深入浅出量化实验室

发布 1.6.9

通联数据金融工程团队

2016 年 01 月 26 日

目录

Part I

写在前面

CHAPTER 1

量化实验室（Mercury）

呈现在您眼前的，是依托通联数据的金融大数据资源，由通联数据金融工程团队开发的新时代金融平台。量化实验室（Mercury）是通联数据切合中国金融市场的需要，推出的具有独创性和前沿性的平台。它的目标:

- 打破现有金融工具互相割裂，互不兼容的格局，开创一个统一，包容的金融数据，分析，交易中心。
- 数据为根，分析为本！缺乏分析的数据是一潭死水，缺乏数据的分析是无本之木。量化实验室将数据与分析紧密结合，融为一体，做到你中有我，我中有你。

为贯彻以上理念，Mercury 提供了两大基石：

QUARTZ

Quartz 是 Mercury 的回测交易框架。在这一框架下，用户可以专注于描述交易算法逻辑，而不用关心底层的实现细节。Quartz 为用户处理包括账户簿记，策略表现评估的功能。现阶段 Quartz 支持日间回测，并且将很快推出日内交易回测的机制。现阶段，Quartz 已经支持回测功能，未来会推出交易功能。方便客户将成功的策略无缝的投入实盘环境。

CAL

CAL 是 Mercury 中的金融计算分析库。涵盖金融市场计算的方方面面，包括定价，风险指标分析，数值计算以及等等。涉及的产品包括：股票、债券、利率、期权、衍生品以及 ABS 等等。CAL 的设计是高度模块化，方便专业金融人士在此基础之上设计自己的模型或者整合进自己的已有系统之中。

CHAPTER 2

如何使用 Mercury

我们的平台是完全在线模式。用户只需要一个拥有权限的账号就可以使用 Mercury 平台，我们为每个用户提供了自己的独立运行环境与开发环境（这也是为什么我们称之为量化实验室的原因）。Mercury 完全使用 Python 开发，所以如果你有 Python 基础的话，会看到这本书中看到例子是那么的眼熟，这么的自在（Python 本身就是让人用着很舒服的语言，不是吗？）。当然如果你没有 Python 基础，但是使用过类似于 Matlab 这样的语言？相信我，转到我们的平台上面也不是费力的，它们的语法结构至少有 50% 是相似的。而且 Mercury 的整合性要更好。后续我们会开发基于工作表（Spreedsheet）的界面，方便那些从 Excel 转来的用户。

CHAPTER 3

如何阅读本文档？

本文档为您详细介绍了通联数据量化实验室（Mercury）平台的各种功能和最佳实践。手册整体可以分别分为三部分，分别介绍通联量化实验室的三大基石。每一个部分都是单独功能模块的介绍，相互之间耦合性不高，完全可以独立阅读。对于每一个部分：

QUARTZ

该部分介绍了回测交易平台 Quartz 的使用方式。该部分坚持“新手入门”的原则，围绕例子展开。Quartz 作为功能明确的策略回测框架，十分紧凑，入口单一，非常合适初学者入门。建议读者由头至尾全文阅读。

CAL

该部分介绍金融计算分析工具 CAL。这部分的风格为“新手入门 + 参考手册”。这部分的前两章：引论，如何为金融产品定价 遵循由浅入深，循序渐进的原则。特别是如何为金融产品定价 介绍了 CAL 中的产品-模型-算法周期，建议按照顺序阅读。剩余的章节为分模块介绍，读者完全可以根据自己的需要，选择性阅读。各模块之间我们做了很多相互的交叉引用，读者可以很轻松的找到它们相互之间的联系。

CHAPTER 4

鸣谢

Part II

QUARTZ

CHAPTER 5

Quartz 简介

5.1 什么是 Quartz

- Quartz 是 Python 环境下的一个标准化的量化投资策略回测框架，支持各种类型的日间股票量化投资策略。依照 Quartz 的策略模板，用户可以开发和检验各种独特的投资策略，并方便地进行回测，了解投资策略在市场中的历史表现，比较不同的投资策略的优劣。
- Quartz 基于 Pandas 的 DataFrame 数据结构，同时兼容各类 Python 的数学、统计、金融和机器学习等等扩展模块，如 Numpy, Scipy, statsmodels, Scikit Learn, talib 等等，能够实现技术分析、多因子、数据挖掘、统计套利、事件驱动等等各种流派和风格的股票投资策略，提供了无限的可能性。
- Quartz 提供了全面的量化投资策略研究的工具，覆盖了策略研究的所有阶段。其集成了数据获取功能，使用的是通联数据的数据 API，能够在线迅速高效地获取股票行情数据、指数行情数据和无风险收益率数据等等，用以进行策略的回测。
- Quartz 的框架可以方便地扩展到不同的数据结构和资产类别，在未来，Quartz 将会支持期货策略、高频交易以及固定收益市场等等。

5.2 什么是交易策略

- 一般而言，不管是选股、择时还是数据挖掘、统计套利，交易策略 (或称为投资策略) 都可以归化为如下方法：**根据一定的规则，进行何时何价买入多少数量的何种资产的决策**。一系列的决策导致了投资组合价值的涨跌变化，也反映了策略的优劣。
- 在 Quartz 中，交易策略被具体化为 **根据一定的规则，判断每个交易日以开盘价买入多少数量的何种股票**。并且 Quartz 定义了统一的策略模板和便捷的工具，使得用户可以快速方便地构建各种个性化的交易策略。
- Quartz 中策略的表现是通过虚拟账户来实现的。策略函数在每个交易日下达一系列交易指令，通过这些交易指令的执行，虚拟账户的头寸和价值也随之改变——此即回测的过程，而回测结果可以反映出当前交易策略是否能够盈利、盈利是否稳定等等信息。

5.3 策略一瞥

优矿平台上的一个简单的 Quartz 策略：

Halloween Cycle

策略思路:

- “万圣节效应”：每年10月到次年5月，股票市场会出现上涨的趋势

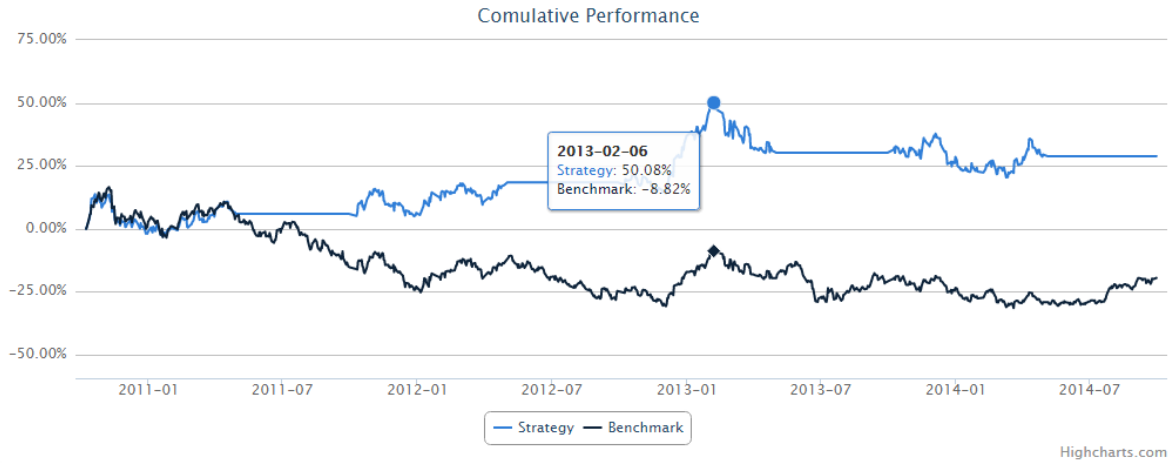
策略实现：

- 股票池：流动性充足的10只个股，包括工商银行、中国石化等
- 每年10月，将账户中现金平均分成10份，分别买入相应的10只个股，满仓；次年5月全部抛出，空仓。

strategy What is Strategy ?

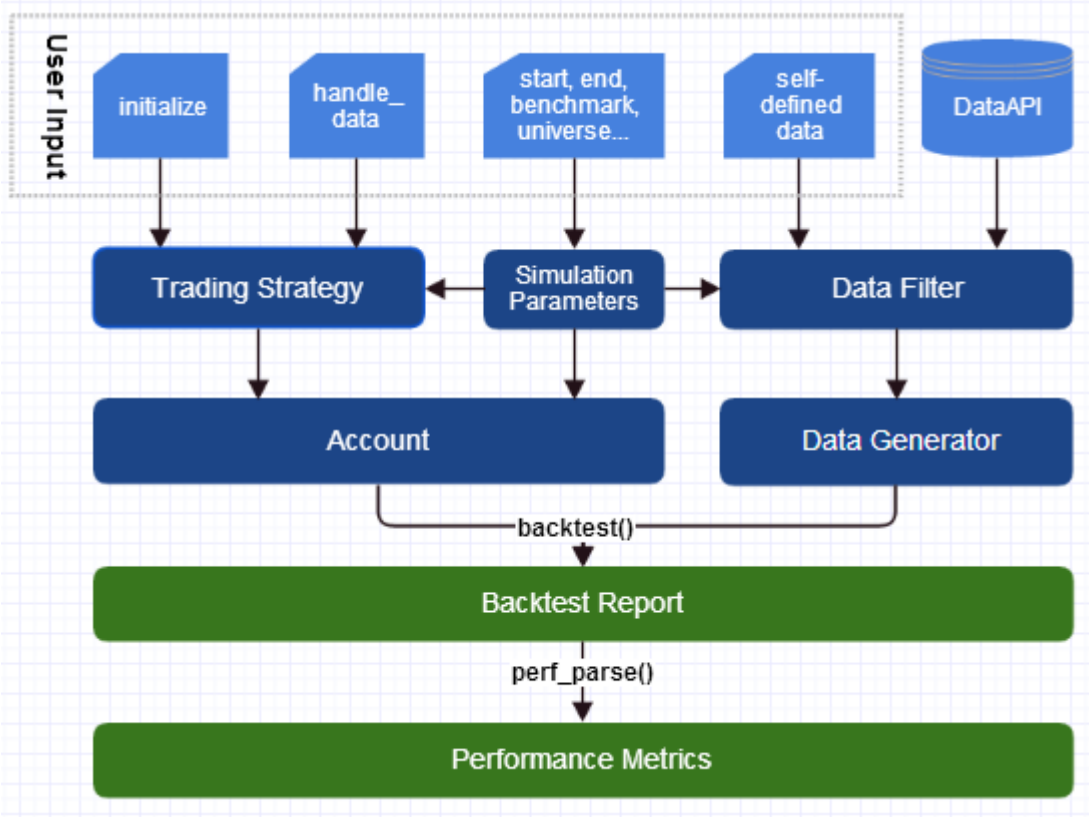
```
start = datetime(2010, 10, 1)
end   = datetime(2014, 10, 1)
benchmark = 'HS300'
universe = ['601398.XSHG', '600028.XSHG', '601988.XSHG', '600036.XSHG', '600030.XSHG',
            '601318.XSHG', '600000.XSHG', '600019.XSHG', '600519.XSHG', '601166.XSHG']
capital_base = 1e6

def initialize(account):
    account.universe = universe

def handle_data(account, data):
    for stock in account.universe:
        if stock not in data: continue

        today = account.current_date
        if (account.position.stkpos.get(stock, 0) == 0) and (today.month == 10 and data[stock]['openPrice']>0):
            amount = int(account.position.cash / len(account.universe) / data[stock]['openPrice'])
            order(stock, amount)
        elif (account.position.stkpos.get(stock, 0) != 0) and (today.month == 5):
            order_to(stock, 0)
```

Annualized Returns	Benchmark Returns	Alpha	Beta	Sharpe Ratio	Volatility	Information Ratio	Max Drawdown
7.4%	-5.0%	0.08	0.52	0.27	0.15	0.77	20.0%



5.4 运行架构

Quartz 的运行架构如下图所示：



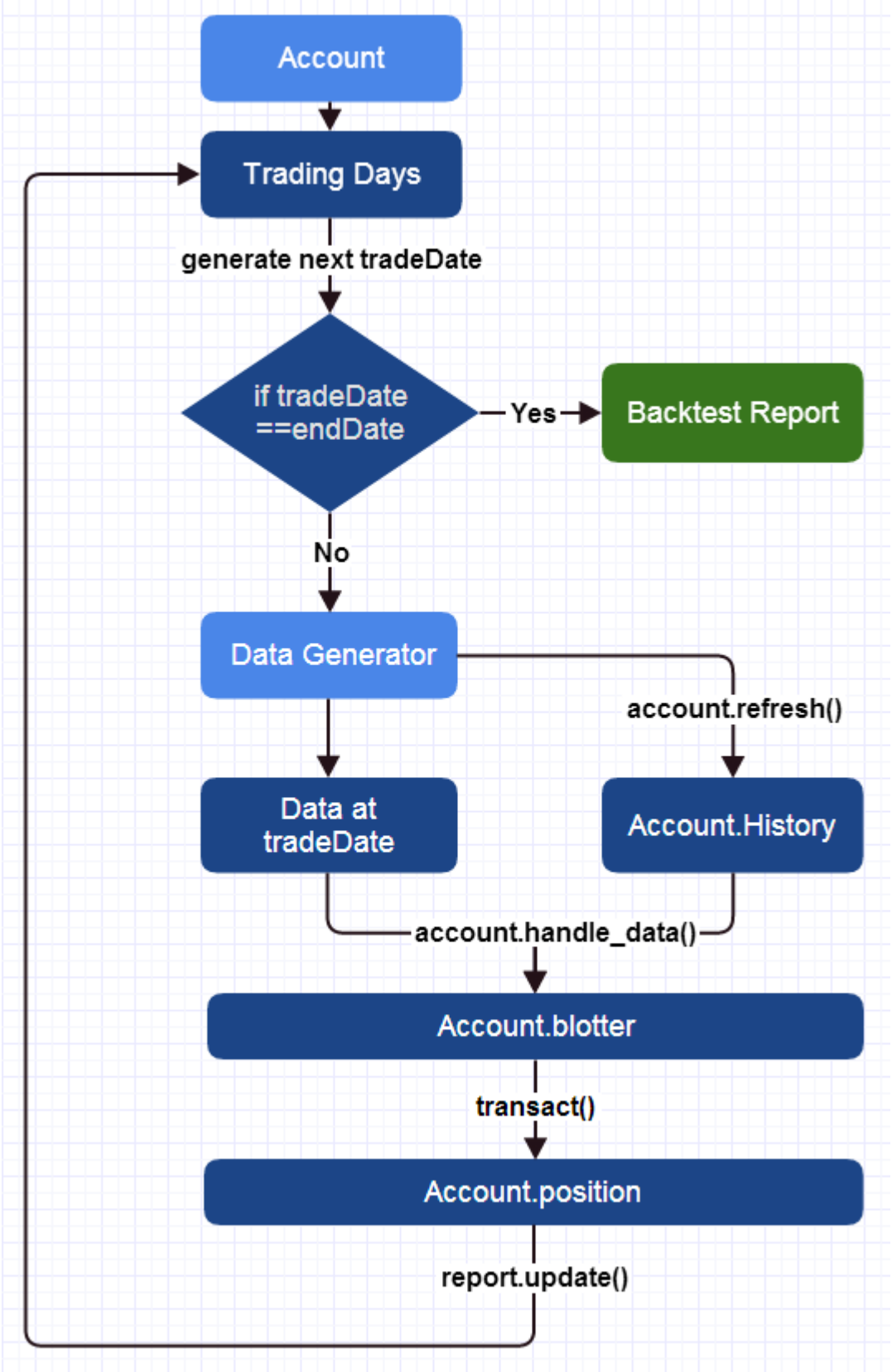
图中浅蓝色的块为输入，绿色的块为输出。用户主要输入为：**回测参数**（Simulation Parameters，包含回测区间和股票池等等）和 **交易策略**（Trading Strategy，包含 initialize 和 handle_data 两个函数）；如有需要，也可添加自定义数据辅助构建交易策略。

从图中可以清楚地看出数据结构的层级，其中的核心是回测参数，因为其中定义的回测区间和股票池等参数是策略构建和数据获取的依据；然后回测参数和交易策略一起对 **虚拟账户**（Account）进行初始化，以作为回测的起点。

另一方面，Quartz 内部集成的数据获取模块会根据回测区间和股票池从通联数据的 DataAPI 中获取数据（Data Filter），并建立 **数据生成器** (Data Generater)，用以生成回测区间中每个交易日的行情数据，进行虚拟账户的模拟交易。

虚拟账户和数据生成器都准备好之后，即可运行回测函数对该策略进行回测，生成 **回测报告**（Backtest Report），其中包括每个交易日虚拟账户的现金股票头寸及策略的交易指令记录等等信息。进而，根据回测报告中的整个账户市场价值的变化，Quartz 会计算这一策略的各项风险收益指标（Performance Metrics），如年化收益率、累计收益率、Sharpe 比率、最大回撤等等（全部指标见这里）。

在以上介绍的 Quartz 运行架构中，最重要的函数是进行回测的 backtest()。为了将 Quartz 的回测机制阐述得更加明白，下面给出了回测日内策略时 backtest() 的流程图：



输入和输出模块的颜色与前一张图是一致的。在 backtest() 中，首先需要 **初始化虚拟账户和数据生成**

器，并 **创建回测报告**。而后对于回测区间内的 **交易日列表进行循环**，在每个交易日内：

1. 生成当日行情数据
2. 更新交易指令簿、历史数据窗口等
3. 执行策略，生成交易指令簿（不可使用当日行情数据）
4. 根据当日行情数据，对交易指令进行模拟交易，更新现金及股票头寸
5. 记录虚拟账户状态至回测报告

最后 **输出整个回测报告**，作为计算风险收益指标的基础。

以上为对 Quartz 整体架构和核心函数运行流程的解析，接下来就可以开始一步步地开始自己的量化策略之旅了。

5.5 Frequently Asked Questions (FAQ)

Q: 回测结果中策略收益无论涨跌比例都很小，为什么？A: 检查参数 capital_base 的数值，如果数值过大而下单的股票数量较小的话，则会出现策略收益过小的情况。

Q: 下单数量必须是整百吗？A: 不需要，回测中允许零数股票买卖，而在模拟交易中，程序在后台会自动帮用户把下单数量处理成最接近的整百（绝对值向下取整，即 345 -> 300, -256 -> -200）。

Q: 下单买入股票但是现金不够了，会发生什么？A: 此单会成交一部分，从输出的回测报告中调出当天的 blotter 细节，可以看到 Order 里有一个属性是 filled，这个属性即是指这一笔 Order 成交的数量。

Q: 遇到涨跌停怎么办？A: Quartz 默认的下单思路是在当日开盘之前决定所有交易指令，除了开盘涨（跌）停并全天封在涨（跌）停板的情况买不到（卖不出去）之外，其他行情在资金和股票允许的情况下都是可以成交的。

Q: 如果买卖数量较大，都能成交吗？A: Quartz 对交易的判断所用买卖数量上限是当日的交易量，如果指令数量超出了该股票当日交易量，只会成交相当于当日交易量的部分。

Q: 有时候 Order 不能成交，为什么？A: 如果是买入的话，原因是当天该股票停牌，即该股票交易量为 0；如果是卖出的话，原因是虚拟账户中没有足够多的该股票。

Q: 使用 account.get_symbol_history 时遇到"Please verify the symbol" 的错误信息，或者使用 account.get_history 或者 account.get_attribute_history 之后使用数据时遇到"KeyError"，为什么？A: 可能是该股票当日停牌，建议在循环股票池的时候采用 for stock in account.universe 的写法。

Q: 有的时候回测没有问题但是模拟交易经常会遇到现金不够的情况，为什么？A: 一般下单的股票数量是使用昨日收盘价进行计算的，当日开盘价相比昨日收盘价会有一定的波动，而模拟交易系统的下单规则是现金不够就打回该笔下单，所以会出现这样的情况。为了避免这样的情况，建议用户在下单时避免使用过于贴近边界条件的数额，比如在预计使用的资金上乘以一个小于 1 的系数，将会很有效地减少这种状况的产生。

Q: 有的时候模拟交易中出现了"订单信号已经产生"却并没有"发送订单到交易系统"，为什么？A: 原因可能在于 blotter 中的订单股数均低于 100 股，故未被确认成有效订单。

CHAPTER 6

10 Minutes to Quartz

Quartz 理想的运行环境是 ipython notebook，并且已经配置好了 Pandas, Numpy 等等工具包。

6.1 导入所需模块

在 Quartz 中构建一个可靠的策略之前，首先需要导入 Quartz 的程序模块。

一般而言，为了能够让程序运转正常，需要导入 Quartz 本身和quartz.api 中的所有函数。quartz.api 当中则包含了许多构建策略不可缺少的函数，如股票池快捷设置、模拟下达交易指令、记录自定义变量等等。

```
import quartz
from quartz.api import *
```

因为 Quartz 当中的数据结构是基于 datetime，Numpy 和 Pandas 的，为了获取数据以及计算指标的方便，也需要导入相应的模块。

此外，根据我们对策略的构思不同，还可以根据情况导入 Scipy, statsmodels, Scikit Learn, talib 等等各种技术分析、数学建模、机器学习的 python 模块。

```
import pandas as pd
import numpy as np
from datetime import datetime
from matplotlib import pylab
```

返回目录

6.2 定义回测参数

在开始着手写一个量化投资策略之前，我们得先做一些假设，定义策略执行和回测的基础和环境。在代码层面上，我们通过参数定义来实现这一点：

```
start = '2014-01-01'                    # 回测起始时间
end   = '2015-01-01'                    # 回测结束时间
benchmark = 'HS300'                     # 参考标准
universe  = ['000001.XSHE', '600000.XSHG']  # 证券池，支持股票和基金
capital_base = 100000                   # 起始资金
refresh_rate = 1                        # 调仓频率，即每 refresh_rate 个交易日执行一次 handle_data() 函数
```

上述 6 个参数是最重要的。通过这 6 个参数，我们就确定了回测的区间、策略表现好坏的参考标准 [1] 、选择证券的范围 [2] 、虚拟的起始资金数量以及回测的频率。

除了以上 6 个参数，根据策略构建的实际需要，还可以定义外部数据、起始证券仓位、手续费标准等等诸多参数，具体参见这里 ，十分灵活。

返回目录

6.3 构建日间策略

模块已经导入好了，参数也已经定义好了，接下来就让我们开始构建第一个 Quartz 框架下的交易策略。

在写策略代码以前，我们需要大概了解一下 Quartz 回测的机制：

- 在每个策略开始回测之前，都会建立一个虚拟的交易账户，这个账户包含了回测区间的交易日、现金与证券的头寸、每日交易指令明细、历史数据接口等等内容
- 在每个交易日的开盘之前，交易策略会根据历史数据或者其他信息进行交易判断，模拟下达交易指令
- 接下来 Quartz 会根据当天的市场数据对这些指令进行能否交易的判断，并更新虚拟账户当中的现金数量、证券头寸和交易指令信息
- 随后该交易日结束，清空虚拟账户中的交易指令列表，循环进入下一个交易日

在这样的框架下，我们需要定义两个函数，来实现交易策略的逻辑：

1. initialize(account)，在回测开始之前运行，用以初始化虚拟账户当中一些特别的信息
2. handle_data(account)，在每个交易日开盘之前运行，用来执行策略，判断交易指令下达的时机和细节

以上两者也构成了 Quartz 中的交易策略的标准化框架。

现在让我们来实现一个最简单的交易策略：**每天买入一手证券池里的所有证券**。

按照标准化框架，我们可以很容易地确定 initialize() 和 handle_data() 各自所要做的事情：

[1] benchmark 目前支持以下 5 个指数：上证综指 (SHCI)、上证 50(SH50)、上证 180(SH180)、沪深 300(HS300) 和中证 500(ZZ500)；此外，用户还可以自定义股票或指数来个性化参照标准

[2] 证券池中的证券代码必须有后缀，上证证券为.XSHG，深证证券为.XSHE

```
def initialize(account):          # 初始化虚拟账户状态
    pass

def handle_data(account):         # 每次交易的买入卖出指令
    for s in account.universe:
        order(s, 100)
```

注意到，initialize() 和 handle_data() 的参数为 account。其中 account 是虚拟交易账户，**不需要用户对其进行赋值**，详情见Account，并且用户可以在这两个函数中为 account 添加新的属性。account.universe 表示交易日当天可以进行交易的证券池。

除此之外，比较陌生的应该就只有 order(s, 100) 这一句了。

注解: order(symbol, amount) 是quartz.api 里的一个函数，用来模拟下达买卖指令，买入 amount 数量的代码为 symbol 的证券 (如果 amount 数量为负，则为卖出相应数量的证券)。

这样，一个不停买入的投资策略就完成了。尽管还很简陋，但这是走向各种复杂策略的第一步！

返回目录

6.4 进行回测

现在策略已经构建好了，那么这个策略的效果究竟如何呢？和沪深 300 指数相比，这个策略是否能够胜出呢？使用历史数据对策略略进行回测可以回答这些问题。

回测使用的是backtest 函数，回测函数的输入是之前我们定义好的参数和策略函数：

```
bt, acct = quartz.backtest(start = start,
                           end    = end,
                           benchmark = benchmark,
                           universe  = universe,
                           capital_base = capital_base,
                           initialize   = initialize,
                           handle_data  = handle_data,
                           refresh_rate = refresh_rate)
```

函数的输出——bt——包含了每个交易日收盘之后虚拟账户的详细信息，包含日期、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，数据结构为 pandas.DataFrame，形如：

	tradeDate	cash	security_position	portfolio_value	benchmark_return	blotter
0	2014-01-02	97841.8440	{u'000001.XSHE': 100, u'600000.XSHG': 100}	99997.8440	-0.003454	[Order(time: datetime.datetime(2014, 1, 2, 0, ...
1	2014-01-03	95696.7010	{u'000001.XSHE': 200, u'600000.XSHG': 200}	99910.7010	-0.013436	[Order(time: datetime.datetime(2014, 1, 3, 0, ...
2	2014-01-06	93574.5810	{u'000001.XSHE': 300, u'600000.XSHG': 300}	99832.5810	-0.022762	[Order(time: datetime.datetime(2014, 1, 6, 0, ...
3	2014-01-07	91502.5110	{u'000001.XSHE': 400, u'600000.XSHG': 400}	99810.5110	-0.000284	[Order(time: datetime.datetime(2014, 1, 7, 0, ...
4	2014-01-08	89422.4330	{u'000001.XSHE': 500, u'600000.XSHG': 500}	99902.4330	0.001747	[Order(time: datetime.datetime(2014, 1, 8, 0, ...
5	2014-01-09	87331.3440	{u'000001.XSHE': 600, u'600000.XSHG': 600}	99997.3440	-0.008783	[Order(time: datetime.datetime(2014, 1, 9, 0, ...
6	2014-01-10	85223.2380	{u'000001.XSHE': 700, u'600000.XSHG': 700}	100084.2380	-0.007817	[Order(time: datetime.datetime(2014, 1, 10, 0,...

注解: 可以通过 bt.at[row, col] 查看行为 row 列为 col 的详细数据，如 bt.at[15,'cash']

而 acct 包含了一些回测的数据，作为输出是为了后续在策略表现分析的函数中使用。

我们现在了解了整个回测期间虚拟账户的情况，接下来就可以利用这些数据计算出这个策略的各种表现指标，于是我们需要用到perf_parse 函数 ：

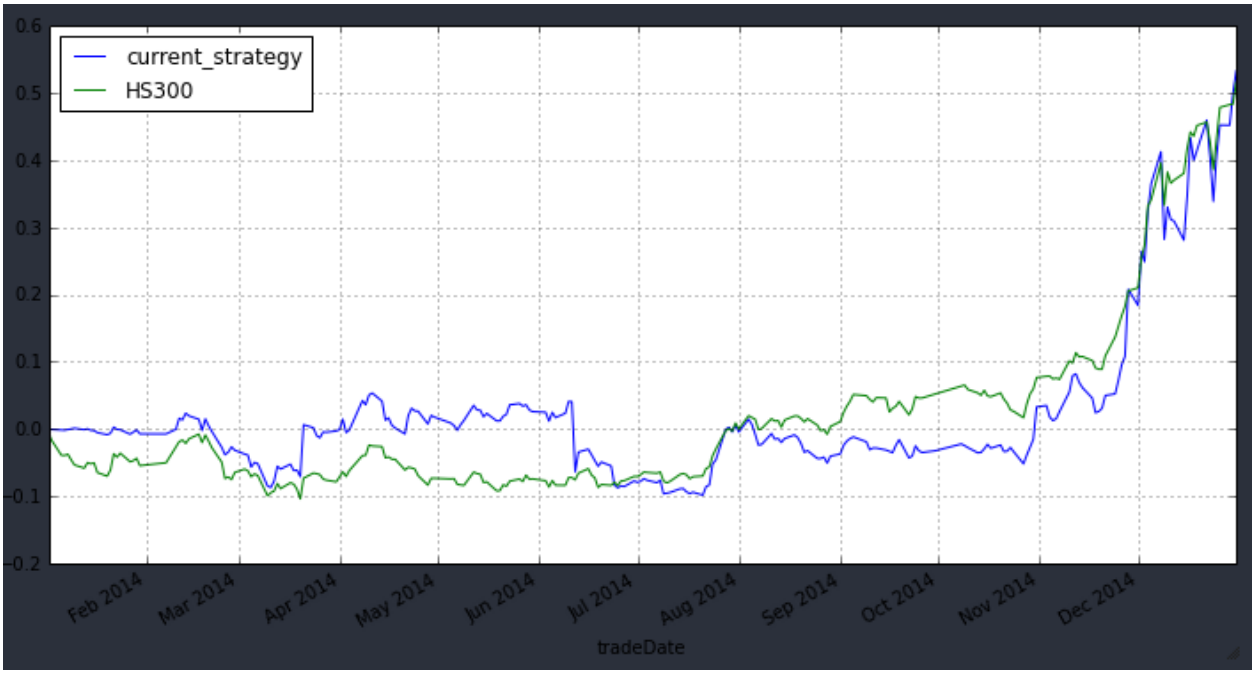
```
perf = quartz.perf_parse(bt, acct)

out_keys = ['annualized_return', 'volatility', 'information_ratio',
            'sharpe', 'max_drawdown', 'alpha', 'beta']
for k in out_keys:
    print '%s: %s' % (k, perf[k])
```

perf_parse 函数的输出是一个字典，包含了许多可能有关策略和参照标准的风险收益指标，具体的指标列表可见这里 。这里只列出了最重要的几个指标：**年化收益率**、**波动率**、**信息比率**、**夏普比率**、**最大回测**、**阿尔法和贝塔**。输出结果如下：

```
annualized_return: 0.544001693983
volatility: 0.309930176843
information_ratio: 0.191885351107
sharpe: 1.60662539884
max_drawdown: 0.144353737704
alpha: -0.0384372445785
beta: 1.19871387581
```

仅仅有这些数据还是不够形象，我们希望能够画出策略和参照标准累计收益率随时间的变化情况，这样可以直观地看出策略与参照标准的对比情况。利用 pandas 数据结构方便的画图功能，我们可以很容易地做到这一点：

```
perf['cumulative_returns'].plot()
perf['benchmark_cumulative_returns'].plot()
pylab.legend(['current_strategy','HS300'])
```



现在我们学会如何在 Quartz 下构建一个投资策略并进行回测了，然而想要开发更加复杂的量化投资策略，还需要一些其他的工具，例如下一节介绍的 **使用历史数据**。

返回目录

6.5 使用历史数据

现在我们已经可以初步了解如何在 Quartz 中写一个策略，并进行回测以及获得其表现数据了。然而前一个策略太过简陋，然而很多想法都是与股票过去一段时间的价格、成交量等等指标有关的，于是我们需要在策略中取得并使用历史数据。下面就让我们看看怎么样在策略中建立历史数据窗口，让我们在写策略的时候如虎添翼。

我们还是通过一个策略的例子来进行介绍：

```
def initialize(account):
    return

def handle_data(account):
    hist = account.get_attribute_history('closePrice', 10)
    for s in account.universe:
        if hist[s][-1]/hist[s][0]-1 > 0.01 and s not in account.valid_secpos:
            order(s, 10000)
        elif hist[s][-1]/hist[s][0]-1 < 0 and s in account.valid_secpos:
            order_to(s, 0)
```

这个策略的思路是：计算股票前 10 个交易日的累计收益率，如果累计收益率大于 1% 并且手中没有该股票则买入 10000 股，如果累计收益率小于 0 并且手中有该股票则全部抛出。

为了方便起见，其他参数仍然使用前面的定义，这里只重新给出了策略主体——initialize() 和 handle_data()。这个策略看上去要复杂多了，细看下来，有几个语句是没见过的，这几个语句就是我们要介绍的历史数据窗口的使用方法。

首先，使用历史数据需要定义窗口长度，并且该窗口长度决定了回测真正的开始日期：从 start 往后数窗口长度个交易日。如果有多个历史数据获取语句的话，会用最长的窗口长度来决定回测何时开始。

其次，我们来看account.get_attribute_history(attribute, period)，其是 Account 虚拟账户的一个方法，其作用是获取所有证券的某个属性、长度为 period 的历史数据窗口。属性历史数据窗口是一个字典，键为股票代码，值为每只证券的历史数据。在示例中，2014 年 1 月 22 日的属性历史数据窗口是这样的：

```
{
  000001.XSHE: [11.76, 11.82, 11.82, 11.6, 11.72, 11.67, 11.69, 11.48, 11.3, 11.36]
  600000.XSHG: [9.2, 9.29, 9.41, 9.39, 9.4, 9.23, 9.18, 9.12, 9.07, 9.16]
}
```

除了属性历史窗口之外，还有另外两种获取历史数据窗口的方法，分别为account.get_symbol_history(symbol, period)（获取单只证券的全部历史属性）与account.get_history(period)()。2014 年 1 月 22 日的这两种历史数据窗口示例分别为：

单只证券

```
account.get_symbol_history('000001.XSHE', 10)
{
  closePrice: [11.76, 11.82, 11.82, 11.6, 11.72, 11.67, 11.69, 11.48, 11.3, 11.36]
  turnoverValue: [538436160.0, 576870530.0, 450487744.0, 559375170.0, 420715264.0, 346733024.0, 350027744.0, 487423200.0, 349154112.
0, 303578624.0]
  turnoverVol: [45776816.0, 48551836.0, 38119764.0, 47887516.0, 36242708.0, 29798810.0, 29959340.0, 42351388.0, 30721992.0, 26579498
.0]
  lowPrice: [11.53, 11.65, 11.66, 11.49, 11.45, 11.56, 11.58, 11.45, 11.25, 11.32]
  highPrice: [11.95, 11.99, 11.95, 11.91, 11.78, 11.74, 11.8, 11.64, 11.48, 11.56]
  openPrice: [11.64, 11.69, 11.78, 11.8, 11.57, 11.7, 11.66, 11.62, 11.48, 11.32]
  preClosePrice: [11.63, 11.76, 11.82, 11.82, 11.6, 11.72, 11.67, 11.69, 11.48, 11.3]
}
```

全部数据

```
account.get_history(10)
{
  000001.XSHE:  {
    closePrice: [11.76, 11.82, 11.82, 11.6, 11.72, 11.67, 11.69, 11.48, 11.3,
 11.36]
    turnoverValue: [538436160.0, 576870530.0, 450487744.0, 559375170.0, 42071
5264.0, 346733024.0, 350027744.0, 487423200.0, 349154112.0, 303578624.0]
    turnoverVol: [45776816.0, 48551836.0, 38119764.0, 47887516.0, 36242708.0,
 29798810.0, 29959340.0, 42351388.0, 30721992.0, 26579498.0]
    lowPrice: [11.53, 11.65, 11.66, 11.49, 11.45, 11.56, 11.58, 11.45, 11.25,
 11.32]
    highPrice: [11.95, 11.99, 11.95, 11.91, 11.78, 11.74, 11.8, 11.64, 11.48,
 11.56]
    openPrice: [11.64, 11.69, 11.78, 11.8, 11.57, 11.7, 11.66, 11.62, 11.48,
11.32]
    preClosePrice: [11.63, 11.76, 11.82, 11.82, 11.6, 11.72, 11.67, 11.69, 11
.48, 11.3]
  }
  600000.XSHG:  {
    closePrice: [9.2, 9.29, 9.41, 9.39, 9.4, 9.23, 9.18, 9.12, 9.07, 9.16]
    turnoverValue: [724415940.0, 1025797630.0, 875061950.0, 618500160.0, 7444
81150.0, 717369980.0, 618488770.0, 601804100.0, 382309888.0, 436604864.0]
    turnoverVol: [78603480.0, 109976760.0, 93379976.0, 65935364.0, 79710264.0
, 77606640.0, 67204744.0, 66047496.0, 42094792.0, 47596032.0]
    lowPrice: [9.11, 9.16, 9.24, 9.33, 9.25, 9.16, 9.17, 9.07, 9.04, 9.09]
    highPrice: [9.32, 9.44, 9.45, 9.47, 9.43, 9.4, 9.26, 9.17, 9.13, 9.26]
    openPrice: [9.14, 9.2, 9.28, 9.44, 9.4, 9.39, 9.22, 9.17, 9.1, 9.1]
    preClosePrice: [9.14, 9.2, 9.29, 9.41, 9.39, 9.4, 9.23, 9.18, 9.12, 9.07]
  }
  benchmark:  {
    preCloseIndex: [2238.001, 2241.911, 2222.221, 2204.851, 2193.679, 2212.84
6, 2208.941, 2211.844, 2178.488, 2165.993]
    return: [0.00174709484044, -0.00878268584257, -0.00781650429908, -0.00506
70090632, 0.00873737679943, -0.00176469578091, 0.00131420440836, -0.015080629
5562, -0.00573562948247, 0.00988784358952]
    closeIndex: [2241.911, 2222.221, 2204.851, 2193.679, 2212.846, 2208.941,
2211.844, 2178.488, 2165.993, 2187.41]
  }
  tradeDate:
    [datetime(2014,01,08), datetime(2014,01,09), datetime(2014,01,10), dateti
me(2014,01,13), datetime(2014,01,14), datetime(2014,01,15), datetime(2014,01,
16), datetime(2014,01,17), datetime(2014,01,20), datetime(2014,01,21)]
}
```

了解了历史数据的格式之后，我们就不难理解 handle_data() 当中的 hist[s][0] 了，其作用即是选取股票 s 历史数据窗口中第一个（也是最早的一个）收盘价。同理可以理解 hist[s][-1]。

注解: 在判断买入卖出时，条件中涉及到了另外一个有助于策略构建的重要工具，这就是 account.valid_secpos。从使用方式中可以看出，这是虚拟账户的一个属性。这个属性的数据结构为字典，键为证券代码，值为虚拟账户当前所持有该证券的数量。

由此即可以理解

```
if s not in account.valid_secpos:
```

的意思为：

```
如果当前虚拟账户的仓位中没有证券 s 的话...
```

此外，虚拟账户还提供了另一个便捷的属性来获得当前的现金头寸，即 account.cash，其为数字，记录的是上一个交易日结束之后的虚拟账户剩余现金头寸

注解: 在示例代码中，除了之前出现过的 order() 之外，出现了另一种交易指令的下达方式order_to() 。与 order 类似，order_to(symbol, amount) 也有股票代码和数量两个参数；不同的地方在于，order_to() 的含义是进行交易使得虚拟账户中的该股票数量在 **这笔交易成功之后成为** amount。

这一策略的表现如下：

```
annualized_return: -0.017402752766
volatility: 0.361213640134
information_ratio: -1.60250887639
sharpe: -0.175252941009
max_drawdown: 0.364498757949
alpha: -0.517433088191
beta: 1.10088629025
<matplotlib.legend.Legend at 0xcf4c240>
```

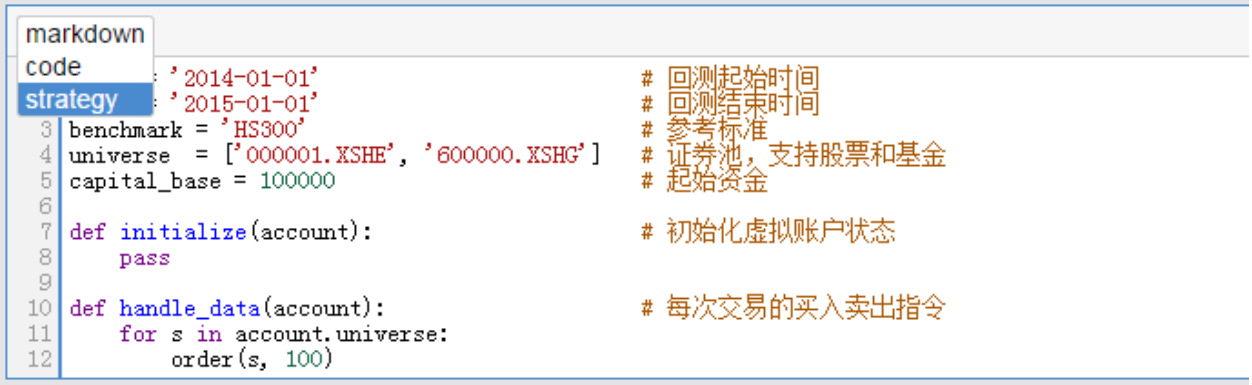
至此，我们已经掌握了 Quartz 中构建策略的步骤和基本工具，现在可以充分发挥自己的聪明才智，开始挖掘收益更高更稳定的量化投资策略了！

返回目录

10 Minutes to Quartz (UQER 版)

在 UQER notebook 当中，每个 cell 的左上角可以选择该 cell 的属性。在这些属性当中，strategy 属性即意味着这个 cell 的内容是一个 Quartz 环境下的量化投资策略。

将 cell 的属性设置成 strategy 之后，可以看到 cell 中出现了一个投资策略的初始模板：



从初始模板开始，可以一步步地构建各种复杂的投资策略。

- 定义回测参数
- 构建日间策略
- 进行回测
- 使用历史数据

7.1 定义参数

初始 strategy 模板由两部分组成：第一部分是前 6 行，这 6 行定义了量化投资策略回测的各项参数，即参数定义部分。

```
start = '2014-01-01'                # 回测起始时间
end   = '2015-01-01'                # 回测结束时间
benchmark = 'HS300'                 # 参考标准
```

```
universe = ['000001.XSHE', '600000.XSHG']  # 证券池，支持股票和基金
capital_base = 100000                      # 起始资金
refresh_rate = 1                           # 调仓频率，即每 refresh_rate 个交易日执行一次 handle_data() 函数
```

上述 6 个参数是最重要的。通过这 6 个参数，我们就确定了回测的区间、策略表现好坏的参考标准 [1] 、选择证券的范围 [2] 、虚拟的起始资金数量以及回测的频率。

除了以上 6 个参数，根据策略构建的实际需要，还可以定义外部数据、起始证券仓位、手续费标准等等诸多参数，具体参见这里 ，十分灵活。

返回目录

7.2 构建日间策略

模板的第二部分包括 initialize() 和 handle_data() 两个函数，是 Quartz 框架下的交易策略的主体。在开始构建策略的主体代码以前，我们需要大概了解一下 Quartz 回测的机制：

- 在每个策略开始回测之前，都会建立一个虚拟的交易账户，这个账户包含了回测区间的交易日、现金与证券的头寸、每日交易指令明细、历史数据接口等等内容
- 在每个交易日的开盘之前，交易策略会根据历史数据或者其他信息进行交易判断，模拟下达交易指令
- 接下来 Quartz 会根据当天的市场数据对这些指令进行能否交易的判断，并更新虚拟账户当中的现金数量、证券头寸和交易指令信息
- 随后该交易日结束，清空虚拟账户中的交易指令列表，循环进入下一个交易日

在这样的框架下，我们需要定义两个函数，来实现交易策略的逻辑：

1. initialize(account)，在回测开始之前运行，用以初始化虚拟账户当中一些特别的信息
2. handle_data(account)，在每个交易日开盘之前运行，用来执行策略，判断交易指令下达的时机和细节

以上两者也构成了 Quartz 中的交易策略的标准化框架。

现在让我们来实现一个最简单的交易策略：**每天买入一手证券池里的所有证券**。

按照标准化框架，我们可以很容易地确定 initialize() 和 handle_data() 各自所要做的事情：

```
def initialize(account):          # 初始化虚拟账户状态
    pass

def handle_data(account):         # 每次交易的买入卖出指令
    for s in account.universe:
        order(s, 100)
```

[1] benchmark 目前支持以下 5 个指数：上证综指 (SHCI)、上证 50(SH50)、上证 180(SH180)、沪深 300(HS300) 和中证 500(ZZ500)；此外，用户还可以自定义股票或指数来个性化参照标准

[2] 证券池中的证券代码必须有后缀，上证证券为.XSHG，深证证券为.XSHE

注意到，initialize() 和 handle_data() 的参数为 account。其中 account 是虚拟交易账户，**不需要用户对其进行赋值**，详情见Account，并且用户可以在这两个函数中为 account 添加新的属性。account.universe 表示交易日当天可以进行交易的证券池。

除此之外，比较陌生的应该就只有 order(s, 100) 这一句了。

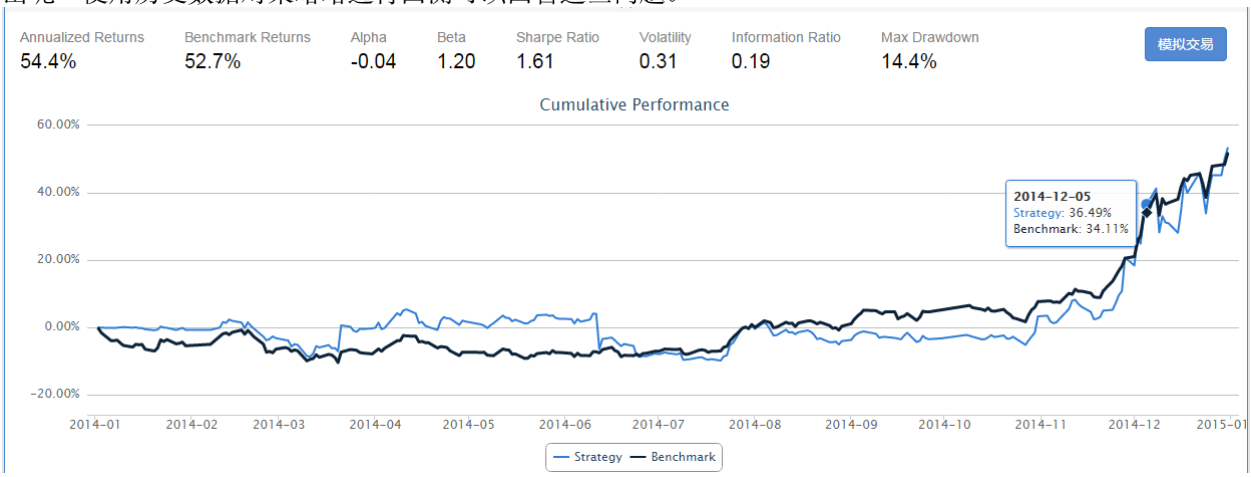
注解: order(symbol, amount) 是quartz.api 里的一个函数，用来模拟下达买卖指令，买入 amount 数量的代码为 symbol 的证券（如果 amount 数量为负，则为卖出相应数量的证券）。

这样，一个不停买入的投资策略就完成了。尽管还很简陋，但这是走向各种复杂策略的第一步！

返回目录

7.3 进行回测

现在策略已经构建好了，那么这个策略的效果究竟如何呢？和沪深 300 指数相比，这个策略是否能够胜出呢？使用历史数据对策略略进行回测可以回答这些问题。



如上图所示，strategy 属性 cell 的输出即为有当前 initialize 和 handle_data 两个函数构建的策略的回测结果。从图中可见回测结果由两部分组成，上面是一些主要的风险收益指标，如年化收益、Sharpe 比率、最大回撤等等；下面是当前策略和参照标准累计收益率随时间变化的对比图像，以更直观地看出策略的表现情况。

此外，strategy 还提供了两个变量记录 quartz 回测的结果，bt **和** perf。

bt 记录了每个交易日收盘之后虚拟账户的详细信息，包含日期、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，数据结构为 pandas.DataFrame，形如：

```
code
1 bt[:10]
```

	tradeDate	cash	security_position	portfolio_value	benchmark_return	blotter
0	2014-01-02	97841.844	{u'000001.XSHE': 100, u'600000.XSHG': 100}	99997.844	-0.003454	[Order(order time: datetime.datetime(2014, 1, ...
1	2014-01-03	95696.701	{u'000001.XSHE': 200, u'600000.XSHG': 200}	99910.701	-0.013436	[Order(order time: datetime.datetime(2014, 1, ...
2	2014-01-06	93574.581	{u'000001.XSHE': 300, u'600000.XSHG': 300}	99832.581	-0.022762	[Order(order time: datetime.datetime(2014, 1, ...
3	2014-01-07	91502.511	{u'000001.XSHE': 400, u'600000.XSHG': 400}	99810.511	-0.000284	[Order(order time: datetime.datetime(2014, 1, ...
4	2014-01-08	89422.433	{u'000001.XSHE': 500, u'600000.XSHG': 500}	99902.433	0.001747	[Order(order time: datetime.datetime(2014, 1, ...
5	2014-01-09	87331.344	{u'000001.XSHE': 600, u'600000.XSHG': 600}	99997.344	-0.008783	[Order(order time: datetime.datetime(2014, 1, ...
6	2014-01-10	85223.238	{u'000001.XSHE': 700, u'600000.XSHG': 700}	100084.238	-0.007817	[Order(order time: datetime.datetime(2014, 1, ...
7	2014-01-13	83097.114	{u'000001.XSHE': 800, u'600000.XSHG': 800}	99889.114	-0.005067	[Order(order time: datetime.datetime(2014, 1, ...
8	2014-01-14	80998.017	{u'000001.XSHE': 900, u'600000.XSHG': 900}	100006.017	0.008737	[Order(order time: datetime.datetime(2014, 1, ...
9	2014-01-15	78886.908	{u'000001.XSHE': 1000, u'600000.XSHG': 1000}	99786.908	-0.001765	[Order(order time: datetime.datetime(2014, 1, ...

注解: 可以通过 bt.at[row, col] 查看行为 row 列为 col 的详细数据，如 bt.at[15,'cash']

perf 是一个字典，包含了许多可能有关策略和参照标准的风险收益指标，可以根据具体的指标名称查询更加详细的指标内容。这里给出了所有指标的列表：

```
code
1 perf.keys()
```

```
['max_drawdown',
 'treasury_return',
 'information_coefficient',
 'benchmark_cumulative_values',
 'benchmark_annualized_return',
 'turnover_rate',
 'cumulative_returns',
 'beta',
 'benchmark_volatility',
 'returns',
 'excess_return',
 'benchmark_returns',
 'benchmark_cumulative_returns',
 'sharpe',
 'alpha',
 'volatility',
 'information_ratio',
 'annualized_return',
 'cumulative_values']
```

现在我们学会如何使用 Mercury notebook 和 Quartz 构建投资策略并进行回测，然而想要开发更加复

杂的量化投资策略，还需要一些其他的工具，例如下一节介绍的 **使用历史数据**。

返回目录

7.4 使用历史数据

现在我们已经可以初步了解如何在 Quartz 中写一个策略，并进行回测以及获得其表现数据了。然而前一个策略太过简陋，然而很多想法都是与股票过去一段时间的价格、成交量等等指标有关的，于是我们需要在策略中取得并使用历史数据。下面就让我们看看怎么样在策略中建立历史数据窗口，让我们在写策略的时候如虎添翼。

我们还是通过一个策略的例子来进行介绍：

```
def initialize(account):
    return

def handle_data(account):
    hist = account.get_attribute_history('closePrice', 10)
    for s in account.universe:
        if hist[s][-1]/hist[s][0]-1 > 0.01 and s not in account.valid_secpos:
            order(s, 10000)
        elif hist[s][-1]/hist[s][0]-1 < 0 and s in account.valid_secpos:
            order_to(s, 0)
```

这个策略的思路是：计算股票前 10 个交易日的累计收益率，如果累计收益率大于 1% 并且手中没有该股票则买入 10000 股，如果累计收益率小于 0 并且手中有该股票则全部抛出。

为了方便起见，其他参数仍然使用前面的定义，这里只重新给出了策略主体——initialize() 和 handle_data()。这个策略看上去要复杂多了，细看下来，有几个语句是没见过的，这几个语句就是我们要介绍的历史数据窗口的使用方法。

首先，使用历史数据需要定义窗口长度，并且该窗口长度决定了回测真正的开始日期：从 start 往后数窗口长度个交易日。如果有多个历史数据获取语句的话，会用最长的窗口长度来决定回测何时开始。

其次，我们来看account.get_attribute_history(attribute, period)，其是 Account 虚拟账户的一个方法，其作用是获取所有证券的某个属性、长度为 period 的历史数据窗口。属性历史数据窗口是一个字典，键为股票代码，值为每只证券的历史数据。在示例中，2014 年 1 月 22 日的属性历史数据窗口是这样的：

```
{
        000001.XSHE: [11.76, 11.82, 11.82, 11.6, 11.72, 11.67, 11.69, 11.48, 11.3, 11.36]
        600000.XSHG: [9.2, 9.29, 9.41, 9.39, 9.4, 9.23, 9.18, 9.12, 9.07, 9.16]
}
```

除了属性历史窗口之外，还有另外两种获取历史数据窗口的方法，分别为account.get_symbol_history(symbol, period)（获取单只证券的全部历史属性）与account.get_history(period)()。2014 年 1 月 22 日的这两种历史数据窗口示例分别为：

单只证券

```
account.get_symbol_history('000001.XSHE', 10)
{
        closePrice: [11.76, 11.82, 11.82, 11.6, 11.72, 11.67, 11.69, 11.48, 11.3, 11.36]
        turnoverValue: [538436160.0, 576870530.0, 450487744.0, 559375170.0, 420715264.0, 346733024.0, 350027744.0, 487423200.0, 349154112.0, 303578624.0]
        turnoverVol: [45776816.0, 48551836.0, 38119764.0, 47887516.0, 36242708.0, 29798810.0, 29959340.0, 42351388.0, 30721992.0, 26579498.0]
        lowPrice: [11.53, 11.65, 11.66, 11.49, 11.45, 11.56, 11.58, 11.45, 11.25, 11.32]
        highPrice: [11.95, 11.99, 11.95, 11.91, 11.78, 11.74, 11.8, 11.64, 11.48, 11.56]
        openPrice: [11.64, 11.69, 11.78, 11.8, 11.57, 11.7, 11.66, 11.62, 11.48, 11.32]
        preClosePrice: [11.63, 11.76, 11.82, 11.82, 11.6, 11.72, 11.67, 11.69, 11.48, 11.3]
}
```

全部数据

```
account.get_history(10)
{
        000001.XSHE: {
                closePrice: [11.76, 11.82, 11.82, 11.6, 11.72, 11.67, 11.69, 11.48, 11.3, 11.36]
                turnoverValue: [538436160.0, 576870530.0, 450487744.0, 559375170.0, 420715264.0, 346733024.0, 350027744.0, 487423200.0, 349154112.0, 303578624.0]
                turnoverVol: [45776816.0, 48551836.0, 38119764.0, 47887516.0, 36242708.0, 29798810.0, 29959340.0, 42351388.0, 30721992.0, 26579498.0]
                lowPrice: [11.53, 11.65, 11.66, 11.49, 11.45, 11.56, 11.58, 11.45, 11.25, 11.32]
                highPrice: [11.95, 11.99, 11.95, 11.91, 11.78, 11.74, 11.8, 11.64, 11.48, 11.56]
                openPrice: [11.64, 11.69, 11.78, 11.8, 11.57, 11.7, 11.66, 11.62, 11.48, 11.32]
                preClosePrice: [11.63, 11.76, 11.82, 11.82, 11.6, 11.72, 11.67, 11.69, 11.48, 11.3]
        }
        600000.XSHG: {
                closePrice: [9.2, 9.29, 9.41, 9.39, 9.4, 9.23, 9.18, 9.12, 9.07, 9.16]
                turnoverValue: [724415940.0, 1025797630.0, 875061950.0, 618500160.0, 744481150.0, 717369980.0, 618488770.0, 601804100.0, 382309888.0, 436604864.0]
                turnoverVol: [78603480.0, 109976760.0, 93379976.0, 65935364.0, 79710264.0, 77606640.0, 67204744.0, 66047496.0, 42094792.0, 47596032.0]
                lowPrice: [9.11, 9.16, 9.24, 9.33, 9.25, 9.16, 9.17, 9.07, 9.04, 9.09]
                highPrice: [9.32, 9.44, 9.45, 9.47, 9.43, 9.4, 9.26, 9.17, 9.13, 9.26]
                openPrice: [9.14, 9.2, 9.28, 9.44, 9.4, 9.39, 9.22, 9.17, 9.1, 9.1]
                preClosePrice: [9.14, 9.2, 9.29, 9.41, 9.39, 9.4, 9.23, 9.18, 9.12, 9.07]
        }
        benchmark: {
                preCloseIndex: [2238.001, 2241.911, 2222.221, 2204.851, 2193.679, 2212.846, 2208.941, 2211.844, 2178.488, 2165.993]
                return: [0.00174709484044, -0.0087826858425 7, -0.00781650429908, -0.005067009 0632, 0.00873737679943, -0.00176469578091, 0.00131420440836, -0.0150806295562, -0.00573
                closeIndex: [2241.911, 2222.221, 2204.851, 2193.679, 2212.846, 2208.941, 2211.844, 2178.488, 2165.993, 2187.41]
        }
        tradeDate:
                [2014-01-08, 2014-01-09, 2014-01-10, 2014-01-13, 2014-01-14, 2014-01-15, 2014-01-16, 2014-01-17, 2014-01-20, 2014-01-21]
}
```

了解了历史数据的格式之后，我们就不难理解 handle_data() 当中的 hist[s][0] 了，其作用即是选取股票 s 历史数据窗口中第一个（也是最早的一个）收盘价。同理可以理解 hist[s][-1]。

注解: 在判断买入卖出时，条件中涉及到了另外一个有助于策略构建的重要工具，这就是 account.valid_secpos。从使用方式中可以看出，这是虚拟账户的一个属性。这个属性的数据结构为字典，键为证券代码，值为虚拟账户当前所持有该证券的数量。

由此即可以理解

```
if s not in account.valid_secpos:
```

的意思为：

```
如果当前虚拟账户的仓位中没有证券 s 的话...
```

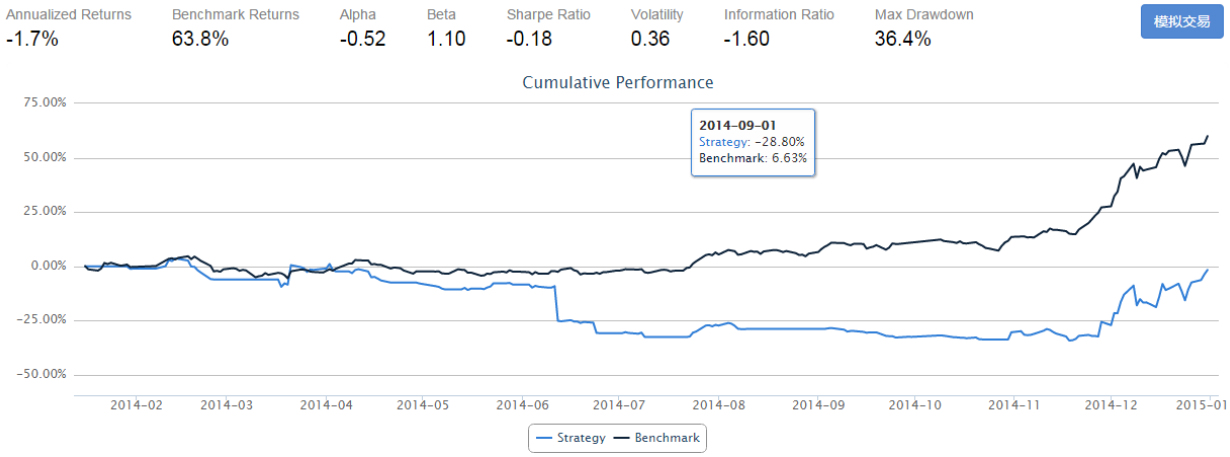
此外，虚拟账户还提供了另一个便捷的属性来获得当前的现金头寸，即 account.cash，其为数字，记录的是上一个交易日结束之后的虚拟账户剩余现金头寸

注解: 在示例代码中，除了之前出现过的 order() 之外，出现了另一种交易指令的下达方式order_to() 。与 order 类似，order_to(symbol, amount) 也有股票代码和数量两个参数；不同的地方在于，order_to() 的含义是进行交易使得虚拟账户中的该股票数量在 **这笔交易成功之后成为** amount。

至此，我们已经掌握了 Quartz 中构建策略的步骤和基本工具，现在可以充分发挥自己的聪明才智，开

始挖掘收益更高更稳定的量化投资策略了！

这一策略的表现如下：



至此，我们已经掌握了 Quartz 中构建策略的步骤和基本工具，现在可以充分发挥自己的聪明才智，开始挖掘收益更高更稳定的量化投资策略了！

返回目录

CHAPTER 8

快速回测简介

在阅读本部分之前，建议先阅读 Ten Minutes to Quartz 与 Ten Minutes to Quartz (UQER)，以对 Quartz 运行的框架、交易策略参数以及相关术语有一定的了解，这样可以更好地理解本部分的内容。

8.1 功能简介

在策略开发中，不可避免地会遇到参数优化的问题，即需要反复设定某些参数的值，反复运行策略，以获得最佳的参数组合。在这种情况下，如果回测所需的行情数据并不依赖于参数 (start, end, benchmark, universe, capital_base 保持不变)，那么可以使用 quick_backtest 功能，省去每次回测获取数据的过程，加速回测。

！注意：quick_backtest 必须在 code 环境下调用

！注意：quick_backtest 仅限日间策略

返回目录

8.2 代码示例

我们先来看一段例子：

```
# 可编辑部分与 strategy 模式一样，其余部分按本例代码编写即可

# ----------回测参数部分开始，可编辑----------

start = '2015-10-01'
end   = '2015-12-01'
```

```
benchmark = 'SH50'
universe = set_universe('SH50')
capital_base = 10000

# ----------回测参数部分结束----------

# 把回测参数封装到 SimulationParameters 中，供 quick_backtest 使用
sim_params = quartz.SimulationParameters(start, end, benchmark, universe, capital_base)

# 获取回测行情数据
idxmap, data = quartz.get_daily_data(sim_params)

# 运行结果
results = {}

# 调整参数快速回测
for x in range(1, 10):

    # ----------策略逻辑部分开始，可编辑----------

    # refresh_rate 的值和回测行情数据无关，所以也可以在参数调优中修改
    # 类似的还有 commission 和 slippage
    refresh_rate = 1
    # commission = ...
    # slippage = ...

    def initialize(account):
        account.my_paramter = x

    def handle_data(account):
        # 调用 account.my_parameter 进行计算或其他操作
        return

    # ----------策略逻辑部分结束----------

    # 把回测逻辑封装到 TradingStrategy 中，供 quick_backtest 使用
    strategy = quartz.TradingStrategy(initialize, handle_data)

    # 回测部分
    bt, acct = quartz.quick_backtest(sim_params, strategy, idxmap, data,
                                     refresh_rate = refresh_rate)
                                     # commission = commission,
                                     # slippage = slippage

    # 对于回测的结果，可以通过 perf_parse 函数计算风险指标
    perf = quartz.perf_parse(bt, acct)
```

```
    # 保存运行结果
    results[x] = perf['returns']

# 直接打印
# print results

# 转换为 DataFrame 并画图
# import pandas
# results = pandas.DataFrame(results)
# results.plot()
```

从代码中可以清晰地看出，这一过程分成了两大部分，其中第一部分是根据 start, end, benchmark, universe, capital_base 获取回测用的数据，并保存在 idxmap 和 data 两个变量当中，以供在不同的参数下回测时反复调用。

第二部分则是参数调优部分，需要定义的东西包括 referesh_rate, commission 等参数和 initialize()、handle_data() 两个决定回测逻辑的函数。在定义完成后，调用 quartz.quick_backtest，把相关参数和策略逻辑都输入，即可得到回测结果 bt。注意到这里 idxmap 和 data 也是输入的一部分，所以节省了获取数据的时间。

对于每一次的回测结果，可以直接打印信息来观察，也可以另外创建一个 list 或者 dict 把其中的信息存下来，之后在其他 cell 中观察比较。

返回目录

CHAPTER 9

日内回测简介

在阅读本部分之前，建议先阅读 Ten Minutes to Quartz，以对 Quartz 运行的框架、交易策略参数以及相关术语有一定的了解，这样可以更好地理解本部分的内容。

9.1 运行框架

Quartz 的日内策略回测框架使用了分钟线进行回测，使回测相比日间策略更加精细，允许用户定义更加丰富的策略细节，使交易策略更加完善。

Quartz 日内策略的运行框架与日间策略基本一致，都遵循着定义参数——获取数据——进行回测——分析结果的逻辑，其中主要的区别在数据获取方面。同时，由于数据的不同，在进行回测上也会有一些针对性的变动。

除了像日间策略一样获取日线数据之外，日内策略在每个交易日都会获取该日的分钟线数据，同时在每一分钟结束的时候（即分钟线数据刷新的时候）运行 handle_data，在下一分钟内进行交易指令的下达和模拟成交，更新所有数据（如 referencePrice、referencePortfolioValue、valid_secpos 等等），直到这一分钟的分钟线数据更新。

Quartz 日间策略在 handle_data 当中提供了交易日当日所有的历史分钟线数据、前一交易日之前的日线历史数据、昨日持仓、未成交订单等等日内交易所需要的信息，此外，在日间策略中支持限价指令。

返回目录

9.2 使用方法

日内回测和日间回测的大部分参数都一样，如起止时间、参考标准、证券池、起始资金等等，唯一的区别在于多了一个标识参数 **freq** 表明这个策略是日内的，如：

```
start = '2014-01-01'                    # 回测起始时间
end   = '2015-01-01'                    # 回测结束时间
benchmark = 'HS300'                     # 参考标准
universe  = ['000001.XSHE', '600000.XSHG']  # 证券池，支持股票和基金
capital_base = 100000                   # 起始资金
refresh_rate = 1                        # 调仓频率，即每 refresh_rate 条分钟线执行一次 handle_data() 函数

freq = 'm'                              # 日内交易

def initialize(account):
    return

def handle_data(account):
    return
```

其中的策略部分——initialize() 和 handle_data() 部分与日间策略几乎一样，只不过 account 会多出一些日内回测专用的属性，这些属性在下一部分会详细列出。

需要注意的是，如果选择在 code 模式下用过 quartz.backtest 函数手动回测，必须参数中加入 freq = freq 才能执行日内回测；不过 quartz.backtest 函数的输出仍是 bt 与 account，调用 quartz.perf_parse 的方法与日间回测一样。

另外，quartz.quick_backtest 不支持日内回测。

```
bt, acct = quartz.backtest(start=start, end=end, benchmark=benchmark,
                           universe=universe, capital_base=capital_base,
                           refresh_rate=refresh_rate, freq=freq)
perf = quartz.perf_parse(bt, acct)
```

对于 bt，其记录的投资组合价值变化仍然是按日的，相应地 perf 中的风险指标也是依据按日记录的 bt 而计算出来的。

返回目录

9.3 特殊属性

限价指令：

```
order('000001.XSHE', 1000, price = 6.5, otype = 'limit')

order('000001.XSHE', -1000, 6.8, 'limit')
```

```
order_to('000001.XSHE', 1000, price = 6.5, otype = 'limit')

order_to('000001.XSHE', 0, 6.8, 'limit')
```

时间控制：

```
# 值为字符串，如'09:45', '13:37' 等等
account.current_minute
```

日内历史窗口：

```
# 数据结构与日间获取的历史窗口数据一致

# range 表示当日历史分钟线的数量，注意同样的函数名称在日间回测时有不同的含义
account.get_symbol_history(symbol, range)

account.get_attribute_history(attribute, range)

account.get_history(range)

# range 表示前一交易日开始算的日线的数量（包含前一交易日），这些函数只有日内回测时才有
account.get_daily_symbol_history(symbol, range)

account.get_daily_attribute_history(attribute, range)

account.get_daily_history(range)
```

持仓与下单：

```
# 当前可卖持仓，数据结构与 valid_secpos 一致
account.avail_secpos

# 当前待成交指令，数据结构为列表，元素为 Order 实例，可以通过 Order 的属性访问相应数据，详见函数列表
account.pending_blotter
```

返回目录

CHAPTER 10

股票筛选器

股票筛选器是 Quartz 提供的一个便于用户动态维护股票池的工具，用户可以基于股票因子数据定义筛选条件，而后在回测中使用它们的结果。

在阅读本部分之前，建议先阅读 Ten Minutes to Quartz 与 Ten Minutes to Quartz (UQER)，以对 Quartz 运行的框架、交易策略参数以及相关术语有一定的了解，这样可以更好地理解本部分的内容。

- 构建筛选条件和筛选器
- 使用筛选器

10.1 构建筛选条件

对于单个因子筛选条件来说，可以通过以下方法进行定义：

```
# from quartz.api import *
# 在非 strategy 环境下最好加上上面这句

# Factor.<factor_name>.< 筛选方法 >，如：
Factor.PE.nlarge(10)
```

其中，共有两个地方需要输入：

1. <factor_name>，因子名，即该筛选条件是基于何种因子的。在股票筛选器中可用的所有因子可以下方式来获得：

```
StockScreener.available_factors()
```

2. < 筛选方法 >，即采用何种筛选方法来挑选股票。Quartz 目前提供了 5 中方法：value_range（值筛选），pct_range（百分比筛选），num_range（序号筛选），nlarge（最大），nsmall（最小），具体使用方法参看Factor。

通过以上方法，就可以很方便的定义出一条因子筛选条件了。然而在实际使用中，我们往往希望使用多个筛选条件综合作用，以获得更佳的效果。Quartz 在单因子筛选条件的基础上，也提供了多条筛选条件复合表达的功能。

在两个筛选条件之间，用户可以进行与（&）和或（|）两种运算，分别代表筛选出股票列表的交运算和并运算，学习过集合论基础知识的话应该能够很容易的理解。

对于三个或更多筛选条件，用户还可以加上括号来构成更为复杂的表达式，例如：

```
(Factor.PE.nlarge(100) | Factor.PB.pct_range(0.95, 1)) & Factor.RSI.value_range(70, 100)
```

在构建完筛选条件的表达式之后，我们可以构建股票筛选器了。构建方法十分简单：

```
StockScreener(< 筛选条件表达式 >)
```

在某些情况下，我们可能需要在筛选器以外自定义一些股票作为筛选器的补充，我们可以使用如下的语句来完成这一任务：

```
StockScreener(< 筛选条件表达式 >) + [自定义股票列表]
```

注意：为了避免性能上的问题，Quartz 将 StockScreener 中使用的筛选条件表达式中的单因子条件数量限制在了 5 条以内（含 5 条）。

返回目录

10.2 使用筛选器

上面我们学习了如何构建一个股票筛选器，那么如何在 Quartz 的回测框架中使用这个筛选器呢？

使用的方法非常简单，只需要把股票池定义成筛选器就可以了：

```
start = ...
end = ...
benchmark = ...
universe = StockScreener(< 筛选条件表达式 >) + [自定义股票列表]
...

def initialize(account):
    ...

def handle_data(account):
    account.universe
```

当使用筛选器定义股票池之后，在每个交易日的 handle_data 中，account.universe 即为前一个交易日通过股票筛选器筛选出来的股票列表，再除去当日不能交易的那些股票。

返回目录

CHAPTER 11

股票行业分类和指数成份股

为了方便使用，我们预设了行业分类和指数成份的实例。在开始研究环境中，可以直接在 set_universe 函数中调用这些实例来实现获取行业分类股票或指数成份股的列表，用于回测。例如：IndZJH.JinRongYeL1 表示证监会一级行业分类中的金融业；IndSW.DianLiL2 表示申万二级行业中的电力；IdxCN.Idx000300 和 IdxCN.IdxHuShen300_1 都表示上交所发布的沪深 300 指数成份股。

11.1 行业分类

如果要设置股票池 universe 为某个行业，可以通过以下方法进行：

```
# set_universe(< 行业分类实例 >, < 日期 >)，如：
set_universe(IndSW.YinHangL2, date='20160101')
```

如上将拿取 2016 年 1 月 1 日时的申万二级行业分类中的银行类股票列表，即 16 只银行股：

```
['000001.XSHE', '002142.XSHE', '600000.XSHG', '600015.XSHG',
 '600016.XSHG', '600036.XSHG', '601009.XSHG', '601166.XSHG',
 '601169.XSHG', '601288.XSHG', '601328.XSHG', '601398.XSHG',
 '601818.XSHG', '601939.XSHG', '601988.XSHG', '601998.XSHG']
```

行业分类实例支持以下内容：

1. IndSW，申万行业
    - 包含：IndSW.CaiJueL1（采掘）等 33 个申万一级行业分类，后缀为"L1"；IndSW.BanDaoTiL2（半导体）等 111 个申万二级行业分类，后缀为"L2"；IndSW.BaiJiuL3（白酒）等 238 个申万三级行业分类，后缀为"L3"；
    - 同时包含实例函数：IndSW.L1()、IndSW.L2() 和 IndSW.L3()，分别返回申万行业分类三个级别所有行业分类说明，返回值为字典（其中键为行业分类实例，值为行业说明）。

- 支持代码补全
- 行业实例命名规范为：行业简称拼音 + 分级后缀

2. IndZZ，中证行业
   - 包含：IndZZ.GongYeL1（工业）等 10 个中证一级行业分类，后缀为"L1"；IndZZ.FangDiChanL2（房地产）等 25 个中证二级行业分类，后缀为"L2"；IndZZ.HangKongGongSiL3（航空公司）等 61 个中证三级行业分类，后缀为"L3"；
   - 同时包含实例函数：IndZZ.L1()、IndZZ.L2() 和 IndZZ.L3()，分别返回中证行业分类三个级别所有行业分类说明，返回值为字典（其中键为行业分类实例，值为行业说明）。
   - 支持代码补全
   - 行业实例命名规范为：行业简称拼音 + 分级后缀
3. IndZJH，证监会行业
   - 包含：IndZJH.JinRongYeL1（金融业）等 18 个证监会一级行业分类，后缀为"L1"；IndZJH.CanYinYeL2（餐饮业）等 74 个证监会二级行业分类，后缀为"L2"；
   - 同时包含实例函数：IndZJH.L1() 和 IndZJH.L2()，分别返回中证行业分类两个级别所有行业分类说明，返回值为字典（其中键为行业分类实例，值为行业说明）。
   - 支持代码补全
   - 行业实例命名规范为：行业简称拼音 + 分级后缀

日期支持以下格式：

- 'YYYYMMDD'，例如 '20160101'
- 'YYYY-MM-DD'，例如 '2016-01-01'
- datetime(year, month, day), 例如 datetime(2016, 1, 1)

注：日期可以不填，默认表示最新数据

返回目录

11.2 指数成分

如果要设置股票池 universe 为某个指数成份股，可以通过以下方法进行：

```
# set_universe(< 指数成分实例 >, < 日期 >)，如：
set_universe(IdxCN.IdxShangZhengZongZhi, date='20160101')
```

如上将拿取 2016 年 1 月 1 日时的上证综指成分股票列表，即：

```
['600000.XSHG','600004.XSHG','600005.XSHG','600006.XSHG',...]
```

指数成分实例支持以下内容：

1. IdxCN，国内指数

    - 包含：IndCN.Idx000300（沪深 300）和 IdxCN.IdxHuShen300_1（沪深 300）等 546 个国内 A 股指数，前缀均为"Idx"；由于有些指数同时在深交所和上交所挂牌，所以会存在类似 IdxCN.IdxHuShen300_1 和 IdxCN.IdxHuShen300_2 同时表示'沪深 300'指数成份股的情况，后缀分别为"_1"（上交所）和"_2"（深交所），实际上它们表示同一指数；
    - 同时包含实例函数 IndSW.All()，返回 IdxCN 支持的所有指数说明，返回值为字典：其中键为指数名称，值为一个字典 (其中 secID 为指数在通联 DataAPI 中的 secID, shortName 为指数的中文信息)。
    - 支持代码补全
    - 指数成分实例命名规范有两套并行体系：1. "Idx"+ 指数编码，例如 IndCN.Idx000001 上证综指；2. "Idx"+ 指数简称，例如 IdxCN.IdxShangZhengZongZhi 同样表示上证综指
    - 使用原方法得到行业成份股的结果与此方法一致，即 set_universe('HS300')，set_universe('000300.ZICN')，set_universe(IndCN.Idx000300) 和 set_universe(IdxCN.IdxHuShen300_1) 结果一致

    日期支持以下格式：

    - 'YYYYMMDD'，例如 '20160101'
    - 'YYYY-MM-DD'，例如 '2016-01-01'
    - datetime(year, month, day), 例如 datetime(2016, 1, 1)

    注：日期可以不填，默认表示最新数据

返回目录

CHAPTER 12

Quartz 函数列表

12.1 api

set_universe(symbol = None, date = None)

获取预设股票代码列表

参数

- symbol (str) – 预设股票代码列表的名字，Quartz 支持以下七个预设列表：上证 50(SH50)、上证 180(SH180)、沪深 300(HS300)、中证 500(ZZ500)、创业板 (CYB)、中小板 (ZXB)、全 A 股 (A)；同时还支持自定义指数代码，可以获取对应指数的最新成分股列表，如"000001.ZICN"
- date (str/datetime) – 预设证券代码列表日期，仅在上证 50、上证 180、沪深 300、中证 500 或自定义指数代码时有效，表示获取该日的该指数成分股；默认为最近一个交易日

返回 预设股票代码列表

返回类型 list

order(symbol, amount, price = 0., otype = 'market')

在 handle_data(account, data) 中使用，向Account 实例 中的 account.blotter 属性添加Order 实例 ；指令含义为买入（卖出）数量为 amount 的股票 symbol

参数

- symbol (str) – 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
- amount (int) – 需要交易的证券代码为 symbol 的证券数量，为正则为买入，为负则为卖出；程序会自动对 amount 向下取整到最近的整百
- price (float) – 交易指令限价（仅日内策略可用）
- otype (str) – 交易指令类型（为 limit 时仅日内策略可用）

返回 无

order_to(symbol, amount, price = 0., otype = 'market')

在 handle_data(account, data) 中使用，向Account 实例 中的 account.blotter 属性添加Order 实例 ；指令含义为买入（卖出）一定量的股票使得股票 symbol 交易后的数量为 amount

参数

- symbol (str) – 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
- amount (int) – 需要交易的证券代码为 symbol 的证券数量，为正则为买入，为负则为卖出；程序会自动对 amount 向下取整到最近的整百
- price (float) – 交易指令限价（仅日内策略可用）
- otype (str) – 交易指令类型（为 limit 时仅日内策略可用）

返回 无

order_pct(self, symbol, pct, price=0., otype='market')

在 handle_data(account, data) 中使用，向 Account 实例中的 account.blotter 属性添加 Order 实例；指令含义为买入（卖出）价值为虚拟账户当前总价值的 pct 百分比的的证券 symbol，仅限市价单

参数

- symbol (str) – 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
- pct (float) – 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 -1 ~ 1，为正则为买入，为负则为卖出；程序会自动对 amount 向下取整到最近的整百
- price (float) – 交易指令限价（仅日内策略可用）
- otype (str) – 交易指令类型（为 limit 时仅日内策略可用）

返回 无

order_pct_to(self, symbol, pct, price=0., otype='market')

在 handle_data(account, data) 中使用，向 Account 实例中的 account.blotter 属性添加 Order 实例；指令含义为买入（卖出）证券 symbol 使得其价值为虚拟账户当前总价值的 pct 百分比，仅限市价单

参数

- symbol (str) – 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
- pct (float) – 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 -1 ~ 1，为正则为买入，为负则为卖出；程序会自动对 amount 向下取整到最近的整百
- price (float) – 交易指令限价（仅日内策略可用）
- otype (str) – 交易指令类型（为 limit 时仅日内策略可用）

返回 无

observe(name, value)

在 handle_data(account, data) 中使用，仅限日间策略。在 backtest 的输出的 pandas.DataFrame 中增加一列自定义需要观测的变量。如果 refresh_rate > 1，那么期间不进行交易时的 name 依然保留上一次调用 handle_date 时候的 value

参数

- name (str) – 需要观测的变量名称
- value – 需要观测的变量值

class Logger(object)

日志记录，在 initialize() 和 handle_data() 中可以以 log.<method name> 的方法调用，具体方法列表见下文

info(self, message)

输出日志信息，类型为 [INFO]。仅能在 initialize() 和 handle_data() 中使用。

参数 message (object/str) – 需要输出的信息，可以是字符串或其他任意可以被 print 语句执行的类

返回 无

debug(self, message)

输出日志信息，类型为 [DEBUG]。仅能在 initialize() 和 handle_data() 中使用。

参数 message (object/str) – 需要输出的信息，可以是字符串或其他任意可以被 print 语句执行的类

返回 无

warn(self, message)

输出日志信息，类型为 [WARN]。仅能在 initialize() 和 handle_data() 中使用。

参数 message (object/str) – 需要输出的信息，可以是字符串或其他任意可以被 print 语句执行的类

返回 无

error(self, message)

输出日志信息，类型为 [ERROR]。仅能在 initialize() 和 handle_data() 中使用。

参数 message (object/str) – 需要输出的信息，可以是字符串或其他任意可以被 print 语句执行的类

返回 无

class Commission(object)

手续费标准，包含如下属性。方法等详情见这里

- self.buycost：买进手续费
- self.sellcost：卖出手续费
- self.unit：手续费单位

class Slippage(object)

滑点标准，包含如下属性。方法等详情见这里

- self.value：滑点值
- self.unit：滑点单位

class Factor(object)

多个因子筛选条件，限制最多 5 条，可以使用 &、|、括号等组成表达式，表达因子之间的逻辑关系，例如：

(Factor.PE.valueRange(None, 100) & _ Factor.<>.pctRange(80%, None)) | _

(Factor.<>.nlarge(100) & _ Factor.<>.nsmall(100))

value_range(self, lbound, ubound)

筛选因子值处于上下界之间的证券（包含两端）

参数

- lbound (float) – 条件参数下界
- ubound (float) – 条件参数上界

返回 自身

返回类型 Factor

pct_range(self, lbound, ubound)

筛选因子值处于百分比上下分位点之间的证券（包含两端），默认升序排列

参数

- lbound (float) – 条件参数下界
- ubound (float) – 条件参数上界

返回 自身

返回类型 Factor

num_range(self, lbound, ubound)

筛选因子值处于上下界序号之间的证券（包含两端），默认升序排列

参数

- lbound (int) – 条件参数下界
- ubound (int) – 条件参数上界

返回 自身

返回类型 Factor

nlarge(self, n)

筛选因子值最大的 n 只证券

参数 n (int) – 条件参数

返回 自身

返回类型 Factor

nsmall(self, n)

筛选因子值最小的 n 只证券

参数 n (int) – 条件参数

返回 自身

返回类型 Factor

class StockScreener(object)

股票筛选器，利用因子筛选条件的表达式生成，如使用，则 Account.universe 为每个交易日的筛选结果，Account.screenerData 为每个交易日的筛选器中需记录值的因子的数据

___init___(self, fct)

初始化

参数 fct (Factor) – 因子筛选条件

返回 无

available_factors(self)

返回可用因子列表

返回 可用因子列表

返回类型 list

12.2 backtest

backtest(start = '2013-01-01', end = '2014-01-01', benchmark = 'HS300', universe = ['000001.XSHE'], capital_base = 100000, initialize = None, handle_data = None, security_base = {}, commission = Commission(), slippage = Slippage(), refresh_rate = 1, freq = 'd')

对交易策略进行回测

参数

- start (str/datetime) – 回测开始日期
- end (str/datetime) – 回测结束日期
- benchmark (str) – 策略对照指数名称, Quartz 支持以下五个指数：上证综指 (SHCI)、上证 50(SH50)、上证 180(SH180)、沪深 300(HS300) 和中证 500(ZZ500)；此外，Quartz 还支持用户自定义的证券代码或指数代码，以获得个性化的对照标准 (代码需带后缀，如 '600000.XSHG', '399006.ZICN' 等等)
- universe (list) – 证券池，由证券代码构成，证券代码必须包含后缀，其中上证证券为.XSHG，深证证券为.XSHE
- capital_base (int) – 初始资金

- initialize (function) – 交易策略-虚拟账户初始函数
- handle_data (function) – 交易策略-每日交易指令判断函数
- security_base (dict) – 初始证券头寸
- commission (Commission) – 手续费标准
- slippage (Slippage) – 滑点标准
- refresh_rate (int) – 调仓间隔，即每隔多少个交易日执行一次 handle_data()
- freq (str) – 回测频率，'d' 为日，'m' 为分钟

返回 回测报告，回测数据。其中回测报告包括日期、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，时间从开始日期及需要获取的最长历史窗口后开始；回测数据以虚拟账户 Account 的方式输出，供 perf_parse 使用

返回类型 pandas.DataFrame, Account

quick_backtest(sim_params, strategy, idxmap_all, data_all, commission = Commission(), slippage = Slippage(), refresh_rate = 1)

在回测参数和行情数据已经加载完毕的情况下进行快速回测，仅限日间策略

参数

- sim_params (SimulationParameters) – 回测参数
- strg (TradingStrategy) – 交易策略
- idxmap_all (dict) – 所有序号映射
- data_all (list) – 所有行情数据
- commission (Commission) – 手续费标准
- slippage (Slippage) – 滑点标准
- refresh_rate (int) – 调仓间隔，即每隔多少个交易日执行一次 handle_data()

返回 回测报告，回测数据。其中回测报告包括日期、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，时间从开始日期及需要获取的最长历史窗口后开始；回测数据以虚拟账户 Account 的方式输出，供 perf_parse 使用

返回类型 pandas.DataFrame, Account

12.3 sim_condition

12.3.1 strategy

class TradingStrategy(object)

交易策略，包含如下属性

- self._initialize：交易策略-虚拟账户初始函数

- self._handle_data：交易策略-每日交易指令判断函数

___init___(self, initialize=None, handle_data=None)

初始化

参数

- initialize (function) – 交易策略-虚拟账户初始函数
- handle_data (function) – 交易策略-每日交易指令判断函数

返回 无

12.3.2 data_generator

get_daily_data(sim_params)

整合日间回测所需要的各种数据

参数 sim_params (SimulationParameters) – 回测参数

返回 所有序号映射，所有行情数据。所有序号映射包括交易日、证券列表和 benchmark、股票属性、benchmark 属性四个不同的序号映射；所有行情数据的最外层是列表，元素是证券和对照标准的行情数据，格式为二维数组，一是属性，二是时间（交易日）

返回类型 dict, list

get_daily_data_generator(data)

日间行情数据生成器，每调用一次生成一个交易日的行情数据

参数 data (list) – 所有行情

返回 单日行情，格式和 data 一样，只是最里层从时间序列变为属性某日的具体值

返回类型 list

get_intraday_data(sim_params, date)

整合日内回测所需要的各种数据

参数

- sim_params (SimulationParameters) – 回测参数
- date (datetime) – 交易日日期

返回 所有序号映射，所有行情数据。所有序号映射包括交易日、证券列表和 benchmark、证券属性、benchmark 属性四个不同的序号映射；所有行情数据的最外层是列表，元素是证券和对照标准的行情数据，格式为二维数组，一是属性，二是时间（交易日）

返回类型 dict, list

get_intraday_data_generator(data)

日内行情数据生成器，每调用一次生成一条分钟线的行情数据

参数 data (list) – 所有行情

返回 单分钟线行情，格式和 data 一样，只是最里层从时间序列变为属性某日的具体值

返回类型 list

12.3.3 env

class SimulationParameters(object)

回测参数，包含如下属性

- self.trading_days：回测期间的所有交易日列表
- self.minute_bars：分钟线列表
- self.first_trading_day：回测期间首个交易日
- self.last_trading_day：回测期间最后交易日
- self.benchmark：策略参照标准
- self.universe：证券池
- self.screener：证券筛选器
- self.capital_base：虚拟账户初始资金
- self.security_base：虚拟账户初始证券头寸

___init___(self, start = None, end = None, benchmark = None, universe = None, capital_base = DEFAULT_CAPITAL_BASE, security_base = {})

初始化，并根据 start 和 end 生成回测区间内的交易日列表 self.trading_days，以及相关参数 self.first_trading_day 和 self.last_trading_day

参数

- start (str/datetime) – 回测开始日期
- end (str/datetime) – 回测结束日期
- benchmark (str) – 策略对照指数名称，Quartz 支持以下五个指数：上证综指 (SHCI)、上证 50(SH50)、上证 180(SH180)、沪深 300(HS300) 和中证 500(ZZ500)；此外还支持用户自定义的证券代码或指数代码，以获得个性化的对照标准（代码需带后缀，如 '600000.XSHG', '399006.ZICN' 等等）
- universe (list) – 证券池，由证券代码构成，证券代码必须包含后缀，其中上证证券为.XSHG，深证证券为.XSHE
- capital_base (int) – 初始资金
- security_base (dict) – 初始证券头寸

class Account(object)

虚拟账户，包含如下属性

- self.sim_params：回测参数

- self.universe_all：所有证券池
- self.universe：根据每个交易日的数据，剔除了数据缺失和数据异常的证券的证券池
- self.current_date：当前回测日期
- self.days_counter：交易日计数
- self.trading_days：回测期间的所有交易日列表
- self.position：现金及证券头寸
- self.cash：现金头寸，浮点数
- self.secpos：证券头寸，字典，键为证券代码，值为头寸
- self.valid_secpos：有效证券头寸，字典，键为证券代码，值为头寸
- self.avail_secpos：可卖证券头寸，字典，键为证券代码，值为头寸
- self.referencePrice：参考价，一般使用的是上一日收盘价或者上一分钟收盘价
- self.referenceReturn：参考收益率，一般使用的是上一日收益率
- self.referencePortfolioValue：参考投资策略价值，使用参考价计算
- self.screenerData：证券筛选器中需要记录值的因子数据
- self.blotter：下单指令列表
- self.commission：手续费标准
- self.slippage：滑点标准
- self.current_minute：当前回测分钟线（仅对日内策略有效）
- self.pending_blotter：所有未成交指令（仅对日内策略有效）

___init___(self, sim_params = None, strg = None, idxmap_all = None, data_all = None, commission = Commission(), slippage = Slippage())

初始化；除输入参数外初始化 self.current_date 为回测区间内的第一个交易日，初始化交易指令簿 self.blotter 为空列表；最后执行 strg.initialize()，根据需要生成历史数据对象

参数

- sim_params (SimulationParameters) – 回测参数
- strg (TradingStrategy) – 交易策略
- idxmap_all (dict) – 所有序号映射
- data_all (list) – 所有行情数据
- commission (Commission) – 手续费标准
- slippage (Slippage) – 滑点标准

handle_data(self, data)

执行 strg.handle_data(self, data)，另外执行 transact(self, data) 判断 self.blotter 中的交易指令的完成情况，更新 self.blotter 和 self.position

参数 data (list) – 所有行情数据的最外层是列表，元素是股票和对照标准的行情数据，格式为二维数组。

_get_daily_symbol_history(self, symbol, time_range)

获取单只证券的日线历史数据，日间回测时通过 get_symbol_history 调用，日内回测时通过 get_daily_symbol_history 调用

参数

- symbol (str) – 证券代码或 benchmark 或 tradeDate
- time_range (int) – 历史数据窗口长度

返回 若 symbol 为 tradeDate，则输出前 time_range 长度的交易日列表；此外，输出历史数据窗口，数据结构为字典，键为相关属性，值为前 time_range 长度的交易日的该属性的值的列表

返回类型 list or dict

_get_daily_attribute_history(self, attribute, time_range)

获取单只证券属性的日线历史数据，日间回测时通过 get_attribute_history 调用，日内回测时通过 get_daily_attribute_history 调用

参数

- attribute (str) – 证券属性名称
- time_range (int) – 历史数据窗口长度

返回 前 time_range 长度交易日的某属性的历史数据窗口，数据结构为字典，键为证券代码，值为前 time_range 长度的交易日的该属性的值的列表

返回类型 dict

_get_daily_history(self, time_range)

获取日线历史数据，日间回测时通过 get_history 调用，日内回测时通过 get_daily_history 调用

参数 time_range (int) – 历史数据窗口长度

返回 前 time_range 长度交易日的历史数据窗口，数据结构为字典，键为证券代码、benchmark、tradeDate，值为字典，其键为相关属性，其值为属性值列表

返回类型 dict

_get_intraday_symbol_history(self, symbol, time_range)

获取单只证券的分钟线历史数据，日内回测时通过 get_symbol_history 调用

参数

- symbol (str) – 证券代码或 benchmark 或 tradeDate
- time_range (int) – 历史数据窗口长度

返回 若 symbol 为 minuteBar，则输出前 time_range 长度的分钟列表；此外，输出历史数据窗口，数据结构为字典，键为相关属性，值为前 time_range 长度分钟的该属性的值的列表

返回类型 list or dict

_get_intraday_attribute_history(self, attribute, time_range)

获取单只证券属性的分钟线历史数据，日内回测时通过 get_attribute_history 调用

参数

- attribute (str) – 证券属性名称
- time_range (int) – 历史数据窗口长度

返回 前 time_range 长度分钟的某属性的历史数据窗口，数据结构为字典，键为证券代码，值为前 time_range 长度的分钟的该属性的值的列表

返回类型 dict

_get_intraday_history(self, time_range)

获取分钟线历史数据，日内回测时通过 get_history 调用

参数 time_range (int) – 历史数据窗口长度

返回 前 time_range 长度分钟的历史数据窗口，数据结构为字典，键为证券代码、benchmark、tradeDate，值为字典，其键为相关属性，其值为属性值列表

返回类型 dict

_referencePrice(self)

获得参考价格，如果是日间或日内首分钟则为前收盘，日内非首分钟则为上一分钟的收盘

返回 self.universe 中所有股票的参考价格

返回类型 dict

_referenceReturn(self)

获得参考收益率，如果是日间或日内首分钟则为前一日的收益率，日内非首分钟则为上一分钟的收益率

返回 self.universe 中所有股票的参考收益率

返回类型 dict

_referencePorfolioValue(self)

获得参考投资组合价值，即现金价值与证券价值之和，其中证券价值用 self._referencePrice() 计算

返回 当前投资组合参考价值

返回类型 float

12.4 trade

12.4.1 cost

class Commission(object)

手续费标准，包含如下属性

- self.buycost：买进手续费
- self.sellcost：卖出手续费
- self.unit：手续费单位

___init___(self, buycost=0.001, sellcost=0.002, unit="perValue")

初始化买单和卖单的成本，含义为成交额的百分比

参数

- buycost (float) – 买进手续费
- sellcost (float) – 卖出手续费
- unit (str) – 手续费单位，可选值 ['perValue', 'perShare']

返回 无

calculate(self, price, direction)

计算考虑手续费之后的实际成交时的股票价格

参数

- price (float) – 成交价
- direction (int) – 交易方向，1 为买入，-1 为卖出

返回 加上手续费的成交价

返回类型 float

class Slippage(object)

滑点标准，包含如下属性

- self.value：滑点值
- self.unit：滑点单位

___init___(self, value=0, unit="perValue")

初始化滑点的值和单位

参数

- value (float) – 滑点值
- unit (str) – 滑点单位，可选值 ['perValue', 'perShare']

返回 无

calculate(self, price, direction)

计算考虑滑点之后的实际成交时的证券价格

参数 price (float) – 成交价

返回 加上滑点的成交价

返回类型 float

12.4.2 order

class Order(object)

交易指令，包含如下属性

- self.order_time：指令下达时间
- self.symbol：指令涉及的证券代码
- self.direction：指令方向，正为买入，负为卖出
- self.amount：指令交易数量
- self.type：指令种类，如'market' 表示按市价成交，'limit' 表示按限价成交
- self.filled_time：指令成交时间
- self.filled_amount：指令成交数量
- self.slippage: 指令成交滑点
- self.commission: 指令成交手续费
- self.transact_price：指令成交价格（包含滑点）
- self.otype: 指令种类
- self.price: 指令限价
- self.state: 指令状态，分为 Normal, NotIncluded, Suspended, UpLimit 和 DownLimit 五种，分别指代正常、未包含在股票池中、停牌、涨停和跌停

___init___(self, symbol, amount, time = None, otype = 'market', price = 0)

初始化；除时间和股票代码之外，会将 amount 处理成方向和绝对数量两部分，方向即为符号；此外，还会初始化已成交量和成交价为 0，供 transact() 函数修改

参数

- symbol (str) – 证券代码, 必须包含后缀, 其中上证证券为.XSHG, 深证证券为.XSHE
- amount (int) – 交易数量，符号表示交易方向，正为买入，负为卖出
- time (datetime) – 指令下达时间
- otype (str) – 指令种类

- price (float) – 指令限价

返回 无

12.4.3 transaction

class Position(object)

账户头寸，包含现金头寸和证券头寸两部分，包含如下属性

- self.cash：现金数量
- self.secpos：证券头寸，数据结构为字典，键为证券代码，值为所持有的证券数量
- self.seccost：证券成本，数据结构为字典，键为证券代码，值为价格
- self.securities：目前持有的证券代码列表
- self.valid_secpos：目前持有的证券仓位

___init___(self, cash=DEFAULT_CAPITAL_BASE, secpos={}, seccost={})

初始化

参数

- cash (float) – 现金数量
- secpos (dict) – 记录各证券的持有数量，键为证券代码，值为所持有的证券数量
- seccost (dict) – 记录各证券的成交价格或期望成交价格，键为证券代码，值为价格

返回 无

evaluate(self, idxmap, data)

计算按当前行情数据收盘时现金头寸和证券头寸的价值

参数

- idxmap (dict) – 所有序号映射
- data (list) – 所有行情数据

返回 现金头寸 + 按收盘价计算的证券头寸

返回类型 float

transact(account, data)

根据行情数据和虚拟账户的交易指令簿判断各交易指令成交情况（日间），并更新虚拟账户的头寸和交易指令簿

参数

- account (Account) – 虚拟账户
- data (list) – 当日或当时的行情数据

返回 无

transact_intraday(account, minute, data):
"""
根据行情数据和虚拟账户的交易指令簿判断各交易指令成交情况（日内），并更新虚拟账户的头寸和交易指令簿

参数

- account (Account) – 虚拟账户
- data (list) – 所有行情数据，quartz.backtest 输出的一部分

返回 无

"""

12.5 performance

12.5.1 journal

class Report(object)

回测记录，包含如下属性

- self.keys：供记录的指标名称
- self.date：记录日期列表
- self.blotter：记录交易指令列表
- self.cash：记录现金列表
- self.security_position：记录证券头寸列表
- self.portfolio_value：记录投资组合价值列表
- self.bm_return：记录参照标准收益率列表

___init___(self)

初始化，共有六项指标：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

返回 无

update(self, account, data)

更新相应指标的数据

参数

- account (Account) – 虚拟账户
- data (list) – 当前交易日的行情数据。数据结构为二维列表，第一维是证券代码和 benchmark，第二维是对应的行情属性的值

返回 无

output(self)

输出成 pandas.DataFrame 格式

返回 编辑好格式的回测记录

返回类型 pandas.DataFrame

12.5.2 perf_parse

perf_parse(backtest_result, account)

根据回测记录计算各项风险收益指标，指标内容见下表：

指标名称	指标含义	计算公式	数据格式
returns	虚拟账户每日收益	虚拟账户当日价值/虚拟账户前一日价值-1	numpy.array
annualized_return	虚拟账户年化收益率	(虚拟账户最终价值/虚拟账户初始价值-1)/回测交易日数量*250	float
volatility	虚拟账户收益波动率	虚拟账户每日收益的年化标准差	float
cumulative_values	虚拟账户初始价值为1时的累计价值	虚拟账户当日价值/虚拟账户初始价值	numpy.array
cumulative_returns	虚拟账户累计收益率	虚拟账户当日价值/虚拟账户初始价值-1	numpy.array
benchmark_returns	参照标准每日收益	参照标准当日指数/参照标准前一日指数-1	numpy.array
benchmark_annualized_return	参照标准每日收益	(参照标准最终指数/参照标准初始指数-1)/回测交易日数量*250	float
benchmark_volatility	参照标准收益波动率	参照标准每日收益的年化标准差	float
benchmark_cumulative_values	参照标准初始价值为1时的累计价值	参照标准当日指数/参照标准初始指数	numpy.array
benchmark_cumulative_returns	参照标准累计收益率	参照标准当日指数/参照标准初始指数-1	numpy.array
treasury_returns	每日无风险利率	中国固定利率国债收益率曲线上10年期国债的年化到期收益率	numpy.array
sharpe	Sharpe比率	(虚拟账户年化收益率 - 回测起始交易日的无风险利率)/虚拟账户收益波动率	float
information_ratio	信息比率	(虚拟账户每日收益 - 参照标准每日收益)的年化均值除以年化标准差	float
information_coefficient	信息相关系数	信息比率/(股票数量 * 回测交易日数量)	float
beta	贝塔	虚拟账户每日收益与参照标准每日收益的协方差/参照标准每日收益的方差	float
alpha	阿尔法	(虚拟账户年化收益 - 无风险收益) - 贝塔 * (参照标准年化收益 - 无风险收益)	float
excess_return	相比于无风险利率的超额收益	虚拟账户年化收益 - 无风险收益	float
max_drawdown	最大回撤	max(1-虚拟账户当日价值/当日之前虚拟账户最高价值)	float
turnover_rate	换手率	买入总价值与卖出总价值孰低者/虚拟账户平均价值	float

参数

- backtest_result (pandas.DataFrame) – 回测记录，由backtest() 生成
- accout (Account) – 包含回测数据的虚拟账户，由backtest() 生成

返回 各类风险指标。键是风险指标名称，值为相应的值

返回类型 dict

12.5.3 risk_metrics

getReturn(values)
根据价值计算收益率

参数 values (list) – 价值列表

返回 收益率列表

返回类型 list

getCumulativeValue(returns)
根据收益率计算累计价值

参数 returns (list) – 收益率列表

返回 累计价值列表

返回类型 list

getAnnualizedReturn(c_values)
根据累计价值计算年化收益率

参数 c_values (list) – 累计价值列表

返回 年化收益率

返回类型 float

getMaxDrawdown(c_values)
根据累计价值计算最大回撤

参数 c_values (list) – 累计价值列表

返回 最大回撤

返回类型 float

getRiskFreeRate(date)
获得无风险利率

参数 date (datetime) – 日期

返回 该日期开始第一个可用的无风险利率

返回类型 float

getCAPM(st_returns, bm_returns, rf)
计算 alpha 和 beta

参数

- st_returns (list) – 策略收益率列表
- bm_returns (list) – 市场收益率列表
- rf (float) – 无风险收益率

返回 alpha, beta

返回类型 float, float

getInformationRatio(st_returns, bm_returns)

计算信息比率

参数

- st_returns (list) – 策略收益率列表
- bm_returns (list) – 市场收益率列表

返回 信息比率

返回类型 float

getTurnoverRate(bt, idxmap, data)

计算换手率

参数

- backtest_result (pandas.DataFrame) – 回测结果，quartz.backtest 输出的一部分
- idxmap (dict) – 所有序号映射，quartz.backtest 输出的一部分
- data (list) – 所有行情数据，quartz.backtest 输出的一部分

返回 换手率

返回类型 float

CHAPTER 13

交易策略示例

13.1 Halloween Cycle

13.1.1 策略思路

- “万圣节效应”: 每年 10 月到次年 5 月，股票市场会出现上涨的趋势

13.1.2 策略实现

- 股票池：流动性充足的 10 只个股，包括工商银行、中国石化等
- 每年 10 月，将账户中现金平均分成 10 份，分别买入相应的 10 只个股，满仓；次年 5 月全部抛出，空仓

13.1.3 策略代码

```
start = '2010-05-01'
end   = '2014-05-01'
benchmark = 'HS300'
universe  = ['601398.XSHG', '600028.XSHG', '601988.XSHG', '600036.XSHG', '600030.XSHG',
             '601318.XSHG', '600000.XSHG', '600019.XSHG', '600519.XSHG', '601166.XSHG']
capital_base = 1000000

def initialize(account):
    pass

def handle_data(account):
    for stock in account.universe:
        today = account.current_date
        if stock not in account.valid_secpos and today.month == 10:
            amount = account.cash / len(account.universe) / account.referencePrice[stock]
            order(stock, amount)
```

```
    elif stock in account.valid_secpos and today.month == 5:
        order_to(stock, 0)
```

13.2 Momentum/Contrarian

13.2.1 策略思路

- Momentum：业绩好的股票会继续保持其上涨的势头，业绩差的股票会保持其下跌的势头
- Contrarian：股票在经过一段时间的上涨之后会出现回落，一段时间的下跌之后会出现反弹

13.2.2 策略实现

- Momentum：每次调仓将股票按照前一段时间的累计收益率排序并分组，买入历史累计收益 **最高**的那一组
- Contrarian：每次调仓将股票按照前一段时间的累计收益率排序并分组，买入历史累计收益 **最低**的那一组

13.2.3 策略代码

Momentum

```
import pandas as pd

start = '2011-01-01'
end   = '2015-03-01'
benchmark = 'SH50'
universe = set_universe('SH50')
capital_base = 100000
refresh_rate = 10

def initialize(account):
    account.amount = 300

def handle_data(account):
    history = account.get_attribute_history('closePrice', 20)

    momentum = {'symbol':[], 'c_ret':[]}
    for stk in account.universe:
        if stk not in history:
            continue

        momentum['symbol'].append(stk)
```

```
        momentum['c_ret'].append(history[stk][-1]/history[stk][0])
    momentum = pd.DataFrame(momentum).sort(columns='c_ret').reset_index()
    momentum = momentum[len(momentum)*4/5:len(momentum)]
    buylist = momentum['symbol'].tolist()
    for stk in account.valid_secpos:
        if stk not in buylist:
            order_to(stk, 0)
    for stk in buylist:
        if stk not in account.valid_secpos:
            order_to(stk, account.amount)
```

Contrarian

```
import pandas as pd

start = '2011-01-01'
end   = '2015-03-01'
benchmark = 'SH50'
universe = set_universe('SH50')
capital_base = 100000
refresh_rate = 10

def initialize(account):
    account.amount = 300

def handle_data(account):
    history = account.get_attribute_history('closePrice', 20)

    contrarian = {'symbol':[], 'c_ret':[]}
    for stk in account.universe:
        if stk not in history:
            continue

        contrarian['symbol'].append(stk)
        contrarian['c_ret'].append(history[stk][-1]/history[stk][0])
    contrarian = pd.DataFrame(contrarian).sort(columns='c_ret').reset_index()
    contrarian = contrarian[:len(contrarian)/5]
    buylist = contrarian['symbol'].tolist()
    for stk in account.valid_secpos:
        if stk not in buylist:
            order_to(stk, 0)
    for stk in buylist:
        if stk not in account.valid_secpos:
            order_to(stk, account.amount)
```

13.3 Global Minimum Variance Portfolio (GMVP)

13.3.1 策略思路

- 对于一系列协方差矩阵为 $\mathbf{\Sigma}$ 的股票而言，确定权重 $\boldsymbol{w} = (w_1, w_2, ..., w_n)^T$ 使得整个投资组合的方差最小，即求解如下最优化问题：

$$\begin{aligned} min \quad & \boldsymbol{w^T \Sigma w} \\ s.t. \quad & \boldsymbol{1^T w} = 1 \end{aligned}$$

- 此最优化问题是一凸规划问题，可通过 Lagrange multiplier 方法求出最优解：

$$\boldsymbol{w^*} = \frac{\boldsymbol{\Sigma^{-1} 1}}{\boldsymbol{1^T \Sigma^{-1} 1}}$$

13.3.2 策略实现

- 在每个调仓日，根据回看窗口计算出 $\hat{\mathbf{\Sigma}}$，进一步计算出能够最小化风险的股票权重之比 $\hat{\boldsymbol{w}}^*$，并根据此权重比例进行下单和调仓

13.3.3 策略代码

```
import numpy as np
import pandas as pd

start = '2011-01-01'
end   = '2015-01-01'
benchmark = 'HS300'
universe  = ['601398.XSHG', '600028.XSHG', '601988.XSHG', '600036.XSHG', '600030.XSHG',
             '601318.XSHG', '600000.XSHG', '600019.XSHG', '600519.XSHG', '601166.XSHG']
capital_base = 1000000
refresh_rate = 10

def initialize(account):
    return

def handle_data(account):
    daily_returns = []
    for stock in account.universe:
        hist = account.get_symbol_history(stock, 20)
        daily_returns.append([hist['closePrice'][t]/hist['preClosePrice'][t] for t in range(1, 20)])
    daily_returns = np.array(daily_returns)

    cov_matrix = np.cov(daily_returns, y=None, rowvar=1, bias=0, ddof=None)
    cov_matrix = np.matrix(cov_matrix)
```

```
# calculate global minimum portfolio weights
one_vector = np.matrix(np.ones(len(account.universe))).transpose()
one_row    = np.matrix(np.ones(len(account.universe)))

cov_matrix_inv = np.linalg.inv(cov_matrix)
numerator   = np.dot(cov_matrix_inv, one_vector)
denominator = np.dot(np.dot(one_row, cov_matrix_inv), one_vector)
new_weights = numerator/denominator

# set leverage to 1
leverage = sum(abs(new_weights))
ptf_value = account.cash
for stock, a in account.valid_secpos.items():
    ptf_value += a * account.referencePrice[stock]
ptf_value /= leverage

change = {}
for i, stock in enumerate(account.universe):
    cur_pos = account.valid_secpos.get(stock, 0)
    new_pos = (ptf_value * new_weights[i])/account.referencePrice[stock]
    change[stock] = int(new_pos) - cur_pos

for stock in sorted(change.keys(), key=change.get):
    order(stock, change[stock])
```

13.4 Value-Weighted Average Price (VWAP)

13.4.1 策略实现

- 计算过去一段时间内的平均成交价 (VWAP) :

$$vwap = \frac{\sum_t TurnoverValue}{\sum_t TurnoverVolume}$$

- 当昨日最低价低于 VWAP 的某一阈值时，买入
- 当昨日最低价高于 VWAP 时，卖出

13.4.2 策略代码

```
start = '2011-01-01'
end   = '2014-08-01'
benchmark = 'SH50'
universe  = set_universe('SH50')
```

```
capital_base = 100000

threshold = 0.05

def initialize(account):
    return

def handle_data(account):
    for stk in account.universe:
        try:
            hist = account.get_symbol_history(stk, 5)
        except:
            continue

        vwap30 = sum(hist['turnoverValue'])/sum(hist['turnoverVol'])
        if hist['lowPrice'][-1] < vwap30 * (1-threshold) and stk not in account.valid_secpos:
            order(stk, 100)
        elif hist['lowPrice'][-1] >= vwap30 and stk in account.valid_secpos:
            order_to(stk, 0)
```

13.5 Lunar Phase

13.5.1 策略实现

1. 满月买进，新月/残月卖出
2. 新月/残月买进，满月卖出

13.5.2 策略代码

共用部分

```
start = '2011-01-01'
end   = '2015-01-01'
benchmark = 'SH50'
universe  = set_universe('SH50')
capital_base = 50000

def GetPhaseOfMoonEighth(year, month, day):
    r = year % 100 % 19
    if r > 9:
        r -= 19
    r = ((r * 11) % 30) + month + day
    if month < 3:
        r += 2
```

```
    r -= 4 if year<2000 else 8.3
    r = int((r+0.5)%30)
    return r+30 if r<0 else r
```

满月买进

```
def initialize(account):
    account.amount = 100

# Buy at Full Moon Sell at Crescent
def handle_data(account):
    dt = account.current_date
    index = GetPhaseOfMoonEighth(dt.year, dt.month, dt.day)

    for stk in account.universe:
        if (14 <= index <= 15) and stk not in account.valid_secpos:
            order(stk, account.amount)
        elif (28 <= index) and stk in account.valid_secpos:
            order_to(stk, 0)
```

满月卖出

```
def initialize(account):
    account.amount = 100

# Buy at Crescent Sell at Full Moon
def handle_data(account, data):
    dt = account.current_date
    index = GetPhaseOfMoonEighth(dt.year, dt.month, dt.day)

    for stk in account.universe:
        if (28 <= index) and stk not in account.valid_secpos:
            order(stk, account.amount)
        elif (14 <= index <=15) and stk in account.valid_secpos:
            order_to(stk, 0)
```

13.6 Poisson Price Change

13.6.1 策略思路

- 假设股价在 N 日内的涨跌天数为截尾 Poisson 分布，根据历史上 N 天的数据预测随后涨跌日数的概率，并依此调仓。

13.6.2 策略实现

- 根据历史 N 日的涨跌，确定 Poisson 分布的参数 λ ：心理线指标 PSY。
- 考虑未来上涨天数不少于 N/2 天的概率以及对应下跌天数的概率，若两者相差足够大，且股价偏离历史均值一定量时，则进行买卖操作。

13.6.3 策略代码

```
start = '2011-08-01'
end   = '2015-03-01'
benchmark = 'HS300'
universe  = ['601398.XSHG', '600028.XSHG', '601988.XSHG', '600036.XSHG', '600030.XSHG',
             '601318.XSHG', '600000.XSHG', '600019.XSHG', '600519.XSHG', '601166.XSHG']
capital_base = 100000
refresh_rate = 9
window = 9

#=================================================================#
#   To run an algorithm in Quartz, you need two functions:      #
#   initialize and handle_data.                                 #
#=================================================================#

# i is a counter who records days have been passed
# universe is pool of stocks

def initialize(account):
    account.universe = universe
    account.num_shares = 0
    account.max_notional = 10000.0
    account.min_notional = -10000.0

def handle_data(account):
    # to check if stock is in the candidate stock pool
    sn = len(account.universe)

    for sid in range(sn):
        try:
            hist = account.get_symbol_history(sid, window)
        except:
            continue
        stock = account.universe[sid]
        price = hist['closePrice'][-1]
        position = account.valid_secpos.get(stock, 0)
        notional = position * price
        num_shares = account.cash / price * 0.01
```

```
prices = hist['closePrice']
if prices is None:
    return

mu = prices.mean()
sd = prices.std()
k = math.sqrt(1 / (1 - 0.865))
lb = mu - k * sd
ub = mu + k * sd
pr_up = 0
pr_dw = 1
up = 0
dw = 0
eq = 1

tdn = len(prices) # trading day number
for i in range(1, tdn):
    if prices[i] > prices[i-1]:
        up += 1
    elif prices[i] < prices[i-1]:
        dw += 1
    else:
        eq += 1

    if up+dw+eq == tdn-1:
        l = (up * 1.0) / (up+dw+eq) * tdn # lambda in Poisson dist, PSY line
        for k in range(tdn, tdn/2, -1):
            pr_up += l ** k * math.exp(-k) / math.factorial(k)
            pr_dw -= l ** k * math.exp(-k) / math.factorial(k)

        diff = abs(pr_up - pr_dw)
        if diff >= 0.1:
            if pr_up < pr_dw or price < lb and notional < account.max_notional:
                order(stock, num_shares * diff)
                account.num_shares = num_shares

            if pr_up > pr_dw or price > ub and notional > account.min_notional:
                order(stock, -num_shares * diff)
```

Part III

CAL

CHAPTER 14

引论

14.1 什么是 CAL？

- CAL (Common Analytic Library)
- 固定收益及衍生品建模
- 核心以 C++ 开发保证高性能
- 有 Python 接口，方便使用

在本文档中，我们会专注于 CAL 的 Python 接口介绍。

14.2 CAL 有什么？

- 市场产品类型全覆盖

  包含股票、期货、固定收益以及衍生品等各种类型产品；
- 中国市场定制：

  Shibor 互换，FR007 互换。。。
- 复杂定价工具：

  OAS 计算框架，Black-Scholes,，Hull-White，Heston，SABR。。。
- 例子与文档：

  notebooks，Python API 文档。。。

14.3 为什么要写 CAL？

现阶段的中国市场缺乏为本土市场定制化的金融分析工具。广大中国中小投资者在面对这个纷繁复杂的市场，往往束手无策，不知分析从何而起：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？这是萦绕在所有投资者脑海中的问题。这些在过去，都要通过购买昂贵的专业终端才能满足，这个是广大个人投资者可望而不可及的。

在上面的大环境下，CAL 应运而生。依托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。CAL 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无往而不利！

CAL 之所以成为 CAL，有它本身的价值：

14.3.1 本地化

这里本地化意味着我们两方面的努力：

1. 本土化金融工具

CAL 开发团队意识到了国内金融创新的潮流，新产品层出不穷，为此积极根据中国市场的需要定制产品：Shibor 市场指数，回购互换，中国国债期货以及等等。这个追踪过程是持续的，我们会密切追踪国内市场的需求，对 CAL 进行大刀阔斧的扩展。

2. 完备的文档

我们会持续编写 CAL 的说明文档，该文档既面向直接用户，也面向二次开发者，深入浅出的详解 CAL 的各个功能模块以及组合方式。在我们看来，该文档的起点低，但落点不低，可以满足从初级用户到 IT 开发人员的多方面需求。除此之外，我们还会持续发布 CAL 团队基于 CAL 的金融工程研究文章以及量化策略报告。这样的文档既可以作为样例供用户模仿学习，也可以作为 CAL 团队与用户交流的桥梁。

3. 快速响应

由于是完全的本土开发团队，我们可以随时跟踪直接用户的需求，快速响应。我们有详尽的开发团队联系方式，并且通过举行培训、建立交流群的方式与用户互动。

14.3.2 平台整合

CAL 是比较纯粹的金融计算库，这意味它有如下的特征：

- 主要能力是计算能力；
- 不涉及数据，不涉及 IO。

这样的设计思路是完全正确的，我们也很认同这一模式。这也是典型的 Quant 库的特征 [27] 。但是我们也意识到了在 Mercury 平台上，我们有着像 DataAPI，Quartz 这样的数据以及回测平台。如何将这三大部分整合起来？我们也在这一方面做出了努力：

- 在工厂函数 模块中，我们将 DataAPI 与 CAL 亲密整合，极大的方便了用户创建金融产品。

14.4 Hello, CAL！

在 IPython 或者 Python 环境下，将 CAL 的 Python 接口导入当前环境，只需键入：

```
In [1]: from CAL.PyCAL import *
```

这里 PyCAL 是将 CAL 的相关工具引入当前命名空间下，方便使用。

```
In [2]: print Config.version
0.6.1-r2588
```

下面让我们运行一段小脚本，来看一下 CAL 能做什么:

```
from CAL.PyCAL import *

today = Date(2014,10,21)
calendar = Calendar('China.IB')

if calendar.isBizDay(today):
    print u'恩，今天是工作日。。。'
else:
    print u'哦也，今天放假！'
```

以上的代码会根据中国银行间市场工作日日历（China.IB）判断今天是不是工作日，输出结果取决于具体当日的日期。本部分写作时为 2014 年 10 月 21 日，输出结果为:

```
恩，今天是工作日。。。
```

在段简单的代码中我们涉及到的 CAL 接口包括：

- Date
- Date.todaysDate
- Calendar
- Calendar.isBizDay

都会在后续的章节中覆盖到。

14.5 CAL 文档结构

引论

即为本章，对于 CAL 项目的来历以及目标做了基本介绍。阅读本章，读者可以对 CAL 有一个概览性的印象。

如何为金融产品定价

介绍了如何使用 CAL 进行金融产品的定价，特别是介绍了 *产品-模型-算法周期*。这章介绍了 CAL 的设计思路以及理念。通过阅读本章，读者可以了解在 CAL 中进行定价的常规流程。有了本章的基础，读者可以自由的组织之后模块中的各个组件，搭建自己的金融建模平台。

工厂函数

该章介绍了 CAL 中的工厂函数。工厂函数的存在，是 CAL 结合 DataAPI，争取为用户降低输入复杂度而做的努力。现阶段工厂函数涵盖：债券，收益率曲线以及期权快照等内容。

日期

该章为读者介绍了 CAL 中，关于日期操作的各种模块功能。金融生活中，常见的日期操作包括：日期的平移、工作日的计算，天数计数的方式以及如何按照一定规则生成日程表。这些功能都已经在 CAL 中实现。

指数

指数模块试图实现现实世界中，例如沪深 300，Shibor 这样市场标的的概念。现阶段，CAL 中有利率指数的实现。

固定收益

本章介绍与固定收益产品相关的 CAL 模块。内容包括：债券函数，债券，利率衍生品。它们的内容分别对应如下：

- 债券函数：介绍了常规债券指标的计算函数，例如：到期收益率、净价/全价。这些函数的存在是方便用户直接快速的使用与债券有关的计算功能；
- 债券：介绍了 CAL 中支持的债券类型，包括：零息债券、固定利率债券、浮动利率债券、本息摊还债券、可回售赎回债券以及可转换债券等；
- 利率衍生品：介绍了 CAL 中支持的利率衍生品类型，包括：互换等。

权益

本章介绍与股权产品相关的 CAL 模块。内容包括：期权函数、期权。它们的内容分别对应如下：

- 期权函数：介绍了常规期权模型的计算函数，例如：BSM 价格、隐含波动率以及其他一些常见期权模型。这些函数的存在是方便用户直接快速的使用与债券有关的计算功能；
- 期权：介绍了 CAL 中支持的期权类型，包括：简单期权、障碍期权，数字期权等。

期限结构

本章介绍金融中期限结构的概念，即一条以时间为指标的曲线。现阶段 CAL 中支持收益率曲线（Yield Curve）的概念。

模型

本章介绍了 CAL 中金融模型的概念。在 CAL 中，我们遵循*产品-模型-算法周期*，将金融产品的分析过程分拆为产品、模型、算法部分。模型专注于描述金融建模的数学表述。

定价引擎

本章介绍了 CAL 中定价算法的概念，即为*产品-模型-算法周期* 的第三部分。现阶段在介绍中，我们将算法分为两大类：期权算法与固定收益算法。

数学

本章介绍 CAL 中使用到的和已经实现的一些数学工具，涵盖：求根算法、优化算法、插值算法以及随机分布等。

枚举类型

介绍了 CAL 中使用的一些枚举类型。这些枚举类型帮助用户以更加友好的方式输入类型选择的参数。

CHAPTER 15

如何为金融产品定价

15.1 需求？

一家典型的金融机构（可能是一家银行，一家证券公司，一家私募。。。）都会有自己的投资头寸。这样的投资头寸包括但不限于：股票、债券、存款、衍生品以及资产证券化产品（ABS）。有些产品，比如股票、交易所债券，由于每天市场价格的波动，会直接反映在机构资产组合的每日损益上（PnL）。除此之外即使是场外产品（OTC），例如衍生品以及 ABS，也面临市场环境的变化，内在价值会产生变化。面对复杂多变的市场情形，机构有必要知道下面的三个问题：

- 我的资产组合面临哪些方面的风险（利率，股权）?
- 我的资产组合会有多大的价格/价值波动（风险，Risk）?
- 我需要如何做去控制风险不超过能承受的范围（对冲，Hedge）?

无论回答上面哪一个问题，都离不了一个核心能力：**定价**。这里 定价的含义包含几方面的含义：

- 获得资产的净现值（NPV）
- 获得资产的风险指标（Greeks，VaR）
- 设计新的产品（Structuring）

15.2 来自 CAL 的回答

CAL 正是为此而生。就像它的名字暗示的那样，CAL 核心能力是它的分析能力，这里意味着三件事：

- CAL 以分析金融产品的结构，价值以及风险为要务；
- CAL 并不包含金融交易能力，这方面的需求可以参考: Quartz；
- 除少数例外以外（例外可以在工厂函数 中找到），CAL 并不直接涉及数据获取或者存储。有这方面的需求的可以关注：DatAPI 。

坚持以上的理念，令 CAL 轻装上阵，令其功能专业，清晰，目标明确。

15.3 如何完成定价？

为了适应高度复杂的金融环境，CAL 是完全基于模块化设计的。这一特征的设计初衷可以参考 [3] 。在 CAL 中定价过程是流程化的，我们称只为：产品-模型-算法周期（Product-Model-Engine cycle，PME）。

15.3.1 产品-模型-算法周期

在 CAL，当我们需要对某一产品进行定价的时候（例如，简单期权），我们需要定义这一过程中涉及的三部分要素：

1. **产品**

产品是定价对象要素的描述，这些要素往往体现在产品说明书或者合约定义文件中。例如对于简单期权，就是到期时间，行权价，行权类型以及等等。这些特征是明确定义的，与任何假设无关。CAL 通过这些信息，构造 产品对象。

2. **模型**

模型是市场运行模式的数学表述。例如，期权定价领域经典的 Black-Scholes 模型。在 CAL 中 模型一般通过参数进行初始化。例如 CAL 中对应 Black-Scholes 模型的类型就以几个输入参数进行初始化：无风险利率，红利率，波动率等等。

3. **算法**

模型仅仅完成了描述，但它没有告诉我们怎么产生数字。算法定义了如何由模型产生数字的过程。例如对于 Black-Scholes 模型，可以由 解析算法产生数值结果。

关系可以由下图表示：

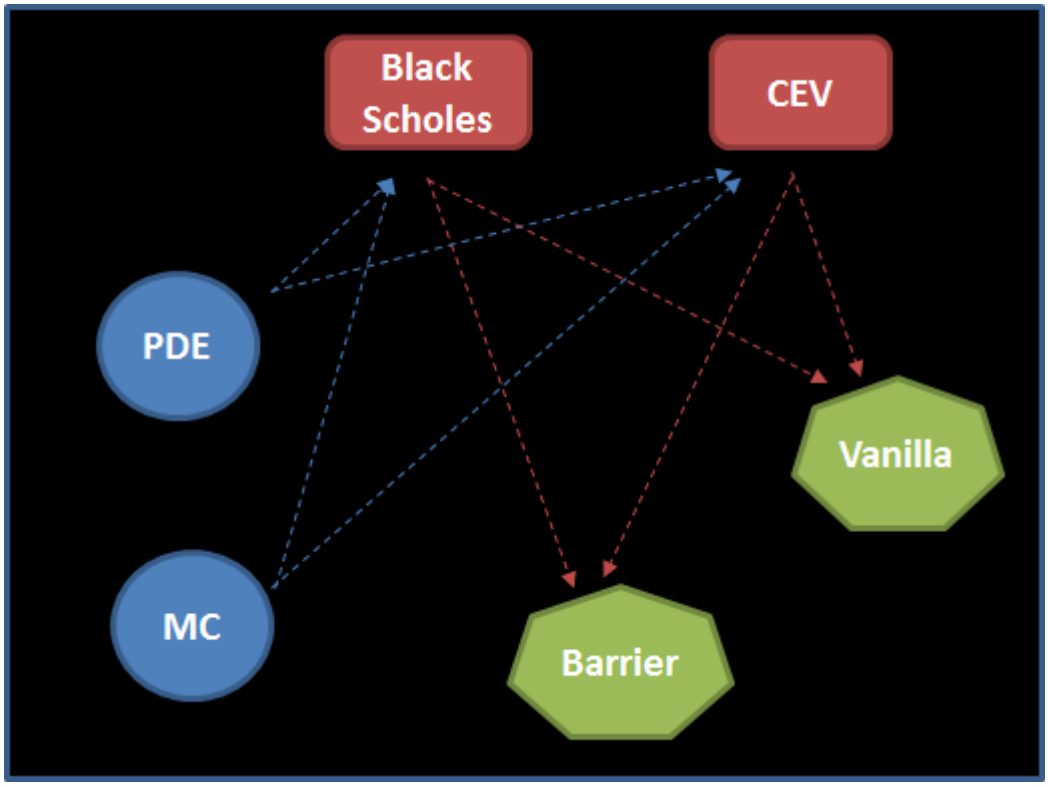


图 15.1: 产品 - 模型 - 算法周期

这样的模块化设计有着显著的优点：

- 单一模块概念清晰，做到解耦合；
- 正确的描述了定价过程的概念，与思考方式一致；
- 模块化使得“搭积木”式的设计方法成为可能。

让我们略微费些口舌来解释一下第三点。让我们仍然以 Black-Scholes 模型为例，Black-Scholes 模型有随机过程描述，但是关于期权价格的计算，它也有偏微分方程描述（Partial Differential Equation，PDE）。另外，这个模型虽然有显式解存在，但是仍然可以通过模拟仿真的方法进行求解（Monte Carlo，MC）。这一系列事实意味着，对于同一个模型（这里是 Black-Scholes），可以使用几种不同的 算法（解析，PDE，MC）。模块化的设计，使得这样的单一模型对应不同算法成为可能。

一般来讲，这三个部分在一个定价过程中不可分割。产品相对独立，算法依赖于模型，需要在模型建立之后，再进行构建。除了上面描述的模型与算法的一对多的关系，也同样存在着模型与算法之间，产品与模型之间多对一的关系。

但是事物也总是有自己的两面性，这样完全去耦合的模块化设计，也有自己的一些劣势：自由与懒惰，鱼与熊掌不可兼得。

模块化设计在给用户充分自由发挥的余地的同时，也使得用户需要设定相当多的细节。这增加了用户在处理一些标准化产品时候的负担。我们也意识到了这一问题。所以在必要的时候，我们会自知地破坏模块化的准则，让利于便宜性：

- 期权函数
- 债券函数
- 工厂函数

以上就是典型的反模块化的情形。

是不是感觉到听的云里雾里，晕头转向？或者早已按耐不住，跃跃欲试？OK，让我们从抽象的描述里面出来，来看一些新鲜的例子吧！

15.3.2 期权

先来看期权的例子，这里我们会专注于之前提到的 Black Scholes 模型，关于更复杂的模型可以见：期权函数

一如既往，我们首先将 CAL 导入当前环境：

```
In [1]: from CAL.PyCAL import *
```

首先开始定义我们的产品，这里我们假设产品的特征：

- 行权价：40
- 到期日：2014 年 5 月 17 日
- 行权方式：欧式
- 期权类型：看涨

```
In [2]: strike = 40.0
   ...: exDate = Date(2014, 5, 17)
   ...: exercise = EuropeanExercise(exDate)
   ...: optionType = Option.Call
   ...:
```

好了，可以把它们组合起来产生期权对象，我们将偿付类型与行权方式组合起来：

```
In [6]: payoff = PlainVanillaPayoff(optionType, strike)
   ...: option = EuropeanOption(payoff, exercise)
   ...:
```

Done！产品已经就位。

现在我们考虑模型，已经决定了是 Black Scholes 模型，那么这里有几个参数需要确定：

- 标的物起始价格为多少？
- 无风险利率是多少？
- 红利率是多少？
- 波动率是多少？

这里我们做如下的假设：

```
In [8]: underlying = 36.00
   ...: riskFree = 0.06
   ...: dividend = 0.00
   ...: volatility = 0.20
   ...: model = BlackScholesMertonProcessConstant(underlying,
   ...:                                           riskFree,
   ...:                                           dividend,
   ...:                                           volatility,
   ...:                                           'Actual/Actual (ISMA)')
   ...:
```

这里我们使用的模型是BlackScholesMertonProcessConstant ， 是常参数的 Black Scholes 模型。很简单不是吗？我们用短短 5 行代码就构建了期权发展史上最重要的模型！

等等还没完，我们的算法在哪里？这里我们先来个解析公式，让 CAL 小试牛刀：

```
In [13]: engine = AnalyticEuropeanEngine(model)
```

我们使用了：AnalyticEuropeanEngine

三大组件皆以就位，看看怎么来定价吧，现在我们有 option、model、engine 三个对象：

```
In [14]: option.setPricingEngine(engine)
   ....: SetEvaluationDate(Date(2013, 5, 17))
   ....: print("NPV: %.4f" % option.NPV())
   ....: print("Delta: %.4f" % option.delta())
   ....: print("Gamma: %.4f" % option.gamma())
   ....: print("Theta : %.4f" % option.theta())
   ....: print("Theta (per week) : %.4f" % (option.thetaPerDay() * 7))
   ....: print("Vega : %.4f" % option.vega())
   ....: print("Rho : %.4f" % option.rho())
   ....:
NPV: 2.1737
Delta: 0.4495
Gamma: 0.0550
Theta : -2.2653
Theta (per week) : -0.0434
Vega : 14.2469
Rho : 14.0100
```

注解: 注意我们这里多了一句语句:

```
SetEvaluationDate(Date(2013, 5, 17))
```

这句是设置全局估值日，也就是你想在哪天算 NPV。这个你得告诉 CAL 吧？

然后是否可以试试其他的算法呢，先让我们试试 PDE 方法，使用FDEuropeanEngine ：(关于 PDE 方法的详细介绍，比如见 [15])

```
In [23]: engine2 = FDEuropeanEngine(model)
   ....: option.setPricingEngine(engine2)
   ....: print("NPV: %.4f" % option.NPV())
   ....: print("Delta: %.4f" % option.delta())
   ....: print("Gamma: %.4f" % option.gamma())
   ....:
NPV: 2.1778
Delta: 0.4494
Gamma: 0.0549
```

再让我们试试使用 Monte Carlo 方法，使用MCEuropeanEngine :(关于 Monte Carlo 方法的介绍，比如见 [18])

```
In [28]: engine3 = MCEuropeanEngine(model, 'ld', 100000, 1)
   ....: option.setPricingEngine(engine3)
   ....: print("NPV: %.4f" % option.NPV())
   ....:
NPV: 2.1734
```

警告: 虽然是应用于同一模型，但是不同的算法不一定能产生完全相同种类的结果。比如在上一例中，PDE 方法与 MC 方法并无法产生与解析相同的结果，比如 PDE 方法没有除 Delta 以及 Gamma 之外的希腊数字；MC 方法直接无法产生任何希腊数字。这里边的原因一部分是出于性能的限制，另一部分原因是这些算法要产生希腊字母需要更多的配置。

15.3.3 债券

现在让我们进入固定收益领域，看看 CAL 的威力。让我们从最简单的债券开始，一个零息债券：

表 15.1: 139903 债券定义

代码	139903.IB
发行日	2013 年 5 月 13 日
到期日	2014 年 2 月 10 日
发行价格	97.9030
日历	China.IB
天数计数	Actual/Actual (ISMA)

```
In [31]: from CAL.PyCAL import *
   ....: # Infor about 139903.IB
   ....: today = Date(2013, 12, 2)
   ....: SetEvaluationDate(today)
   ....: settlementDays = 1
   ....: faceAmount = 100.0
   ....: issuePrice = 97.9030
   ....: issueDate = Date(2013, 5, 13)
   ....: maturityDate = Date(2014, 2, 10)
```

```
....: calendar = Calendar('China.IB')
....: dayCounter = DayCounter('Actual/Actual (ISMA)')
....:
```

好了，可以构造零息债券了：

```
In [42]: testBond = CTBZeroBond(settlementDays,
....:                           faceAmount,
....:                           issuePrice,
....:                           issueDate,
....:                           maturityDate,
....:                           calendar,
....:                           dayCounter)
....:
```

同样的产品已经就绪，现在该是模型登场的时候了。这里我们的模型是一条收益率曲线：FlatForwardRefEvaDate

```
In [43]: yc = FlatForwardRefEvaDate(settlementDays = 0,
....:                               calendar = 'NullCalendar',
....:                               forward = 0.05,
....:                               dayCounter = 'Actual/Actual (ISMA)')
....:
```

紧接着的就是算法，这里我们使用简单的折现算法：DiscountingBondEngine

```
In [44]: engine = DiscountingBondEngine(yc)
....: testBond.setPricingEngine(engine)
....: print('Clean Price: %.4f' % testBond.cleanPrice())
....:
Clean Price: 97.6167
```

你可以很容易改变模型：

```
In [47]: yc = FlatForwardRefEvaDate(settlementDays = 0,
....:                               calendar = 'NullCalendar',
....:                               forward = 0.04,
....:                               dayCounter = 'Actual/Actual (ISMA)')
....:
```

然后重新计算：

```
In [48]: engine = DiscountingBondEngine(yc)
....: testBond.setPricingEngine(engine)
....: print('Clean Price: %.4f' % testBond.cleanPrice())
....:
Clean Price: 97.7794
```

像我们之前说的那样，你可以把相同的模型与算法可以应用于不同的产品，形成 *产品与 模型的多对一* 模式。让我们看一个固定利率国债的例子：

表 15.2: 120014 债券定义

代码	120014.IB
发行日期	2012 年 8 月 16 日
起息日	2012 年 8 月 16 日
到期日	2017 年 8 月 16 日
付息频率	Annual
票息	2.95%
日历	Null Calendar
天数计数惯例	Actual/Actual (ISMA)

我们会用一个工厂函数BuildBond 完成这个债券的创建，具体何为工厂函数，请见：工厂函数

```
In [51]: # Infor about 120014.IB
   ....: testBond2 = BuildBond('120014.XIBE', paymentDateAdjusted = False)
   ....:
```

```
In [53]: testBond2.setPricingEngine(engine)
   ....: print('Clean Price: %.4f' % testBond2.cleanPrice())
   ....:
Clean Price: 96.3191
```

15.3.4 内嵌期权债券

热身做完了，让我们开始做一些更复杂的练习吧。事实上国内市场，除了国债，其他类型的债券很多都含有内嵌期权的结构，例如：可赎回，可回售，可转换。关于中国市场债券发展的概况，可见: [43]

这里我们介绍一下如何为一个可赎回/可回售债券定价，请耐心，这个例子会有一些长。

表 15.3: 可赎回债券定义

发行日期	2004 年 9 月 15 日
起息日	2004 年 9 月 15 日
到期日	2012 年 9 月 15 日
付息频率	Quarterly
票息	4.65%
日历	Null Calendar
天数计数惯例	Actual/Actual (ISMA)
行权频率	Quarterly
赎回价格	100

让我们继续一步一步的来，先定义我们的产品，这是最繁琐的一步：

```
In [55]: callSchedule = CallabilitySchedule()
   ....: callPrice = 100.0
   ....: numberOfCallDates = 24
   ....: callDate = Date(2006, 9 ,15)
```

```
....: nullCalendar = Calendar('NullCalendar')
....: for i in xrange(numberOfCallDates):
....:     myPrice = CallabilityPrice(callPrice, CallabilityPrice.Clean)
....:     callSchedule.addCallability(Callability(myPrice, Callability.Call, callDate))
....:     callDate = nullCalendar.advanceDate(callDate, '3M', BizDayConvention.Following);
....:
```

上面这段冗长的代码定义了债券的行权日程，从 2006 年 9 月 15 日开始，每 3 个月一次行权机会。

接着是生成债券对象，这个就与普通债券没有太大的区别了：

```
In [61]: dated = Date(2004, 9, 16)
....: issue = dated
....: maturity = Date(2012, 9, 15)
....: sch = Schedule(dated, maturity, '3M', 'NullCalendar',
....:                BizDayConvention.Unadjusted, BizDayConvention.Unadjusted,
....:                DateGeneration.Backward, False);
....: coupon = .0465;
....: redemption = 100.0;
....: faceAmount = 100.0;
....: settlementDays = 1
....: callableBond = CallableFixedRateBond(settlementDays, faceAmount, sch,
....:                                      coupon,
....:                                      'Actual/Actual (ISMA)',
....:                                      callSchedule,
....:                                      BizDayConvention.Following,
....:                                      redemption, issue)
....:
```

完成了最复杂的部分，模型部分就显得小菜一碟了，这里我们选择使用 Hull White 模型，关于 Hull White 模型或者其他短期利率模型的介绍，请见: [8]：

```
In [70]: rate = 0.055
....: today = Date(2007, 10, 16)
....: SetEvaluationDate(today)
....: yc = FlatForward(today, rate, 'Actual/360', Compounding.Compounded,
....:                  Frequency.Semiannual)
....: sigma = 0.03
....: reversionParameter = 0.03
....: hw1 = HullWhite(yc,reversionParameter,sigma)
....:
```

这里我们想多费些口舌。Hull White 模型有三个基本参数：回归均值，回归速度，波动率，其中回归均值可以从输入的收益率曲线直接导出（Implied）。所以这里可以看到，我们首先建立了一条平坦的收益率曲线，然后将它作为参数输入 Hull White 模型的构造函数中。

最后我们使用三叉树方法去基于 Hull White 模型计算。关于三叉树方法的具体介绍，请见 [26]

```
In [77]: engine1 = TreeCallableFixedRateBondEngine(hw1, 100)
   ....: callableBond.setPricingEngine(engine1)
   ....: print('Clean Price: %.4f' % callableBond.cleanPrice())
   ....:
Clean Price: 91.9891
```

试一下我们刚才所说的：模型与 算法之间可以多对一！换一个模型试试，这里我们选择 Blakc Karasinski 模型，具体介绍可以参考: [5]

```
In [80]: bk1 = BlackKarasinski(yc,reversionParameter,sigma)
```

然后重复之前相同的算法步骤：

```
In [81]: engine2 = TreeCallableFixedRateBondEngine(bk1, 100)
   ....: callableBond.setPricingEngine(engine2)
   ....: print('Clean Price: %.4f' % callableBond.cleanPrice())
   ....:
Clean Price: 96.1572
```

CHAPTER 16

工厂函数

16.1 债券工厂函数

债券工厂函数与 DataAPI 紧密结合，通过输入证券代码的形式，直接构建公开市场已经存在的债券：

```
In [1]: from CAL.PyCAL import *
   ...: bond = BuildBond('139903.XIBE')
   ...:
```

读者可以对比一下，如果由手工输入的代码量会比上面的代码大很多。在之前的章节中，我们已经提到了，**工厂函数**是为了提供方便性，不得不破坏解耦合性而做的妥协。类似的妥协，我们还可以在债券函数 、期权函数 中看到。

16.2 收益率曲线工厂函数

我们也可以通过名称构造收益率曲线，现阶段我们只支持国债收益率曲线。

```
In [3]: refDate = Date(2014, 10, 29);
   ...: curve = BuildCurve('TREASURY.XIBE', refDate);
   ...:
```

现在看看这条曲线长什么样？

```
In [5]: curve.curveProfile()
Out[5]:
                 date  discount  forward(%)  zero(%)
2014-10-29 2014-10-29    1.0000      3.2146   3.2668
2014-11-29 2014-11-29    0.9973      3.2433   3.2668
2014-12-29 2014-12-29    0.9945      3.4028   3.3241
2015-01-29 2015-01-29    0.9916      3.4491   3.4043
2015-02-28 2015-02-28    0.9889      3.3024   3.3951
2015-03-29 2015-03-29    0.9863      3.2851   3.3861
2015-04-29 2015-04-29    0.9836      3.2979   3.3766
```

2015-05-29	2015-05-29	0.9809	3.3319	3.3780
2015-06-29	2015-06-29	0.9781	3.3347	3.3795
2015-07-29	2015-07-29	0.9754	3.3621	3.3809
2015-08-29	2015-08-29	0.9726	3.4007	3.3881
2015-09-29	2015-09-29	0.9698	3.4147	3.3954
2015-10-29	2015-10-29	0.9671	3.4138	3.4024
2015-11-29	2015-11-29	0.9643	3.4084	3.4071
2015-12-29	2015-12-29	0.9616	3.4172	3.4116
2016-01-29	2016-01-29	0.9588	3.4263	3.4163
2016-02-29	2016-02-29	0.9560	3.4354	3.4210
2016-03-29	2016-03-29	0.9534	3.4439	3.4254
2016-04-29	2016-04-29	0.9506	3.4530	3.4301
2016-05-29	2016-05-29	0.9479	3.4618	3.4347
2016-06-29	2016-06-29	0.9451	3.4709	3.4394
2016-07-29	2016-07-29	0.9424	3.4797	3.4439
2016-08-29	2016-08-29	0.9397	3.4887	3.4486
2016-09-29	2016-09-29	0.9369	3.4978	3.4533
2016-10-29	2016-10-29	0.9342	3.4801	3.4579
2016-11-29	2016-11-29	0.9314	3.4582	3.4603
2016-12-29	2016-12-29	0.9288	3.4627	3.4626
2017-01-29	2017-01-29	0.9261	3.4673	3.4650
2017-02-28	2017-02-28	0.9234	3.4717	3.4673
2017-03-29	2017-03-29	0.9209	3.4760	3.4695
...	...	...	...	...
2062-05-29	2062-05-29	0.1126	5.1810	4.6927
2062-06-29	2062-06-29	0.1121	5.1831	4.6938
2062-07-29	2062-07-29	0.1117	5.1851	4.6949
2062-08-29	2062-08-29	0.1112	5.1873	4.6960
2062-09-29	2062-09-29	0.1107	5.1894	4.6971
2062-10-29	2062-10-29	0.1102	5.1914	4.6982
2062-11-29	2062-11-29	0.1097	5.1936	4.6993
2062-12-29	2062-12-29	0.1093	5.1956	4.7004
2063-01-29	2063-01-29	0.1088	5.1977	4.7015
2063-02-28	2063-02-28	0.1083	5.1998	4.7026
2063-03-29	2063-03-29	0.1079	5.2018	4.7036
2063-04-29	2063-04-29	0.1074	5.2039	4.7047
2063-05-29	2063-05-29	0.1069	5.2060	4.7058
2063-06-29	2063-06-29	0.1065	5.2081	4.7069
2063-07-29	2063-07-29	0.1060	5.2101	4.7080
2063-08-29	2063-08-29	0.1055	5.2123	4.7091
2063-09-29	2063-09-29	0.1051	5.2144	4.7102
2063-10-29	2063-10-29	0.1046	5.2164	4.7113
2063-11-29	2063-11-29	0.1042	5.2186	4.7124
2063-12-29	2063-12-29	0.1037	5.2206	4.7135
2064-01-29	2064-01-29	0.1033	5.2227	4.7146
2064-02-29	2064-02-29	0.1028	5.2249	4.7157
2064-03-29	2064-03-29	0.1024	5.2268	4.7167

```
2064-04-29  2064-04-29    0.1019      5.2290    4.7178
2064-05-29  2064-05-29    0.1015      5.2310    4.7189
2064-06-29  2064-06-29    0.1010      5.2331    4.7200
2064-07-29  2064-07-29    0.1006      5.2352    4.7211
2064-08-29  2064-08-29    0.1002      5.2373    4.7222
2064-09-29  2064-09-29    0.0997      5.2394    4.7233
2064-10-29  2064-10-29    0.0993      5.2415    4.7244

[601 rows x 4 columns]
```

是不是变得很简单？

16.3 校正工具工厂函数

我们会在曲线校正 章节给大家介绍如何从市场报价获取收益率曲线。在这里我们想介绍，像债券工厂函数一样，用于曲线校正的市场工具，也可以通过工厂函数方便的构造：

```
In [6]: codes = ['120007.XIBE', '120017.XIBE', '140022.XIBE',
   ...:          '140015.XIBE', '140004.XIBE', '130023.XIBE',
   ...:          '140008.XIBE', '130020.XIBE', '140013.XIBE']
   ...: prices = [99.7613, 99.6708, 100.3650,
   ...:           100.9778, 100.6094, 100.5826,
   ...:           102.0884, 100.0731, 102.0485]
   ...: instruments = BuildBondHelper(codes, prices)
   ...: calCurve = CalibratedYieldCurve(refDate,
   ...:                                 instruments,
   ...:                                 'Actual/Actual (ISMA)',
   ...:                                 'Zero',
   ...:                                 'Cubic')
   ...:
```

所有的工作都完成了。CAL 的 BuildBondHelper 会根据用户输入的证券代码 codes，从数据 API 中获取需要的证券定义，并根据报价 prices 构造市场工具对象。这可以极大的简化用户的输入。

16.4 期权工厂函数

获取当前时刻的市场快照：OptionsDataSnapShot

```
In [10]: table = OptionsDataSnapShot()
   ....: table[:5]
   ....:
Out[11]:
    dataDate  dataTime  optionId      instrumentID contractType  \
0 2016-01-19  15:45:03  10000287  510050C1603M02650          CO
1 2016-01-19  15:45:12  10000288  510050C1603M02700          CO
```

```
2  2016-01-19  15:44:51  10000289  510050C1603M02750          CO
3  2016-01-19  15:45:06  10000290  510050C1603M02800          CO
4  2016-01-19  15:45:12  10000291  510050C1603M02850          CO

   strikePrice     expDate  lastPrice
0       2.6500  2016-03-23     0.0115
1       2.7000  2016-03-23     0.0098
2       2.7500  2016-03-23     0.0082
3       2.8000  2016-03-23     0.0070
4       2.8500  2016-03-23     0.0060
```

根据市场快照计算风险值：OptionsAnalyticResult

```
In [12]: result = OptionsAnalyticResult(table)
   ....: result[:5]
   ....:
Out[13]:
    dataDate  dataTime  optionId       instrumentID contractType  \
0  2016-01-19  15:45:03  10000287  510050C1603M02650          CO
1  2016-01-19  15:45:12  10000288  510050C1603M02700          CO
2  2016-01-19  15:44:51  10000289  510050C1603M02750          CO
3  2016-01-19  15:45:06  10000290  510050C1603M02800          CO
4  2016-01-19  15:45:12  10000291  510050C1603M02850          CO

   strikePrice     expDate  lastPrice     vol  delta  gamma    vega     rho  \
0       2.6500  2016-03-23     0.0115  0.3534 0.0845 0.4914  0.1381  0.0295
1       2.7000  2016-03-23     0.0098  0.3627 0.0722 0.4251  0.1226  0.0253
2       2.7500  2016-03-23     0.0082  0.3699 0.0610 0.3658  0.1076  0.0213
3       2.8000  2016-03-23     0.0070  0.3781 0.0523 0.3168  0.0953  0.0183
4       2.8500  2016-03-23     0.0060  0.3861 0.0449 0.2750  0.0845  0.0157

    theta
0 -0.1430
1 -0.1301
2 -0.1163
3 -0.1051
4 -0.0950
```

也可以直接获取当前快照的计算结果：

```
In [14]: result = OptionsAnalyticResult()
   ....: result[:5]
   ....:
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-14-b636b427d915> in <module>()
----> 1 result = OptionsAnalyticResult()
```

```
C:\Anaconda2\lib\site-packages\CAL\Factory\Options.pyc in OptionsAnalyticResult(optionData, spotPrice, evaluationDate)
    147
    148     if spotPrice is None:
--> 149         equityData = DataAPI.MktTickRTSnapshotGet(securityID='510050.XSHG')
    150         spotPrice = equityData['lastPrice'][0]
    151

C:\Anaconda2\lib\site-packages\DataAPI\DATAYES.pyc in MktTickRTSnapshotGet(securityID, exchangeCD, assetClass, field, pane
   1657
   1658         if csvString is None or len(csvString) == 0 or (csvString[0] == '-' and csvString[1] != '1') or csvString[0] == '{':
-> 1659             raise Exception((u'%s for request: %s' % (csvString if csvString is not None else 'Query Error', ''.join(requestString
   1660         elif csvString[:2] == '-1':
   1661             csvString = ''

Exception: -5:Server Busy for request: /api/market/getTickRTSnapshot.csv?appname=mercury&securityID=510050.XSHG&exchange
Out[15]:
     dataDate  dataTime  optionId       instrumentID contractType  \
0  2016-01-19  15:45:03  10000287  510050C1603M02650           CO
1  2016-01-19  15:45:12  10000288  510050C1603M02700           CO
2  2016-01-19  15:44:51  10000289  510050C1603M02750           CO
3  2016-01-19  15:45:06  10000290  510050C1603M02800           CO
4  2016-01-19  15:45:12  10000291  510050C1603M02850           CO

   strikePrice     expDate  lastPrice     vol  delta  gamma    vega     rho  \
0       2.6500  2016-03-23     0.0115  0.3534 0.0845 0.4914  0.1381  0.0295
1       2.7000  2016-03-23     0.0098  0.3627 0.0722 0.4251  0.1226  0.0253
2       2.7500  2016-03-23     0.0082  0.3699 0.0610 0.3658  0.1076  0.0213
3       2.8000  2016-03-23     0.0070  0.3781 0.0523 0.3168  0.0953  0.0183
4       2.8500  2016-03-23     0.0060  0.3861 0.0449 0.2750  0.0845  0.0157

    theta
0 -0.1430
1 -0.1301
2 -0.1163
3 -0.1051
4 -0.0950
```

接口

---

class Factory.BondBuilder(Instruments.Bond)

由证券代码生成的债券对象

securityID(self)

返回 证券代码

返回类型 str

issuer(self)

返回 发行人

返回类型 str

exchange(self)

返回 交易中心

返回类型 str

fullName(self)

返回 全称

返回类型 str

maturity(self)

返回 期限，比如'3Y'

返回类型 str

shortName(self)

返回 简称

返回类型 str

Factory.BuildBond(secID, settlementDays = 1, paymentDateAdjusted = True)

使用证券代码生成债券对象

参数

- secID (list of str) – 证券代码
- settlementDays (int) – 清算速度
- paymentDateAdjusted (bool) – 是否根据日历调整付息日

返回 生成的债券对象

返回类型 Factory.BondBuilder

class Factory.YieldCurveBuilder(YieldCurve.YieldCurveProxy)

收益率曲线工厂对象，现阶段只支持国债曲线，例如：TREASURY.XIBE 代表银行间固定利率国债收益率曲线

name(self)

返回 曲线名称

返回类型 str

Factory.BuildCurve(curveName, date)

参数

- curveName (str) – 曲线名称，例如：TREASURY.XIBE 代表银行间固定利率国债收益率曲线
- date (Dates.Date) – 曲线基准日

返回 收益率曲线工厂对象

返回类型 Factory.YieldCurveBuilder

Factory.BuildBondHelper(secID, quotes, settlementDays = 1, paymentDateAdjusted = True)

参数

- secID (list of str) – 证券代码
- quotes (list of float) – 证券报价
- settlementDays (int) – 清算速度
- paymentDateAdjusted (bool) – 是否根据日历调整付息日

返回 生成的债券 helper 对象

返回类型 Factory.BondHelperBuilde

CHAPTER 17

日期

17.1 日期

如何操作日期？这个问题在金融里面无处不在。

- 如何从一段格式化字符串中获取正确的日期信息？
- 如何从某个日期出发，向前（或向后）递推 n 个周期？
- 如何判断一个日期是否是正确的工作日？
- 如何计算两个日期之间的距离？换算成年约为多少？
- 如何处理金融中的繁复的工作日惯例？(向前（Preceding）? 向后（Following）? 调整的向后（Modified Following）

Python 虽然有自己的内置日期对象，但是无法回答上面所有的问题。我们需要设计新的日期类型，量身定做满足以上需求的定制化的日期类型。

17.1.1 CAL 日期

为了解决上述问题, CAL 定义了自己的日期对象: Date。配合Calendar, DayCounter, BizDayConvention 等类型的使用，可以方便的完成以上的任务。

基本的，我们通过年，月，日的形式构造一个日期对象:

```
In [1]: from CAL.PyCAL import *
```

```
In [2]: print Date(2014, 10, 22)
2014-10-22
```

当然我们也可以通过字符串的形式读入日期：

```
In [3]: print Date.parseISO('2014-10-22')
2014-10-22
```

可以轻易的取出日期中的各种成分：

```
In [4]: today = Date.todaysDate()
   ...: print 'Year: ' + str(today.year())
   ...: print 'Month: ' + str(today.month())
   ...: print 'Day: ' + str(today.dayOfMonth())
   ...:
Year: 2016
Month: 1
Day: 19
```

在我们介绍后面Period 类型之前，可以先看一下结合Period 如何实现日期的加减：

```
In [8]: span = Period('3M')
   ...: nextDate = today + span
   ...: print nextDate
   ...:
2016-04-19
```

```
In [11]: prevDate = today - span
   ....: print prevDate
   ....:
2015-10-19
```

除此之外我们还可以轻松的获取诸如某月第某个周几是哪个日期这样的信息。这种类似的信息在计算期货合约到期日和交割日的时候特别有用。

```
In [13]: print Date.NthWeekDay(size = 4, weekday = 0, month = 10, year = 2014)
2014-10-25
```

上面的式子返回 2014 年（year）第 4 个（size）周六（weekday）是几号。

除此之外，date 可以进行常规的比较运算：

```
In [14]: date1 = Date(2014, 10, 22)
   ....: date2 = Date(2013, 10, 22)
   ....: print date1 < date2
   ....:
False
```

关于后 4 个问题的回答，可以在对应的Period，Calendar，DayCounter 模块中找到答案。

17.1.2 CAL 日期的转换

除了上面遇到的问题，我们还经常遇到需要从外部导入数据的情况，例如：

- 我手上现在有一个 Python 标准库 datetime 对象，怎么把它转化为 CAL 的 Date 类型？
- 我从外部导入一个 excel 文件，它的日期值是一个长整数，这个时候我该怎么做？
- 我的 date 实际上一个有一定格式的字符串，这我又该怎么办？

同时我们也面临着如何做反向操作的问题（即如何将 CAL 的日期类型转换为外部类型）。

1. Python 的标准库 datetime 类型

```
In [17]: pyDate = dt.datetime(2014, 8, 12)
   ....: calDate = Date.fromDateTime(pyDate)
   ....: calDate
   ....:
Out[19]: Date(2014,8,12)
```

```
In [20]: calDate.toDateTime()
Out[20]: datetime.datetime(2014, 8, 12, 0, 0)
```

2. Excel 的日期长整数

```
In [21]: serialNumber = 41863L
   ....: calDate = Date.fromExcelSerialNumber(serialNumber)
   ....: calDate
   ....:
Out[23]: Date(2014,8,12)
```

```
In [24]: calDate.toExcelSerialNumber()
Out[24]: 41863L
```

3. 格式化字符串

现阶段，CAL 支持从 ISO 标准日期格式读入日期：

```
In [25]: calDate = Date.parseISO('2014-08-12')
   ....: calDate
   ....:
Out[26]: Date(2014,8,12)
```

```
In [27]: calDate.toISO()
Out[27]: '2014-08-12'
```

除此之外，CAL 支持更复杂的字符串格式化操作，这些操作与内置类型 datetime 是完全兼容的：

```
In [28]: calDate = Date.strptime('2014/12/3', '%Y/%m/%d')
   ....: calDate
   ....:
Out[29]: Date(2014,12,3)
```

可以将 CAL.Date 转换为你想要的字符串格式：

```
In [30]: calDate.strftime('%Y%m%d')
Out[30]: '20141203'
```

17.1.3 日期的功能函数

除了上面介绍的功用外，CAL 的日期函数也提供了一系列的操作来完成一些常用的功能。

1. 判断某个年份是否为闰年

```
In [31]: print Date.isLeap(2014)
   ....: print Date.isLeap(2016)
   ....:
False
True
```

2. 判断某个日期是否为月末

```
In [33]: print Date.isEndOfMonth(Date(2014,2,1))
   ....: print Date.isEndOfMonth(Date(2014,2,28))
   ....: print Date.isEndOfMonth(Date(2016,2,28))
   ....:
False
True
False
```

17.1.4 接口

class Dates.Date

___init___(self, year = None, month = None, days = None)

通过年，月，日构造日期

参数

- year (int) – 年
- month (int) – 月
- days (int) – 日

返回 日期

返回类型 Dates.Date

toTimestamp(self)

将 Date 对象转换为 pandas Timestamp 对象

返回 pandas.Timestamp 对象

返回类型 pandas.Timestamp

dateRange(startDate, endDate)

返回区间内所有日期，包含起始日，不包括结束日

参数

- startDate (Dates.Date) – 起始日
- endDate (Dates.Date) – 结束日

返回 日期序列

返回类型 list

year(self)
返回当前日期年

返回 年

返回类型 int

month(self)
返回当前日期月

返回 月

返回类型 int

dayOfMonth(self)
返回当前日期日

返回 日

返回类型 int

___sub___(self, span)
从当前日期向前移动 span 周期

参数 span (Periods.Period) – 周期

返回 日期

返回类型 Date

___add___(self, add)
从当前日期向后移动 span 周期

参数 span (Periods.Period) – 周期

返回 日期

返回类型 Date

___lt___(self, date2)
判断当前日期是否小于 date2

参数 date2 (Dates.Date) – 比较日期

返回 比较结果

返回类型 bool

___le___(self, date2)
判断当前日期是否小于等于 date2

参数 date2 (Dates.Date) – 比较日期

返回 比较结果

返回类型 bool

___eq___(self, date2)

判断当前日期是否等于 date2

参数 date2 (Dates.Date) – 比较日期

返回 比较结果

返回类型 bool

___str___(self)

返回日期的字符串表示

返回 字符串表示

返回类型 str

___repr___(self)

返回日期的表达式

返回 内部表示

返回类型 str

toExcelSerialNumber(self)

返回日期的 excel 长整型数

返回 Excel 长整型数

返回类型 long

toISO(self)

返回日期的 ISO 字符串表示，即 YYYY-MM-DD

返回 ISO 字符串表示

返回类型 str

isLeap(year)

返回指定年份是否是闰年

参数 year (int) – 指定的年份

返回 是否为闰年

返回类型 bool

isEndOfMonth(date)

判断指定日期是否为月份最后一日

参数 date (Dates.Date) – 指定日期

返回 是否为当月最后一日

返回类型 bool

NthWeekDay(size, weekday, month, year)
计算指定年月下某个指定 week day 的日期

参数

- size (int) – 第几个
- weekday (int) – 周几，其中周六为 0
- month (int) – 月
- year (int) – 年

返回 满足条件的日期

返回类型 Dates.Date

todaysDate()
返回系统今天的日期

返回 今天的日期

返回类型 Dates.Date

parseISO(dateStr)
按照 ISO 格式读取 Date 字符串 (YYYY-MM-DD)

参数 dateStr (str) – ISO 标准日期字符串

返回 解析后的日期

返回类型 Dates.Date

fromDateTime(dtObj)
将 Python date 或 datetime 类型转换为 CAL 的 Date 类型

参数 dtObj (datetime.datetime) – python 的 datetime 对象

返回 日期

Rtypr Dates.Date

fromExcelSerialNumber(serialNumber)
将 Excel 的日期长整数转化为 CAL 的 Date 类型

参数 serialNumber (long) – Excel 的日期长整数

返回 日期

返回类型 Dates.Date

17.2 工作日历

生活中有工作日以及节假日的概念，基本上所有的金融活动都是在工作日展开，比如是周一至周五。为了正确的区分这两个不同的日期概念，需要 CAL 提供这方面接口方法。当然实际的情况比周一周五这里提

到的概念要复杂多：

- 除了正常的周六周日还有哪些法定节假日？
- 哪些周末因为调休的缘故变成了工作日？
- 某个周六是银行间市场的交易日，但是是交易所市场的休息日，怎么办？
- 从今天开始往后推 3 个月，然后希望那天是个工作日，如果不是的话顺延，这个该怎么做？
- 今天不是工作日，我要向前调整至最近的一个工作日该怎么做？

CAL 通过内建Calendar 对象回答了以上所有的问题。

17.2.1 CAL 工作日历

我们可以通过一条简单的字符串描述一个工作日历，一般形式为：< 国家 >.< 市场 >，部分情况下 < 市场 > 可以不给。完整的这样的字符串列表请见: 日历列表

```
In [1]: from CAL.PyCAL import *
   ...: cal = Calendar('China.IB')
   ...:
```

现在我们拥有了一个日历对象，来看看它是啥？

```
In [3]: print cal
China inter bank market calendar
```

Ok！它是中国银行间市场交易日历。

让我们测试一下它的功能。2014 年 10 月 11 日是十一以后的第一个周六，按照国务院调休的规定，那天是开市的。所以我们来看一下它是否了解这个信息？

```
In [4]: testDate = Date(2014, 10, 11)
   ...: cal.isBizDay(testDate)
   ...:
Out[5]: True
```

恩，看来它做的不错。

但是不是所有的交易场所都调休，比如上海证券交易所还是正常休市的，那怎么办呢？没问题，我们可以构造一个不同的日历：

```
In [6]: cal2 = Calendar('China.SSE')
   ...: print cal2
   ...:
Shanghai stock exchange calendar
```

那天应该不是交易所的交易日：

```
In [8]: cal2.isBizDay(testDate)
Out[8]: False
```

成功达到目的。

通过 CAL，我们还可以按照不同的方法合并两个不同的工作日历。比如合并两个日历中的工作日，然后测试一下这个合并是否有效

```
In [9]: joinCal = JointCalendar(cal, cal2, CalendarJoinRule.JoinBizDays)
   ...: joinCal.isBizDay(testDate)
   ...:
Out[10]: True
```

是的，工作的很好。我们也可以合并日历中的节假日：

```
In [11]: joinCal = JointCalendar(cal, cal2, CalendarJoinRule.JoinHolidays)
   ....: joinCal.isBizDay(testDate)
   ....:
Out[12]: False
```

同样准确无误。

注解: 这里我们只展示了两个Calendar 的合并。但是实际上还可以继续复合，理论上合并的Calendar 个数是没有上限的。

让我们来看看银行间市场 2014 年 10 月有哪些节假日吧：

```
In [13]: cal.holDatesList(Date(2014,10,1), Date(2014,10,31))
Out[13]:
[Date(2014,10,1),
 Date(2014,10,2),
 Date(2014,10,3),
 Date(2014,10,4),
 Date(2014,10,5),
 Date(2014,10,6),
 Date(2014,10,7),
 Date(2014,10,12),
 Date(2014,10,18),
 Date(2014,10,19),
 Date(2014,10,25),
 Date(2014,10,26)]
```

又有那些工作日呢？

```
In [14]: cal.bizDatesList(Date(2014,10,1), Date(2014,10,31))
Out[14]:
[Date(2014,10,8),
 Date(2014,10,9),
 Date(2014,10,10),
 Date(2014,10,11),
 Date(2014,10,13),
 Date(2014,10,14),
```

```
Date(2014,10,15),
Date(2014,10,16),
Date(2014,10,17),
Date(2014,10,20),
Date(2014,10,21),
Date(2014,10,22),
Date(2014,10,23),
Date(2014,10,24),
Date(2014,10,27),
Date(2014,10,28),
Date(2014,10,29),
Date(2014,10,30),
Date(2014,10,31)]
```

可以注意到输出的时候日期区间包含头，但是不包含尾。

当我们发现需要在一个已知的日历中增加或者删除节假日的时候，可以这么做：

```
In [15]: cal.addHoliday(testDate)
   ....: cal.isBizDay(testDate)
   ....:
Out[16]: False
```

```
In [17]: cal.removeHoliday(testDate)
   ....: cal.isBizDay(testDate)
   ....:
Out[18]: True
```

警告: 虽然日历中节假日的添加和删除似乎是针对某个特定的日历对象，但实际上这一改动是全局的。任何一个日历对象本身的变动，都会影响所有的与他同名的日历对象：

```
In [19]: cal1 = Calendar('China.IB')
   ....: testDate = Date(2015, 2, 6)
   ....: print cal1.isBizDay(testDate)
   ....:
True

In [22]: cal1.addHoliday(testDate)
   ....: print cal1.isBizDay(testDate)
   ....:
False
```

现在，让我们创建一个新的同名的日历:

```
In [24]: cal2 = Calendar('China.IB')
   ....: print cal1.isBizDay(testDate)
   ....:
False
```

新的日历与旧的同名日历有相同的节假日安排。

现在让我们来回答之前提出的最后两个问题：

- 从今天开始往后推 3 个月，然后希望那天是个工作日，如果不是的话顺延，这个该怎么做？
- 今天不是工作日，我要向前调整至最近的一个工作日该怎么做？

先来第一个问题，我们取定今天是 2014 年 7 月 11 日，加上 3 个月以后是 2014 年 10 月 11 日。下面代码展示了不同的日历下的不同结果：

```
In [26]: referenceDate = Date(2014, 7, 11)
   ....: period = Period('3M')
   ....: cal1 = Calendar('China.IB')
   ....: resDate = cal1.advanceDate(referenceDate, period, BizDayConvention.Following)
   ....: print 'Inter bank: ' + str(resDate)
   ....: cal2 = Calendar('China.SSE')
   ....: resDate = cal2.advanceDate(referenceDate, period, BizDayConvention.Following)
   ....: print 'Shanghai Stock Exchange: ' + str(resDate)
   ....:
Inter bank: 2014-10-11
Shanghai Stock Exchange: 2014-10-13
```

现阶段 CAL 中支持的时间单位类型有：

- Y， 日历年
- M， 日历月
- W， 日历周

- D ， 日历日
- B ， 工作日

再来看第二个问题，2014 年 10 月 12 日是休息日，它的前一个工作日是什么时候呢？

```
In [34]: referenceDate = Date(2014, 10, 12)
   ....: resDate = cal.adjustDate(referenceDate, BizDayConvention.Preceding)
   ....: print 'Inter bank: ' + str(resDate)
   ....: resDate = cal2.adjustDate(referenceDate, BizDayConvention.Preceding)
   ....: print 'Shanghai Stock Exchange: ' + str(resDate)
   ....:
Inter bank: 2014-10-11
Shanghai Stock Exchange: 2014-10-10
```

17.2.2 接口

class Calendars.Calendar

___init___(self, holCenter)
构造日历对象，holCenter 为具体的日历字符串（不区别大小写）:

- CHINA.SSE
- CHINA.IB
- UNITEDSTATES.GOVERNMENTBOND
- UNITEDSTATES.NERC
- UNITEDSTATES.NYSE
- NULLCALENDAR
- TARGET
- JAPAN
- UNITEDKINGDOM
- NULL

参数 holCenter (str) – 日历描述字符串

返回 日历对象

返回类型 Calendars.Calendar

___str___(self)
返回日历的字符串表示

isHoliday(self, testDate)
判断某个日期是不是节假日

参数 testDate (Dates.Date) – 待测试日期

返回 测试结果

返回类型 bool

isBizDay(self, testDate)
判断某个日期是不是工作日

参数 testDate (Dates.Date) – 待测试日期

返回 测试结果

返回类型 bool

isMonthEnd(self, testDate)
判断某个日期是不是月末

参数 testDate (Dates.Date) – 待测试日期

返回 测试结果

返回类型 bool

bizDatesList(self, startDate, endDate)
生成指定区间内的工作日列表

参数

- startDate (Dates.Date) – 参考时间段起始
- endDate (Dates.Date) – 参考时间段结束

返回 日期列表

返回类型 list

holDatesList(self, startDate, endDate)
生成指定区间内的节假日列表

参数

- startDate (Dates.Date) – 参考时间段起始
- endDate (Dates.Date) – 参考时间段结束

返回 日期列表

返回类型 list

addHoliday(self, newHolDate)
向日历中添加新的节假日

参数 newHolDate (Dates.Date) – 新节假日

removeHoliday(self, targetHolDate)
从日历中删除节假日

参数 targetHolDate (Dates.Date) – 旧节假日

advanceDate(self, todayDate, period, convention, endOfMonth = False)
在指定日历和惯例下，向前递推一个时间段

参数

- todayDate (Dates.Date) – 当前日期
- period (Periods.Period) – 时间段
- convention (BizDayConvention) – 工作日惯例
- endOfMonth (bool) – 是否遵从月末惯例

返回 日期

返回类型 Dates.Date

adjustDate(self, todayDate, convention)
按照指定日历和惯例，调整至最近的工作日

参数

- todayDate (Dates.Date) – 当前日期
- convention (BizDayConvention) – 工作日惯例

返回 日期

返回类型 Dates.Date

class Calendars.JointCalendar(Calendar)

___init___(self, cal1, cal2, rule = CalendarJoinRule.JoinHolidays)
通过两个存在的日历按照一定规则构造新的日历对象

参数

- cal1 (Calendars.Calendar) – 日历 1
- cal2 (Calendars.Calendar) – 日历 2
- rule (CalendarJoinRule) – 合并方法

返回 合并日历

返回类型 Calendar.JointCalendar

17.3 天数计数惯例

金融世界里经常会探讨两个量的比较。比如我们想比较国债收益率 4.6%，与同业存款收益率 4.58%，到底哪个更高？这里面单纯比较这两个数据大小是没有意义的，因为这里涉及到不同天数计数惯例的问题：国债使用 Actual/Actual 而同业存款可能使用 Actual/360。

除此以为，我们还经常面对一个问题：给定两个日期，这两个日期间的距离是多远？

由此我们需要抽象DayCounter 对象来表达这一概念。

17.3.1 CAL 的天数计数对象

CAL 中定义了一个一般的DayCounter 类型去表达天数计数惯例这一概念。它的构造函数引入一个字符串表述，然后生成对应的对象：

```
In [1]: from CAL.PyCAL import *
   ...: dc1 = DayCounter('Actual/Actual (ISMA)')
   ...: print dc1
   ...:
Actual/Actual (ISMA)
```

让我们来回答之前提出的第一个问题：如果同样是 3 个月，从 2014 年 10 月 21 日到 2015 年 1 月 21 日，国债收益率和同业存款到底哪个高？

首先让我们假设同业存款的天数计数惯例是 Actual/360, 那么：

```
In [4]: dc2 = DayCounter('Actual/360')
   ...: print dc2
   ...:
Actual/360
```

假设我们初始投资 100 元，让我们看按照单利，到底哪个收益率更高？

```
In [6]: # Invest in treasury bond
   ...: startDate = Date(2014, 10, 21)
   ...: endDate = Date(2015, 1, 21)
   ...: portfolio1 = 100.0 * (1.0 + dc1.yearFraction(startDate, endDate) * 0.046)
   ...: print '%.4f' % portfolio1
   ...:
101.1500
```

再看看同业存款：

```
In [11]: # Invest in deposit
   ....: portfolio2 = 100.0 * (1.0 + dc2.yearFraction(startDate, endDate) * 0.0458)
   ....: print '%.4f' % portfolio2
   ....:
101.1704
```

我们看到，虽然同业存款收益率看着比国债低，但是由于计息方式的不同，实际在以上假设下，同业存款是更加划算的投资！

除此之外，由于天数惯例种类繁多，我们设计了一些特殊类。这些类专注于某一种类别的天数计数惯例，这样减轻用户输入时候的记忆负担。例如：

```
In [14]: dc1 = DayCounter('Actual/Actual (ISMA)')
   ....: dc2 = ActualActual(ActualActual.ISMA)
```

```
....: print dc1
....: print dc2
....:
Actual/Actual (ISMA)
Actual/Actual (ISMA)
```

17.3.2 接口

class DayCounters.DayCounter

___init___(self, dc)

通过字符串描述构造计数惯例对象，现阶段支持的字符串描述包括：

- ACTUAL/365 (FIXED)
- ACTUAL/365 (NO LEAP
- ACTUAL/ACTUAL (ISMA)
- ACTUAL/360
- ACTUAL/ACTUAL (NO LEAP)
- SIMPLE

参数 dc (str) – 字符串描述，例如'Actual/365 (Fixed)'

返回 计数惯例对象

返回类型 DayCounters.DayCounter

yearFraction(self, startDate, endDate, referenceStartDate = None, referenceEndDate = None)

计算 Day counte fraction (DCF)

参数

- startDate (Dates.Date) – 起始日
- endDate (Dates.Date) – 结束日
- referenceStartDate (Dates.Date) – 参考起始日
- referenceEndDate (Dates.Date) – 参考结束日

返回 DCF

返回类型 float

___str___(self)

返回 字符串描述

返回类型 str

class DayCounters.Actual360(DayCounter)

___init___(self)
构造 Actual/360 天数计数惯例

返回 天数计数惯例对象

返回类型 DayCounters.Actual/360

class DayCounters.Actual365Fixed(DayCounter)

___init___(self)
构造 Actual/365 (Fixed) 天数计数惯例

返回 天数计数惯例对象

返回类型 DayCounters.Actual365Fixed

class DayCounters.ActualActual(DayCounter)

AFB

Bond

Euro

Historical

ISDA

ISMA

___init___(self, convention = ActualActual.ISMA)

根据不同的地区惯例构建 Actual/Actual

返回 天数计数惯例对象

返回类型 DayCounters.ActualActual

class DayCounters.Actual365NoLeap(DayCounter)

___init___(self)
构造 Actual/365 (No Leap) 天数计数惯例

返回 天数计数惯例对象

返回类型 DayCounters.Actual365NoLeap

class DayCounters.ActualActualNoLeap(DayCounter)

___init___(self)
构造 Actual/Actual (No Leap) 天数计数惯例

返回 天数计数惯例对象

返回类型 DayCounters.ActualActualNoLeap

class DayCounters.Thirty360(DayCounter)

BondBasis

EurobondBasis

European

Italian

USA

___init___(self, convention = Thirty360.BondBasis)
构造 30/360 天数计数惯例

返回 天数计数惯例对象

返回类型 DayCounters.Thirty360

17.4 时间长度

日期Date 扮演了生活中时间点的角色，但是这还不够。我们还需要时间长度的概念，比如说：

- 什么是 1 周？
- 7 天和 1 周有区别吗？
- 从今天开始向后移动 3 个月，这是什么意思？

CAL 通过定义自己的Period 类型来解决这些问题。

17.4.1 CAL 时间长度

CAL 的Period 类型主要支持如下的几种时间周期：D（日），W（周），M（月），Y（年）。一般来讲一个合法的Period 字符串符合这样的形式：'XD','XW', XM' 或者'XY'，这里 X 是一个整数：

> **警告:** 现在的 CAL 中' 日' 是工作日的概念，而不是日历日。

```
In [1]: from CAL.PyCAL import *
   ...: period = Period('3D')
   ...: print period
   ...:
3D
```

事实上 CAL 还可以理解更复杂一些的复合形式：

```
In [4]: period = Period('1Y3M')
   ...: print period
   ...: print period.units()
   ...: print period.length()
   ...:
1Y3M
2
15
```

这里Period 的 units 方法返回这个时间长度的单位，2 表示是'M'（月）。length 返回为 15，表示总长度是 15 个月。符合我们的预期。

关于使用Period 来完成时间推移的方法，请见相关页面：Date，Calendar。

17.4.2 接口

class Periods.Period

___init___(self, perStr)
通过时间字符串构造对象

参数 perStr (str) – 时间长度字符串

返回 时间长度对象

返回类型 Periods.Period

___str___(self)
返回对象的字符串表现形式

返回 时间长度字符串

返回类型 str

length(self)
长度单位个数 (例如 '3M', 则返回 3)

返回 长度单位个数

返回类型 int

units(self)
长度单位，例如 Days，Months，Years

返回 长度单位

返回类型 TimeUnit

PeriodFromFrequency(frequency)
从频率说明构建时间长度：例如 Frequency.Annual -> '1Y'

参数 frequency (Frequency) – 时间频率

返回 时间长度对象

返回类型 Periods.Period

17.5 日程

我们经常要按照一定规则产生一组日期序列，比如以下的语言描述：

- 从 2010 年 10 月 21 日开始，到 2014 年 10 月 21 日结束，每三个月，产生一组时间序列。如果遇到中国银行间市场休息日则顺延至下一工作日

让我们用 CAL 中的Schedule 类型来解决这个问题。

17.5.1 CAL 日程

我们就按照上面的问题来生成一个日程表：

```
In [1]: from CAL.PyCAL import *
   ...: effectiveDate = Date(2010, 10, 21)
   ...: terminationDate = Date(2014, 10, 21)
   ...: tenor = Period('3M')
   ...: calendar = Calendar('China.IB')
   ...: convention = BizDayConvention.Following
   ...: terminationDateConvention = BizDayConvention.Following
   ...: rule = DateGeneration.Forward
   ...: endOfMonth = False
   ...: sch = Schedule(effectiveDate,
   ...:                terminationDate,
   ...:                tenor,
   ...:                calendar,
   ...:                convention,
   ...:                terminationDateConvention,
   ...:                rule,
   ...:                endOfMonth)
   ...:
```

注解: 我们这里实际上输入类似于 calendar 这样的参数的时候可以直接使用字符串。事实上，在 CAL，所以需要输入类似Calendar, DayCounter, Period 类型的时候，都可以直接输入字符串。

让我们看一下这个Schedule 具体包含哪些日期？

```
In [11]: for date in sch:
   ....:     print repr(date)
   ....:
Date(2010,10,21)
Date(2011,1,21)
```

```
Date(2011,4,21)
Date(2011,7,21)
Date(2011,10,21)
Date(2012,1,21)
Date(2012,4,23)
Date(2012,7,23)
Date(2012,10,22)
Date(2013,1,21)
Date(2013,4,22)
Date(2013,7,22)
Date(2013,10,21)
Date(2014,1,21)
Date(2014,4,21)
Date(2014,7,21)
Date(2014,10,21)
```

确实是每 3 个月一个间隔，并且避开了节假日。

我们也可以很方便的用下标的形式访问Schedule 中的成员：

```
In [12]: print sch[3]
2011-07-21
```

17.5.2 接口

class Schedules.Schedule

___init___(self, effectiveDate, terminationDate, tenor, calendar, convention, terminationDateConvention, rule, endOfMonth, firstDate = None, nextToLastDate = None)

生成日程对象

参数

- effectiveDate (Dates.Date) – 起始有效日
- terminationDate (Dates.Date) – 终止日
- tenor (Periods.Period) – 间隔
- calendar (Calendars.Calendar) – 工作日历
- convention (BizDayConvention) – 工作日惯例
- terminationDateConvention (BizDayConvention) – 终止日工作日惯例
- rule (DateGeneration) – 时间点产生方法
- endOfMonth (bool) – 是否使用月末惯例
- firstDate (Dates.Date) – 第一个规则日

- nextToLastDate (Dates.Date) – 最后一个规则日

返回 日程对象

返回类型 Schedules.Schedule

size(self)

日程总长度 (日期期数)

返回 长度

返回类型 int

date(self, index)

第 index 期的日期

参数 index (int) – 下标

返回 目标日期

返回类型 Dates.Date

isRegular(self, index)

第 index 期是否为规则

参数 index (int) – 下标

返回 判断结果

返回类型 bool

___getitem___(self, index)

拿到第 index 期的日期

参数 index (int) – 下标

返回 目标日期

返回类型 Dates.Date

class Schedules.ScheduleCreatedFromDates(Schedule)

___init___(self, dates, calendar, rollingConvention)

由 Date 的序列产生 schedule

参数

- dates (list) – Date 序列
- calendar (Calendars.Calendar) – 日历
- rollingConvention (BizDayConvention) – 工作日惯例

返回 日程对象

返回类型 Schedules.ScheduleCreatedFromDates

class Schedules.CallabilitySchedule

___init___(self)
构造行权日程

返回 行权日程

返回类型 Schedules.CallabilitySchedule

addCallability(self, callability)
增加行权日

参数 callability (Exercises.Callability) –

class Schedules.DividendSchedule

___init___(self)
构造股息日程

返回 股息日程对象

返回类型 Schedules.DividendSchedule

addDividend(self, dividend)

CHAPTER 18

指数

18.1 利率指数

利率相关交易产品中，如果涉及浮动利率支付的，总是会涉及与某个市场利率指数挂钩，这个概念在 CAL 中以 InterestRateIndex 实现。

18.1.1 Ibor 利率指数

最常见的利率指数是 Ibor 利率指数（I nter B ank O ffering R ate，银行间市场拆出利率)。一个典型的 Ibor 利率指数包含以下几个基本要素：

- 利率期限长度，例如 3M ， 指标的利率是一个 3 个月的利率指数。
- 重置期限，例如 2D，也就是指重置日为利率估值日（也就是期限起始日）之前几个工作日。
- 工作日历，确定所有日期，所使用的工作日历。
- 工作日惯例，节假日的处理方法。
- 天数计数惯例，计算利息时候使用的天数计数方法。

这里让我们使用最常用的 3 个月期限 Shibor 利率指数（Sh anghai I nter B ank O ffering R ate，上海银行间市场同业拆出利率）为例：

- 利率期限长度：3M
- 重置期限: 1D
- 工作日历: China.IB ， 中国银行间市场工作日历
- 工作日惯例：Following ， 下一工作日
- 天数计数惯例：Actual/360

18.1.2 重置规则

1. 确定重置日

最常见的确定重置日的方法是通过估值日向前推 n 个工作日的方法，这里的 n 等于重置期限。

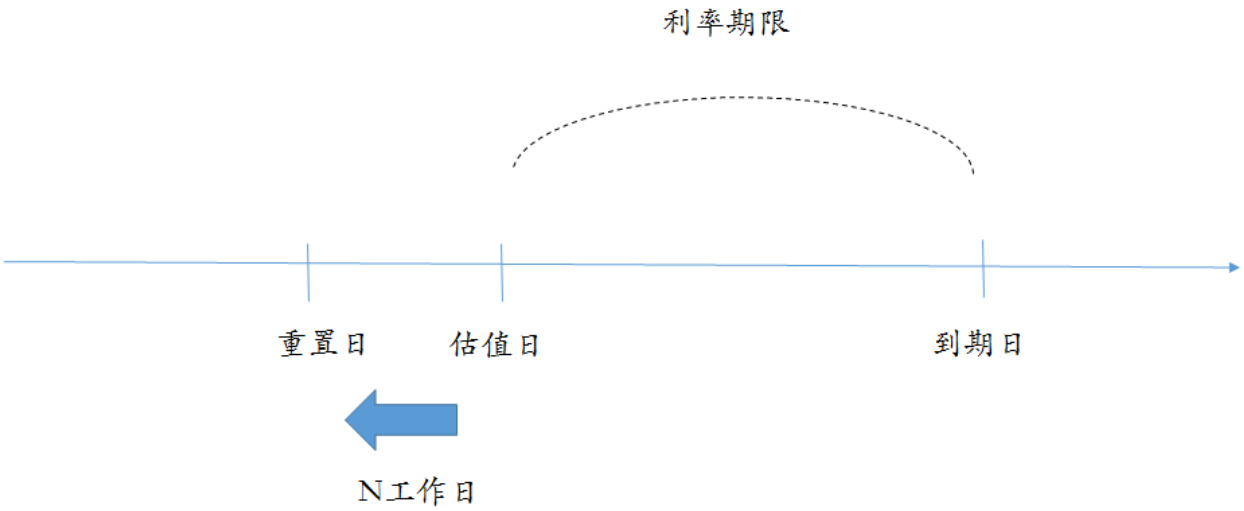


图 18.1: 重置日确定规则

```
In [1]: index = Shibor('3M')
```

假设估值日为 2015 年 2 月 13 日，则按照重置日规则：

```
In [2]: valueDate = Date(2015,2,13)
   ...: fixingDate = index.fixingDate(valueDate)
   ...: print fixingDate
   ...:
2015-02-12
```

为 2015 年 2 月 12 日。

如果估值日为 2015 年 2 月 25 日，则重置日为：

```
In [5]: valueDate = Date(2015,2,25)
   ...: fixingDate = index.fixingDate(valueDate)
   ...: print fixingDate
   ...:
2015-02-17
```

为 2015 年 2 月 17 日（春节前最后一个交易日）。

2. 确定重置水平

在重置日当天，会根据官方发布的利率指数确定重置水平。这里面有个问题，如果重置日是在未来的，这个重置水平怎么确定呢？

这里面我们区分未来的重置日和过去的重置日，分割点为 **今天**（就是指从 EvaluationDate 返回的值）。如果重置日大于或者等于今天，重置水平会根据收益率曲线去估计；如果重置日在今天以前，则会在历史重置水平去获取（这里的历史重置水平需要用户自己通过 API 进行维护）。

```
In [8]: cal = Calendar('China.IB')
   ...: eval = cal.adjustDate(Date.todaysDate(), BizDayConvention.Following)
   ...: SetEvaluationDate(eval)
   ...: yc = FlatForward(EvaluationDate(), 0.05, 'Actual/360')
   ...: index = Shibor('3M', yc)
   ...:
```

如果重置日在未来：

```
In [13]:
   ....: fixingDate = cal.advanceDate(EvaluationDate(), '3M', BizDayConvention.Following)
   ....: index.fixing(fixingDate)
   ....:
Out[14]: 0.050317307618302616
```

但是如果重置日发生在过去：

```
In [15]: fixingDate = Date(2015,1,4)
   ....: try:
   ....:     index.fixing(fixingDate)
   ....: except RuntimeError as e:
   ....:     print e
   ....:
Missing Shibor3M Actual/360 fixing for January 4th, 2015
```

用户需要自己添加历史重置水平：

```
In [17]: index.addFixing(fixingDate, 0.05)
   ....: index.fixing(fixingDate)
   ....:
Out[18]: 0.05
```

警告: 记住在利率指数对象中添加历史重置水平，这一改动是全局的。任何一个利率指数对象本身的变动，都会影响所有的与他同名并且同期限的利率指数对象：

```
In [19]: newIndex = Shibor('3M')
   ....: newIndex.fixing(fixingDate)
   ....:
Out[20]: 0.05
```

新的利率指数对象带有原先添加的历史重置设置。

18.1.3 接口

class Index.InterestRateIndex(Index.Index)

familyName(self)
返回利率指数所属的指数类型，例如:'Shibor'

tenor(self)
返回标的利率期限长度，例如 :'3M'

fixingDays(self)
返回重置日长度

fixingDate(self, valueDate)
根据估值日返回重置日

dayCounter(self)
返回标的利率的天数计数惯例

maturityDate(self, valueDate)
根据估值日返回到期日

valueDate(self, fixingDate)
根据重置日返回估值日

class Index.IborIndexProxy(Index.InterestRateIndex)

businessDayConvention(self)
返回标的 ibor 利率的工作日惯例

endOfMonth(self)
返回标的 ibor 利率的月末惯例

forwardingTermStructure(self)
返回标的 ibor 利率指数对应的利率曲线

class Index.IborIndex(Index.IborIndexProxy)

___init___(self, familyName, tenor, fixingDays, calendar, convention, endOfMonth, dayCounter, yieldCurve, currency = None)
构造 Ibor 类型利率指数

参数

- familyName (str) – 利率指数所属的指数类型
- tenor (Periods.Period) – 标的利率期限
- fixingDays (int) – 重置日期限
- calendar (Calendars.Calendar) – 工作日日历
- convention (BizDayConvention) – 工作日惯例
- endOfMonth (boolean) – 月末惯例
- dayCounter (DayCounters.DayCounter) – 天数计数惯例

- yieldCurve (YieldCurve.YieldCurveProxy) – 利率曲线
- currency (str) – 货币

class Index.Shibor(IborIndex)

___init___(self, tenor, yieldCurve = None)
构造 Shibor 类型利率指数

参数

- tenor (Periods.Period) – 标的利率期限
- yieldCurve (YieldCurve.YieldCurveProxy) – 利率曲线

class Index.RepoChina(IborIndex)

___init___(self, tenor, yieldCurve = None)
构造 RepoChina 类型利率指数

参数

- tenor (Periods.Period) – 标的利率期限
- yieldCurve (YieldCurve.YieldCurveProxy) – 利率曲线

class Index.USDLibor(IborIndex)

___init___(self, tenor, yieldCurve = None)
构造 USDLibor 类型利率指数

参数

- tenor (Periods.Period) – 标的利率期限
- yieldCurve (YieldCurve.YieldCurveProxy) – 利率曲线

class Index.Euribor(IborIndex)

___init___(self, tenor, yieldCurve = None)
构造 Euribor 类型利率指数

参数

- tenor (Periods.Period) – 标的利率期限
- yieldCurve (YieldCurve.YieldCurveProxy) – 利率曲线

18.2 接口

class Index.Index

name(self)
返回该指数的名称

fixingCalendar(self)
返回用于重置日确定的日历

isValidFixingDate(self, date)
测试某日是否是有效的重置日

参数 date (Dates.Date) – 待测试日期

fixing(self, fixingDate, forecastTodaysFixing = False)
获取指定日的重置水平

参数

- fixingDate (Dates.Date) – 指定重置日
- forecastTodaysFixing (boolean) – 如果重置日为全局估值日，是否使用预测水平？

返回 重置水平

返回类型 float

addFixing(self, fixingDate, fixing)
设定指定日的重置水平

参数

- fixingDate (Dates.Date) – 重置日
- fixing (float) – 重置水平

addFixings(self, fixingDates, fixings)
设定一系列指定日的重置水平

参数

- fixingDates (list) – 重置日序列
- fixings (list) – 重置水平序列

CHAPTER 19

固定收益

19.1 债券

19.1.1 CAL 债券

CAL 定义了一组接口用于描述债券通用的性质，比如到期时间，净价等。这一组接口被包含于基类Bond 中。具体类例如CTBZeroBond 直接派生自Bond ， 然后定义一些特有的性质。

CTBZeroBond

CTBZeroBond 用于建模中国财政部发行的贴现国债，交易代码例如：139903。根据 [40] 的规定，中国的零息国债有应计利息的概念，定义为从发行日到到期日之间折价的比例，具体算法请见：[41]

，计算式如下：

$$\text{AI} = \text{DCF}(\frac{Settlement - Issue}{Maturity - Issue}) \times (Redemption - IssuePrice) \tag{19.1}$$

注解: 这里的 DCF 是某种具体的天数计数惯例，中国国债一般使用 Actual/Actual (ISMA)。

让我们先来试试看这个类的用法，先定义一些基本特征：

表 19.1: 139903 债券定义

代码	139903.IB
发行日	2013 年 5 月 13 日
到期日	2014 年 2 月 10 日
发行价格	97.9030
日历	China.IB
天数计数	Actual/Actual (ISMA)

```
In [1]: from CAL.PyCAL import *
   ...: # Infor about 139903.IB
   ...: today = Date(2013, 12, 2)
   ...: SetEvaluationDate(today)
   ...: settlementDays = 1
   ...: faceAmount = 100.0
   ...: issuePrice = 97.9030
   ...: issueDate = Date(2013, 5, 13)
   ...: maturityDate = Date(2014, 2, 10)
   ...: calendar = Calendar('China.IB')
   ...: dayCounter = DayCounter('Actual/Actual (ISMA)')
   ...:
```

好了，可以构造零息债券了：

```
In [12]: testBond = CTBZeroBond(settlementDays,
   ....:                        faceAmount,
   ....:                        issuePrice,
   ....:                        issueDate,
   ....:                        maturityDate,
   ....:                        calendar,
   ....:                        dayCounter)
   ....:
```

首先测试一下基本的信息：

```
In [13]: print testBond.issueDate()
   ....: print testBond.maturityDate()
   ....: print testBond.dayCounter()
   ....:
2013-05-13
2014-02-10
Actual/Actual (ISMA)
```

下面我测试最关键的部分，我们的CTBZeroBond 对象是否可以正确的计算应计利息：

表 19.2: 中债登结算数据 (139903.IB)

应计利息	1.5670
净价	97.5988
到期收益率	0.0445
复利方法	Simple
复利频率	Annual

```
In [16]: compounded = Compounding.Simple
   ....: frequency = Frequency.Annual
   ....: cleanPrice = 97.5988
```

```
....: calculatedYield = testBond.yieldFromCleanPrice(cleanPrice,
....:                                                dayCounter,
....:                                                compounded,
....:                                                frequency)
....: print 'Yield: %.4f' % (calculatedYield * 100)
....: print 'Accrued: %.4f' % testBond.accruedAmount()
....:
Yield: 4.4500
Accrued: 1.5670
```

很好，完全吻合！

FixedRateBond

更常见的债券是有规则付息日期的固定利率债券。例如如下的 120014.IB:

表 19.3: 120014 债券定义

代码	120014.IB
发行日期	2012 年 8 月 16 日
起息日	2012 年 8 月 16 日
到期日	2017 年 8 月 16 日
付息频率	Annual
票息	2.95%
日历	Null Calendar
天数计数惯例	Actual/Actual (ISMA)

警告: 这里我们使用了 NullCalendar 。真实债券付息实际上按照的 China.IB ， 但是计算诸如到期收益率等指标时使用的是 NullCalendar 。这里为了计算需要，我们选用后者。

让我们来看，如何把它定义成 CAL 的FixedRateBond, 同样的先定义一些它的属性：

注解: 每次都输入这么一堆特征，真头大！但是有什么办法呢，这就是金融的世界！看看你的贷款合同不也是这样麻烦吗？

```
In [22]: # Infor about 120014.IB
....: startDate = Date(2012, 8, 16)
....: maturityDate = Date(2017, 8, 16)
....: frequency = Frequency.Annual
....: faceAmount = 100.0
....: coupon = 0.0295
....: accrualCalendar = Calendar('NullCalendar')
....: paymentCalendar = Calendar('NullCalendar')
....: convention = BizDayConvention.Unadjusted
```

```
....: dayCounter = DayCounter('Actual/Actual (ISMA)')
....:
```

终于搞定了输入参数，下面让我们去构造一个真正的FixedRateBond 。从构造Schedule 开始：

```
In [32]: # Set up schedule
....: schedule = Schedule(startDate,
....:                     maturityDate,
....:                     Period.PeriodFromFrequency(frequency),
....:                     accrualCalendar,
....:                     convention,
....:                     convention,
....:                     DateGeneration.Backward,
....:                     False)
....:
```

千辛万苦，胜利就在眼前：

```
In [34]: # Set up the bond
....: testBond = FixedRateBond(settlementDays,
....:                          faceAmount,
....:                          schedule,
....:                          coupon,
....:                          dayCounter,
....:                          convention)
....:
```

大功告成！

注解: CAL 的哲学之一：给用户尽可能多的自由度，方便做二次开发。但是这样的处理也使得很多必须为 API 提供相当多的细节，增加了输入的复杂度。但是不要着急，我们自己也意识到了这一苦难，所以我们的哲学也不唯一:) 在Bond Functions 你就可能看到另一种。

和之前零息债一样，首先看一下基本信息：

```
In [36]: print testBond.startDate()
....: print testBond.frequency()
....: print testBond.maturityDate()
....: print testBond.dayCounter()
....:
2012-08-16
1
2017-08-16
Actual/Actual (ISMA)
```

接着是一些常规的计算，算法同样可见 [40] ， [41] ， 基本的公式，比如到期收益率：

$$PV = \frac{C/f}{(1+y/f)^d} + \frac{C/f}{(1+y/f)^{d+1}} \cdots + \frac{C/f}{(1+y/f)^{d+n-1}} \tag{19.2}$$

其中：

$$
\begin{aligned}
PV &= \text{债券全价} \\
C &= \text{票面利率} \\
f &= \text{年付息频率} \\
y &= \text{到期收益率} \\
d &= \text{结算日到下一付息日的 DCF} \\
n &= \text{结算日到到期日的付息次数}
\end{aligned}
$$

看我们能不能复制公开市场的结果：

表 19.4: 中债登结算数据 (120014.IB)

应计利息	0.881
净价	95.4326
全价	96.314
到期收益率	0.043074
复利方法	Compounded
复利频率	Annual

```
In [40]: cleanPrice = 95.4326
   ....: calculatedYield = testBond.yieldFromCleanPrice(cleanPrice,
   ....:                                                dayCounter,
   ....:                                                Compounding.Compounded,
   ....:                                                frequency)
   ....: print 'Yield: %.4f' % (calculatedYield*100)
   ....: calculatedCleanPrice = testBond.cleanPriceFromYield(
   ....:                        calculatedYield,
   ....:                        dayCounter,
   ....:                        Compounding.Compounded,
   ....:                        frequency)
   ....: print 'Clean Price: %.4f' % calculatedCleanPrice
   ....: accrue = testBond.accruedAmount()
   ....: print 'Accrue: %.3f' % accrue
   ....: print 'Dirty Price: %.3f' % (calculatedCleanPrice + accrue)
   ....:
Yield: 4.3074
Clean Price: 95.4326
Accrue: 0.881
Dirty Price: 96.314
```

债券风险指标

前面两节，我们介绍了债券中最常见的零息债券以及固定利率债券，也展示了价格指标的简单计算 (净价，全价，应计利息，到期收益率)。但是，即使是这么简单的产品，我们上面的介绍仍没有涉及到一个方

面：**债券风险分析**。

债券风险分析当中，一个基本的问题，就是回答交易员：

- 如果市场利率向上跳动 1 个 BP，我的头寸会亏损多少？

这里有一个答案（当然并不是全部）：这可以由久期与凸性计算而来。这里我们无意于介绍关于 久期与凸性的具体定义，有兴趣的读者，请参考 [37] 。

让我做一些热身运动，仍然使用之前的例子，让我们假设到期收益率的变动情形为 10BP：

```
In [48]: up = testBond.cleanPriceFromYield(
   ....:                         calculatedYield + 0.001,
   ....:                         dayCounter,
   ....:                         Compounding.Compounded,
   ....:                         frequency)
   ....: down = testBond.cleanPriceFromYield(
   ....:                         calculatedYield - 0.001,
   ....:                         dayCounter,
   ....:                         Compounding.Compounded,
   ....:                         frequency)
   ....: print "Scenario(+10bp): %.4f" % up
   ....: print "Scenario(-10bp): %.4f" % down
   ....:
Scenario(+10bp): 95.1076
Scenario(-10bp): 95.7591
```

表 19.5: 利率假设情形

利率情形	+10bp	0bp	-10bp
净价	95.1076	95.4326	95.7590

```
In [52]: duration = BondAnalytic.duration(testBond,
   ....:                                  calculatedYield,
   ....:                                  dayCounter,
   ....:                                  Compounding.Compounded,
   ....:                                  frequency)
   ....: dirtyPrice = calculatedCleanPrice + testBond.accruedAmount()
   ....: # Using Duration approximation
   ....: up = calculatedCleanPrice - duration * dirtyPrice * 0.001
   ....: down = calculatedCleanPrice + duration * dirtyPrice * 0.001
   ....: print "Duration: %.4f" % duration
   ....: print "Scenario(+10bp): %.4f" % up
   ....: print "Scenario(-10bp): %.4f" % down
   ....:
Duration: 3.3821
Scenario(+10bp): 95.1069
Scenario(-10bp): 95.7583
```

用久期做逼近价格变化的结果：

表 19.6: 利率假设情形

利率情形	+10bp	0bp	-10bp
净价	95.1076	95.4326	95.7590
久期逼近	95.1069	——	95.7583

我们可以看到这个结果已经足够精确，误差在 0.01% 以下。但是我们可以做的更好，再如果多加一点凸性：

```
In [60]: convexity = BondAnalytic.convexity(testBond,
   ....:                                    calculatedYield,
   ....:                                    dayCounter,
   ....:                                    Compounding.Compounded,
   ....:                                    frequency)
   ....: up = calculatedCleanPrice - duration * dirtyPrice * 0.001 \
   ....:      + 0.5 * convexity * dirtyPrice * 0.001 * 0.001
   ....: down = calculatedCleanPrice + duration * dirtyPrice * 0.001 \
   ....:      + 0.5 * convexity * dirtyPrice * 0.001 * 0.001
   ....: print "Convexity: %.4f" % convexity
   ....: print "Scenario(+10bp): %.4f" % up
   ....: print "Scenario(-10bp): %.4f" % down
   ....:
Convexity: 15.0295
Scenario(+10bp): 95.1076
Scenario(-10bp): 95.7591
```

表 19.7: 利率假设情形

利率情形	+10bp	0bp	-10bp
净价	95.1076	95.4326	95.7590
久期逼近	95.1069	——	95.7583
凸性逼近	95.1076	——	95.7591

我们几乎可以得到 6 位有效数字精度的结果。

一个与久期比较类似的概念是 PV01 ：如果市场收益率曲线变动 1 个 bp，债券价格的波动。它们的区别仅在于基准不同，在大多数情况下，这两个值的区别不大 (数值意义下，久期乘以结算价值乘以 0.0001，约等于 PV01)：

- 久期是标的 到期收益率变化，债券价格的变化；
- PV01 是市场收益率曲线平移，债券价格的变化。

让我们用 2014 年 11 月 3 日的国债收益率曲线做个试验：

这里我们使用的 yc 是那天的国债收益率曲线。我们可以按照之前介绍的方法，做债券基于收益率曲线的估值计算。

```
In [66]: SetEvaluationDate(refDate)
   ....: bondEngine = DiscountingBondEngine(yc)
```

```
   ....: testBond.setPricingEngine(bondEngine)
   ....: print 'Clean: %.4f' % testBond.cleanPrice()
   ....:
Clean: 98.8157
```

```
In [70]: ytm =  testBond.yieldFromCleanPrice(testBond.cleanPrice(),
   ....:                                     dayCounter,
   ....:                                     Compounding.Compounded,
   ....:                                     frequency)
   ....: print 'Yield: %.4f' % ytm
   ....:
Yield: 0.0340
```

这里我们可以比较一下，PV01 与久期的差别：

```
In [72]: duration = BondAnalytic.duration(testBond,
   ....:                                  ytm,
   ....:                                  dayCounter,
   ....:                                  Compounding.Compounded,
   ....:                                  frequency)
   ....: pv01 = BondAnalytic.pv01(testBond,
   ....:                          yc,
   ....:                          dayCounter,
   ....:                          Compounding.Compounded,
   ....:                          frequency)
   ....: print 'Duration * Dirty: %.4f' % (duration * (testBond.cleanPrice() + testBond.accruedAmount()) * 0.0001)
   ....: print 'PV01: %.4f' % pv01
   ....:
Duration * Dirty: 0.0259
PV01: 0.0260
```

CallableFixedRateBond

不是所有的债券都像上面说的那么简单。很多债券会有内嵌的期权结构，或者给予发行者权力去以固定价格赎回债券（可赎回），或者给予投资者权力以固定价格将债券卖还给发行商（可回售）。

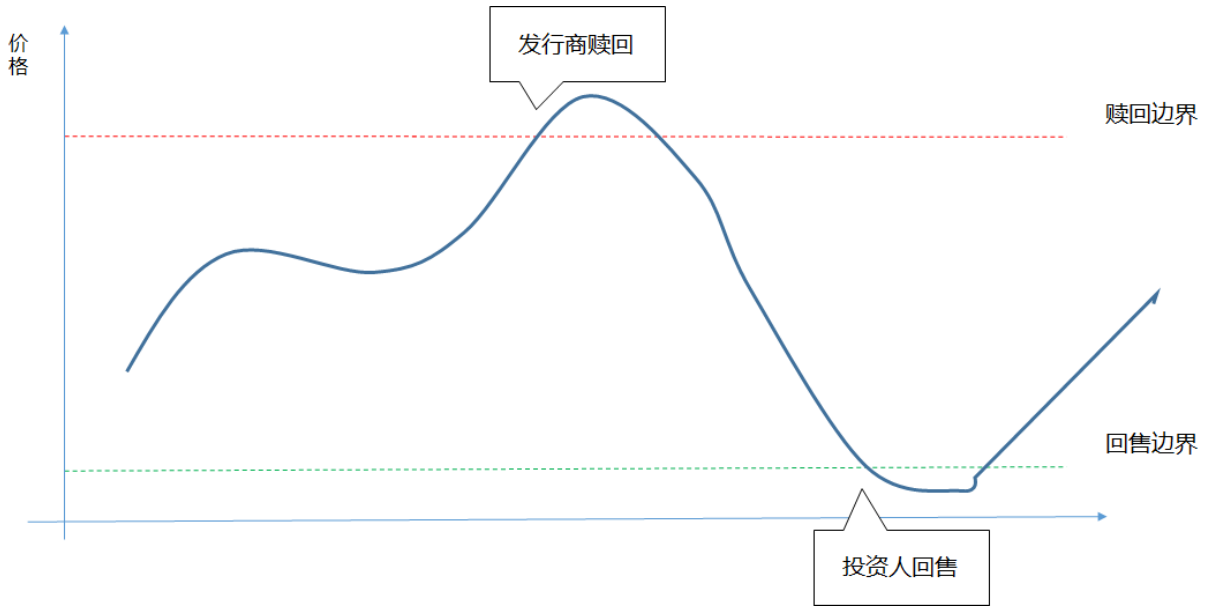


图 19.1: 可回售赎回债券示意图

在这种情况下，单一的收益率曲线建模已经不足以处理该类型债券估值的情形。这里我们不打算对于估值建模进行展开，读者可以参考债券定价算法 。

这里我们将专注于如何构造一个可回售赎回债券对象，这里我们选取的对象是国家开发银行 2005 年第三期金融债券，交易代码 050203.XIBE：

- 起息日期 2005 年 3 月 29 日
- 到期日期 2020 年 3 月 29 日
- 每年付息 1 次
- 票面利率为 4.67%
- 行权日为 2015 年 3 月 29 日
- 行权价为 100，按照净价赎回

1. 确定行权日以及行权价

在这个例子中，只有一个行权日：

```
In [76]: price = CallabilityPrice(100.0, CallabilityPrice.Clean)
   ....: callability = Callability(price, Callability.Put, Date(2015, 3, 29))
   ....: callabilitySchedule = CallabilitySchedule()
   ....: callabilitySchedule.addCallability(callability)
   ....:
```

2. 确定付息日程

这一步和大多数普通债券是完全一致的：

```
In [80]: schedule = Schedule(Date(2005, 3, 29),
   ....:                     Date(2020, 3, 29),
   ....:                     Period('1Y'),
   ....:                     'NullCalendar',
   ....:                     BizDayConvention.Unadjusted,
   ....:                     BizDayConvention.Unadjusted,
   ....:                     DateGeneration.Backward,
   ....:                     False)
   ....:
```

3. 组装成债券

```
In [81]: testBond = CallableFixedRateBond(1,
   ....:                                  100.0,
   ....:                                  schedule,
   ....:                                  0.0467,
   ....:                                  'Actual/Actual (ISMA)',
   ....:                                  callabilitySchedule)
   ....:
```

现在这个债券就可以用于一些特定的算法（这些算法适用于可回售债券），例如我们后面将提到的TreeCallableFixedRateBondEngine 。

ConvertibleFixedCouponBond

现金流分析

当我们手上有一个现成的债券的时候，可以很容易的获得关于这个债券的现金流分布描述：

```
In [82]: testBond.flowAnalysis()
Out[82]:
              AMOUNT  NOMINAL ACCRUAL_START_DATE ACCRUAL_END_DATE  \
PAYMENT_DATE
2006-03-29    4.6700 100.0000       2005-03-29      2006-03-29
2007-03-29    4.6700 100.0000       2006-03-29      2007-03-29
2008-03-29    4.6700 100.0000       2007-03-29      2008-03-29
2009-03-29    4.6700 100.0000       2008-03-29      2009-03-29
2010-03-29    4.6700 100.0000       2009-03-29      2010-03-29
2011-03-29    4.6700 100.0000       2010-03-29      2011-03-29
2012-03-29    4.6700 100.0000       2011-03-29      2012-03-29
2013-03-29    4.6700 100.0000       2012-03-29      2013-03-29
2014-03-29    4.6700 100.0000       2013-03-29      2014-03-29
2015-03-29    4.6700 100.0000       2014-03-29      2015-03-29
2016-03-29    4.6700 100.0000       2015-03-29      2016-03-29
2017-03-29    4.6700 100.0000       2016-03-29      2017-03-29
2018-03-29    4.6700 100.0000       2017-03-29      2018-03-29
2019-03-29    4.6700 100.0000       2018-03-29      2019-03-29
```

```
2020-03-29    4.6700 100.0000        2019-03-29       2020-03-29
2020-03-29  100.0000      #NA              #NA              #NA

            ACCRUAL_DAYS INDEX FIXING_DAYS FIXING_DATES INDEX_FIXING  \
PAYMENT_DATE
2006-03-29           365  #NA        #NA        #NA         #NA
2007-03-29           365  #NA        #NA        #NA         #NA
2008-03-29           366  #NA        #NA        #NA         #NA
2009-03-29           365  #NA        #NA        #NA         #NA
2010-03-29           365  #NA        #NA        #NA         #NA
2011-03-29           365  #NA        #NA        #NA         #NA
2012-03-29           366  #NA        #NA        #NA         #NA
2013-03-29           365  #NA        #NA        #NA         #NA
2014-03-29           365  #NA        #NA        #NA         #NA
2015-03-29           365  #NA        #NA        #NA         #NA
2016-03-29           366  #NA        #NA        #NA         #NA
2017-03-29           365  #NA        #NA        #NA         #NA
2018-03-29           365  #NA        #NA        #NA         #NA
2019-03-29           365  #NA        #NA        #NA         #NA
2020-03-29           366  #NA        #NA        #NA         #NA
2020-03-29           #NA  #NA         #NA         #NA         #NA

                    DAY_COUNTER ACCRUAL_PERIOD EFFECTIVE_RATE
PAYMENT_DATE
2006-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2007-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2008-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2009-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2010-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2011-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2012-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2013-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2014-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2015-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2016-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2017-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2018-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2019-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2020-03-29  Actual/Actual (ISMA)        1.0000         0.0467
2020-03-29                 #NA          #NA          #NA
```

19.1.2 接口

class Bonds.Bond(Instruments.Instrument)

nextCouponRate(self, refDate)
下一付息利率 (晚于 refDate)

参数 refDate (Dates.Date) – 基准日期

返回 下一付息日

返回类型 Dates.Date

cleanPriceFromYield(self, bondYield, dayCounter, compounding, freq, settlementDate = None)
根据债券的到期收益率计算净价

参数

- bondYield (float) – 债券到期收益率
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- compounding (Compounding) – 复利方法
- freq (Frequency) – 计息频率
- settlementDate (Dates.Date) – 结算日

返回 净价

返回类型 float

yieldFromCleanPrice(self, cleanPrice, dayCounter, compounding, freq, settlementDate = None)
根据债券净价计算债券到期收益率

参数

- cleanPrice (float) –
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- compounding (Compounding) – 复利方法
- freq (Frequency) – 计息频率
- settlementDate (Dates.Date) – 结算日

返回 到期收益率

返回类型 float

bondYield(self, dayCounter, compounding, freq)
计算债券到期收益率

参数

- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- compounding (Compounding) – 复利方法
- freq (Frequency) – 计息频率

返回 到期收益率

返回类型 float

accruedAmount(self, settlementDate = None)

计算应计利息

参数 settlementDate (Dates.Date) – 清算日

返回 应计利息

返回类型 float

cleanPrice(self)

计算净价

返回 净价

返回类型 float

dirtyPrice(self)

计算全价

返回 全价

返回类型 float

issueDate(self)

发行日

返回 发行日

返回类型 Dates.Date

startDate(self)

计息起始日

返回 计息起始日

返回类型 Dates.Date

maturityDate(self)

到期日

返回 到期日

返回类型 Dates.Date

settlementDate(self)

清算日

返回 清算日

返回类型 Dates.Date

settlementDays(self)

清算速度

返回 清算需要的天数

返回类型 int

cashflows(self)

现金流（仅对固定利率债券有效）

返回 现金流的列表

返回类型 list

settlementValue(self)

清算价值（清算日的 NPV）

返回 清算价值

返回类型 float

zSpread(self, targetCleanPrice, yieldCurve, dayCounter, compounding, frequency, settlementDate = None)

计算 Z-Spread（假设该债券已经配置了一个计算引擎，这个需要用户自己保证）

参数

- targetCleanPrice (float) – 目标净价
- yieldCurve (YieldCurve.YieldCurveProxy) – 目标收益率曲线
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- compounding (Compounding) – 复利方法
- freq (Frequency) – 计息频率

返回 Z-Spread

返回类型 float

cleanPriceFromZSpread(self, yieldCurve, zSpread, dc, compounding, frequency, settlementDate = None)

根据 Z-Spread 计算债券的净价（假设该债券已经配置了一个计算引擎，这个需要用户自己保证）

参数

- yieldCurve (YieldCurve.YieldCurveProxy) – 目标收益率曲线
- zSpread (float) – Z-Spraed
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- compounding (Compounding) – 复利方法
- freq (Frequency) – 计息频率

返回 净价

返回类型 float

NPV(self)

返回 NPV（Net Present Value）

返回类型 float

class Bonds.CTBZeroBond(Bonds.Bond)

___init___(self, settlementDays, faceAmount, issuePrice, issueDate, maturityDate, calendar = 'China.IB', dayCounter = 'Actual/Actual (ISMA)'))

构造中国财政部零息债券对象

参数

- settlementDays (int) – 清算速度
- faceAmount (float) – 面值
- issuePrice (float) – 发行价格
- issueDate (Dates.Date) – 发行日期
- maturityDate (Dates.Date) – 到期日
- calendar (Calendars.Calendar) – 日历
- dayCounter (DayCounters.DayCounter) – 天数计数惯例

返回 CTBZeroBond 对象

返回类型 Bonds.CTBZeroBond

dayCounter(self)

返回计息用的天数计数惯例 (仅用于计算应计利息)

返回 天数计数惯例

返回类型 DayCounters.DayCounter

ZeroCouponBond(Bonds.Bond):

Bonds.___init___(self, settlementDays, calendar, faceAmount, maturityDate, paymentConvention = BizDayConvention.Following, redemption = 100.0, issueDate = None)

构造常规零息利率债券

参数

- settlementDays (int) – 清算速度
- Calendars.Calendar – 工作日日历
- faceAmount (float) – 面值
- maturityDate (Dates.Date) – 到期日
- paymentConvention (BizDayConvention) – 付息工作日惯例
- redemption (float) – 到期偿付
- issueDate (Dates.Date) – 发行日

返回 ZeroCouponBond 对象

返回类型 Bonds.ZeroCouponBond

class Bonds.FixedRateBond(Bonds.Bond)

___init___(self, settlementDays, faceAmount, schedule, coupons, paymentDayCounter, paymentConvention = BizDayConvention.Following, redemption = 100.0, issueDate = None, paymentCalendar = 'NullCalendar')

根据日程生成固定利率国债对象

参数

- settlementDays (int) – 清算速度
- faceAmount (float) – 面值
- schedule (Schedules.Schedule) – 付息日程
- coupons (float or list) – 票息
- paymentDayCounter (DayCounters.DayCounter) – 付息天数计数惯例
- paymentConvention (BizDayConvention) – 付息工作日惯例
- redemption (float) – 到期偿付
- issueDate (Dates.Date) – 发行日
- paymentCalendar (Calendars.Calendar) – 付息日历

返回 固定利率国债对象

返回类型 Bonds.FixedRateBond

class Bonds.ConvertibleFixedCouponBond(Bonds.Bond)

可转换债券

___init___(self, exercise, conversionRatio, dividends, callabilitySchedule, creditSpread, issueDate, settlementDays, coupons, dayCounter, schedule, redemption = 100.0)

参数

- exercise (Exercises.Exercise) – 行权方式
- conversionRatio (float) – 转换比例
- dividends (Schedules.DividendSchedule) – 红利日程
- callabilitySchedule (Schedules.CallabilitySchedule) – 赎回以及回售日程
- creditSpread (float) – 信用利差
- issueDate (Dates.Date) – 发行日
- settlementDays (int) – 清算速度
- coupons (float) – 票息

- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- schedule (Schedules.Schedule) – 付息日程
- redemption (float) – 到期返还

返回 可转换债券对象

返回类型 Bonds.ConvertibleFixedCouponBond

class Bonds.CallableFixedRateBond(Bonds.Bond)

___init___(self, settlementDays, faceAmount, schedule, coupons, accrualDayCounter, callabilitySchedule, paymentConvention = BizDayConvention.Following, redemption = 100.0, issueDate = None)

参数

- settlementDays (int) – 清算速度
- faceAmount (float) – 面值
- schedule (Schedules.Schedule) – 付息日程
- coupons (float) – 票息
- accrualDayCounter (DayCounters.DayCounter) – 天数计数惯例
- callabilitySchedule (Schedules.CallabilitySchedule) – 赎回以及回售日程
- paymentConvention (BizDayConvention) –
- redemption (float) – 到期返还
- issueDate (Dates.Date) – 发行日

返回 可回售赎回债券对象

返回类型 Bonds.CallableFixedRateBond

class Bonds.BondAnalytic

atmRate(bond, yc, settlementDate = None, cleanPrice = None)

计算平价利率

参数

- bond (Bonds.Bond) – 目标债券
- yc (YieldCurve.YieldCurveProxy) – 收益率曲线
- settlementDate (Dates.Date) – 结算日
- cleanPrice (float) – 目标净价

bps(bond, ytm, dayCounter, compounding, frequency, settlementDate = None)

计算基点价值

参数

- bond (Bonds.Bond) – 目标债券
- ytm (float) – 到期收益率
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- compounding (Compounding) – 复利方式
- frequency (Frequency) – 复利频率
- settlementDate (Dates.Date) – 结算日

duration(bond, ytm, dayCounter, compounding, frequency, type = DurationType.Modified, settlementDate = None)

计算久期

参数

- bond (Bonds.Bond) – 目标债券
- ytm (float) – 到期收益率
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- compounding (Compounding) – 复利方法
- frequency (Frequency) – 复利频率
- type (DurationType) – 久期类型
- settlementDate (Dates.Date) – 结算日

convexity(bond, ytm, dayCounter, compounding, frequency, settlementDate = None)

计算凸性

参数

- bond (Bonds.Bond) – 目标债券
- ytm (float) – 到期收益率
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- compounding (Compounding) – 复利方法
- frequency (Frequency) – 复利频率
- settlementDate (Dates.Date) – 结算日

pv01(bond, yc, dayCounter, compounding, frequency, settlementDate = None)

计算 PV01

参数

- bond (Bonds.Bond) – 目标债券
- yc (YieldCurve.YieldCurveProxy) – 收益率曲线

- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- compounding (Compounding) – 复利方法
- frequency (Frequency) – 复利频率
- settlementDate (Dates.Date) – 结算日

19.2 债券函数

19.2.1 接口

BondUtilities.CTBZeroBondYield(evaluationDate, issueDate, issuePrice, maturityDate, cleanPrice, dayCounter = 'Actual/Actual (ISMA)', settlementDays = 0, rawOutput = False)

计算零息利率债券到期收益率

参数

- evaluationDate (Dates.Date) – 估值日
- issueDate (Dates.Date) – 债券发行日
- issuePrice (float) – 发行价格
- maturityDate (Dates.Date) – 到期日
- cleanPrice (float) – 净价
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- settlementDays (int) – 清算速度
- rawOutput (bool) – False 生成 pandas 格式 DataFrame, True 原始的 numpy 数组

返回 到期收益率

返回类型 pandas.DataFrame 或者 numpy.array

BondUtilities.CTBZeroBondEval(evaluationDate, issueDate, issuePrice, maturityDate, ytm, dayCounter = 'Actual/Actual (ISMA)', settlementDays = 0, rawOutput = False)

计算零息利率债券净价

参数

- evaluationDate (Dates.Date) – 估值日
- issueDate (Dates.Date) – 债券发行日
- issuePrice (float) – 发行价格
- maturityDate (Dates.Date) – 到期日
- ytm (float) – 到期收益率

- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- settlementDays (int) – 清算速度
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 净价

返回类型 pandas.DataFrame 或者 numpy.array

BondUtilities.FixedRateBondYield(evaluationDate, startDate, tenor, coupon, cleanPrice, compounding = Compounding.Compounded, frequency = Frequency.Annual, dayCounter = 'Actual/Actual (ISMA)', settlementDays = 0, rawOutput = False)

计算固定利率债券到期收益率

参数

- evaluationDate (Dates.Date) – 估值日
- startDate (Dates.Date) – 债券起息日
- tenor (Periods.Period) – 期限
- coupon (float) – 票息
- cleanPrice (float) – 净价
- compounding (Compounding) – 复利方法
- frequency (Frequency) – 付息周期，Annual, Semiannual, Quarterly, Bimonthly, Monthly ...
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- settlementDays (int) – 清算速度
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 到期收益率

返回类型 pandas.DataFrame 或者 numpy.array

BondUtilities.FixedRateBondEval(evaluationDate, startDate, tenor, coupon, ytm, compounding = Compounding.Compounded, frequency = Frequency.Annual, dayCounter = 'Actual/Actual (ISMA)', settlementDays = 0, rawOutput = False)

计算固定利率债券净价

参数

- evaluationDate (Dates.Date) – 估值日
- startDate (Dates.Date) – 债券起息日
- tenor (Periods.Period) – 期限
- coupon (float) – 票息

- ytm (float) – 到期收益率
- compounding (Compounding) – 复利方法
- frequency (Frequency) – 付息周期，Annual，Semiannual，Quarterly，Bimonthly，Monthly ...
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- settlementDays (int) – 清算速度
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 净价

返回类型 pandas.DataFrame 或者 numpy.array

19.3 利率衍生品

19.3.1 利率互换

作为中国固定收益市场流动性最好的利率衍生品，利率互换是 CAL 的金融建模中不可忽视的一大功能模块。这里我们从最常见的标准互换（Vanilla Swap）开始。

标准互换（Vanilla Swap）

最常见的互换即为固定端换浮动端的利率互换（Vanilla Swap）。在整个互换周期中，固定利率付息方（Payer）按照固定的本金、固定的付息频率、事先约定的利率支付固定利率收取方（Receiver）利息；同时固定利率收取方按照固定的本金、固定的付息频率以及当前的某个市场浮动利率标的向固定利率支付浮动利息。

首先我们需要定义一些参数去描述一个简单的互换。这里我们选用的 Swap 类型是基于 Shibor 的互换（‘ShiborSwap‘ ）：

- evaluationDate 估值日
- nominal 互换面值
- swapTenor 互换总期限
- fixedTenor 固定端付息期限，这里我们使用一年一次（1Y ）
- floatTenor 浮动端付息期限，这里我们使用三月一次（“3M” ）
- curve 挂钩市场水平的收益率曲线，这里我们使用估值日当天的中国银行间市场国债收益率曲线
- shiborIndex 浮动端依赖的 Shibor 指数
- fixedRate 固定端利率

```
In [1]: evaluationDate = Date(2015,3,18)
   ...: SetEvaluationDate(evaluationDate)
   ...: nominal = 10000000.0
   ...: swapTenor = '3Y'
   ...: fixedTenor = '1Y'
   ...: floatTenor = '3M'
   ...: curve = BuildCurve('Treasury.XIBE', evaluationDate)
   ...: shiborIndex = Shibor(floatTenor, curve)
   ...: fixedRate = 0.045
   ...:
```

- startDate 这里我们将估值日向前移动一个工作日，获取互换的起始日。这里这样操作的原因是由于 Shibor 指数重置规则。

```
In [10]: cal = Calendar('China.IB')
   ....: startDate = cal.advanceDate(evaluationDate, '1B', BizDayConvention.Following)
   ....: swap = ShiborSwap(SwapLegType.Payer,
   ....:                   nominal,
   ....:                   startDate,
   ....:                   swapTenor,
   ....:                   fixedTenor,
   ....:                   fixedRate,
   ....:                   shiborIndex)
   ....:
```

现金流分析

固定端现金流：按年支付

```
In [13]: swap.legAnalysis(0)
Out[13]:
                AMOUNT       NOMINAL ACCRUAL_START_DATE ACCRUAL_END_DATE  \
PAYMENT_DATE
2016-03-21   453698.6301 10000000.0000       2015-03-19      2016-03-21
2017-03-20   448767.1233 10000000.0000       2016-03-21      2017-03-20
2018-03-19   448767.1233 10000000.0000       2017-03-20      2018-03-19

            ACCRUAL_DAYS INDEX FIXING_DAYS FIXING_DATES INDEX_FIXING  \
PAYMENT_DATE
2016-03-21           368  #NA         #NA          #NA          #NA
2017-03-20           364  #NA         #NA          #NA          #NA
2018-03-19           364  #NA         #NA          #NA          #NA

                  DAY_COUNTER ACCRUAL_PERIOD EFFECTIVE_RATE
PAYMENT_DATE
2016-03-21   Actual/365 (Fixed)       1.0082         0.0450
2017-03-20   Actual/365 (Fixed)       0.9973         0.0450
```

2018-03-19	Actual/365 (Fixed)	0.9973	0.0450

浮动端现金流：每三个月一付

```
In [14]: swap.legAnalysis(1)
Out[14]:
              AMOUNT       NOMINAL ACCRUAL_START_DATE ACCRUAL_END_DATE  \
PAYMENT_DATE
2015-06-19  77861.2329 10000000.0000       2015-03-19      2015-06-19
2015-09-21  79575.0412 10000000.0000       2015-06-19      2015-09-21
2015-12-21  76343.3519 10000000.0000       2015-09-21      2015-12-21
2016-03-21  75558.5603 10000000.0000       2015-12-21      2016-03-21
2016-06-20  79321.5141 10000000.0000       2016-03-21      2016-06-20
2016-09-19  80428.2205 10000000.0000       2016-06-20      2016-09-19
2016-12-19  81535.0484 10000000.0000       2016-09-19      2016-12-19
2017-03-20  82616.3428 10000000.0000       2016-12-19      2017-03-20
2017-06-19  82433.6034 10000000.0000       2017-03-20      2017-06-19
2017-09-19  84174.0177 10000000.0000       2017-06-19      2017-09-19
2017-12-19  84077.0329 10000000.0000       2017-09-19      2017-12-19
2018-03-19  83946.6416 10000000.0000       2017-12-19      2018-03-19

            ACCRUAL_DAYS              INDEX FIXING_DAYS FIXING_DATES  \
PAYMENT_DATE
2015-06-19            92  Shibor3M Actual/360           1  2015-03-18
2015-09-21            94  Shibor3M Actual/360           1  2015-06-18
2015-12-21            91  Shibor3M Actual/360           1  2015-09-18
2016-03-21            91  Shibor3M Actual/360           1  2015-12-18
2016-06-20            91  Shibor3M Actual/360           1  2016-03-18
2016-09-19            91  Shibor3M Actual/360           1  2016-06-17
2016-12-19            91  Shibor3M Actual/360           1  2016-09-18
2017-03-20            91  Shibor3M Actual/360           1  2016-12-16
2017-06-19            91  Shibor3M Actual/360           1  2017-03-17
2017-09-19            92  Shibor3M Actual/360           1  2017-06-16
2017-12-19            91  Shibor3M Actual/360           1  2017-09-18
2018-03-19            90  Shibor3M Actual/360           1  2017-12-18

            INDEX_FIXING DAY_COUNTER ACCRUAL_PERIOD EFFECTIVE_RATE
PAYMENT_DATE
2015-06-19        0.0305  Actual/360        0.2556         0.0305
2015-09-21        0.0305  Actual/360        0.2611         0.0305
2015-12-21        0.0302  Actual/360        0.2528         0.0302
2016-03-21        0.0299  Actual/360        0.2528         0.0299
2016-06-20        0.0314  Actual/360        0.2528         0.0314
2016-09-19        0.0318  Actual/360        0.2528         0.0318
2016-12-19        0.0323  Actual/360        0.2528         0.0323
2017-03-20        0.0327  Actual/360        0.2528         0.0327
2017-06-19        0.0326  Actual/360        0.2528         0.0326
```

2017-09-19	0.0329	Actual/360	0.2556	0.0329
2017-12-19	0.0333	Actual/360	0.2528	0.0333
2018-03-19	0.0336	Actual/360	0.2500	0.0336

定价分析

让我们现在先假设使用之前的那条收益率曲线作为折现曲线。

```
In [15]: engine = DiscountingSwapEngine(curve)
   ....: swap.setPricingEngine(engine)
   ....: print('%.2f' % swap.NPV())
   ....:
-349781.21
```

固定端现值：

```
In [18]: print('%.2f' % swap.fixedLegNPV())
-1268669.51
```

浮动端现值：

```
In [19]: print('%.2f' % swap.floatingLegNPV())
918888.30
```

显然这个不是一个平价互换（Par Swap），需要调整固定端利率至：

```
In [20]: print('%.2f' % (swap.fairRate()*100) + '%')
3.26%
```

或者在浮动端加上如下的息差：

```
In [21]: print('%.2f' % (swap.fairSpread()*100) + '%')
1.21%
```

接口

class Swaps.SwapProxy(Instrument.Instrument)

startDate(self)

返回 开始日期

返回类型 Dates.Date

maturityDate(self)

返回 到期日

返回类型 Dates.Date

fairRate(self)

返回 平价利率

返回类型 float

fairSpread(self)

返回 平价浮动端利差

返回类型 float

fixedLegBPS(self)

返回 固定端 bps

返回类型 float

floatingLegBPS(self)

返回 浮动端 bps

返回类型 float

floatingSchedule(self)

返回 浮动端付息日程

返回类型 Schedules.Schedule

fixedSchedule(self)

返回 固定端付息日程

返回类型 Schedules.Schedule

class Swaps.VanillaSwap(Swaps.SwapProxy)

___init___(self, swapType, nominal, fixedSchedule, fixedRate, fixedDayCount, floatingSchedule, floatingIndex, floatingDayCount, spread = 0.0)

参数

- swapType (SwapLegType) – 互换类型
- nominal (float) – 名义本金
- fixedSchedule (Schedules.Schedule) – 固定端日程
- fixedRate (float) – 固定端利率
- fixedDayCount (DayCounters.DayCounter) – 固定端天数计数惯例
- floatingSchedule (Schedules.Schedule) – 浮动端日程
- floatingIndex (Index.IborIndex) – 浮动端利率指数
- floatingDayCount (DayCounters.DayCounter) – 浮动端天数计数惯例
- spread (float) – 浮动端利差

class Swaps.ShiborSwap(Swaps.SwapProxy)

___init___(self, swapType, nominal, startDate, swapTenor, fixedTenor, fixedRate, shiborIndex)

参数

- swapType (SwapLegType) – 互换类型
- nominal (float) – 名义本金
- startDate (Dates.Date) – 开始日期
- swapTenor (Periods.Period) – 互换期限
- fixedTenor (Periods.Period) – 固定端利率周期
- fixedRate (float) – 固定端利率
- shiborIndex (Index.Shibor) – Shibor 利率指数

class Swaps.RepoCompoundingSwap(Swaps.SwapProxy)

___init___(self, swapType, nominal, startDate, swapTenor, paymentTenor, fixedRate, rateSpread, repoIndex)

参数

- swapType (SwapLegType) – 互换类型
- nominal (float) – 名义本金
- startDate (Dates.Date) – 开始日期
- swapTenor (Periods.Period) – 互换期限
- paymentTenor (Periods.Period) – 付息周期
- fixedRate (float) – 固定端利率
- rateSpread (float) – 浮动端利差
- repoIndex (Index.RepoChina) – Repo 利率指数

19.3.2 利率互换

作为中国固定收益市场流动性最好的利率衍生品，利率互换是 CAL 的金融建模中不可忽视的一大功能模块。这里我们从最常见的标准互换（Vanilla Swap）开始。

标准互换（Vanilla Swap）

最常见的互换即为固定端换浮动端的利率互换（Vanilla Swap）。在整个互换周期中，固定利率付息方（Payer）按照固定的本金、固定的付息频率、事先约定的利率支付固定利率收取方（Receiver）利息；同时

固定利率收取方按照固定的本金、固定的付息频率以及当前的某个市场浮动利率标的向固定利率支付浮动利息。

首先我们需要定义一些参数去描述一个简单的互换。这里我们选用的 Swap 类型是基于 Shibor 的互换（‘ShiborSwap‘ ）:

- evaluationDate 估值日
- nominal 互换面值
- swapTenor 互换总期限
- fixedTenor 固定端付息期限，这里我们使用一年一次（1Y ）
- floatTenor 浮动端付息期限，这里我们使用三月一次（“3M” ）
- curve 挂钩市场水平的收益率曲线，这里我们使用估值日当天的中国银行间市场国债收益率曲线
- shiborIndex 浮动端依赖的 Shibor 指数
- fixedRate 固定端利率

```
In [1]: evaluationDate = Date(2015,3,18)
   ...: SetEvaluationDate(evaluationDate)
   ...: nominal = 10000000.0
   ...: swapTenor = '3Y'
   ...: fixedTenor = '1Y'
   ...: floatTenor = '3M'
   ...: curve = BuildCurve('Treasury.XIBE', evaluationDate)
   ...: shiborIndex = Shibor(floatTenor, curve)
   ...: fixedRate = 0.045
   ...:
```

- startDate 这里我们将估值日向前移动一个工作日，获取互换的起始日。这里这样操作的原因是由于 Shibor 指数重置规则。

```
In [10]: cal = Calendar('China.IB')
   ....: startDate = cal.advanceDate(evaluationDate, '1B', BizDayConvention.Following)
   ....: swap = ShiborSwap(SwapLegType.Payer,
   ....:                   nominal,
   ....:                   startDate,
   ....:                   swapTenor,
   ....:                   fixedTenor,
   ....:                   fixedRate,
   ....:                   shiborIndex)
   ....:
```

19.3.3 现金流分析

固定端现金流：按年支付

```
In [13]: swap.legAnalysis(0)
Out[13]:
              AMOUNT       NOMINAL ACCRUAL_START_DATE ACCRUAL_END_DATE  \
PAYMENT_DATE
2016-03-21  453698.6301 10000000.0000       2015-03-19      2016-03-21
2017-03-20  448767.1233 10000000.0000       2016-03-21      2017-03-20
2018-03-19  448767.1233 10000000.0000       2017-03-20      2018-03-19

            ACCRUAL_DAYS INDEX FIXING_DAYS FIXING_DATES INDEX_FIXING  \
PAYMENT_DATE
2016-03-21           368  #NA         #NA         #NA         #NA
2017-03-20           364  #NA         #NA         #NA         #NA
2018-03-19           364  #NA         #NA         #NA         #NA

                  DAY_COUNTER ACCRUAL_PERIOD EFFECTIVE_RATE
PAYMENT_DATE
2016-03-21  Actual/365 (Fixed)        1.0082         0.0450
2017-03-20  Actual/365 (Fixed)        0.9973         0.0450
2018-03-19  Actual/365 (Fixed)        0.9973         0.0450
```

浮动端现金流：每三个月一付

```
In [14]: swap.legAnalysis(1)
Out[14]:
              AMOUNT       NOMINAL ACCRUAL_START_DATE ACCRUAL_END_DATE  \
PAYMENT_DATE
2015-06-19  77861.2329 10000000.0000       2015-03-19      2015-06-19
2015-09-21  79575.0412 10000000.0000       2015-06-19      2015-09-21
2015-12-21  76343.3519 10000000.0000       2015-09-21      2015-12-21
2016-03-21  75558.5603 10000000.0000       2015-12-21      2016-03-21
2016-06-20  79321.5141 10000000.0000       2016-03-21      2016-06-20
2016-09-19  80428.2205 10000000.0000       2016-06-20      2016-09-19
2016-12-19  81535.0484 10000000.0000       2016-09-19      2016-12-19
2017-03-20  82616.3428 10000000.0000       2016-12-19      2017-03-20
2017-06-19  82433.6034 10000000.0000       2017-03-20      2017-06-19
2017-09-19  84174.0177 10000000.0000       2017-06-19      2017-09-19
2017-12-19  84077.0329 10000000.0000       2017-09-19      2017-12-19
2018-03-19  83946.6416 10000000.0000       2017-12-19      2018-03-19

            ACCRUAL_DAYS          INDEX FIXING_DAYS FIXING_DATES  \
PAYMENT_DATE
2015-06-19            92  Shibor3M Actual/360           1  2015-03-18
2015-09-21            94  Shibor3M Actual/360           1  2015-06-18
2015-12-21            91  Shibor3M Actual/360           1  2015-09-18
2016-03-21            91  Shibor3M Actual/360           1  2015-12-18
2016-06-20            91  Shibor3M Actual/360           1  2016-03-18
2016-09-19            91  Shibor3M Actual/360           1  2016-06-17
```

```
2016-12-19          91  Shibor3M Actual/360          1  2016-09-18
2017-03-20          91  Shibor3M Actual/360          1  2016-12-16
2017-06-19          91  Shibor3M Actual/360          1  2017-03-17
2017-09-19          92  Shibor3M Actual/360          1  2017-06-16
2017-12-19          91  Shibor3M Actual/360          1  2017-09-18
2018-03-19          90  Shibor3M Actual/360          1  2017-12-18

            INDEX_FIXING DAY_COUNTER ACCRUAL_PERIOD EFFECTIVE_RATE
PAYMENT_DATE
2015-06-19        0.0305  Actual/360        0.2556         0.0305
2015-09-21        0.0305  Actual/360        0.2611         0.0305
2015-12-21        0.0302  Actual/360        0.2528         0.0302
2016-03-21        0.0299  Actual/360        0.2528         0.0299
2016-06-20        0.0314  Actual/360        0.2528         0.0314
2016-09-19        0.0318  Actual/360        0.2528         0.0318
2016-12-19        0.0323  Actual/360        0.2528         0.0323
2017-03-20        0.0327  Actual/360        0.2528         0.0327
2017-06-19        0.0326  Actual/360        0.2528         0.0326
2017-09-19        0.0329  Actual/360        0.2556         0.0329
2017-12-19        0.0333  Actual/360        0.2528         0.0333
2018-03-19        0.0336  Actual/360        0.2500         0.0336
```

19.3.4 定价分析

让我们现在先假设使用之前的那条收益率曲线作为折现曲线。

```
In [15]: engine = DiscountingSwapEngine(curve)
   ....: swap.setPricingEngine(engine)
   ....: print('%.2f' % swap.NPV())
   ....:
-349781.21
```

固定端现值：

```
In [18]: print('%.2f' % swap.fixedLegNPV())
-1268669.51
```

浮动端现值：

```
In [19]: print('%.2f' % swap.floatingLegNPV())
918888.30
```

显然这个不是一个平价互换（Par Swap），需要调整固定端利率至：

```
In [20]: print('%.2f' % (swap.fairRate()*100) + '%')
3.26%
```

或者在浮动端加上如下的息差：

```
In [21]: print('%.2f' % (swap.fairSpread()*100) + '%')
1.21%
```

接口

class Swaps.SwapProxy(Instrument.Instrument)

startDate(self)

返回 开始日期

返回类型 Dates.Date

maturityDate(self)

返回 到期日

返回类型 Dates.Date

fairRate(self)

返回 平价利率

返回类型 float

fairSpread(self)

返回 平价浮动端利差

返回类型 float

fixedLegBPS(self)

返回 固定端 bps

返回类型 float

floatingLegBPS(self)

返回 浮动端 bps

返回类型 float

floatingSchedule(self)

返回 浮动端付息日程

返回类型 Schedules.Schedule

fixedSchedule(self)

返回 固定端付息日程

返回类型 Schedules.Schedule

class Swaps.VanillaSwap(Swaps.SwapProxy)

___init___(self, swapType, nominal, fixedSchedule, fixedRate, fixedDayCount, floatingSchedule, floatingIndex, floatingDayCount, spread = 0.0)

参数

- swapType (SwapLegType) – 互换类型
- nominal (float) – 名义本金
- fixedSchedule (Schedules.Schedule) – 固定端日程
- fixedRate (float) – 固定端利率
- fixedDayCount (DayCounters.DayCounter) – 固定端天数计数惯例
- floatingSchedule (Schedules.Schedule) – 浮动端日程
- floatingIndex (Index.IborIndex) – 浮动端利率指数
- floatingDayCount (DayCounters.DayCounter) – 浮动端天数计数惯例
- spread (float) – 浮动端利差

class Swaps.ShiborSwap(Swaps.SwapProxy)

___init___(self, swapType, nominal, startDate, swapTenor, fixedTenor, fixedRate, shiborIndex)

参数

- swapType (SwapLegType) – 互换类型
- nominal (float) – 名义本金
- startDate (Dates.Date) – 开始日期
- swapTenor (Periods.Period) – 互换期限
- fixedTenor (Periods.Period) – 固定端利率周期
- fixedRate (float) – 固定端利率
- shiborIndex (Index.Shibor) – Shibor 利率指数

class Swaps.RepoCompoundingSwap(Swaps.SwapProxy)

___init___(self, swapType, nominal, startDate, swapTenor, paymentTenor, fixedRate, rateSpread, repoIndex)

参数

- swapType (SwapLegType) – 互换类型
- nominal (float) – 名义本金
- startDate (Dates.Date) – 开始日期
- swapTenor (Periods.Period) – 互换期限
- paymentTenor (Periods.Period) – 付息周期

- fixedRate (float) – 固定端利率
- rateSpread (float) – 浮动端利差
- repoIndex (Index.RepoChina) – Repo 利率指数

权益

20.1 期权函数

在本模块下，我们提供了一系列函数计算基于不同模型的简单期权的价格和隐含波动率。这里所有的函数都只针对简单期权（Vanilla Option），如果希望建模更复杂的期权产品，请使用Options。

20.1.1 CAL 的期权计算函数

以下的介绍兼顾了模型出现的历史顺序以及由易到难的原则。通过这段概览，读者也可以完成对期权模型发展史的简单回顾。

注解: 以下所有的函数实际上都可以输入向量参数，其行为与 numpy 中的向量函数是一致的。

Black - Scholes - Merton 模型

> It's better to 'estimate'a model than to test it. Best of all, though, is to 'explore'a model. — Fischer Black [13]。

假设标的资产价格满足如下的随机过程:

$$\mathrm{d}S = \mu S\mathrm{d}t + \sigma S\mathrm{d}W(t). \tag{20.1}$$

其中 $\mathrm{d}W(t)$ 为布朗运动，σ 为波动率。式子右端，第一项称为漂移项，第二项称为扩散项。

Black-Scholes-Merton 公式 [6] 是衍生品发展历史上最重要的公式：

$$\mathrm{Call}(S, K, r, \tau, \sigma) = S\mathrm{N}(d_1) - K\mathrm{e}^{-r\tau}\mathrm{N}(d_2),$$

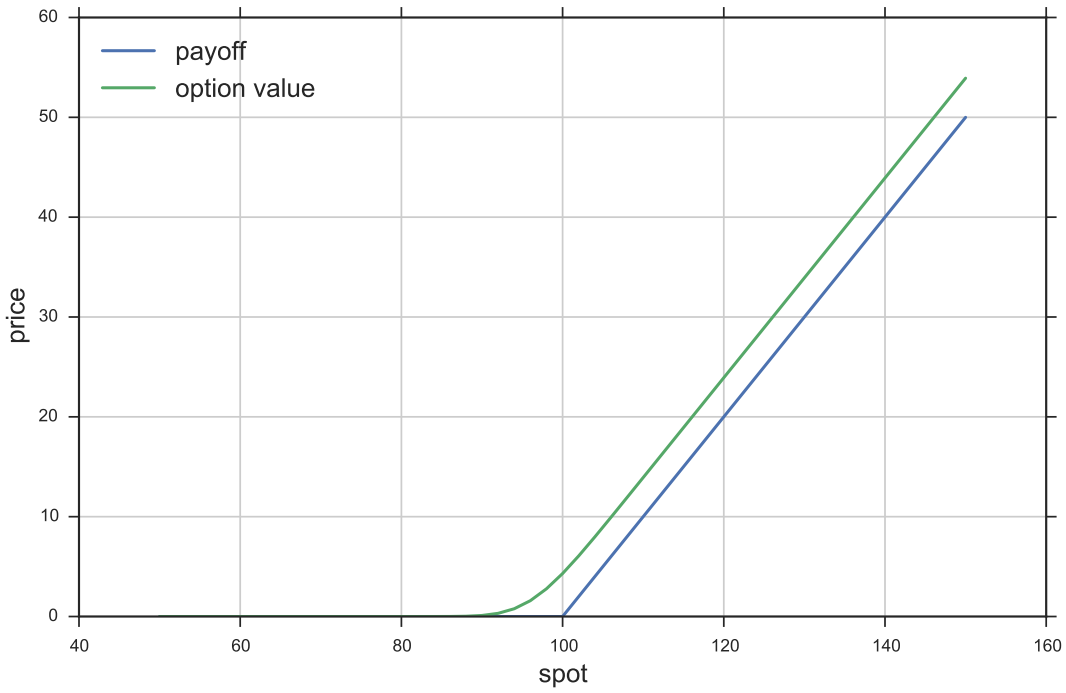
$$d_1 = \frac{\ln(S/K) + (r + \frac{1}{2}\sigma^2)\tau}{\sigma\sqrt{\tau}}, \tag{20.2}$$

$$d_2 = d_1 - \sigma\sqrt{\tau}.$$

其中 S 为标的价格，K 为执行价格，r 为无风险利率，$\tau = T - t$ 为剩余到期时间。$\mathrm{N}(x)$ 为标准正态分布的累积概率密度函数。$\mathrm{Call}(S, K, r, \tau, \sigma)$ 为看涨期权的价格。这里的标的可能是一只股票，一张债券，一个利率水平以及等等。

内置函数BSMPrice 实现了这一计算公式：

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
import numpy as np
import seaborn as sns
sns.set(style="ticks")

def plot_bsm_price(K, T, r, sigma):
    S = np.linspace(50, 150, 51)
    price = BSMPrice(1, K, S, r, 0.0, sigma, T)
    pylab.figure(figsize=(10,6))
    pylab.plot(S, np.maximum(S - K, 0.0))
    pylab.plot(S,price['price'])
    pylab.xlabel('spot', fontsize = 15)
    pylab.ylabel('price', fontsize = 15)
    pylab.legend(['payoff', r'option value'], loc = 0 , fontsize = 15)
    pylab.grid()
plot_bsm_price(K=100, T = 2, r=0.02, sigma=0.03)
pylab.show()
```



这么就完了吗？事实上，我们还能计算希腊字母！对于一个典型的欧式看涨期权而言：

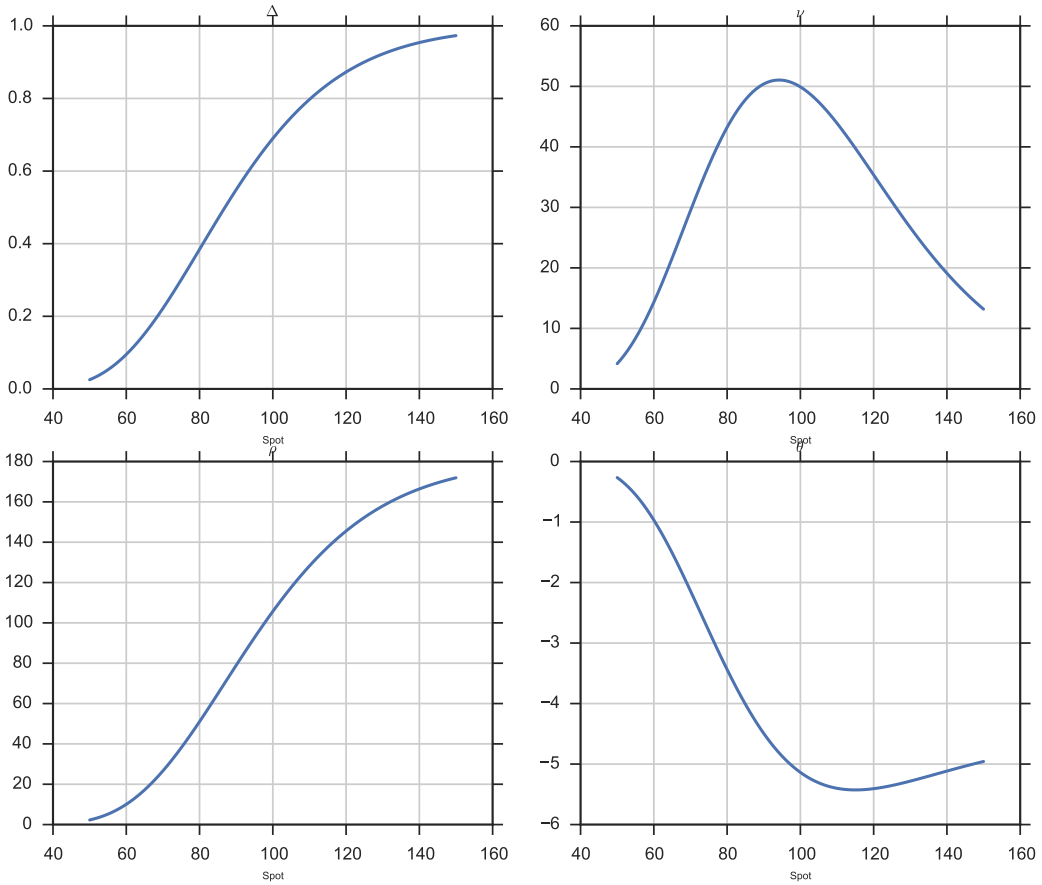
- 期权价格是标的价格 S 的单调增函数；
- 期权价格是关于波动率 σ 的单调增函数；
- 期权价格是关于无风险利率 r 的单调增函数；
- 期权价格是关于时间 t 的单调减函数，期权价格随着时间衰减。

以上的关系可以由一组 Greeks 表述 [31] 。所谓 Greeks 是期权价格关于以上参数的导数：

$$\begin{aligned}
\Delta &= \frac{\partial V}{\partial S} = \mathrm{N}(d_1), \\
\nu &= \frac{\partial V}{\partial \sigma} = S\Phi(d_1)\sqrt{\tau} = K\mathrm{e}^{-r\tau}\Phi(d_2), \\
\rho &= \frac{\partial V}{\partial r} = KT\mathrm{e}^{-r\tau}\Phi(d_2), \\
\theta &= \frac{\partial V}{\partial t} = -\frac{\partial V}{\partial \tau} = -\frac{S\Phi(d_1)\sigma}{2\sqrt{\tau}} - rK\mathrm{e}^{-r\tau}\mathrm{N}(d_2).
\end{aligned} \tag{20.3}$$

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
import numpy as np
import seaborn as sns
sns.set(style="ticks")

S = np.linspace(50, 150, 101)
K = 100.0
T = 2.0
r = 0.05
sigma = 0.2

price = BSMPrice(1, K, S, r, 0.0, sigma, T)

fig, ((ax1, ax2), (ax3, ax4)) = pylab.subplots(2,2, figsize = (10,8))
ax1.plot(S, price['delta'])
ax1.set_title('$\Delta$', fontdict = {'size':9})
ax1.set_xlabel("Spot", fontdict = {'size':6})
ax1.grid()
ax2.plot(S, price['vega'])
ax2.set_title(r'$\nu$', fontdict = {'size':9})
ax2.set_xlabel("Spot", fontdict = {'size':6})
ax2.grid()
```

```
ax3.plot(S, price['rho'])
ax3.set_title(r'$\rho$', fontdict = {'size':9})
ax3.set_xlabel("Spot", fontdict = {'size':6})
ax3.grid()
ax4.plot(S, price['theta'])
ax4.set_title(r'$\theta$', fontdict = {'size':9})
ax4.set_xlabel("Spot", fontdict = {'size':6})
ax4.grid()

pylab.show()
```



隐含波动率 (Implied Volatility)

> (Implied volatility) The wrong number in the wrong formula to get the right price. — Riccardo Rebonato [33] 。

BSM 公式中 5 个参数的地位并不是对等的。行权价 K 在期权发行时即已确定；当前价格 S 与剩余到期时间 τ 是可以直接观察的变量；无风险利率 r 需要一定的技巧去估计，有一定的不确定性。但是无风险利率变化不会很大，并且对期权价格的影响也较小。

波动率 σ 的观测与估计是最大的不确定因素，有如下几个原因：

- 波动率无法于市场直接观测得到；
- 波动率有模型依赖，在 BSM 假设下，波动率是对数正态分布的波动率，但是这个假设并不一定成立；
- 波动率并不稳定，是个随机变量，同时也不是独立同分布变量；
- 市场波动率结构与 BSM 假设不一致，存在波动率微笑（volatility smile）的现象。

作为 BSM 期权定价最重要的参数，波动率 σ 是标的资产本身的波动率。由此可以得出的结论：在 BSM 框架下，对于同一标的而言，任意期限，任意行权价的期权，波动率都应该是唯一的一个常数。真的是这样吗？

在市场上我们看到的一张期权不同到期期限和行权价格的价格表。我们选取在 7 月 11 日，沪深 300 指数看涨期权，当日沪深 300 指数收盘价 2148。删去交易极为不活跃的远月合约以及极度虚值或者极度实值期权，得到如下的表格：

表 20.1: 沪深 300 指数看涨期权

TICKER	Price	Strike
IO1407-C-2100.CFE	50.10	2100
IO1407-C-2150.CFE	18.90	2150
IO1407-C-2200.CFE	7.10	2200
IO1407-C-2250.CFE	2.90	2250
IO1407-C-2300.CFE	0.60	2300
IO1408-C-2100.CFE	72.10	2100
IO1408-C-2150.CFE	45.60	2150
IO1408-C-2200.CFE	30.50	2200
IO1408-C-2250.CFE	16.40	2250
IO1408-C-2300.CFE	6.80	2300
IO1408-C-2350.CFE	5.70	2350

但是我们更关心的是当时的报价所反映的市场对波动率的估计，这个估计的波动率称为隐含波动率（Implied Volatility）。这里的过程实际上是在 BSM 公式中，假设另外 4 个参数确定，期权价格已知，反解 σ：

$$f(\sigma) = \text{Call}(S, K, r, \tau, \sigma) = S\text{N}(d_1) - K\text{e}^{-r\tau}\text{N}(d_2),$$

$$\text{IF } f(\sigma) = V, \text{Then } \sigma = f^{-1}(V) \text{ ?}$$

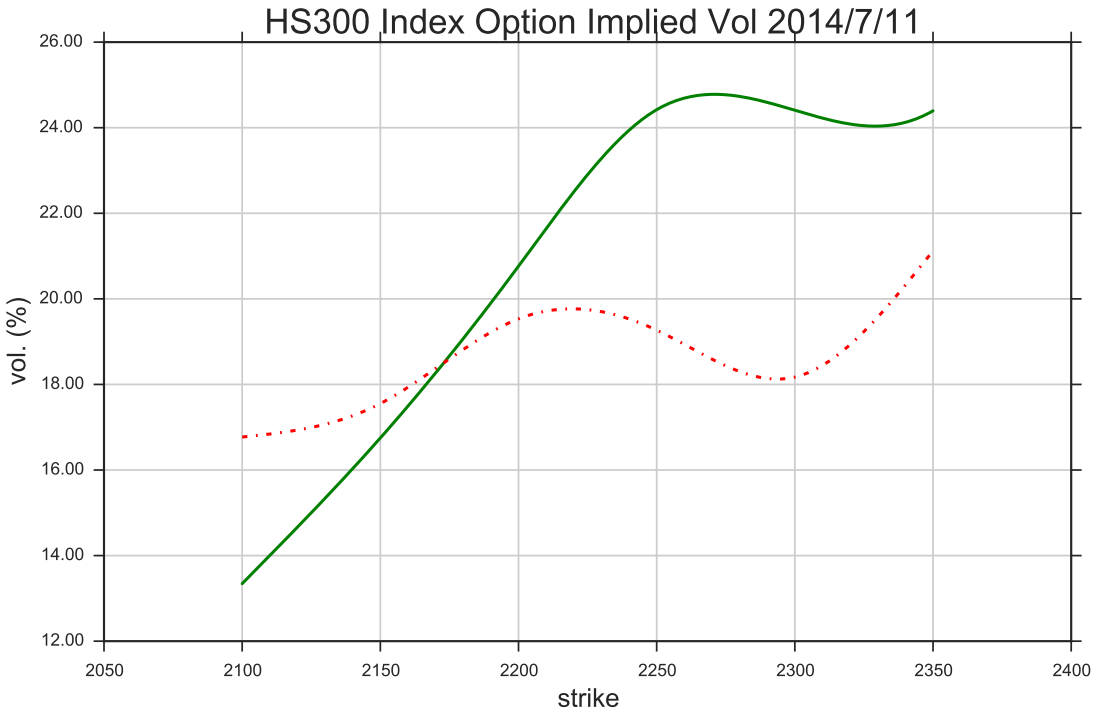
由于对于欧式看涨期权而言，其价格为对应波动率的单调递增函数，所以这个求解过程是稳定可行的。一般来说我们可以类似于试错法来实现（已有的文献已经有很多高效率的算法解决类似的单变量求根问题，例如 [32] ）。

下面利用函数BSMImpliedVolatity 计算上表对应期权的隐含波动率：

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
```

```
from matplotlib import pylab
from matplotlib import ticker
import numpy as np
import seaborn as sns
sns.set(style="ticks")

scale_pow = 2
def my_formatter_fun(x, p):
    return "%.2f" % (x * (10 ** scale_pow))

strikes = np.array([2100, 2150.0, 2200.0, 2250.0, 2300.0, 2100.0, 2150.0, 2200.0, 2250.0, 2300.0, 2350.0])
prices = np.array([50.10, 18.90, 7.10, 2.90, 0.60, 72.10, 45.60, 30.50, 16.40, 6.8, 5.7])
daysToMaturity = np.array([7, 7, 7, 7, 7, 35, 35, 35 ,35, 35 ,35])
spot = 2148.0
r = 0.0 # Assume 0 risk free
maturity = daysToMaturity / 365.0
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices)
impliedVolatilities

benchStrikes = np.linspace(2100, 2350, 200)

# July contract
iv7m = impliedVolatilities[:5]
strikes7m = strikes[:5]
iv7m = list(iv7m['vol'].values)
strikes7m = list(strikes7m)
cubic7m = CubicNaturalSpline(strikes7m, iv7m)
bench7m = [cubic7m(x, True) for x in benchStrikes]

# August contract
iv8m = impliedVolatilities[5:]
strikes8m = strikes[5:]
iv8m = list(iv8m['vol'].values)
strikes8m = list(strikes8m)
cubic8m = CubicNaturalSpline(strikes8m, iv8m)
bench8m = [cubic8m(x, True) for x in benchStrikes]

# Plot
fig = pylab.figure(figsize = (10,6))
ax = fig.gca()
ax.plot(benchStrikes,bench7m, 'g-')
ax.plot(benchStrikes,bench8m, 'r-.')
ax.set_xlim((2050,2400))
ax.get_yaxis().set_major_formatter(ticker.FuncFormatter(my_formatter_fun))
ax.set_ylabel('vol. (%)', fontsize = 15)
ax.set_xlabel('strike', fontsize = 15)
pylab.title('HS300 Index Option Implied Vol 2014/7/11', fontdict = {'size':20})
```

```
pylab.grid()
```



Bachelier 模型

> In mathematics, the art of posing a question is more important than the art of solving one. — Georg Cantor

作为热身，我们先来会一会之前见过的老朋友 Bachelier [2] 。虽然这个模型的历史已经超过 100 年了，但是它具有类似于 BSM 的优点：

- 模型参数直观，符合交易员的直觉；
- 参数少，模型的稳定性好；
- 易于做扩展；
- 具有显式看涨看跌期权计算公式。

Bachelier 模型具有如下描述：

$$\mathrm{d}S = \mu(S,t)\mathrm{d}t + \sigma \mathrm{d}W(t). \tag{20.4}$$

关于看涨期权，可以有如此的显式公式：

$$\text{Call}(S, K, r, \tau, \sigma) = (S - K\text{e}^{-r\tau})\text{N}(d) + \widetilde{\nu}\Phi(d),$$

$$d = \frac{S\text{e}^{r\tau} - K}{\widetilde{\nu}} \tag{20.5}$$

$$\widetilde{\nu} = \sigma\sqrt{\frac{1 - \text{e}^{-2r\tau}}{2r}}.$$

参数以及函数的含义如 BSM 模型。

这里我们简单的讲一下 Bachelier 模型相对于 BSM 模型的不同点。在 Bachelier 模型下，标的的价格是布朗运动，而不是几何布朗运动。更明确的讲，价格的增量本身是满足正态分布，而不是收益率。简单起见，我们假设过程的漂移项为 0，直观的看我们注意到：

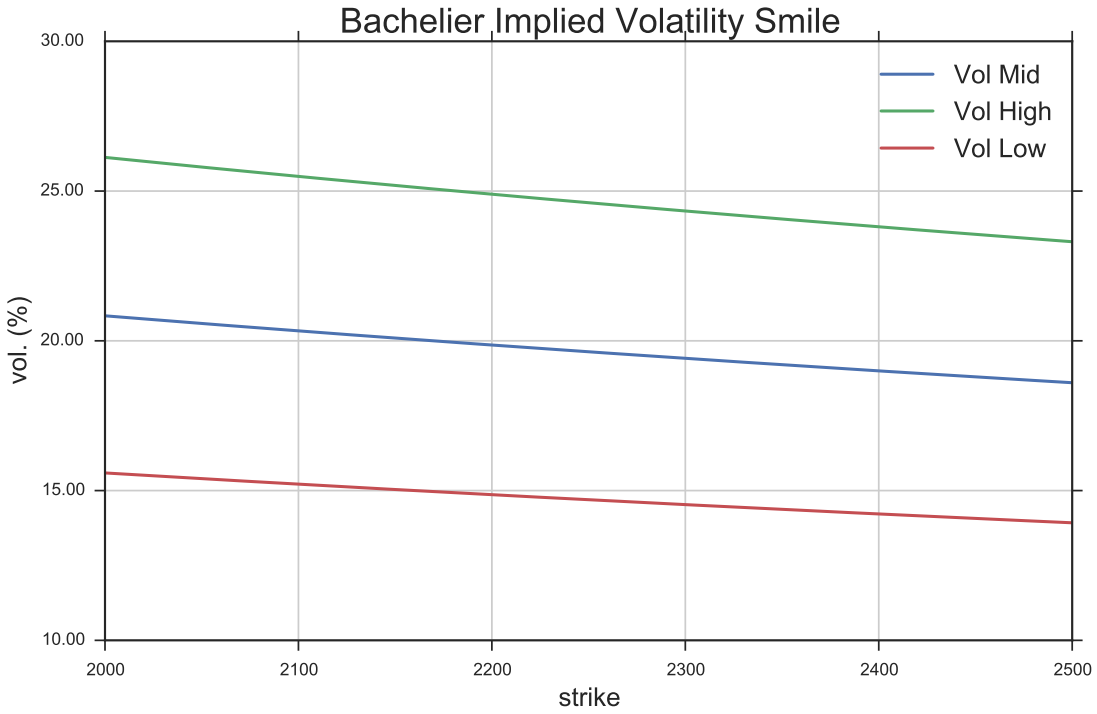
$$\text{d}S = \sigma\text{d}W(t) = \frac{\sigma}{S}S\text{d}W(t).$$

到这里，如果不严格的讲，假设，真实市场是满足 Bachelier 的假设。但是我们“错误”的使用了 BSM 模型，那么我们计算的隐含波动率实际上是将 Bachelier 模型下的波动水平强行“转换”成几何布朗运动的波动，这里，“差不多”就是 $\frac{\sigma}{S}$ 。这样的话，我们就获得了随着标的价格上升，同时下降的波动率微笑。

下面的代码，测试在 Bachelier 假设下，使用BachelierImpliedVolatility 我们能获得什么样的波动率微笑：

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
from matplotlib import ticker
import numpy as np
import seaborn as sns
sns.set(style="ticks")

scale_pow = 2
def my_formatter_fun(x, p):
    return "%.2f" % (x * (10 ** scale_pow))

spot = 2148.0
strikes = np.linspace(2000, 2500, 251)
sigma = 0.2 * spot
r = 0.0 # # Assume 0 risk free
maturity = 3.0

# sigma = 0.2 * spot
prices = BachelierPrice(1, strikes, spot, r, sigma, maturity)
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)
fig = pylab.figure(figsize = (10, 6))
ax = fig.gca()
```

```
ax.plot(strikes, impliedVolatilities)

# sigma = 0.25 * spot
sigma = 0.25 * spot
prices = BachelierPrice(1, strikes, spot, r, sigma, maturity)
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)
pylab.plot(strikes, impliedVolatilities)

# sigma = 0.15 * spot
sigma = 0.15 * spot
prices = BachelierPrice(1, strikes, spot, r, sigma, maturity)
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)
ax.plot(strikes, impliedVolatilities)

ax.grid()
ax.set_xlim((2000,2500))
ax.set_ylim((0.10,0.30))
ax.set_xlabel('strike', fontsize = 15)
ax.get_yaxis().set_major_formatter(ticker.FuncFormatter(my_formatter_fun))
ax.set_ylabel('vol. (%)', fontsize = 15)
ax.legend(['Vol Mid', "Vol High", "Vol Low"],fontsize = 15, loc = 'best')
ax.set_title('Bachelier Implied Volatility Smile', fontdict = {'size':20})

pylab.show()
```



BaroneAdesi - Whaley **美式期权定价模型**

平移扩散模型

1983 年, Rubinstein [34] 提出了 Displaced Diffusion 模型 (又被称为 Shifted-Lognormal 模型 [8] , 下简称 DD 模型)。就像它名字暗示的一样, 该模型与 Lognormal 的 BSM 模型紧密相关 (事实上, 它与 Bachelier 模型的联系也一样紧密)。标底资产的价格满足如下的过程：

$$\frac{\mathrm{d}(S+\alpha)}{S+\alpha} = \mu \mathrm{d}t + \sigma \mathrm{d}W(t). \tag{20.6}$$

其中 α 是偏移量，其他参数如 BSM 模型。

DD 模型实际上可以看作 BSM 模型与 Bachelier 模型的加权平均，它与两者的关系可以有下面的渐近表达式：

$$\text{When } \alpha = 0,\ DD = \text{Log-Normal} = \text{BSM},$$

$$\text{When } \alpha \to +\infty,\ DD = \text{Normal} = \text{Bachelier}.$$

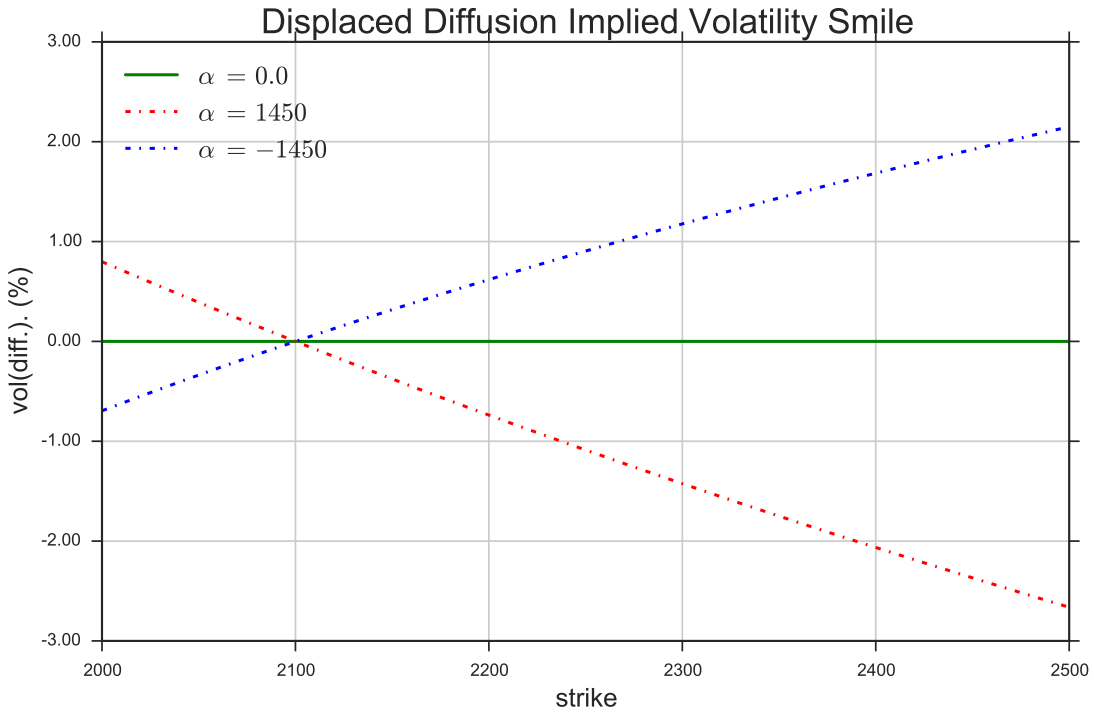
实际上令 $\widetilde{S} = S + \alpha$: $\frac{\mathrm{d}(\widetilde{S})}{\widetilde{S}} = \mu \mathrm{d}t + \sigma \mathrm{d}W(t)$. 即为 BSM 假设的情形，所以我们可以参考的得到在 DD 模型下看涨期权的定价公式：

$$\begin{aligned} \mathrm{DD}(S, K, r, \tau, \sigma, \alpha) &= \mathrm{Call}(S+\alpha, K+\alpha, r, \tau, \sigma) \\ &= (S+\alpha)\mathrm{N}(d_1) - (K+\alpha)\mathrm{e}^{-r\tau}\mathrm{N}(d_2) \\ d_1 &= \frac{\ln((S+\alpha)/(K+\alpha)) + (r + \frac{1}{2}\sigma^2)\tau}{\sigma\sqrt{\tau}}, \\ d_2 &= d_1 - \sigma\sqrt{\tau}. \end{aligned} \tag{20.7}$$

下面的代码展示了在不同的 α 设置下，可以产生的波动率微笑。可以看到 DD 模型相对于 Bachelier 模型有更大的灵活性。为了展示的方便，下面图中 Y 轴所表示的是对应行权价期权与平值期权波动率的差，使用的函数为DisplacedDiffussionPrice ：

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
from matplotlib import ticker
import numpy as np
import seaborn as sns
sns.set(style="ticks")

scale_pow = 2
def my_formatter_fun(x, p):
```

```
    return "%.2f" % (x * (10 ** scale_pow))

spot = 2148.0
strikes = np.linspace(2000, 2500, 251)
sigma = 0.4
r = 0.0 # 假设 0 利率
maturity = 3.0

# \alpha == 0, exactly as BSM
alpha = 0.0
prices = DisplacedDiffussionPrice(1, strikes, spot, alpha, r, 0.0, sigma, maturity)
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)

fig = pylab.figure(figsize = (10, 6))
ax = fig.gca()
ax.plot(strikes, impliedVolatilities['vol'].values - impliedVolatilities['vol'].values[50], 'g')

# \alpha == 1450
alpha = 1450.0
prices = DisplacedDiffussionPrice(1, strikes, spot, alpha, r, 0.0, sigma, maturity)
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)
ax.plot(strikes, impliedVolatilities['vol'].values - impliedVolatilities['vol'].values[50], 'r-.')

# \alpha == -1450
alpha = -1450.0
prices = DisplacedDiffussionPrice(1, strikes, spot, alpha, r, 0.0, sigma, maturity)
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)
ax.plot(strikes, impliedVolatilities['vol'].values - impliedVolatilities['vol'].values[50], 'b-.')

ax.set_xlabel('strike', fontsize = 15)
ax.get_yaxis().set_major_formatter(ticker.FuncFormatter(my_formatter_fun))
ax.set_ylabel('vol(diff.). (%)', fontsize = 15)
ax.legend([r'$\alpha \ = \ 0.0$', r'$\alpha \ = \ 1450$', r'$\alpha \ = \ -1450$'],fontsize = 15, loc = 'best')
ax.set_title('Displaced Diffusion Implied Volatility Smile', fontdict = {'size':20})
ax.grid()

pylab.show()
```



常弹性方差模型

实际上有其它的方法产生类似于 DD 模型的效果。Cox [12] 提出了 Constant Elasticity of Variance Model 模型，在这个框架下，标的资产的价格满足如下的过程：

$$\mathrm{d}S = \mu S\mathrm{d}t + \sigma S^{\beta/2}\mathrm{d}W(t). \tag{20.8}$$

其中 $\beta - 2$ 称为弹性指数，$0 \leq \beta \leq 2$ 。

对于这个模型，有如下的看涨期权定价公式 [24] ：

$$\begin{aligned}
&\text{Let } a = \frac{1}{2}\sigma^2(2-\beta)^2, k^* = \frac{2(r-a)}{\sigma^2(2-\beta)[\mathrm{e}^{(r-a)(2-\beta)\tau}-1]},\\
&x = k^* S^{2-\beta}\mathrm{e}^{(r-a)(2-\beta)\tau},\\
&\mathrm{Call}(S,K,r,\tau,\sigma,\beta) = S\mathrm{e}^{-r\tau}\sum_{n=0}^{\infty}\frac{\mathrm{e}^{-x}x^n\mathrm{G}(n+1+1/(2-\beta), k^*K^{2-\beta})}{\Gamma(n+1)}\\
&\quad - K\mathrm{e}^{-r\tau}\sum_{n=0}^{\infty}\frac{\mathrm{e}^{-x}x^{n+1/(2-\beta)}\mathrm{G}(n+1, k^*K^{2-\beta})}{\Gamma(n+1+1/(2-\beta))}.
\end{aligned} \tag{20.9}$$

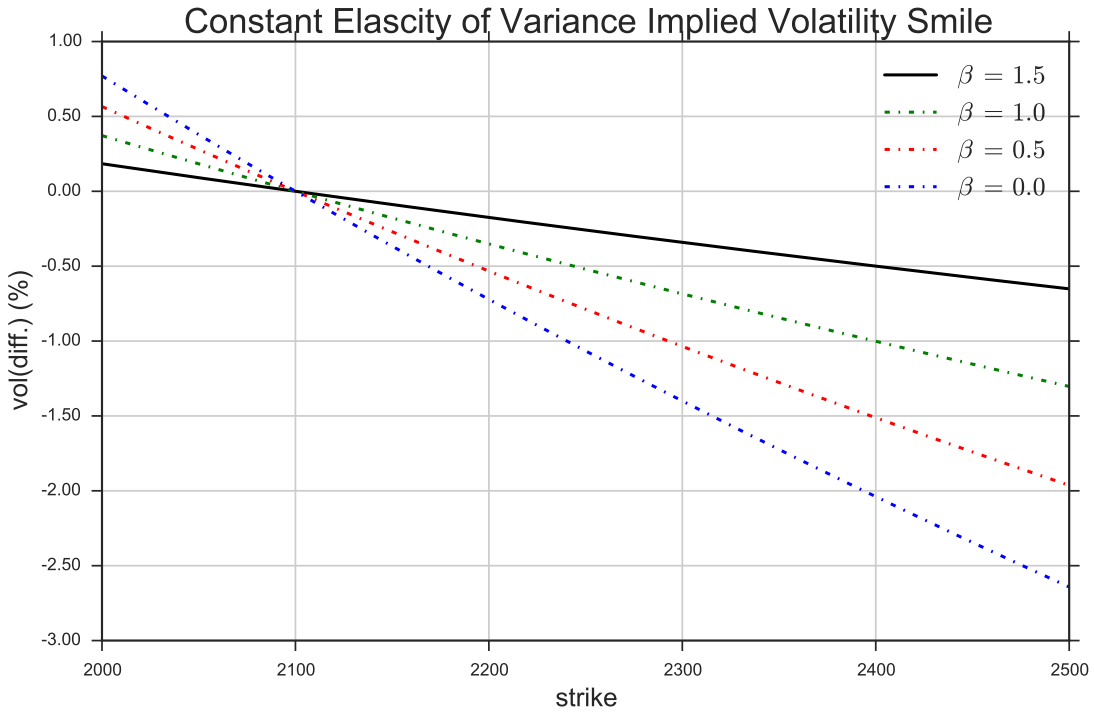
其中 $\mathrm{G}(m,v) = \frac{1}{\Gamma(m)}\int_v^\infty \mathrm{e}^{-u}u^{m-1}\mathrm{d}u$ 为标准 Complementary Gamma 分布函数。$\Gamma(z)$ 为常规的 gamma 函数。

由下面的图可以看到的，CEV 模型产生的波动率微笑与 DD 模型 α 取正时的很接近。这一现象已经被注意到，事实上，有一个渐近分解，使得一个 CEV 模型” 约等于” 一个 DD 模型 [1] 。

以下的代码展示 CEV 模型的波动率微笑，使用函数CEVPrice ：

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
from matplotlib import ticker
import numpy as np
import seaborn as sns
sns.set(style="ticks")

scale_pow = 2
def my_formatter_fun(x, p):
    return "%.2f" % (x * (10 ** scale_pow))

spot = 2148.0
strikes = np.linspace(2000, 2500, 251)
r = 0.0 # 假设 0 利率
sigma = 0.3
maturity = 3.0

# \beta == 0.75
beta = 0.75
vol = sigma * (spot ** (1.0 - beta))
prices = CEVPrice(1, strikes, spot, beta, r, vol, maturity)
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)

fig = pylab.figure(figsize = (10, 6))
ax = fig.gca()
ax.plot(strikes, impliedVolatilities['vol'].values - impliedVolatilities['vol'].values[50], 'k')

# \beta == 0.5
beta = 0.5
vol = sigma * (spot ** (1.0 - beta))
prices = CEVPrice(1, strikes, spot, beta, r, vol, maturity)
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)
ax.plot(strikes, impliedVolatilities['vol'].values - impliedVolatilities['vol'].values[50], 'g-.')

# \beta == 0.25
beta = 0.25
vol = sigma * (spot ** (1.0 - beta))
prices = CEVPrice(1, strikes, spot, beta, r, vol, maturity)
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)
ax.plot(strikes, impliedVolatilities['vol'].values - impliedVolatilities['vol'].values[50], 'r-.')
```

```
# \beta == 0.0
beta = 0.0
vol = sigma * (spot ** (1.0 - beta))
prices = CEVPrice(1, strikes, spot, beta, r,vol, maturity)
impliedVolatilities = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)
ax.plot(strikes, impliedVolatilities['vol'].values - impliedVolatilities['vol'].values[50], 'b-.')

ax.set_xlabel('strike', fontsize = 15)
ax.get_yaxis().set_major_formatter(ticker.FuncFormatter(my_formatter_fun))
ax.set_ylabel('vol(diff.) (%)', fontsize = 15)
ax.legend([r'$\beta \ = \ 1.5$', r'$\beta \ = \ 1.0$', r'$\beta \ = \ 0.5$', r'$\beta \ = \ 0.0$'],fontsize = 15, loc = 'best')
ax.set_title('Constant Elascity of Variance Implied Volatility Smile', fontdict = {'size':20})
ax.grid()

pylab.show()
```



警告: 这里我们 β 的定义遵循了 Cox 最初的选择，实际上我们模型的配置为：

$$\mathrm{d}S = \mu S \mathrm{d}t + \sigma S^{\widetilde{\beta}} \mathrm{d}W(t).$$

SABR/Hagan 模型

上面介绍的关于 BSM 模型的扩展，有一个共同的问题。他们并不能产生真正的波动率微笑（两头上翘，中部下沉），只能产生波动率倾斜（volatility skew)。虽然已有证明，任意的市场上的波动率微笑，都可以被一个局部波动率函数完全精确捕捉，但是这个函数可能变得非常复杂，显然不是之前描述的那些简单形式

[16] 。为了突破这一困境，同时保持函数形式的简单可理解，一些新的模型被提了出来。这些模型的共同特征是将波动率视为随机过程，将原先的单因子的架构改为多因子。

Hagan 等人提出了一个随机波动率模型 [19] （之所以又被称为 SABR 模型，是由于它又可以被叫做 Stochastic Alpha Beta Rho model ）。它的远期标的价格过程满足 CEV 过程，同时波动率满足对数正态分布，如下：

$$\begin{cases} \mathrm{d}F(t) & = \alpha(t)F(t)^{\beta}\mathrm{d}W_1(t) \\ \mathrm{d}\alpha(t) & = \nu\alpha(t)\mathrm{d}W_2(t) \\ \mathrm{d}W_1\mathrm{d}W_2 & = \rho\mathrm{d}t,\ \alpha(0)=\alpha. \end{cases} \tag{20.10}$$

其中 F 为标的资产的远期价格, ν 为波动率的波动率。这个模型在增加了一个因子的情况下，使得模型的复杂性大大上升，一般意义上的显式解已经不存在了。Hagan 等人运用渐近分解技术，给出了一个精度令人满意的隐含波动率的逼近解：

$$\begin{aligned} \sigma_B(K,F) \approx & \frac{\alpha}{(FK)^{(1-\beta)/2}\{1+\frac{(1-\beta)^2}{24}\ln^2 F/K+\frac{(1-\beta)^4}{1920}\ln^4 F/K\}}\cdot(\frac{z}{x(z)}) \\ & \cdot\{1+[\frac{(1-\beta)^2}{24}\frac{\alpha^2}{(F/K)^{1-\beta}}+\frac{1}{4}\frac{\rho\beta\nu\alpha}{(F/K)^{(1-\beta)/2}}+\frac{2-3\rho^2}{24}\nu^2]\cdot\tau\} \end{aligned} \tag{20.11}$$

其中：

$$z=\frac{\nu}{\alpha}(FK)^{(1-\beta)/2}\ln F/K,$$

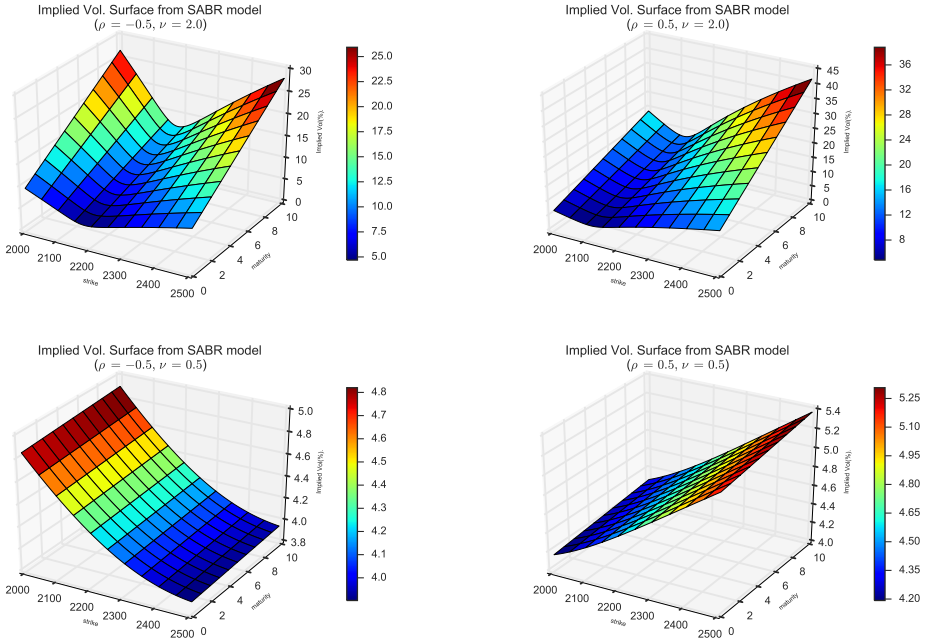
$$x(z)=\ln\{\frac{\sqrt{1-2\rho z+z^2}+z-\rho}{1-\rho}\}.$$

使用上面的逼近式，任何人都可以快速的在 SABR 模型下计算得对应行权价的隐含波动率。下面的代码，计算 SABR 隐含波动率曲面，使用函数SABRImpliedVolSurface ：

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import seaborn as sns
sns.set(style="ticks")

forward = 2148.0
strikes = np.linspace(2000, 2500, 100)
maturity = np.linspace(0.2, 10.0, 100)
alpha = 0.3
```

```
beta = 0.75
nu = 2.0
rho = - 0.5

strikeGrid, maturityGrid = np.meshgrid(strikes, maturity)

# rho = -0.5, nu = 2.0
impliedVolatilities = SABRImpliedVolSurface(strikeGrid, forward, maturityGrid, alpha, beta, nu, rho, True)
fig= pylab.figure(figsize = (16,10))
ax = fig.add_subplot(2, 2, 1, projection = '3d')
surface = ax.plot_surface(strikeGrid, maturityGrid, impliedVolatilities*100, cmap=cm.jet)
fig.colorbar(surface,shrink=0.75)
ax.set_xlabel("strike", fontdict={"size":6})
ax.set_ylabel(r"maturity", fontdict={"size":6})
ax.set_zlabel(r"Implied Vol(%).", fontdict={"size":6})
ax.set_title("Implied Vol. Surface from SABR model \n" + r" ($\rho \ =\ -0.5, \ \nu \ = \ 2.0$)" , fontdict={"size":12})

# rho = 0.5, nu = 2.0
rho = 0.5
impliedVolatilities = SABRImpliedVolSurface(strikeGrid, forward, maturityGrid, alpha, beta, nu, rho, True)
ax = fig.add_subplot(2, 2, 2, projection = '3d')
surface = ax.plot_surface(strikeGrid, maturityGrid, impliedVolatilities*100, cmap=cm.jet)
fig.colorbar(surface,shrink=0.75)
ax.set_xlabel("strike", fontdict={"size":6})
ax.set_ylabel(r"maturity", fontdict={"size":6})
ax.set_zlabel(r"Implied Vol(%).", fontdict={"size":6})
ax.set_title("Implied Vol. Surface from SABR model \n" + r" ($\rho \ =\ 0.5, \ \nu \ = \ 2.0$)" , fontdict={"size":12})

# rho = - 0.5, nu = 0.5
rho = - 0.5
nu = 0.2
impliedVolatilities = SABRImpliedVolSurface(strikeGrid, forward, maturityGrid, alpha, beta, nu, rho, True)
ax = fig.add_subplot(2, 2, 3, projection = '3d')
surface = ax.plot_surface(strikeGrid, maturityGrid, impliedVolatilities*100, cmap=cm.jet)
fig.colorbar(surface,shrink=0.75)
ax.set_xlabel("strike", fontdict={"size":6})
ax.set_ylabel(r"maturity", fontdict={"size":6})
ax.set_zlabel(r"Implied Vol(%).", fontdict={"size":6})
ax.set_title("Implied Vol. Surface from SABR model \n" + r" ($\rho \ =\ -0.5, \ \nu \ = \ 0.5$)" , fontdict={"size":12})

# rho = 0.5, nu = 0.5
rho = 0.5
nu = 0.2
impliedVolatilities = SABRImpliedVolSurface(strikeGrid, forward, maturityGrid, alpha, beta, nu, rho, True)
ax = fig.add_subplot(2, 2, 4, projection = '3d')
surface = ax.plot_surface(strikeGrid, maturityGrid, impliedVolatilities*100, cmap=cm.jet)
fig.colorbar(surface,shrink=0.75)
```

```
ax.set_xlabel("strike", fontdict={"size":6})
ax.set_ylabel(r"maturity", fontdict={"size":6})
ax.set_zlabel(r"Implied Vol(%).", fontdict={"size":6})
ax.set_title("Implied Vol. Surface from SABR model \n" + r" ($\rho \ =\ 0.5, \ \nu \ = \ 0.5$)" , fontdict={"size":12})

pylab.show()
```

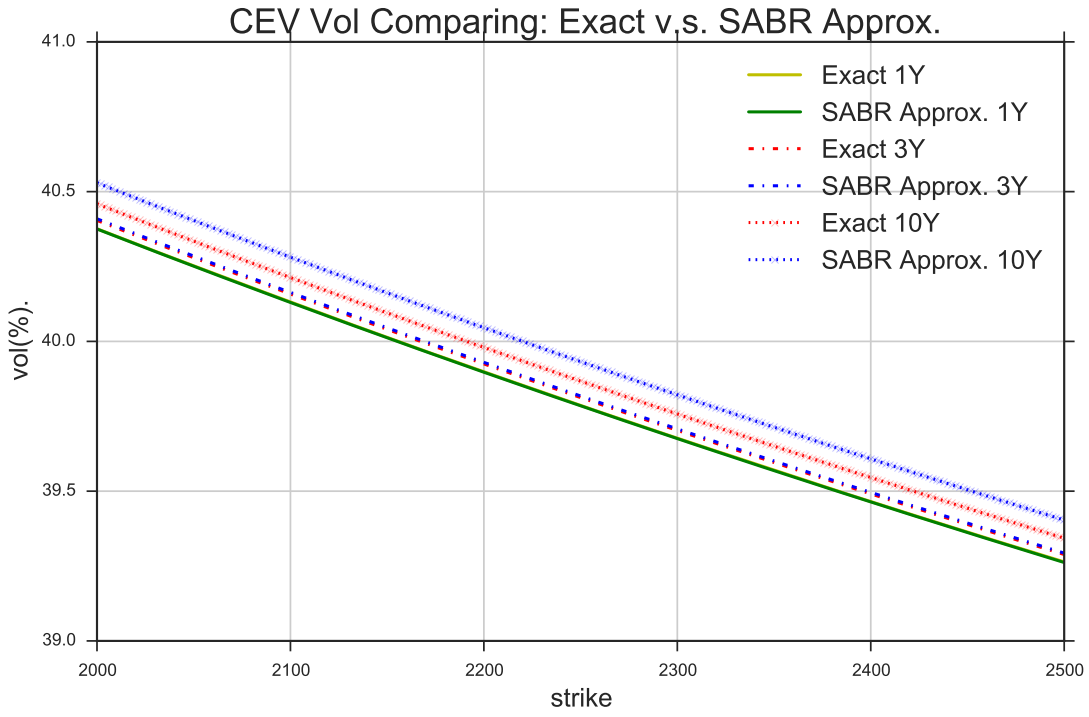


CEV 模型与 SABR 模型的关系

在介绍新的随机波动率模型之前，我们提一下 CEV 模型与 SABR 模型之间的关系。实际上如果我们在 SABR 模型中令 $\nu = 0$ ， SABR 模型就退化为一个 CEV 模型。所以 SABR 模型的逼近结果实际上可以用来实现 CEV 定价公式的快速解 [10] 。由图可知，SABR 渐近分解对于 CEV 模型已经足够精确，误差是随着期限的增长而增长的。

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
import numpy as np
import seaborn as sns
sns.set(style="ticks")

spot = 2148
strikes = np.linspace(2000, 2500, 251)
```

```
sigma = 0.4
r = 0.0 # 假设 0 利率
maturity = 1.0

fig, ax1 = pylab.subplots(1,1, figsize = (10, 6))

# \beta == 0.75, maturity = 1.0
beta = 0.75
vol = sigma * (spot ** (1.0 - beta))
prices = CEVPrice(1, strikes, spot, beta, r, vol, maturity)
impliedVolatilities1 = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)

ax1.plot(strikes, impliedVolatilities1['vol'].values*100, 'y')

impliedVolatilities2 = SABRImpliedVolSurface(strikes, spot, maturity, vol, beta, 0.0, 0.0)
ax1.plot(strikes, impliedVolatilities2*100, 'g')

# \beta == 0.75, maturity = 3.0
beta = 0.75
maturity = 3.0
vol = sigma * (spot ** (1.0 - beta))
prices = CEVPrice(1, strikes, spot, beta, r, vol, maturity)
impliedVolatilities1 = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)

ax1.plot(strikes, impliedVolatilities1['vol'].values*100, 'r-.')
impliedVolatilities2 = SABRImpliedVolSurface(strikes, spot, maturity, vol, beta, 0.0, 0.0)
ax1.plot(strikes, impliedVolatilities2*100, 'b-.')

# \beta == 0.75, maturity = 10.0
beta = 0.75
maturity = 10.0
vol = sigma * (spot ** (1.0 - beta))
prices = CEVPrice(1, strikes, spot, beta, r, vol, maturity)
impliedVolatilities1 = BSMImpliedVolatity(1, strikes, spot, r, 0.0, maturity, prices['price'].values)

ax1.plot(strikes, impliedVolatilities1['vol'].values*100, 'r:x', markersize = 4)

impliedVolatilities2 = SABRImpliedVolSurface(strikes, spot, maturity, vol, beta, 0.0, 0.0)
ax1.plot(strikes, impliedVolatilities2*100, 'b:x', markersize = 4)
ax1.set_ylim((39.00,41.00))
ax1.set_xlabel("strike", fontdict = {'size':15})
ax1.set_ylabel("vol(%).", fontdict = {'size':15})
ax1.grid()
ax1.set_title("CEV Vol Comparing: Exact v.s. SABR Approx.", fontdict={"size":20})
ax1.legend(["Exact 1Y", "SABR Approx. 1Y", "Exact 3Y", "SABR Approx. 3Y", "Exact 10Y", "SABR Approx. 10Y"], fontsize = 15
```



无套利 SABR/Hagan 模型

Heston 模型

现在我们将介绍最后一个随机波动率模型：Heston 模型 [23]。Heston 模型的历史比 SABR 模型的历史更早。在 Heston 模型中，Heston 将方差过程定义为满足一个 CIR 过程 [12] 。由于 CIR 过程具有中值回归的性质，Heston 模型可以舒缓 SABR 模型中波动率爆破 (volatility explode) 的问题。

Heston 模型如下：

$$
\begin{aligned}
\mathrm{d}S(t) &= \mu S(t)\mathrm{d}t + \sqrt{V(t)}S(t)\mathrm{d}W_1, \\
\mathrm{d}V(t) &= \kappa(\theta - V(t))\mathrm{d}t + \nu\sqrt{V(t)}\mathrm{d}W_2, \\
S(0) &= S_0, \\
V(0) &= V_0, \\
\langle \mathrm{d}W_1, \mathrm{d}W_2 \rangle &= \rho \mathrm{d}t.
\end{aligned}
\tag{20.12}
$$

其中 θ 为回归均值水平，κ 为回归速度。

Heston 的方差过程由于有均值回归的性质，它正好弥补了 SABR 模型的两个缺点：

- Heston 模型的波动率微笑会随着期限衰减；

- Heston 模型的波动率水平不会指数增长。

但是 Heston 模型相对于 SABR 模型还是有一个劣势：Heston 只有半显示解，Heston 模型的期权定价公式依赖于 Fourier 变换，仍然是一个较为复杂的积分过程，所以 Heston 公式的求解速度远远逊于 SABR 模型的渐近表达式。我们把它罗列如下，结果来自于 [23] ：

$$C(S,\tau,K) = SF_1 - K\mathrm{e}^{-r\tau}F_2. \tag{20.13}$$

其中：

$$F_j = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \mathrm{Re}(f_j(\phi)\frac{\exp(-i\phi\ln K)}{i\phi})\mathrm{d}\phi,\ j=1,2,$$

$$f_j = \exp(i\phi(x_0 + r\tau) - s_{2j}(V_0 + \kappa\theta\tau) + H_{1j}(\tau)V_0 + H_{2j}(\tau)),$$

$$= \exp(i\phi(x_0 + r\tau))f_j^{Heston}(\phi),$$

这里面：

$$f_j^{Heston}(\phi) = \exp(-s_{2j}(V_0 + \kappa\theta\tau) + H_{1j}(\tau)V_0 + H_{2j}(\tau)),$$

$$H_{1j}(\tau) = \frac{1}{\gamma_2}[\gamma_1 s_{2j}(1+\mathrm{e}^{-\gamma_1\tau}) - (1-\mathrm{e}^{-\gamma_1\tau})(2s_{1j} + \kappa s_{2j})],$$

$$H_{2j}(\tau) = \frac{2\kappa\theta}{\sigma^2}\ln[\frac{2\gamma_1}{\gamma_2}\exp(\frac{1}{2}(\kappa-\gamma_1)\tau)],$$

$$\gamma_1 = \sqrt{\kappa^2 + 2\sigma^2 s_{1j}},$$

$$\gamma_2 = 2\gamma_1\mathrm{e}^{-\gamma_1\tau} + (\kappa + \gamma_1 - \sigma^2 s_{2j})(1-\mathrm{e}^{-\gamma_1\tau}),$$

$$s_{11} = -(1+i\phi)[\frac{\rho\kappa}{\sigma} - \frac{1}{2} + \frac{1}{2}(1+i\phi)(1-\rho^2)],$$

$$s_{21} = (1+i\phi)\frac{\rho}{\sigma},$$

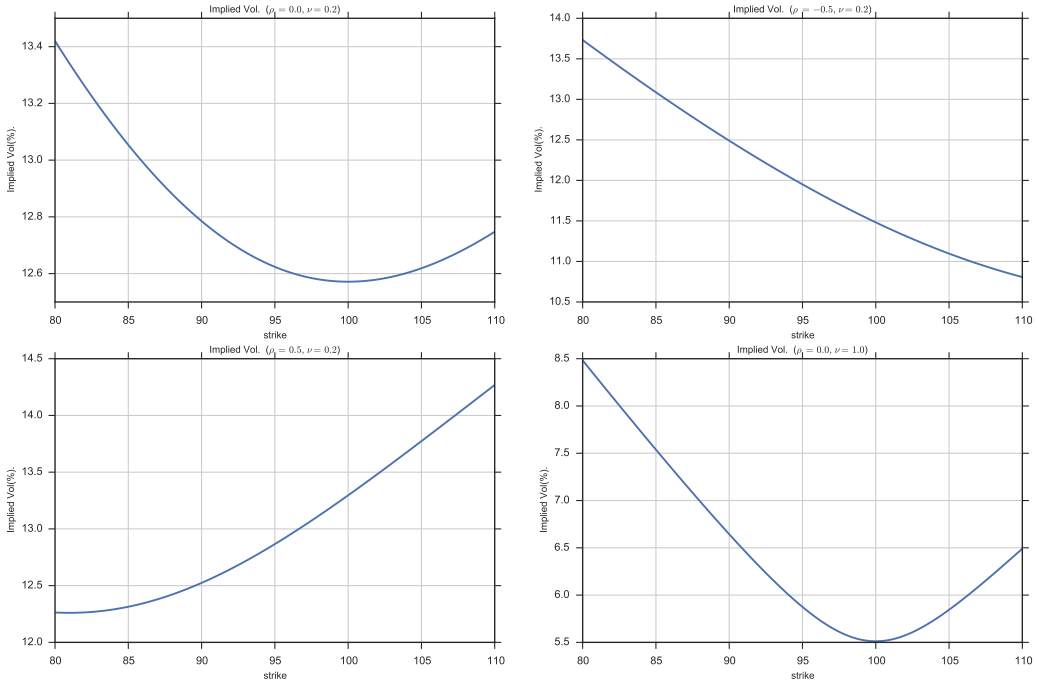
$$s_{12} = -i\phi[\frac{\rho\kappa}{\sigma} - \frac{1}{2} + \frac{1}{2}i\phi(1-\rho^2)],$$

$$s_{22} = i\phi\frac{\rho}{\sigma}.$$

下面的代码展示了 Heston 模型的波动率曲面在不同参数设置下的表现。其中使用函数HestonPrice 。看得出来 Heston 可以根据需要产生微笑 (smile) 或者倾斜 (skew)：

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
import numpy as np
import seaborn as sns
sns.set(style="ticks")

spot = 100.0
strikes = np.linspace(80, 110, 100)
maturity = 10
optionType = -1
riskFree = 0.0
dividend = 0.0
v0 = 0.025
kappa = 0.1
theta = 0.025
sigma = 0.2
rho = 0.0

strikeGrid, maturityGrid = np.meshgrid(strikes, maturity)

# rho = 0.0
rho = 0.0
price = HestonPrice(optionType, strikes, spot, riskFree, dividend, maturity, v0, kappa, theta, sigma, rho, True)
impliedVolatilities1 = BSMImpliedVolatity(optionType, strikes, spot, riskFree, dividend, maturity, price, True)
fig= pylab.figure(figsize = (16,10))
ax = fig.add_subplot(2, 2, 1)
surface = ax.plot(strikes, impliedVolatilities1*100)
ax.set_xlabel("strike", fontdict={"size":9})
ax.set_ylabel(r"Implied Vol(%).", fontdict={"size":9})
ax.set_title("Implied Vol. " + r" ($\rho \ =\ 0.0, \ \nu = \ 0.2$)" , fontdict={"size":9})
ax.grid()

# rho = - 0.5
rho = - 0.5
price = HestonPrice(optionType, strikes, spot, riskFree, dividend, maturity, v0, kappa, theta, sigma, rho, True)
impliedVolatilities1 = BSMImpliedVolatity(optionType, strikes, spot, riskFree, dividend, maturity, price, True)
ax = fig.add_subplot(2, 2, 2)
ax.set_xlabel("strike", fontdict={"size":9})
ax.set_ylabel(r"Implied Vol(%).", fontdict={"size":9})
surface = ax.plot(strikes, impliedVolatilities1*100)
ax.set_title("Implied Vol. " + r" ($\rho \ =\  - 0.5, \ \nu = \ 0.2$)" , fontdict={"size":9})
ax.grid()

# rho = 0.5
rho = 0.5
```

```
price = HestonPrice(optionType, strikes, spot, riskFree, dividend, maturity, v0, kappa, theta, sigma, rho, True)
impliedVolatilities1 = BSMImpliedVolatity(optionType, strikes, spot, riskFree, dividend, maturity, price, True)
ax = fig.add_subplot(2, 2, 3)
ax.set_xlabel("strike", fontdict={"size":9})
ax.set_ylabel(r"Implied Vol(%).", fontdict={"size":9})
surface =ax.plot(strikes, impliedVolatilities1*100)
ax.set_title("Implied Vol. " + r" ($\rho \ =\  0.5, \ \nu = \ 0.2$)" , fontdict={"size":9})
ax.grid()

# rho = 0.5
rho = 0.0
sigma = 1.0
price = HestonPrice(optionType, strikes, spot, riskFree, dividend, maturity, v0, kappa, theta, sigma, rho, True)
impliedVolatilities1 = BSMImpliedVolatity(optionType, strikes, spot, riskFree, dividend, maturity, price, True)
ax = fig.add_subplot(2, 2, 4)
ax.set_xlabel("strike", fontdict={"size":9})
ax.set_ylabel(r"Implied Vol(%).", fontdict={"size":9})
surface = ax.plot(strikes, impliedVolatilities1*100)
ax.set_title("Implied Vol. " + r" ($\rho \ =\  0.0, \ \nu = \ 1.0$)" , fontdict={"size":9})
ax.grid()
```



写在最后

All models are wrong, but some are useful. — George E. P. Box [7]

我们回顾了从全常值参数的 BSM 模型到多因子的随机波动率模型，例如 SABR 以及 Heston。这个模型进化的过程，反映了金融从业者在面对市场变化或者产生新的认识的时候，艰苦卓绝的纠错过程。但是，这并不显然的表示，后来的模型就优于早期模型。我们这里的展示的顺序并不表示市场对其认同度的排序。这里的顺序更多的按照历史顺序以及逻辑的复杂程度。作为一个简单的回顾，这里我们忽略了很多模型检验的部分。这里我们只讨论了波动率微笑的静态形态，我们没有深入涉及波动率的时间结构，更没有涉及波动率的动态。关于波动率的动态 (dynamics of volatility) 对模型评价的影响，可以参考例如 [19] 。在那里 Hagan 等人设计新的模型动机之一，就是为了去捕捉市场上波动率动态正确方向。

20.1.2 接口

OptionUtilities.BSMPrice(optionType, strike, spot, riskFree, dividend, volatility, maturity, rawOutput = False)

Black-Scholes-Merton 期权定价公式，见: [6]

参数

- optionType (int) – 期权类型，1 for CALL 或者 -1 for PUT
- strike (float) – 行权价格
- spot (float) – 标的股票价格
- riskFree (float) – 无风险利率
- dividend (float) – 股票红利率
- volatility (float) – 波动率
- maturity (float) – 到期时间 (年)
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 期权价格

返回类型 pandas.DataFrame 或者 numpy.array

OptionUtilities.BSMImpliedVolatity(optionType, strike, spot, riskFree, dividend, maturity, quotePrice, rawOutput = False)

Black-Scholes-Merton Implied volatility 期权隐含波动率，见: [6]

参数

- optionType (int) – 期权类型，1 for CALL 或者 -1 for PUT
- strike (float) – 行权价格
- spot (float) – 标的股票价格
- riskFree (float) – 无风险利率
- dividend (float) – 股票红利率
- maturity (float) – 到期时间 (年)
- quotePrice (float) – 期权报价

- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 期权隐含波动率

返回类型 pandas.DataFrame 或者 numpy.array

OptionUtilities.BachelierPrice(optionType, strike, forward, riskFree, volatility, maturity, rawOutput = False)

Bachelier 期权定价公式，见: [2]

参数

- optionType (int) – 期权类型，1 for CALL 或者 -1 for PUT
- strike (float) – 行权价格
- forward (float) – 标的远期价格
- riskFree (float) – 无风险利率
- volatility (float) – 波动率
- maturity (float) – 到期时间 (年)
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 期权价格

返回类型 pandas.DataFrame 或者 numpy.array

OptionUtilities.BachelierImpliedVolatility(optionType, strike, forward, riskfree, maturity, bachelierPrice, rawOutput = False)

Bachelier 隐含波动率公式，见: [9]

参数

- optionType (int) – 期权类型，1 for CALL 或者 -1 for PUT
- strike (float) – 行权价格
- forward (float) – 标的远期价格
- riskFree (float) – 无风险利率
- maturity (float) – 到期时间 (年)
- bachelierPrice (float) – 期权价格
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 期权隐含波动率

返回类型 pandas.DataFrame 或者 numpy.array

OptionUtilities.BaroneAdesiWhaleyPrice(optionType, strike, spot, riskFree, dividend, volatility, maturity, rawOutput = False)

BaroneAdesi-Whaley 美式期权逼近公式，见: [4]

参数

- optionType (int) – 期权类型，1 for CALL 或者 -1 for PUT
- strike (float) – 行权价格
- spot (float) – 标的股票价格
- riskFree (float) – 无风险利率
- dividend (float) – 股票红利率
- volatility (float) – 波动率
- maturity (float) – 到期时间 (年)
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 期权价格

返回类型 pandas.DataFrame 或者 numpy.array

OptionUtilities.DisplacedDiffussionPrice(optionType, strike, spot, displacement, riskFree, dividend, volatility, maturity, rawOutput = False)

Displaced Diffusion Model 期权定价公式，见: [34]

参数

- optionType (int) – 期权类型，1 for CALL 或者 -1 for PUT
- strike (float) – 行权价格
- spot (float) – 标的股票价格
- displacement (float) – 价格漂移
- riskFree (float) – 无风险利率
- dividend (float) – 股票红利率
- volatility (float) – 波动率
- maturity (float) – 到期时间 (年)
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 期权价格

返回类型 pandas.DataFrame 或者 numpy.array

OptionUtilities.CEVPrice(optionType, strike, spot, beta, riskFree, volatility, maturity, rawOutput = False)

Constant Elasticity of Variance Model 期权定价公式，见: [11]

参数

- optionType (int) – 期权类型，1 for CALL 或者 -1 for PUT
- strike (float) – 行权价格
- spot (float) – 标的股票价格

- beta (float) –
- riskFree (float) – 无风险利率
- volatility (float) – 波动率
- maturity (float) – 到期时间 (年)
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 期权价格

返回类型 pandas.DataFrame 或者 numpy.array

OptionUtilities.SABRImpliedVolSurface(strike, forward, maturity, alpha, beta, nu, rho, rawOutput = False)

SABR Model 期权隐含波动率公式，见: [19]

参数

- strike (float) – 行权价格
- forward (float) – 标的远期价格
- maturity (float) – 到期时间 (年)
- alpha (float) – 波动率初始值
- beta (float) – 弹性系数
- nu (float) – 波动率的波动率
- rho (float) – 波动率过程与标的资产过程相关系数
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 期权隐含波动率

返回类型 pandas.DataFrame 或者 numpy.array

OptionUtilities.NoArbSABRImpliedVolSurface(strike, forward, maturity, alpha, beta, nu, rho, rawOutput = False)

No Arbitrage SABR Model 期权隐含波动率公式，见: [14]

参数

- strike (float) – 行权价格
- forward (float) – 标的远期价格
- maturity (float) – 到期时间 (年)
- alpha (float) – 波动率初始值
- beta (float) – 弹性系数
- nu (float) – 波动率的波动率
- rho (float) – 波动率过程与标的资产过程相关系数

- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 期权隐含波动率

返回类型 pandas.DataFrame 或者 numpy.array

OptionUtilities.HestonPrice(optionType, strike, spot, riskFree, dividend, maturity, v0, kappa, theta, sigma, rho, rawOutput = False)

Heston 期权价格，见: [23]

参数

- optionType (int) – 期权类型，1 for CALL 或者 -1 for PUT
- strike (float) – 行权价格
- spot (float) – 标的股票价格
- riskFree (float) – 无风险利率
- dividend (float) – 股票红利率
- maturity (float) – 到期时间 (年)
- v0 (float) –
- kappa (float) –
- theta (float) –
- sigma (float) –
- rho (float) –
- rawOutput (bool) – False 生成 pandas 格式 DataFrame，True 原始的 numpy 数组

返回 期权价格

返回类型 pandas.DataFrame 或者 numpy.array

20.2 波动率

市场上期权价格一般以隐含波动率的形式报出，一般来讲在市场交易时间，交易员可以看到类似的波动率矩阵 (Volatilitie Matrix):

表 20.2: 2015 年 3 月 3 日 10 时波动率矩阵

行权价	2015/3/25	2015/4/25	2015/6/25	2015/9/25
2.20	32.56%	29.75%	29.26%	27.68%
2.30	28.84%	29.20%	27.39%	26.51%
2.40	27.66%	27.35%	25.89%	25.28%
2.50	26.97%	25.57%	25.80%	25.41%
2.60	27.77%	24.82%	27.34%	24.81%

交易员可以看到市场上离散值的信息，但是如果可以获得一些隐含的信息更好：例如，在 2015 年 6 月 25 日以及 2015 年 9 月 25 日之间，波动率的形状会是怎么样的？

20.2.1 波动率曲面

上面问题的答案引出波动率曲面（Volatility Surface）的概念。简单的讲，波动率曲面是将市场观察的离散点以某种算法扩展至时间以及价格二维曲面的方法。在 CAL 中有数种内置方法帮助用户构造波动率曲面。

方差 Black 方差曲面插值（BlackVarianceSurface）

这是最直接的波动率曲面构造方法，使用双线性插值 (Bilinear Interpolation) 同时在时间方向和价格方向进行曲面插值。更精确的讲，在行权价个方向：

$$\sigma^2(K,t) = \frac{K_2 - K}{K_2 - K_1}\sigma^2(K_1,t) + \frac{K - K_1}{K_2 - K_1}\sigma^2(K_2,t)$$

这里 $K_1 \le K \le K_2$.

在时间方向：

$$\sigma^2(K)t = \frac{t_2 - t}{t_2 - t_1}\sigma^2(K,t_1)t_1 + \frac{t - t_1}{t_2 - t_1}\sigma^2(K,t_2)t_2$$

这里 $t_1 \le t \le t_2$.

这个简单模型的优势在于：

- 稳定，模型依赖少；
- 如果原始的输入数据是无套利的话，产生的波动率曲面也是无套利的；
- 市场观察点可以完美复制。

让我们首先准备输入的数据：

- eval 估值日，这里是 2015 年 3 月 3 日；
- dates 期限，市场观察点的期权到期期限；
- strikes 行权价，市场观察点的行权价；
- blackVolMatrix 市场期权波动率矩阵。

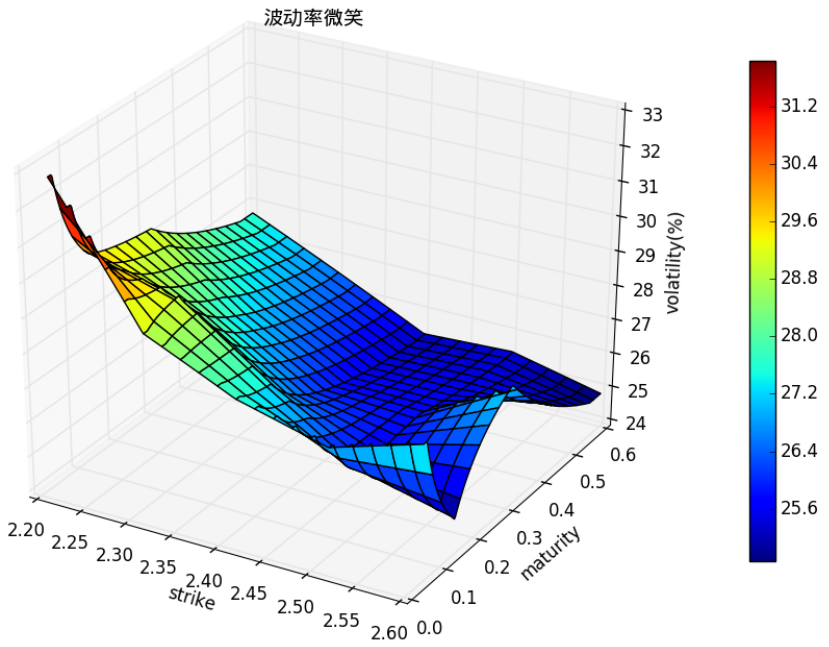
这里输入的数据就是之前展示的表格。

```
In [1]: eval = Date(2015,3,3)
   ...: SetEvaluationDate(eval)
   ...: calendar = Calendar('NullCalendar')
   ...: dates = [Date(2015,3,25), Date(2015,4,25), Date(2015,6,25), Date(2015,9,25)]
   ...: strikes = [2.2, 2.3, 2.4, 2.5, 2.6]
   ...: blackVolMatrix = [[ 0.32562851,  0.29746885,  0.29260648,  0.27679993],
   ...:                   [ 0.28841840,  0.29196629,  0.27385023,  0.26511898],
```

```
   ...:                 [ 0.27659511,  0.27350773,  0.25887604,  0.25283775],
   ...:                 [ 0.26969754,  0.25565971,  0.25803327,  0.25407669],
   ...:                 [ 0.27773032,  0.24823248,  0.27340796,  0.24814975]]
   ...:
```

```
In [7]: surface = BlackVarianceSurface(eval, calendar, dates, strikes, blackVolMatrix, 'Actual/365 (Fixed)')
   ...: surface.volatilityProfileFromDates(strikes, dates)
   ...:
Out[8]:
       2015-03-25  2015-04-25  2015-06-25  2015-09-25
2.2000      0.3256      0.2975      0.2926      0.2768
2.3000      0.2884      0.2920      0.2739      0.2651
2.4000      0.2766      0.2735      0.2589      0.2528
2.5000      0.2697      0.2557      0.2580      0.2541
2.6000      0.2777      0.2482      0.2734      0.2481
```

```
In [9]: fig = pylab.figure(figsize = (12,8))
   ...: surface.plotSurface(startStrike = 2.2, endStrike = 2.6, startMaturity = 0.05, maturity = 0.6)
   ...:
```



像我们之前介绍的，我们有其他的模型构造波动率曲面，这里我们介绍两种。

SABR **波动率曲面 (** SABRCalibratedVolSrurface **)**

在这个波动率曲面的构造方法中，我们使用 SABR 模型作为基础。SABR 模型我们已经在期权函数 中有所涉及。更多的信息可以参考原始的论文 [19] 。

波动率曲面的构造分成两步：

1. 构造每个期限上的 SABR 模型

SABR 模型在每个期限上可以由 4 个参数确定: α, β, ν, ρ 。我们使用经典的最小误差平方和来确定，具体的说：

我们有以下的 SABR 模型函数，产生隐含波动率：

$$\sigma_{SABR}(K, T; \alpha, \beta, \nu, \rho)$$

我们希望找到具体的参数 α, β, ν, ρ 使得与市场报价的误差最小：

$$\arg\min_{\alpha,\beta,\nu,\rho} \sum_{i=0}^{n} (\sigma_{SABR}(K, T; \alpha, \beta, \nu, \rho) - \sigma(K, T))^2$$

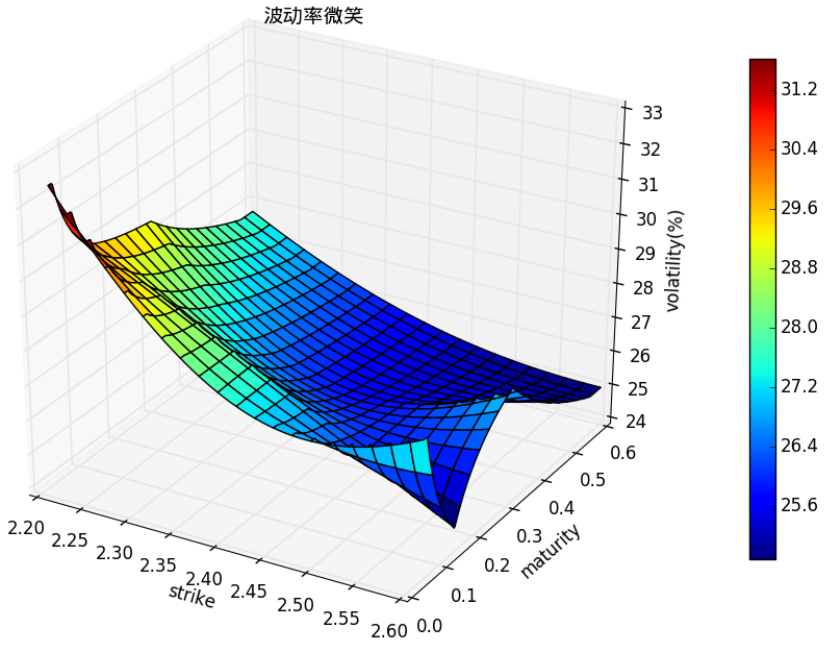
2. 双线性插值产生整个波动率曲面

当我们计算得到需要的每个期限上的 SABR 模型，然后用 SABR 波动率代替输入的波动率，组成新的波动率矩阵。最后使用上文中介绍的基于方差矩阵的双线性插值产生波动率曲面。

```
In [11]: spot = 2.40
   ....: forwards = [spot for d in dates]
   ....:
```

```
In [13]: surface = SABRCalibratedVolSruface(strikes, forwards, dates, blackVolMatrix)
   ....: surface.volatilityProfileFromDates(strikes, dates)
   ....:
Out[14]:
        2015-03-25  2015-04-25  2015-06-25  2015-09-25
2.2000      0.3227      0.2997      0.2942      0.2767
2.3000      0.2928      0.2877      0.2714      0.2644
2.4000      0.2740      0.2742      0.2591      0.2557
2.5000      0.2700      0.2599      0.2601      0.2508
2.6000      0.2784      0.2450      0.2721      0.2494
```

```
In [15]: fig = pylab.figure(figsize = (12,8))
   ....: surface.plotSurface(startStrike = 2.2, endStrike = 2.6, startMaturity = 0.05, maturity = 0.6)
   ....:
```



SVI 波动率曲面 (SviCalibratedVolSruface)

在这个波动率曲面的构造方法中，我们使用 SVI 模型作为基础。SVI 模型由 Gatheral 提出 [17] ，具体的：

$$\text{var}(K; a, b, \sigma, \rho, m) = a + b\left\{\rho(k-m) + \sqrt{(k-m)^2 + \sigma^2)}\right\}$$

波动率曲面的构造分成两步：

1. 构造每个期限上的 SVI 模型

SVI 模型在每个期限上可以由 5 个参数确定: a, b, σ, ρ, m 。我们使用经典的最小误差平方和来确定，具体的说：

我们有以下的 SVI 模型函数，产生隐含方差：

$$\text{var}_{SVI}(K; a, b, \sigma, \rho, m)$$

我们希望找到具体的参数 a, b, σ, ρ, m 使得与市场报价的误差最小：

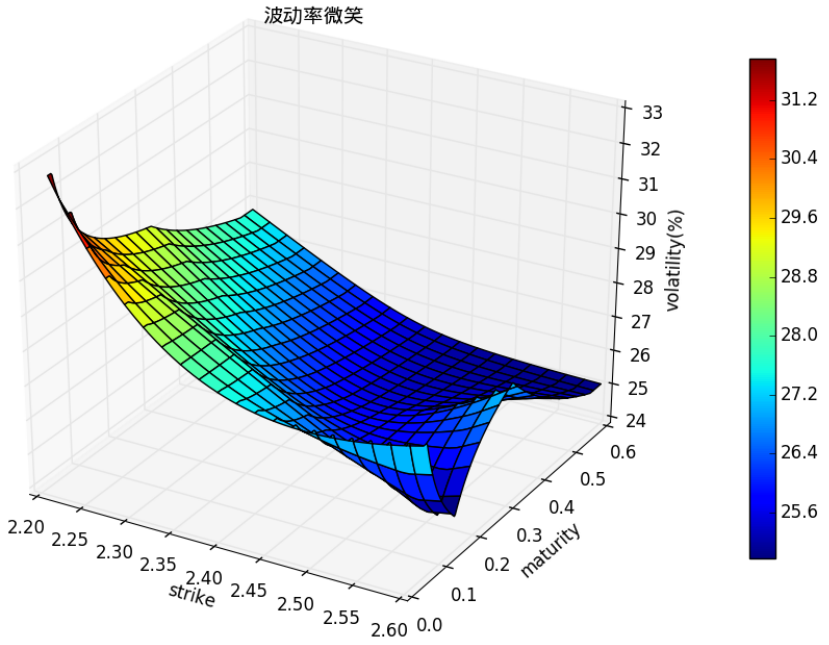
$$\arg\min_{\alpha,\beta,\nu,\rho} \sum_{i=0}^{n} (\text{var}_{SVI}(K; a, b, \sigma, \rho, m) - \text{var}(K))^2$$

2. 双线性插值产生整个波动率曲面

当我们计算得到需要的每个期限上的 SVI 模型，然后用 SVI 波动率代替输入的波动率，组成新的波动率矩阵。最后使用上文中介绍的基于方差矩阵的双线性插值产生波动率曲面。

```
In [17]: surface = SviCalibratedVolSruface(strikes, forwards, dates, blackVolMatrix)
   ....: surface.volatilityProfileFromDates(strikes, dates)
   ....:
Out[18]:
        2015-03-25  2015-04-25  2015-06-25  2015-09-25
2.2000      0.3253      0.3020      0.2926      0.2772
2.3000      0.2896      0.2873      0.2739      0.2642
2.4000      0.2746      0.2726      0.2589      0.2543
2.5000      0.2719      0.2577      0.2580      0.2511
2.6000      0.2765      0.2482      0.2734      0.2503
```

```
In [19]: fig = pylab.figure(figsize = (12,8))
   ....: surface.plotSurface(startStrike = 2.2, endStrike = 2.6, startMaturity = 0.05, maturity = 0.6)
   ....:
```



20.2.2 接口

class OptionUtilities.BlackVarianceSurface

___init___(self, referenceDate, calendar, dates, strikes, blackVolMatrix, dayCounter)

参数

- referenceDate (Dates.Date) – 基准日

- calendar (Calendars.Calendar) – 工作日历
- dates (list) – 到期时间序列
- strikes (list) – 行权价序列
- blackVolMatrix (matrix) – 波动率矩阵
- dayCounter (DayCounters.DayCounter) – 天数计数惯例

class OptionUtilities.SviCalibratedVolSruface

___init___(self, strikes, forwards, dates, volatilites, dayCounter = 'Actual/365 (Fixed)', a = None, b = None, sigma = None, rho = None, m = None)

参数

- strikes (list) – 行权价序列
- forwards (list) – 远期价格序列
- dates (list) – 到期时间序列
- volatilities (matrix) – 波动率矩阵
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- a (float) – 模型参数 a
- b (float) – 模型参数 b
- sigma (float) – 模型参数 σ
- rho (float) – 模型参数 ρ
- m (float) – 模型参数 m

class OptionUtilities.SABRCalibratedVolSruface

___init___(self, strikes, forwards, dates, volatilites, dayCounter = 'Actual/365 (Fixed)', alpha = None, beta = None, nu = None, rho = None)

参数

- strikes (list) – 行权价序列
- forwards (list) – 远期价格序列
- dates (list) – 到期时间序列
- volatilities (matrix) – 波动率矩阵
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- alpha (float) – 模型参数 α
- beta (float) – 模型参数 β
- nu (float) – 模型参数 ν

- rho (float) – 模型参数 ρ

20.3 期权

20.3.1 行权方式

接口

class Instruments.ExerciseProxy

exerciseType(self)

返回 行权类型

返回类型 int, 0: American, 1: Bermudan, 2: European

dates(self)

返回 行权日期列表

返回类型 list of Dates.Date

class Instruments.Exercise(Exercises.ExerciseProxy)

class Instruments.EuropeanExercise(Exercises.Exercise)
欧式行权

___init___(self, exerciseDate)

参数 exerciseDate (Dates.Date) – 行权日

返回 欧式行权对象

返回类型 Exercises.EuropeanExercise

class Instruments.AmericanExercise(Exercises.Exercise)
美式行权

___init___(self, earliestDate, latestDate, payOffAtExpiry = False)

参数

- earliestDate (Dates.Date) – 最早行权日期
- latestDate (Dates.Date) – 最晚行权日期
- payOffAtExpiry (bool) – 到期时偿付？

返回 美式行权对象

返回类型 Exercises.AmericanExercise

class Instruments.BermudanExercise(Exercises.Exercise)
百慕大式行权

___init___(self, exerciseDates, payOffAtExpiry = False)

参数

- exerciseDates (list of Dates.Date) – 行权日列表
- payOffAtExpiry (bool) – 到期时偿付？

返回 百慕大式行权对象

返回类型 Exercises.BermudanExercise

class Instruments.CallabilityPrice

Dirty
使用全价

Clean
使用净价

___init___(self, amount, type = Clean)
可赎回或回购债券价格条款

参数

- amount (float) –
- type (CallabilityPrice.Type) – 价格类型，Dirty 或者 Clean

返回 可赎回或回购债券价格条款对象

返回类型 Exercises.CallabilityPrice

amount(self)
返回行权价

type(self)
返回价格类型

class Instruments.Callability
可赎回或回购债券条款

Call
可赎回行权

Put
可回售行权

___init___(self, callabilityPrice, type, date)

参数

- callabilityPrice (Exercises.CallabilityPrice) – 可赎回或回购债券价格条款
- type (Callability.Type) – 行权类型，例如 Callability.Call
- date (Dates.Date) – 行权日

返回 可赎回或回购债券条款

返回类型 Exercises.Callability

type(self)
返回行权类型

date(self)
返回行权日

20.3.2 偿付方式

接口

class Instruments.PayoffProxy

__call__(self, price)
偿付函数

参数 price (float) – 标的价格

返回 偿付

返回类型 float

class Instruments.PlainVanillaPayoff(Payoffs.PayoffProxy)

__init__(self, type, strike)
简单偿付类型

参数

- type (int) – 偿付类型，1：Call，-1：Put
- strike (float) – 行权价

返回 简单偿付对象

返回类型 Payoffs.PlainVanillaPayoff

20.3.3 期权

基本上来讲，常用期权的结构无外乎可以分解为两部分：

- 行权结构 + 偿付类型

我们已经在之前的两部分分别介绍了：行权方式 以及偿付方式 。所以到这里，引入期权类型变得水到渠成。

1. 美式简单期权

定义行权方式：

```
In [1]: startDate = Date(2014, 1, 1)
   ...: maturityDate = Date(2014, 12, 31)
   ...: exercise = AmericanExercise(startDate, maturityDate)
   ...:
```

定义偿付方式：

```
In [4]: strike = 100.0
   ...: payoff = PlainVanillaPayoff(Option.Call, strike)
   ...:
```

最后期权对象：

```
In [6]: option = VanillaOption(payoff, exercise)
```

2. 欧式障碍期权

定义行权方式

```
In [7]: maturityDate = Date(2014, 12, 31)
   ...: exercise = EuropeanExercise(maturityDate)
   ...:
```

使用上例中一样的偿付方式，直接开始定义期权对象

```
In [9]: type = Barrier.DownOut
   ...: barrier = 100.0
   ...: rebate = 0.0
   ...: barrietOption = BarrierOption(type,
   ...:                               barrier,
   ...:                               rebate,
   ...:                               payoff,
   ...:                               exercise)
   ...:
```

关于使用这些期权对象进行估值的内容，我们将在期权定价算法 中进行介绍。

接口

class Instruments.OptionProxy(Instrument.Instrument)

class Instruments.AnalyticOption(OptionProxy)

delta(self)

返回 delta

返回类型 float

gamma(self)

返回 gamma

返回类型 float

theta(self)

返回 theta

返回类型 float

thetaPerDay(self)

返回 thetaPerDay

返回类型 float

vega(self)

返回 vega

返回类型 float

rho(self)

返回 rho

返回类型 float

dividendRho(self)

返回 dividendRho

返回类型 float

strikeSensitivity(self)

返回 strikeSensitivity

返回类型 float

class Instruments.VanillaOption(Options.AnalyticOption)

___init___(self, vanillaPayoff, exercise)

简单期权类型

参数

- vanillaPayoff (Payoffs.PlainVanillaPayoff) – 简单偿付对象
- exercise (Exercises.Exercise) – 行权方式

返回 简单期权对象

返回类型 Options.VanillaOption

class Instruments.EuropeanOption(Options.VanillaOption)

___init___(self, vanillaPayoff, exercise)
欧式期权类型

参数

- vanillaPayoff (Payoffs.PlainVanillaPayoff) – 简单偿付对象
- exercise (Exercises.Exercise) – 行权方式

返回 欧式期权对象

返回类型 Options.EuropeanOption

class Instruments.Barrier

DownIn

DownOut

UpIn

UpOut

class Instruments.BarrierOption(Options.OptionProxy)

___init___(self, barrirtType, barrier, rebate, payoff, exercise)
障碍期权类型

参数

- barrirtType (Options.Barrier) – 障碍期权类型
- barrier (float) – 障碍水平
- rebate (float) – 敲出返还
- payoff (Payoffs.PayoffProxy) – 偿付对象
- exercise (Exercises.Exercise) – 行权类型

返回 障碍期权对象

返回类型 Options.BarrierOption

CHAPTER 21

期限结构

21.1 收益率曲线

21.1.1 什么是收益率曲线

收益率曲线（Yield Curve）是时间价值的描述，它反映了投资者在某一特定市场下，投资一元在未来可以获取的价值（future value，FV），或者说为了获取未来一元的价值现在需要投入的本金（net present value，NPV）。

实务上考虑，一条收益率曲线的实现往往需要满足以下的几个要素：

- 基准日（reference date）

基准日即为曲线眼中的“今天”，在此之前的日期曲线并不关心。同时在基准日，任意一元的未来价值和现值都是一元。

- 折现因子

即在某个“未来”（这里未来指的是基准日之后的日子）日子发生一元现金流的现值（NPV）。现在投资一元在未来的相同日期获得的价值（FV）即为折现因子的倒数。

- 零息利率（zero yield)

更多的时候我们谈论的是一份投资的回报率（利率），而不是直接讨论折现因子。事实上，零息利率与折现因子是收益率曲线的两种不同表述形式。如果你已经知道某个时间点的折现因子，那你也就可以计算出那个时间点的零息利率；反之亦然。

让我们用一个数学式子让这个概念更加清晰：设基准日为 t_0，观察日为 t_1 ；假设是简单复利（Simple Compounding）则：

$$D_{t_1} = (1 + r_1 \times (t_1 - t_0)),$$

其中 D_{t_1} 为折现因子，r_1 为零息利率

- 远期利率（forward rate）

在很多的情形下，我们要回答类似这样的问题：如果我 3 个月以后投资 1 元，那我 6 个月以后可以拿到多少钱？这就是被称作远期利率：在未来的一个日子，看未来的未来的投资回报。远期利率仍然只是收益

率曲线的某种表现形式，有了之前提到的折现因子或者零息利率的任意一个，我们可以推到出远期利率。关于这三者之间关系的详细讨论，请见比如: [37]

设基准日为 t_0，观察日为 t_1 ，远期日为 t_2 ，那么远期利率与折现因子的关系为：

$$D_{t_1}/D_{t_2} = (1 + f_{12} \times (t_2 - t_1))$$

其中 f_{12} 为时刻 t_1 到远期日 t_2 的远期利率。

关于零息利率与远期利率的关系，更容易用一张图去表示：

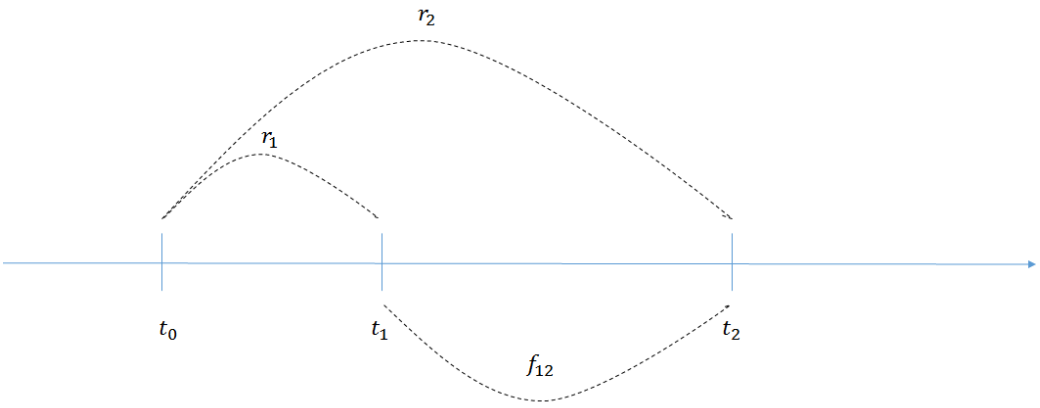


图 21.1: 零息利率与远期利率关系示意

21.1.2 中债收益率曲线

在完成了收益率曲线的抽象描述以后，让我们看看实务中收益率曲线长什么样？仅就中国市场而言，最重要的市场标杆是中债登编制的收益率曲线，见: [42]

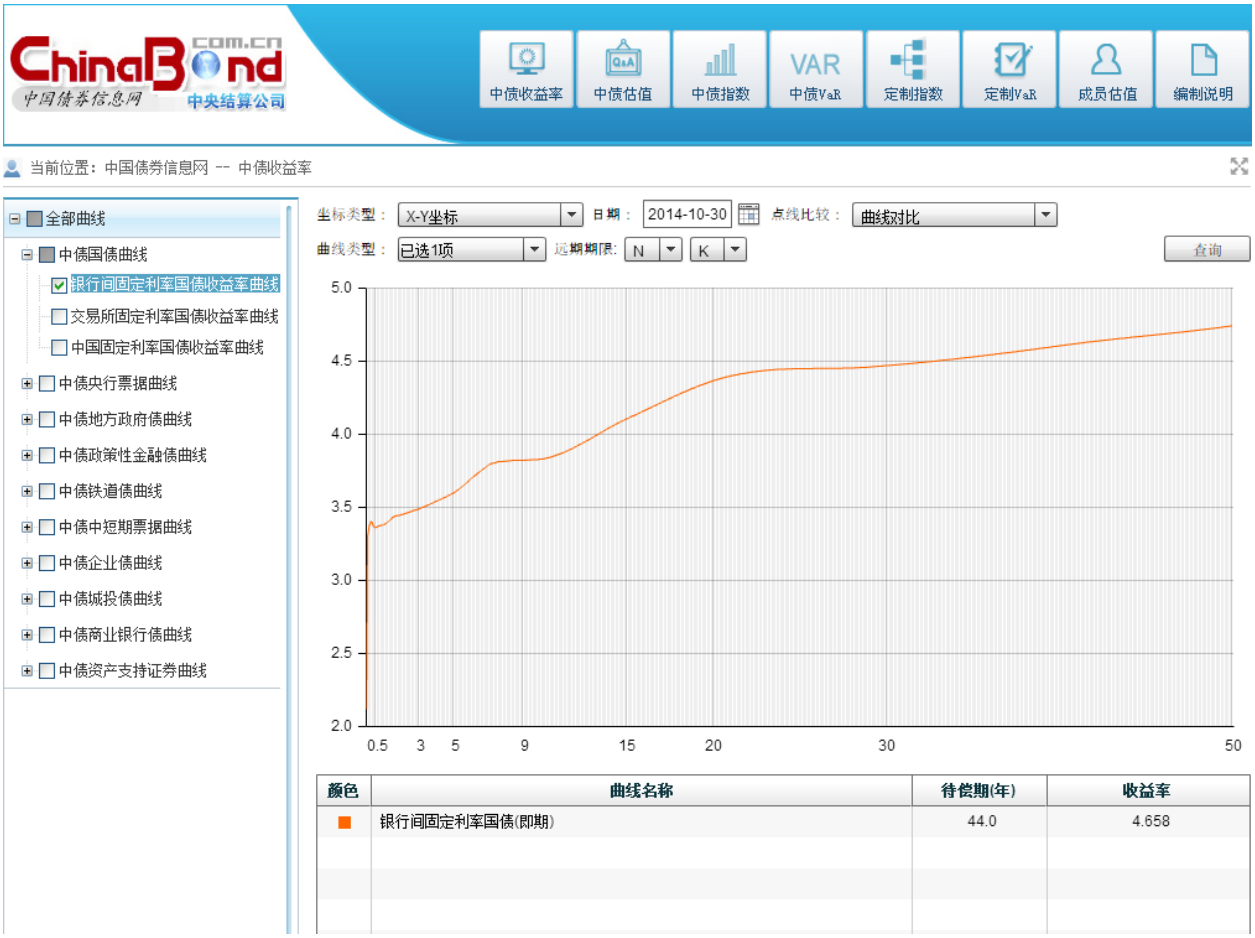


图 21.2: 中债登收益率曲线（2014 年 10 月 30 日）

上图中是中债登主页，收益率曲线工具的截图：2014 年 10 月 30 日，银行间固定利率国债，即期收益率曲线。让我们简单解释几个名词的概念。2014 年 10 月 30 日就是基准日；银行间固定利率国债，意味着这这条曲线适用于银行间市场的，固定利率，国债。即期是曲线的类型，这里我们进一步展开。中债登公布的曲线类型有 4 种：到期，即期，远期的到期，远期的即期.

- 到期

曲线上面的每个点是对应期限的固定利率国债的到期收益率。曲线上有几个关键期限点，这些点的数据来自于市场报价，其他时间点的值由插值产生。

- 即期

曲线上面的每个点是对应期限的零息利率国债的收益率，即零息收益率曲线。同样的，曲线上有几个关键期限点，这些点的数据来自于市场报价，其他时间点的值由插值产生。

- 远期的到期

相当于将未来某个时间点作为基准点时，到期收益率曲线的形状。

- 远期的即期

相当于将未来某个时间点作为基准点时，即期收益率曲线的形状。

现阶段，CAL 的收益率曲线模块可以非常方便的处理第二种以及第四种情形，尚不支持到期收益率曲线类型。但是事实上，对于同一条收益率曲线而言，它的到期形式和即期形式只是同样市场情形的不同表述。能够表述即期收益率情形，已经获取了市场全部的信息，可以推到出到期收益率。当然出于未来用户方便的考虑，我们也会很快扩展我们的模块，让收益率曲线模块支持到期形式。

21.1.3 CAL 构造收益率曲线

```
In [1]: referenceDate = Date(2014, 10 ,30)
   ...: curve = BuildCurve('Treasury.XIBE', referenceDate);
   ...:
```

这里我们使用了工厂函数 来简化我们的输入过程。后面我们会学到，如何直接从市场数据出发构造收益率曲线。

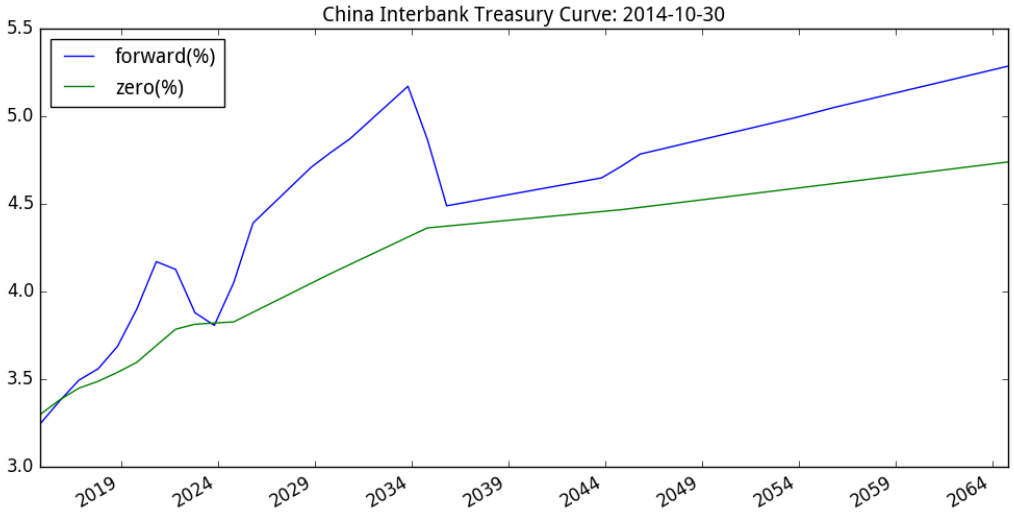
这条曲线长什么样？收益率曲线的 curveProfile 成员函数会生成一个概览性的表格，即为下表。在下表中，你可以看到，第一个日期的折现因子（discount 列）为 1，这一日期就是曲线的基准日。

```
In [3]: res = curve.curveProfile()[0::12]
   ...: res
   ...:
Out[4]:
```

	date	discount	forward(%)	zero(%)
2014-10-30	2014-10-30	1.0000	3.2446	3.2978
2015-10-30	2015-10-30	0.9673	3.3746	3.3810
2016-10-30	2016-10-30	0.9343	3.4939	3.4487
2017-10-30	2017-10-30	0.9022	3.5596	3.4879
2018-10-30	2018-10-30	0.8700	3.6876	3.5391
2019-10-30	2019-10-30	0.8380	3.9010	3.5964
2020-10-30	2020-10-30	0.8044	4.1703	3.6917
2021-10-30	2021-10-30	0.7709	4.1262	3.7849
2022-10-30	2022-10-30	0.7411	3.8791	3.8135
2023-10-30	2023-10-30	0.7135	3.8067	3.8203
2024-10-30	2024-10-30	0.6867	4.0515	3.8268
2025-10-30	2025-10-30	0.6576	4.3910	3.8818
2026-10-30	2026-10-30	0.6290	4.4969	3.9368
2027-10-30	2027-10-30	0.6010	4.6027	3.9918
2028-10-30	2028-10-30	0.5736	4.7089	4.0470
2029-10-30	2029-10-30	0.5469	4.7927	4.1021
2030-10-30	2030-10-30	0.5212	4.8707	4.1542
2031-10-30	2031-10-30	0.4961	4.9707	4.2062
2032-10-30	2032-10-30	0.4718	5.0709	4.2585
2033-10-30	2033-10-30	0.4482	5.1709	4.3106
2034-10-30	2034-10-30	0.4254	4.8700	4.3628
2035-10-30	2035-10-30	0.4068	4.4890	4.3732
2036-10-30	2036-10-30	0.3889	4.5089	4.3836

2037-10-30	2037-10-30	0.3717	4.5288	4.3939
2038-10-30	2038-10-30	0.3552	4.5486	4.4043
2039-10-30	2039-10-30	0.3394	4.5685	4.4147
2040-10-30	2040-10-30	0.3241	4.5884	4.4251
2041-10-30	2041-10-30	0.3096	4.6083	4.4354
2042-10-30	2042-10-30	0.2956	4.6281	4.4458
2043-10-30	2043-10-30	0.2822	4.6480	4.4562
2044-10-30	2044-10-30	0.2693	4.7132	4.4666
2045-10-30	2045-10-30	0.2568	4.7843	4.4801
2046-10-30	2046-10-30	0.2447	4.8102	4.4937
2047-10-30	2047-10-30	0.2332	4.8360	4.5072
2048-10-30	2048-10-30	0.2221	4.8620	4.5207
2049-10-30	2049-10-30	0.2116	4.8879	4.5343
2050-10-30	2050-10-30	0.2015	4.9138	4.5478
2051-10-30	2051-10-30	0.1918	4.9397	4.5613
2052-10-30	2052-10-30	0.1825	4.9656	4.5749
2053-10-30	2053-10-30	0.1736	4.9915	4.5885
2054-10-30	2054-10-30	0.1651	5.0205	4.6020
2055-10-30	2055-10-30	0.1570	5.0497	4.6157
2056-10-30	2056-10-30	0.1493	5.0759	4.6294
2057-10-30	2057-10-30	0.1419	5.1021	4.6431
2058-10-30	2058-10-30	0.1348	5.1283	4.6569
2059-10-30	2059-10-30	0.1280	5.1545	4.6706
2060-10-30	2060-10-30	0.1216	5.1808	4.6843
2061-10-30	2061-10-30	0.1154	5.2070	4.6980
2062-10-30	2062-10-30	0.1095	5.2332	4.7117
2063-10-30	2063-10-30	0.1039	5.2594	4.7254
2064-10-30	2064-10-30	0.0986	5.2856	4.7392

```
In [5]: Plot(res, settings = {'y':['forward(%)','zero(%)'],
   ...:                       'title': 'China Interbank Treasury Curve: 2014-10-30',
   ...:                       'figsize':(12,6)})
   ...:
```



让我们看一下市场的数据是多少，这里我们会使用 工厂函数：

```
In [6]: refDate = Date(2014,10,30)
   ...: curve = BuildCurve('Treasury.XIBE', refDate)
   ...: curve.curveProfile()
   ...:
Out[8]:
                date  discount  forward(%)  zero(%)
2014-10-30  2014-10-30    1.0000      3.2446   3.2978
2014-11-30  2014-11-30    0.9972      3.2841   3.2978
2014-12-30  2014-12-30    0.9945      3.4248   3.3767
2015-01-30  2015-01-30    0.9916      3.3656   3.4043
2015-02-28  2015-02-28    0.9890      3.2772   3.3902
2015-03-30  2015-03-30    0.9864      3.2490   3.3757
2015-04-30  2015-04-30    0.9837      3.2739   3.3606
2015-05-30  2015-05-30    0.9810      3.3352   3.3644
2015-06-30  2015-06-30    0.9782      3.3429   3.3684
2015-07-30  2015-07-30    0.9755      3.3461   3.3722
2015-08-30  2015-08-30    0.9727      3.3476   3.3752
2015-09-30  2015-09-30    0.9700      3.3533   3.3781
2015-10-30  2015-10-30    0.9673      3.3746   3.3810
2015-11-30  2015-11-30    0.9645      3.4015   3.3867
2015-12-30  2015-12-30    0.9618      3.4122   3.3923
2016-01-30  2016-01-30    0.9590      3.4233   3.3980
2016-02-29  2016-02-29    0.9563      3.4340   3.4036
2016-03-30  2016-03-30    0.9536      3.4448   3.4091
2016-04-30  2016-04-30    0.9508      3.4559   3.4148
2016-05-30  2016-05-30    0.9481      3.4666   3.4204
2016-06-30  2016-06-30    0.9453      3.4777   3.4261
```

2016-07-30	2016-07-30	0.9426	3.4884	3.4317
2016-08-30	2016-08-30	0.9398	3.4995	3.4374
2016-09-30	2016-09-30	0.9371	3.5106	3.4431
2016-10-30	2016-10-30	0.9343	3.4939	3.4487
2016-11-30	2016-11-30	0.9316	3.4729	3.4520
2016-12-30	2016-12-30	0.9289	3.4791	3.4553
2017-01-30	2017-01-30	0.9262	3.4855	3.4586
2017-02-28	2017-02-28	0.9236	3.4916	3.4617
2017-03-30	2017-03-30	0.9210	3.4978	3.4649
...	...	...	...	...
2062-05-30	2062-05-30	0.1120	5.2222	4.7060
2062-06-30	2062-06-30	0.1115	5.2244	4.7071
2062-07-30	2062-07-30	0.1110	5.2266	4.7083
2062-08-30	2062-08-30	0.1105	5.2288	4.7094
2062-09-30	2062-09-30	0.1100	5.2310	4.7106
2062-10-30	2062-10-30	0.1095	5.2332	4.7117
2062-11-30	2062-11-30	0.1091	5.2354	4.7129
2062-12-30	2062-12-30	0.1086	5.2376	4.7140
2063-01-30	2063-01-30	0.1081	5.2398	4.7152
2063-02-28	2063-02-28	0.1077	5.2419	4.7163
2063-03-30	2063-03-30	0.1072	5.2440	4.7174
2063-04-30	2063-04-30	0.1067	5.2462	4.7186
2063-05-30	2063-05-30	0.1063	5.2484	4.7197
2063-06-30	2063-06-30	0.1058	5.2506	4.7209
2063-07-30	2063-07-30	0.1053	5.2528	4.7220
2063-08-30	2063-08-30	0.1049	5.2550	4.7232
2063-09-30	2063-09-30	0.1044	5.2572	4.7243
2063-10-30	2063-10-30	0.1039	5.2594	4.7254
2063-11-30	2063-11-30	0.1035	5.2616	4.7266
2063-12-30	2063-12-30	0.1030	5.2638	4.7277
2064-01-30	2064-01-30	0.1026	5.2660	4.7289
2064-02-29	2064-02-29	0.1021	5.2681	4.7300
2064-03-30	2064-03-30	0.1017	5.2703	4.7312
2064-04-30	2064-04-30	0.1012	5.2725	4.7323
2064-05-30	2064-05-30	0.1008	5.2747	4.7334
2064-06-30	2064-06-30	0.1003	5.2769	4.7346
2064-07-30	2064-07-30	0.0999	5.2790	4.7357
2064-08-30	2064-08-30	0.0995	5.2813	4.7369
2064-09-30	2064-09-30	0.0990	5.2835	4.7381
2064-10-30	2064-10-30	0.0986	5.2856	4.7392

[601 rows x 4 columns]

平坦收益率曲线

有些时候我们需要构造一条平坦的收益率曲线，主要有这样的用途：

- 作为一条基差曲线，在大多数情景测试中，基差是常值；
- 作为期权类型定价工具的参数。在期权模型中，收益率曲线的形状并不重要，大多数时候，一条平坦的收益率曲线已经满足需求；
- 作为模型原型设计（protype）时期的测试参数使用。

CAL 中有两个函数可以满足这样的功能，它们的区别主要是 基准日的设置方式不同：

- FlatForward

FlatForward 的基准日是一个固定值，直接由用户指定：

```
In [9]: refDate = Date(2014, 10, 31)
   ...: curve = FlatForward(refDate, 0.05, 'Actual/365 (Fixed)')
   ...: curve.curveProfile()[:5]
   ...:
Out[11]:
                date  discount  forward(%)  zero(%)
2014-10-31  2014-10-31    1.0000       5.0000   5.1271
2014-11-30  2014-11-30    0.9959       5.0000   5.1271
2014-12-31  2014-12-31    0.9917       5.0000   5.1271
2015-01-31  2015-01-31    0.9875       5.0000   5.1271
2015-02-28  2015-02-28    0.9837       5.0000   5.1271
```

很清楚的可以看到，基准日被设定成了 2014 年 10 月 31 日。

- FlatForwardRefEvaDate

FlatForwardRefEvaDate 的基准日是相对于全局估值日的一个偏移。会议一下，全局估值日可以由用户使用 SetEvaluationDate 函数修改，那这意味着改变了收益率曲线的基准日。

```
In [12]: today = Date(2014, 10, 31)
   ....: SetEvaluationDate(today)
   ....: curve = FlatForwardRefEvaDate(1, 'China.IB', 0.05, 'Actual/365 (Fixed)')
   ....:
```

可以看到，我们把全局估值设为 2014 年 10 月 31 日，偏移天数设为 1 天，使用的日历为“China.IB”。这意味着基准日是全局估值日的下一工作日。由于估值日那天是周五，所以下一工作日是第二周的周一，即为 2014 年 11 月 3 日。让我们检查一下：

```
In [15]: curve.curveProfile()[:5]
Out[15]:
                date  discount  forward(%)  zero(%)
2014-11-03  2014-11-03    1.0000       5.0000   5.1271
2014-12-03  2014-12-03    0.9959       5.0000   5.1271
2015-01-03  2015-01-03    0.9917       5.0000   5.1271
2015-02-03  2015-02-03    0.9875       5.0000   5.1271
2015-03-03  2015-03-03    0.9837       5.0000   5.1271
```

完全正确。让我们来换一换全局估值日：

```
In [16]: yesterDay = Date(2014, 10, 30)
   ....: SetEvaluationDate(yesterDay)
   ....: curve.curveProfile()[:5]
   ....:
Out[18]:
              date  discount  forward(%)  zero(%)
2014-10-31 2014-10-31    1.0000      5.0000   5.1271
2014-11-30 2014-11-30    0.9959      5.0000   5.1271
2014-12-31 2014-12-31    0.9917      5.0000   5.1271
2015-01-31 2015-01-31    0.9875      5.0000   5.1271
2015-02-28 2015-02-28    0.9837      5.0000   5.1271
```

同样的，基准日跟随估值日产生变化。

插值收益率曲线

除了上面收益率平坦的情形，用户在大多数的情况下需要构造的是一条曲线。一般来说，用户会在市场上观察到几个关键期限的值（例如，折现因子，零息利率，远期利率），然后以这几个点作为基石，构造收益率曲线，这一过程，成为曲线插值。关于收益率曲线插值的详细回顾，请参考: [20] 。本节主要围绕 CAL 的 API 函数InterpolatedYieldCurve 简单介绍一些这方面的概念。假设我们有下面的数据：

表 21.1: 假设的市场收益水平

期限（年）	0	1	2	3	5	7	10
零息利率	4.0	4.2	4.3	4.7	5.4	5.7	6.0
贴现因子	1.000000	0.959693	0.919245	0.871284	0.768771	0.678383	0.558395

上面的例子，就是同样一条收益率曲线的两种表达方式，分别通过零息利率以及贴现因子。

我们可以首先尝试按照零息利率水平进行插值：

```
In [19]: tenors = ['0Y', '1Y', '2Y', '3Y', '5Y', '7Y', '10Y']
   ....: zeros = [0.040, 0.042, 0.043, 0.047, 0.054, 0.057, 0.060]
   ....: today = Date(2014, 10, 31)
   ....: dates = [today + Period(tenor) for tenor in tenors]
   ....: interpCurve = InterpolatedYieldCurve(dates, zeros,
   ....:                                      dayCounter = 'Actual/Actual (ISMA)',
   ....:                                      compounding = Compounding.Compounded,
   ....:                                      frequency = Frequency.Annual)
   ....:
```

上面的代码，使用默认设置，按照线性方式（linear）在零息利率（zero）上做插值。

```
In [24]: interpCurve.curveProfile(dc = 'Actual/Actual (ISMA)')[0:121:12]
Out[24]:
              date  discount  forward(%)  zero(%)
2014-10-31 2014-10-31    1.0000      3.9221   4.0000
2015-10-31 2015-10-31    0.9597      4.2582   4.2000
```

2016-10-31	2016-10-31	0.9192	4.6888	4.3000
2017-10-31	2017-10-31	0.8713	5.6668	4.7000
2018-10-31	2018-10-31	0.8212	6.2588	5.0494
2019-10-31	2019-10-31	0.7688	6.4475	5.4000
2020-10-31	2020-10-31	0.7232	6.2540	5.5499
2021-10-31	2021-10-31	0.6784	6.3715	5.7000
2022-10-31	2022-10-31	0.6370	6.3937	5.7999
2023-10-31	2023-10-31	0.5970	6.5827	5.8999
2024-10-31	2024-10-31	0.5584	6.7716	6.0000

从上面的表中我们可以看到两件事情：

- 关键期限点上的值是精确匹配输入，其他点的值可以验证是线性插值产生；
- 零息利率与折现因子确实是同一条曲线的两种表现形式，用户并没有输入折现因子，但是输出的折现因子值精确匹配输入表格的第三行。

回头再看一下用折现因子作为插值的标的：

```
In [25]: discounts = [1.000000, 0.959693, 0.919245,
   ....:              0.871284, 0.768771, 0.678383, 0.558395]
   ....: interpCurve = InterpolatedYieldCurve(dates, discounts,
   ....:                                      dayCounter = 'Actual/Actual (ISMA)',
   ....:                                      trait = 'discount')
   ....: interpCurve.curveProfile(dc = 'Actual/Actual (ISMA)')[0:121:12]
   ....:
Out[27]:
                date  discount  forward(%)  zero(%)
2014-10-31 2014-10-31    1.0000       4.0307   4.1130
2015-10-31 2015-10-31    0.9597       4.2073   4.2000
2016-10-31 2016-10-31    0.9192       4.8088   4.3000
2017-10-31 2017-10-31    0.8713       5.6938   4.7000
2018-10-31 2018-10-31    0.8200       6.2506   5.0855
2019-10-31 2019-10-31    0.7688       6.2730   5.4000
2020-10-31 2020-10-31    0.7236       6.2459   5.5405
2021-10-31 2021-10-31    0.6784       6.2789   5.7000
2022-10-31 2022-10-31    0.6384       6.2652   5.7705
2023-10-31 2023-10-31    0.5984       6.6839   5.8716
2024-10-31 2024-10-31    0.5584       7.1627   6.0000
```

不考虑可以忽略的舍入误差，相同的结论可以在折现因子上面得到。

最后，我们来看一下不同的插值方法的性质来结束本节。我们考虑使用零息利率上的线性插值以及样条（cubic）插值，还有折现因子上的对数线性（log linear）插值。为了让比较的结果更清晰，我们选择比较远期利率。

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
```

```
import pandas as pd
import seaborn as sns
sns.set(style="ticks")

# 设置基本的数据
today = Date(2014, 10 ,31)
tenors = ['0Y', '1Y', '2Y', '3Y', '5Y', '7Y', '10Y']
zeros = [0.040, 0.042, 0.043, 0.047, 0.054, 0.057, 0.060]

discounts = [1.000000, 0.959693, 0.919245,
             0.871284, 0.768771, 0.678383, 0.558395]

dates = [today + Period(tenor) for tenor in tenors]

# 零息利率 + 线性插值
linearZero = InterpolatedYieldCurve(dates, zeros,
                                    dayCounter = 'Actual/Actual (ISMA)',
                                    compounding = Compounding.Compounded,
                                    frequency = Frequency.Annual)

# 零息利率 + 样条插值
cubicZero = InterpolatedYieldCurve(dates, zeros,
                                    dayCounter = 'Actual/Actual (ISMA)',
                                    interp = 'cubic',
                                    compounding = Compounding.Compounded,
                                    frequency = Frequency.Annual)

# 折现因子 + 对数线性插值
logLinerDiscount = InterpolatedYieldCurve(dates, discounts,
                                    dayCounter = 'Actual/Actual (ISMA)',
                                    trait = 'discount',
                                    interp = 'loglinear',
                                    compounding = Compounding.Compounded,
                                    frequency = Frequency.Annual)

linearZeroProfile = linearZero.curveProfile(dc = 'Actual/Actual (ISMA)')
cubicZeroProfile = cubicZero.curveProfile(dc = 'Actual/Actual (ISMA)')
logLinearDiscountProfile = logLinerDiscount.curveProfile(dc = 'Actual/Actual (ISMA)')

data = pd.DataFrame()

data['Zero (linear)'] = linearZeroProfile['forward(%)']
data['Zero (cubic)'] = cubicZeroProfile['forward(%)']
data['Discount (log linear)'] = logLinearDiscountProfile['forward(%)']

data.plot(figsize = (12,6))
```

21.1.4 从市场报价校正曲线（Curve Calibration）

收益率曲线是市场融投资成本水平的描述，所以最自然的想法：如何直接从市场融资工具的价格水平中，直接获取收益率曲线的信息？这个过程，我们称之为 收益率曲线校正（curve calibration）：构造收益率曲线，使得其估值结果，恰好符合市场水平。流程如下：

- 选择关心的市场

这里 市场意味着处于类似水平的融资水平的市场工具的集合。在这一概念下，例如，国债与企业债显然属于不同的市场。国债属于国家背书，享受最高的信用水平；企业债的信用水平依赖于企业的财务状况，信用水平较低，有违约风险。显然在这一情形下，企业债工具的融资水平显然不能与国债的融资水平混为一谈。

- 挑选市场基准工具

即便在单一市场，也可能有成千上百的金融工具。以国债为例，2014 年 11 月 4 日，市场上未到期债券大概有 241 支（限于银行间市场）。如果让整个收益率曲线拟合所有的债券，会面临下面的几个问题：

1. 拟合过程不稳定，甚至会失败；
2. 过拟合的问题。收益率曲线中吸收了过多的数据噪声；
3. 收益率曲线没有解释能力（Fit more，explain less）。

为了规避以上的几个问题，市场数据必须要进行挑选，例如，以下几个要素要考虑：

1. 流动性是否充分？流动性高的产品，价格中的市场噪音的成分少；
2. 市场存量是否足够？存量大的产品，市场代表性强；
3. 合理分配期限，各个到期期限的产品都要有所挑选，代表了收益率曲线的不同部分。

一般来说，单一市场选择的市场工具，为十支左右。

- 校正至市场

这一步中，根据我们选取的市场工具的报价，构造收益率曲线，完全满足市场信息。这一步往往通过数值优化等算法实现，一个经典的算法框架称之为拔靴法（Bootstrap）。关于这方面内容的介绍，可以参考: [3] 或者 [1]

这里我们会给出一个使用 CAL 函数完成这一任务的例子。首先我们选择市场数据，这里选择的是国债：

```
In [28]: codes = ['120007.XIBE', '120017.XIBE', '140022.XIBE',
   ....:          '140015.XIBE', '140004.XIBE', '130023.XIBE',
   ....:          '140008.XIBE', '130020.XIBE', '140013.XIBE']
   ....: prices = [99.7613, 99.6708, 100.3650,
   ....:           100.9778, 100.6094, 100.5826,
   ....:           102.0884, 100.0731, 102.0485]
   ....: data = pd.DataFrame({'Market':prices}, index = codes)
   ....:
```

这里 codes 是所选证券的代码，prices 是市场报价。

接着让我们来看，如果我们随便拿一条收益率曲线的结果：

```
In [31]: refDate = Date(2014, 11, 3)
   ....: SetEvaluationDate(refDate)
   ....: flatCurve = FlatForward(refDate, 0.05, 'Actual/Actual (ISMA)')
   ....:
```

来看一下，如果用这条收益率曲线估值的结果：

```
In [34]: bondEngine = DiscountingBondEngine(flatCurve)
   ....: bonds = BuildBond(codes)
   ....: newCodes = [bond.securityID() for bond in bonds]
   ....: calculatedPrice = []
   ....: for code, bond in zip(codes, bonds):
   ....:     bond.setPricingEngine(bondEngine)
   ....:     calculatedPrice.append(bond.cleanPrice())
   ....:
   ....: data['5% flat'] = pd.Series(calculatedPrice, newCodes)
   ....: data
   ....:
Out[40]:
              Market  5% flat
120007.XIBE  99.7613  98.8522
120017.XIBE  99.6708  98.4643
140022.XIBE 100.3650  98.6404
140015.XIBE 100.9778  97.9756
140004.XIBE 100.6094  96.9157
130023.XIBE 100.5826  96.5202
140008.XIBE 102.0884  95.5888
130020.XIBE 100.0731  94.5256
140013.XIBE 102.0485  93.8441
```

很显然的随意的指定的一条曲线，无法完全复制市场。这是我们完全可以期望的，我们还没告诉它市场长什么样。

让我们使用CalibratedYieldCurve ：

```
In [41]: instruments = BuildBondHelper(codes, prices)
   ....: calCurve = CalibratedYieldCurve(refDate,
   ....:                                  instruments,
   ....:                                  'Actual/Actual (ISMA)',
   ....:                                  'Zero',
   ....:                                  'Cubic')
   ....:
```

OK, curve is ready!

```
In [43]: bondEngine = DiscountingBondEngine(calCurve)
   ....: calculatedPrice = []
   ....: for code, bond in zip(codes, bonds):
   ....:     bond.setPricingEngine(bondEngine)
```

```
   ....:     calculatedPrice.append(bond.cleanPrice())
   ....: res = pd.Series(calculatedPrice, newCodes)
   ....: data['Calibrated'] = res
   ....: data
   ....:
Out[45]:
              Market  5% flat  Calibrated
120007.XIBE  99.7613  98.8522     99.7613
120017.XIBE  99.6708  98.4643     99.6708
140022.XIBE 100.3650  98.6404    100.3650
140015.XIBE 100.9778  97.9756    100.9778
140004.XIBE 100.6094  96.9157    100.6094
130023.XIBE 100.5826  96.5202    100.5826
140008.XIBE 102.0884  95.5888    102.0884
130020.XIBE 100.0731  94.5256    100.0731
140013.XIBE 102.0485  93.8441    102.0485
```

现在，完美的复制市场（Perfect calibration）！最后让我们回味一下这条曲线的形状吧：

In [46]: calCurve.curveProfile()

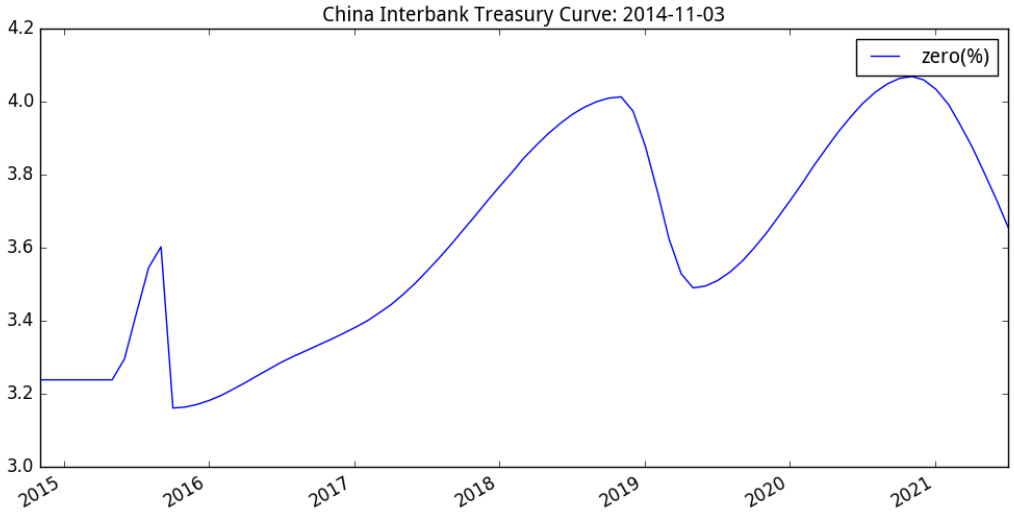
Out[46]:

	date	discount	forward(%)	zero(%)
2014-11-03	2014-11-03	1.0000	3.1869	3.2382
2014-12-03	2014-12-03	0.9973	3.1869	3.2382
2015-01-03	2015-01-03	0.9947	3.1869	3.2382
2015-02-03	2015-02-03	0.9921	3.1869	3.2382
2015-03-03	2015-03-03	0.9894	3.1869	3.2382
2015-04-03	2015-04-03	0.9868	3.1869	3.2382
2015-05-03	2015-05-03	0.9842	3.1870	3.2382
2015-06-03	2015-06-03	0.9813	3.9354	3.2950
2015-07-03	2015-07-03	0.9778	4.4198	3.4202
2015-08-03	2015-08-03	0.9742	4.3758	3.5455
2015-09-03	2015-09-03	0.9709	3.5355	3.6025
2015-10-03	2015-10-03	0.9719	3.1078	3.1610
2015-11-03	2015-11-03	0.9693	3.1736	3.1636
2015-12-03	2015-12-03	0.9667	3.2397	3.1709
2016-01-03	2016-01-03	0.9641	3.3072	3.1820
2016-02-03	2016-02-03	0.9614	3.3730	3.1964
2016-03-03	2016-03-03	0.9587	3.4342	3.2130
2016-04-03	2016-04-03	0.9559	3.4875	3.2311
2016-05-03	2016-05-03	0.9532	3.5301	3.2501
2016-06-03	2016-06-03	0.9503	3.5589	3.2689
2016-07-03	2016-07-03	0.9475	3.5707	3.2870
2016-08-03	2016-08-03	0.9447	3.5625	3.3034
2016-09-03	2016-09-03	0.9419	3.5841	3.3186
2016-10-03	2016-10-03	0.9391	3.6138	3.3336
2016-11-03	2016-11-03	0.9362	3.6533	3.3488

2016-12-03	2016-12-03	0.9334	3.7042	3.3647
2017-01-03	2017-01-03	0.9305	3.7680	3.3816
2017-02-03	2017-02-03	0.9275	3.8463	3.4000
2017-03-03	2017-03-03	0.9245	3.9407	3.4202
2017-04-03	2017-04-03	0.9214	4.1051	3.4438
...	...	...	...	...
2019-02-03	2019-02-03	0.8551	-2.7455	3.7511
2019-03-03	2019-03-03	0.8570	-2.2647	3.6253
2019-04-03	2019-04-03	0.8580	-0.2434	3.5286
2019-05-03	2019-05-03	0.8570	3.4298	3.4899
2019-06-03	2019-06-03	0.8543	3.9720	3.4951
2019-07-03	2019-07-03	0.8513	4.4780	3.5097
2019-08-03	2019-08-03	0.8480	4.9438	3.5327
2019-09-03	2019-09-03	0.8443	5.3649	3.5628
2019-10-03	2019-10-03	0.8404	5.7366	3.5989
2019-11-03	2019-11-03	0.8363	6.0543	3.6397
2019-12-03	2019-12-03	0.8320	6.3135	3.6841
2020-01-03	2020-01-03	0.8276	6.5096	3.7309
2020-02-03	2020-02-03	0.8230	6.6380	3.7790
2020-03-03	2020-03-03	0.8185	6.6940	3.8270
2020-04-03	2020-04-03	0.8139	6.6732	3.8739
2020-05-03	2020-05-03	0.8095	6.5708	3.9184
2020-06-03	2020-06-03	0.8051	6.3824	3.9594
2020-07-03	2020-07-03	0.8009	6.1033	3.9956
2020-08-03	2020-08-03	0.7970	5.7289	4.0258
2020-09-03	2020-09-03	0.7933	5.2547	4.0489
2020-10-03	2020-10-03	0.7900	4.6760	4.0637
2020-11-03	2020-11-03	0.7872	3.9882	4.0688
2020-12-03	2020-12-03	0.7850	2.7055	4.0596
2021-01-03	2021-01-03	0.7836	1.5437	4.0333
2021-02-03	2021-02-03	0.7830	0.5122	3.9926
2021-03-03	2021-03-03	0.7829	-0.3797	3.9398
2021-04-03	2021-04-03	0.7834	-1.1227	3.8773
2021-05-03	2021-05-03	0.7843	-1.7075	3.8077
2021-06-03	2021-06-03	0.7856	-2.1247	3.7332
2021-07-03	2021-07-03	0.7871	-2.3651	3.6564

[81 rows x 4 columns]

```
In [47]: Plot(calCurve.curveProfile(), settings = {'y':['zero(%)'],
   ....: 'title': 'China Interbank Treasury Curve: 2014-11-03',
   ....: 'figsize':(12,6)})
   ....:
```



看上去不是那么的优美:(但是这毕竟就是市场啊。。。

这里我们只展示了如何使用债券数据去校正一条收益率曲线，但是事实上，几乎所有的固定收益工具都可以用来作为校正的基础工具（instruments）。例如浮息债券，互换，远期等等。同时这里为了简单演示起见，我们都使用了*工厂函数*去构造*校正工具*。事实上，我们也可以通过下面的 api 函数手工构造。虽然这样做会更麻烦一点，但是自由度更高。

21.1.5 接口

收益率曲线

class YieldCurve.YieldCurveProxy

收益率曲线抽象基类

enableExtrapolation(self)

允许收益率曲线进行外插值

zeroYield(self, date, dayCounter = 'Actual/365 (Fixed)', comp = Compounding.Compounded, freq = Frequency.Annual, extrapolate = True)

获取零息利率

参数

- date (Dates.Date) – 目标日
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- comp (Compounding) – 复利方法
- freq (Frequency) – 复利频率
- extrapolate (bool) – 是否允许外插值

返回 从基准日到目标日的零息利率

返回类型 float

forwardRate(self, startDate, endDate, dayCounter = 'Actual/365 (Fixed)', comp = Compounding.Simple, freq = Frequency.Annual, extrapolate = True)

获取远期利率

参数

- startDate (Dates.Date) – 起始日
- endDate (Dates.Date) – 到期日
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- comp (Compounding) – 复利方法
- freq (Frequency) – 复利频率
- extrapolate (bool) – 是否允许外插值

返回 从起始日到期日远期利率

返回类型 float

instantaneousForward(self, date, extrapolate = True)

获取瞬时远期利率

参数

- date (Dates.Date) – 目标日期
- extrapolate (bool) – 是否允许外插值

返回 目标日期远期利率

返回类型 float

discount(self, date, extrapolate = True)

获取折现因子

参数

- date (Dates.Date) – 目标日期
- extrapolate (bool) – 是否允许外插值

返回 从基准日到目标日期的折现因子

返回类型 float

referenceDate(self)

返回曲线的基准日

返回 曲线基准日

返回类型 Dates.Date

dayCounter(self)
返回收益率曲线使用的天数计数惯例

返回 天数计数惯例

返回类型 DayCounters.DayCounter

maxDate(self)
返回曲线可以达到的最远日期

返回 最远日期

返回类型 Dates.Date

curveProfile(self, dc = None, comp = Compounding.Compounded, freq = Frequency.Annual)
返回曲线的折现因子，零息利率以及远期利率表格

参数

- dc (DayCounters.DayCounter) – 天数计数惯例（仅用于 zero）
- comp (Compounding) – 复利方法（仅踊跃 zero）
- freq (Frequency) – 复利频率（仅用于 zero）

返回 折现因子，零息利率以及远期利率表格

返回类型 pandas.DataFrame

class YieldCurve.InterpolatedYieldCurve(YieldCurve.YieldCurveProxy)
收益率曲线（插值产生）

___init___(self, dates, rates, dayCounter = 'Actual/365 (Fixed)', trait = 'zero', interp = 'linear', compounding = Compounding.Continuous, frequency = Frequency.Annual)
构建收益率曲线（基于插值）

参数

- dates (list of Dates.Date) – 日期序列
- rates (list) – 利率序列
- dayCounter (DayCounters.DayCounter) – 天数计数惯例
- trait (str) – 利率类型
- interp (str) – 插值方法
- compounding (Compounding) – 复利方法
- frequency (Frequency) – 复利频率

返回 收益率曲线

返回类型 YieldCurve.InterpolatedYieldCurve

class YieldCurve.FlatForward(YieldCurve.YieldCurveProxy)
平坦远期利率曲线

___init___(self, referenceDate, forward, dayCounter, compounding = Compounding.Continuous, frequency = Frequency.Annual)
构造平坦远期利率曲线

参数

- referenceDate (Dates.Date) – 曲线基准日
- forward (float) – 远期利率
- dayCounter (DayCounters.DayCounter) – 计息天数惯例
- compounding (Compounding) – 复利方法
- frequency (Frequency) – 复利频率

返回 平坦远期利率曲线

返回类型 YieldCurve.FlatForward

class YieldCurve.FlatForwardRefEvaDate(YieldCurve.YieldCurveProxy)
平坦远期利率曲线 (基准日基于全局估值日)

___init___(self, settlementDays, calendar, forward, dayCounter, compounding = Compounding.Continuous, frequency = Frequency.Annual)
构造平坦远期利率曲线 (基准日基于全局估值日)

参数

- settlementDays (int) – 结算天数
- calendar (Calendars.Calendar) – 工作日历日历
- forward (float) – 远期利率
- dayCounter (DayCounters.DayCounter) – 计息天数惯例
- compounding (Compounding) – 复利方法
- frequency (Frequency) – 复利频率

返回 平坦远期利率曲线

返回类型 YieldCurve.FlatForwardRefEvaDate

class YieldCurve.ForwardSpreadedTermStructure(YieldCurve.YieldCurveProxy)
基于远期的带利差收益率曲线

___init___(self, baseYieldCurve, spread)
构造基于远期的带利差收益率曲线

参数

- baseYieldCurve (YieldCurve.YieldCurveProxy) – 基础收益率曲线
- spread (float) – 利差

返回 基于远期的带利差收益率曲线

返回类型 YieldCurve.ForwardSpreadedTermStructure

class YieldCurve.CalibratedYieldCurve(YieldCurve.YieldCurveProxy)

基于市场报价构造收益率曲线

___init___(self, referenceDate, instruments, dayCounter, trait = 'Forward', interp = 'Backward-Flat')

参数

- referenceDate (Dates.Date) –
- instruments (list of RateHelper) –
- dayCounter (DayCounters.DayCounter) –
- trait (str) – 标的的利率类型，支持的输入有:'forward','discount','zero'
- interp (str) – 插值类型，支持的输入有：'linear', 'loglinear', 'cubic', 'ForwardFlat', 'BackwardFlat'

返回 基于市场报价构造的收益率曲线

返回类型 YieldCurve.CalibratedYieldCurve

校正工具 (calibration helper **)**

模型

22.1 短期利率模型

本节的主要介绍基于 [8]

22.1.1 单因子短期利率模型

单因子短期利率模型是基于短期利率的随机过程满足某个 1 维随机过程的模型:

$$dr(t) = \mu(t, r(t))dt + \sigma(t, r(t))dW(t) \tag{22.1}$$

1. Vasciek **模型**

$$dr(t) = k(\theta - r(t))dt + \sigma dW(t) \tag{22.2}$$

2. Cox-Ingersoll-Ross **模型**

$$dr(t) = k(\theta - r(t))dt + \sigma\sqrt{r(t)}dW(t) \tag{22.3}$$

3. Hull-White **模型**

$$dr(t) = (\theta(t) - ar(t))dt + \sigma dW(t) \tag{22.4}$$

4. Black-Karasinski **模型**

$$dln(r(t)) = (\theta(t) - aln(r(t)))dt + \sigma dW(t) \tag{22.5}$$

5. **扩展** Cox-Ingersoll-Ross **模型**

$$dr(t) = (\theta(t) - ar(t))dt + \sigma\sqrt{r(t)}dW(t) \tag{22.6}$$

5. **扩展** Hull-White **模型**

$$dr(t) = (\theta(t) - a(t)r(t))dt + \sigma(t)dW(t) \tag{22.7}$$

模型需要和计算算法共同工作，就像我们在产品-模型-算法周期 中描述的那样。现阶段，我们提供三叉树算法 (Trinomial Tree)，该算法在 Hull-White 的经典论文 [26] 中有非常详尽的描述。

22.1.2 两因子短期利率模型

1. G2++ 模型

$$r(t) = x(t) + y(t) + \phi(t) \tag{22.8}$$

其中:

$$dx(t) = -ax(t)dt + \sigma dW_1(t)$$

$$dy(t) = -by(t)dt + \eta dW_2(t)$$

$$dW_1(t)dW_2(t) = \rho dt$$

22.1.3 接口

class ShortRateModels.ShortRateModel
短期利率模型基类

class ShortRateModels.Vasicek(ShortRateModels.ShortRateModel)
Vasicek 短期利率模型，见 [39]

class ShortRateModels.HullWhite(ShortRateModels.ShortRateModel)
单因子 Hull-White 短期利率模型，见 [25]

class ShortRateModels.BlackKarasinski(ShortRateModels.ShortRateModel)
Black-Karasinski 短期利率模型，见 [5]

class ShortRateModels.GeneralizedHullWhite(ShortRateModels.ShortRateModel)
扩展单因子 Hull-White 短期利率模型

class ShortRateModels.CoxIngersollRoss(ShortRateModels.ShortRateModel)
Cox-Ingersoll-Ross 短期利率模型，见 [12]

class ShortRateModels.ExtendedCoxIngersollRoss(ShortRateModels.ShortRateModel)
扩展的 Cox-Ingersoll-Ross 短期利率模型。

class ShortRateModels.G2(ShortRateModels.ShortRateModel)
二因子 G2 短期利率模型，见 [8]

CHAPTER 23

定价引擎

class PricingEngine.PricingEngine

23.1 债券定价算法

23.1.1 折现定价模型

当谈到固定收益定价模型的时候，有些作者会告诉我们：这就是折现！[35] 。我们无法完全赞同这一观点，还是存在很多情形使得单纯的折现不能完全反映市场的现状。但是，不可否认的，折现定价模型是市场上最常用的模型，同时也应该是最重要。任何金融计算库都面临着如何正确、高效的实现经典的折现定价模型。以下部分，我们会详细阐述 CAL 中的折现定价实现。

不考虑违约风险，债券是一组现金流的集合，这些现金流的要素包括：

- 现金流量；
- 现金流发生的时间。

我们不能简单的把现金总量累加，作为整个债券的价格。不同时间发生的现金流，价值并不一样。现在给你 1 万元和 10 年后给你 1 万元，你也感觉不一样，不是吗？将未来时间发生的现金流等价于当前的现金流价值的过程，称之为：**折现**。

如我们在收益率曲线 部分中介绍的一样，**收益率曲线**即为货币的时间价值的体现。所以在折现定价模型中，收益率曲线扮演着核心角色。确切的讲，例如 t 时刻发生的现金流 C 在当前的价值为：

$$\mathrm{PV} = C \times \mathrm{Discount}(t)$$

这里的 $\mathrm{Discount}(t)$ 是从收益率曲线上面取到的折现因子。对于有很多现金流组成的固定利息债券，它的折现过程可以由下图表示：

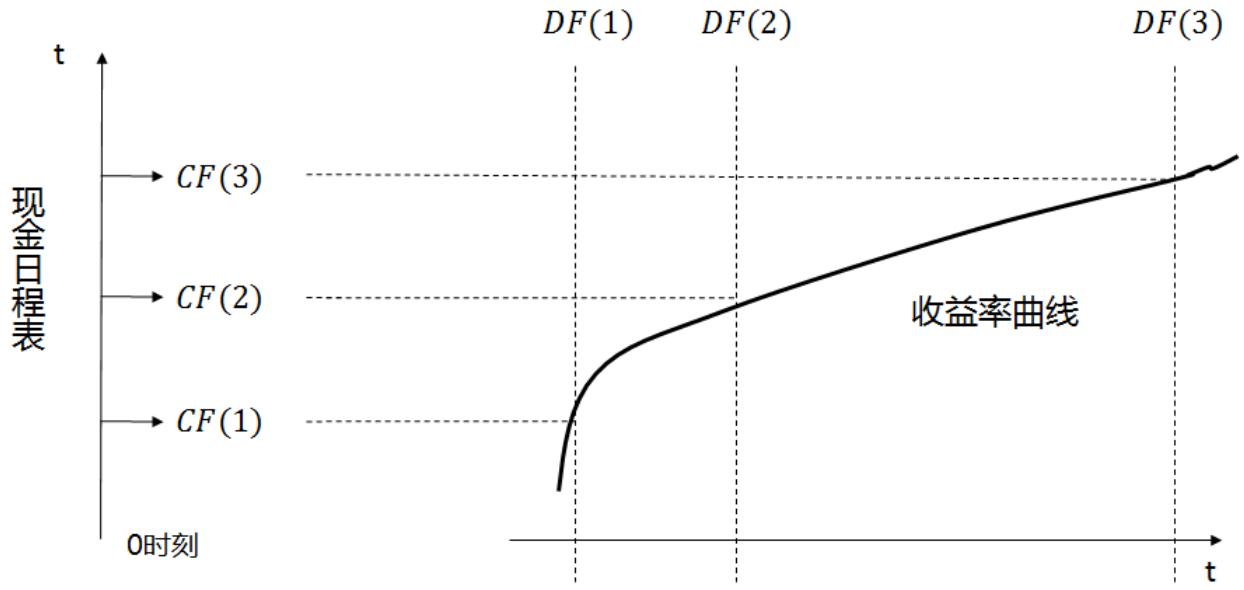


图 23.1: 债券折现定价示意图

用数学式子话来表示也很简单：

$$PV = \sum_{i=1}^{N} C(t_i) \times \text{Discount}(t_i)$$

这里 t_i 是现金流发生的时间点，$\text{Discount}(t_i)$ 是收益率曲线上对应时间点的折现因子。虽然数学原理很简单，但是真的要处理好折现模型，有许多细节要处理：

- 工作日惯例
- 利率的天数计数惯例
- 收益率曲线的插值方法

这些细节，我们已经在日期 等章节中介绍。如果需要更详细的介绍，请参考相关章节，或者 [37] 。下面的表格展示了在其他条件都相同的情况下（2012 年 8 月 164i 开始，2017 年 8 月 16 日到期，每半年付息，息票利率是 5% 的债券），不同的天数计数惯例对实际发生的付息现金流的影响，可以看到这种因素的影响不可忽略。关于这些市场惯例的非常详尽的整理，也可以关注: [22] 。

表 23.1: 不同计息方式的影响

付息日	Actual/Actual (ISMA)	Actual/360
2013/2/18	2.5000	2.5556
2013/8/16	2.5000	2.5139
2014/2/17	2.5000	2.5556
2014/8/18	2.5000	2.5139
2015/2/16	2.5000	2.5556
2015/8/17	2.5000	2.5139
2016/2/16	2.5000	2.5556
2016/8/16	2.5000	2.5278
2017/2/16	2.5000	2.5556
2017/8/16	2.5000	2.5139

我们这里会展示如何处理这些细节的正确步骤。让我们继续遵守产品-模型-算法周期 。

这里我们不会过多的介绍关于债券定义的细节，这部分的细节我们已经在债券 章节有所涉及。让我们首先关注本节的重点，贴现定价算法需要知道一条作为基准的收益率曲线，基本上的做法是:

```
In [1]: bondEngine = DiscountingBondEngine(yc)
```

这里 yc 是预先定义好的一个收益率曲线对象，然后如其他产品一样，将它赋予某个债券对象：

```
In [2]: bond.setPricingEngine(bondEngine)
   ...: print 'NPV: %.4f' % bond.NPV()
   ...:
NPV: 95.1027
```

23.1.2 接口

class PricingEngine.DiscountingBondEngine(PricingEngine.PricingEngine)

债券贴现定价引擎

___init___(self, yc)

构造债券贴现定价引擎

参数 yc (YieldCurve.YieldCurveProxy) – 收益率曲线

返回 债券贴现定价引擎对象

返回类型 PricingEngine.DiscountingBondEngine

class PricingEngine.BinomialConvertibleEngine(PricingEngine.PricingEngine)

可转换债券二叉树算法

___init___(self, BlackSholesProcess, type, steps)

构造可转换债券二叉树算法

参数

- BlackSholesProcess (StochasticProcess.GeneralizedBlackScholesProcess) – Black Scholes 过程
- type (str) – 二叉树类型
- steps (int) – 二叉树步数

返回 可转换债券二叉树算法对象

返回类型 PricingEngine.BinomialConvertibleEngine

class PricingEngine.TreeCallableFixedRateBondEngine(PricingEngine.PricingEngine)

基于三叉树的可回售债券定价算法

___init___(self, shortRateModel, timeSteps, yc = None):

构造基于三叉树的可回售债券定价算法

参数

- shortRateModel (ShortRateModels.ShortRateModel) – 短期利率模型
- timeSteps (int) – 时间步数
- yc (YieldCurve.YieldCurveProxy) – 收益率曲线

返回 基于三叉树的可回售债券定价算法对象

返回类型 PricingEngine.TreeCallableFixedRateBondEngine

23.2 利率衍生品定价算法

23.2.1 折现定价模型

23.2.2 接口

class PricingEngines.DiscountingSwapEngine(PricingEngine.PricingEngine)

利率互换折现定价模型

___init___(self, yc)

参数 yc (YieldCurve.YieldCurveProxy) – 收益率曲线

返回 利率互换折现定价模型

返回类型 PricingEngine.DiscountingSwapEngine

23.3 期权定价算法

在之前章节中，关于期权定价之快，我们已经介绍了产品-模型-算法周期 中的前两部分: 产品 与模型 。在这一章中我们将引入最后一部分: **算法**。我们仍将以例子的方式展开，并且遵循由简入繁的原则。

23.3.1 简单期权（Vanliia）算法

关于简单期权（Vanilla Option）的算法，包括解析法，PDE 方法，MC 方法，我们已经在如何为金融产品定价 部分做了介绍，读者可以参考该部分的介绍。

23.3.2 障碍期权（Barrier）算法

障碍期权是另外一大类常用期权，它的产生也是出于市场需求：

- 客户希望能降低期权成本;
- 客户愿意放弃一些极端收益;
- 客户对于未来的市场，特别是标的资产的波动性有一定的预期。

关于连续观察的单障碍期权，有显示的计算公式，关于这部分的推导，请见: [36]

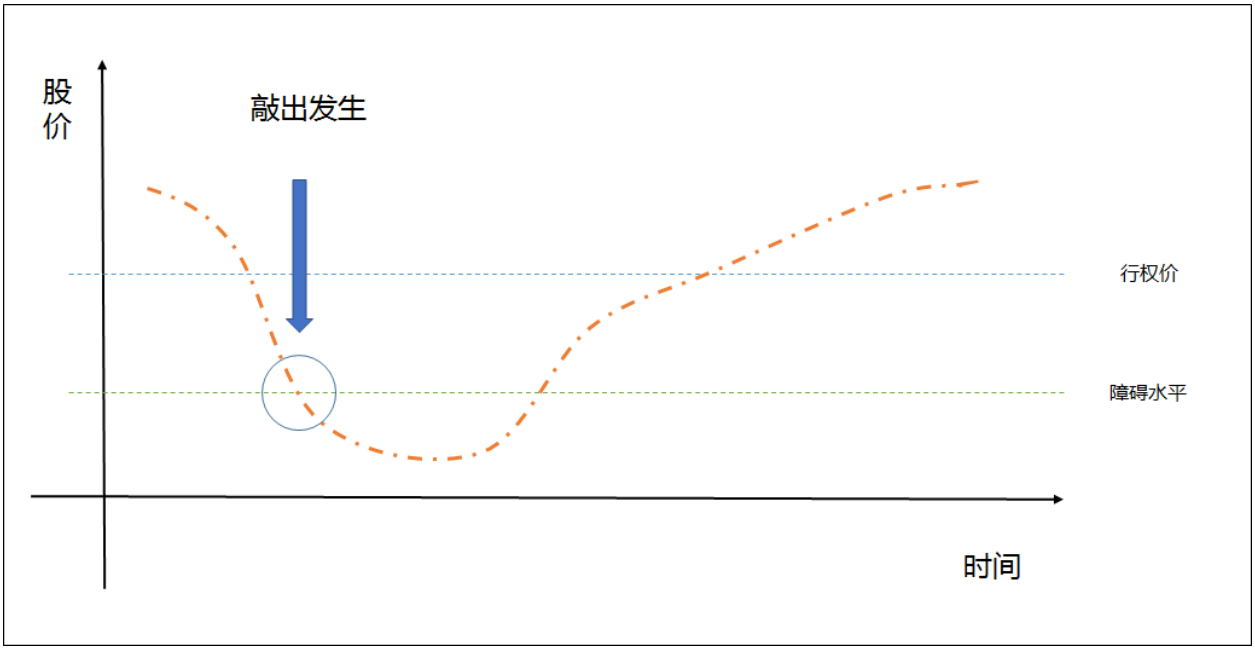


图 23.2: 向下敲出（Down and Out）期权

这里我们举一个看涨向下敲出期权作为例子（Down and out call option），取自 [21]：

$$C_{do}(K > H) = \mathrm{A} - \mathrm{C} - \mathrm{F},$$

$$\begin{aligned} \mathrm{A} &= Se^{(b-r)T}\mathrm{N}(x) - Ke^{-rT}\mathrm{N}(x - \sigma\sqrt{T}), \\ \mathrm{C} &= Se^{(b-r)T}(H/S)^{2(\mu+1)}\mathrm{N}(y) - Ke^{-rT}(H/S)^{2\mu}\mathrm{N}(y - \sigma\sqrt{T}), \\ \mathrm{F} &= K[(H/S)^{\mu+\lambda}\mathrm{N}(z) + (H/S)^{\mu-\lambda}\mathrm{N}(z - 2\lambda\sigma\sqrt{T})]. \end{aligned} \tag{23.1}$$

其中：

$$b = r - d,$$

$$x = \frac{\ln(S/K)}{\sigma\sqrt{T}} + (1+\mu)\sigma\sqrt{T},$$

$$y = \frac{\ln(H^2/SK)}{\sigma\sqrt{T}} + (1+\mu)\sigma\sqrt{T},$$

$$z = \frac{\ln(H/S)}{\sigma\sqrt{T}} + \lambda\sigma\sqrt{T}.$$

我们很容易的在 CAL 中实现这一定价，让我们仍然遵循产品-模型-算法周期。

首先我们定义看涨偿付 (call payoff)：

```
In [1]: strike = 40.0
   ...: type = Option.Call
   ...: payoff = PlainVanillaPayoff(type, strike)
   ...:
```

接着是欧式行权，我们还是从最简单的开始：

```
In [4]: exDate = Date(2014, 5, 17)
   ...: exercise = EuropeanExercise(exDate)
   ...:
```

```
In [6]: option = BarrierOption(Barrier.DownOut,
   ...:                        30.0,
   ...:                        0.0,
   ...:                        payoff,
   ...:                        exercise)
   ...:
```

下面仍然使用 Black-Scholes 模型：

```
In [7]: underlying = 36.00
   ...: riskFree = 0.06
   ...: dividend = 0.04
   ...: volatility = 0.40
   ...: model = BlackScholesMertonProcessConstant(underlying,
   ...:                                           riskFree,
   ...:                                           dividend,
   ...:                                           volatility,
   ...:                                           'Actual/Actual (ISMA)')
   ...:
```

最后使用障碍期权的解析算法：AnalyticBarrierEngine:

```
In [12]: engine = AnalyticBarrierEngine(model)
```

像往常的一样，让我们看一下期权的价格：

```
In [13]: option.setPricingEngine(engine)
   ....: evaDate = Date(2013, 5, 17)
   ....: SetEvaluationDate(evaDate)
   ....: print 'NPV: %.4f' % option.NPV()
   ....:
NPV: 3.4253
```

现在让我们来看一下，简答欧式期权价格与欧式障碍期权价格的关系：可以看到，随着障碍水平的下降，障碍期权的价格逐渐接近简单期权

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
import numpy as np
import pandas as pd
import seaborn as sns
sns.set(style="ticks")

strike = 40.0
type = Option.Call
payoff = PlainVanillaPayoff(type, strike)

exDate = Date(2014, 5, 17)
exercise = EuropeanExercise(exDate)

barriers = np.linspace(20.0, 30, 30)

underlying = 36.00
riskFree = 0.06
dividend = 0.04
volatility = 0.40
model = BlackScholesMertonProcessConstant(underlying,
                                          riskFree,
                                          dividend,
                                          volatility,
                                          'Actual/Actual (ISMA)')
engine = AnalyticBarrierEngine(model)
SetEvaluationDate(Date(2013, 5, 17))

prices = []

# 障碍期权价格
for barrier in barriers:
```

```
        option = BarrierOption(Barrier.DownOut, barrier, 0.0, payoff, exercise)
        option.setPricingEngine(engine)
        prices.append(option.NPV())

# 简单期权价格
option = VanillaOption(payoff, exercise)
engine = AnalyticEuropeanEngine(model)
option.setPricingEngine(engine)
vanilla = option.NPV()

data = pd.DataFrame({'Barrier':prices, 'Vanilla':vanilla}, index = barriers)

data.plot(style = ['-*','x-'])
pylab.xlabel('Barrier Level')
pylab.title('Barrier v.s. Vanilla', fontsize = 20)
pylab.show()
```

同简单期权一样，障碍期权也可以有其他的定价算法，例如它同样有基于 PDE 的算法：

```
In [17]: engine = FdBlackScholesBarrierEngine(model)
   ....: option.setPricingEngine(engine)
   ....: print 'NPV: %.4f' % option.NPV()
   ....:
NPV: 3.4248
```

基于 CAL 的灵活接口，我们也可以做关于 PDE 方法的收敛性测试：

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
import numpy as np
import pandas as pd
import seaborn as sns
sns.set(style="ticks")

strike = 40.0
type = Option.Call
payoff = PlainVanillaPayoff(type, strike)

exDate = Date(2014, 5, 17)
exercise = EuropeanExercise(exDate)

gridsteps = range(10,101,5)

underlying = 36.00
riskFree = 0.06
```

```
dividend = 0.04
volatility = 0.40
model = BlackScholesMertonProcessConstant(underlying,
                                          riskFree,
                                          dividend,
                                          volatility,
                                          'Actual/Actual (ISMA)')

SetEvaluationDate(Date(2013, 5, 17))
option = BarrierOption(Barrier.DownOut, 30, 0.0, payoff, exercise)

prices = []

# 障碍期权价格 ( PDE 方法 )
for step in gridsteps:
    engine = FdBlackScholesBarrierEngine(model, gridPoints = step)
    option.setPricingEngine(engine)
    prices.append(option.NPV())

# 障碍期权价格 ( 解析方法 )
engine = AnalyticBarrierEngine(model)
option.setPricingEngine(engine)
analytical = option.NPV()

data = pd.DataFrame({'Barrier (PDE)':prices, 'Barrier (Anal.)':analytical}, index = gridsteps)

data.plot(style = ['x-','-*'])
pylab.xlabel('Grids')
pylab.title('PDE v.s. Analytic', fontsize = 20)
pylab.show()
```

23.3.3 数字期权 (Digital Option) 算法

在本节中，我们将介绍另一类常用的期权：**数字期权 (Digital Option)** 。基本上，数字期权分成两种：支付标的 (AssetOrNothing) 以及支付现金 (CashOrNothing)，用式子可以表述它们的偿付形式如下：

$$\text{AssetOrNothing} = S_{S>K},$$

$$\text{CashOrNothing} = 1_{S>k}.$$

这里我们可以指出简单期权与数字期权的关系：

$$\text{Vanilla} = \text{Max}(S - K, 0)$$

$$= S_{S>k} - K_{S>K} = \text{AssetOrNothing} - K \times \text{CashOrNothing}.$$

所以事实上，简单期权可以看做是数字期权的线性组合。这一关系，在组合风险管理，特别是对冲操作中意义重大。除此之外，我们也很容易的可以得到这两种看涨数字期权的定价公式 [21] ：

$$c_{asset} = Se^{-dT}\mathrm{N}(d_1),$$
$$c_{cash} = Ke^{-rt}\mathrm{N}(d_2). \tag{23.2}$$

其中：

$$d_1 = \frac{\ln(S/K) + (r - d + \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}},$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

这部分的介绍我们将不引入任何新的 API，实际上这也是 CAL 模块化理念的一大优势，我们可以通过复用已经展示过的 API 函数，经过不同的组合，进而完成对数字期权的定价。

让我们从支付现金的数字期权开始：

```
In [20]: payoff = CashOrNothingPayoff(type, strike, 1.0)
   ....: option = EuropeanOption(payoff, exercise)
   ....: engine = AnalyticEuropeanEngine(model)
   ....: option.setPricingEngine(engine)
   ....: print 'NPV: %.4f' % option.NPV()
   ....: print 'Delta: %.4f' % option.delta()
   ....: print 'Gamma: %.4f' % option.gamma()
   ....:
NPV: 0.3199
Delta: 0.0240
Gamma: 0.0000
```

当然了，同样的 PDE 算法也可以直接应用于数字期权：

```
In [27]: engine = FDEuropeanEngine(model)
   ....: option.setPricingEngine(engine)
   ....: print 'NPV: %.4f' % option.NPV()
   ....:
NPV: 0.3230
```

最后我们来看一下 PDE 的收敛结果。我们可以看到，收敛趋势是呈锯齿型，这个是由于价格方向网格的问题。初始值的不连续性在后续的求解中一直在传播。关于这一问题的描述，请见: [38]

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
import numpy as np
import pandas as pd
import seaborn as sns
```

```
sns.set(style="ticks")

strike = 40.0
type = Option.Call
payoff = CashOrNothingPayoff(type, strike, 1.0)

exDate = Date(2014, 5, 17)
exercise = EuropeanExercise(exDate)

gridsteps = range(10,201,5)

underlying = 36.00
riskFree = 0.06
dividend = 0.04
volatility = 0.40
model = BlackScholesMertonProcessConstant(underlying,
                                          riskFree,
                                          dividend,
                                          volatility,
                                          'Actual/Actual (ISMA)')

SetEvaluationDate(Date(2013, 5, 17))
option = VanillaOption(payoff, exercise)

prices = []

# 数字期权价格 ( PDE 方法 )
for step in gridsteps:
    engine = FDEuropeanEngine(model, gridPoints = step)
    option.setPricingEngine(engine)
    prices.append(option.NPV())

# 数字期权价格 ( 解析方法 )
engine = AnalyticEuropeanEngine(model)
option.setPricingEngine(engine)
analytical = option.NPV()

data = pd.DataFrame({'Digital (PDE)':prices, 'Digital (Anal.)':analytical}, index = gridsteps)

data.plot(style = ['x-','-*'])
pylab.xlabel('Grids')
pylab.title('PDE v.s. Analytic', fontsize = 20)
pylab.ylim((0.25, 0.35))
pylab.show()
```

为了解决这一问题, [1] 中提到了阻尼的方法 (damping) 以及网格平移技术, 我们在FdBlackScholesVanillaEngine 中实现了这一算法:

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
import numpy as np
import pandas as pd
import seaborn as sns
sns.set(style="ticks")

strike = 40.0
type = Option.Call
payoff = CashOrNothingPayoff(type, strike, 1.0)

exDate = Date(2014, 5, 17)
exercise = EuropeanExercise(exDate)

gridsteps = range(10,201,5)

underlying = 36.00
riskFree = 0.06
dividend = 0.04
volatility = 0.40
model = BlackScholesMertonProcessConstant(underlying,
                                          riskFree,
                                          dividend,
                                          volatility,
                                          'Actual/Actual (ISMA)')

SetEvaluationDate(Date(2013, 5, 17))
option = VanillaOption(payoff, exercise)

prices = []

# 数字期权价格 ( PDE 方法 )
for step in gridsteps:
    engine = FdBlackScholesVanillaEngine(model, gridPoints = step)
    option.setPricingEngine(engine)
    prices.append(option.NPV())

# 数字期权价格 ( 解析方法 )
engine = AnalyticEuropeanEngine(model)
option.setPricingEngine(engine)
analytical = option.NPV()

data = pd.DataFrame({'Digital (PDE)':prices, 'Digital (Anal.)':analytical}, index = gridsteps)

data.plot(style = ['x-','-*'])
```

```
pylab.xlabel('Grids')
pylab.title('PDE v.s. Analytic (with damping)', fontsize = 20)
pylab.ylim((0.25, 0.35))
pylab.show()
```

显而易见的，阻尼法以及网格平移很成功的降低了误差，并且与 [1] 中的结果是一致的。

23.3.4 美式行权

之前的部分我们都只是涉及了欧式行权的情形。事实上，上面的各种计算算法，有些都可以直接运用于美式/百慕大式行权，这里我们不在赘述。我们只是代码举一个例子：

```
In [30]: europeanExercise = EuropeanExercise(exDate)
   ....: americanExercise = AmericanExercise(evaDate, exDate)
   ....: payoff = PlainVanillaPayoff(type, strike)
   ....: engine = FdBlackScholesVanillaEngine(model, 500, 500, 2)
   ....: americanOption = VanillaOption(payoff,
   ....:                                americanExercise)
   ....: europeanOption = VanillaOption(payoff,
   ....:                                europeanExercise)
   ....: europeanOption.setPricingEngine(engine)
   ....: americanOption.setPricingEngine(engine)
   ....: print 'NPV (european): %.4f' % europeanOption.NPV()
   ....: print 'NPV (american): %.4f' % americanOption.NPV()
   ....:
NPV (european): 4.3143
NPV (american): 4.3225
```

作为美式行权的一种特殊形式，百慕大行权也很常见，我们可以类似的方法去处理：

```
In [40]: exerciseDates = [exDate - Period('3M'), exDate - Period('6M'), exDate]
   ....: bermExercise = BermudanExercise(exerciseDates)
   ....: bermOption = VanillaOption(payoff,
   ....:                            bermExercise)
   ....: bermOption.setPricingEngine(engine)
   ....: print 'NPV (bermudan): %.4f' % bermOption.NPV()
   ....:
NPV (bermudan): 4.3197
```

三者 NPV 之间的关系，满足：

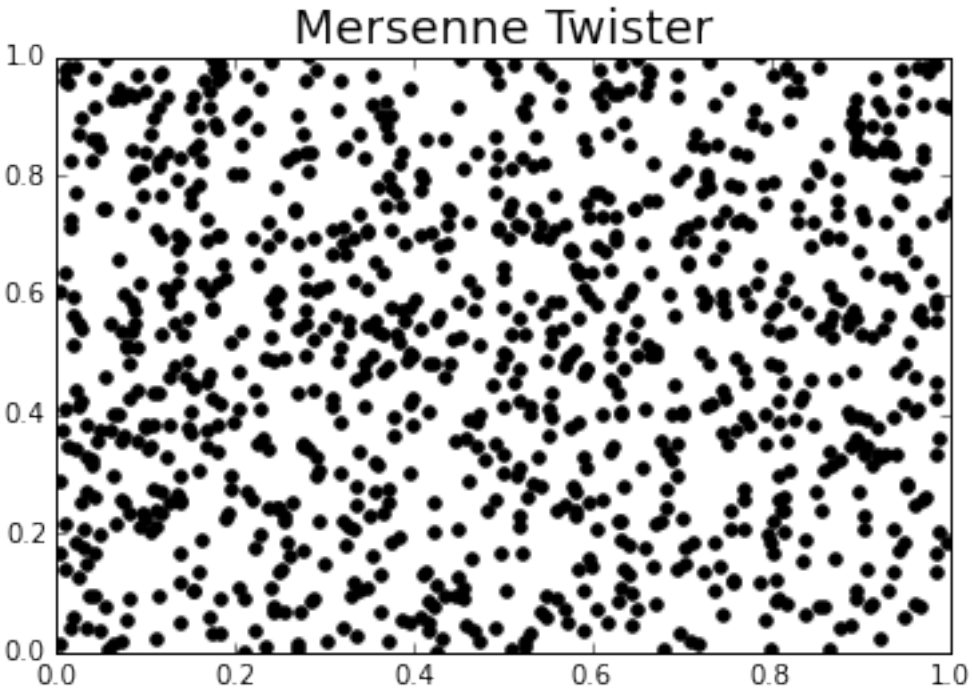
$$\text{NPV}_{European} \leq \text{NPV}_{Bermudan} \leq \text{NPV}_{American}$$

23.3.5 蒙特卡洛方法 (Monte Carlo)

之前的部分我们设计到了解析法以及 PDE 方法，这里我们单独辟出一个章节介绍另外一种常用算法: **蒙特卡洛** (Monte Carlo) 。

谈到蒙特卡洛方法，我们首先要提到的是其基础数学工具：随机数发生器。基本上在 CAL 中，我们常用的随机数发生器是两种：伪随机数与拟随机数：

- 伪随机数 (Pseudo): 典型代表为 Mersenne Twister 随机数；
- 拟随机数 (Quasi): 又称低差别序列，典型代表为 Sobol 随机数。



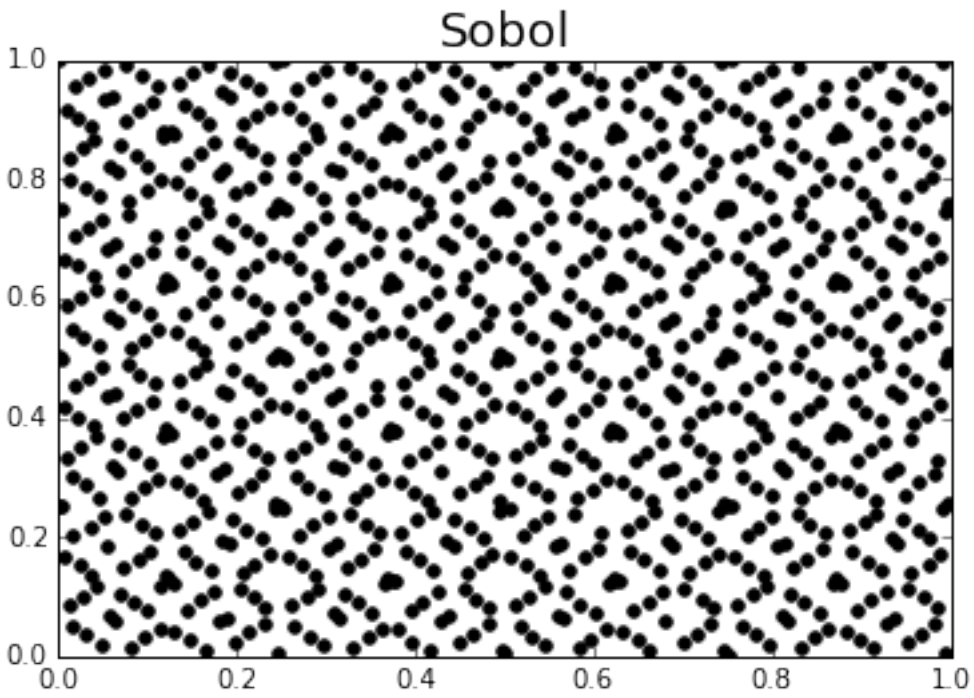


图 23.3: 均匀分布投影图 (第 1 维与第 2 维)

下面的脚本，会分别使用以上两种随机数发生器做蒙特卡洛模拟，并且计算欧式期权的价格，并与解析结果比较：拟随机数的代表 Sobol 随机数的收敛速度显著优于伪随机数的代表 Mersenne Twister 随机数。关于这方面的进一步的讨论以及其他随机数的介绍，请见 [18]

```
# -*- coding: utf-8 -*-

from CAL.PyCAL import *
from matplotlib import pylab
import numpy as np
import pandas as pd
import seaborn as sns
sns.set(style="ticks")

strike = 40.0
type = Option.Call
payoff = PlainVanillaPayoff(type, strike)

exDate = Date(2014, 5, 17)
exercise = EuropeanExercise(exDate)

logSamples = range(5, 22)
samples = [2**x - 1 for x in logSamples]

underlying = 36.00
```

```
riskFree = 0.06
dividend = 0.04
volatility = 0.40
model = BlackScholesMertonProcessConstant(underlying,
                                          riskFree,
                                          dividend,
                                          volatility,
                                          'Actual/Actual (ISMA)')

SetEvaluationDate(Date(2013, 5, 17))
option = VanillaOption(payoff, exercise)

prices1 = []
prices2 = []

# 简单期权价格 ( MC 方法 )
for sample in samples:
    engine1 = MCEuropeanEngine(model, 'pr', sample, 1, seed = 42)
    option.setPricingEngine(engine1)
    prices1.append(option.NPV())
    engine2 = MCEuropeanEngine(model, 'ld', sample, 1)
    option.setPricingEngine(engine2)
    prices2.append(option.NPV())

# 简单期权价格 ( 解析方法 )
engine = AnalyticEuropeanEngine(model)
option.setPricingEngine(engine)
analytical = option.NPV()

data = pd.DataFrame({'European (Mersenne Twister)':prices1, 'European (Sobol)': prices2, 'European (Anal.)':analytical}, index = l

data.plot(style = ['x-', 'k--', 'ro-'])
pylab.xlabel('No. of Samples ($2^x$)')
pylab.title('MC v.s. Analytic', fontsize = 20)
pylab.show()
```

23.3.6 接口

class PricingEngine.AnalyticEuropeanEngine(PricingEngine.PricingEngine)

欧式期权解析算法

___init___(self, BlackSholesProcess)

参数 BlackSholesProcess (StochasticProcess.GeneralizedBlackScholesProcess) – Black - Scholes 过程

返回 欧式期权算法对象

返回类型 PricingEngine.AnalyticEuropeanEngine

class PricingEngine.FDEuropeanEngine(PricingEngine.PricingEngine)
欧式期权 PDE 算法

___init___(self, BlackSholesProcess, timeSteps = 100, gridPoints = 100)

参数

- BlackSholesProcess (StochasticProcess.GeneralizedBlackScholesProcess) – Black - Scholes 过程
- timeSteps (int) – 时间方向步数
- gridPoints (int) – 价格方向步数

返回 欧式期权 PDE 算法对象

返回类型 PricingEngine.FDEuropeanEngine

class PricingEngine.MCEuropeanEngine(PricingEngine.PricingEngine)
欧式期权 Monte Carlo 算法

___init___(BlackSholesProcess, traits, requiredSamples, timeSteps = 100, brownianBridge = False, antitheticVariate = Fals, seed = 0)

参数

- BlackSholesProcess (StochasticProcess.GeneralizedBlackScholesProcess) – Black - Scholes 过程
- traits (str) – 随机数类型选择：'pr' 为伪随机数；'ld' 为低差别序列
- requiredSamples (int) – 采样数
- timeSteps (int) – 时间步长
- brownianBridge (bool) – 是否使用布朗桥方法
- antitheticVariate (bool) – 是否使用对偶变量法
- seed (int) – 随机数种子

返回 欧式期权 MC 算法对象

返回类型 PricingEngine.MCEuropeanEngine

class PricingEngine.AnalyticBarrierEngine(PricingEngine.PricingEngine)
障碍期权解析算法

___init___(BlackSholesProcess)

参数 BlackSholesProcess (StochasticProcess.GeneralizedBlackScholesProcess) – Black - Scholes 过程

返回 障碍期权解析算法对象

返回类型 PricingEngine.AnalyticBarrierEngine

class PricingEngine.FdBlackScholesBarrierEngine(PricingEngine.PricingEngine)
障碍期权 PDE 算法

___init___(BlackSholesProcess, timeSteps = 100, gridPoints = 100, dampingSteps = 2)

参数

- BlackSholesProcess (StochasticProcess.GeneralizedBlackScholesProcess) – Black - Scholes 过程
- timeSteps (int) – 时间方向步数
- gridPoints (int) – 价格方向步数
- dampingSteps (int) – 阻尼步骤

返回 障碍期权 PDE 算法

返回类型 PricingEngine.FdBlackScholesBarrierEngine

class PricingEngine.FdBlackScholesVanillaEngine(PricingEngine.PricingEngine)
Black Scholes PDE 算法

___init___(BlackSholesProcess, timeSteps = 100, gridPoints = 100, dampingSteps = 2)

参数

- BlackSholesProcess (StochasticProcess.GeneralizedBlackScholesProcess) – Black - Scholes 过程
- timeSteps (int) – 时间方向步数
- gridPoints (int) – 价格方向步数
- dampingSteps (int) – 阻尼步骤

返回 Black Scholes PDE 算法

返回类型 PricingEngine.FdBlackScholesVanillaEngine

CHAPTER 24

数学

24.1 随机过程

24.1.1 接口

class StochasticProcess.GeneralizedBlackScholesProcess
广义 Black Scholes 过程

class StochasticProcess.BlackScholesMertonProcessConstant(StochasticProcess.GeneralizedBlackScholesProcess)
常参数 Black Scholes Merton 过程

___init___(self, spot, riskFree, dividend, vol, dc = 'Actual/360')
构造常参数 Black Scholes Merton 过程

参数

- spot (float) – 标的初值
- riskFree (float) – 无风险利率
- dividend (float) – 红利
- vol (float) – 波动率
- dc (DayCounters.DayCounter) – 天数计数惯例

返回 常参数 Black Scholes Merton 过程

返回类型 StochasticProcess.BlackScholesMertonProcessConstant

24.2 函数求解

24.2.1 接口

class Solvers.SolverBase
所有函数求解器的基类

solve(self, function, guess, accuracy = 1e-6, step = 1e-2)
从初始值出发求解函数的零点

参数

- function (Function) – 目标函数
- guess (float) – 初始值
- accuracy (float) – 要求精度
- step (float) – 初始步长

返回 函数零点的自变量值

返回类型 float

solveInInterval(self, function, guess, xmin, xmax, accuracy = 1e-6)
从初始值出发在区间内求解函数的零点

参数

- function (Function) – 目标函数
- guess (float) – 初始值
- xmin (float) – 区间下界
- xmax (float) – 区间上界
- accuracy (float) – 要求精度

返回 函数零点的自变量值

返回类型 float

class Solvers.Brent(Solvers.SolverBase)
Brent 型 1 维函数求解器

___init___(self)

class Solvers.NewtonSafe(Solvers.SolverBase)
Newton Safe 型 1 为函数求解器

___init___(self)

24.3 非线性函数优化

24.3.1 接口

class Optimizers.Optimizer
所有非线性函数优化器的基类

solve(self, function, guess, maxIteration = 50000, accuracy = 1e-8)
求解函数最小值

参数

- function (Function) – 目标函数
- guess (list) – 初始值
- maxIteration (int) – 最大迭代
- accuracy (float) – 精度

返回 求解结果

返回类型 list

class Optimizers.Simplex(Optimizers.Optimizer)
Nelder–Mead 单纯型方法

___init___(self, lam = 1.0)
构造 Nelder–Mead 单纯型方法

参数 lam (float) – 单纯型初始边长

class Optimizers.BFGS(Optimizers.Optimizer)
BFGS 优化法

___init___(self)
构造 BFGS 优化法

class Optimizers.LevenbergMarquardt(Optimizers.Optimizer)
Levenberg Marquardt 非线性拟合方法

___init___(self, epsfcn = 1e-8, xtol = 1e-8, gtol = 1e-8)
构造 Levenberg Marquardt 非线性拟合方法

参数

- epsfcn (float) –
- xtol (float) –
- gtol (float) –

class Optimizers.SimulatedAnnealingMT(Optimizers.Optimizer)
模拟退火 (SimulatedAnnealing) 优化方法基于 Mersenne Twister 随机数

___init___(self, lam, t0, epsilon, m)
构造模拟退火 (SimulatedAnnealing) 优化方法

参数

- lam (float) – Simple 方法使用的边长参数
- t0 (float) – 初始温度
- epsilon (float) – 降温量
- m (float) – 降温速度

CHAPTER 25

枚举类型

25.1 本金摊还方法

25.1.1 接口

定义了多种本金摊还的方法

AmortizingType.EqualPayment
: 等额本息

AmortizingType.EqualPrinciple
: 等额本金

25.2 亚式期权平均方法

25.2.1 接口

定义了亚式期权计算偿付的时候的方法

AverageType.Arithmetic
: 算术平均

AverageType.Geometric
: 几何平均

25.3 二叉树类型

25.3.1 接口

定义了二叉树模型的树的种类

CRR.JR

CRR.EQP

CRR.TIAN

CRR.LR

CRR.JOSHI

25.4 工作日惯例

25.4.1 接口

定义了例如结算日，付息日遭遇节假日时候处理方式

BizDayConvention.Following
: 调整至下一工作日

BizDayConvention.ModifiedFollowing
: 调整至下一工作日，但是如果发生跨月，则调整至前一工作日。

BizDayConvention.Preceding
: 调整至前一工作日

BizDayConvention.ModifiedPreceding
: 调整至前一工作日，但是如果发生跨月，则调整至下一工作日

BizDayConvention.HalfMonthModifiedFollowing

BizDayConvention.Nearest
: 向前或者向后调整至最近的一个工作日，方向取决与最近工作日的方向。

BizDayConvention.Unadjusted
: 不调整

25.5 日历合并规则

25.5.1 接口

描述了两个工作日历合并的方法。

CalendarJoinRule.JoinBizDays
: 合并两个日历的工作日，即新日历的工作日是第一个工作日历的工作日，或者是第二个的

CalendarJoinRule.JoinHolidays
: 合并两个日历的节假日，即新日历的工作日是第一个工作日历的节假日，或者是第二个的

25.6 复利方法

描述我们生活中常用的复利方法

25.6.1 接口

Compounding.Simple
: 简单利率, $1 + rt$

Compounding.Compounded
: 复利，$(1 + r)^t$

Compounding.SimpleThenCompounded
: 先简单后复利，$(1 + rt) \quad t < 1$; $(1 + r)^t \quad t > 1$

Compounding.Continuous
: 连续利率，$\exp^{rt}$

25.7 日期生成方法

如何产生日期序列

25.7.1 接口

DateGeneration.Backward
: 从到期日开始向前生成日期直到开始日

DateGeneration.Forward
: 从开始日开始向后生成日期直到到期日

25.8 久期类型

久期类型

25.8.1 接口

DurationType.Maculay
: Maculay 久期

DurationType.Simple
: 简单久期

DurationType.Modified
: 修正久期

25.9 日期频率

25.9.1 接口

Frequency.NoFrequency
: 无频率，用于做占位符

Frequency.Once
: 整个周期内 1 次

Frequency.Annual
: 按年

Frequency.EveryFourthMonth
: 每四个月

Frequency.Semiannual
: 每半年

Frequency.Quarterly
: 每季度

Frequency.Bimonthly
: 每两个月

Frequency.Monthly
: 每个月

Frequency.EveryFourthWeek
: 每四周

Frequency.Biweekly
: 每两周

Frequency.Weekly
: 每周

Frequency.Daily
: 每日

25.10 Sobol 随机数产生方法

25.10.1 接口

SobolType.Jaeckel
: 使用 Jaeckel 方法，请参考 [30]

SobolType.JoeKuoD6
: 使用 Joe 和 Kuo 实现的算法，具体请参考 [28] 以及 [29]

25.11 利率互换类型

25.11.1 接口

SwapLegType.Receiver
: 收固定端利息，付浮动端利息

SwapLegType.Payer
: 付固定端利息，收浮动端利息

Part IV

附录

[1] Leif Andersen and Vladimir V Piterbarg. Interest Rate Modeling, Volume I: Foundations and Vanilla Models. Atlantic Financial Press London, 2010.

[2] Louis Bachelier. Louis Bachelier's theory of speculation: the origins of modern finance. Princeton University Press, 2011.

[3] Luigi Ballabio. Implementing quantlib. 2005.

[4] GIOVANNI BARONE-ADESI and Robert E Whaley. Efficient analytic approximation of american option values. The Journal of Finance, 42(2):301–320, 1987.

[5] Fischer Black and Piotr Karasinski. Bond and option pricing when short rates are lognormal. Financial Analysts Journal, pages 52–59, 1991.

[6] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. The journal of political economy, pages 637–654, 1973.

[7] George EP Box and Norman R Draper. Empirical model-building and response surfaces. John Wiley & Sons, 1987.

[8] Damiano Brigo and Fabio Mercurio. Interest rate models-theory and practice: with smile, inflation and credit. Springer, 2007.

[9] Jaehyuk Choi, Kwangmoon Kim, and Minsuk Kwak. Numerical approximation of the implied volatility under arithmetic brownian motion. Applied Mathematical Finance, 16(3):261–268, 2009.

[10] Iain J Clark. Foreign Exchange Option Pricing: A Practitioners Guide. John Wiley & Sons, 2011.

[11] John Cox. Notes on option pricing i: Constant elasticity of variance diffusions. Technical Report, working paper, Stanford University, 1975.

[12] John C Cox, Jonathan E Ingersoll Jr, and Stephen A Ross. A theory of the term structure of interest rates. Econometrica: Journal of the Econometric Society, pages 385–407, 1985.

[13] Emanuel Derman. On fischer black: Intuition is a merging of the understander with the understood. 2005.

[14] Paul Doust. No-arbitrage sabr. Journal of Computational Finance, 2012.

[15] Daniel J Duffy. Finite Difference methods in financial engineering: a Partial Differential Equation approach. John Wiley & Sons, 2006.

[16] Bruno Dupire. Pricing and hedging with smiles. Mathematics of derivative securities. Dempster and Pliska eds., Cambridge Uni. Press, 1997.

[17] Jim Gatheral and Antoine Jacquier. Arbitrage-free svi volatility surfaces. Quantitative Finance, 14(1):59–71, 2014.

[18] Paul Glasserman. Monte Carlo methods in financial engineering. volume 53. Springer, 2004.

[19] Patrick S Hagan, Deep Kumar, Andrew S Lesniewski, and Diana E Woodward. Managing smile risk. 70+ DVD's FOR SALE & EXCHANGE, pages 249, 2002.

[20] Patrick S Hagan and Graeme West. Interpolation methods for curve construction. Applied Mathematical Finance, 13(2):89–129, 2006.

[21] Espen Gaarder Haug. Option pricing formulas. McGrew-Hill, 2007.

[22] Marc Henrard. Interest rate instruments and market conventions guide. OpenGamma Quantitative Research,, 2012.

[23] Steven L Heston. A closed-form solution for options with stochastic volatility with applications to bond and currency options. Review of financial studies, 6(2):327–343, 1993.

[24] YL Hsu, TI Lin, and CF Lee. Constant elasticity of variance option pricing model: Integration and detailed derivation. In Handbook of Quantitative Finance and Risk Management, pages 471–480. Springer, 2010.

[25] John Hull and Alan White. Pricing interest-rate-derivative securities. Review of financial studies, 3(4):573–592, 1990.

[26] John C Hull and Alan D White. Numerical procedures for implementing term structure models i: Single-factor models. The Journal of Derivatives, 2(1):7–16, 1994.

[27] Tom Hyer. Derivatives Algorithms. World Scientific, 2010.

[28] Stephen Joe and Frances Y Kuo. Remark on algorithm 659: Implementing sobol's quasirandom sequence generator. ACM Transactions on Mathematical Software (TOMS), 29(1):49–57, 2003.

[29] Stephen Joe and Frances Y Kuo. Constructing sobol sequences with better two-dimensional projections. SIAM Journal on Scientific Computing, 30(5):2635–2654, 2008.

[30] Peter Jäckel and Russ Bubley. Monte Carlo methods in finance. J. Wiley, 2002.

[31] Black - Scholes - Merton model. http://en.wikipedia.org/wiki/black–scholes_model. 2014.

[32] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical Recipes 3rd Edition: The Art of Scientific Computing. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[33] Riccardo Rebonato. Volatility and Correlation in the Pricing of Equity, FX, and Interest-rate Options. John Wiley, 1999.

[34] Mark Rubinstein. Displaced diffusion option pricing. The Journal of Finance, 38(1):213–217, 1983.

[35] Amir Sadr. Interest rate swaps and their derivatives: A practitioner's guide. volume 510. John Wiley & Sons, 2009.

[36] Steven E Shreve. Stochastic calculus for finance II: Continuous-time models. volume 11. Springer, 2004.

[37] Donald J Smith. Bond math: the theory behind the formulas. John Wiley & Sons, 2011.

[38] Domingo Tavella and Curt Randall. Pricing financial instruments: The finite difference method. John Wiley & Sons Chichester, 2000.

[39] Oldrich Vasicek. An equilibrium characterization of the term structure. Journal of financial economics, 5(2):177–188, 1977.

[40] 中国人民银行. 中国人民银行关于全国银行间债券市场债券到期收益率计算有关事项的通知. 2004.

[41] 中国人民银行. （附件）中国人民银行关于完善全国银行间债券到期收益率计算标准有关事项的通知. 2006.

[42] 中国债券信息网. http://www.chinabond.com.cn/. 2014.

[43] 沈炳熙 and 曹媛媛. 中国债券市场: 30 年改革与发展. 北京大学出版社, 2010.