售费用，TTM 值

续表

因子符号	类　别	名　称
TCostTTM	基础科目与衍生类因子	营业总成本，TTM 值
TProfitTTM	基础科目与衍生类因子	利润总额，TTM 值
TRevenueTTM	基础科目与衍生类因子	营业总收入，TTM 值
TotalFixedAssets	基础科目与衍生类因子	固定资产合计
TotalPaidinCapital	基础科目与衍生类因子	全部投入资本
ValueChgProfit	基础科目与衍生类因子	价值变动净收益
WorkingCapital	基础科目与衍生类因子	运营资本
ACCA	现金流指标	现金流资产比和资产回报率之差
CFO2EV	现金流指标	经营活动产生的现金流量净额与企业价值之比
CTOP	现金流指标	现金流市值比
CTP5	现金流指标	5 年平均现金流市值比
CashRateOfSales	现金流指标	经营活动产生的现金流量净额与营业收入之比
CashRateOfSalesLatest	现金流指标	经营活动产生的现金流量净额与营业收入之比（Latest）
CashToCurrentLiability	现金流指标	现金比率
FinancingCashGrowRate	现金流指标	筹资活动产生的现金流量净额增长率
InvestCashGrowRate	现金流指标	投资活动产生的现金流量净额增长率
NOCFToOperatingNI	现金流指标	经营活动产生的现金流量净额与经营活动净收益之比
NOCFToOperatingNILatest	现金流指标	经营活动产生的现金流量净额与经营活动净收益之比
NetProfitCashCover	现金流指标	净利润现金含量
OperCashGrowRate	现金流指标	经营活动产生的现金流量净额增长率
OperCashInToAsset	现金流指标	总资产现金回收率
OperCashInToCurrentLiability	现金流指标	现金流动负债比
SaleServiceCashToOR	现金流指标	销售商品提供劳务收到的现金与营业收入之比
SalesServiceCashToORLatest	现金流指标	销售商品提供劳务收到的现金与营业收入之比（Latest）
DAREC	分析师预期类因子	分析师推荐评级变化
DAREV	分析师预期类因子	分析师赢利预测变化
DASREV	分析师预期类因子	分析师盈收预测变化
EPIBS	分析师预期类因子	投资回报率预测
EgibsLong	分析师预期类因子	长期赢利增长预测

续表

因 子 符 号	类　别	名　称
FEARNG	分析师预期类因子	未来预期赢利增长
FSALESG	分析师预期类因子	未来预期盈收增长
FY12P	分析师预期类因子	分析师赢利预测
GREC	分析师预期类因子	分析师推荐评级变化趋势
GREV	分析师预期类因子	分析师赢利预测变化趋势
GSREV	分析师预期类因子	分析师盈收预测变化趋势
REC	分析师预期类因子	分析师推荐评级
SFY12P	分析师预期类因子	分析师营收预测

6.8　资产配置

资产配置从投资学开设以来就是一个重要的课题，在学术上，海内外取得了重要的突破和进展。早在 19 世纪 90 年代，海外诸多研究及对冲基金的实践都表明资产配置对于投资组合的业绩贡献超过 90%。在公募 FOF 指引正式出台之后，资产配置更是成为发行 FOF 产品、开发 FOF 策略不可或缺的一部分。

在实际应用中，资产配置使用的方法相当有限，且每种方法都有其特定的局限性。大类资产配置策略大体可以分为主动策略和量化策略。

- 主动策略主要是指依据宏观经济、政策等非量化指标来对各资产的预期收益率做出判断，主动调整各类资产的配置权重，例如耶鲁模式。
- 量化策略则是指根据各资产的风险、收益等量化特征，为了达到特定的风险收益目标而进行模型构建、求解、测验等，最终得出各资产配置权重的方法，主要有经典的马科维茨均值方差模型、Black-Litterman 模型、风险平价等。

这里着重介绍资产配置的经典量化方法，包括马科维茨的有效边界、Black-Litterman 模型和风险平价。

6.8.1 有效边界

1. 投资者偏好

马科维茨均值方差模型最早提出将数理统计的方法应用到投资组合选择上，并将资产的期望收益率的波动率定义为风险。在该定义下，我们使用收益率的均值 $E(r)$和标准差 $\sigma(r)$来刻画“收益”和“风险”。

通常，我们认为人们是“风险厌恶”的，并构造如下形式的效用函数来代表投资者的投资偏好：

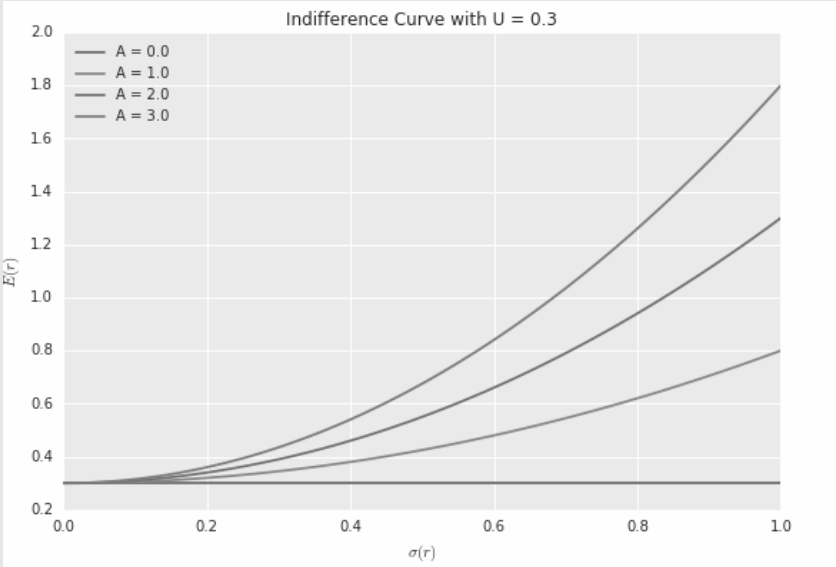
$$U(r) = E(r) - \frac{1}{2} A\sigma^2(r)$$

其中 $E(r)$表示投资组合的预期收益率，$\sigma^2(r)$表示投资组合的方差；预期收益率越高，效用值越高，收益方差越大，效用值越小。这表明投资者喜欢更高的 $E(r)$，而不喜欢高的 $\sigma^2(r)$。由于不同的投资者对于风险和收益有不同的偏好，因此在效用函数中加入风险厌恶系数参数 A 来表示投资者的不同偏好，A 越大，则表示投资者为了追求更高的收益愿意承担更小的风险，或者说该投资者需要更高的收益来补偿面临的风险。

下面的代码构造了不同的 A 对于投资者效用的影响，从输出的图形中可以看出，A 越大，投资者的相同效用值的无差异曲线越陡峭，投资者为承担风险需要的收益补偿越大：

```
import seaborn
import numpy as np
import matplotlib.pylab as pl

U = 0.3
sigmas = np.linspace(0., 1., 101)
As = [0., 1., 2., 3.]
indifference_curves = []
for A in As:
    Es = [U + 0.5 * A * sig*sig for sig in sigmas]
    indifference_curves.append(Es)

fig = pl.figure(figsize=(9, 6))
plots = []
for i, Es in enumerate(indifference_curves):
    plots.append(pl.plot(sigmas, Es, label="A = {}".format(As[i])))
pl.title("Indifference Curve with U = {}".format(U))
pl.xlabel("$\sigma(r)$")
```

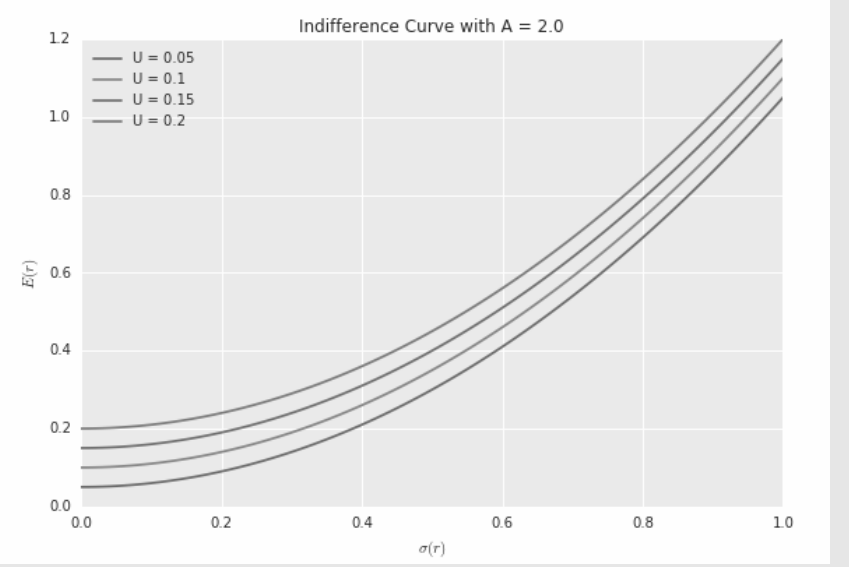
```
pl.ylabel("$E(r)$")
pl.legend(loc='best')
pl.show()
```



U 同时是 E 和 σ 的函数，所以在 σ-E 图上，对于确定的 A，不同的 U 表现为一组不相交的抛物线，这就是效用的无差异曲线，越往左上方的无差异曲线代表越高的效用，因此投资者总是偏好于位于左上方的无差异曲线上面的投资组合。下面的代码绘制出了 $A = 2$ 时的一组无差异曲线：

```
import seaborn
import numpy as np
import matplotlib.pylab as pl

A = 2
sigmas = np.linspace(0., 1., 101)
Us = [0.05, 0.1, 0.15, 0.2]
indifference_curves = []
for U in Us:
    Es = [U + 0.5 * A * sig*sig for sig in sigmas]
    indifference_curves.append(Es)

fig = pl.figure(figsize=(9, 6))
plots = []
for i, Es in enumerate(indifference_curves):
```

```
    plots.append(pl.plot(sigmas, Es, label="U = {}".format(Us[i])))
pl.title("Indifference Curve with A = {}".format(A))
pl.xlabel("$\sigma(r)$")
pl.ylabel("$E(r)$")
pl.legend(loc='best')
pl.show()
```



2. 资产组合

假设有两种资产 E_1 和 E_2，其预期收益率和方差分别为 r_1、σ_1^2 和 r_2、σ_2^2，收益率相关系数为 ρ。另有，$r_1 < r_2$、$0 < \sigma_1 < \sigma_2$。如果同时投资于两种资产，权重分别为 w_1、$1-w_1$，则组合的期望收益率和方差可表示为：

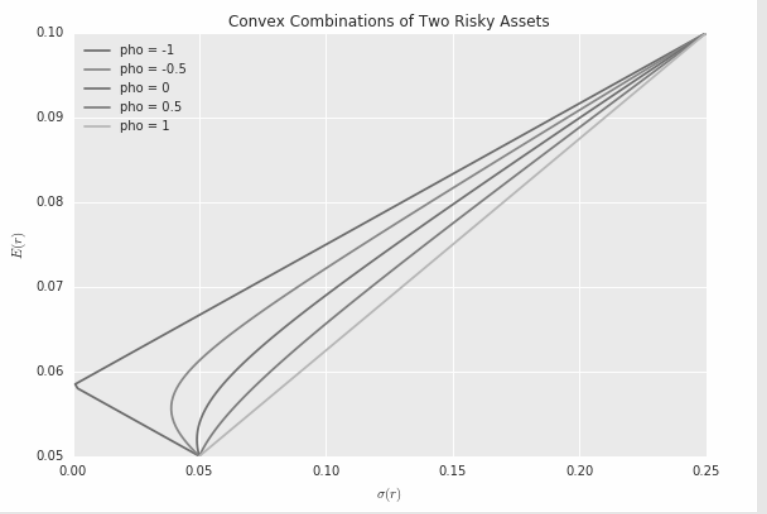
$$r = w_1 r_1 + (1-w_1) r_2$$

$$\sigma^2 = w_1^2 \sigma_1^2 + (1-w_1)^2 \sigma_2^2 + 2w_1(1-w_1)\rho\sigma_1\sigma_2$$

很容易证明，当且仅当 $\rho = 1$ 时资产组合标准差与预期收益呈线性关系。由于 ρ 的取值范围为-1～1，因此在通常情况下 $\sigma^2 = w_1^2\sigma_1^2 + (1-w_1)^2\sigma_1^2 + 2w_1(1-w_1)\rho\sigma_1\sigma_2 < (w_1\sigma_1 + (1-w_1)\sigma_2)^2$，即组合标准差小于两种资产标准差的加权平均，收益-标准差点在两种资产收益-标准差点连线的左侧。甚至在大多数情况下，当开始把波动率更大的资产 2 引入组合时，其收益波动甚至比只投资于资产 1 时更小，当经过最小方差临界点时才会慢慢增大。这也体现了投

资组合的重要性：通过分散投资以更小的组合风险获得更高的收益。如下代码显示了当投资产品只有两种时不同的相关系数对应的不同的有效边界：

```
    import seaborn
    import numpy as np
    import matplotlib.pylab as pl

    phos = [-1, -0.5, 0, 0.5, 1]
    r1, sigma1 = 0.05, 0.05
    r2, sigma2 = 0.10, 0.25

    rs = [r1*t + r2*(1-t) for t in np.linspace(0., 1., 101)]
    all_sigmas = []
    for pho in phos:
       all_sigmas.append(np.sqrt([t*t*sigma1*sigma1 + (1-t)*(1-t)*sigma2*
sigma2 + 2*pho*t*(1-t)*sigma1*sigma2 for t in np.linspace(0., 1., 101)]))

    fig = pl.figure(figsize=(9, 6))
    plots = []
    for i, sigmas in enumerate(all_sigmas):
       plots.append(pl.plot(sigmas, rs, label="pho = {}".format(phos[i])))
    pl.title("Convex Combinations of Two Risky Assets")
    pl.xlabel("$\sigma(r)$")
    pl.ylabel("$E(r)$")
    pl.legend(loc='best')
    pl.show()
```



3. 有效边界和投资组合选择

当投资者面临的可选资产大于两种时，标准差和收益的关系就不仅仅局限于一条曲线了，通过权重的选取，投资者可选的收益-标准差点就构成一个有边界的面。人们趋利避险的心理决定了理性投资人在面临同样的风险时，会选择预期收益率更高的组合；而在预期收益相同时，会选择风险较低的组合。在所有可选的预期收益-标准差点中，位于最左侧的部分构成了一条边界线，其中从最小方差点往上的部分构成了有效边界。在该边界线右下方的所有点都是无效的投资组合，没有人会选择；在该边界线左上方的所有点是不可能达到的投资组合。

在如下代码中给出了利用通联数据构造投资组合，并获取有效边界上几个特殊点的函数及相关的调用示例。其中，get_efficient_frontier()函数用于获取有效边界；draw_efficient_frontier()函数用于绘制有效边界图形；get_minimum_variance_portfolio()函数用于求解最小方差组合点；get_maximum_utility_portfolio()函数用于求解给定效用函数的效用最大化点；get_maximum_sharpe_portfolio()函数用于寻找最大夏普率点。

```
    def describe(return_table, is_print=True):
        """
        输出收益率矩阵的描述性统计量，包括：
            年化收益率
            年化标准差
            相关系数矩阵

        Args:
            return_table (DataFrame)：收益率矩阵，列为资产，值为按日期升序排列的收益率
            is_print (bool)：是否直接输出

        Returns:
            dict：描述性统计量字典，键为"annualized_return","annualized_
volatility","covariance_matrix"和"coefficient_matrix"

        Examples:
            >> describe(return_table)
            >> describe(return_table, is_print=True)
        """

        import pandas as pd
        from scipy.stats.mstats import gmean
```

```
        output = {}
        output['annualized_return']=pd.DataFrame(dict(zip(return_table.
columns,gmean(return_table+1.)**252-1.)),index=[0],columns=return_table.colu
mns)
        output['annualized_volatility']=pd.DataFrame(return_table.std()*np.
sqrt(250)).T
        output['covariance_matrix'] = return_table.cov() * 250.
        output['coefficient_matrix'] = return_table.corr()

        if is_print:
            for key, val in output.iteritems():
                print "{}:\n{}\n".format(key, val)

        return output

    def get_efficient_frontier(return_table,allow_short=False, n_samples=25):
        """
        计算 Efficient Frontier

        Args:
            return_table (DataFrame): 收益率矩阵，列为资产，值为按日期升序排列的收益率
            n_samples (int): 用于计算 Efficient Frontier 的采样点数量

        Returns:
            DataFrame: Efficient Frontier 的结果，列为"returns", "risks", "weights"
        """

        import numpy as np
        import pandas as pd
        from cvxopt import matrix, solvers

        assets = return_table.columns
        n_asset = len(assets)
        if n_asset < 2:
            raise ValueError("There must be at least 2 assets to calculate the
efficient frontier!")

        output = describe(return_table, is_print=False)
        covariance_matrix = matrix(output['covariance_matrix'].as_matrix())
        expected_return = output['annualized_return'].iloc[0, :].as_matrix()
```

```
        risks, returns, weights = [], [], []
        for level_return in np.linspace(min(expected_return),max(expected
_return), n_samples):
            P = 2 * covariance_matrix
            q = matrix(np.zeros(n_asset))

            if allow_short:
                G = matrix(0., (n_asset, n_asset))
            else:
                G = matrix(np.diag(-1 * np.ones(n_asset)))

            h = matrix(0., (n_asset, 1))
            A = matrix(np.row_stack((np.ones(n_asset), expected_return)))
            b = matrix([1.0, level_return])
            solvers.options['show_progress'] = False
            sol = solvers.qp(P, q, G, h, A, b)
            risks.append(np.sqrt(sol['primal objective']))
            returns.append(level_return)
            weights.append(dict(zip(assets, list(sol['x'].T))))

        output = {"returns": returns,
                  "risks": risks,
                  "weights": weights}
        output = pd.DataFrame(output)
        return output

    def draw_efficient_frontier(effcient_frontier_output):
        """
        绘出 Efficient Frontier

        Args:
            effcient_frontier_output: Efficient Frontier 的计算结果，即
get_efficient_frontier 的输出
        """

        import seaborn
        from matplotlib import pyplot as plt

        fig = plt.figure(figsize=(7, 4))
        ax = fig.add_subplot(111)
        ax.plot(effcient_frontier_output['risks'],
```

```
effcient_frontier_output['returns'])
        ax.set_title('Efficient Frontier', fontsize=14)
        ax.set_xlabel('Standard Deviation', fontsize=12)
        ax.set_ylabel('Expected Return', fontsize=12)
        ax.tick_params(labelsize=12)
        plt.show()

    def get_minimum_variance_portfolio(return_table,allow_short=False,show_
details=True):
        """
        计算最小方差组合

        Args:
            return_table (DataFrame): 收益率矩阵，列为资产，值为按日期升序排列的收益率
            allow_short (bool): 是否允许卖空
            show_details (bool): 是否显示细节

        Returns:
            dict: 最小方差组合的权重信息，键为资产名，值为权重
        """

        import numpy as np
        from cvxopt import matrix, solvers

        assets = return_table.columns
        n_asset = len(assets)
        if n_asset < 2:
            weights = np.array([1.])
            weights_dict = {assets[0]: 1.}
        else:
            output = describe(return_table, is_print=False)
            covariance_matrix = matrix(output['covariance_matrix'].as_matrix())
            expected_return = output['annualized_return'].iloc[0, :].as_matrix()

            P = 2 * covariance_matrix
            q = matrix(np.zeros(n_asset))

            if allow_short:
                G = matrix(0., (n_asset, n_asset))
            else:
                G = matrix(np.diag(-1 * np.ones(n_asset)))
```

```
            h = matrix(0., (n_asset, 1))
            A = matrix(np.ones(n_asset)).T
            b = matrix([1.0])
            solvers.options['show_progress'] = False
            sol = solvers.qp(P, q, G, h, A, b)
            weights = np.array(sol['x'].T)[0]
            weights_dict = dict(zip(assets, weights))

        r = np.dot(weights, output['annualized_return'].iloc[0, :].as_matrix())
        v = np.sqrt(np.dot(np.dot(weights, covariance_matrix), weights.T))

        if show_details:
            print """
    Minimum Variance Portfolio:
        Short Allowed: {}
        Portfolio Return: {}
        Portfolio Volatility: {}
        Portfolio Weights: {}
    """.format(allow_short, r, v, "\n\t{}".format("\n\t".join("{}: {:.1%}".
format(k, v) for k, v in weights_dict.items()))).strip()

        return weights_dict

    def get_maximum_utility_portfolio(return_table, risk_aversion=3.,
allow_short=False, show_details=True):
        """
        计算最大效用组合，目标函数为：期望年化收益率 - 风险厌恶系数 × 期望年化方差

        Args:
            return_table (DataFrame)：收益率矩阵，列为资产，值为按日期升序排列的收益率
            risk_aversion (float)：风险厌恶系数，越大表示对风险越厌恶，默认为 3.0
            allow_short (bool)：是否允许卖空
            show_details (bool)：是否显示细节

        Returns:
            dict：最小方差组合的权重信息，键为资产名，值为权重
        """

        import numpy as np
        from cvxopt import matrix, solvers
```

```
        assets = return_table.columns
        n_asset = len(assets)
        if n_asset < 2:
            weights = np.array([1.])
            weights_dict = {assets[0]: 1.}
        else:
            output = describe(return_table, is_print=False)
            covariance_matrix = matrix(output['covariance_matrix'].as_matrix())
            expected_return = output['annualized_return'].iloc[0, :].as_matrix()

            if abs(risk_aversion) < 0.01:
                max_ret = max(expected_return)
                weights = np.array([1. if expected_return[i] == max_ret else 0.
for i in range(n_asset)])
                weights_dict = {asset: weights[i] for i, asset in enumerate
(assets)}
            else:
                P = risk_aversion * covariance_matrix
                q = matrix(-expected_return.T)

                if allow_short:
                    G = matrix(0., (n_asset, n_asset))
                else:
                    G = matrix(np.diag(-1 * np.ones(n_asset)))

                h = matrix(0., (n_asset, 1))
                A = matrix(np.ones(n_asset)).T
                b = matrix([1.0])
                solvers.options['show_progress'] = False
                sol = solvers.qp(P, q, G, h, A, b)
                weights = np.array(sol['x'].T)[0]
                weights_dict = dict(zip(assets, weights))

        r = np.dot(weights, output['annualized_return'].iloc[0, :].as_matrix())
        v = np.sqrt(np.dot(np.dot(weights, covariance_matrix), weights.T))

        if show_details:
            print """
    Maximum Utility Portfolio:
        Risk Aversion: {}
```

```
        Short Allowed: {}
        Portfolio Return: {}
        Portfolio Volatility: {}
        Portfolio Weights: {}
    """.format(risk_aversion, allow_short, r, v, "\n\t{}".format("\n\t".
join("{}: {:.1%}".format(k, v) for k, v in weights_dict.items()))).strip()

        return weights_dict

    def get_maximum_sharpe_portfolio(return_table, riskfree_rate=0.,
allow_short=False, show_details=True):
        """
        计算最大效用组合，目标函数为：（期望年化收益率 - 无风险收益率）/ 期望年化方差

        Args:
            return_table (DataFrame): 收益率矩阵，列为资产，值为按日期升序排列的收益率
            riskfree_rate (float): 无风险收益率
            allow_short (bool): 是否允许卖空
            show_details (bool): 是否显示细节

        Returns:
            dict: 最小方差组合的权重信息，键为资产名，值为权重
        """

        import numpy as np
        from cvxopt import matrix, solvers

        assets = return_table.columns
        n_asset = len(assets)
        if n_asset < 2:
            output = describe(return_table, is_print=False)
            r = output['annualized_return'].iat[0, 0]
            v = output['annualized_volatility'].iat[0, 0]
            weights_dict = {assets[0]: 1.}
        else:
            efs = get_efficient_frontier(return_table, allow_short=allow_short,
n_samples=100)
            i_star = max(range(100), key=lambda x: (efs.at[x, "returns"] -
riskfree_rate) / efs.at[x, "risks"])
            r = efs.at[i_star, "returns"]
            v = efs.at[i_star, "risks"]
```

```
            weights_dict = efs.at[i_star, "weights"]

        s = (r - riskfree_rate) / v

        if show_details:
            print """
    Maximum Sharpe Portfolio:
        Riskfree Rate: {}
        Short Allowed: {}
        Portfolio Return: {}
        Portfolio Volatility: {}
        Portfolio Sharpe: {}
        Portfolio Weights: {}
    """.format(riskfree_rate, allow_short, r, v, s, "\n\t{}".format("\n\t".
join("{}: {:.1%}".format(k, v) for k, v in weights_dict.items()))).strip()

        return weights_dict
```

如下代码利用上述理论和函数构造了一个简单的均值方差模型示例：

```
    start = '20130101'
    end = '20160630'
    indices = ['000300.ZICN', '000905.ZICN', '399006.ZICN', '000012.ZICN',
'000013.ZICN']
    df = DataAPI.MktIdxdGet(indexID=indices, beginDate=start, endDate=end,
field="tradeDate,indexID,closeIndex")
    df = df.pivot(index="tradeDate", columns="indexID", values="closeIndex")
    for index in indices:
        df[index] = df[index] / df[index].shift() - 1.
    return_table = df.dropna()

    describe(return_table, is_print=True)
```

运行结果如下：

```
    covariance_matrix:
    indexID   000012.ZICN  000013.ZICN  000300.ZICN  000905.ZICN  399006.ZICN
    indexID
    000012.ZICN    0.000057    0.000009    0.000047    0.000089    0.000034
    000013.ZICN    0.000009    0.000031    0.000108    0.000173    0.000179
    000300.ZICN    0.000047    0.000108    0.080919    0.078297    0.071111
    000905.ZICN    0.000089    0.000173    0.078297    0.106164    0.110480
    399006.ZICN    0.000034    0.000179    0.071111    0.110480    0.148057
```

```
    annualized_return:
    indexID  000012.ZICN  000013.ZICN  000300.ZICN  000905.ZICN  399006.ZICN
    0          0.045391     0.075276     0.068568     0.206761     0.408574

    annualized_volatility:
    indexID  000012.ZICN  000013.ZICN  000300.ZICN  000905.ZICN  399006.ZICN
    0          0.007538     0.005588     0.284462     0.325829     0.384782

    coefficient_matrix:
    indexID    000012.ZICN  000013.ZICN  000300.ZICN  000905.ZICN  399006.ZICN
    indexID
    000012.ZICN     1.000000     0.222367     0.021862     0.036328     0.011754
    000013.ZICN     0.222367     1.000000     0.067919     0.095261     0.083447
    000300.ZICN     0.021862     0.067919     1.000000     0.844759     0.649678
    000905.ZICN     0.036328     0.095261     0.844759     1.000000     0.881214
    399006.ZICN     0.011754     0.083447     0.649678     0.881214     1.000000

    {'annualized_return': indexID  000012.ZICN  000013.ZICN  000300.ZICN
000905.ZICN  399006.ZICN
     0          0.045391     0.075276     0.068568     0.206761     0.408574,
     'annualized_volatility': indexID  000012.ZICN  000013.ZICN  000300.ZICN
000905.ZICN  399006.ZICN
     0          0.007538     0.005588     0.284462     0.325829     0.384782,
     'coefficient_matrix':
     indexID    000012.ZICN  000013.ZICN  000300.ZICN  000905.ZICN  399006.ZICN
     indexID
     000012.ZICN    1.000000     0.222367     0.021862     0.036328     0.011754
     000013.ZICN    0.222367     1.000000     0.067919     0.095261     0.083447
     000300.ZICN    0.021862     0.067919     1.000000     0.844759     0.649678
     000905.ZICN    0.036328     0.095261     0.844759     1.000000     0.881214
     399006.ZICN    0.011754     0.083447     0.649678     0.881214     1.000000,
     'covariance_matrix':
     indexID    000012.ZICN  000013.ZICN  000300.ZICN  000905.ZICN  399006.ZICN
     indexID
     000012.ZICN    0.000057     0.000009     0.000047     0.000089     0.000034
     000013.ZICN    0.000009     0.000031     0.000108     0.000173     0.000179
     000300.ZICN    0.000047     0.000108     0.080919     0.078297     0.071111
     000905.ZICN    0.000089     0.000173     0.078297     0.106164     0.110480
     399006.ZICN    0.000034     0.000179     0.071111     0.110480     0.148057}
```

对 efficient_frontier 的计算如下：

```
    efficient_frontier = get_efficient_frontier(return_table, allow_short=
False, n_samples=50)
    efficient_frontier
```

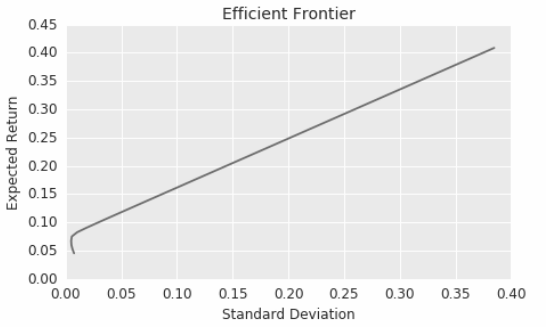
计算结果如下：

	returns	risks	weights
0	0.045391	0.007538	{u'000300.ZICN': 1.15393430344e-08, u'000905.Z...
1	0.052803	0.006133	{u'000300.ZICN': 0.000496525017029, u'000905.Z...
2	0.060215	0.005181	{u'000300.ZICN': 0.000251697279178, u'000905.Z...
3	0.067627	0.004964	{u'000300.ZICN': 0.000223375191279, u'000905.Z...
4	0.075038	0.005558	{u'000300.ZICN': 0.000130923763251, u'000905.Z...
5	0.08245	0.010298	{u'000300.ZICN': 8.53789819096e-07, u'000905.Z...
6	0.089862	0.018088	{u'000300.ZICN': 5.32468255522e-06, u'000905.Z...
7	0.097274	0.02635	{u'000300.ZICN': 9.02636326881e-07, u'000905.Z...
8	0.104686	0.034751	{u'000300.ZICN': 2.02472361778e-07, u'000905.Z...
9	0.112098	0.043209	{u'000300.ZICN': 9.28878058971e-08, u'000905.Z...
10	0.11951	0.051697	{u'000300.ZICN': 4.57879968585e-08, u'000905.Z...
11	0.126922	0.060201	{u'000300.ZICN': 2.40064060703e-08, u'000905.Z...
12	0.134334	0.068717	{u'000300.ZICN': 1.37843225515e-06, u'000905.Z...
13	0.141746	0.077239	{u'000300.ZICN': 8.4285683672e-07, u'000905.ZI...
14	0.149157	0.085766	{u'000300.ZICN': 5.49491378923e-07, u'000905.Z...
15	0.156569	0.094297	{u'000300.ZICN': 3.77948806475e-07, u'000905.Z...
16	0.163981	0.102831	{u'000300.ZICN': 2.70306319752e-07, u'000905.Z...
17	0.171393	0.111366	{u'000300.ZICN': 2.00553907455e-07, u'000905.Z...
18	0.178805	0.119904	{u'000300.ZICN': 1.52644281534e-07, u'000905.Z...
19	0.186217	0.128443	{u'000300.ZICN': 1.17142800495e-07, u'000905.Z...
20	0.193629	0.136982	{u'000300.ZICN': 9.02783351161e-08, u'000905.Z...
21	0.201041	0.145523	{u'000300.ZICN': 6.97438031359e-08, u'000905.Z...
22	0.208453	0.154064	{u'000300.ZICN': 5.9059023565e-08, u'000905.ZI...
23	0.215865	0.162606	{u'000300.ZICN': 5.19484511925e-08, u'000905.Z...
24	0.223276	0.171149	{u'000300.ZICN': 4.55263038143e-08, u'000905.Z...

25	0.230688	0.179692	{u'000300.ZICN': 3.98048975884e-08, u'000905.Z...
26	0.2381	0.188235	{u'000300.ZICN': 3.47759997238e-08, u'000905.Z...
27	0.245512	0.196779	{u'000300.ZICN': 3.04007403968e-08, u'000905.Z...
28	0.252924	0.205323	{u'000300.ZICN': 2.66383477155e-08, u'000905.Z...
29	0.260336	0.213868	{u'000300.ZICN': 2.34494796732e-08, u'000905.Z...
30	0.267748	0.222412	{u'000300.ZICN': 2.07935283905e-08, u'000905.Z...
31	0.27516	0.230957	{u'000300.ZICN': 1.86292415345e-08, u'000905.Z...
32	0.282572	0.239502	{u'000300.ZICN': 1.69144006915e-08, u'000905.Z...
33	0.289984	0.248047	{u'000300.ZICN': 1.56046914802e-08, u'000905.Z...
34	0.297395	0.256592	{u'000300.ZICN': 2.78591582895e-08, u'000905.Z...
35	0.304807	0.265138	{u'000300.ZICN': 2.40132032721e-08, u'000905.Z...
36	0.312219	0.273683	{u'000300.ZICN': 2.06266291619e-08, u'000905.Z...
37	0.319631	0.282229	{u'000300.ZICN': 1.77207719046e-08, u'000905.Z...
38	0.327043	0.290774	{u'000300.ZICN': 2.38027675854e-08, u'000905.Z...
39	0.334455	0.29932	{u'000300.ZICN': 3.21755527866e-08, u'000905.Z...
40	0.341867	0.307866	{u'000300.ZICN': 4.45262652532e-08, u'000905.Z...
41	0.349279	0.316412	{u'000300.ZICN': 6.45951915931e-08, u'000905.Z...
42	0.356691	0.324958	{u'000300.ZICN': 1.02657055122e-07, u'000905.Z...
43	0.364103	0.333504	{u'000300.ZICN': 1.87751964819e-09, u'000905.Z...
44	0.371514	0.342051	{u'000300.ZICN': 1.0304584484e-07, u'000905.ZI...
45	0.378926	0.350597	{u'000300.ZICN': 3.08767068474e-09, u'000905.Z...
46	0.386338	0.359143	{u'000300.ZICN': 1.40775462715e-08, u'000905.Z...
47	0.39375	0.367689	{u'000300.ZICN': 1.31902224778e-08, u'000905.Z...
48	0.401162	0.376236	{u'000300.ZICN': 7.95099027876e-09, u'000905.Z...
49	0.408574	0.384782	{u'000300.ZICN': -1.89291467521e-10, u'000905....

画出有效前沿图像：

```
draw_efficient_frontier(efficient_frontier)
```



计算最小方差组合：

```
    get_minimum_variance_portfolio(return_table, allow_short=False, show_
details=True)
```

结果如下：

```
    Minimum Variance Portfolio:
       Short Allowed: False
       Portfolio Return: 0.0658694175084
       Portfolio Volatility: 0.00493711522906
       Portfolio Weights:
       000300.ZICN: 0.0%
       000905.ZICN: 0.0%
       399006.ZICN: 0.0%
       000013.ZICN: 68.4%
       000012.ZICN: 31.6%

    {'000012.ZICN': 0.31561408123537693,
     '000013.ZICN': 0.68417715824887437,
     '000300.ZICN': 0.00010187640216775817,
     '000905.ZICN': 4.762500377417469e-05,
     '399006.ZICN': 5.9259109806737253e-05}
```

计算最大效用组合，目标函数为期望年化收益率–风险厌恶系数×期望年化方差：

```
    get_maximum_utility_portfolio(return_table, risk_aversion=3, allow_short=
False, show_details=True)
```

结果如下：

```
    Maximum Utility Portfolio:
```

```
        Risk Aversion: 3
        Short Allowed: False
        Portfolio Return: 0.325596301804
        Portfolio Volatility: 0.289106409166
        Portfolio Weights:
        000300.ZICN: 0.0%
        000905.ZICN: 0.0%
        399006.ZICN: 75.1%
        000013.ZICN: 24.9%
        000012.ZICN: 0.0%

    {'000012.ZICN': 2.2333561647811078e-08,
     '000013.ZICN': 0.24895952361049398,
     '000300.ZICN': 1.6722006123307858e-09,
     '000905.ZICN': 2.2668476811452807e-09,
     '399006.ZICN': 0.75104045011689602}
```

计算最大效用组合，目标函数为：（期望年化收益率–无风险收益率）/ 期望年化方差：

```
    get_maximum_sharpe_portfolio(return_table, riskfree_rate=0.03, allow_
short=True, show_details=True)
```

结果如下：

```
    Maximum Sharpe Portfolio:
        Riskfree Rate: 0.03
        Short Allowed: True
        Portfolio Return: 0.074739017609
        Portfolio Volatility: 0.00546891894139
        Portfolio Sharpe: 8.18059621811
        Portfolio Weights:
        000300.ZICN: 0.0%
        000905.ZICN: -0.5%
        399006.ZICN: 0.4%
        000013.ZICN: 96.0%
        000012.ZICN: 4.1%

    {'000012.ZICN': 0.04111092040174396,
     '000013.ZICN': 0.959549446701585,
     '000300.ZICN': 7.529386541597854e-05,
     '000905.ZICN': -0.0046427924089252285,
     '399006.ZICN': 0.003907131440180278}
```

6.8.2　Black-Litterman 模型

1. Black-Litterman 模型概述

基于马科维茨均值-方差模型的资产组合分析需要获取各类资产的预期收益和方差。通常可以通过情景分析法和历史数据法估算预期收益和收益率方差。情景分析法主要根据当前行情和宏观经济环境等因素形成主观的预期，主观性和随意性显然过强；历史数据法则完全根据过去的历史收益率计算收益均值和方差，用来代替对未来的预期，这也是目前的主流做法。这种做法存在几个问题，一是历史数据往往由于历史的一些宏观环境或随机因素而存在比较大的波动性，这也意味着历史的收益率和方差在未来有可能由于宏观环境和随机因素的改变而不会保持一致；二是由于采取的历史数据的时间段不同，估算出的预期收益率和方差也会有较大的差别，导致得出的最优资产配置比例也会有较大的差别。高盛的 Black F.和 Litterman R 在 1991 年的一篇论文中提到：他们在对全球债券投资组合的研究中发现，在对德国债券预期报酬率做 0.1%的小幅修正后，该类资产的投资比例竟由原来的 10.0%提高至 55.0%，这也意味着马科维茨的均值方差模型得到的投资组合对于输入的参数过于敏感。

Black 和 Litteraman 在前述均值方差模型的基础上，通过历史数据估计基准预期和方差，导入投资者的主观预期，把历史数据法和情景分析法结合起来，形成新的市场收益预期，从而解决了在前述模型中预期收益和方差估计存在的问题。

2. Black-Litterman 模型（简称 B-L 模型）简介

B-L 模型在均衡收益的基础上通过投资者的观点修正了期望收益，使得均值方差组合优化中的期望收益更为合理，而且将投资者的观点融入到模型中，在一定程度上是对马科维茨均值方差组合理论的改进。

B-L 模型假设各资产收益率 R 服从联合正态分布，$R \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$，其中$\boldsymbol{\mu}$和$\boldsymbol{\Sigma}$是各资产预期收益率和协方差的估计值。现在假设估计向量$\boldsymbol{\mu}$本身也是随机的，且服从正态分布：$\boldsymbol{\mu} \sim N(\Pi, \tau\boldsymbol{\Sigma})$，其中$\Pi$为先验期望收益率的期望值，通常由历史平均收益率表示。模型引入投资者个人观点的方式是用线性方程组表示，每个方程表示一个观点。例如，如果投资者认为在未来第 3 种资产会比第 1 种资产的收益率高 2%，就可以表示为

$$-1\times\boldsymbol{\mu}_1 + 0\times\boldsymbol{\mu}_2 + 1\times\boldsymbol{\mu}_3 + \cdots + 0\times\boldsymbol{\mu}_N = 2\%$$

如果投资者认为在未来第 2 种资产的收益率应该为 5%，就可以表示为

$$0\times\boldsymbol{\mu}_1+1\times\boldsymbol{\mu}_2+0\times\boldsymbol{\mu}_3+\cdots+0\times\boldsymbol{\mu}_N=5\%$$

用 $\boldsymbol{P}$ 来表示该观点线性方程组的系数矩阵时，观点方程组可表示为：$\boldsymbol{P}_{\boldsymbol{\mu}}=\boldsymbol{q}$。由于投资人的观点也存在不确定性，因此在 $\boldsymbol{q}$ 的基础上还可以加上一个随机误差项：$\boldsymbol{P}_{\boldsymbol{\mu}}=\boldsymbol{q}+\epsilon$，其中 $\epsilon\sim N(0,\boldsymbol{\Omega})$，因而 $\boldsymbol{P}_{\boldsymbol{\mu}}\sim N(\boldsymbol{q},\boldsymbol{\Omega})$。这里，$\boldsymbol{P}$ 被称为 Pick Matrix，为 $K\times N$ 矩阵，表示对于 N 种资产的 K 个观点；$\boldsymbol{q}$ 为 $K\times 1$ 看法向量；$\boldsymbol{\Omega}$为看法向量误差项的 $K\times K$ 协方差矩阵，表示投资者的观点的不确定程度。我们通常对于$\boldsymbol{\Omega}$采用$\boldsymbol{\Omega}$=diag($\tau\boldsymbol{P}\boldsymbol{\Sigma}\boldsymbol{P}^{\boldsymbol{T}}$)的方式构造。

前面我们讨论过市场的看法：$\boldsymbol{\mu}\sim N(\Pi,\tau\boldsymbol{\Sigma})$，用 $\boldsymbol{P}$ 调整后的市场看法可表示为：$P_{\boldsymbol{\mu}}\sim N(P\Pi,\tau\boldsymbol{P}\Sigma\boldsymbol{P}^{\boldsymbol{T}})$。

投资人观点和市场看法的差距服从分布：$N(\boldsymbol{q}-\boldsymbol{P}\Pi,\boldsymbol{\Omega}+\tau\boldsymbol{P}\Sigma\boldsymbol{P}^{\boldsymbol{T}})$

然后根据贝叶斯法则，结合先验信息和投资者观点，可以计算调整后的预期收益率和收益率方差分别为：

$$\tilde{\Pi}=\Pi+\tau\boldsymbol{\Sigma}\boldsymbol{P}^{T}(\boldsymbol{\Omega}+\tau\boldsymbol{P}\boldsymbol{\Sigma}\boldsymbol{P}^{T})^{-1}(\boldsymbol{q}-\boldsymbol{P}\Pi)$$

$$\mathbf{M}=\tau\boldsymbol{\Sigma}-\tau\boldsymbol{\Sigma}\boldsymbol{P}^{T}(\boldsymbol{\Omega}+\tau\boldsymbol{P}\boldsymbol{\Sigma}\boldsymbol{P}^{T})^{-1}\boldsymbol{P}\tau\boldsymbol{\Sigma}$$

$$\Sigma_{\boldsymbol{P}}=\boldsymbol{\Sigma}+M$$

其中，Π 为先验的期望收益，通过历史平均年化收益得到；$\tilde{\Pi}$ 为个人观点调整后的期望收益；$\boldsymbol{\Sigma}$为资产收益率之间的协方差矩阵；M 为后验分布预期收益率的方差；$\boldsymbol{\Sigma}_{\boldsymbol{P}}$ 为调整后预期收益率方差；τ 为均衡收益方差的刻度值，体现了对个人观点在总体估计中的权重，通常取值为 0.025～0.05；$\boldsymbol{P}$、$\boldsymbol{q}$、$\boldsymbol{\Omega}$为观点矩阵。

在得到调整后的期望收益率和方差后，就可以根据马科维茨均值方差模型计算最优权重。

$$\hat{\omega}=(\delta\boldsymbol{\Sigma}_{\boldsymbol{P}})^{-1}\tilde{\Pi}$$ （无卖空约束情况，其中δ为风险厌恶系数）

3. Black-Litterman 模型应用

在实践中应用该模型，简要来说主要有如下步骤。

（1）计算得到先验的期望收益。

（2）个人观点模型化，观点可以涉及单个资产，也可以有多个资产，最后按照一定的规则将所有观点构建成矩阵 $\boldsymbol{P}$、$\boldsymbol{Q}$ 和 $\boldsymbol{\Omega}$。

（3）计算调整后的预期收益率、调整后的收益率方差。

（4）根据调整后的期望收益，利用均值方差模型计算最优权重。

下面的代码提供了若干应用 B-L 模型进行优化的函数，其中 get_BL_efficient_frontier()用于获取根据调整后的收益率期望和方差得到的有效边界；draw_efficient_frontier()用于绘制有效边界图形；get_BL_minimum_variance_portfolio()用于获取最小方差投资组合；get_BL_maximum_utility_portfolio()用于获取基于给定效用函数的最大化效用投资组合；get_maximum_sharpe_portfolio()用于获取最大夏普率投资组合。

```
    def describe(return_table, is_print=True):
        """
        输出收益率矩阵的描述性统计量，包括：
            年化收益率
            年化标准差
            相关系数矩阵

        Args:
            return_table (DataFrame)：收益率矩阵，列为资产，值为按日期升序排列的收益率
            is_print (bool)：是否直接输出

        Returns:
            dict：描述性统计量字典，键为"annualized_return","annualized_
volatility","covariance_matrix"和"coefficient_matrix"

        Examples:
            >> describe(return_table)
            >> describe(return_table, is_print=True)
        """
        import numpy as np
        import pandas as pd
        from scipy.stats.mstats import gmean

        output = {}
        output['annualized_return'] = pd.DataFrame(dict(zip(return_table.
columns, gmean(return_table+1.)**252 - 1.)), index=[0], columns=return_
table.columns)
        output['annualized_volatility'] = pd.DataFrame(return_table.std() *
np.sqrt(250)).T
        output['covariance_matrix'] = return_table.cov() * 250.
        output['coefficient_matrix'] = return_table.corr()
```

```
        if is_print:
            for key, val in output.iteritems():
                print "{}:\n{}\n".format(key, val)

    return output


    def get_BL_efficient_frontier(return_table,tau=0.05,P=None,Q=None,Omega=
None,allow_short=False, n_samples=25):
        """
        计算 Efficient Frontier

        Args:
            return_table (DataFrame): 收益率矩阵，列为资产，值为按日期升序排列的收益率
            n_samples (int): 用于计算 Efficient Frontier 的采样点数量
            P(np.array): 观点矩阵
            Q(np.array): 观点收益矩阵
            Omega(np.array): 观点置信度矩阵
            tau(float): 为均衡收益方差的刻度值，体现了对个人观点在总体估计中的权重

        Returns:
            DataFrame: Efficient Frontier 的结果，列为"returns", "risks", "weights"
        """

        import numpy as np
        import pandas as pd
        from cvxopt import matrix, solvers

        assets = return_table.columns
        n_asset = len(assets)
        if n_asset < 2:
            raise ValueError("There must be at least 2 assets to calculate the
efficient frontier!")

        output = describe(return_table, is_print=False)
        covmat =(output['covariance_matrix'])
        expected_return = output['annualized_return'].iloc[0, :]

        # 求解调整后的期望收益、方差
        adjustedReturn = expected_return + tau*covmat.dot(P.transpose()).dot
(np.linalg.inv(Omega+tau*(P.dot(covmat).dot(P.transpose())))).dot(Q -
P.dot(expected_return))
        right = (tau)*covmat.dot(P.transpose()).dot(np.linalg.inv(Omega+
```

```
P.dot(covmat).dot(P.transpose()))).dot(P.dot(tau*covmat))
        right = right.transpose()
        right = right.set_index(expected_return.index)
        M = tau*covmat - right
        Sigma_p = covmat + M
        adjustedReturn = adjustedReturn.as_matrix()
        Sigma_p = matrix(Sigma_p.as_matrix())

        risks, returns, weights = [], [], []
        for level_return in np.linspace(min(adjustedReturn),
max(adjustedReturn), n_samples):
            P = 2 * Sigma_p
            q = matrix(np.zeros(n_asset))

            if allow_short:
                G = matrix(0., (n_asset, n_asset))
            else:
                G = matrix(np.diag(-1 * np.ones(n_asset)))

            h = matrix(0., (n_asset, 1))
            A = matrix(np.row_stack((np.ones(n_asset), adjustedReturn)))
            b = matrix([1.0, level_return])
            solvers.options['show_progress'] = False
            sol = solvers.qp(P, q, G, h, A, b)
            risks.append(np.sqrt(sol['primal objective']))
            returns.append(level_return)
            weights.append(dict(zip(assets, list(sol['x'].T))))

        output = {"returns": returns,
                  "risks": risks,
                  "weights": weights}
        output = pd.DataFrame(output)
        return output

    def draw_efficient_frontier(effcient_frontier_output):
        """
        绘出 Efficient Frontier

        Args:
            effcient_frontier_output: Efficient Frontier 的计算结果，即
get_efficient_frontier 的输出
        """
```

```
        import seaborn
        from matplotlib import pyplot as plt

        fig = plt.figure(figsize=(7, 4))
        ax = fig.add_subplot(111)
        ax.plot(effcient_frontier_output['risks'],
effcient_frontier_output['returns'])
        ax.set_title('Efficient Frontier', fontsize=14)
        ax.set_xlabel('Standard Deviation', fontsize=12)
        ax.set_ylabel('Expected Return', fontsize=12)
        ax.tick_params(labelsize=12)
        plt.show()

    def get_BL_minimum_variance_portfolio(return_table,tau=0.05,P=None,
Q=None,Omega=None, allow_short=False, show_details=True):
        """
        计算最小方差组合

        Args:
            return_table (DataFrame): 收益率矩阵，列为资产，值为按日期升序排列的收益率
            allow_short (bool): 是否允许卖空
            show_details (bool): 是否显示细节
            P(np.array): 观点矩阵
            Q(np.array): 观点收益矩阵
            Omega(np.array): 观点置信度矩阵
            tau(float): 为均衡收益方差的刻度值，体现了对个人观点在总体估计中的权重

        Returns:
            dict: 最小方差组合的权重信息，键为资产名，值为权重
        """

        import numpy as np
        from cvxopt import matrix, solvers

        assets = return_table.columns
        n_asset = len(assets)
        if n_asset < 2:
            weights = np.array([1.])
            weights_dict = {assets[0]: 1.}
        else:
            output = describe(return_table, is_print=False)
            covmat =(output['covariance_matrix'])
            expected_return = output['annualized_return'].iloc[0, :]
```

```
            # 求解调整后的期望收益、方差
            adjustedReturn=expected_return+tau*covmat.dot(P.transpose()).dot
(np.linalg.inv(Omega+tau*(P.dot(covmat).dot(P.transpose())))).dot(Q -
P.dot(expected_return))
            right = (tau)*covmat.dot(P.transpose()).dot(np.linalg.inv (Omega+P.
dot(covmat).dot(P.transpose()))).dot(P.dot(tau*covmat))
            right = right.transpose()
            right = right.set_index(expected_return.index)
            M = tau*covmat - right
            Sigma_p = covmat + M
            adjustedReturn = adjustedReturn.as_matrix()
            Sigma_p = matrix(Sigma_p.as_matrix())

            P = 2 * Sigma_p
            q = matrix(np.zeros(n_asset))

            if allow_short:
                G = matrix(0., (n_asset, n_asset))
            else:
                G = matrix(np.diag(-1 * np.ones(n_asset)))

            h = matrix(0., (n_asset, 1))
            A = matrix(np.ones(n_asset)).T
            b = matrix([1.0])
            solvers.options['show_progress'] = False
            sol = solvers.qp(P, q, G, h, A, b)
            weights = np.array(sol['x'].T)[0]
            weights_dict = dict(zip(assets, weights))

        r = np.dot(weights, output['annualized_return'].iloc[0, :].as_matrix())
        v = np.sqrt(np.dot(np.dot(weights, Sigma_p), weights.T))

        if show_details:
            print """
    Minimum Variance Portfolio:
        Short Allowed: {}
        Portfolio Return: {}
        Portfolio Volatility: {}
        Portfolio Weights: {}
    """.format(allow_short,r,v,"\n\t{}".format("\n\t".join("{}:{:.1%}".forma
t(k, v) for k, v in weights_dict.items()))).strip()
```

```
        return weights_dict

    def get_BL_maximum_utility_portfolio(return_table,tau=0.05,P=None,Q=None,
Omega=None, risk_aversion=3., allow_short=False, show_details=True):
        """
        计算最大效用组合，目标函数为：期望年化收益率 - 风险厌恶系数 × 期望年化方差

        Args:
            return_table (DataFrame): 收益率矩阵，列为资产，值为按日期升序排列的收益率
            risk_aversion (float): 风险厌恶系数，越大表示对风险越厌恶，默认为 3.0
            allow_short (bool): 是否允许卖空
            show_details (bool): 是否显示细节
            P(np.array): 观点矩阵
            Q(np.array): 观点收益矩阵
            Omega(np.array): 观点置信度矩阵
            tau(float): 为均衡收益方差的刻度值，体现了对个人观点在总体估计中的权重

        Returns:
            dict: 最小方差组合的权重信息，键为资产名，值为权重
        """

        import numpy as np
        from cvxopt import matrix, solvers

        assets = return_table.columns
        n_asset = len(assets)
        if n_asset < 2:
            weights = np.array([1.])
            weights_dict = {assets[0]: 1.}
        else:
            output = describe(return_table, is_print=False)
            covmat =(output['covariance_matrix'])
            expected_return = output['annualized_return'].iloc[0, :]

            # 求解调整后的期望收益、方差
            adjustedReturn = expected_return + tau*covmat.dot(P.transpose()).
dot(np.linalg.inv(Omega+tau*(P.dot(covmat).dot(P.transpose())))).dot(Q -
P.dot(expected_return))
            right = (tau)*covmat.dot(P.transpose()).dot(np.linalg.inv(Omega+
P.dot(covmat).dot(P.transpose()))).dot(P.dot(tau*covmat))
            right = right.transpose()
            right = right.set_index(expected_return.index)
            M = tau*covmat - right
```

```
            Sigma_p = covmat + M
            adjustedReturn = adjustedReturn.as_matrix()
            Sigma_p = matrix(Sigma_p.as_matrix())

            if abs(risk_aversion) < 0.01:
                max_ret = max(adjustedReturn)
                weights = np.array([1. if adjustedReturn[i] == max_ret else 0. for
i in range(n_asset)])
                weights_dict = {asset: weights[i] for i, asset in enumerate
(assets)}
            else:
                P = risk_aversion * Sigma_p
                q = matrix(-adjustedReturn.T)

                if allow_short:
                    G = matrix(0., (n_asset, n_asset))
                else:
                    G = matrix(np.diag(-1 * np.ones(n_asset)))

                h = matrix(0., (n_asset, 1))
                A = matrix(np.ones(n_asset)).T
                b = matrix([1.0])
                solvers.options['show_progress'] = False
                sol = solvers.qp(P, q, G, h, A, b)
                weights = np.array(sol['x'].T)[0]
                weights_dict = dict(zip(assets, weights))

        r = np.dot(weights, output['annualized_return'].iloc[0, :].as_matrix())
        v = np.sqrt(np.dot(np.dot(weights, Sigma_p), weights.T))

        if show_details:
            print """
    Maximum Utility Portfolio:
        Risk Aversion: {}
        Short Allowed: {}
        Portfolio Return: {}
        Portfolio Volatility: {}
        Portfolio Weights: {}
    """.format(risk_aversion, allow_short, r, v, "\n\t{}".format("\n\t".
join("{}: {:.1%}".format(k, v) for k, v in weights_dict.items()))).strip()

        return weights_dict
```

```
    def get_maximum_sharpe_portfolio(return_table, riskfree_rate=0.,tau=0.05,
P=None,Q=None,Omega=None,allow_short=False, show_details=True):
        """
        计算最大效用组合，目标函数为：（期望年化收益率 - 无风险收益率）/ 期望年化方差

        Args:
            return_table (DataFrame): 收益率矩阵，列为资产，值为按日期升序排列的收益率
            riskfree_rate (float): 无风险收益率
            allow_short (bool): 是否允许卖空
            show_details (bool): 是否显示细节
            P(np.array): 观点矩阵
            Q(np.array): 观点收益矩阵
            Omega(np.array): 观点置信度矩阵
            tau(float): 为均衡收益方差的刻度值，体现了对个人观点在总体估计中的权重

        Returns:
            dict: 最小方差组合的权重信息，键为资产名，值为权重
        """

        import numpy as np
        from cvxopt import matrix, solvers

        assets = return_table.columns
        n_asset = len(assets)
        if n_asset < 2:
            output = describe(return_table, is_print=False)
            r = output['annualized_return'].iat[0, 0]
            v = output['annualized_volatility'].iat[0, 0]
            weights_dict = {assets[0]: 1.}
        else:
            efs = get_BL_efficient_frontier(return_table,tau,P=P,Q=Q,Omega=
Omega,allow_short=allow_short, n_samples=100)
            i_star = max(range(100), key=lambda x: (efs.at[x, "returns"] -
riskfree_rate) / efs.at[x, "risks"])
            r = efs.at[i_star, "returns"]
            v = efs.at[i_star, "risks"]
            weights_dict = efs.at[i_star, "weights"]

        s = (r - riskfree_rate) / v

        if show_details:
            print """
    Maximum Sharpe Portfolio:
```

```
        Riskfree Rate: {}
        Short Allowed: {}
        Portfolio Return: {}
        Portfolio Volatility: {}
        Portfolio Sharpe: {}
        Portfolio Weights: {}
    """.format(riskfree_rate, allow_short, r, v, s, "\n\t{}".format("\n\t".
join("{}: {:.1%}".format(k, v) for k, v in weights_dict.items()))).strip()

        return weights_dict
```

下面，我们利用沪深 300 指数、中证 500 指数、创业板指、国债指数、企业债指数构造一个利用 B-L 模型的简单示例（让我们边运行边看结果）：

```
    import numpy as np
    import pandas as pd
    from cvxopt import matrix, solvers
    start = '20130101'
    end = '20160630'
    indices = ['000300.ZICN', '000905.ZICN', '399006.ZICN', '000012.ZICN',
'000013.ZICN']
    df = DataAPI.MktIdxdGet(indexID=indices, beginDate=start, endDate=end,
field="tradeDate,indexID,closeIndex")
    df = df.pivot(index="tradeDate", columns="indexID", values="closeIndex")
    for index in indices:
        df[index] = df[index] / df[index].shift() - 1.
    return_table = df.dropna()
```

输出收益率矩阵的描述性统计量：

```
    describe(return_table, is_print=False)
```

运行结果如下：

```
    {'annualized_return': indexID  000012.ZICN  000013.ZICN  000300.ZICN
000905.ZICN  399006.ZICN
     0         0.045391    0.075276    0.068568    0.206761    0.408574,
     'annualized_volatility': indexID  000012.ZICN  000013.ZICN  000300.ZICN
000905.ZICN  399006.ZICN
     0         0.007538    0.005588    0.284462    0.325829    0.384782,
     'coefficient_matrix': indexID     000012.ZICN  000013.ZICN  000300.ZICN
000905.ZICN  399006.ZICN
     indexID
     000012.ZICN   1.000000    0.222367    0.021862    0.036328    0.011754
     000013.ZICN   0.222367    1.000000    0.067919    0.095261    0.083447
```

```
    000300.ZICN    0.021862    0.067919    1.000000    0.844759    0.649678
    000905.ZICN    0.036328    0.095261    0.844759    1.000000    0.881214
    399006.ZICN   0.011754    0.083447    0.649678    0.881214    1.000000,
    'covariance_matrix': indexID     000012.ZICN  000013.ZICN  000300.ZICN
000905.ZICN  399006.ZICN
    indexID
    000012.ZICN    0.000057    0.000009    0.000047    0.000089    0.000034
    000013.ZICN    0.000009    0.000031    0.000108    0.000173    0.000179
    000300.ZICN    0.000047    0.000108    0.080919    0.078297    0.071111
    000905.ZICN    0.000089    0.000173    0.078297    0.106164    0.110480
    399006.ZICN   0.000034    0.000179    0.071111    0.110480    0.148057}
```

生成观点矩阵 ***P***、***Q***、***Omega***，我们知道历史数据由于包含了 14～15 牛市，所以历史平均收益偏高，尤其是创业板，在刚经历 3 轮股灾的现在，显然要调整股市的期望收益。给出下面 3 个观点。

◎ 中证 500 的收益率超过沪深 300 的收益率的 5%（相对于原来差距的近 20%显著下调）。

◎ HS300 的收益率超过国债的收益率的 1%（相对于原来的差距微调）。

◎ 创业板指收益比企业债指数高 5%（相对于原来差距的近 33%显著下调）。

代码如下：

```
output = describe(return_table, is_print=False)
covariance_matrix = output['covariance_matrix']
expected_return = output['annualized_return'].iloc[0, :]
tau = 0.05

P = np.array([[-1,1,0,0,0],[1,0,0,-1,0],[0,0,1,0,-1]])
print "P:",P
Q = np.array([0.05,0.01,0.05])
print "Q:",Q
Omega = tau*(P.dot(covariance_matrix).dot(P.transpose()))
Omega = np.diag(np.diag(Omega,k=0))
print "Omega:",Omega
```

运行结果如下：

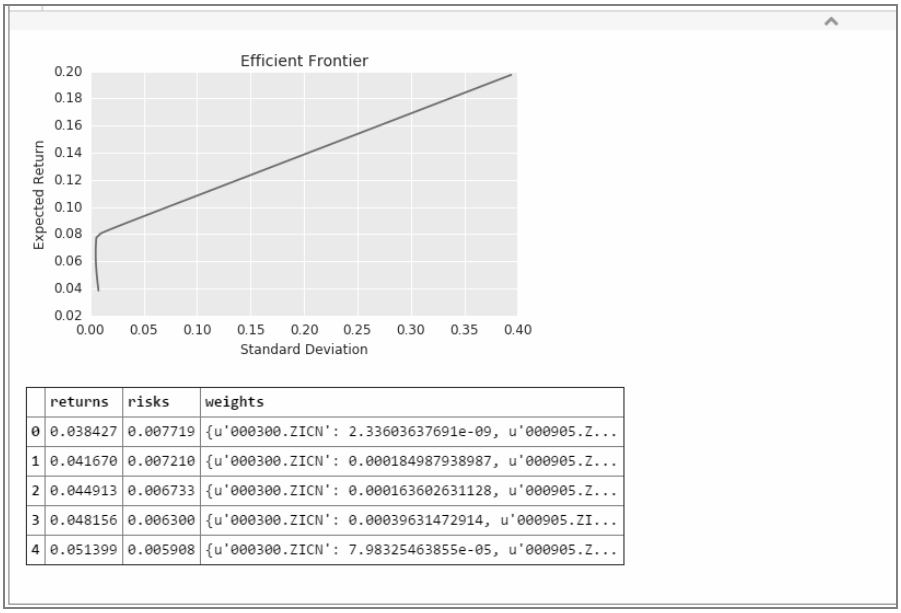
```
P:
 [[-1  1  0  0  0]
 [ 1  0  0 -1  0]
 [ 0  0  1  0 -1]]
```

```
    Q: [ 0.05  0.01  0.05]
    Omega: [[  3.46573857e-06   0.00000000e+00   0.00000000e+00]
     [  0.00000000e+00   5.30213659e-03   0.00000000e+00]
     [  0.00000000e+00   0.00000000e+00   4.33769150e-03]]
```

计算调整期望收益后的 efficient_frontier：

```
    efficient_frontier=get_BL_efficient_frontier(return_table,tau,P=P,Q=Q,Om
ega=Omega,allow_short=False, n_samples=50)
    draw_efficient_frontier(efficient_frontier)
    efficient_frontier.head()
```

运行结果如图 6-27 所示。



	returns	risks	weights
0	0.038427	0.007719	{u'000300.ZICN': 2.33603637691e-09, u'000905.Z...
1	0.041670	0.007210	{u'000300.ZICN': 0.000184987938987, u'000905.Z...
2	0.044913	0.006733	{u'000300.ZICN': 0.000163602631128, u'000905.Z...
3	0.048156	0.006300	{u'000300.ZICN': 0.00039631472914, u'000905.ZI...
4	0.051399	0.005908	{u'000300.ZICN': 7.98325463855e-05, u'000905.Z...

图 6-27

计算调整期望收益后的最小方差组合权重：

```
    get_BL_minimum_variance_portfolio(return_table,tau=0.05,P=P,Q=Q,Omega=Om
ega, allow_short=False, show_details=True)
```

运行结果如下：

```
    Minimum Variance Portfolio:
       Short Allowed: False
       Portfolio Return: 0.0658687274541
```

```
        Portfolio Volatility: 0.00505884193248
        Portfolio Weights:
        000300.ZICN: 0.0%
        000905.ZICN: 0.0%
        399006.ZICN: 0.0%
        000013.ZICN: 68.4%
        000012.ZICN: 31.6%

    {'000012.ZICN': 0.3156066104339815,
     '000013.ZICN': 0.68419104722807123,
     '000300.ZICN': 9.9310644937163376e-05,
     '000905.ZICN': 4.5873704735818272e-05,
     '399006.ZICN': 5.7158078857651119e-05}
    get_BL_maximum_utility_portfolio(return_table,tau=0.05,P=P,Q=Q,Omega=Ome
ga, risk_aversion=3., allow_short=False, show_details=True)
```

运行结果如下：

```
    Maximum Utility Portfolio:
        Risk Aversion: 3.0
        Short Allowed: False
        Portfolio Return: 0.160427558899
        Portfolio Volatility: 0.101068004022
        Portfolio Weights:
        000300.ZICN: 0.0%
        000905.ZICN: 0.0%
        399006.ZICN: 25.5%
        000013.ZICN: 74.5%
        000012.ZICN: 0.0%

    {'000012.ZICN': 2.5108544530353643e-08,
     '000013.ZICN': 0.74451864713587357,
     '000300.ZICN': 1.9316138730165122e-07,
     '000905.ZICN': 1.8268669702184613e-07,
     '399006.ZICN': 0.25548095190749759}
    get_maximum_sharpe_portfolio(return_table,riskfree_rate=0.,tau=0.05,P=P,
Q=Q,Omega=Omega,allow_short=True, show_details=True)
```

运行结果如下：

```
    Maximum Sharpe Portfolio:
        Riskfree Rate: 0.0
        Short Allowed: True
```

```
    Portfolio Return: 0.0737396091973
    Portfolio Volatility: 0.00531415124104
    Portfolio Sharpe: 13.8760840354
    Portfolio Weights:
    000300.ZICN: 0.2%
    000905.ZICN: -0.4%
    399006.ZICN: 0.2%
    000013.ZICN: 88.4%
    000012.ZICN: 11.7%

{'000012.ZICN': 0.11651100638381016,
 '000013.ZICN': 0.8844011631137552,
 '000300.ZICN': 0.0015404556032805692,
 '000905.ZICN': -0.004275824074859658,
 '399006.ZICN': 0.0018231989740137779}
```

6.8.3　风险平价模型

假设有两种资产 E_1 和 E_2 可供选择，如果同时投资于这两种资产，权重分别为 w_1、w_2，则组合风险可表示为

$$\sigma_P = \sqrt{w_1^2\sigma_1^2 + w_2^2\sigma_2^2 + 2w_1w_2Cov(R_1, R_2)}$$

用资产 E_1 的权重 w_1 增长带来的组合风险边际增长定义资产 E_1 的边际风险贡献，可表示为

$$MRC_1 = \frac{\partial\sigma_P}{\partial w_1} = \frac{w_1\sigma_1^2 + w_2Cov(R_1, R_2)}{\sigma_P} = \frac{Cov(R_1, R_2)}{\sigma_P}$$

则资产 E_1 对组合的总风险贡献为

$$TRC_1 = w_1MRC_1 = w_1\frac{Cov(R_1, R_2)}{\sigma_P}$$

组合 P 总风险可表示为各项资产总风险贡献之和：

$$\sigma_P = TRC_1 + TRC_2$$

传统资产配置模型的传统大类资产配置方法更多地关注投资组合的总体风险，优化的目标是使组合整体的风险最小，而不关心各个标的对整体风险的贡献。这导致组合整体风险对某个资产暴露极高的风险敞口，从而使得组合的业绩与该资产表现极其相关，并没有

起到很好的分散效果。例如，传统资产配置往往采取 60/40 策略，即 60%投资于股票，40%投资于债券，假设股票风险为 4.5%，债券风险为 1.62%，两者协方差为 0.021%，则组合风险为 2.95%，可计算得到股票对组合的风险贡献为 89.34%，股票风险贡献权重接近 90%，在组合中股票风险占显著主导地位，组合风险并没有得到有效分散。

基于此，PanAgora 基金的首席投资官 Edward Qian 博士提出了著名的风险评价（Risk Parity）策略，被 Bridgewater（桥水联合基金）运用于实际投资中。该策略提倡配置风险，而不是配置资产。传统的资产配置方法控制的是绝对风险，也就是整个投资组合的波动性。而风险平价控制的是相对风险，让各资产类别的风险处于相对平衡的水平。由于组合的风险达到了平衡，所以理论上可以抵御各种风险事件，也就是所谓的“全天候”（All Weather）策略。

用σ表示投资组合的波动率，w_i为资产 i 的权重，$\boldsymbol{\Omega}$为投资组合收益率的协方差矩阵，$(\boldsymbol{\Omega}w)_i$为向量$\boldsymbol{\Omega}w$的第 i 行元素，则资产 i 对组合的风险贡献可以用 $w_i(\boldsymbol{\Omega}w)_i$来表示。要保证每个资产对组合的风险贡献一致，则可以用如下公式优化问题：

$$\min f(w)=\sum_{i=1}^{N}\sum_{j=1}^{N}[w_i(\boldsymbol{\Omega}w)_i-w_j(\boldsymbol{\Omega}w)_j]^2$$

$$\text{s.t.}\quad w^{\boldsymbol{T}}1=1$$

$$w_i\geqslant 0$$

在下面的例子中，我们利用沪深 300、中小板、恒生指数、标普 500 及上证国债构造风险平价组合：

```
import pandas as pd
import numpy as np
from datetime import datetime as dt
import matplotlib.pyplot as plt
from CAL.PyCAL import font
from scipy.optimize import minimize
import seaborn as sns
sns.set_style('whitegrid')


def get_smart_weight(cov_mat, method='min variance', wts_adjusted=False):
    '''
    功能：输入协方差矩阵，得到不同优化方法下的权重配置
    输入：
        cov_mat  pd.DataFrame,协方差矩阵，index 和 column 均为资产名称
```

```
            method  优化方法，可选的有 min variance、risk parity、max diversification、
equal weight
        输出：
            pd.Series  index 为资产名，values 为 weight
        PS:
            依赖 scipy package
        '''

        if not isinstance(cov_mat, pd.DataFrame):
            raise ValueError('cov_mat should be pandas DataFrame! ')

        omega = np.matrix(cov_mat.values)  # 协方差矩阵

        # 定义目标函数
        def fun1(x):
            return np.matrix(x) * omega * np.matrix(x).T

        def fun2(x):
            tmp = (omega * np.matrix(x).T).A1
            risk = x * tmp
            delta_risk = [sum((i - risk)**2) for i in risk]
            return sum(delta_risk)

        def fun3(x):
            den = x * omega.diagonal().T
            num = np.sqrt(np.matrix(x) * omega * np.matrix(x).T)
            return num/den

        # 初始值 + 约束条件
        x0 = np.ones(omega.shape[0]) / omega.shape[0]
        bnds = tuple((0,None) for x in x0)
        cons = ({'type':'eq', 'fun': lambda x: sum(x) - 1})
        options={'disp':False, 'maxiter':1000, 'ftol':1e-20}

        if method == 'min variance':
            res = minimize(fun1, x0, bounds=bnds, constraints=cons,
method='SLSQP', options=options)
        elif method == 'risk parity':
            res = minimize(fun2, x0, bounds=bnds, constraints=cons,
method='SLSQP', options=options)
        elif method == 'max diversification':
```

```
            res = minimize(fun3, x0, bounds=bnds, constraints=cons,
method='SLSQP', options=options)
        elif method == 'equal weight':
            return pd.Series(index=cov_mat.index, data=1.0 / cov_mat.shape[0])
        else:
            raise ValueError('method should be min variance/risk parity/max
diversification/equal weight! ! ! ')

        # 权重调整
        if res['success'] == False:
            # print res['message']
            pass
        wts = pd.Series(index=cov_mat.index, data=res['x'])
        if wts_adjusted == True:
            wts = wts[wts >= 0.0001]
            return wts / wts.sum() * 1.0
        elif wts_adjusted == False:
            return wts
        else:
            raise ValueError('wts_adjusted should be True/False! ')

    def get_dates(start_date, end_date, frequency='daily'):
        '''
        功能：输入起始日期和频率，即可获得日期列表（daily 包括起始日，其余的都是位于起始日
中间的）
        输入参数：
           start_date，开始日期，'xxxxxxxx'形式
           end_date，截止日期，'xxxxxxxx'形式
           frequency，频率，daily 为所有交易日，daily1 为所有自然日，weekly 为每周最后
一个交易日，weekly2 为每隔两周，monthly 为每月最后一个交易日，quarterly 为每季最后一个交
易日
        输出参数：
           获得 list 型日期列表，以'xxxxxxxx'形式存储
        PS:
            要用到 DataAPI.TradeCalGet! ! !
        '''

        data = DataAPI.TradeCalGet(exchangeCD=u"XSHG",beginDate=start_date,
endDate=end_date,field=u"calendarDate,isOpen,isWeekEnd,isMonthEnd,isQuarterE
nd",pandas="1")
        if frequency == 'daily':
```

```
            data = data[data['isOpen'] == 1]
        elif frequency == 'daily1':
            pass
        elif frequency == 'weekly':
            data = data[data['isWeekEnd'] == 1]
        elif frequency == 'weekly2':
            data = data[data['isWeekEnd'] == 1]
            data = data[0:data.shape[0]:2]
        elif frequency == 'monthly':
            data = data[data['isMonthEnd'] == 1]
        elif frequency == 'quarterly':
            data = data[data['isQuarterEnd'] == 1]
        else:
            raise ValueError('调仓频率必须为 daily/daily1/weekly/weekly2/
monthly/quarterly! ! ! ')
        # date_list = map(lambda x: x[0:4]+x[5:7]+x[8:10], data['calendarDate'].
values.tolist())
        date_list = data['calendarDate'].values.tolist()
        return date_list


    def shift_date(date, n, direction='back'):
        '''
        功能：给定 date，获取该日期前、后 n 个交易日对应的交易日
        输入：
            date  'yyyymmdd'类型字符串
            n  非负整数，取值区间（0,720）
            direction  方向，取值为 back/forward
        PS:
            get_dates()
        '''

        last_two_year = str(int(date[:4])-3) + '0101'
        forward_two_year = str(int(date[:4])+3) + '1231'
        if direction == 'back':
            date_list = get_dates(last_two_year, date, 'daily')
            return date_list[len(date_list)-1-n]
        elif direction == 'forward':
            date_list = get_dates(date, forward_two_year, 'daily')
            return date_list[n]
        else:
```

```
            raise ValueError('direction should be back/forward! ! ! ')


    def cal_maxdrawdown(data):
        '''
        功能：给定净值数据（list, np.array, pd.Series, pd.DataFrame），返回最大回撤
        输入：
            data, list/np.array/pd.Series/pd.DataFrame，净值曲线，初始金为 1
        输出：
            list/np.array/pd.Series 返回 float
            pd.DataFrame 返回 pd.DataFrame，index 为 DataFrame.columns
        '''

        if isinstance(data, list):
            data = np.array(data)
        if isinstance(data, pd.Series):
            data = data.values

        def get_mdd(values): # values 为 np.array 的净值曲线，初始资金为 1
            dd = [values[i:].min() / values[i] - 1 for i in range(len(values))]
            return abs(min(dd))

        if not isinstance(data, pd.DataFrame):
            return get_mdd(data)
        else:
            return data.apply(get_mdd)


    def cal_indicators(df_daily_return):
        '''
        功能：给定 daily return，计算各组合的评价指标，包括：年化收益率、年化标准差、夏普
值、最大回撤
        输入：
            df_daily_return  pd.DataFrame，index 为升序排列的日期，columns 为各组合
名称，value 为 daily_return
        '''

        df_cum_value = (df_daily_return + 1).cumprod()
        res = pd.DataFrame(index=['年化收益率','年化标准差','夏普值','最大回撤'],
columns=df_daily_return.columns, data=0.0)
        res.loc['年化收益率'] = (df_daily_return.mean() * 250).apply(lambda x:
```

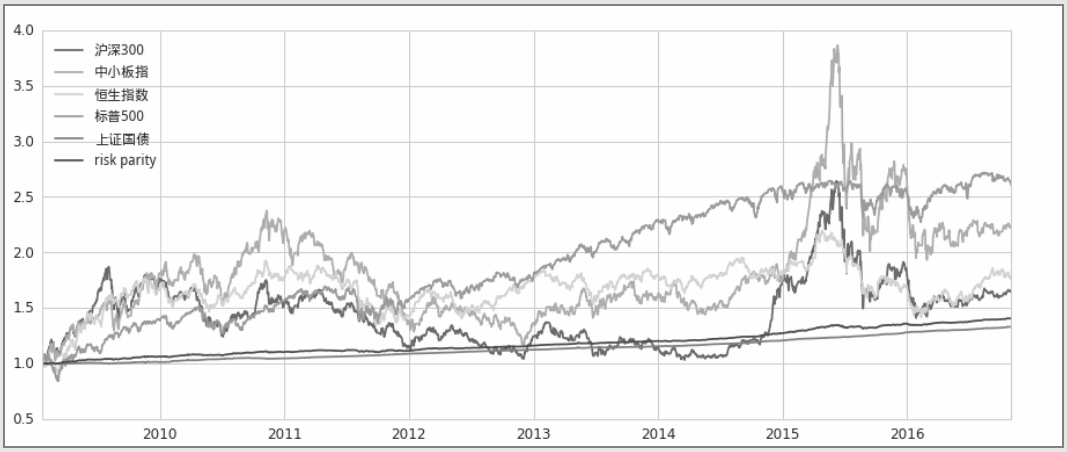
```
'%.2f%%' % (x*100))
        res.loc['年化标准差'] = (df_daily_return.std() * np.sqrt(250)).apply
(lambda x: '%.2f%%' % (x*100))
        res.loc['夏普值'] = (df_daily_return.mean() / df_daily_return.std() *
np.sqrt(250)).apply(lambda x: np.round(x, 2))
        res.loc['最大回撤'] = cal_maxdrawdown(df_cum_value).apply(lambda x:
'%.2f%%' % (x*100))
        return res


    start_date = '2007-01-01'
    end_date = '2017-12-31'
    idx = ['000300', '399005', 'HSI', 'SPX', '000012']  # 沪深 300、中小板指、恒
生指数、标普 500、国债
    data1=DataAPI.MktIdxdGet(ticker=idx,beginDate=start_date,endDate=end_dat
e, field=u"secShortName,tradeDate,CHGPct", pandas="1")
    total_daily_return= data1.pivot(index='tradeDate', columns='secShortName',
values='CHGPct').dropna(how='all').fillna(0.0)
    # total_daily_return.index = map(lambda x: x.replace('-',''),
total_daily_return.index)
    total_daily_return.head()
```

secShortName	上证国债	中小板指	恒生指数	标普 500	沪深 300
tradeDate					
2007-01-03	0.000000	0.000000	0.000000	-0.001199	0.000000
2007-01-04	0.000018	-0.007266	0.003048	0.001228	0.012758
2007-01-05	0.000377	0.032893	0.009273	-0.006085	0.002801
2007-01-08	0.000027	0.022728	-0.008986	0.002220	0.028308
2007-01-09	0.000278	0.010723	-0.006569	-0.000517	0.032150

```
    # backtest
    selected_daily_return = total_daily_return[['沪深 300', '中小板指', '恒生指
数', '标普 500', '上证国债']]
    Ndays = 500   # 用多少天估算协方差矩阵
    starts = shift_date(selected_daily_return.index[0], Ndays, 'forward')
    ends = selected_daily_return.index[-1]
    portfolio_cum_value = pd.DataFrame(index=selected_daily_return.loc
[starts:ends].index, columns=['min variance', 'risk parity', 'max
```

```
diversification','equal weight'], data=0.0)  # 记录组合累计净值
    portfolio_positions = {}  # 记录组合各资产持仓
    allocation_methods = {'min variance', 'risk parity', 'max
diversification','equal weight'}
    for k in allocation_methods:
        portfolio_positions[k] = pd.DataFrame(index=portfolio_cum_value.index,
columns=selected_daily_return.columns, data=0.0)
    date_list = sorted(get_dates(starts, ends, 'quarterly')+[starts, ends])
    for i in range(len(date_list)-1):
        current_period = date_list[i]
        next_period = date_list[i+1]
        tmp_date = shift_date(current_period, Ndays)
        cov_mat = selected_daily_return.loc[tmp_date:current_period].cov()*250
        # 权重优化
        for j in allocation_methods:
            wts = get_smart_weight(cov_mat, method=j, wts_adjusted=False)
            daily_rtn = selected_daily_return.loc[current_period:next_period]
            daily_rtn.ix[0] = 0.0
            assets_positions = (daily_rtn + 1).cumprod() * wts
            portfolio_positions[j].loc[assets_positions.index,:] =
(assets_positions.T / assets_positions.sum(axis=1)).T
            cum_value = assets_positions.sum(axis=1)
            if i == 0:
                portfolio_cum_value.loc[cum_value.index, j] = cum_value * 1.0
            else:
                portfolio_cum_value.loc[cum_value.index, j] = cum_value *
portfolio_cum_value.loc[cum_value.index[0],j]

    assets_cum_value = (total_daily_return.loc[starts:ends] + 1).cumprod()
    assets_cum_value['date'] = map(lambda x: dt.strptime(x, '%Y-%m-%d'),
assets_cum_value.index)

    fig = plt.figure(figsize=(15,6))
    ax = fig.add_subplot(111)
    font.set_size(12)
    colors = ['royalblue','orange','powderblue','sage','gray','dimgray']
    columns = ['沪深 300', '中小板指', '恒生指数', '标普 500', '上证国债']
    for i in range(len(columns)):
        ax.plot(assets_cum_value['date'].values,
assets_cum_value[columns[i]].values, color=colors[i])
```

```
    ax.plot(assets_cum_value['date'].values, portfolio_cum_value['risk
parity'].values, color='r')
    ax.legend([i.decode('utf8') for i in columns] + ['risk parity'], prop=font,
loc='best')
    ax.xaxis.set_tick_params(labelsize=12)
    ax.yaxis.set_tick_params(labelsize=12)
    ax.grid(True)
```




```
    assets_indicators = cal_indicators(selected_daily_return.loc[starts:
ends])
    portfolio_indicators = cal_indicators(portfolio_cum_value.pct_change().
dropna())
    assets_indicators['risk parity'] = portfolio_indicators['risk parity']
    assets_indicators
```

secShortName	沪深 300	中小板指	恒生指数	标普 500	上证国债	risk parity
年化收益率	7.53%	10.43%	10.61%	13.74%	3.02%	3.89%
年化标准差	23.72%	26.32%	19.07%	15.91%	0.85%	1.63%
夏普值	0.32	0.4	0.56	0.86	3.56	2.38
最大回撤	52.42%	55.22%	37.26%	22.60%	1.11%	2.46%

从优化结果中我们可以看到，在收益率上风险平价模型并没有体现优势，但风险平价组合收益异常稳健，在兼具国债安全性的同时获得了其他市场带来的一部分收益，其年化标准差、夏普率、最大回撤表现都很优秀。

6.9 时间序列分析

在金融世界中绝大多数都是金融数据，从股票的价格到 GDP 数值都是随时间变化的变量，其在不同时间的不同值构成的序列即所谓的时间序列。同时，金融时间序列又与其他普通的时间序列有较大的区别：金融理论及金融时间序列都包含不确定因素，例如，对资产波动率有各种不同的定义，一个股票收益率序列的波动率是我们不能直接观察到的。

正因为带有不确定性，所以统计的理论和方法在金融时间序列分析中起到重要的作用。因此，时间序列分析是研究金融数据的重要工具，本节将由浅入深地介绍金融时间序列分析的常用方法及 Python 工具，并展示效果。

6.9.1 与时间序列分析相关的基础知识

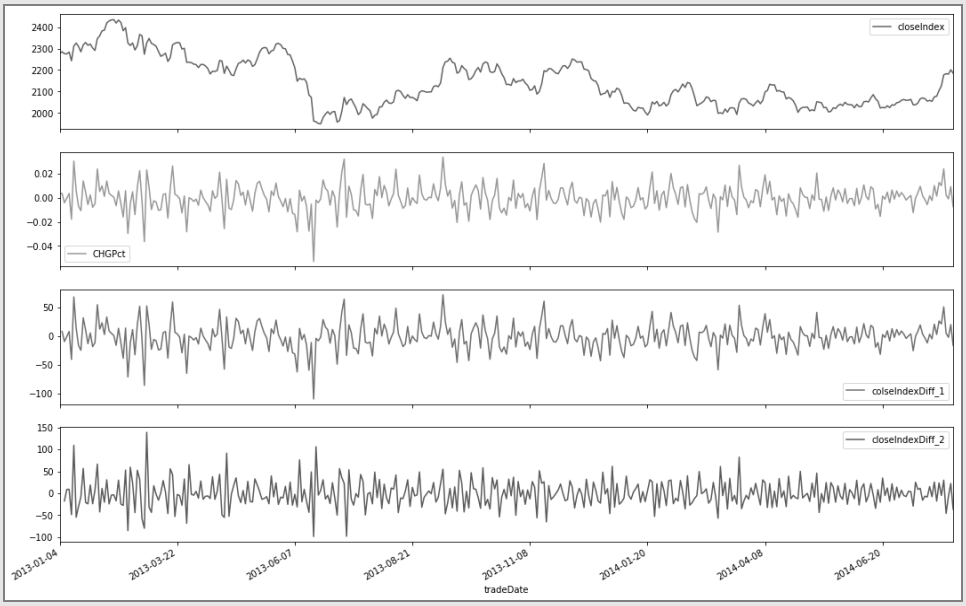
什么是时间序列呢？简而言之，时间序列是指对某个或者某组变量 $x(t)$进行观察和测量，由在一系列时刻 t_1、t_2……t_n所得到的离散数字组成的序列集合。其相关知识点如下。

- 趋势：是时间序列在长时期内呈现出来的持续向上或持续向下的变动。
- 季节变动：是时间序列在一年内重复出现的周期性波动，受诸如气候条件、生产条件、节假日或人们的风俗习惯等各种因素的影响。
- 循环波动：是时间序列呈现出的非固定长度的周期性变动。循环波动的周期可能会持续一段时间，但与趋势不同，它不是朝着单一方向的持续变动，而是涨落相同的交替波动。
- 不规则波动：是在时间序列中除去趋势、季节变动和周期波动之后的随机波动。不规则波动通常夹杂在时间序列中，致使时间序列产生一种波浪形或震荡式的变动。只含有随机波动的序列也被称为平稳序列。

那么什么是平稳性呢？在百度词条中是这样讲的：“粗略地讲，平稳时间序列是一个时间序列，如果均值没有发生系统性变化（无趋势）、方差没有发生系统性变化，且严格消除了周期性变化，就可称之为是平稳的”，如下所示：

```
In [1]:import numpy as np
       import pandas as pd
       import matplotlib.pyplot as plt
```

```
            from scipy import stats
            import statsmodels.api as sm
            %matplotlib inline
    In [2]: IndexData = pd.read_csv('01pct.csv')
            IndexData = IndexData.set_index(IndexData['tradeDate'])
            IndexData['colseIndexDiff_1'] = IndexData['closeIndex'].diff(1)  #
1 阶差分处理
            IndexData['closeIndexDiff_2'] =
IndexData['colseIndexDiff_1'].diff(1)  # 2 阶差分处理
            IndexData.plot(subplots=True,figsize=(18,12))
    Out[2]:
```



上图中第 1 张图为上证综指部分年份的收盘指数，是一个非平稳时间序列；而下面的两张图为平稳时间序列（当然这里没有检验，只是为了让大家看出差异，关于检验序列的平稳性后续会进行讨论），实际上是对第 1 个序列做了差分处理，方差和均值基本平稳，成为了平稳时间序列，我们在后面会谈到这种处理。

下面给出对平稳性的定义。

（1）严平稳：如果对所有的时刻 t，任意正整数 k 和任意 k 个正整数的联合分布 $(t_1,t_2.....t_k)$，$(r_{t_1},r_{t_2},.....r_{t_k})$ 与 $(r_{t_1+t},r_{t_2+t},.....r_{t_k+t})$ 的联合分布相同，则可称时间序列 $\{r_t\}$ 是严平稳的。

也就是说，$(r_{t_1}, r_{t_2}, ...r_{t_k})$ 的联合分布在时间的平移变换下保持不变，是个很强的条件，难以用经验方法验证。因此我们经常假定的是平稳性的一个较弱的方式。

（2）弱平稳：如果时间序列$\{r_t\}$满足“$E(r_t)=\mu, \mu.$是常数$Cov(r_t, r_{t-l})=\gamma_l, \gamma_l$ 且只依赖于 l”的条件，则时间序列$\{r_t\}$为弱平稳的。即该序列的均值 r_t 与 r_{t-l} 的协方差不随时间而改变，l 为任意整数。

在金融数据中，通常我们所说的平稳序列是指弱平稳的序列。

1. 时序中的大数定理 LLN

可能有读者会问为何要“强行”引入弱稳定性的概念，其实时间序列的弱平稳和协方差的绝对可加性，都是为了保证大数定理在时间序列上成立，这是所有统计分析的基础。

2. 差分

差分（这里为前向）就是求时间序列$\{r_t\}$在 t 时刻的值，不妨将 r_t 与 t-1 时刻的值 r_{t-1} 的差记作 $\mathrm{d}t$，则会得到一个新序列$\{\mathrm{d}t\}$，为一阶差分，对新序列$\{\mathrm{d}t\}$再进行同样的操作，则为二阶差分。

非平稳序列通常可以经过 d 次差分，处理成弱平稳或者近似弱平稳的时间序列，我们发现二阶差分得到的序列比一阶差分效果更好。

3. 相关系数

对于两个向量，我们希望定义它们是不是相关，一个很自然的想法是将向量与向量的夹角作为距离的定义，夹角小则距离小，夹角大则距离大。

在学中学数学时，我们就经常使用余弦公式来计算角度：

$$cos<\vec{a},\vec{b}>=\frac{\vec{a}\cdot\vec{b}}{|\vec{a}|\cdot|\vec{b}|}$$

而将 $\vec{a}\cdot\vec{b}$ 叫作内积，例如

$$(x_1, y_1)\cdot(x_2, y_2)=x_1x_2+y_1y_2$$

我们再来看看相关系数的定义公式。$\boldsymbol{X}$ 和 $\boldsymbol{Y}$ 的相关系数为：

$$\rho_{xy}=\frac{Cov(\boldsymbol{X},\boldsymbol{Y})}{\sqrt{Var(\boldsymbol{X})Var(\boldsymbol{Y})}}$$

而根据样本估算的公式为：

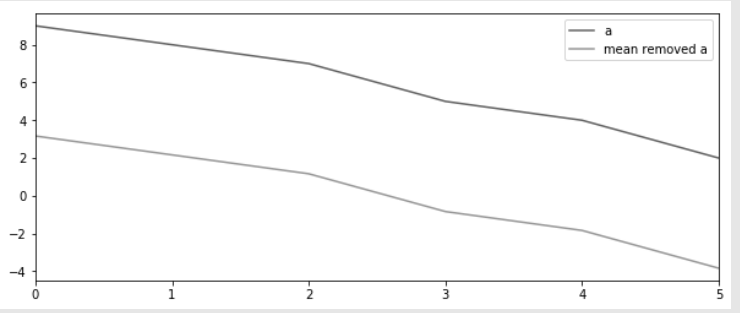
$$\rho_{xy}=\frac{\sum_{t=1}^{T}(\boldsymbol{x}_t-\bar{\boldsymbol{x}})(\boldsymbol{y}_t-\bar{\boldsymbol{y}})}{\sqrt{\sum_{t=1}^{T}(\boldsymbol{x}_t-\bar{\boldsymbol{x}})^2\sum_{t=1}^{T}(\boldsymbol{y}_t-\bar{\boldsymbol{y}})^2}}=\frac{\overrightarrow{(\boldsymbol{X}-\bar{\boldsymbol{x}})}\cdot\overrightarrow{(\boldsymbol{Y}-\bar{\boldsymbol{y}})}}{\left|\overrightarrow{(\boldsymbol{X}-\bar{\boldsymbol{x}})}\right|\cdot\left|\overrightarrow{(\boldsymbol{Y}-\bar{\boldsymbol{y}})}\right|}$$

我们发现，相关系数实际上就是计算了向量空间中两个向量的夹角，协方差是去均值后两个向量的内积。

如果两个向量平行，则相关系数等于 1 或者-1，同向的时候是 1，反向的时候就是-1。如果两个向量垂直，则夹角的余弦等于 0，说明二者不相关。两个向量的夹角越小，相关系数的绝对值越接近 1，相关性越高。只不过这里在计算时对向量做了去均值处理，即中心化操作，而不是直接用向量 $\boldsymbol{X}$、$\boldsymbol{Y}$ 进行计算。

减去均值的操作并不影响角度计算，是一种“平移”的效果，如下所示：

```
In [3]:a = pd.Series([9,8,7,5,4,2])
       b = a - a.mean() # 去均值
       plt.figure(figsize=(10,4))
       a.plot(label='a')
       b.plot(label='mean removed a')
       plt.legend()
Out[3]:
```



4. 自相关函数

相关系数度量了两个向量的线性相关性，而我们有时很想知道在平稳时间序列$\{r_t\}$中

r_t 与它的过去值 r_{t-l} 的线性相关性，这时我们把相关系数的概念推广到自相关系数。

r_t 与 r_{t-l} 的相关系数被称为 r_t 的间隔为 1 的自相关系数，通常记为 ρ_l。具体计算公式为

$$\rho_l = \frac{Cov(r_t, r_{t-l})}{\sqrt{Var(r_t)Var(r_{t-l})}} = \frac{Cov(r_t, r_{t-l})}{Var(r_t)}$$

这里用到了弱平稳序列的性质：

$$Var(r_t) = Var(r_{t-l})$$

可以看到，由于对投资组合添加了一定的噪声，所以导致计算出的因子权重与原先设置的有一点偏差。

对一个平稳时间序列的样本 $\{r_t\}$，$1 \leqslant t \leqslant T$，则对间隔为 1 的样本自相关系数的估算为

$$\hat{\rho}_l = \frac{\sum_{t=l+1}^{T}(r_t - \bar{r})(r_{t-l} - \bar{r})}{\sum_{t=1}^{T}(r_t - \bar{r})^2}, 0 \leqslant l \leqslant T-1$$

则函数 $\hat{\rho}_1$、$\hat{\rho}_2$、$\hat{\rho}_3$ 等被称为 r_t 的样本自相关函数（ACF）。

当在自相关函数中所有的值都为 0 时，我们就认为该序列是完全不相关的，因此，我们经常需要检验多个自相关系数是否为 0。

5. 混成检验

原假设为 $H0: \rho_1 = ... = \rho_m = 0$，备择假设为 $H1: \exists i \in \{1,...,m\}, \rho_i \neq 0$，混成检验统计量的计算公式为

$$Q(m) = T(T+2)\sum_{l=1}^{m}\frac{\hat{\rho}_l^2}{T-l}$$

$Q(m)$渐进服从自由度为 m 的 χ^2 分布。

决策规则为“$Q(m) > \chi_\alpha^2$,拒绝$H0$”，即在 Q(m)的值大于自由度为 m 的卡方分布 100(1-α)分位点时，我们拒绝 $H0$。

大部分软件会给出 $Q(m)$的 *p-value*，则当 *p-value* 小于等于显著性水平 α 时拒绝 $H0$。

示例如下：

```
    In [4]:data = IndexData['closeIndex'] # 上证指数
           m = 10 # 我们检验 10 个自相关系数

           acf,q,p = sm.tsa.acf(data,nlags=m,qstat=True)  ## 计算自相关系数及
p-value
           out = np.c_[range(1,11), acf[1:], q, p]
           output=pd.DataFrame(out, columns=['lag', "AC", "Q", "P-value"])
           output = output.set_index('lag')
           output
    Out[4]:
```

	AC	Q	P-value
lag			
1.0	0.977190	366.688991	9.842648e-82
2.0	0.951513	715.277906	4.779432e-156
3.0	0.927073	1047.065270	1.109192e-226
4.0	0.902993	1362.675600	8.565497e-294
5.0	0.878258	1662.026278	0.000000e+00
6.0	0.857131	1947.908436	0.000000e+00
7.0	0.836825	2221.134260	0.000000e+00
8.0	0.812991	2479.709003	0.000000e+00
9.0	0.789723	2724.350491	0.000000e+00
10.0	0.766245	2955.283132	0.000000e+00

我们取显著性水平为 0.05，可以看出，所有 *p-value* 都小于 0.05，所以我们拒绝原假设 *H*0。因此，我们认为该序列（即上证指数序列）是序列相关的。

再来看看同期上证指数的日收益率序列：

```
    In [5]:data2 = IndexData['CHGPct'] # 上证指数日涨跌
           m = 10 # 我们检验 10 个自相关系数

           acf,q,p = sm.tsa.acf(data2,nlags=m,qstat=True)  ## 计算自相关系数及
p-value
           out = np.c_[range(1,11), acf[1:], q, p]
           output=pd.DataFrame(out, columns=['lag', "AC", "Q", "P-value"])
           output = output.set_index('lag')
```

```
        output
Out[5]:
```

	AC	Q	P-value
lag			
1.0	0.065301	1.637490	0.200670
2.0	-0.014762	1.721391	0.422868
3.0	-0.022240	1.912341	0.590798
4.0	0.008116	1.937837	0.747191
5.0	-0.049070	2.872308	0.719664
6.0	-0.064429	4.487638	0.610989
7.0	0.080509	7.016621	0.427151
8.0	0.010938	7.063429	0.529805
9.0	0.028106	7.373304	0.598314
10.0	0.080362	9.913438	0.448120

可以看出，*p-value* 均大于显著性水平 0.05。我们选择假设 H0，即上证指数日收益率序列没有显著的相关性。

6. 白噪声序列

如果随机变量 $X(t)$（t=1、2、3……）是由一个不相关的随机变量的序列构成的，即对于所有 S 不等于 T，随机变量 X_t 和 X_s 的协方差为零，则称其为纯随机过程。

对于一个纯随机过程来说，如果其期望和方差均为常数，则称之为白噪声过程。白噪声过程的样本实称被称为白噪声序列，简称白噪声。之所以称之为白噪声，是因为它和白光的特性类似，白光的光谱在各个频率上都有相同的强度，白噪声的谱密度在各个频率上的值相同。特别地，如果 X_t 还服从均值为 0、方差为 σ^2 的正态分布，则称这个序列为高斯白噪声。

7. 线性时间序列

时间序列 $\{r_t\}$ 如果能被写成

$$r_t = \mu + \sum_{i=0}^{\infty} \psi_i a_{t-i} \quad （\mu 为 r_t \text{ 的均值，} \psi_0 = 1, \{a_t\} 为白噪声序列）$$

则我们称$\{r_t\}$为线性序列。其中a_t被称为在t时刻的新息（Innovation）或扰动（Shock）。

很多时间序列是线性的，即为线性的时间序列，所以有很多相应的线性时间序列模型，例如接下来要介绍的 AR、MA、ARMA 都是线性模型，但并不是所有的金融时间序列都是线性的。

对于弱平稳序列，我们利用白噪声的性质很容易得到r_t的均值和方差：

$$E(r_t)=\mu, Var(r_t)=\sigma_a^2\sum_{i=0}^{\infty}\psi_i^2 \quad （\sigma_a^2为a_t的方差）$$

因为$Var(r_t)$一定小于正无穷，因此$\{\psi_i^2\}$必须是收敛序列，因此满足

$$i\to\infty时，\ \psi_i^2\to 0$$

即随着i的增大，远处的扰动a_{t-i}.对r_t的影响会逐渐消失。

本节介绍了一些基本知识和概念，例如平稳性、相关性、白噪声和线性序列，并没有深入介绍，对读者目前来说是“够用”的，在后面的章节中补充一些细节。下面开始介绍一些线性模型。

6.9.2　自回归（AR）模型

我们在 6.10.1 节计算了上证指数部分数据段的 ACF，可知间隔为 1 时自相关系数是显著的，这说明在t-1 时刻的数据r_{t-1}在预测t时刻的r_t时可能是有用的，我们根据这一点可以建立下面的模型：

$$r_t=\phi_0+\phi_1 r_{t-1}+a_t$$

其中$\{a_t\}$是白噪声序列。这个模型与简单的线性回归模型有相同的形式，也叫作一阶自回归（AR）模型，简称 AR(1)模型。

从 AR(1)很容易推广到 AR(p)模型：

$$r_t=\phi_0+\phi_1 r_{t-1}+\cdots+\phi_p r_{t-p}+a_t$$

1. AR(p)模型的特征根及平稳性检验

我们先假定序列是弱平稳的，则有$\begin{cases}E(r_t)=\mu\\ Var(r_t)=\gamma_0\\ Cov(r_t,r_{t-j})=\gamma_j\end{cases}$，其中$\mu$及$\gamma_0$是常数。

因为$\{a_t\}$是白噪声序列，因此有

$$E(a_t)=0, Var(a_t)=\sigma_a^2$$

所以

$$E(r_t)=\phi_0+\phi_1 E(r_{t-1})+\phi_2 E(r_{t-2})+\cdots+\phi_p E(r_{t-p})$$

根据平稳性的性质，又有

$$E(r_t)=E(r_{t-1})=\ldots=u$$

所以

$$\mu=\phi_0+\phi_1\mu+\phi_2\mu+\cdots+\phi_p\mu E(r_t)=\mu=\frac{\phi_0}{1-\phi_1-\phi_2-\cdots-\phi_p}$$

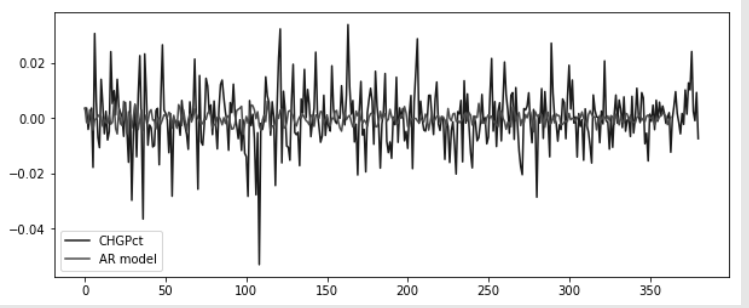
对上式而言，假定分母不为 0，则我们将下面的方程称为特征方程：

$$1-\phi_1 x-\phi_2 x^2-\cdots-\phi_p x^p=0$$

该方程所有解的倒数被称为该模型的特征根，如果所有特征根的模都小于 1，则该AR(p)序列是平稳的。之所以会有对特征根的限制，是为了保证$Var(r_t)$的存在。

下面我们就用该方法检验上证指数日收益率序列的平稳性：

```
In [6]:temp = np.array(data2) # 载入收益率序列
       model = sm.tsa.AR(temp)
       results_AR = model.fit()
       plt.figure(figsize=(10,4))
       plt.plot(temp,'b',label='CHGPct')
       plt.plot(results_AR.fittedvalues, 'r',label='AR model')
       plt.legend()
Out[6]:
```

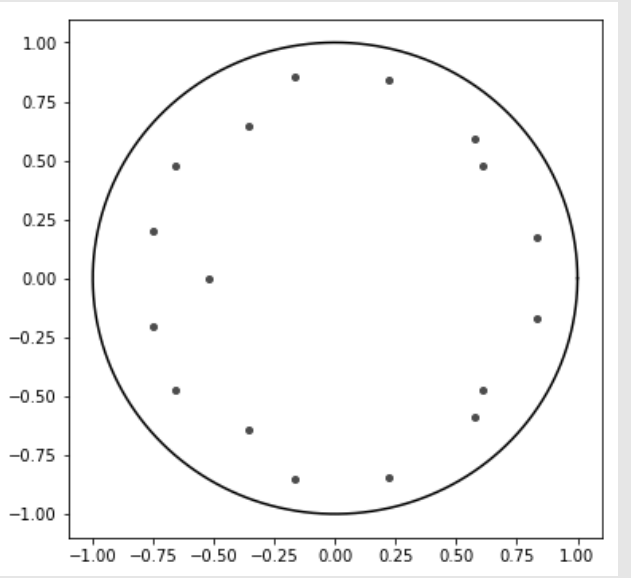


我们可以看看模型有多少阶：

```
In [7]:print(len(results_AR.roots))
Out[7]:17
```

可以看出，自动生成的 AR 模型是 17 阶的。下一节再进行关于阶次的讨论。我们画出模型的特征根，来检验平稳性：

```
In [8]:pi,sin,cos = np.pi,np.sin,np.cos
       r1 = 1
       theta = np.linspace(0,2*pi,360)
       x1 = r1*cos(theta)
       y1 = r1*sin(theta)
       plt.figure(figsize=(6,6))
       plt.plot(x1,y1,'k')  # 画单位圆
       roots = 1/results_AR.roots
       # 注意，这里 results_AR.roots 是计算的特征方程的解，特征根应该取倒数
       for i in range(len(roots)):
           plt.plot(roots[i].real,roots[i].imag,'.r',markersize=8) #画特征根
       plt.show()
```



可以看出，所有特征根都在单位圆内，所以序列为平稳的。

2. AR(*p*)模型的定阶

一般有两种方法来决定 p：

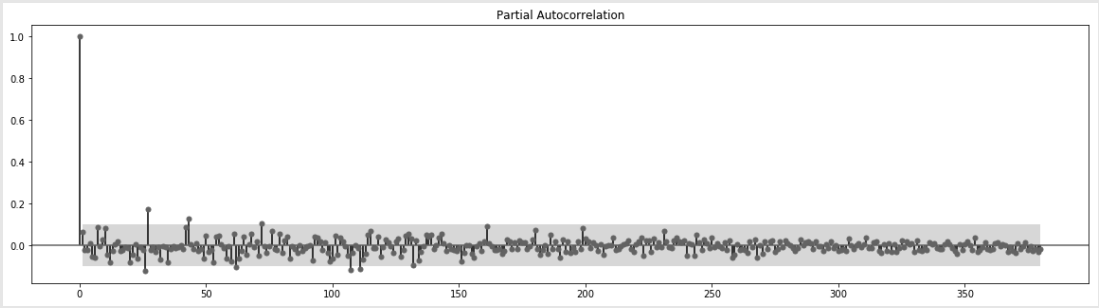
◎ 利用偏相关函数（Partial Auto Correlation Function，PACF）；

◎ 利用信息准则函数。

这里重点介绍偏相关函数的一个性质：AR(p)序列的样本偏相关函数是 p 步截尾的。截尾指快速收敛应该是快速降到几乎为 0 或者在置信区间以内。

具体看看下面的例子，还是基于之前的上证指数日收益率序列：

```
In [9]:fig = plt.figure(figsize=(20,5))
       ax1=fig.add_subplot(111)
       fig = sm.graphics.tsa.plot_pacf(temp,ax=ax1)
```



按照截尾来看，模型阶次 p 在 110 以上，但是之前调用的自动生成的 AR 模型，阶数为 17。当然，我们很少会用到这么高的阶次。

3. 信息准则：AIC、BIC 和 HQ

对于这么多可供选择的模型，我们通常采用 AIC 法则。我们知道，自由参数数量的增加提高了拟合的优良性，AIC 鼓励数据拟合的优良性，但是尽量避免出现过度拟合（Overfitting）的情况，所以我们优先考虑的模型应是 AIC 值最小的。赤池信息准则是寻找可以更好地解释数据但包含更少的自由参数的模型。目前选择模型除了会采用 AIC 法则，还常采用如下准则：

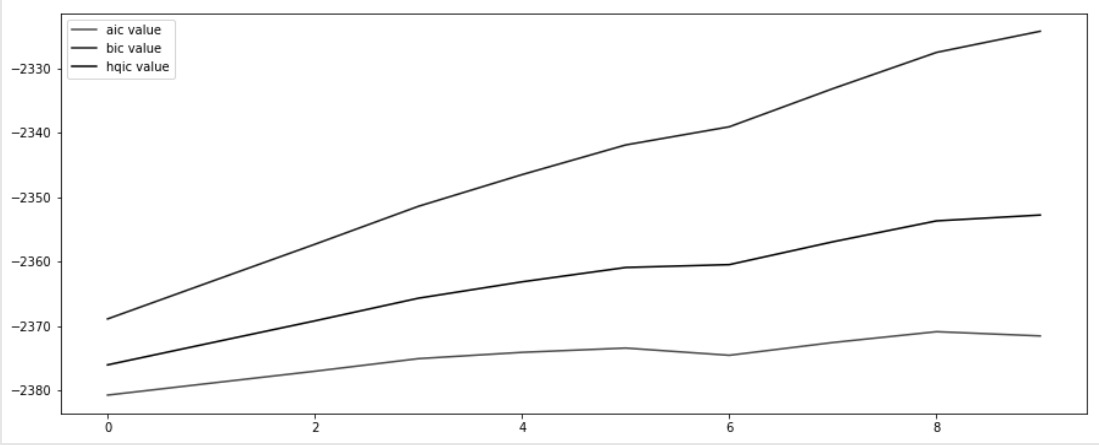
◎ AIC = -2 ln(L) + 2k，即赤池信息量（Akaike Information Criterion）。

◎ BIC = -2 ln(L) + ln(n) × k，即贝叶斯信息量（Bayesian Information Criterion）。

◎　HQ = -2 ln(L) + ln(ln(n)) × k，即 Hannan-Quinn Criterion。

下面测试在以上 3 个准则下确定的 p，仍然用上证指数日收益率序列。为了减少计算量，我们只计算前 10 个来看看效果：

```
In [10]:aicList = []
         bicList = []
         hqicList = []
         for i in range(1,11):  #从 1 阶开始算
             order = (i,0)  # 这里使用了 ARMA 模型，order 代表模型的(p,q)值，我们令 q 始终为 0，只考虑了 AR 情况。
             tempModel = sm.tsa.ARMA(temp,order).fit()
             aicList.append(tempModel.aic)
             bicList.append(tempModel.bic)
             hqicList.append(tempModel.hqic)
In [11]:plt.figure(figsize=(15,6))
         plt.plot(aicList,'r',label='aic value')
         plt.plot(bicList,'b',label='bic value')
         plt.plot(hqicList,'k',label='hqic value')
         plt.legend(loc=0)
```



可以看出，3 个准则在第 1 点均取到最小值，也就是说，p 的最佳取值应该为 1，我们只计算了前 10 个，结果未必正确。总之，我们的目的是了解方法。

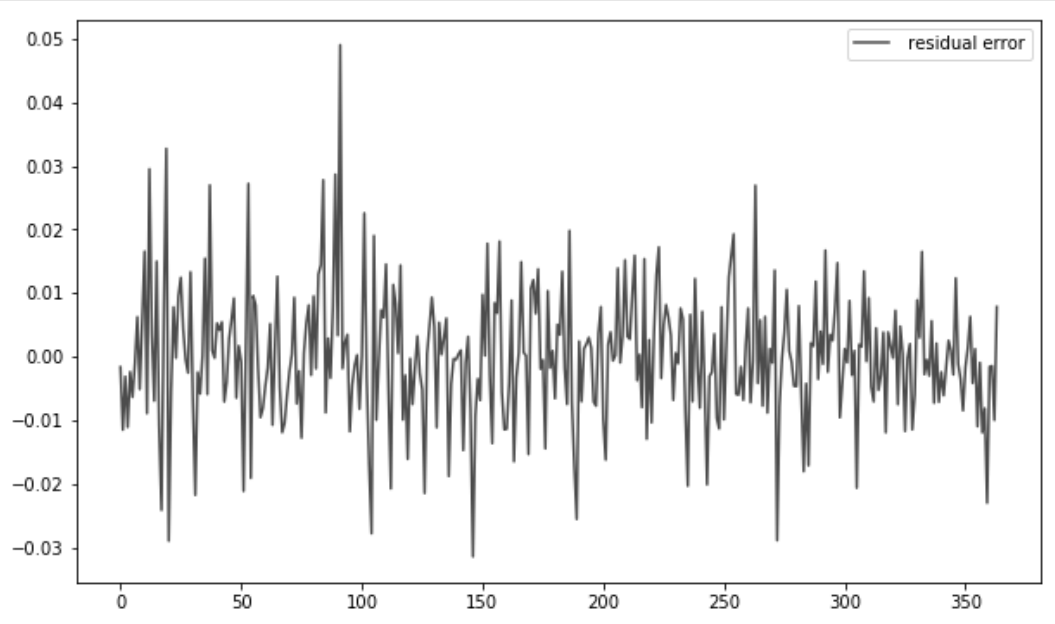
当然，利用上面的方法进行逐个计算是很耗时间的，实际上，有的函数可以直接按照准则计算出合适的 order，这是针对 ARMA 模型的，后续再进行讨论。

4. 模型的检验

如果模型是充分的，则其残差序列应该是白噪声，我们在第 1 章里讲到的混成检验可以用于检验残差与白噪声的接近程度。

先求出残差序列：

```
In [12]: delta = results_AR.fittedvalues  - temp[17:]  # 残差
         plt.figure(figsize=(10,6))
         plt.plot(delta,'r',label=' residual error')
         plt.legend(loc=0)
```




```
In [13]: acf,q,p = sm.tsa.acf(delta,nlags=10,qstat=True)  ## 计算自相关系数及 p-value
         out = np.c_[range(1,11), acf[1:], q, p]
         output=pd.DataFrame(out, columns=['lag', "AC", "Q", "P-value"])
         output = output.set_index('lag')
         output
Out[13]:
```

	AC	Q	P-value
lag			
1.0	-0.001225	0.000551	0.981277
2.0	-0.007836	0.023149	0.988492
3.0	-0.002310	0.025119	0.998949
4.0	-0.007185	0.044223	0.999759
5.0	-0.000229	0.044243	0.999978
6.0	-0.001313	0.044884	0.999998
7.0	-0.005752	0.057229	1.000000
8.0	-0.005667	0.069248	1.000000
9.0	-0.011054	0.115102	1.000000
10.0	0.004700	0.123416	1.000000

5. 拟合优度

我们使用下面的统计量来衡量拟合优度：

$$R^2 = 1 - \frac{\text{残差的平方和}}{\text{总的平方和}}$$

但是，对于一个给定的数据集，R^2 是非降函数，为了克服该缺点，推荐使用调整后的 R^2：

$$AdjR^2 = 1 - \frac{\text{残差的平方}}{r_i\text{的方差}}$$

它的值在 0 和 1 之间，越接近 1，拟合效果越好。

下面计算之前对上证指数日收益率的 AR 模型的拟合程度：

```
In [14]: score = 1 - delta.var()/temp[17:].var()
         print(score)
Out[14]: 0.0405231650285
```

可以看出，模型的拟合程度并不好，当然这并不重要，也许是这个序列并不适合用 AR 模型拟合。

6. 预测

我们首先把原来的样本分为训练集和测试集，再来看预测效果，还是以之前的数据为例：

```
In [15]:train = temp[:-10]
        test = temp[-10:]
        output = sm.tsa.AR(train).fit()
        output.predict()
Out[15]:array([  3.42464295e-03,  -2.01270731e-03,  -2.12777697e-03,
        2.45802768e-03,    9.25838275e-04,  -4.65826593e-03,
        -8.85754456e-04,  -6.37079284e-04,   4.08355465e-04,….])
In [16]:predicts = output.predict(355, 364, dynamic=True)
        print(len(predicts))
        comp = pd.DataFrame()
        comp['original'] = temp[-10:]
        comp['predict'] = predicts
        comp
Out[16]:
```

	original	predict
0	-0.002228	0.000909
1	0.010223	-0.001472
2	0.001449	-0.002108
3	0.012785	-0.002158
4	0.010238	-0.000218
5	0.024139	0.000115
6	0.002408	0.001476
7	-0.000893	-0.000119
8	0.009315	-0.000520
9	-0.007385	-0.000161

6.9.3 滑动平均（MA）模型

这里直接给出 MA(q)模型的形式：

$$r_t = c_0 + a_t - \theta_1 a_{t-1} - \cdots - \theta_q a_{t-q}$$

c_0 是一个常数项。这里的 a_t 是 AR 模型在 t 时刻的扰动或者信息，可以发现，该模型使用了过去 q 个时期的随机干扰或预测误差来线性表达当前的预测值。

1. MA 模型的性质

MA 模型的性质如下。

（1）平稳性。MA 模型总是弱平稳的，因为它们是白噪声序列（残差序列）的有限线性组合。因此，根据弱平稳的性质可以得出：

$$E(r_t) = c_0 Var(r_t) = (1 + \theta_1^2 + \theta_2^2 + \cdots + \theta_q^2)\sigma_a^2$$

（2）自相关函数。对 q 阶的 MA 模型，其自相关函数 ACF 总是 q 步截尾的，因此 MA(q)序列只与其前 q 个延迟值线性相关，所以它是一个“有限记忆”的模型。

这一点可以用于确定模型的阶次，后面会进行介绍。

（3）可逆性。在满足可逆条件时，可以将 MA(q)模型改写为 AR(p)模型。这里不进行推导，只给出 1 阶和 2 阶 MA 的可逆性条件。

1 阶为

$$|\theta_1| < 1$$

2 阶为

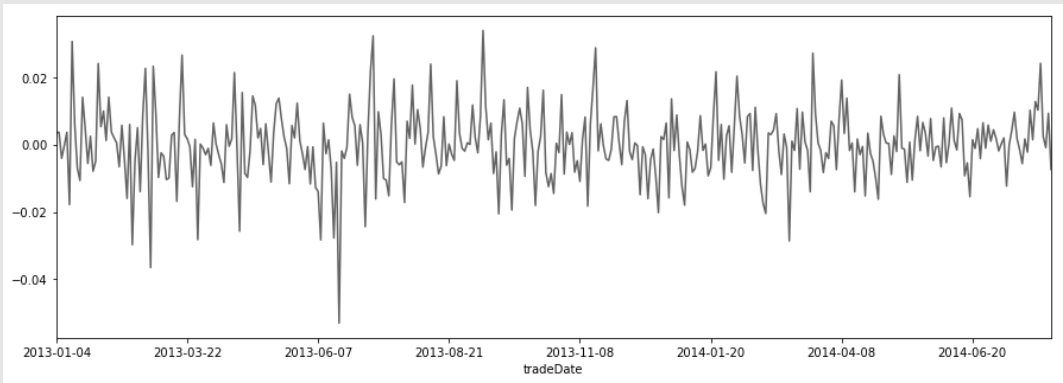
$$|\theta_2| < 1, \theta_1 + \theta_2 < 1$$

2. MA 的阶次判定

我们通常利用上面介绍的第 2 条性质：MA(q)模型的 ACF 函数 q 步截尾来判断模型阶次，示例如下。

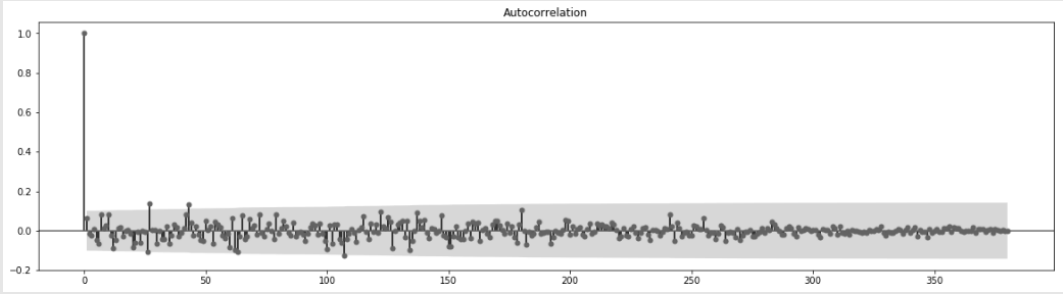
使用上证指数的日涨跌数据（从 2013 年 1 月至 2014 年 8 月）来进行分析，先取数据：

```
In [17]: data = np.array(IndexData['CHGPct']) # 上证指数日涨跌
         IndexData['CHGPct'].plot(figsize=(15,5))
```



可以看出，序列看上去是弱平稳的。下面画出序列的 ACF：

```
In [18]: fig = plt.figure(figsize=(20,5))
         ax1=fig.add_subplot(111)
         fig = sm.graphics.tsa.plot_acf(data,ax=ax1)
```



我们发现 ACF 函数在 43 处截尾，之后的 ACF 函数均在置信区间内，我们判定该序列的 MA 模型阶次为 43 阶。

3. 建模和预测

由于在 sm.tsa 中没有单独的 MA 模块，所以这里用到了 ARMA 模块，只要将其中 AR 的阶 p 设为 0 即可。

函数 sm.tsa.ARMA 的输入参数中的 order(p,q)代表了 AR 和 MA 的阶次。模型阶次增高，计算量急剧增长，因此这里建立 10 阶的模型作为示例，如果按上一节的判断阶次来建模，则计算时间过长。

我们用最后 10 个数据作为 out-sample 的样本，来对比预测值：

```
In [19]: order = (0,10)
         train = data[:-10]
         test = data[-10:]
         tempModel = sm.tsa.ARMA(train,order).fit()
```

先来看看拟合效果，计算公式为

$$AdjR^2 = 1 - \frac{\text{残差的平方}}{r_i\text{的方差}}$$

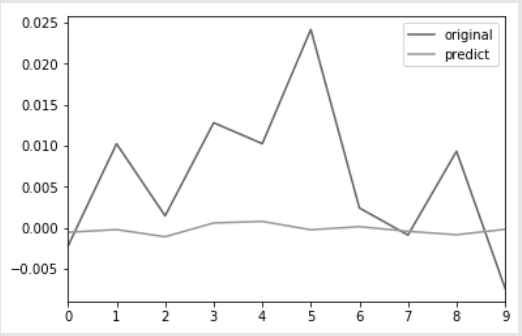
代码如下：

```
In [20]: delta = tempModel.fittedvalues - train
         score = 1 - delta.var()/train.var()
         print(score)
Out[20]: 0.0278739998881
```

可以看出，score 远小于 1，拟合效果不好。

然后，我们用建立的模型预测最后 10 个数据：

```
In [21]: predicts = tempModel.predict(371, 380, dynamic=True)
         print(len(predicts))
         comp = pd.DataFrame()
         comp['original'] = test
         comp['predict'] = predicts
         comp.plot()

Out[21]: 10
```



可以看出，建立的模型效果很差，预测值明显小了 1 到 2 个数量级！就算只看涨跌方向，正确率也不足 50%，所以该模型不适用于原数据。不过没关系，如果预测指数日涨跌真的有这么简单就太奇怪了。

6.9.4 自回归滑动平均（ARMA）模型

在某些应用中，我们需要高阶的 AR 或 MA 模型才能充分地描述数据的动态结构，这样问题会变得很烦琐。为了解决这个问题，自回归滑动平均（ARMA）模型应运而生，其基本思想是把 AR 和 MA 模型结合在一起，使所使用的参数数量保持很少。

模型的形式为

$$r_t = \phi_0 + \sum_{i=1}^{p} \phi_i r_{t-i} + a_t + \sum_{i=1}^{q} \theta_i a_{t-i}$$

其中，$\{a_t\}$为白噪声序列，p 和 q 都是非负整数。AR 和 MA 模型都是 ARMA(p,q)的特殊形式。

利用向后推移算子 B（即上一时刻），上述模型可写为

$$(1-\phi_1 B-\cdots-\phi_p B^p) r_t = \phi_0 + (1-\theta_1 B-\cdots-\theta_q B^q) a_t$$

这时求 r_t 的期望，得到：

$$E(r_t) = \frac{\phi_0}{1-\phi_1-\cdots-\phi_p}$$

和上期我们的 AR 模型一模一样。因此有着相同的特征方程：

$$1-\phi_1 x-\phi_2 x^2-\cdots-\phi_p x^p = 0$$

该方程所有解的倒数被称为该模型的特征根，如果所有特征根的模都小于 1，则该 ARMA 模型是平稳的。

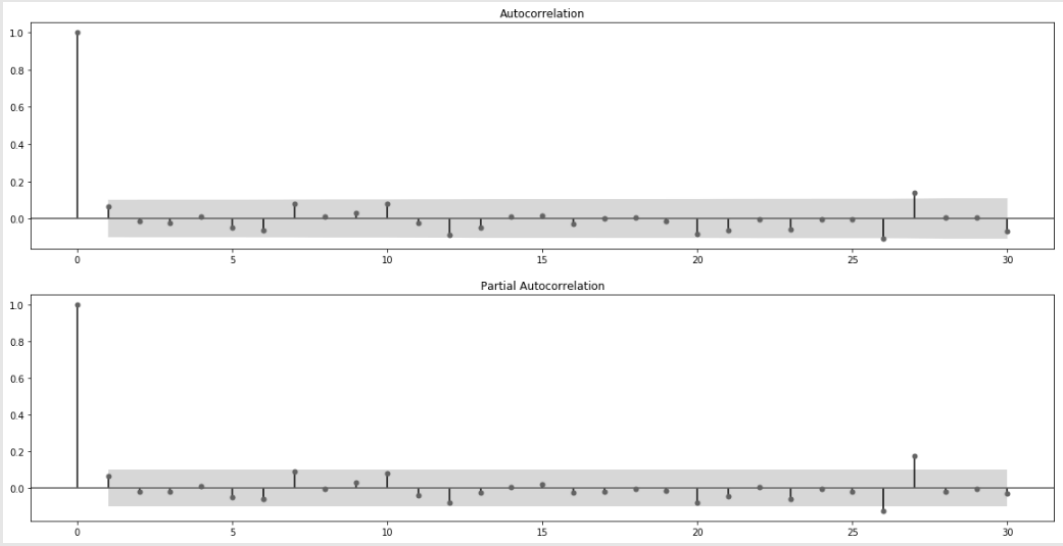
有一点很关键：ARMA 模型的应用对象应该为平稳序列，我们下面的步骤都是建立在假设原序列平稳的条件下的。

1. PACF、ACF 判断模型阶次

我们通过观察 PACF 和 ACF 截尾，分别判断 p、q 的值（限定滞后阶数 50）：

```
In [22]:data = np.array(IndexData['CHGPct']) # 上证指数日涨跌

         fig = plt.figure(figsize=(20,10))
         ax1=fig.add_subplot(211)
         fig = sm.graphics.tsa.plot_acf(data,lags=30,ax=ax1)
```

```
            ax2 = fig.add_subplot(212)
            fig = sm.graphics.tsa.plot_pacf(data,lags=30,ax=ax2)
```



可以看出，模型的阶次应该为(27,27)。然而，这么高的阶次建模的计算量是巨大的。

为什么不再将滞后阶数限制小一些？如果将 lags 设置为 25、20 或者更小，则阶数为(0,0)，这显然不是我们想要的结果。

综合来看，由于计算量太大，在这里就不使用(27,27)建模了，而是采用另一种方法来确定阶数。

2. 信息准则定阶

在 6.10.2 节讲到过，目前选择模型常用 AIC、BIC、HQ 准则。我们常用的是 AIC 准则，AIC 鼓励数据拟合的优良性，但是尽量避免出现过度拟合（Overfitting）的情况，所以优先考虑的模型应是 AIC 值最小的那个模型。

下面分别应用以上 3 种法则为我们的模型定阶，数据仍然是上证指数日涨跌幅序列。为了控制计算量，我们限制 AR 的最大阶不超过 6，MA 的最大阶不超过 4。但是这样带来的坏处是可能为局部最优：

```
    In [23]:print("AIC", sm.tsa.arma_order_select_ic(data,max_ar=6,max_ma=4,
ic='aic')['aic_min_order'])  # AIC
            print("BIC", sm.tsa.arma_order_select_ic(data,max_ar=6,max_ma=4,
```

```
ic='bic')['bic_min_order'])  # BIC
           print("HQIC", sm.tsa.arma_order_select_ic(data,max_ar=6,max_ma=4,
ic='hqic')['hqic_min_order']) # HQIC
    Out[23]:
    AIC (3, 3)
    BIC (0, 0)
    HQIC (0, 0)
```

可以看出，AIC 准则求解的模型阶次为(3,3)。这里以 AIC 准则为准，至于到底哪种准则更好，读者可以分别建模进行对比。

3. 模型的建立及预测

AIC 准则求解的模型阶次(3,3)来建立 ARMA 模型，源数据为上证指数日涨跌幅数据，最后 10 个数据用于预测：

```
    In [24]:order = (3,2)
            train = data[:-10]
            test = data[-10:]
            tempModel = sm.tsa.ARMA(train,order).fit()
```

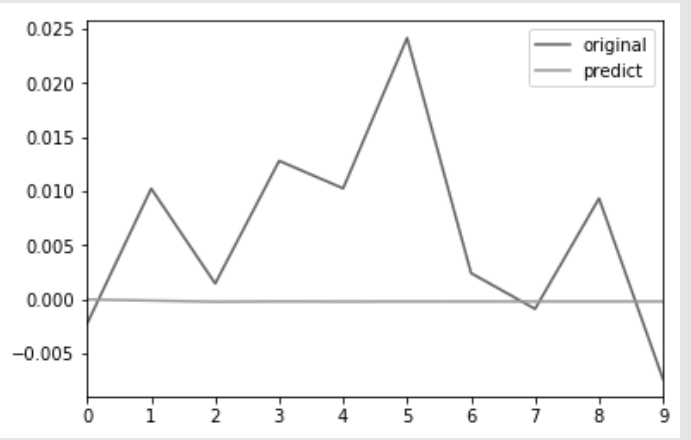
同样，先来看看拟合效果：

```
    In [25]:delta = tempModel.fittedvalues - train
            score = 1 - delta.var()/train.var()
            print(score)
    Out[25]:0.0055187509265
```

如果对比之前建立的 AR、MA 模型，则可以发现在拟合精度上有所提升，但仍然准确度欠佳。

接下来看看预测效果：

```
    In [26]:predicts = tempModel.predict(371, 380, dynamic=True)
            print(len(predicts))
            comp = pd.DataFrame()
            comp['original'] = test
            comp['predict'] = predicts
            comp.plot()
    Out[26]:10
```



可以看出，虽然精确性还是很差，不过相比之前的 MA 模型，只看涨跌的话，胜率为 55.6%，效果还是好了不少。

6.9.5　自回归差分滑动平均（ARIMA）模型

到目前为止，我们研究的序列都集中在平稳序列，即 ARMA 模型研究的对象为平稳序列。如果序列是非平稳的，则可以考虑使用 ARIMA 模型。

ARIMA 比 ARMA 仅多了个“I”，代表其比 ARMA 多了一层内涵，也就是差分。

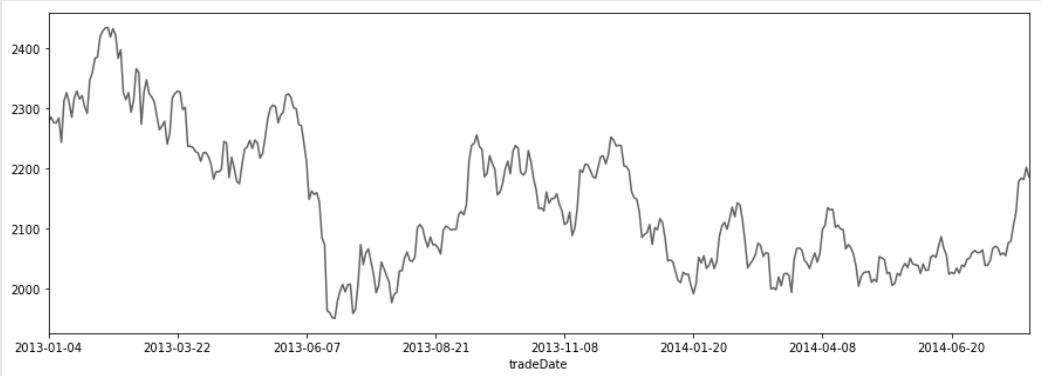
一个非平稳序列在经过 *d* 次差分后，可以转化为平稳时间序列。关于 *d* 的具体取值，我们对差分 1 次后的序列进行平稳性检验，如果是非平稳的，则继续差分，直到 *d* 次后检验为平稳序列。

1. 单位根检验

ADF 是一种常用的单位根检验方法，它的原假设为序列具有单位根，即非平稳。对于一个平稳的时序数据，就需要在给定的置信水平上显著，拒绝原假设。

下面给出示例，先看看上证综指的日指数序列：

```
In [27]:data2 = IndexData['closeIndex'] # 上证指数
        data2.plot(figsize=(15,5))
```



可以看出，该序列显然是非平稳的。接着进行 ADF 单位根检验：

```
    In [28]:temp = np.array(data2)
            t = sm.tsa.stattools.adfuller(temp)  # ADF 检验
            output=pd.DataFrame(index=['Test Statistic Value', "p-value",
"Lags Used", "Number of Observations Used","Critical Value(1%)","Critical
Value(5%)","Critical Value(10%)"],columns=['value'])
            output['value']['Test Statistic Value'] = t[0]
            output['value']['p-value'] = t[1]
            output['value']['Lags Used'] = t[2]
            output['value']['Number of Observations Used'] = t[3]
            output['value']['Critical Value(1%)'] = t[4]['1%']
            output['value']['Critical Value(5%)'] = t[4]['5%']
            output['value']['Critical Value(10%)'] = t[4]['10%']
            output
    Out[28]:
```

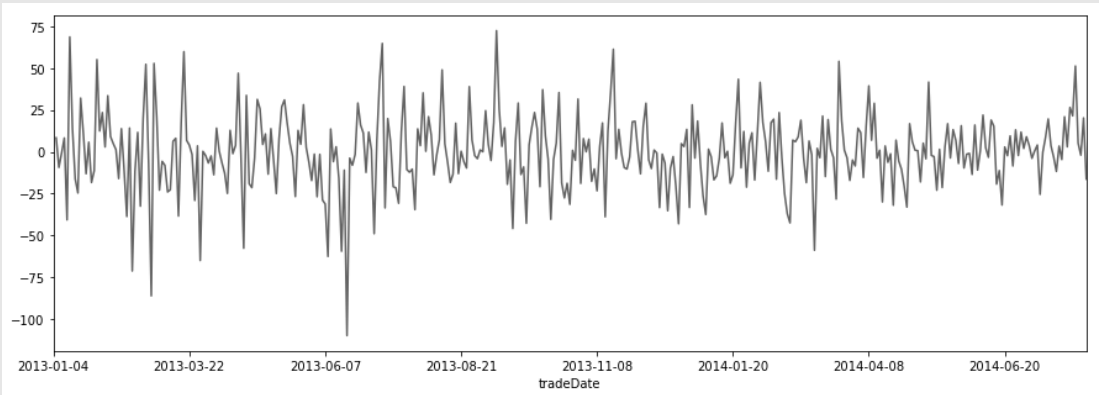
	value
Test Statistic Value	-2.30472
p-value	0.170449
Lags Used	1
Number of Observations Used	379
Critical Value(1%)	-3.44772
Critical Value(5%)	-2.8692
Critical Value(10%)	-2.57085

可以看出，p-value 为 0.170 448 9，大于显著性水平。原假设为：序列具有单位根，

即非平稳，不能被拒绝。因此，上证指数的日指数序列是非平稳的。

我们将序列进行 1 次差分后再次检验：

```
In [29]: data2Diff = data2.diff()  # 差分
         data2Diff.plot(figsize=(15,5))
```



可以看出，序列近似平稳序列。接着进行 ADF 检验：

```
In [30]: temp = np.array(data2Diff)[1:] # 差分后的第 1 个值为 NaN，舍去
         t = sm.tsa.stattools.adfuller(temp)  # ADF 检验
         print("p-value:   ",t[1])
Out[30]: p-value:    2.31245750144e-30
```

可以看出，*p-value* 非常接近于 0，拒绝原假设，因此该序列为平稳的。

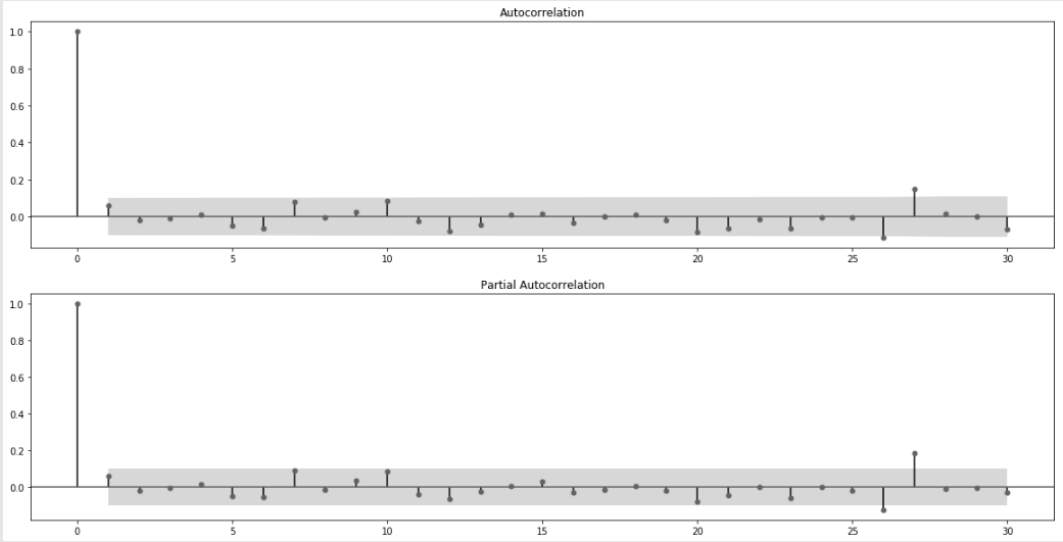
可见，经过 1 次差分后的序列是平稳的，对于原序列，*d* 的取值为 1 即可。

2. ARIMA(p,d,q)模型阶次确定

我们在上面一小节确定了差分次数 *d*，接下来就可以将差分后的序列建立 ARMA 模型。

首先，还是尝试通过 PACF 和 ACF 来判断 *p*、*q*：

```
In [31]: temp = np.array(data2Diff)[1:] # 差分后的第 1 个值为 NaN，舍去
         fig = plt.figure(figsize=(20,10))
         ax1 = fig.add_subplot(211)
         fig = sm.graphics.tsa.plot_acf(temp,lags=30,ax=ax1)
         ax2 = fig.add_subplot(212)
         fig = sm.graphics.tsa.plot_pacf(temp,lags=30,ax=ax2)
```



可以看出，模型的阶次为(27,27)，还是太高了。建模计算量太大，我们再看看 AIC 准则：

```
    In [32]:sm.tsa.arma_order_select_ic(temp,max_ar=6,max_ma=4,ic='aic')['ai
c_min_order']  # AIC
    Out[32]:(2, 2)
```

根据 AIC 准则，差分后的序列的 ARMA 模型阶次为(2,2)。因此，我们要建立的 ARIMA 模型阶次(p,d,q) = (2,1,2)。

3. ARIMA 模型建立及预测

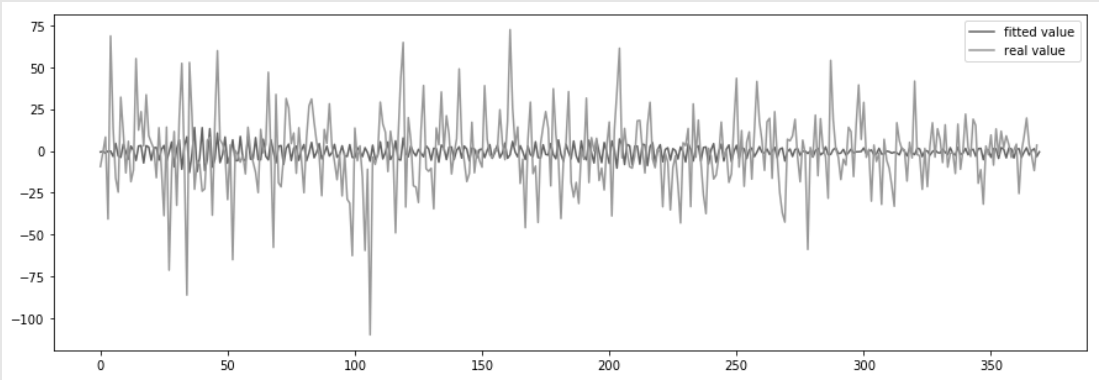
根据上一结果确定的模型阶次，我们对差分后的序列建立 ARMA(2,2)模型：

```
    In [33]:order = (2,2)
            data = np.array(data2Diff)[1:] # 差分后的第 1 个值为 NaN
            rawdata = np.array(data2)
            train = data[:-10]
            test = data[-10:]
            model = sm.tsa.ARMA(train,order).fit()
```

我们先看看差分序列的 ARMA 拟合值：

```
    In [34]:plt.figure(figsize=(15,5))
            plt.plot(model.fittedvalues,label='fitted value')
```

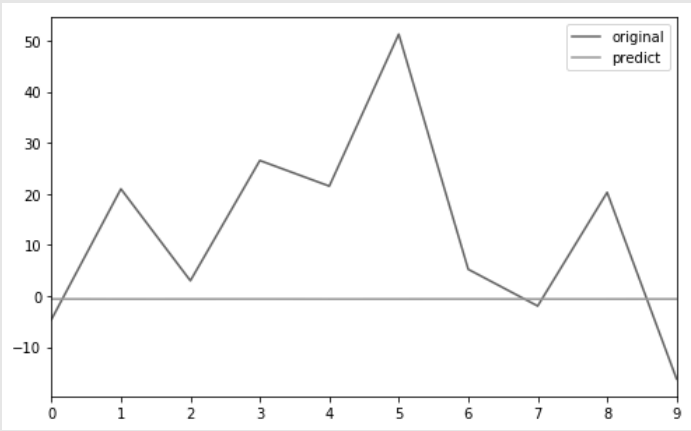
```
        plt.plot(train[1:],label='real value')
        plt.legend(loc=0)
```




```
In [35]:delta = model.fittedvalues - train
        score = 1 - delta.var()/train[1:].var()
        print(score)
Out[35]:0.0397490021589
```

再看看对差分序列的预测情况：

```
In [36]:predicts = model.predict(10,381, dynamic=True)[-10:]
        print(len(predicts))
        comp = pd.DataFrame()
        comp['original'] = test
        comp['predict'] = predicts
        comp.plot(figsize=(8,5))
Out[36]:10
```

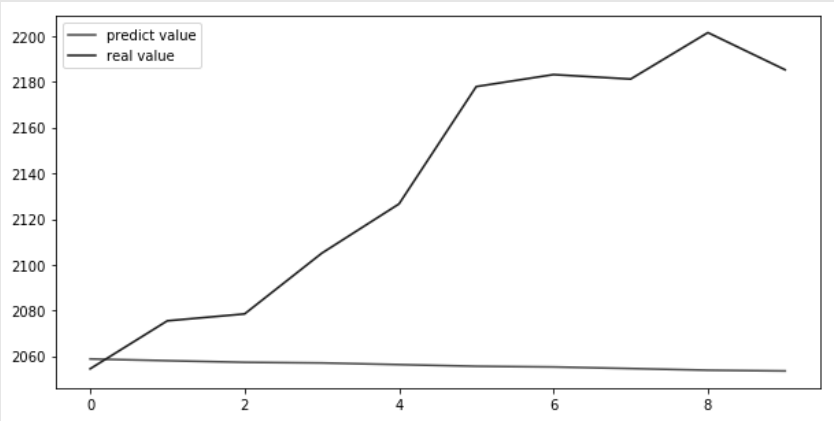


可以看出，差分序列 ARMA 模型的拟合效果和预测结果并不好，预测值非常小，这代表什么？这代表新的值被认为很接近上一时刻的值。

这个影响可能来自模型阶次，看来模型阶次还是得尝试更高阶的。这里就不建模了（计算时间太长），大家有兴趣可以试试高阶的模型。

然后，我们将预测值还原（即在上一时刻指数值的基础上加上对差分差值的预估）：

```
In [37]: rec = [rawdata[-11]]
         pre = model.predict(371, 380, dynamic=True) # 差分序列的预测
         for i in range(10):
         rec.append(rec[i]+pre[i])
         plt.figure(figsize=(10,5))
         plt.plot(rec[-10:],'r',label='predict value')
         plt.plot(rawdata[-10:],'blue',label='real value')
         plt.legend(loc=0)
```



我们发现，由于对差分序列的预测很差，所以在还原到原序列后，预测值几乎在预测的前一个值上小幅波动，模型仍不够好，结果不重要，已经写出方法了。

6.10 组合优化器的使用

6.10.1 优化器的概念

在多因子选股模型当中，常见的组合构建的方式为排序法和筛选法。排序法选出第几

分位组的股票，然后进行简单的等权或者市值加权；筛选法会先对股票进行分层，在每一层里选出对应的第几分位组的股票。然而这两种方法在组合风险的控制上并不够精细，因此需要使用组合优化的算法来给出个股的权重。优矿的优化器正是为了实现这一功能而推出的。优化器的模型大概可以用下面的数学式子进行表示：

$$max: \omega^T \alpha$$

$$st: \min\_weight <= \omega_{i,t} <= \max\_weight$$

$$\sum_k \omega_{i,t} = 1$$

$$\sum_k |\omega_{i,t} - \omega_{i,t-1}| <= \text{turnover\_target}$$

$$x_{style,lower} <= X_{style}\omega <= x_{style,upper}$$

$$x_{indu,lower} <= X_{indu}\omega <= x_{indu,upper}$$

$$IE(\omega - \omega_{benchmark}) = 0$$

$$MCE(\omega - \omega_{benchmark}) = 0$$

$$\omega^T{}_{active} \Sigma \omega_{active} <= 0.5$$

$$0 <= \omega^T \Sigma \omega <= risk$$

$$0 <= (\omega - \omega_b)^T \Sigma (\omega - \omega_b) <= trackingerror$$

上面的公式给出了 7 个约束，如下所述。

◎ 个股权重约束：这是一个很明显的约束，因为 A 股市场不允许做空，所以我们对个股的权重必须要控制在 0 到 1 之间。

◎ 权重之和约束：这个约束是强制的，它会使我们的组合的最终权重之和为 1，即全额投资。

◎ 换手率约束：我们对换手率进行控制，换手率的计算公式为：$\sum_k |\omega_{i,t} - \omega_{i,t-1}|$，注意，这个约束带绝对值，求解起来要用特殊的方法。

◎ 风格因子约束：我们要控制组合在某些风格因子上的暴露。

◎ 行业因子约束：我们要控制组合在行业上的暴露。

◎ 风险约束：二次约束，用以约束组合的风险。

◎ 跟踪误差约束：二次约束，用以控制组合的跟踪误差，与风险约束的区别在于这里使用的是主动权重。

6.10.2 优化器的 API 接口

在优矿专业版中有专门的优化器 API 接口，创建一个优化器对象：

```
    UqerOptimizer(signal, construct_date, risk_model='short', benchmark_str=
'ZZ500', **kwargs)
```

该接口用于创建一个优化器对象，对其参数的说明如下。

◎ signal(Series，dict，list，numpy.ndarray)：信号，可以是预期收益率。如果为 Series，那么索引为 str 格式的 secID，值为该期信号值；如果为 dict，那么键为 str 格式的 secID，值为该期信号值。

◎ construct_date(str)：组合构建的日期，合理的格式为'YYYYMMDD'。

◎ risk_model(str)：风险模型预测周期，可选的参数为'day'、'short'、'long'，默认为'short'。

◎ benchmark_str(str)：指定基准，默认为中证 500 指数，允许的值包括'ZZ500'、'HS300'、'SH50'、'SH180'和'SZZS'。

◎ assets(Series，dict 或者 list)：用于导入上期的持仓权重，主要用于换手率约束当中。如果为 Series，那么索引为股票名称，值为上期权重；如果为 dict，那么键为股票名称，值为上期权重。

UqerOptimizer 的相关属性如下。

◎ assets(DataFrame)：一个 index 为股票代码，列为'init_weights'、'min'、'max'的 DataFrame，用来记录初始权重和权重上下限约束。

◎ optimal(bool)：记录组合优化的状态，默认为 False，优化成功则为 True。

◎ construct_date：记录组合构建的日期。

◎ risk_model_type(str)：记录风险模型存储的类型。

◎ optimal(bool)：记录优化是否成功，True 表示优化成功。

◎ custom_constraints(dict)：记录用户传入的自定义约束，如果不传，则为 None。

UqerOptimizer 的相关方法 add_constraints(name, **kwargs)用于添加约束，由于其参数可以改变，所以这里大概介绍该函数支持的添加约束的方式，如下所述。

（1）添加换手率约束：add_constraint(turnover_target)。

◎ turnover_target(float)：指定组合的换手率上限。

（2）添加个股权重约束：add_constraint(min_weights, max_weights, default_min_weight, default_max_weight)。

◎ min_weights(pd.Series，dict)：如果为 Series，则 index 为股票，值为权重；如果为 dict，则键为 str 格式的 ticker，值为权重，用来指定 index 所包含的股票的权重下限，可不输入。

◎ max_weights(pd.Series，dict)：如果为 Series，则 index 为股票，值为权重；如果为 dict，则键为 str 格式的 ticker，值为权重，用来指定 index 所包含的股票的权重上限，可不输入。

◎ default_min_weight(float)：用来指定组合个股默认的权重下限，优先级低于 min_weight。

◎ default_max_weight(float)：用来指定组合个股默认的权重上限，优先级低于 max_weight。

（3）添加风格因子约束：add_constraint(spec_style)。

◎ style_value(dict)：用来设置风格因子约束，键为风格因子的名称，值为目标的暴露值。风格因子只支持风险模型当中的风格因子：BETA、MOMENTUM、SIZE、EARNYILD、RESVOL、GROWTH、BTOP、LEVERAGE、LIQUIDTY 和 SIZENL。风格约束的值合理的值应在-3 到 3 之间，具体要看 universe。

（4）添加行业中性约束：add_constraint(is_industry_neutralize)。

◎ is_industry_neutralize(bool)：是否是行业中性。

（5）自定义行业中性：add_constraint(spec_indu)。

◎ spec_indu(Series, dict)：允许用户自己传入行业配置方案，键值为行业名称，value

为用户自己配置的行业权重，如果为字典，则 index 为 secID。注意行业为申万一级行业分类，名称为中文，可以通过 DataAPI 中的行业分类获得。

（6）添加风险约束：add_constraint(risk)。

◎ risk(float)：合理的值应大于 0。

（7）添加跟踪误差约束：add_constraint(tracking_error)。

◎ tracking_error(float)：合理的值应大于 0。

（8）添加自定义约束：add_constraint(custom_constraints)，有以下两种方式。

◎ custom_constraints(dict)为字典，包括如下三个键。

- 'data'：表示传入的约束矩阵，不允许有缺失值，而且应该确保 signal 中的每个股票都有值，格式为 DataFrame 或 Series，其中，index 为股票的 secID。
- 'upper'：可接收的数据类型为 list，np.array，记录约束的上限，长度与 data 的列长一致，注意，即使自定义约束只有一个，也要将其在列表当中。
- 'lower'：可接收的数据类型为 list，np.array，记录约束的上限，其他同'upper'。

◎ 另一种添加约束的方式如下：

```
    add_constraint(is_industry_neutralize, turnover_target, max_weights,
min_weights, default_max_weight, default_min_weight, spec_style, spec_industry,
tracking_error, risk, custom_constraints)。
```

注意：这两种方式都必须输入参数名。

solve()用于求解优化问题，会在 assets 属性当中加'optimal_weights'列，如果求解成功，则该列为优化后的权重，否则仍然返回上期权重。

6.10.3 优化器实例

这里通过一个具体的例子来介绍优化器的具体用法：

```
    import quartz_extensions.Optimizer as opt

    # 获取信号
    date = '20160928'
    data = DataAPI.MktStockFactorsOneDayProGet(tradeDate=date, secID=set_
```

```
universe('HS300', date), field=u"secID, tradeDate, OperatingProfitPS",
pandas="1")
    signal = (data.pivot(index='tradeDate', columns='secID', values=
'OperatingProfitPS').dropna(axis=1)).iloc[0, :]
    signal = standardize(neutralize(winsorize(signal), date)).dropna()

    # 创建优化器对象
    pspec = opt.UqerOptimizer(signal, date, benchmark_str='HS300')
    # 添加约束
    # 个股上下限约束
    pspec.add_constraint(default_min_weight=0., default_max_weight=0.05)
    # 行业中性
    pspec.add_constraint(is_industry_neutralize=True)
    # 风格约束
    pspec.add_constraint(spec_style={'SIZE':-0.05})
    pspec.solve()
    pspec.optimal
```

结果如下：

```
    True
    pspec.optimal
    True
```

1. 优化后的结果

优化后的结果存在 assets 这个属性中，我们可以大体看看一部分股票的结果：

```
    weights = pspec.assets[pspec.assets.optimal_weights > 0.00001]
    weights.head()
```

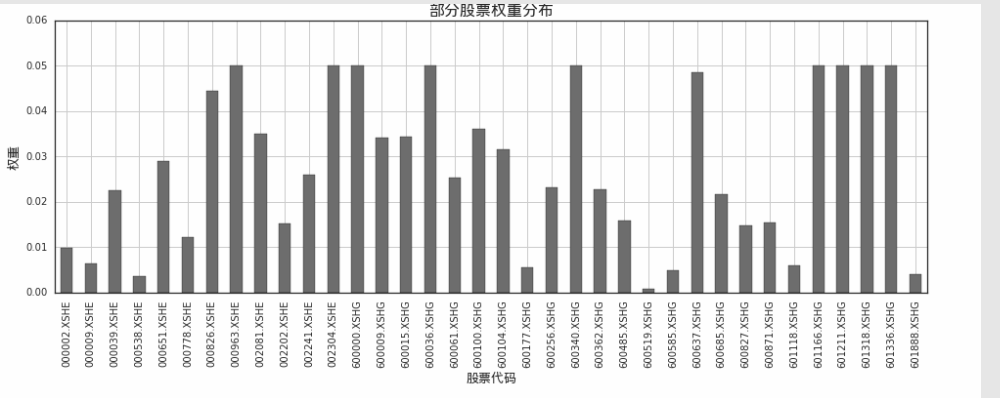
	init_weights	max	min	optimal_weights
secID				
000002.XSHE	0.003333	0.05	0.0	0.00982
000009.XSHE	0.003333	0.05	0.0	0.00646
000039.XSHE	0.003333	0.05	0.0	0.02260
000538.XSHE	0.003333	0.05	0.0	0.00354
000651.XSHE	0.003333	0.05	0.0	0.02897

这里没有传入上一期权重，所以初始化了一个等权组合，如果要添加换手率约束，那么还需要在创建优化器对象时传入上一期权重。

2. 权重分布图

这里展示了权重大于 0.00001 的股票的权重的分布：

```
import matplotlib.pyplot as plt
import seaborn as sns
from CAL.PyCAL import *
sns.set_style('white')
fig = plt.figure(figsize=(16,5))
ax = fig.add_subplot(111)
ax = weights.optimal_weights.plot(kind='bar', ax=ax)
ax.set_title(u'部分股票权重分布', fontproperties=font, fontsize=16)
ax.set_xlabel(u'股票代码', fontproperties=font, fontsize=13)
ax.set_ylabel(u'权重', fontproperties=font, fontsize=13)
ax.grid()
```

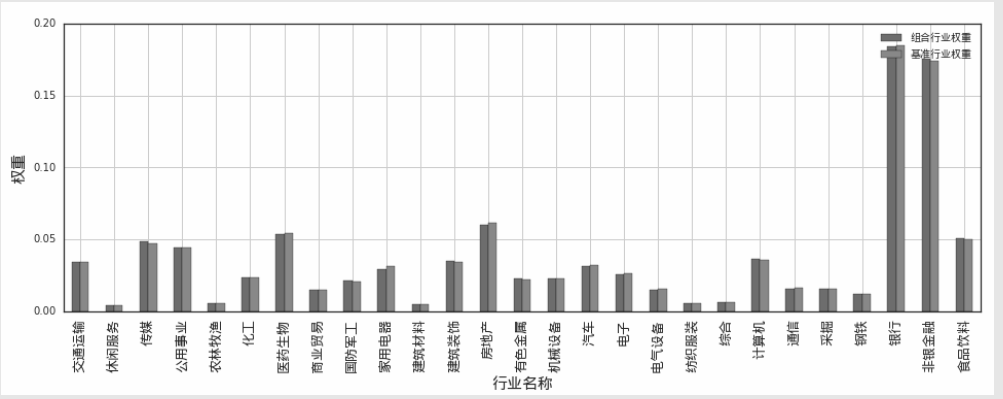


3. 行业分布比较

因为我们在这里做的是行业中性，所以优化后组合的行业权重应与基准的行业权重一致：

```
    # 获得基准成分股行业分类
    equ_indu = DataAPI.EquIndustryGet(industryVersionCD=u"010303", secID=set_
universe('HS300', '20160930'), intoDate='20160930', field=u"secID,
industryName1", pandas="1")
```

```
    benchmark_equ_weight = DataAPI.IdxCloseWeightGet(ticker='000300',
beginDate='20160930', endDate='20160930', field=u"consID,weight",pandas="1")
    benchmark_equ_weight.rename(columns={'consID': 'secID', 'weight':u'基准行
业权重'}, inplace=True)

    # 获得基准行业权重
    benchmark_indu_weight =equ_indu.merge(benchmark_equ_weight, on='secID',
how='inner').groupby(by='industryName1').sum()/100
    equ_indu_ = DataAPI.EquIndustryGet(industryVersionCD=u"010303", secID=
list(pspec.assets.index), intoDate='20160930', field=u"secID,industryName1",
pandas="1")
    pspec_indu_weight=equ_indu_.merge(pspec.assets[['optimal_weights']],
left_on='secID', right_index=True, how='inner').groupby(by='industryName1').
sum()
    pspec_indu_weight.rename(columns={'optimal_weights':u'组合行业权重'},
inplace=True)
    data = pspec_indu_weight.merge(benchmark_indu_weight, left_index=True,
right_index=True, how='inner')
    fig = plt.figure(figsize=(16,5))
    ax = fig.add_subplot(111)
    ax = data.plot(kind='bar', ax=ax)
    labels = [label.decode("utf-8") for label in data.index.values]
    ax.set_xticklabels(labels, fontproperties=font, fontsize=13)
    ax.legend(prop=font)
    ax.set_xlabel(u'行业名称', fontproperties=font, fontsize=15)
    ax.set_ylabel(u'权重', fontproperties=font, fontsize=15)
    ax.grid()
```



可以看到，优化器将组合的行业权重和基准的行业权重的分布变得基本一样。

6.11 期权策略：Greeks 和隐含波动率微笑计算

在本文中，我们将通过通联数据提供的数据进行上证 50ETF 期权的隐含波动率微笑计算。

6.11.1 数据准备

将上海银行间同业拆借利率 SHIBOR 作为无风险利率参考：

```
    from CAL.PyCAL import *
    import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt
    from matplotlib import rc
    rc('mathtext', default='regular')
    import seaborn as sns
    sns.set_style('white')
    import math
    from scipy import interpolate
    from scipy.stats import mstats
    from pandas import Series, DataFrame, concat
    import time
    from matplotlib import dates
    ## 银行间质押式回购利率
    def getHistDayInterestRateInterbankRepo(date):
        cal = Calendar('China.SSE')
        period = Period('-10B')
        begin = cal.advanceDate(date, period)
        begin_str = begin.toISO().replace('-', '')
        date_str = date.toISO().replace('-', '')
        # 以下 indicID 对应的银行间质押式回购利率周期分别为：
        # 1D、7D、14D、21D、1M、3M、4M、6M、9M、1Y
        indicID = [u"1090000566", u"1090000567", u"1090000568", u"1090000569",
u"1090000570", u"1090000571", u"1090000572", u"1090000573", u"1090000574",
u"1090000575"]

        period = np.asarray([1.0, 7.0, 14.0, 21.0, 30.0, 90.0, 120.0, 180.0, 270.0,
```

```
360.0]) / 360.0
        period_matrix = pd.DataFrame(index=indicID, data=period, columns=
['period'])
        field = [u"indicID",u"publishDate",u"periodDate",u"dataValue"]
        interbank_repo = DataAPI.EcoDataProGet(indicID, begin_str, date_str)
        interbank_repo = interbank_repo.groupby('indicID').first()
        interbank_repo.index = [str(i) for i in interbank_repo.index]
        interbank_repo = concat([interbank_repo, period_matrix], axis=1,
join='inner').sort_index()
        return interbank_repo

    ## 银行间同业拆借利率
    def getHistDaySHIBOR(date):
        date_str = date.toISO().replace('-', '')
        # 以下 indicID 对应的 SHIBOR 周期分别为:
        # 1D、7D、14D、1M、3M、6M、9M、1Y
        indicID = [u'1090000537',u'1090000538',u'1090000539',u'1090000540',
                   u'1090000541',u'1090000542',u'1090000543',u'1090000544']
        period = np.asarray([1.0, 7.0, 14.0, 30.0, 90.0, 180.0, 270.0, 360.0])
/ 360.0
        period_matrix = pd.DataFrame(index=indicID, data=period, columns=
['period'])
        field = [u"indicID",u"publishDate",u"periodDate",u"dataValue"]
        interest_shibor = DataAPI.EcoDataProGet(indicID, beginDate=date_str,
endDate=date_str)[field]
        interest_shibor = interest_shibor.set_index('indicID')
        interest_shibor.index = [str(i) for i in interest_shibor.index]
        interest_shibor = concat([interest_shibor, period_matrix], axis=1,
join='inner').sort_index()
        return interest_shibor

    ## 插值得到给定的周期的无风险利率
    def periodsSplineRiskFreeInterestRate(date, periods):
        # 此处使用 SHIBOR 来插值
        init_shibor = getHistDaySHIBOR(date)

        shibor = {}
        min_period = min(init_shibor.period.values)
        min_period = 25.0/360.0
        max_period = max(init_shibor.period.values)
        for p in periods.keys():
```

```
            tmp = periods[p]
            if periods[p] > max_period:
                tmp = max_period * 0.99999
            elif periods[p] < min_period:
                tmp = min_period * 1.00001
            sh = interpolate.spline(init_shibor.period.values,
init_shibor.dataValue.values, [tmp], order=3)
            shibor[p] = sh[0]/100.0
        return shibor
```

6.11.2 Greeks 和隐含波动率计算

这里计算如下 Greeks。

- ◎ delta：期权价格关于标的价格的一阶导数。
- ◎ gamma：期权价格关于标的价格的二阶导数。
- ◎ rho：期权价格关于无风险利率的一阶导数。
- ◎ theta：期权价格关于到期时间的一阶导数。
- ◎ vega：期权价格关于波动率的一阶导数。

有以下注意事项。

- ◎ 在计算隐含波动率时采用 Black-Scholes-Merton 模型，此模型在平台 Python 包的 CAL 中已有实现。
- ◎ 对无风险利率使用 SHIBOR。
- ◎ 在期权的时间价值为负时（此种情况在 50ETF 期权里时有发生），无法通过 BSM 模型计算隐含波动率，所以此时将期权隐含波动率设为 0.0，实际上，此时的隐含波动率和各风险指标并无实际参考价值。

代码如下：

```
## 使用 DataAPI.OptGet、DataAPI.MktOptdGet 拿到计算所需的数据
def getOptDayData(opt_var_sec, date):
    date_str = date.toISO().replace('-', '')

    #使用 DataAPI.OptGet 拿到已退市和上市的所有期权的基本信息
```

```
        info_fields = [u'optID', u'varSecID', u'varShortName', u'varTicker',
u'varExchangeCD', u'varType', u'contractType', u'strikePrice', u'contMultNum',
u'contractStatus', u'listDate', u'expYear', u'expMonth', u'expDate',
u'lastTradeDate', u'exerDate', u'deliDate', u'delistDate']
        opt_info = DataAPI.OptGet(optID='', contractStatus=[u"DE",u"L"],
field=info_fields, pandas="1")

        #使用 DataAPI.MktOptdGet 拿到历史上某一天的期权成交信息
        mkt_fields = [u'ticker', u'optID', u'secShortName', u'exchangeCD',
u'tradeDate', u'preSettlePrice', u'preClosePrice', u'openPrice',
u'highestPrice', u'lowestPrice', u'closePrice', u'settlPrice', u'turnoverVol',
u'turnoverValue', u'openInt']
        opt_mkt = DataAPI.MktOptdGet(tradeDate=date_str, field=mkt_fields,
pandas = "1")

        opt_info = opt_info.set_index(u"optID")
        opt_mkt = opt_mkt.set_index(u"optID")
        opt = concat([opt_info, opt_mkt], axis=1, join='inner').sort_index()
        return opt

    ## 分析历史某一日的期权收盘价信息，得到隐含波动率微笑和期权风险指标
    def getOptDayAnalysis(opt_var_sec, date):
        opt = getOptDayData(opt_var_sec, date)

        #使用 DataAPI.MktFunddGet 拿到期权标的的日行情
        date_str = date.toISO().replace('-', '')
        opt_var_mkt = DataAPI.MktFunddGet(secID=opt_var_sec,tradeDate=date_str,
beginDate=u"",endDate=u"",field=u"",pandas="1")
        #opt_var_mkt = DataAPI.MktFunddAdjGet(secID=opt_var_sec,beginDate=
date_str,endDate=date_str,field=u"",pandas="1")

        # 计算 shibor
        exp_dates_str = opt.expDate.unique()
        periods = {}
        for date_str in exp_dates_str:
            exp_date = Date.parseISO(date_str)
            periods[exp_date] = (exp_date - date)/360.0
        shibor = periodsSplineRiskFreeInterestRate(date, periods)
```

```
        settle = opt.settlPrice.values          # 期权 settle price
        close = opt.closePrice.values           # 期权 close price
        strike = opt.strikePrice.values         # 期权 strike price
        option_type = opt.contractType.values   # 期权类型
        exp_date_str = opt.expDate.values       # 期权行权日期
        eval_date_str = opt.tradeDate.values    # 期权交易日期

        mat_dates = []
        eval_dates = []
        spot = []
        for epd, evd in zip(exp_date_str, eval_date_str):
            mat_dates.append(Date.parseISO(epd))
            eval_dates.append(Date.parseISO(evd))
            spot.append(opt_var_mkt.closePrice[0])
        time_to_maturity = [float(mat - eva + 1.0)/365.0 for (mat, eva) in
zip(mat_dates, eval_dates)]

        risk_free = []  # 无风险利率
        for s, mat, time in zip(spot, mat_dates, time_to_maturity):
            #rf = math.log(forward_price[mat] / s) / time
            rf = shibor[mat]
            risk_free.append(rf)

        opt_types = []   # 期权类型
        for t in option_type:
            if t == 'CO':
                opt_types.append(1)
            else:
                opt_types.append(-1)

        # 使用通联 CAL 包中的 BSMImpliedVolatity 计算隐含波动率
        calculated_vol = BSMImpliedVolatity(opt_types, strike, spot, risk_free,
0.0, time_to_maturity, settle)
        calculated_vol = calculated_vol.fillna(0.0)

        # 使用通联 CAL 包中的 BSMPrice 计算期权风险指标
        greeks = BSMPrice(opt_types, strike, spot, risk_free, 0.0,
calculated_vol.vol.values, time_to_maturity)
        greeks.vega = greeks.vega #/ 100.0
```

```
        greeks.rho = greeks.rho #/ 100.0
        greeks.theta = greeks.theta #* 365.0 / 252.0 #/ 365.0

        opt['strike'] = strike
        opt['optType'] = option_type
        opt['expDate'] = exp_date_str
        opt['spotPrice'] = spot
        opt['riskFree'] = risk_free
        opt['timeToMaturity'] = np.around(time_to_maturity, decimals=4)
        opt['settle'] = np.around(greeks.price.values.astype(np.double),
decimals=4)
        opt['iv'] = np.around(calculated_vol.vol.values.astype(np.double),
decimals=4)
        opt['delta'] = np.around(greeks.delta.values.astype(np.double),
decimals=4)
        opt['vega'] = np.around(greeks.vega.values.astype(np.double),
decimals=4)
        opt['gamma'] = np.around(greeks.gamma.values.astype(np.double),
decimals=4)
        opt['theta'] = np.around(greeks.theta.values.astype(np.double),
decimals=4)
        opt['rho'] = np.around(greeks.rho.values.astype(np.double),
decimals=4)

        fields = [u'ticker', u'contractType', u'strikePrice', u'expDate',
u'tradeDate', u'closePrice', u'settlPrice', 'spotPrice', u'iv', u'delta',
u'vega', u'gamma', u'theta',  u'rho']
        opt = opt[fields].reset_index().set_index('ticker').sort_index()
        #opt['iv'] = opt.iv.replace(to_replace=0.0, value=np.nan)
        return opt
```

尝试用 getOptDayAnalysis 计算 2015 年 9 月 24 日这一天的风险指标：

```
    # Uqer 计算期权的风险数据
    opt_var_sec = u"510050.XSHG"    # 期权标的
    date = Date(2015, 9, 24)

    option_risk = getOptDayAnalysis(opt_var_sec, date)
    option_risk.head(2)
```

	optID	contractType	strikePrice	expDate	tradeDate	closePrice	settlPrice	spotPrice	iv	delta	vega	gamma	theta	rho
ticker														
510050C1510M01850	10000405	CO	1.85	2015-10-28	2015-09-24	0.3268	0.3555	2.187	0.4317	0.9101	0.1099	0.5550	-0.2992	0.1568
510050C1510M01900	10000406	CO	1.90	2015-10-28	2015-09-24	0.2791	0.3102	2.187	0.4161	0.8810	0.1347	0.7058	-0.3435	0.1550

getOptDayAnalysis 函数计算结果如下：

```
    # 本文计算结果 option_risk

    near_exp = np.sort(option_risk.expDate.unique())[0]    # 近月期权行权日

    opt_call_uqer = option_risk[option_risk.expDate==near_exp][option_risk.
contractType=='CO'].set_index('strikePrice')
    opt_put_uqer = option_risk[option_risk.expDate==near_exp][option_risk.
contractType=='PO'].set_index('strikePrice')

    ## ------------------------------------------------
    ## 风险指标
    fig = plt.figure(figsize=(10,12))
    fig.set_tight_layout(True)

    # ------ Delta ------
    ax = fig.add_subplot(321)
    ax.plot(opt_call_uqer.index, opt_call_uqer['delta'], '-o')
    ax.plot(opt_put_uqer.index, opt_put_uqer['delta'], '-o')
    ax.legend(['call-uqer', 'put-uqer'])
    ax.grid()
    ax.set_xlabel(u"strikePrice")
    ax.set_ylabel(r"Delta")
    plt.title('Delta Comparison')

    # ------ Theta ------
    ax = fig.add_subplot(322)
    ax.plot(opt_call_uqer.index, opt_call_uqer['theta'], '-o')
    ax.plot(opt_put_uqer.index, opt_put_uqer['theta'], '-o')
    ax.legend(['call-uqer', 'put-uqer'])
    ax.grid()
    ax.set_xlabel(u"strikePrice")
    ax.set_ylabel(r"Theta")
    plt.title('Theta Comparison')
```

```
# ------ Gamma ------
ax = fig.add_subplot(323)
ax.plot(opt_call_uqer.index, opt_call_uqer['gamma'], '-o')
ax.plot(opt_put_uqer.index, opt_put_uqer['gamma'], '-o')
ax.legend(['call-uqer', 'put-uqer'], loc=0)
ax.grid()
ax.set_xlabel(u"strikePrice")
ax.set_ylabel(r"Gamma")
plt.title('Gamma Comparison')

# # ------ Vega ------
ax = fig.add_subplot(324)
ax.plot(opt_call_uqer.index, opt_call_uqer['vega'], '-o')
ax.plot(opt_put_uqer.index, opt_put_uqer['vega'], '-o')
ax.legend(['call-uqer',  'put-uqer'], loc=4)
ax.grid()
ax.set_xlabel(u"strikePrice")
ax.set_ylabel(r"Vega")
plt.title('Vega Comparison')

# ------ Rho ------
ax = fig.add_subplot(325)
ax.plot(opt_call_uqer.index, opt_call_uqer['rho'], '-o')
ax.plot(opt_put_uqer.index, opt_put_uqer['rho'], '-o')
ax.legend(['call-uqer', 'put-uqer'], loc=3)
ax.grid()
ax.set_xlabel(u"strikePrice")
ax.set_ylabel(r"Rho")
plt.title('Rho Comparison')
```

对于近月期权，我们分别对比了 5 个 Greeks 风险指标：Delta、Theta、Gamma、Vega 和 Rho（见图 6-31），在每个小图中将 Call 和 Put 分开比较，横轴为行权价。可以看出，这里的计算结果和上交所的参考数值比较符合。在接下来的 50ETF 期权分析中，我们将使用这里的计算方法来计算期权隐含波动率和 Greeks 风险指标。

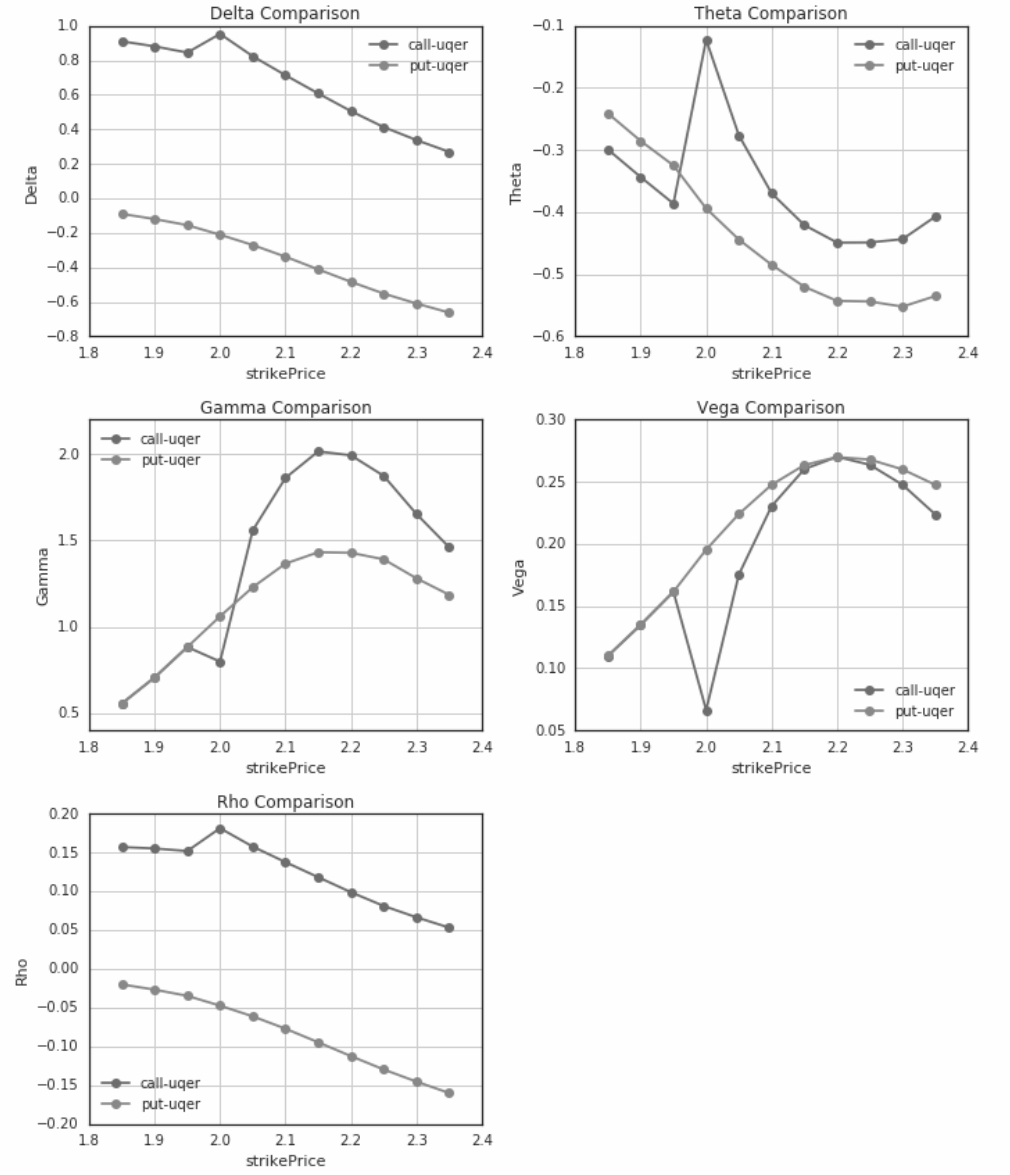


图 6-31

下面将以上数据进行整理，格式会更简洁一些：

```
# 每日期权分析数据整理
```

```
    def getOptDayGreeksIV(date):
        # Uqer 计算期权的风险数据
        opt_var_sec = u"510050.XSHG"    # 期权标的
        opt = getOptDayAnalysis(opt_var_sec, date)

        # 整理数据部分
        opt.index = [index[-10:] for index in opt.index]
        opt =
opt[['contractType','strikePrice','expDate','closePrice','iv','delta','theta
','gamma','vega','rho']]
        opt_call = opt[opt.contractType=='CO']
        opt_put = opt[opt.contractType=='PO']
        opt_call.columns = pd.MultiIndex.from_tuples([('Call', c) for c in
opt_call.columns])
        opt_call[('Call-Put', 'strikePrice')] = opt_call[('Call',
'strikePrice')]
        opt_put.columns = pd.MultiIndex.from_tuples([('Put', c) for c in
opt_put.columns])
        opt = concat([opt_call, opt_put], axis=1, join='inner').sort_index()
        opt = opt.set_index(('Call','expDate')).sort_index()
        opt = opt.drop([('Call','contractType'), ('Call','strikePrice')],
axis=1)
        opt = opt.drop([('Put','expDate'), ('Put','contractType'),
('Put','strikePrice')], axis=1)
        opt.index.name = 'expDate'
        ## 以上得到了完整的历史某日数据，格式简洁明了
        return opt
    # 输入：
    date = Date(2015, 9, 24)
    option_risk = getOptDayGreeksIV(date)
    option_risk.head(10)
```

6.11.3　隐含波动率微笑

下面展示某一天的隐含波动率微笑：

```
    def plotSmileVolatility(date):
        # Uqer 计算期权的风险数据
        opt = getOptDayGreeksIV(date)
```

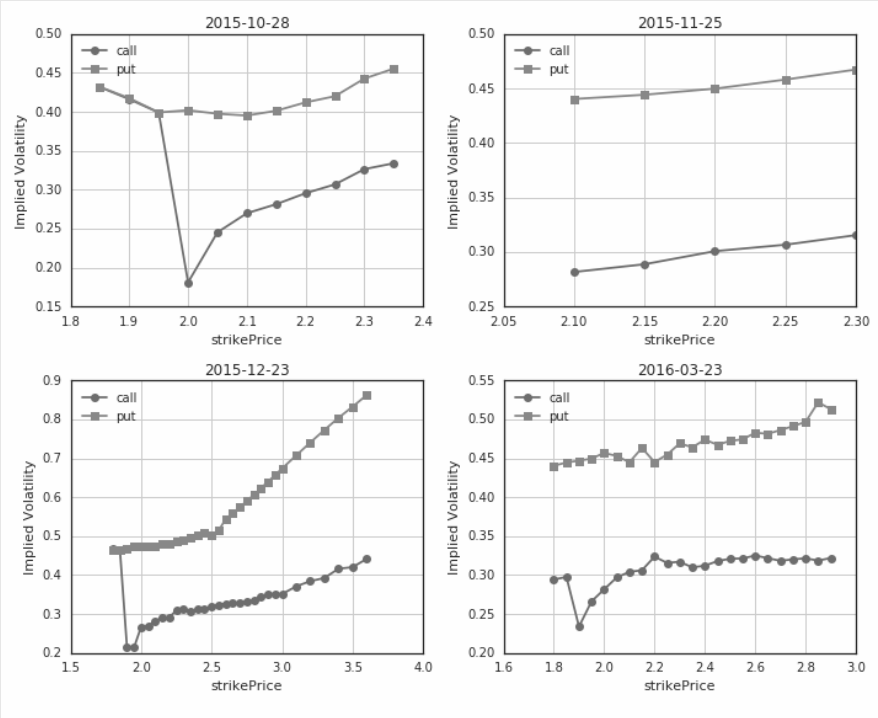
	Call							Call-Put	Put						
	closePrice	iv	delta	theta	gamma	vega	rho	strikePrice	closePrice	iv	delta	theta	gamma	vega	rho
expDate															
2015-10-28	0.3268	0.4317	0.9101	-0.2992	0.5550	0.1099	0.1568	1.85	0.0129	0.4319	-0.0900	-0.2410	0.5551	0.1100	-0.0201
2015-10-28	0.2791	0.4161	0.8810	-0.3435	0.7058	0.1347	0.1550	1.90	0.0176	0.4174	-0.1197	-0.2854	0.7063	0.1352	-0.0268
2015-10-28	0.2360	0.3990	0.8449	-0.3862	0.8823	0.1615	0.1517	1.95	0.0232	0.3992	-0.1552	-0.3247	0.8822	0.1615	-0.0348
2015-10-28	0.1955	0.1811	0.9532	-0.1225	0.7980	0.0663	0.1811	2.00	0.0345	0.4020	-0.2105	-0.3940	1.0601	0.1954	-0.0474
2015-10-28	0.1599	0.2453	0.8237	-0.2764	1.5588	0.1754	0.1574	2.05	0.0474	0.3975	-0.2703	-0.4441	1.2290	0.2241	-0.0612
2015-10-28	0.1275	0.2698	0.7137	-0.3696	1.8625	0.2304	0.1374	2.10	0.0643	0.3952	-0.3381	-0.4847	1.3660	0.2476	-0.0771
2015-10-28	0.0990	0.2814	0.6081	-0.4208	2.0162	0.2602	0.1180	2.15	0.0869	0.4013	-0.4114	-0.5200	1.4317	0.2635	-0.0946
2015-10-28	0.0768	0.2955	0.5057	-0.4489	1.9934	0.2701	0.0987	2.20	0.1146	0.4121	-0.4836	-0.5428	1.4284	0.2699	-0.1124
2015-10-28	0.0584	0.3068	0.4132	-0.4487	1.8746	0.2637	0.0810	2.25	0.1450	0.4200	-0.5517	-0.5438	1.3908	0.2679	-0.1296
2015-10-28	0.0470	0.3264	0.3381	-0.4434	1.6538	0.2476	0.0664	2.30	0.1826	0.4426	-0.6091	-0.5520	1.2809	0.2600	-0.1452

下面展示波动率微笑：

```
        exp_dates = np.sort(opt.index.unique())
        ## ----------------------------------------------
        fig = plt.figure(figsize=(10,8))
        fig.set_tight_layout(True)

        for i in range(exp_dates.shape[0]):
            date = exp_dates[i]
            ax = fig.add_subplot(2,2,i+1)
            opt_date = opt[opt.index==date].set_index(('Call-Put',
'strikePrice'))
            opt_date.index.name = 'strikePrice'

            ax.plot(opt_date.index, opt_date[('Call', 'iv')], '-o')
            ax.plot(opt_date.index, opt_date[('Put', 'iv')], '-s')
            ax.legend(['call', 'put'], loc=0)
            ax.grid()
            ax.set_xlabel(u"strikePrice")
            ax.set_ylabel(r"Implied Volatility")
            plt.title(exp_dates[i])
```

输入：

```
plotSmileVolatility(Date(2015,9,24))
```

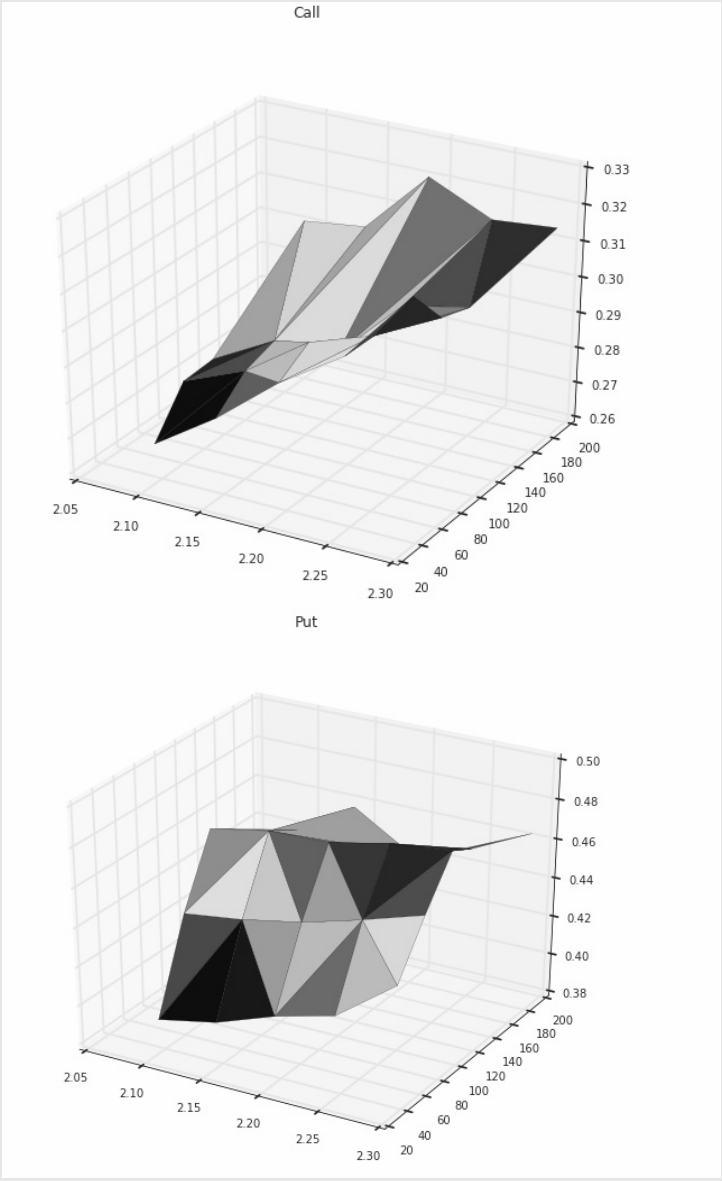


计算行权价和行权日期两个方向上的隐含波动率微笑：

```
opt = plotSmileVolatilitySurface(Date(2015,9,24))
opt  # Call 和 Put 分开显示，行 index 为行权价，列 index 为剩余到期天数
```

输出：

```
{'Call':    34      62      90      181
 2.10       0.2698  0.2817  0.2823  0.3042
 2.15       0.2814  0.2888  0.2916  0.3063
 2.20       0.2955  0.3008  0.2922  0.3237
 2.25       0.3068  0.3067  0.3093  0.3157
 2.30       0.3264  0.3155  0.3128  0.3172,
 'Put':     34      62      90      181
 2.10       0.3952  0.4403  0.4740  0.4449
 2.15       0.4013  0.4442  0.4794  0.4632
 2.20       0.4121  0.4498  0.4802  0.4451
 2.25       0.4200  0.4581  0.4863  0.4547
 2.30       0.4426  0.4673  0.4893  0.4691}
```



在如上所示的波动率曲面结构图中，上图为 Call，下图为 Put，此处没有进行任何插值处理，所以略显粗糙。另外，Put 的隐含波动率明显大于 Call，从期限结构来说，波动率呈现远高近低的特征。

关于期权策略，在优矿社区中有更多的策略可供大家参考、学习。

第 7 章

量化投资十问十答

作为投资者，我们常听到的一句话是“不要把鸡蛋放入同一个篮子中”，可见分散投资可以降低风险，但如何选择不同的篮子，在每个篮子中放多少鸡蛋，这便是见仁见智的事情了，量化投资就是解决这样问题的一种工具。量化投资涉及很多方面，有各种场景下的应用，定义也比较困难，我们姑且将其定义为借助于数学知识、统计学知识开发出策略模型，根据策略模型给出的信号严格执行信号的投资过程。其本质就是从数据的角度提炼出市场不够有效的成分，用模型加以概括。

为了方便大家在短时间内更好地了解量化投资，并且能在实际投资中进行应用，笔者准备了十道自问自答题目，希望对大家做投资有一点点帮助。

一、量化投资可以在哪些方面帮助我投资?

答：总的来说，量化投资在投资的前、中、后都有强大的应用。在投资前可以根据量化投资策略选股，包括基于股票基本面或者技术面因素等；在投资中可以通过量化手段关注当前持仓股票是否有重大事件发生或者有影响股价波动的其他因素，即对持仓风险的监控与识别；在投资后可以做一些总结与归因，明确自己投资股票的风格与行业偏重。

二、量化投资是万能的吗？

答：千万不要把量化投资当作印钞机，它并不是万能的，只能说我们可以通过它获得概率上的优势。并且，量化投资并不是“一招鲜吃遍天”，需要根据市场的不断变化对策略或模型进行完善和优化，将多套策略结合才能使系统具有可持续地稳定赢利的能力，其中就包括从历史的数据中通过量化手段找到合理的规律，在投资决策中进行应用。量化投资更像西医，对于所有病症都有大量的临床案例和测量参数，而且材料齐全、容易复制。另外，量化投资肯定要与主观相结合，否则，光靠量化投资、机器和数学模型，很难在所有时期都有效。

三、主动选股和量化投资的区别是什么？

答：主动选股与量化投资并非互相矛盾的，相反，二者是相辅相成、互为补充的。从某种意义上来说，量化投资赚的是市场空隙的钱，市场的大部分机会还是来自基本面上的价值发现。但当在市场上有成千上万只股票的时候，强大的定量投资的信息处理能力就能体现出它的优势，从而捕捉更多的投资机会。主动投资和量化投资类似于中医和西医，中医主要通过望、闻、问、切等医疗手段，依靠经验进行诊断；西医则不同，西医主要借助于医学仪器，对于各项检查结果都有详细的数据评价标准，最后判断症结所在，进而对症下药。

四、我不会编程，高大上的量化投资会不会让我遥不可及？

答：量化投资其实是一种思维，虽然涉及大量的编程工作，目前在国内还只是博士、硕士才能完成的工作，但其本质思想还是总结与学习历史经验。对于普通投资者来说，目前也有非常容易上手的第三方量化平台可供大家使用。另外，量化投资并非只适用于选股，对市场的监控、技术指标挖掘等都可以用量化投资实现。可以说，量化投资可以在多方面对投资进行辅助。

五、量化投资有哪些优点？

答：对于量化投资的优点，可以从以下 5 个方面进行说明。

（1）系统性：量化投资主要包括多层次的量化模型，可以从多个角度对行情、财务、预期、舆情等数据进行观察。

（2）及时性：及时、快速地跟踪市场的变化，不断发现能够提供超额收益的模型，寻找交易机会。

（3）准确性：准确、客观地评价交易机会，克服主观情绪，寻找估值洼地。

（4）纪律性：不随投资者情绪的变化而变化，克服人性的弱点。

（5）分散化：依靠概率或者根据策略筛选出的投资组合取胜，而不是依靠一个或者几个品种取胜。

六、你觉得做量化投资最大的困难是什么？

答：要理解市场，无论是进行主观投资还是进行量化投资，我们都在市场上进行博弈，只是技术手段不一样。有时，量化模型是会失效的，如果一味地追求程序、追求模型，就会导致收益大幅回撤，这也是有纯理科或者 IT 背景的人做量化投资时容易犯的错误。

七、能举例说明量化投资是怎么做的吗？

答：量化投资是将影响股价的各种因素共同考虑并完成建模的过程。例如，量化投资中的多因子模型考虑了宏观因素、行业因素、公司因素等，其中包括因子体系的构建，比如市场因子（风险因子、流动性、换手率等）、风格因子（规模、价值、成长等）、财务因子（赢利能力、偿债能力、现金流量等）、动量因子等，从多个角度系统地评价股票投资价值。

八、量化投资的应用有哪些？

答：量化投资技术几乎覆盖了投资的全过程和各种领域，包括量化选股、量化择时、股指期货套利、商品期货套利、统计套利、算法交易、资产配置、风险控制等。例如，量化选股是利用数量化方法来选择股票组合；量化择时是通过对宏微观指标的量化分析来判断市场走势；算法交易是通过计算机程序发出交易指令如 TWAP/VWAP 等。

九、量化投资是不是追求高胜率的方法？

答：高胜率是所有策略追求的终极目标，但单纯的高胜率不代表能稳定赢利，例如一个投资策略进行了 10 次，9 次都能赚钱，但亏损那次亏得特别多（低盈亏比），甚至将所有赢利回吐，这样的策略意义就不大。

从理论上讲，交易要想赢利，就必须处理好交易成功率和盈亏比的关系。盈亏比就是交易多次以后，赚钱的交易平均赢利点数除以亏损的交易平均亏损的点数。例如，每笔交易赢利 10%，每次亏损幅度为 5%，盈亏比就是 2，在不考虑手续费的基础上，做对一次

交易，可以亏两次。所以，在每次交易平均仓位都一致的情况下，盈亏比和成功率就直接决定了交易的业绩。

十、围棋出现了 AlphaGo，投资会不会出现 AlphaStock？

答：不排除这个可能，但是从目前来看，价格也是由多方面因素共同决定的。市场的复杂度远高于围棋，有些风险很难量化，也就是说虽然人工智能在某些方面远胜于人类，但是目前还未出现可以直接实盘的人工智能，其任重道远。另外，就算最后出现了可以用于实盘的 AlphaStock，市场也不会是人类与 AlphaStock 之间的博弈，而是在不同人的 AlphaStock 之间的博弈。