构建 QuantLib

深度探索量化金融 C++ 源代码

Luigi Ballabio 著

徐瑞龙 译

构建 QuantLib

深度探索量化金融 C++ 源代码

Luigi Ballabio and Ruilong Xu

這本書的網址是 http://leanpub.com/implementingquantlib-cn

此版本發布於 2019-09-26

Luigi Ballabio 的其他著作

QuantLib Python Cookbook

Implementing QuantLib

此译作献给所有爱我的人

Contents

1. 声明

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

2. 译后记

2.1 初心

当我还在读研究生的时候，为了实践某些计算金融学的课题自学了 C++ 语言（当时 C++ 大概是“老派 quant”最为推崇的语言，现在“新派 quant”可能都转向 python 了），进而接触到当时就已经久负盛名的 QuantLib。

刚开始使用 QuantLib 时就遇到了 Luigi 在书中指出的问题：**缺乏相应文档的弊端开始显示**，我对这个库的整体架构一无所知，函数之间的调用关系似懂非懂，类之间的继承关系不清不楚，更棘手的是当时没有遇到任何中英文文献帮助我解决上述问题。

后来事情出现了好转，我读到了名为《QuantLib 实现》的翻译文章（对不起，我已经记不起译者的名字了），并了解到 Luigi 正在撰写相关文档描述 QuantLib 的整体设计和实现细节，尽管当时文档的内容还非常有限。

直到 2013 年，我得知 Luigi 开始在其网站 以博客的形式连载这些文档。当时我曾计划，等 Luigi 差不多写完，我就开始着手翻译工作，因为我主观上觉察到先前的那位译者似乎中断了翻译计划。

但是我高估了意大利人写书的效率。一两年之后我发现 Luigi 的写作转移到了 leanpub 上，接着就是漫长的等待。

2.2 收获

2017 年 8 月份，《Implementing QuantLib》的第一个完成版本发布，我的工作在这之后的 11 月份开始。历时一年多的翻译工作，多少有些收获。

最直接的，我得到了一篇几百页的中译本手稿。不过，我更想说的是那些比较抽象的收获。

- 成为领域专家

要写出公认专业的算法和软件，首先要成为特定领域的专家，而不是相反。所以，首要的是去学习，从现成的工具着手，从最简单的问题着手，先成为一个专家。

- 设计模式与知识结构

如果要成功地应用设计模式描述领域问题，那么设计模式所表现出的结构应该和这一领域的知识结构相契合。

举个具体的例子，QuantLib 的核心目的是为一系列金融工具的定价提供一个统一的框架。那么，什么是“定价”？依我的理解，所谓定价就是**计算未来预期现金流的贴现值**，关键词有这么几个：计算、预期、现金流、贴现。

这几个关键词正好描述了定价问题的知识结构，计算的方式可以分为解析解法、数值解法和随机模拟解法，等等；预期可以指风险中性测度下的期望，或是当前期限结构隐含的均衡结果，等等；现金流可以是固定的（固息债）也可以是随机的（浮息债和期权）；贴现则整个就是期限结构选择的问题。每一个关键词都可以成为一个庞杂的研究课题。QuantLib 目前的设计正是将上述几块的知识抽象成模块，再细分成类，将方法封装进类，最后用“配置类实例”的方式将这几若干实例组合成为一个用于定价计算综合实例（从宏观上看就是策略模式）。

- 知识结构与设计模式

也可以反过来思考，要想理解一个领域的知识结构（前提是已经入门），发现新的细节，就去看这个领域相关的软件库是怎么架构的（最好是 OOP 的）。我关于定价本质的理解正是在研究 QuantLib 架构之后获得启发的，同时也见识到了许多未尝接触过的新课题，比如多曲线贴现。

2.3 彩蛋

Luigi 非常热衷于在书中以双关语的形式引经据典，从古典到流行，从神话到科幻，几乎无所不包。这些埋藏的文化彩蛋通常是一些欧美文化圈的人才能熟知的典故，对于汉语文化圈的人来讲可能非常陌生，甚至不知所云。比如这个，第 7 章中出现的“curiouser and curiouser”，这句话是一个彩蛋，它最著名的出处是《爱丽丝梦游仙境》。不过这里用作双关语，用来形容“奇异（Curiously）递归模板模式”中让人费解的继承关系。

在翻译的过程中，我竭尽所能发现尽可能多的彩蛋，考证其出处、注解其语义。寻找彩蛋的过程让耗时枯燥的翻译工作妙趣横生，这完全超出了我的预料。

目前已经找出了大约五十个彩蛋，均以译注的形式呈现出来。

2.4 鸣谢

最后，特别感谢林晓博士在百忙之中抽出时间来为本书作序。

3. 译序

我很荣幸能为徐瑞龙先生翻译的《构建 QuantLib》一书写个序言。借这个机会，我想就自己二十多年的从业经验，向读者介绍一下 Quant 这个行业的历史和现状，以及未来发展所面临的机会和挑战，并从这些方面来说明阅读本书的重要性。

Quant 是 Quantitative Analysis 的简称，许多人都翻译为量化分析。这个词的由来与衍生产品有关。在 1990 年左右，在欧美的银行间市场普遍兴起了做衍生产品交易的热潮。这种衍生产品包括利率的期权，汇率的期权，股票的期权，或商品的期权等等。当银行准备买入或者卖出一个期权的时候，遇到的第一个问题是怎样确定价格，遇到的第二个问题是怎样管理交易的风险。说到风险管理，我们需要了解的一点是，衍生产品的生命期限往往很长，就像我们持有股票一样，今天赢了钱，明天市场变了，说不定又输回去了，所以需要进行风险对冲。这些都需要进行准确的数量分析预测，算多了不行，算少了也不行。量化分析这种工作就是这样产生的。

在业界中还有一个叫“量化交易”的名词需要区分一下。量化交易一般指制定某些交易策略，尤其是非人工的由机器自动执行的交易策略，例如对股票，当价格掉到什么位置就买入，升到什么价格就卖出，等等。所以量化交易与我们这里所说的量化分析，本质上不是一回事。

从事量化分析工作的人，称为量化分析师（Quantitative Analyst），也简称为 Quant。当银行前台的营销人员和交易员接到客户的需求，要做一种新的衍生产品交易的时候，量化分析师就行动起来，对产品进行设计，提出一个定价模型，并用 C/C++ 编写出计算程序。交易员使用程序计算出产品的价格，就可以与客户成交了。这个过程，快的时候，可能就是一天时间就能把交易做成。在国有银行工作过的朋友可能会有疑问，银行是否要做出一个完整的计算机网络系统才能做交易。其实在欧美，银行非常多，互相之间竞争非常激烈。所谓的客户，一般都是对冲基金公司，保险公司等投资人，甚至不少是国家队的（可以想想黑石公司），他们对市场走势有很激进的分析，会通过设杠杆加结构，提出各种高收益高风险的衍生产品交易。这对银行无疑就是一个很严峻的挑战。举例说，如果对冲基金公司觉得某个交易能使他得到 20% 以上的回报，他是不在乎支付银行 2% 的费用的，甚至可以把本金做到 10 个亿美元，而这对银行就是 2000 万美元的收入了。想想看，一个电话的功夫就把 2000 万美元拿到手，是一件多么刺激的事情。但是另一方面，既然是设了杠杠加了结构，交易的价格就不是我们普通人能一眼看出来的，所以，为了抢到这些交易，各家银行都要聘用很高水平的量化分析师来开展工作，同时也尽量简化交易流程，以便把更多的交易抢到手。

对一个新的衍生产品，为了抢占时机，国际大银行一般都不会等到做成一个完整的计算机网络系统才开始做交易的。但是，如果交易做多了，成规模了，银行就会把技术部门调过来做一个完整的网络系统，有输入交易的页面，有存放交易的数据库（之前往往就是用一

个 Excel 文件来输入和存放交易），有交易的授权授信控制和限额管理（之前的单笔交易就是用手工进行这些工作），还有交易的组合管理的各种报表（之前可能就没有这些报表）等等。系统的定价引擎仍然是量化分析师在前面设计开发好的。近几年，定价引擎除了为前台的计价和组合管理提供服务外，还扩展到了为中台计算 VaR、CVA、PFE、保证金、为后台计算结算清算的现金流等等。这样，量化分析师的工作量也成倍增加。在花旗，摩根大通等大银行，全球量化分析师的总人数有几百甚至上千人，支持着利率、汇率、商品、股票、信用等等各个领域的衍生产品交易业务，当然，相应的技术部门的人员（称为 Tech）还会多好几倍。

除了前台，银行的中台（风险管理部）还有一个量化分析师的风险团队，承担模型考核（Model Validation）的任务，他们不参与前台的产品和模型开发，而是独立地对前台开发的衍生产品，定价模型和计算方法进行考核研究，确保产品是合规合法合理的，模型是无套利的，方法是有效无误，避免给银行带来风险。做模型考核的量化分析师也需要独立地编写量化定价模型的计算程序，以便把前台的模型复制出来。美国的银监会（OCC）和美联储，每年都到各个银行进行两次模型检查，从阅读模型考核报告开始，了解银行做些什么新产品，使用什么模型定价，是否合规合法合理。我在纽约工作的那段时间，一个银行的模型考核团队仅 10 多个人，管理着全银行 1000 多个模型文档。但是在 2008 年的金融危机后，各大银行都认真地检讨了因模型的使用不当而导致的巨大损失，从而将模型考核团队增加到了 100-200 人。

有人会把银行的量化分析师与保险公司的精算师进行比较。他们都是设计产品建立开发定价模型的，都是公司业务中挑大梁的核心人员。有一点不一样，因为保险业已经发展了上百年，精算师的资格体系齐全，分很多等级，需要通过严格的考试一级一级地考上去。银行间的衍生产品交易市场比较年轻，发展很快，所以只要您能设计产品，开发模型，银行就让你上岗，不需要进行资格考试。实际上，在 1995 年前后，纽约地区的许多报纸上都出现大幅广告，高薪招聘数学与物理专业的博士为量化分析师，他们可以没有任何的银行从业经验，入职后再学习怎样做模型。

量化分析师的岗位在银行中如此重要，薪酬也高，必然引来许多人的争夺。然而，这份工作确实不是一般人能够胜任的。想成为一位合格的量化分析师，总结起来，应该具备三个方面的技能：一是数学好，二是金融好，三是编程好。所谓数学好，并不是泛泛的，主要是在应用数学方面，包括概率与随机过程，线性代数，偏微分方程，和各种的数值计算方法，对这些学科非常精通。因为我们所做的模型，在数学上都是把某个市场工具 (例如股票) 的价格当作一个随机变量处理，我们所关注并要计算的某个衍生产品的价格，则是这个随机变量的函数，服从一个随机微分方程。所以我们计价，就是求解这个微分方程的解答。所谓金融好，就是对衍生产品交易的各种产品条款很熟悉。因为我们最终是给这些交易计算价格的，如果看不懂它们的条款属性，怎么能给它们建立模型进行计算呢？所谓编程好，就是会编写代码来实现对金融产品的模型定价。过去银行对传统业务进行系统开发，往往是由业务人员写一个需求书，把要计算的内容和方法都用伪代码的形式写下来，然后让技术人员写成源代码。但是衍生产品的模型定价非常复杂，以致无法用伪代码写出来，即使能写出来，换一个人来重新写源代码，时间成本与出错率都会很高。这样，量化分析师就不得不承担起编程的工作了。由于历史的原因，和计算速度的要求，银行中的模型程序库一般都是用 C/C++ 写的。所以会用 C/C++ 语言编程是必须的。

然而没有人是天生下来就具备以上三种技能，就会写模型代码的，都要经过学习。这样，能不能找到一个学习环境就非常重要了。最好的学习环境肯定是走进银行，直接到一个衍生产品的研发部门或者相邻的部门，通过阅读银行已有的模型定价系统的源代码进行学习。90 年代在纽约或伦敦工作的我们那一代留学生是很幸运的，只要数学好会编程，就有机会进入某个银行的量化分析团队。但是后来银行招聘的人多了，就没有这么好的机会了。怎么办呢？感谢我们的前辈，他们开发了一个开源的量化模型程序库 QuantLib，把银行中对各类衍生产品的计价模型的源代码收集起来了，给我们学习提供了很好的教材。

其实 QuantLib 的意义远远不止给年轻人提供了一个入门学习的环境。对于我们在行业内工作了多年的量化分析师，也是一个知识交流与更新的地方。虽然各家银行在市场上互相进行衍生产品交易，但是各家银行都在独立地开发自己的定价模型，不会互相分享。一家银行要开发一个新的模型产品，如果自己没有头绪，只有两种办法：一是到市场上挖人，这当然成本很高；二是在 QuantLib 或其他期刊上找资料。所以 QuantLib 也是一个很重要的资源。现在国内有一种很流行的理财产品：挂钩 A 股指数的增长率 X，如果未来 3 个月，X 低于 3%，理财的收益率锁定为 3%；如果 X 在 3% 至 7% 之间，则理财的收益率与 X 相同；如果 X 高于 7%，则理财的收益率为 7%。我本人 2003 年在花旗工作时也做过同样的产品，使用随机波动率模型（如 Heston 模型），和 Euler 格式或 Milstein 格式进行数值计算。虽然那时是完成了工作，但总有一种感觉，在做模型拟合时，那几个 Heston 模型参数总是没力气改变交易的价格，拟合的结果很随意，精度比较差。几年后，我看到了 Leif Andersen 的文章和他发布在 QuantLib 上的源代码，学习了他研究出来的方法，才使得我的知识上了一个台阶。Andersen 原来是纽约大学的教授，现在是美洲银行的 Global Head of Quant。在 QuantLib 上有很多名人的杰作，他们做出来的模型方法，一般都比我们自己蒙着头想出来的要好，需要我们好好学习。

目前我们看到的 QuantLib，已经把各种资产类别的衍生产品的定价模型整合在一起。如果回顾一下模型发展的历史，这需要很长期的经验积累才能做到的。在早年的以做衍生产品闻名的 Salomon Brothers 公司中，其定价模型库 Riskman 实质上是一堆零散独立的小程序，每个程序仅为一个产品计算价格，每个程序都要将用到的支付频率、台历等写死在程序之中。经过多年的发展，现在银行的定价模型库就像 QuantLib 一样，可以做的很紧凑，各种交易、各种模型和各种计算方法都可以写进去，互相交叉组合来使用。增加一个新产品，就像在一个大房间内增加一张桌子，一张床，开发效率很高。一个好的设计，不但提高了系统开发的速度，对交易业务也有了很大的提升。过去，要使用 Salomon 时期的那种小程序对 5000 个交易计算 VaR，是根本做不到的。如果模型库设计不好，需要将中间计算结果反复导入导出，即使使用 300 个 CPU 并行计算也要耗费 3 个小时。但是，把模型库设计好，仅使用一台 4 个 CPU 的电脑就可以在 10 分钟内计算出来。如果从头开始一个新的模型系统的设计开发，QuantLib 的架构是一个很好的参考。

以上关于 QuantLib 的三点，都是我们阅读本书的意义所在。因为 QuantLib 已经把各种模型的整合在一起了，其基础架构的内容就会很多，加上 QuantLib 使用高级语言 C/C++ 写成，使用了大量的 Template，阅读起来不会很容易，所以，本书将是一把好钥匙，帮您打开模型宝库的大门。

最后，我想介绍一下国内衍生产品模型定价的应用和发展情况。与欧美国家相比，我国的衍生产品市场还是处于初级发展阶段。在银行间市场上，目前只有利率互换和普通汇率期

权两个产品的交易比较活跃，但这两个产品都是简单产品，使用很简单的模型定价。多年来，许多商业银行一直在筹备开展利率期权和奇异汇率期权的业务，由于缺乏人才进行相关的模型与定价系统研发，大家还在观望摸索着。券商和基金公司开展衍生产品业务比较活跃一些，可以做多种挂钩股票和债券的期权。但是，由于券商与基金公司的客户群远不如银行那么大，交易量有限。借助理财业务的平台，许多银行与金融机构做了大量的结构性存款业务。这种产品，原则上需要定价模型来计算收益与风险的。在缺乏模型与系统的情况下，往往就把交易的价格设在一个非常安全的位置，结果把有风险的结构性存款设计成了高息揽存、刚性兑付的固定利率存款，偏离了监管机构关于理财产品的管理规定。2017年，中国人民银行、银监会、证监会、保监会、外汇局联合发布了一个《关于规范金融机构资产管理业务的指导意见》，责令商业银行与金融机构在几年内必须彻底停止各种假结构的交易。我觉得，这正是推动业界开展模型定价业务的一个强劲动力，将会有越来越多的银行招聘量化分析师来开发自己的衍生产品的定价模型库，促进业务的发展。

面临这样一个可遇不可求的市场机遇，我热切希望，徐瑞龙先生翻译的《构建 QuantLib》这本书能帮助更多的人将自己打造成为精通数学模型、精通金融产品、精通算法编程的Quant，投身到这个行业来，为祖国银行业的发展贡献力量。

林晓

2019 年 8 月 31 日

于建信金融科技上海事业群

4. 导论

凭借着年轻人的热情，QuantLib 的官方网站 曾经声明 QuantLib 旨在成为“标准的免费、开源金融软件库”。如果以不太严格的角度理解上述声明，人们可能会说它已经取得了一些成功——有一段时间是市面上唯一的开源金融软件库，虽然开始时采用了相当狡猾的诡计[1]。

无论是否成为标准，该项目正在蓬勃发展。在写作本书的时候，每个新版本都被下载了几千次，用户贡献源源不断，QuantLib 似乎被用在了现实世界中——就我所能猜到的通常情况而言，在金融世界中被秘密地使用。总而言之，作为项目管理员，我可以宣布自己心满意足。

但更重要的是，缺乏相应文档的弊端开始显现。虽然有详细的类参考说明可用（这是一件容易的事，因为可以自动生成），但它让人只见树不见树林。因此一个新用户可能会产生这样的印象：QuantLib 的开发人员一定和 Lewis Carroll 的诗《Hunting of the Snark》[2]里面的 Bellman 有相同的想法：

> “What use are Mercator’s North Poles and Equators,
> Tropics, Zones and Meridian Lines?”
> So the Bellman would cry: and the crew would reply,
> “They are merely conventional signs!”
>
> “Mercator[3] 的北极和赤道有什么用处，
> 还有热带、时区和子午线?”
> Bellman 叫喊道，船员们会回答，
> “它们只是些传统的标记！”

本书的目的是填补现有的部分空白。这是一篇关于 QuantLib 的设计和实现的报告，Mel Brooks 的《Young Frankenstein》恰如其分的反映了我[4]是如何创作这本书的，虽然精神上相似，但希望结果可能不那么可怕。如果你已经是，或者想成为 QuantLib 的用户，你可以在这里找到关于 QuantLib 设计的有用信息，这些信息在阅读代码时可能不太明显。如果你正在从事量化金融工作，即使平时不使用 QuantLib，你仍然可以将其视为金融软件库设计的一手资料来阅读。你会发现它涵盖了你可能会遇到的问题，以及一些可能的解决方案及其

[1] 一些富有教养的用户偶尔将我们的“QuantLib”称为“the QuantLib”。尽管我喜欢这个表达，但是迄今为止虚谦和习惯阻止我使用这种表达。

[2] 译注：《Hunting of the Snark》，译作《猎鲨记》，Lewis Carroll 于 1874 年创作的打油诗。Lewis Carroll 是英国作家、数学家、逻辑学家、摄影家，以及儿童文学作品《爱丽丝梦游仙境》与《爱丽丝镜中奇遇》的作者。

[3] 译注：Gerardus Mercator，16 世纪地图制图学家。

[4] 当然，应该说是“我们”。

基本原理。根据你面对问题的限制，你可能（甚至很可能）会选择其他解决方案，但你依然可以从 QuantLib 中获益。

在我的描述中，我还会指出当前实现中的缺点，不是为了贬低这个库（毕竟我已经深度参与其中），而是为了更有建设性的目的。一方面，描述现有的陷阱将有助于开发人员避免它们；另一方面，它可能会展示如何改进这个库。事实上，已经发生的事情是，审阅本书的代码使我可以回头对其进行更好的改进。

由于时间和篇幅的原因，我无法涵盖 QuantLib 的各个方面。在本书的前半部分，我将介绍一些最重要的类，例如用于构建金融工具和期限结构的类，这会让你看到 QuantLib 更宏观的架构。后半部分，我将介绍一些专门的框架，例如用于创建蒙特卡罗或有限差分模型的框架。其中一些框架比其余的另一些更精巧，我希望它们目前的缺点与它们的优点一样有趣。

本书主要针对想要用自己的工具或模型扩展 QuantLib 的用户，如果你希望这样做，类架构和框架的描述将为你提供将代码整合进 QuantLib，并充分利用其功能的有关信息。如果你不是这类用户，请不要合上该书，你也可以找到有用的信息。不过，你也许应该另外看一看《QuantLib Python Cookbook》这本书。

现在，按照传统，列举一些关于本书风格和要求的说明。

C++ 和量化金融方面的知识是必需的。我没有假装能够教你其中任何一个，而且这本书已经足够厚了。在这里，我只描述 QuantLib 的实现和设计，剩下的工作留给其他更好的作者，让他们来描述量化金融领域的问题，以及 C++ 语言的语法和技巧。

正如你已经注意到的那样，“我”会使用第一人称单数来写作。诚然，这可能看起来比较以自我为中心。事实上，我希望你还没有厌烦地放下这本书，但如果“我们”要使用第一人称复数形式，“我们”会感到相当浮夸。本书的作者对使用第三人有相同的感觉。经过一番思考，我选择了一种不那么正式但更舒适的风格（正如你所指出的那样，这也包括缩写的自由使用）。出于同样的原因，我会称“你”而不是通常使用的“读者”。单数的使用也有助于避免混淆，当我使用复数形式时，我指的是 QuantLib 开发人员作为一个整体所做的工作。

如果本身很有趣或者与最终结果相关，我会描述设计的演化过程。为了清楚起见，在大多数情况下，我会跳过死胡同和弯路，将不同时期作出的设计决策汇总在一起，只显示最终的设计，有时是简化过的。这仍然让我有很多话要说：用 Alabama Shakes 的歌词来说，“why is an awful lot of question”。[5]

我将指出所描述的代码中使用的设计模式。请注意，我并不是主张在你的代码里塞满设计模式，它们要在该用的时候用，而不是为了用而用。[6]然而，QuantLib 现在是若干年来编码和重构的结果，两者都基于用户的反馈和随时间推移而增加的新需求。设计朝向模式演变是很自然的。

[5]译注：出自美国乐队 Alabama Shakes 的《Boys & Girls》。
[6]对这一点更彻底的阐述参见 (Kerievsky, 2004)。

本书中代码的使用约定与 QuantLib 中的相同，我将在附录 B 中进行概述。有一点需要进行说明：由于对长度的限制，我可能会从类型名中隐去 `std` 和 `boost` 命名空间。当代码需要图表补充解释时，我将使用 UML，对于那些不熟悉这种语言的人，可以在 (Fowler, 2003) 中找到一个简明的指南。

嗯，就是这样。让我们开始吧。

5. 金融工具与定价引擎

金融库必须提供金融工具定价方法的说法肯定会吸引 de La Palisse 先生[1]。但是，这只是整个问题的一部分。金融库还必须为开发人员提供相关方法，以便通过添加新的定价功能扩展当前框架。

可预见的扩展有两种，库必须允许其中任何一种。一方面，必须可以增加新的金融工具；另一方面，增加现有金融工具新的定价方法必须是可行的。这两种扩展都有一些需求，或者用模式术语来说，解决方案必须协调一致。本章详细介绍了这些需求，并介绍了 QuantLib 得以满足这些需求的设计。

5.1 `Instrument` 类

在我们的领域，金融工具本身就是一个概念。仅仅因为这个原因，任何有自尊的面向对象程序员都会将它编码为一个基类，从中派生出特定的金融工具。

当然，有了这个想法是能够编写这样的代码，

```
for (i = portfolio.begin(); i != portfolio.end(); ++i)
    totalNPV += i->NPV();
```

我们不必关心每种金融工具的具体类型。但是，这也无法让我们知道传递给 `NPV` 方法的参数，甚至背后调用了什么方法。因此，即使上述两条看起来无害的线索告诉我们，我们必须退后一步，思考一下接口。

5.1.1 接口与需求

存在着各种各样的交易资产，从最简单的到最复杂的，特定类型金融工具（比如股票期权）的任何特定方法对于其他类型的金融工具（例如利率互换）而言是不适用的。因此，只有很少的通用方法适合作为 `Instrument` 类的接口。我们将目标局限于那些返回其现值（可能关联一个误差估计值）并指示金融工具是否到期的方法。因为我们无法事先指定这些接口需要哪些参数，[2]所以，这些方法不接受参数输入，任何需要的输入都必须由金融工具对象本身存储。结果，接口如下所示。

[1]译注：de La Palisse 先生出自《The Song of La Palice》，法国诗人 Bernard de la Monnoye 为法国贵族与军事统帅 Jacques de la Palice 写的歌，歌词衍生出法语词汇 Lapalissade，意为显而易见的真相。

[2]甚至像可变模版参数（variadic template）一样新奇的 C++11 特性也无济于事。

`Instrument` 类的主要接口。

```
class Instrument {
  public:
    virtual ~Instrument();
    virtual Real NPV() const = 0;
    virtual Real errorEstimate() const = 0;
    virtual bool isExpired() const = 0;
};
```

最好的做法是，这些方法首先被声明为纯虚方法。但是，正如 Sportin' Life 在 Gershwin 的《Porgy and Bess》[3]中所指出的那样——这不是必须的。可能有一些行为可以在基类中编码。为了弄清楚是否属于这种情况，我们必须分析从通用金融工具中预期得到什么，并检查它是否能以通用的方式实现。在不同的时间发现了两个这样的需求，在 QuantLib 的发展过程中它们的实现发生了变化，我在这里呈现它们目前的形式。

一个需求是，一个给定的金融工具可能以不同的方式定价（例如，用一个或多个解析公式或数值方法），而不必借助于继承。在这一点上，你可能会想到“策略模式”。确实如此，我将在第 2.2 节讨论其实现。

第二个需求来自于我们观察到金融工具的价值取决于市场数据。这些数据的值随时间而变化，进而金融工具的价值产生变化；另一个变化因素是任何单一的市场数据都可以由不同的来源提供。我们希望金融工具能够保持与这些数据源的联系，以便在不同的调用中，它们的方法可以获取最新的值并相应地重新计算结果；另外，我们希望能够在数据源之间透明地切换，并让金融工具仅仅将其视为数据值发生的更改。

我们也担心潜在的效率损失。例如，我们可以通过将其金融工具存储在容器中，定期轮询它们的价值并添加到最终结果中，从而及时监控投资组合的价值。在一个简单的实现中，即使对那些输入没有变化的金融工具也会引发重新计算。因此，我们决定向金融工具方法添加一个缓存机制：存储之前的结果，并且只有在任何输入发生变化时才重新计算的机制。

5.1.2 实现

管理缓存和重新计算金融工具价值的代码是通过两种设计模式为通用金融工具编写的。

当任何输入改变时，通过观察者模式 (Gamma *et al*, 1995) 通知金融工具对象。附录 A 中简要描述了该模式[4]，我在这里只描述相关内容。

显然，金融工具扮演观察者的角色，而输入数据则扮演被观察者的角色。为了在被通知输入发生变化后可以访问新值，观察者需要维护表示输入对象的引用。这可能暗示使用某种智能指针。然而，指针的行为不足以完全描述我们的问题。正如我已经提到的，变化可能

[3]译注：《Porgy and Bess》，《波吉与贝丝》，又译做《乞丐与荡妇》，歌剧，由 Gershwin 谱曲，Sportin' Life 是剧中的一名毒品贩子。

[4]这不意味着你不用读 Gang of Four 的那本书。

不仅源于数据源的值随时间而变化，我们可能还想切换到其他数据源。存储（智能）指针可以让我们访问指向对象的当前值，但是我们的指针副本，对于观察者来说是私有的，不能指向其他不同的对象。因此，我们需要的是指向指针的指针的智能等价物。这个特性在 QuantLib 中以类模板的形式实现，并命名为 `Handle`。同样，细节在附录 A 中给出，与此讨论相关的事实是给定 `Handle` 的副本共享一个对象的链接。当链接指向另一个对象时，会通知所有副本，并允许其持有者访问新指针。此外，`Handle` 将来自指向对象的任何通知传递给它们的观察者。

最后，实现了作为可观察数据，并可以存储到 `Handle` 中的类。最基本的是 `Quote` 类，代表了单一可变的市场数据值。金融工具估值的其他输入可以包括更复杂的对象，如利率或波动率期限结构。[5]

另一个问题是抽象出用于存储和重新计算缓存结果的代码，同时仍然将其留给派生类来实现任何特定的计算。在早期版本的 QuantLib 中，该功能包含在 `Instrument` 类中。之后，它被提取并编码到另一个类中，不出所料地称为 `LazyObject`，现在被 QuantLib 的其他部分重用。下面的代码显示了该类的概要。

`LazyObject` 类的概要。

```
class LazyObject : public virtual Observer,
                   public virtual Observable {
  protected:
    mutable bool calculated_;
    virtual void performCalculations() const = 0;
  public:
    void update() { calculated_ = false; }
    virtual void calculate() const {
        if (!calculated_) {
            calculated_ = true;
            try {
                performCalculations();
            } catch (...) {
                calculated_ = false;
                throw;
            }
        }
    }
};
```

代码很简单。定义了一个布尔数据成员 `calculated_`，用于跟踪结果是否被计算并且仍然有效。`update` 方法实现了 `Observer` 接口，并在被观察对象发出通知时被调用，它将上面的布尔值设置为 `false`，并使之前的结果失效。

`calculate` 方法通过模板方法模式 (Gamma *et al*, 1995) 实现，有时候也称为 NVI（non-virtual interface）模式。正如 Gang of Four 的那本书所述，算法的不变部分（在这里是管理缓存结果）在基类中实现；变化部分（这里是实际计算）被委托给一个虚方法，即 `performCalculations`，

[5] 很可能，这些对象最终也依赖于 `Quote` 实例，例如，利率期限结构可能依赖于用于计算利率的存款利率和互换利率。

它在基类方法的主体中被调用。因此，派生类将只实现它们的具体计算，而不必关心缓存：相关代码将由基类注入。

缓存的逻辑很简单。如果当前结果不再有效，我们让派生类执行所需的计算并将新结果标记为最新。如果目前的结果是有效的，我们什么也不做。

但是，实现并不那么简单。你可能想知道为什么我们在设置 `calculated_` 时必须事先插入 `try` 代码块，并且在再次抛出异常之前处理程序回滚更改。毕竟，我们可以更简单地编写算法的主体，例如，如下所示，看似等价的代码，不会捕获和重新抛出异常：

```
if (!calculated_) {
    performCalculations();
    calculated_ = true;
}
```

原因在于有些情况（例如，惰性对象是利率期限结构时，该对象会惰性地 bootstrap）中 `performCalculations` 发生递归调用 `calculate`。如果 `calculate_` 没有设置为 `true`，`if` 条件仍然可以触发，`performCalculations` 会再次被调用，导致无限递归。将此标志设置为 `true` 可防止发生这种情况。但是，如果抛出异常，现在必须小心将其恢复为 `false`。然后重新抛出异常，以便它可以被错误处理程序捕获。

`LazyObject` 提供了更多的方法，使用户能够防止或强制重新计算结果。在这里不讨论它们。如果你有兴趣，你可以听从 Obi-Wan Kenobi 大师给出的建议：“Read the source, Luke.”[6]

题外话：常量还是非常量？

解释为什么 `NPV_` 被声明为 `mutable` 是个有趣的话题，因为这是在实现缓存或惰性计算时经常出现的问题。问题的关键在于 `NPV` 方法在逻辑上是一个常量函数：计算一个金融工具的价值不会修改它自身。因此，用户有权期望可以在常量实例上调用此类方法。反过来，`NPV` 的常量函数性迫使我们将 `calculate` 和 `performCalculations` 声明为 `const` 的。然而，我们选择惰性计算结果并将其存储起来以备后用，这就需要在这种方法的主体中分配一个或多个数据成员。通过将缓存变量声明为 `mutable` 来解决僵局，这让我们（和派生类的开发者）满足了这两个需求，即 `NPV` 方法的常量函数性和对数据成员的惰性赋值。

另外，应该注意的是，C++11 标准现在要求常量方法是线程安全的，也就是说，在同一时间调用常量成员的两个线程不应该在竞争状态下发生（要了解所有信息，请参阅 (Sutter, 2013)）。为了使代码符合新标准，我们应该使用互斥锁来保护对 `mutable` 成员的更新。这可能需要对设计进行一些修改。

`Instrument` 类继承自 `LazyObject`。为了实现前面介绍的接口，它使用特定于金融工具的代码来修饰 `calculate` 方法。下面的代码中显示了所得到的方法以及其他支持代码。

[6]译注：source 是指“源代码”，与 force（原力）谐音，这里是在模仿《星球大战》中 Obi-Wan Kenobi 的台词——“Use the Force, Luke.”

Instrument 类的概要。

```
class Instrument : public LazyObject {
  protected:
    mutable Real NPV_;
  public:
    Real NPV() const {
        calculate();
        return NPV_;
    }
    void calculate() const {
        if (isExpired()) {
            setupExpired();
            calculated_ = true;
        } else {
            LazyObject::calculate();
        }
    }
    virtual void setupExpired() const {
        NPV_ = 0.0;
    }
};
```

再次，添加的代码遵循模板方法模式，将特定金融工具的计算委托给派生类。该类定义了一个 NPV_ 数据成员来存储计算结果，派生类可以声明其他数据成员来存储特定的结果。[7]calculate 方法的主体调用虚方法 isExpired 来检查金融工具是否到期。如果是这种情况，它会调用另一个虚方法，即 setupExpired，它负责给结果赋予有意义的值，其默认实现将 NPV_ 设置为 0，并且可以由派生类调用。随后 calculated_ 被设置为 true。如果金融工具没有到期，则调用 LazyObject 的 calculate 方法，然后根据需要调用 performCalculations。这对后一种方法施加了一个约束，即它在派生类中的实现需要将 NPV_（以及任何其他特定金融工具的数据成员）设置为计算结果。最后，NPV 方法确保在返回答案之前调用 calculate。

5.1.3 示例：利率互换

我将展示一个特定金融工具的实现来结束本节，其实现基于先前描述的若干工具。

所选择的金融工具是利率互换。如你所知，这是一个关于定期交换现金流的合约。该工具的净现值通过增加或减去贴现现金流（取决于是支付还是收取的）来计算。

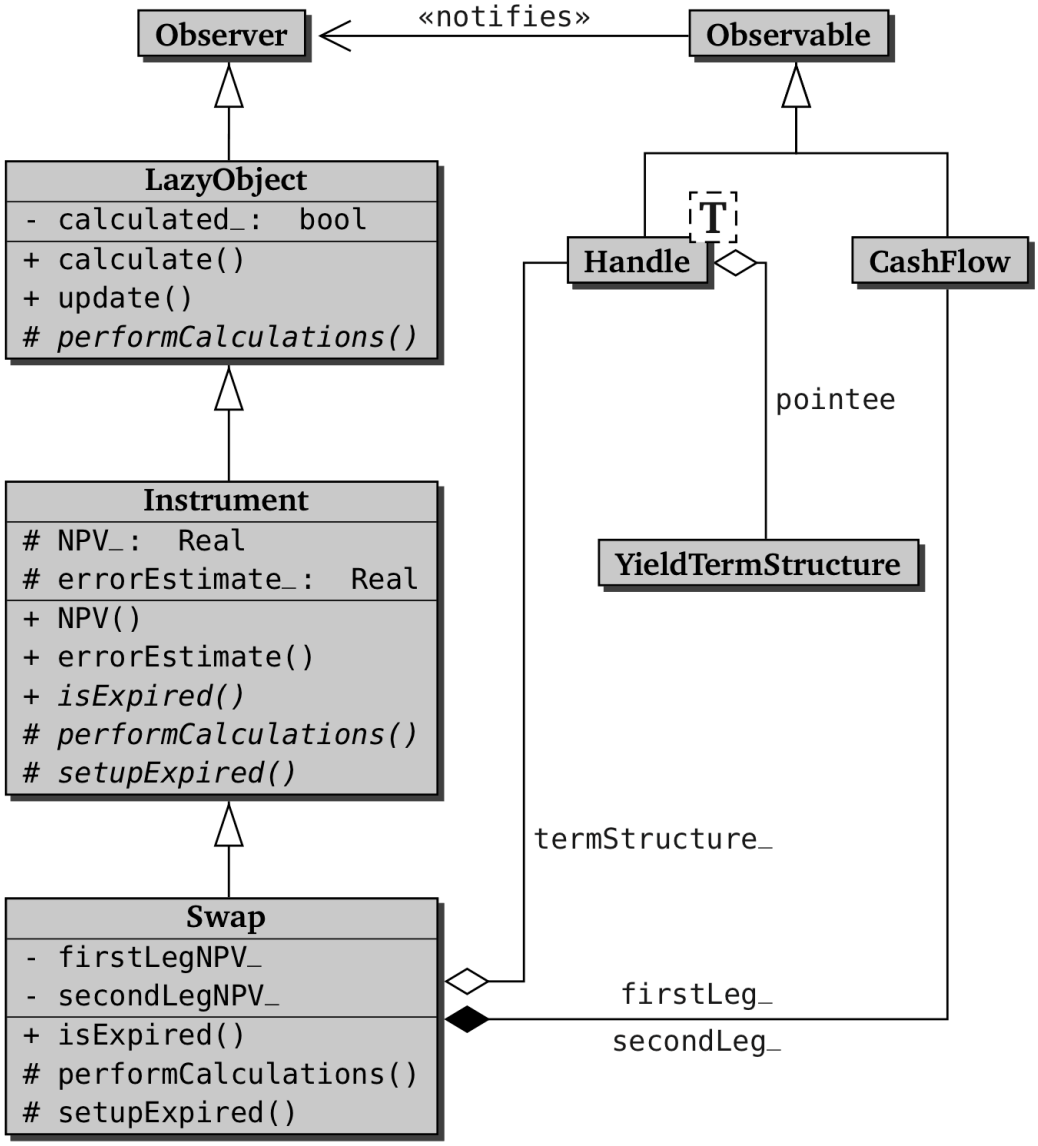
毫不奇怪，互换作为 Instrument 的派生类实现[8]。其概要显示如下。

[7] Instrument 类还定义了一个 errorEstimate_ 成员，为了简明起见，此处省略。NPV_ 的讨论也是这样。
[8] 本节中显示的实现有些过时。然而，由于它提供了一个简单的例子，因此原始实现仍然在这里使用。

Swap 类的部分接口。

```
class Swap : public Instrument {
  public:
    Swap(const vector<shared_ptr<CashFlow>>& firstLeg,
         const vector<shared_ptr<CashFlow>>& secondLeg,
         const Handle<YieldTermStructure>& termStructure);
    bool isExpired() const;
    Real firstLegBPS() const;
    Real secondLegBPS() const;
  protected:
    // methods
    void setupExpired() const;
    void performCalculations() const;
    // data members
    vector<shared_ptr<CashFlow>> firstLeg_, secondLeg_;
    Handle<YieldTermStructure> termStructure_;
    mutable Real firstLegBPS_, secondLegBPS_;
};
```

它包含用于计算的若干数据成员对象，即第一组和第二组现金流、用于现金流贴现的利率期限结构，以及用于存储其他额外结果的两个变量。此外，它声明了 Instrument 接口的方法，以及其他方法返回利率互换才有的特定结果。下图显示了 Swap 和相关类的类图。



Swap 的类图

这个类对 Instrument 框架的适配分三步完成，第三步是可选的，取决于派生类。下面的代码中会显示相关的方法。

Swap 类的部分实现。

```
Swap::Swap(const vector<shared_ptr<CashFlow>>& firstLeg,
           const vector<shared_ptr<CashFlow>>& secondLeg,
           const Handle<YieldTermStructure>& termStructure)
    : firstLeg_(firstLeg), secondLeg_(secondLeg),
      termStructure_(termStructure) {
    registerWith(termStructure_);
    vector<shared_ptr<CashFlow>>::iterator i;
    for (i = firstLeg_.begin(); i != firstLeg_.end(); ++i)
        registerWith(*i);
    for (i = secondLeg_.begin(); i != secondLeg_.end(); ++i)
        registerWith(*i);
}
bool Swap::isExpired() const {
    Date settlement = termStructure_->referenceDate();
    vector<shared_ptr<CashFlow>>::const_iterator i;
    for (i = firstLeg_.begin(); i != firstLeg_.end(); ++i)
        if (!(*i)->hasOccurred(settlement))
            return false;
    for (i = secondLeg_.begin(); i != secondLeg_.end(); ++i)
        if (!(*i)->hasOccurred(settlement))
            return false;
    return true;
}
void Swap::setupExpired() const {
    Instrument::setupExpired();
    firstLegBPS_ = secondLegBPS_ = 0.0;
}
void Swap::performCalculations() const {
    NPV_ = -Cashflows::npv(firstLeg_, **termStructure_)
           + Cashflows::npv(secondLeg_, **termStructure_);
    errorEstimate_ = Null<Real>();
    firstLegBPS_ = -Cashflows::bps(firstLeg_, **termStructure_);
    secondLegBPS_ = Cashflows::bps(secondLeg_, **termStructure_);
}
Real Swap::firstLegBPS() const {
    calculate();
    return firstLegBPS_;
}
Real Swap::secondLegBPS() const {
    calculate();
    return secondLegBPS_;
}
```

第一步在类的构造函数中执行，该类的构造函数所接受的参数（并拷贝到将相应数据成员）包括用于交换的两个现金流序列和用于贴现计算的利率期限结构。这一步骤本身就是将互换对象注册为现金流和期限结构的观察者。如前所述，这使得它们可以在每次发生变化时能够通知互换对象并触发其重新计算。

第二步是实现所要求的接口。`isExpired` 方法的逻辑非常简单，遍历存储的现金流，检查它们的支付日期。一旦发现还没有发生的支付，它就会将互换对象报告为未到期。如果没有

找到，互换已经到期。在这种情况下，将调用 setupExpired 方法。它的实现调用基类中的 setupExpired 方法，进而处理从 Instrument 继承的数据成员，然后将互换特有的结果设置为 0。

> **题外话：句柄还是共享指针。**
>
> 你可能想知道为什么 Swap 的构造函数以句柄接受贴现曲线，而以简单的共享指针接受现金流。原因是我们可能决定切换到不同的曲线（可以通过句柄完成），而现金流是互换定义的一部分，因此是不可变的。

最后要求的方法是 performCalculations。通过调用 Cashflows 类中的两个外部函数来执行计算。[9]第一个，即 npv，是上述算法的简单转换：它循环一列现金流，添加未来现金流的贴现金额。我们将 NPV_ 变量设置为两组现金流的差。第二个，即 bps，计算一列现金流的基点敏感度（BPS）。我们在每组现金流上调用一次，并将结果存储在相应的数据成员中。由于结果没有误差数字，因此 errorEstimate_ 变量被设置为 Null<Real>()——一个特定的浮点值，用作指示无效数字。[10]

第三步也是最后一步，只有在类定义了额外结果的情况下需要执行。它包括编写相应的方法（这里是 firstLegBPS 和 secondLegBPS），它们确保在返回存储结果之前（惰性地）执行计算。

现在实现完成了。已经存在了 Instrument 类，Swap 类将受益于它的代码。因此，它将根据来自其输入的通知自动缓存并重新计算结果，除了注册调用以外，没有在 Swap 中写入相关代码。

5.1.4 未来的发展

你可能已经注意到我对前一个例子和 Instrument 类的处理存在缺陷。尽管是通用的，但我们实现的 Swap 类无法管理涉及两种不同货币的利率互换。如果用户要添加两个基于不同货币的金融工具，则会出现类似的问题，用户必须在添加二者之前手动将其中一个值转换为另一个的货币。

这样的问题源于实现的一个缺点：我们使用 Real 类型（即简单的浮点数）来表示金融工具或现金流的值。因此，这样的结果会忽略现实世界中附加的货币信息。

如果我们通过 Money 类表达这样的结果，那么缺陷可能会被消除。这种类的实例包含货币信息，还取决于用户设置，它们能够在加或减时自动执行转换为通用货币。

[9]如果你碰巧感觉被愚弄了，请考虑这个例子的重点是展示如何将计算打包到类中，而不是显示如何实现计算。你的好奇心将在后面的专门讨论现金流和相关函数的章节中得到满足。

[10]NaN 可能是一个更好的选择，但检测它的手段并不简便。还有一种可能的方法是使用 boost::optional。

但是，这将是一个重大变化，以很多方式影响了大部分代码。因此，在我们解决它之前，我们需要一些深思熟虑（如果我们真的解决了这个问题的话）。

另一个（也是更微妙的）缺陷是 `Swap` 类无法明确区分它所表示的两个抽象化组件。即，指定合约的数据（现金流的技术参数）与用于对该工具定价的市场数据（当前贴现曲线）之间没有明确的区分。

解决方案是只在金融工具中存储第一组数据（即那些将在其术语表中的），并将市场数据保存在其他地方。[11]实现这一点的方法是下一节的主题。

5.2 定价引擎

现在我们转到上一节中提到的第二个需求。对于任何给定的金融工具，并不总是存在唯一的定价方法。此外，可能出于不同原因需要使用多种方法。让我们以经典教科书中的欧式股票期权为例。有人可能想通过解析的 Black-Scholes 公式来定价，以从市场价格中推算出隐含波动率；通过随机波动率模型来校准后者，并将其用于奇异期权；通过有限差分法来比较解析结果与数值结果，并验证其有限差分法的实现；或使用蒙特卡罗模型，把欧式期权作为奇异期权的控制变量。

因此，我们希望单一金融工具可以用不同方式定价。当然，不希望赋予 `performCalculations` 方法不同的实现，因为这会迫使人们为单一金融工具类型使用不同的类。在我们的例子中，我们最终会得到一个 `EuropeanOption` 基类，从中可以派生出 `AnalyticEuropeanOption`、`McEuropeanOption` 等等。至少在两个方面这是错误的。在概念层面上，当需要单一实例时，会引入不同的实例：正如 Gertrude Stein 所言，欧式期权就像欧式期权就是欧式期权。[12]在可用层面上，会使运行时切换定价方法变得不可能。

解决方案是使用策略模式，即让金融工具接受一个封装了所要执行计算的对象。我们称这种对象为“定价引擎”。一个给定的金融工具将能够采取多种可用引擎中的任何一种（当然对应于金融工具的类型），为选定的引擎传递所需的参数，使其能够计算金融工具的价值以及任何其他想要的量，并且获取结果。因此，`performCalculations` 方法大致可以如下实现：

```
void SomeInstrument::performCalculations() const {
    NPV_ = engine_->calculate(arg1, arg2, ..., argN);
}
```

这里假设虚方法 `calculate` 已经在引擎接口中定义并且在特定引擎中实现。

不幸的是，上述方法不会如此。问题是，我们只想在 `Instrument` 类中实现一次调度代码。但是，该类不知道参数的数量和类型。不同的派生类在数据成员的数量和类型上可能有很

[11]除了在概念上更清楚之外，这对于实现金融工具的序列化和反序列化的外部功能是有用的，例如与 FpML（http://www.fpml.org/）格式的相互转换。

[12]译注：Gertrude Stein，美国作家与诗人，作者在模仿她在《Sacred Emily》中的名句——“Rose is a rose is a rose is a rose”。

大的不同，返回的结果也是如此。例如，利率互换可能会为其固定利率返回公允价值以及浮动利差，而无处不在的欧式期权可能会返回任何希腊值。

如上所述，向引擎传递显式参数的接口会导致不想看到的后果。不同金融工具的定价引擎会有不同的接口，这会阻止我们定义一个单独的基类。因此，调用引擎的代码将不得不在每个金融工具类中重复。这样想下去是要发疯的。[13]

我们选择的解决方案是通过称为 `arguments` 和 `results` 的不透明结构体（opaque structure）从引擎传递并接受参数和结果。由此衍生而来的两种结构体，用金融工具的特定数据加以扩充，将出现在任何定价引擎中，金融工具对象会写入和读取这些数据以便与定价引擎交换信息。

下面的代码显示了 `PricingEngine` 类的接口和它内部的 `arguments` 和 `results` 类，以及辅助的 `GenericEngine` 类模板。后者实现了大部分 `PricingEngine` 接口，只给特定引擎的开发人员遗留了 `calculate` 方法的实现。`arguments` 和 `results` 类提供了一些方法，这些方法可以简化它们作为数据下拉框来使用：写入输入数据之后调用 `arguments::validate`，以确保它们的值位于有效范围内；而 `results::reset` 将在引擎开始计算之前调用，以清除先前的结果。

`PricingEngine` 类与相关类的接口。

```
class PricingEngine : public Observable {
  public:
    class arguments;
    class results;
    virtual ~PricingEngine() {}
    virtual arguments* getArguments() const = 0;
    virtual const results* getResults() const = 0;
    virtual void reset() const = 0;
    virtual void calculate() const = 0;
};
class PricingEngine::arguments {
  public:
    virtual ~arguments() {}
    virtual void validate() const = 0;
};
class PricingEngine::results {
  public:
    virtual ~results() {}
    virtual void reset() = 0;
};
// ArgumentsType must inherit from arguments;
// ResultType from results.
template <class ArgumentsType, class ResultsType>
class GenericEngine : public PricingEngine {
  public:
    PricingEngine::arguments* getArguments() const {
        return &arguments_;
    }
    const PricingEngine::results* getResults() const {
```

[13]译注：原文是“This way madness lies”，化用了莎士比亚的《李尔王》中的台词“That way madness lies”。

```
        return &results_;
    }
    void reset() const { results_.reset(); }
  protected:
    mutable ArgumentsType arguments_;
    mutable ResultsType results_;
};
```

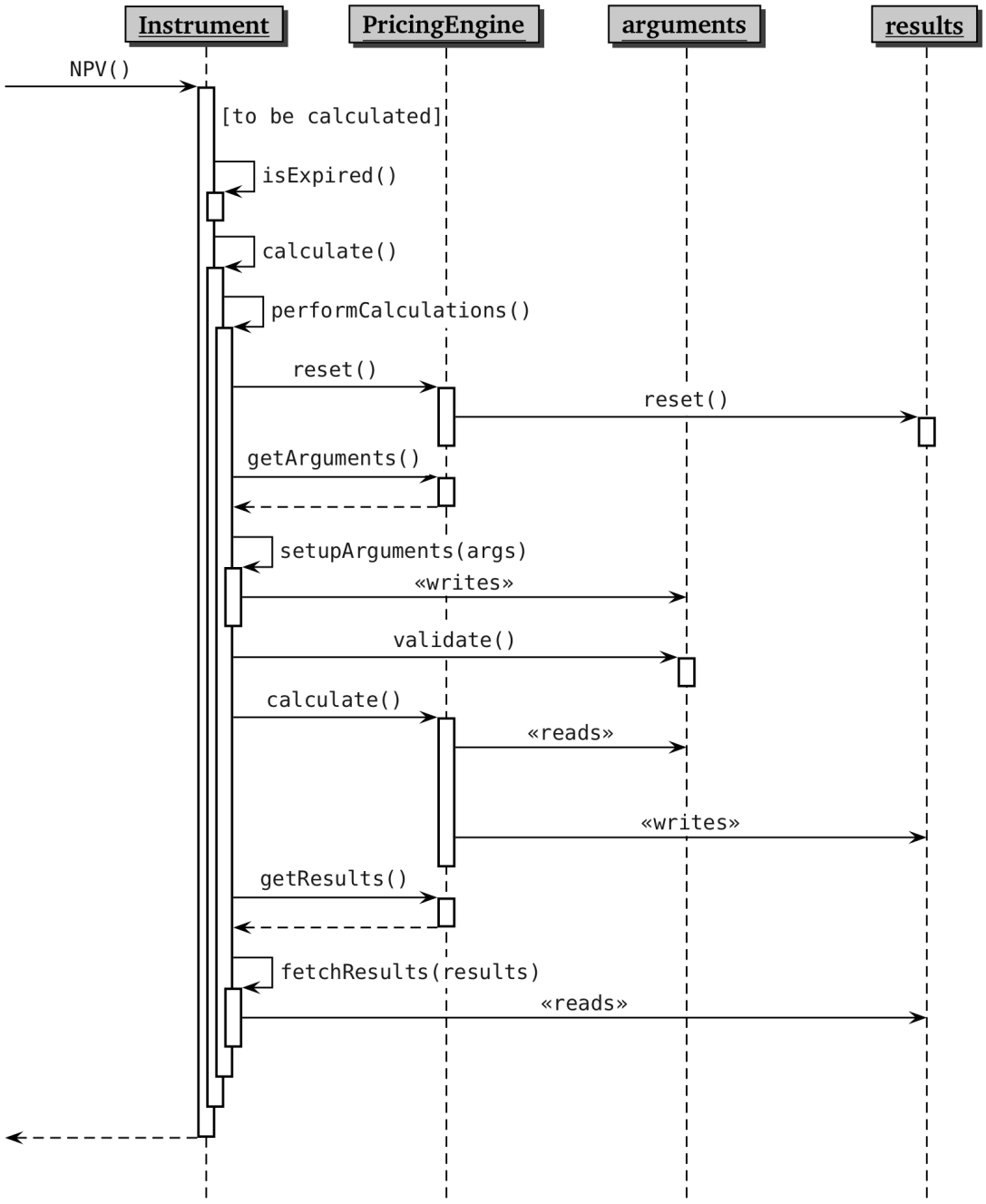
用我们的新类武装起来，现在可以编写一个通用的 `performCalculation` 方法。除了已经提到的策略模式之外，我们将使用模板方法模式来允许任何给定的金融工具去填充缺失的部分。下面的代码显示了最终的实现。请注意，定义的内部类 `Instrument::results` 继承自 `PricingEngine::results`，并包含任何金融工具需要提供的结果。[14]

摘录的 `Instrument` 类代码。

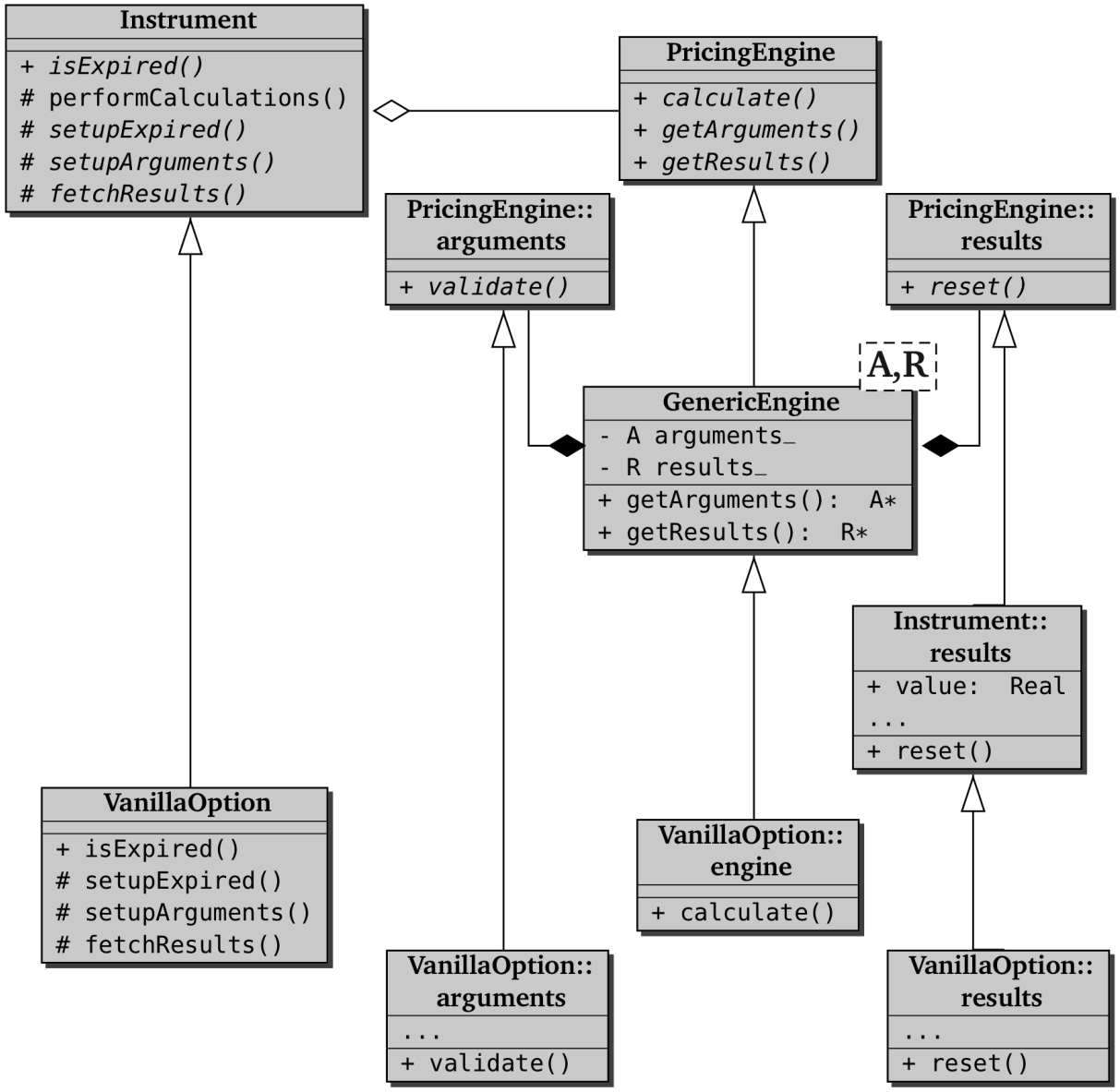
```
class Instrument : public LazyObject {
  public:
    class results;
    virtual void performCalculations() const {
        QL_REQUIRE(engine_, "null pricing engine");
        engine_->reset();
        setupArguments(engine_->getArguments());
        engine_->getArguments()->validate();
        engine_->calculate();
        fetchResults(engine_->getResults());
    }
    virtual void setupArguments(
        PricingEngine::arguments*) const {
        QL_FAIL("setupArguments() not implemented");
    }
    virtual void fetchResults(
        const PricingEngine::results* r) const {
        const Instrument::results* results = dynamic_cast<const Value*>(r);
        QL_ENSURE(results != 0, "no results returned");
        NPV_ = results->value;
        errorEstimate_ = results->errorEstimate;
    }
    template <class T>
    T result(const string& tag) const;
  protected:
    boost::shared_ptr<PricingEngine> engine_;
};
class Instrument::results
    : public virtual PricingEngine::results {
  public:
    Value() { reset(); }
    void reset() {
        value = errorEstimate = Null<Real>();
    }
    Real value;
    Real errorEstimate;
};
```

[14] `Instrument::results` 类还包含一个 `std::map`，定价引擎可以用来存储额外的结果。为了简明起见，相关代码在这里省略。

至于 performCalculation，实际的工作被分解到许多协作类——金融工具、定价引擎、参数和结果类。借助下面两副图中的第一个 UML 时序图，最有助于理解这种协作的动态性（在下面的段落中描述），第二副图显示了类（和一个可能的具体金融工具）之间的静态关系。



金融工具和定价引擎相互作用的时序图



`Instrument` 和 `PricingEngine` 的类图，以及一个相关的派生金融工具类

对金融工具 `NPV` 方法的调用最终会触发（如果金融工具未到期，并且需要计算相关量）调用 `performCalculations` 方法。金融工具和定价引擎之间的相互作用开始于此。首先，金融工具验证引擎是否可用，如果不可用，则中止计算。如果找到一个引擎，金融工具会提示它自行重置。该消息通过其 `reset` 方法传递给特定金融工具的结果结构体。执行后，该结构体处于空白状态，准备写入新的结果。

此时，模板方法模式进入场景。该金融工具请求定价引擎的参数结构体，该参数结构体作为 `arguments` 指针返回。然后指针传递给金融工具的 `setupArguments` 方法，该方法充当模式

中的可变部分。根据具体的金融工具，这种方法验证传递来的参数是正确的类型，并继续用正确的值填充其数据成员。最后，通过调用 `validate` 方法，参数被要求对新写入的值执行任何需要的检查。

现在已经准备好了策略模式。它的参数被设置，选中的引擎被要求执行其特定的计算，并在其 `calculate` 方法中实现。在处理过程中，引擎将从其参数结构体中读取所需的输入，并将相应的输出写入其结果结构体中。

引擎完成其工作后，控制权返回到 `Instrument` 实例并且模板方法模式继续展开。被调用的 `fetchResults` 方法现在必须向引擎请求结果，向下搜索它们以访问所包含的数据，并将这些值复制到它自己的数据成员中。`Instrument` 类定义了一个默认实现，它提取所有金融工具共有的结果，派生类可能会扩展它以读取特定结果。

> **题外话：非纯虚方法。**
>
> 在查看 `Instrument` 类的最终实现时，你可能想知道为什么 `setupArguments` 方法被定义为抛出异常而不是声明为纯虚方法。原因不是强迫新金融工具的开发人员实现一个无意义的方法，而是要他们决定某些类应该简单地覆盖 `performCalculation` 方法。

5.2.1 示例：普通香草期权

现在，一个例子是必要的。一个警告：尽管 QuantLib 中存在一个类，它实现了普通的香草期权，即简单的看涨看跌期权，以欧式、美式或百慕大式行权，这样的类实际上是深度类层级结构的最底层。将 `Instrument` 类作为根，这样的层级结构首先用一个 `Option` 类专门化，接着再用一个 `OneAssetOption` 类（单一底层资产的期权）专门化，通过另一个或两个类，直到它最终定义了我们感兴趣的 `VanillaOption` 类。

这样做是有好处的，例如，`OneAssetOption` 类中的代码可以自然地重用于亚式期权，而 `Option` 类中的代码可以在实现各种篮子期权时复用。不幸的是，这导致用于定价的代码在所描述继承链的所有成员之间传播，这让例子变得不明确。因此，我将描述一个简化的 `VanillaOption` 类，它与 QuantLib 中的实现相同，但直接从 `Instrument` 类继承，所有中间类中实现的代码将显示为在示例类中的实现，而不是继承。

下面的代码显示了我们的香草期权类的接口。它声明了 `Instrument` 的接口所要求的方法，并为用户提供了额外的结果，也就是期权的希腊值，正如前一节所指出的那样，相应的数据成员被声明为 `mutable`，以便它们的值可以在常量函数 `calculate` 中设置。

VanillaOption 类的接口。

```
class VanillaOption : public Instrument {
  public:
    // accessory classes
    class arguments;
    class results;
    class engine;
    // constructor
    VanillaOption(const boost::shared_ptr<Payoff>&,
                  const boost::shared_ptr<Exercise>&);
    // implementation of instrument method
    bool isExpired() const;
    void setupArguments(Arguments*) const;
    void fetchResults(const Results*) const;
    // accessors for option-specific results
    Real delta() const;
    Real gamma() const;
    Real theta() const;
    // ...more greeks
  protected:
    void setupExpired() const;
    // option data
    boost::shared_ptr<Payoff> payoff_;
    boost::shared_ptr<Exercise> exercise_;
    // specific results
    mutable Real delta_;
    mutable Real gamma_;
    mutable Real theta_;
    // ...more
};
```

除了自己的数据和方法外，VanillaOption 声明了许多辅助类：即特定的参数和结果结构体，以及基本定价引擎类。它们被定义为内部类以突出它们与期权类之间的关系，它们的接口显示在下面的代码中。

VanillaOption 内部类的接口。

```
class VanillaOption::arguments
    : public PricingEngine::arguments {
  public:
    // constructor
    arguments();
    void validate() const;
    boost::shared_ptr<Payoff> payoff;
    boost::shared_ptr<Exercise> exercise;
};
class Greeks : public virtual PricingEngine::results {
  public:
    Greeks();
    Real delta, gamma;
    Real theta;
    Real vega;
    Real rho, dividendRho;
```

```
};
class VanillaOption::results
    : public Instrument::results,
      public Greeks {
  public:
    void reset();
};
class VanillaOption::engine
    : public GenericEngine<VanillaOption::arguments,
                           VanillaOption::results> {};
```

可以对这些辅助类进行两点评论。首先，我在我的例子中引入了一个例外，我没有将所有数据成员都声明为 `results` 类，这是为了指出实现细节。人们可能想要定义一些结构体，这些结构体具有一些相关和常用的结果，这样的结构体可以通过继承的方式复用，例如在这里由 `Instrument::results` 组成的 `Greeks` 结构体来获得最终结构。在这种情况下，必须使用虚继承 `PricingEngine::results` 来避免臭名昭着的继承钻石（例如，参见 (Stroustrup, 2013)，书名称位于索引中）。

第二个意见是，如图所示，从类模板 `GenericEngine` 继承（用正确的参数和结果类型实例化）就足以为特定金融工具的定价引擎提供基类。我们将看到派生类只需要实现它们的 `calculate` 方法。

我们现在转向实现 `VanillaOption` 类，如下所示。

`VanillaOption` 类的实现。

```
VanillaOption::VanillaOption(
    const boost::shared_ptr<StrikedTypePayoff>& payoff,
    const boost::shared_ptr<Exercise>& exercise)
    : payoff_(payoff), exercise_(exercise) {}
bool VanillaOption::isExpired() const {
    Date today = Settings::instance().evaluationDate();
    return exercise_->lastDate() < today;
}
void VanillaOption::setupExpired() const {
    Instrument::setupExpired();
    delta_ = gamma_ = theta_ = ... = 0.0;
}
void VanillaOption::setupArguments(
    PricingEngine::arguments* args) const {
    VanillaOption::arguments* arguments =
        dynamic_cast<VanillaOption::arguments*>(args);
    QL_REQUIRE(arguments != 0, "wrong argument type");
    arguments->exercise = exercise_;
    arguments->payoff = payoff_;
}
void VanillaOption::fetchResults(
    const PricingEngine::results* r) const {
```

```
    Instrument::fetchResults(r);
    const VanillaOption::results* results =
        dynamic_cast<const VanillaOption::results*>(r);
    QL_ENSURE(results != 0, "wrong result type");
    delta_ = results->delta;
    ...    // other Greeks
}
Real VanillaOption::delta() const {
    calculate();
    QL_ENSURE(delta_ != Null<Real>(), "delta not given");
    return delta_;
}
```

它的构造函数需要一些定义该金融工具的对象。其中大部分将在后面的章节或附录 A 中描述。目前，需要说明 payoff 包含期权的敲定价和类型（即看涨或看跌期权），并且 exercise 包含有关行权日期和方式（即欧式、美式或百慕大式）的信息。传递来的参数存储在相应的数据成员中。另外请注意，它们不包括市场数据，那些数据将在其他地方传递来。

与到期有关的方法很简单，isExpired 检查是否过了最新的行权日期，而 setupExpired 调用基类实现并将特定于金融工具的数据设置为 0。

setupArguments 和 fetchResults 方法更有趣一些。前者首先将泛型 arguments 指针向下转换为所需的实际类型，然后转向实际的工作，如果传递另一种类型则引发异常。在某些情况下，数据成员只是逐字地复制到相应的参数槽中。但是，可能会出现这样的情况：许多引擎需要进行相同的计算（比如将日期转换为时间），setupArguments 提供了一个一次完成计算的地方。

fetchResults 方法是 setupArguments 的对偶。它也首先向下转换传递来的 results 指针，在验证了指针的实际类型之后，只是将结果复制到自己的数据成员中。

任何返回额外结果的方法（例如 delta）和 NPV 的工作方式一样：它会调用 calculate 方法，检查相应的结果是否被缓存（因为任何给定的引擎可能会，也可能不会计算它），并返回。

虽然很简单，一旦金融工具被配置引擎并执行所需的计算，上述实现是我们保证金融工具能够工作的一切基础。下面的代码描述了一种引擎，实现了欧式期权的 Black-Scholes-Merton 定价公式。

`VanillaOption` 类的一种定价引擎。

```
class AnalyticEuropeanEngine
    : public VanillaOption::engine {
  public:
    AnalyticEuropeanEngine(
        const shared_ptr<GeneralizedBlackScholesProcess>&
            process)
        : process_(process) {
        registerWith(process);
    }
    void calculate() const {
        QL_REQUIRE(
            arguments_.exercise->type() == Exercise::European,
            "not an European option");
        shared_ptr<PlainVanillaPayoff> payoff =
            dynamic_pointer_cast<PlainVanillaPayoff>(arguments_.payoff);
        QL_REQUIRE(process, "Black-Scholes process needed");
        // ... other requirements
        Real spot = process_->stateVariable()->value();
        // ... other needed quantities
        BlackCalculator black(payoff, forwardPrice,
                              stdDev, discount);
        results_.value = black.value();
        results_.delta = black.delta(spot);
        // ... other greeks
    }
  private:
    shared_ptr<GeneralizedBlackScholesProcess> process_;
};
```

其构造函数接受（并为自己注册）一个 Black-Scholes 随机过程，该过程包含有关底层资产的市场数据信息，包括当前值、无风险利率、股息利率和波动率。再一次，实际计算隐藏在另一个类的接口后面，即 `BlackCalculator` 类。但是，代码有足够的细节来显示一些相关的功能。

该方法首先验证一些先决条件。这可能会让人感到意外，因为计算的参数在调用 `calculate` 方法时已经被验证。然而，任何给定的引擎在执行计算之前都可以有进一步的要求。就我们的引擎而言，其中一个要求是该期权是欧式的，并且支付方式是看涨或看跌，这也意味着 `payoff` 将被向下转换到所需的类。[15]

在该方法的中间部分，引擎从传递来的参数中提取尚未处理过的任何信息。此处显示的是底层资产的当前价格，引擎所需的其他量，例如底层资产的远期价格和到期日的无风险贴现因子也被提取。[16]

最后，执行计算并将结果存储在 `results` 结构体中。这就结束了 `calculate` 方法和示例。

[15]对于共享指针，`boost::dynamic_pointer_cast` 是 `dynamic_cast` 的等价物。
[16]该引擎的完整代码可以在 QuantLib 的源代码中找到。

6. 期限结构

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

6.1 `TermStructure` 类

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

6.1.1 接口与需求

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

6.1.2 实现

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

6.2 利率期限结构

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

6.2.1 接口与实现

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

6.2.2 贴现因子、远期利率和零息利率曲线

6.2.3 示例：bootstrap 一个插值曲线

6.2.4 示例：向利率曲线添加 z-spread

6.3 其他期限结构

6.3.1 违约概率期限结构

6.3.2 通胀期限结构

6.3.3 波动率期限结构

6.3.4 股票波动率期限结构

6.3.5 利率波动率期限结构

7. 现金流与票息

7.1 CashFlow 类

7.2 票息

7.2.1 固定利率票息

7.2.2 浮动利率票息

7.2.3 示例：LIBOR 票息

7.2.4 示例：有 cap/floor 的票息

7.2.5 产生现金流序列

7.2.6 其他票息和将来的开发

7.3 现金流分析

7.3.1 示例：固息债

8. 参数模型与校准

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

8.1 `CalibrationHelper` 类

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

8.1.1 示例：Heston 模型

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

8.2 参数

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

8.3 `CalibratedModel` 类

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

8.3.1 示例：Heston 模型（续）

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9. 蒙特卡罗框架

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9.1 路径生成

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9.1.1 随机数生成

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9.1.2 随机过程

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9.1.3 随机路径生成器

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9.2 在路径上定价

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9.3 整合

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9.3.1 蒙特卡罗特性

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9.3.2 蒙特卡罗模型

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9.3.3 蒙特卡罗模拟

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

9.3.4 示例：一篮子期权

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

10. 树框架

该内容没有包含在样本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

10.1 `Lattice` 和 `DiscretizedAsset` 类

该内容没有包含在样本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

10.1.1 示例：离散债券

该内容没有包含在样本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

10.1.2 示例：离散期权

该内容没有包含在样本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

10.2 树和基于树的网格

该内容没有包含在样本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

10.2.1 `Tree` 类模板

该内容没有包含在样本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

10.2.2 二叉树和三叉树

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

10.2.3 `TreeLattice` 类模板

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

10.3 基于树的定价引擎

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

10.3.1 示例：可赎回固息债

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11. 有限差分框架

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.1 旧框架

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.1.1 微分算子

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.1.2 演化格式

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.1.3 边界条件

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.1.4 步骤条件

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.1.5 `FiniteDifferenceModel` 类

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.1.6 示例：美式期权

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.1.7 时间依赖算子

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.2 新框架

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.2.1 网格器

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.2.2 算子

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.2.3 示例：Black-Scholes 算子

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

11.2.4 初始、边界和步骤条件

11.2.5 格式和求解器

12. 完结

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

13. 附录 A：零碎的知识

一些有关使用 QuantLib 的基本问题已经在前面的章节中进行了阐述，以免积累的技术细节破坏它们的可读性（如果有的话）。正如 Douglas Adams 在《Hitchhiker》三部曲的第四本书中指出，[1]

> [An excessive amount of detail] is guff. It doesn't advance the action. It makes for nice fat books such as the American market thrives on, but it doesn't actually get you anywhere.
>
> （[过多的细节] 都是废话。它没有推进行动。它就像美国市场繁荣所依赖的那些厚厚的书一样，实际上没有带给你任何信息。）

本附录提供了一些此类问题的快速参考。它既不是详尽的，也不是系统的，[2]如果你需要这种文档，请查看 QuantLib 的网站 上提供的参考手册。

13.1 基本类型

QuantLib 接口不使用内置类型。相反，提供了许多类型定义，例如 `Time`、`Rate`、`Integer` 或 `Size`。它们都映射到基本类型（我们讨论过使用全功能类型，可能还有范围检查，但我们抛弃了这个想法）。此外，所有浮点类型都定义为 `Real`，反过来又定义为 `double`。这样就可以通过改变 `Real` 来一致地改变所有这些。

原则上，这将允许人们选择所需的精度。但对此，测试套件回答“Fiddlesticks!”，[3]因为当 `Real` 定义为 `float` 或 `long double` 时，它会显示一些失败。类型定义的价值在于使代码更加清晰，并且允许那些像我一样曾经作为物理学家的人进行量纲分析。例如，如果 `exp(r)` 或 `r+s*t` 之类的表达式在它们之前出现的是 `Rate r`、`Spread s` 和 `Time t`，则可以立即标记为可疑的。

当然，所有那些花哨的类型只是 `double` 的别名，编译器并没有真正区分它们。如果有更强的区分，那会很好。例如，人们可以重载方法来区分传递来的是价格或是波动率。

一种可能性是 `BOOST_STRONG_TYPEDEF` 宏，它是 Boost 提供的众多实用功能之一。它的用法如下，

---

[1]不，这不是一个错误。这是对五本书构成的三部曲的不准确命名。说来话长。译注：《Hitchhiker》即《银河系漫游指南》。

[2]也不是自动档的，也不是液压的，这是油脂闪电（Grease Lightning）。译注：“油脂闪电”（Grease Lightning）出自 1971 年的一部音乐剧《油脂（Grease）》，油脂闪电是剧中一部汽车的名字，也是剧中一首歌曲的名字。“也不是系统的，也不是自动档的，也不是液压的，这是油脂闪电”是在化用剧中的歌词“Why this car is automatic. It's systematic. It's hydromatic. Why it's grease lightning”。

[3]译注：Fiddlesticks，意为我反对，我不同意。

```
BOOST_STRONG_TYPEDEF(double, Time)
BOOST_STRONG_TYPEDEF(double, Rate)
```

并创建一个适当的类，提供与底层类型之间的相互转换。这将允许重载方法，但缺点是并非所有转换都是显式的。这会破坏向后兼容性并使事情变得尴尬。[4]

此外，宏定义的类会使所有运算符重载：你可以愉快地为利率添加时间，即使它没有意义（是的，再次进行量纲分析）。如果类型系统阻止这种编译，那将是很好的，但是对下面的情况仍然是允许的，例如将利差添加到利率产生另一利率，或者将利率乘以时间产生纯粹的数字。

如何以通用的方式做到这一点，并且理想情况下没有运行时成本，这由 Barton 和 Nackman (Barton and Nackman, 1995) 首先提出，他们想法的一个变体在 `Boost::Units` 库中实现，而一个更简单的版本是由我在进行物理学研究中实现的。[5]但是，这可能太过火了，我们不必处理长度、质量、时间等等所有可能的组合。

未来 QuantLib 的理想折衷方案可能是实现包装器类（例如 Boost 强类型定义），并明确定义允许哪些类型的运算符。像往常一样，我们不是第一个遇到这个问题的人：这个想法已经流传了一段时间，一个提案被提出 (Brown, 2013) 以添加成为下一个版本 C++ 的新功能，名为 `opaque typedef`，可以更容易地定义这种类型。

最后一点：在这些类型中，至少有一个不是自己确定的（如 `Rate` 或 `Time`），而是取决于其他类型。价格的波动率和利率的波动率具有不同的量纲，因此应该具有不同的类型。简而言之，`Volatility` 应该是模板类型。

13.2 日期计算

日期计算是量化金融的基本工具之一。可以预料，QuantLib 为此任务提供了许多函数，我将在以下小节中简要介绍其中的一些内容。

13.2.1 日期与周期

`Date` 类的实例表示特定日期，例如 2014 年 11 月 15 日。此类提供了许多方法来检索基本信息，例如工作日、月内年内某天；静态信息，例如允许的最小和最大日期（当下，分别为 1901 年 1 月 1 日和 2199 年 12 月 31 日），或某一年是否为闰年；或其他信息，例如日期的 Excel 兼容序列数字，或者给定日期是否是该月的最后日期。可用方法及其接口的完整列表记录在参考手册中。不包括时间信息（虽然我们一直在谈论这个）。

[4]例如，一个简单的表达式 `Time t = 2.0;` 将不能编译。你要写 `f(Time(1.5))` 而不是直接写 `f(1.5)`，即使 `f` 未被重载。
[5]我不会在这里解释它，但它值得一读，它酷的要命。

利用 C++ 的特性，Date 类还会重载许多运算符，以便日期的代数计算可以自然地编写。例如，可以编写诸如 ++d 之类的表达式，它将日期 d 推后一天；d + 2，产生给定日期后两天的日期；d2 - d1，计算两个日期之间的天数；d - 3 * Weeks，产生给定日期前三周的日期（顺便提一下 TimeUnit 枚举的成员，其他成员是 Days、Months 和 Years）；或者 d1 < d2，如果第一个日期早于第二个日期，则结果为 true。在 Date 类中实现的代数计算依照日历日，既不考虑银行假日也不考虑工作日约定。

Period 类通过存储 TimeUnit 和整数来模拟时间长度，例如两天、三周或五年。它提供有限的代数计算和偏序关系。以非数学化的方式讲，这意味着两个 Period 实例之间可能可以比较，也可能不能比较。很明显，比如 11 个月不到一年，但是如果不知道哪两个月，就不可能确定 60 天是否多于或少于两个月。无法确定比较时，抛出异常。

当然，即使比较是显而易见的，我们还是设法隐藏了一些惊喜。例如，比较

```
Period(7, Days) == Period(1, Weeks)
```

返回 true。看来是对的吧？坚持这个想法。

13.2.2 日历

假日和工作日是 Calendar 类的领域。存在若干派生类，它们定义了许多市场的假日。基类定义用于确定日期是否与假日或工作日相对应的简单方法，以及用于执行诸如将假日调整到最近工作日之类任务的更复杂方法（其中“最近”根据的是 BusinessDayConvention 枚举所列出的若干工作日约定），或在给定期间或工作日数内推进日期。

看看日历的行为如何根据其描述的市场而变化可能会很有趣。一种方法是在 Calendar 实例中存储相应市场的假日列表。但是，为了可维护性，我们想要编写实际的日历规则（例如“11 月的第四个星期四”或“每年的 12 月 25 日”），而不是列举几个世纪的结果日期。另一个显而易见的方法是使用多态和模板方法模式。派生类日历将覆盖 isBusinessDay 方法，从中可以实现所有其他方法。这很好，但它的缺点是日历需要在 shared_ptr 中传递和存储。但是，这个类在概念上很简单，而且经常使用，我们希望用户实例化它并更容易地传递它，也就是说，没有动态分配的额外冗余。

最终的解决方案显示在下面的代码中。它是 *pimpl* 惯用法的变体，也让人想起策略模式或桥接模式。这时候，技术潮人们可能也称之为类型擦除。

Calendar 类的框架。

```
class Calendar {
  protected:
    class Impl {
      public:
        virtual ~Impl() {}
        virtual bool isBusinessDay(const Date&) const = 0;
    };
    boost::shared_ptr<Impl> impl_;
  public:
    bool isBusinessDay(const Date& d) const {
        return impl_->isBusinessDay(d);
    }
    bool isHoliday(const Date& d) const {
        return !isBusinessDay(d);
    }
    Date adjust(const Date& d,
                BusinessDayConvention c = Following) const {
        // uses isBusinessDay() plus some logic
    }
    Date advance(const Date& d,
                 const Period& period,
                 BusinessDayConvention c = Following,
                 bool endOfMonth = false) const {
        // uses isBusinessDay() and possibly adjust()
    }
    // more methods
};
```

简而言之：Calendar 声明了一个多态内部类 Impl 实现工作日规则，并存储指向其中一个实例的指针。Calendar 类的非虚方法 isBusinessDay 委托到 Calendar::Impl 中的相应方法，在某种程度上的模板方法模式之后，Calendar 其他方法也是非虚的，并且根据 isBusinessDay（直接或间接地）实现。[6]

派生的日历类可以通过定义从 Calendar::Impl 派生的内部类来提供专门的行为。它们的构造函数将创建一个指向 Impl 实例的共享指针，并将其存储在基类的数据成员 impl_ 中。任何需要存储 Calendar 实例的类都可以安全地拷贝生成的日历。即使在切片时，由于包含指向多态 Impl 类的指针，它将保持正确的行为。最后，我们可以注意到，同一派生日历类的实例可以共享相同的 Impl 实例。这可以看作是享元模式的一个实现，为一个看似简单的类居然动用了大约两个半设计模式。

关于 Calendar 实现的讨论已经足够了，现在回到行为上。这是我在上一节中提到的惊喜。还记得吗？Period(1, Weeks) 等于 Period(7, Days)。除了日历的 advance 方法之外，7 天意味着 7 个工作日。因此，我们有一种情况，其中两个周期 p1 和 p2 相等（即，p1 == p2 返回 true），但是 calendar.advance(p1) 与 calendar.advance(p2) 不同。耶，我们就这样。

我不确定我在这里有一个好的解决方案。如果我们想要向后兼容，那么 Days 的当前用法必须以相同的方式继续。所以说，开始将 calendar.advance(7, Days) 解释为 7 个日历日是不

[6] 相同的技术应用于许多其他类中，例如下一节中的 DayCounter 或者第 5 章 中的 Parameter。

可能的。一种方法可能是保持当前的情况，引入两个新的枚举 BusinessDays 和 CalendarDays 来消除歧义，并弃用 Days。另一种是通过规定 7 天的时间实际上不等于一周来消除不一致性。我对这个方案并不是很满意。

如果我们放弃向后兼容性（传说中的 QuantLib 2.0），那么有更多的可能性。一种是始终使用 Days 作为日历日，并将 BusinessDays 添加为不同的枚举。另一种（我越想就越喜欢它），就像我想的那样，总是使用 Days 作为日历日，并将一个特定的方法 advanceBusinessDays 添加到 Calendar（或者可能是一个重载 advance(n, BusinessDays)），BusinessDays 是一个单独类的实例）。但是，这意味着 3 个工作日不会是一个周期。

正如我所说，没有明显的解决方案。如果你有任何其他建议，我会洗耳恭听。

13.2.3 天数计算规则

DayCounter 类提供了计算两个日期之间距离的方法，根据不同的约定，结果可以是天数，也可以是一年的分数。存在诸如 Actual360 或 Thirty360 之类的派生类，它们使用与 Calendar 类相同的技术（在上一节中描述过）实现多态行为。

不幸的是，接口有一点粗糙。yearFraction 方法并没有仅接受两个日期，而是声明为，

```
Time yearFraction(const Date&,
                  const Date&,
                  const Date& refPeriodStart = Date(),
                  const Date& refPeriodEnd = Date()) const;
```

一个特定的天数计算规则（即 ISMA actual/actual 约定）需要两个可选日期，需要在两个输入日期之外指定基准周期。为了保持一个通用的接口，我们不得不将两个额外的日期添加到方法的签名中，用于全部天数计算器（其中大部分都很乐意忽略它们）。这不是这一个天数计算器搞的唯一恶作剧，你会在下一节看到另一个。

13.2.4 时间表

Schedule 类显示在下面的代码中，用于生成付息日期序列。

`Schedule` 类的接口。

```
class Schedule {
  public:
    Schedule(const Date& effectiveDate,
             const Date& terminationDate,
             const Period& tenor,
             const Calendar& calendar,
             BusinessDayConvention convention,
             BusinessDayConvention terminationDateConvention,
             DateGeneration::Rule rule,
             bool endOfMonth,
             const Date& firstDate = Date(),
             const Date& nextToLastDate = Date());
    Schedule(const std::vector<Date>&,
             const Calendar& calendar = NullCalendar(),
             BusinessDayConvention convention = Unadjusted);
    Size size() const;
    bool empty() const;
    const Date& operator[](Size i) const;
    const Date& at(Size i) const;
    const_iterator begin() const;
    const_iterator end() const;
    const Calendar& calendar() const;
    const Period& tenor() const;
    bool isRegular(Size i) const;
    Date previousDate(const Date& refDate) const;
    Date nextDate(const Date& refDate) const;
    // other inspectors and utilities
};
```

按照实践惯例和 ISDA 约定，这个类必须接受很多参数。你可以将它们视为其构造函数的参数列表。（哦，我想你会原谅我不去解释所有这些参数的，我相信你能猜出它们的意思。）它们可能太多了，这就是 QuantLib 使用命名参数惯用法（已在第 4 章 中描述）提供一个不太笨重的工厂类的原因。在其帮助下，可以将时间表实例化为

```
Schedule s = MakeSchedule()
                 .from(startDate)
                 .to(endDate)
                 .withFrequency(Semiannual)
                 .withCalendar(TARGET())
                 .withNextToLastDate(stubDate)
                 .backwards();
```

一方面，其他方法包括存储数据的检查器；另一方面，包括给类提供一个序列接口的方法，例如 `size`、`operator[]`、`begin` 和 `end`。

`Schedule` 类的第二个构造函数，接受预先计算的日期向量。但它只是稍微起点儿作用：生成的 `Schedule` 实例根本没有它们的检查器应该返回的一些数据，因此这些方法在调用时会抛出异常。其中有 `tenor` 和 `isRegular`，我需要花一些文字解释。

首先，`isRegular(i)` 不是指第 `i` 个日期，而是指第 `i` 个区间。也就是说，第 `i` 和 `i+1` 日期之间的区间。这么说，“regular（规则的）”是什么意思？当基于期限构建时间表时，大多数区间对应于传递来的期限（因此是规则的），但是第一个和最后一个区间可能更短或更长，这取决于我们是否传递明确的第一个或倒数第二个日期。我们可以这样做，例如，当我们想要指定一个短的首次付息时。

如果我们使用预先计算的日期集来构建时间表，我们没有期限信息，我们无法判断给定的间隔是否是规则的。[7]坏消息是，这使得无法使用该时间表建立债券票息序列。如果我们将它传递给固息债的构造函数，我们将得到一个异常。为什么，哦，为什么债券在构建票息时需要缺失的信息？因为债券的天数计算规则可能是 ISMA actual/actual，这需要一个基准周期，并且为了计算基准周期，我们需要知道付息期限。

幸运的是，解决这个问题应该不难。一种方法是检查天数计算规则，并且只在需要时尝试计算基准周期。这样构造函数仍然会为 ISMA actual/actual 引发异常，但是对于所有其他约定都会成功。另一种方法可能是将期限和规则性信息添加到时间表中，以便相应的方法可以工作。但我不确定这有多大意义。

13.3 金融相关类

鉴于我们的领域，只能预期许多类直接模拟金融概念。本节描述了一些这样的类。

13.3.1 市场报价

至少有两种可能性对报价值进行模拟。一种是将报价模拟为一系列静态值，每个值都有一个相关的时间戳，当前值是最新的；另一种方法是将当前值模拟为动态变化的报价值。

两种观点都很有用。事实上，两者都在 QuantLib 中实现。第一个模型对应于 `TimeSeries` 类，我不在这里详细描述。它基本上是日期和值之间的映射，具有在给定日期检索值，并迭代现有值的方法，并且它从未真正用在 QuantLib 的其他部分。第二个产生了 `Quote` 类，如下面的代码所示。

[7]好吧，我们可以使用启发式算法，但它可能会很快变得很丑陋。

Quote 类的接口。

```
class Quote : public virtual Observable {
  public:
    virtual ~Quote() {}
    virtual Real value() const = 0;
    virtual bool isValid() const = 0;
};
```

它的接口足够简单。该类继承自 Observable 类，因此它可以在其值发生变化时通知其依赖对象。它声明了 isValid 方法，以判断报价是否包含有效值（相应的，没有值或者可能是过期值），以及 value 方法，它返回当前值。

这两种方法足以提供所需的行为。行为或值取决于市场价值的任何其他对象（例如，第 3 章 的 bootstrap 的辅助类），可以将句柄存储到相应的报价，并作为观察者注册它们。从那时起，它将能够随时访问当前值。

QuantLib 定义了许多报价类，即 Quote 接口的实现。其中一些的返回值衍生自其他类，例如 ImpliedStdDevQuote 将期权价格转换为隐含波动率。其他的适配其他对象，ForwardValueQuote 在底层期限结构发生变化时返回确定的远期指数，而 LastFixingQuote 返回时间序列中的最新值。

目前，只有一种实现是外部值的真正来源，这就是 SimpleQuote 类，如下面的代码所示。

SimpleQuote 类的实现。

```
class SimpleQuote : public Quote {
  public:
    SimpleQuote(Real value = Null<Real>())
        : value_(value) {
    }
    Real value() const {
        QL_REQUIRE(isValid(), "invalid SimpleQuote");
        return value_;
    }
    bool isValid() const {
        return value_ != Null<Real>();
    }
    Real setValue(Real value) {
        Real diff = value - value_;
        if (diff != 0.0) {
            value_ = value;
            notifyObservers();
        }
        return diff;
    }
  private:
    Real value_;
};
```

它在某种意义上很简单，它没有实现任何特定的数据提供接口：通过调用适当的方法手动设置新值。最新值（可能等于 `Null<Real>()`，以表示无值）存储在数据成员中。`Quote` 接口是通过让 `value` 方法返回存储的值来实现的，而 `isValid` 方法则检查它是否为空值。用于提供新值的方法是 `setValue`，它接受新值，通知其观察者是否与最新存储的值不同，并返回旧值和新值之间的增量。[8]

我将以一些简短的注解结束本小节。首先，报价值的类型被约束为 `Real`。到目前为止，这并不是一个限制，而且现在将 `Quote` 定义为类模板已经太晚了，所以这不可能改变。

第二个原因，最初的想法是 `Quote` 接口将充当实际数据提供的适配器，不同的实现调用不同的 API，并允许 QuantLib 以统一的方式使用它们。然而，到目前为止，没有人提供这样的实现。我们得到的比较接近的实现是使用 Excel 作为数据源，并将其值设置为 `SimpleQuote` 的实例。

最后一点（有点长），将来可能会修改 `SimpleQuote` 的接口以允许更高级的用途。在为一组相关报价设置新值时（例如，用于 bootstrap 曲线的利率报价），最好只在设置所有值后触发单个通知，而不是让每个报价在更新时发送通知。这种行为会更快，因为通知链变得非常紧凑，并且更安全，因为在更新报价的子集之后没有观察者会冒重新计算的风险。这个更改（即 `setValue` 中的额外参数 `silent` 等于 `true` 时将通知停止）已经在 QuantLib 的分支中实现，并且也可以添加到 QuantLib 中。

13.3.2 利率

`InterestRate` 类（显示在下面的代码中）封装了一般的利率计算。此类的实例是根据利率、天数计算规则、付息规则和付息频率构建的（但请注意，无论频率如何，利率的值始终是年化的）。这允许人们指定诸如“5%，actual/365，连续复利”或“2.5%，actual/365，半年复利”的利率。可以看出，并不总是需要频率。我稍后会回到这里。

`InterestRate` 类的概要。

```
enum Compounding {
    Simple,         // 1+rT
    Compounded,     // (1+r)^T
    Continuous,     // e^{rT}
    SimpleThenCompounded
};
class InterestRate {
  public:
    InterestRate(Rate r,
                 const DayCounter&,
                 Compounding,
                 Frequency);
    // inspectors
    Rate rate() const;
```

[8]选择返回最新增量有点儿不同寻常，C 和 C++ 中的习惯选择是返回旧的值。

```
    const DayCounter& dayCounter();
    Compounding compounding() const;
    Frequency frequency() const;
    // automatic conversion
    operator Rate() const;
    // implied discount factor and compounding after a given time
    // (or between two given dates)
    DiscountFactor discountFactor(Time t) const;
    DiscountFactor discountFactor(const Date& d1,
                                  const Date& d2) const;
    Real compoundFactor(Time t) const;
    Real compoundFactor(const Date& d1,
                        const Date& d2) const;
    // other calculations
    static InterestRate impliedRate(Real compound,
                                    const DayCounter&,
                                    Compounding,
                                    Frequency,
                                    Time t);
    // same with dates
    InterestRate
    equivalentRate(Compounding,
                   Frequency,
                   Time t) const;
    // same with dates
};
```

除了显而易见的检查器外，该类还提供了许多方法。一个是 `Rate`（即 `double`）的转换运算符。在事后的想法中，这是有风险的，因为转换后的值会丢失任何天数和付息信息。例如，在预期连续复利的情况下，这可能允许一个单利混进来。为了向后兼容，在首次引入 `InterestRate` 类时添加了转换，它可能会在 QuantLib 的未来版本中删除，具体取决于我们想要强制用户使用的安全级别。[9]

其他方法完成一组基本计算。`compoundFactor` 根据给定的利率返回时间 t（或等效地，在两个日期 d_1 和 d_2 之间）上的单位付息金额；`discountFactor` 方法返回两个日期之间或一段时间上的贴现因子，即付息因子的倒数；给定一组约定和时间，`impliedRate` 方法返回一个利率，产生给定的付息因子；以及 `equivalentRate` 方法将利率转换为具有不同约定下的等价利率（即导致相同付息金额的利率）。

与 `InterestRate` 构造函数一样，这些方法中的一些接受付息频率。正如我所提到的，这并不总是有意义的。事实上，`Frequency` 枚举有一个 `NoFrequency` 项来涵盖这种情况。

显然，这有点儿坏味道。理想情况下，付息频率应仅与那些需要它的付息约定相关联，而完全省略那些不需要它的（例如 `Simple` 和 `Continuous`）。如果 C++ 支持它，我们会写类似下面的东西

[9] 核心开发者们对安全性有不同的看法，从“照顾用户，不要让他伤害自己”到“给他自己的那部分遗产，拍拍他的背，并让他自己去闯。”

```
enum Compounding {
    Simple,
    Compounded(Frequency),
    Continuous,
    SimpleThenCompounded(Frequency)
};
```

这与函数式语言中的代数数据类型或 Scala 中的 case 类相似。[10]但不幸的是，这不是一个选项。要拥有这样的东西，我们必须选择完整功能的策略模式，并将 Compounding 转换为类架构。对于这个类的需求来说，这可能是过度的，所以我们要保留枚举和坏味道。

13.3.3 指数

像其他类一样，例如 Instrument 和 TermStructure，Index 类是一个非常大的伞形结构：它涵盖了诸如利率指数、通胀指数、股票指数等概念——你大体明白了吧。

无需多言，建模实体的多样性足以使 Index 类具有很少的接口来调用它自己。如下面的代码所示，其所有方法都与确定指数有关。

Index 类的接口。

```
class Index : public Observable {
  public:
    virtual ~Index() {}
    virtual std::string name() const = 0;
    virtual Calendar fixingCalendar() const = 0;
    virtual bool isValidFixingDate(const Date& fixingDate) const = 0;
    virtual Real fixing(const Date& fixingDate,
                        bool forecastTodaysFixing = false) const = 0;
    virtual void addFixing(const Date& fixingDate,
                           Real fixing,
                           bool forceOverwrite = false);
    void clearFixings();
};
```

isValidFixingDate 方法告诉我们在给定日期是否（或将要）进行确定；fixingCalendar 方法返回用于确定有效日期的日历；还有 fixing 方法检索过去日期的确定，或预测未来日期的确定。其余的方法专门处理过去的确定：name 方法，它返回一个必须对每个指数唯一的标识符，用于索引（双关语，我不是故意的[11]）到存储确定的映射中；addFixing 方法存储一个确定（或许多，在此处未显示的其他重载方法中）；以及 clearFixing 方法清除给定指数的所有存储的确定。

为什么是映射，以及在 Index 类中什么位置？好吧，过去的确定应该被共享，而不是针对具体的实例，我们从这个需求出发。如果存储一个 6 个月的 Euribor 确定，我们希望该确定对

[10]两者都支持对象上的模式匹配，这类似于更简洁的增强版 switch。如果你有时间，去看看吧。
[11]译注：原文中“索引”和“指数”是同一个单词——index。

于相同指数的所有实例都可见，[12]而不仅仅是我们调用 addFixing 方法的特定实例。这是通过在幕后定义和使用 IndexManager 单体来完成的。坏味道？当然，就像所有单身汉[13]一样。另一种方法是在每个派生类中定义静态类变量来存储确定。但是这会迫使我们在每个派生类中拷贝它们而没有真正的改进（它将与单体一样抵触并发）。无论如何，这是我们在下一个 QuantLib 大型修订版中需要重新考虑的事情之一，[14]我们必须解决并发问题。

由于返回的指数确定可能会改变（因为它们的预测值依赖于其他不同的对象，或者因为添加了新的可用确定并替换了预测），Index 类继承自 Observable，以便金融工具可以注册其实例，并收到有关此类更改的通知。

在这个时候，Index 并没有继承自 Observer，尽管它的派生类确实如此（毫不奇怪，因为预测确定几乎总是取决于一些可观察的市场报价）。这不是一个明确的设计选择，而是代码演变的产物，并可能在将来的版本中发生变化。但是，即使我们让 Index 继承 Observer，我们仍然会被迫在派生类中有一些代码重复，原因可能值得详细描述。

我已经提到过，确定可以改变，原因有两个。一个是指数依赖于其他可观察对象来预测其确定。在这种情况下，它只是注册它们（这是在每个派生类中完成的，因为每个类都有不同的可观察对象）。另一个原因是可能会提供新的确定，而且处理起来更棘手。通过在特定指数实例上调用 addFixing 来存储确定，因此似乎不需要外部通知，并且指数可以只调用 notifyObservers 方法来通知其观察者，但事实并非如此。正如我所说，这些确定是共享的，如果我们存储今天 3 个月的 Euribor 确定，它将可用于所有这类指数的实例，因此我们希望所有这些实例都知道这一变化。此外，金融工具和曲线可能已经注册了 Index 实例中的任何一个，因此所有这些实例都必须依次发送通知。

解决方案是让相同指数的所有实例通过共享对象进行通信。也就是说，我们使用了相同的 IndexManager 单体来存储所有指数确定。正如我所说，IndexManager 将唯一指数标记映射到一组确定。另外，通过创建 ObservableValue 类的一组实例，它提供了在为特定标记添加一个或多个确定时注册和接收通知的方法（此类将在本附录后面描述，这里不需要详述）。

所有组件现已到位。在构造时，任何 Index 实例都会向 IndexManager 请求其 name 方法，返回标记相对应的共享观察对象。例如，当我们在某个特定的 6 个月 Euribor 指数调用 addFixings 时，确定将被存储到 IndexManager 中。观察者将向当时所有 6 个月 Euribor 指数发送通知，一切都是那么美好。

然而，C++ 仍然向我们的齿轮中抛出一个小扳手。鉴于上述情况，在 Index 构造函数中调用下面的代码

```
registerWith(IndexManager::instance().notifier(name()));
```

[12]请注意，“相同指数的所有实例”在这里指的是相同特定指数的实例，而不是相同类（可能将分为不同的指数）的。例如，USDLibor(3*Months) 和 USDLibor(6*Months) 不是同一指数的实例；两个不同的 USDLibor(3*Months) 是。

[13]译注：双关语，单体和单身汉的词根相同。

[14]如果过去这些年有任何迹象，我们预期可能会在 2025 年左右。不，只是开个玩笑，也许吧。

显得很诱人。但是，它不会起作用。因为在基类的构造函数中，对虚方法 name 的调用不会是多态的。[15]从这里可以看出我之前提到的几段代码重复，为了起作用，必须将上述方法调用添加到每个派生指数类的构造函数中。Index 类本身没有构造函数（除了编译器提供的默认构造函数）。

作为从 Index 派生的具体类的示例，下面的代码显示出 InterestRateIndex 类的框架。

InterestRateIndex 类的框架。

```
class InterestRateIndex : public Index, public Observer {
  public:
    InterestRateIndex(const std::string& familyName,
                      const Period& tenor,
                      Natural settlementDays,
                      const Currency& currency,
                      const Calendar& fixingCalendar,
                      const DayCounter& dayCounter)
        : familyName_(familyName), tenor_(tenor), ... {
        registerWith(Settings::instance().evaluationDate());
        registerWith(IndexManager::instance().notifier(name()));
    }
    std::string name() const;
    Calendar fixingCalendar() const;
    bool isValidFixingDate(const Date& fixingDate) const {
        return fixingCalendar().isBusinessDay(fixingDate);
    }
    Rate fixing(const Date& fixingDate,
                bool forecastTodaysFixing = false) const;
    void update() { notifyObservers(); }
    std::string familyName() const;
    Period tenor() const;
    // other inspectors
    Date fixingDate(const Date& valueDate) const;
    virtual Date valueDate(const Date& fixingDate) const;
    virtual Date maturityDate(const Date& valueDate) const = 0;
  protected:
    virtual Rate forecastFixing(const Date& fixingDate) const = 0;
    std::string familyName_;
    Period tenor_;
    Natural fixingDays_;
    Calendar fixingCalendar_;
    Currency currency_;
    DayCounter dayCounter_;
};
std::string InterestRateIndex::name() const {
    std::ostringstream out;
    out << familyName_;
    if (tenor_ == 1 * Days) {
        if (fixingDays_ == 0)
            out << "ON";
        else if (fixingDays_ == 1)
```

[15]也许你不熟悉 C++ 的黑暗角落：当执行基类的构造函数时，尚未构建派生类中定义的任何数据成员。由于任何特定于派生类的行为都可能依赖于这些尚未存在的数据，因此在基类构造函数体中调用任何虚方法时 C++ 会摒弃多态并使用基类的实现。

```
            out << "TN";
        else if (fixingDays_ == 2)
            out << "SN";
        else
            out << io::short_period(tenor_);
    } else {
        out << io::short_period(tenor_);
    }
    out << " " << dayCounter_.name();
    return out.str();
}

Rate InterestRateIndex::fixing(const Date& d,
                               bool forecastTodaysFixing) const {
    QL_REQUIRE(isValidFixingDate(d), ...);
    Date today = Settings::instance().evaluationDate();
    if (d < today) {
        Rate pastFixing = IndexManager::instance().getHistory(name())[d];
        QL_REQUIRE(pastFixing != Null<Real>(), ...);
        return pastFixing;
    }
    if (d == today && !forecastTodaysFixing) {
        Rate pastFixing = ...;
        if (pastFixing != Null<Real>())
            return pastFixing;
    }
    return forecastFixing(d);
}

Date InterestRateIndex::valueDate(const Date& d) const {
    QL_REQUIRE(isValidFixingDate(d)...);
    return fixingCalendar().advance(d, fixingDays_, Days);
}
```

正如你所料，除了从 Index 继承的行为之外，此类还定义了大量特定行为。首先，它也继承自 Observer，因为 Index 没有。InterestRateIndex 构造函数接受指定指数所需的数据：一个家族名，如 Euribor，对于同一家族的不同指数是共同的，例如 3 个月和 6 个月的 Euribor；指定家族中特定指数的期限；以及其他信息，例如结算日数、指数基础货币、确定日历和用于计息的天数计算规则。

当然，传递来的数据被拷贝到相应的数据成员中。然后指数注册了几个可观察对象。第一个是全局估值日期，这是必要的，因为正如我稍后将解释的那样，当一个实例被要求今天进行确定时，类中会有一些特定于日期的行为被触发。第二个可观察对象包含在 IndexManager 中，并在存储新的确定时提供通知。我们可以在这里识别这个可观察对象：InterestRateIndex 类具有确定指数所需的所有信息，因此它可以实现 name 方法并调用它。但是，这也意味着从 InterestRateIndex 派生的类不能覆盖 name 方法，由于重写的方法不会在此构造函数的主体中调用（如前所述），因此它们将注册错误的通知器。不幸的是，这不能在 C++ 中强制执行，C++ 没有像 Java 中的 final 或者 C# 中的 sealed 一样的关键字。但另一种方法是要求所有派生自 InterestRateIndex 的类都注册了 IndexManager，这同样不可强制执行，可能更容易出错，而且肯定不太方便。

InterestRateIndex 中定义的其他方法有不同的用途。一些实现了 Index 和 Observer 所要求

接口。最简单的是 update，它只是传递任何通知；fixedCalendar，它返回存储的日历实例的副本；以及 isValidFixingDate，它根据确定日历检查日期。

name 方法有点复杂。它将家族名、期限的简短表示，以及天数计算规则拼接在一起，以获得指数的名称，例如“Euribor 6M Act / 360”或“USD Libor 3M Act / 360”；检测特殊的期限，例如隔夜、明日对后日（tomorrow-next）和即期对次日（spot-next），以便使用相应的首字母缩略词。

fixing 方法包含最多的逻辑。首先，检查所要求的确定日期，如果不对其进行确定则引发异常。然后，根据今天的日期检查确定日期。如果确定日期已过去，它必须存储在 IndexManager 单体中。如果没有，会引发异常，因为我们无法预测过去的确定。如果今天要求确定，指数首先尝试在 IndexManager 中寻找确定，如果找到则返回它，否则，确定尚未提供。在这种情况下，指数预测确定的值，以及未来的确定日期，这是通过调用 forecastFixing 方法完成的，该方法在此类中声明为纯虚的，并在派生类中实现。正如我所提到的，fixing 方法中的逻辑也是为什么指数在估值日期中注册的原因，指数的行为取决于今天日期的值，因此需要在更改时通知它。

最后，InterestRateIndex 类定义了其他不从 Index 继承的方法。它们中的大多数是检查器，它们返回存储的数据，如家族名或期限，其他一些处理日期计算。valueDate 方法接受确定日期，并返回依赖此利率的金融工具的起始日期（例如，对于大多数货币而言，基于 LIBOR 的存款，在确定日期后两个工作日开始）。maturityDate 方法接受值日期，并返回底层金融工具的到期日（例如，存款的到期日）。以及 fixingDate 方法，该方法是 valueDate 的逆，接受起始日期并返回相应的确定日期。其中一些方法是虚的，因此可以覆盖它们的行为。例如，虽然 valueDate 的默认行为是在给定日历上延后给定的确定天数，但 LIBOR 指数首先要求在伦敦日历上延后，然后在对应于指数货币的日历上调整结果日期。出于某种原因，fixDate 不是虚的。这可能是一个疏忽，应该在未来的版本中修复。

> **题外话：多大程度的泛化？**
>
> InterestRateIndex 类的一些方法在设计时显然主要考虑了 LIBOR，因为这是在 QuantLib 中实现的那种类型的第一个指数。一方面，这使得该类不像人们想要的那样通用：例如，如果我们决定将 5-10 年的互换利率利差本身视为利率指数，我们很难将它适配到基类的接口和它的单个 tenor 方法。但另一方面，在没有几个类作为例子要实现的情况下泛化接口很少是明智的。两个指数之间的利差（只是个利差，而不是指数）可能不是这样一个类。

13.3.4 行权与支付

我将会用一些金融工具中使用的领域相关类来结束本节。

首先是 Exercise 类，如下面的代码所示。

`Exercise` 类及其派生类的接口。

```
class Exercise {
  public:
    enum Type {
        American,
        Bermudan,
        European
    };
    explicit Exercise(Type type);
    virtual ~Exercise();
    Type type() const;
    Date date(Size index) const;
    const std::vector<Date>& dates() const;
    Date lastDate() const;
  protected:
    std::vector<Date> dates_;
    Type type_;
};
class EarlyExercise : public Exercise {
  public:
    EarlyExercise(Type type,
                  bool payoffAtExpiry = false);
    bool payoffAtExpiry() const;
};
class AmericanExercise : public EarlyExercise {
  public:
    AmericanExercise(const Date& earliestDate,
                     const Date& latestDate,
                     bool payoffAtExpiry = false);
};
class BermudanExercise : public EarlyExercise {
  public:
    BermudanExercise(const std::vector<Date>& dates,
                     bool payoffAtExpiry = false);
};
class EuropeanExercise : public Exercise {
  public:
    EuropeanExercise(const Date& date);
};
```

正如你所预期的那样，基类声明了检索行权日期上信息的方法。实际上有不少方法。有一个 `dates` 方法返回一组行权日期，一个 `date` 方法返回特定指数上的一个行权日期，一个便捷方法 `lastDate`，正如你可能已经猜到的那样，返回最后一个行权日期。所以有一些冗余。[16]另外，还有一种 `type` 方法让我在再次看代码时扎耳挠腮。

`type` 方法返回行权的类型，从内部枚举 `Exercise::Type` 中声明的集合（欧式、百慕大式或美式）中选择它的值。对它自己而言这并不费解，但它与我们接下来做的有些不同，即使用继承来声明 `AmericanExercise`、`BermudanExercise` 和 `EuropeanExercise` 类。一方面，在

[16]封装的爱好者可能更喜欢接受一个索引的版本，而不是返回一个向量，因为后者揭示了关于类的内部存储的必要性。

Exercise 基类中使用虚析构函数似乎表明，如果想要定义新类型的行权方式，继承是可行的方法。另一方面，在基类中枚举行权的类型类似乎违背了这种扩展，因为继承一个新的行权类也需要在枚举中添加一个新的情况。对于继承，人们还可以争辩说，围绕 QuantLib 的既定惯用法是传递给 Exercise 类的智能指针，并且对于继承，该类没有定义除析构函数之外的任何虚方法，并且任何派生类实例的行为仅由存储在基类中的数据成员的值赋予。简而言之：似乎我们在写这段代码时，我们比我现在更加困惑。

如果我现在编写它，我可能会保留枚举，并使其成为一个具体的类：派生类可以创建可安全切片成为 Exercise 实例的对象，当传递它们时，或者它们可以转换为函数直接返回 Exercise 实例。尽管这可能会使面向对象的纯粹主义者感到厌烦，但代码中有许多地方需要检查行权的类型，并且与使用类型转换或某种访问者模式相比，枚举可能是一种实用的选择。派生类中缺少特定行为对我来说似乎是另一个暗示。

写到这里时，我想到一个行权可能是一个 Event，如第 4 章 所述。但是，这并不总是与 Exercise 类模拟的相匹配。在 European 行权的情况下，我们也可以将其建模为 Event 实例；在 Bermudan 行权的情况下，Exercise 实例可能对应于一组 Event 实例；在 American 行权的情况下，我们在这里建模的是行权范围。事实上，接口的含义在这种情况下也会发生变化，因为 dates 方法不再返回集合所有可能的行权日期，但只是范围中的第一个和最后一个日期。正如经常发生的那样，看起来很明显的小事情在近距离观察时很难建模。

紧接着是 Payoff 类，以及它的一些派生类，在下面的代码中显示。

Payoff 类以及一些派生类的接口。

```
class Payoff : std::unary_function<Real, Real> {
  public:
    virtual ~Payoff() {}
    virtual std::string name() const = 0;
    virtual std::string description() const = 0;
    virtual Real operator()(Real price) const = 0;
    virtual void accept(AcyclicVisitor&);
};
class TypePayoff : public Payoff {
  public:
    Option::Type optionType() const;
  protected:
    TypePayoff(Option::Type type);
};
class FloatingTypePayoff : public TypePayoff {
  public:
    FloatingTypePayoff(Option::Type type);
    Real operator()(Real price) const;
    // more Payoff interface
};
class StrikedTypePayoff : public TypePayoff {
  public:
    Real strike() const;
    // more Payoff interface
```

```
  protected:
    StrikedTypePayoff(Option::Type type,
                      Real strike);
};
class PlainVanillaPayoff : public StrikedTypePayoff {
  public:
    PlainVanillaPayoff(Option::Type type,
                       Real strike);
    Real operator()(Real price) const;
    // more Payoff interface
};
```

它的接口包括一个 `operator()`，返回给定底层资产值的支付值，一个支持访问者模式的 `accept` 方法，以及一些检查器（`name` 和 `description` 方法）用于报告，可能是太多了。

事后看来，我们试图在掌握足够的用例之前对类进行建模。不幸的是，导致接口被卡住。最大的问题是 `operator()` 对单个底层证券的依赖性，它排除了基于多个底层资产价值的情况。[17]另一个问题是过度依赖继承。例如，我们有一个 `TypePayoff` 类，它添加了一个类型（`Call` 或 `Put`，这可能也是限制性的）和相应的检查器。添加敲定价的 `StrikedTypePayoff`。以及，最后的 `PlainVanillaPayoff`，它模拟一个简单的看涨或看跌支付，并最终从 `Payoff` 类中删除三层的继承：可能太多了，考虑到它用于实现一个教科书期权，并将被看作人们接触 QuantLib 的开始。

我们可能犯的另一个错误是将一个指向 `Payoff` 实例的指针添加到 `Option` 类作为数据成员，意图是它应该包含有关支付的信息。这导致我们写出类似 `FloatingTypePayoff` 的类，也列在代码中。它用于实现浮动回望期权，并存储有关类型的信息（如在值、看涨或看跌）。但由于敲定价是在期权到期时确定的，因此无法指定敲定价，也无法通过我们指定的接口实现支付。如果调用它的 `operator()` 会抛出异常。在这种情况下，我们不妨没有支付，只需将类型传递给回望期权。也就是说，如果它的基类 `Option` 不期望得到支付。另一件我们决定清理代码时要记住的事。

13.4 数学相关类

除了 C++ 标准库提供的工具之外，QuantLib 还需要一些数学工具。以下是其中一些内容的简要概述。

13.4.1 插值

插值属于一种在 QuantLib 中不常见的类：即使用起来也可能不安全的类。

[17]这也限制了我们模拟基于底层资产多重确定的支付。例如，普通期权的支付被传递给亚式期权，并且确定的平均值在传递到支付之前，要先在外部完成。人们可能希望将整个过程描述为“支付”。

基类 `Interpolation` 显示在下面的代码中。它对两个底层随机访问序列 x 和 y 的值插值，并提供 `operator()` 返回插值，以及一些其他便捷方法。就像我们在上一节中看到的 `Calendar` 类一样，它通过 pimpl 惯用法实现多态行为：它声明了一个 `Impl` 内部类，其派生类将实现特定的插值，`Interpolation` 类将调用传递给它自己的方法。另一个内部类模板 `templateImpl` 实现了通用机制，并存储了底层数据。

`Interpolation` 类的概要。

```
class Interpolation : public Extrapolator {
  protected:
    class Impl {
      public:
        virtual ~Impl() {}
        virtual void update() = 0;
        virtual Real xMin() const = 0;
        virtual Real xMax() const = 0;
        virtual Real value(Real) const = 0;
        virtual Real primitive(Real) const = 0;
        virtual Real derivative(Real) const = 0;
    };
    template <class I1, class I2>
    class templateImpl : public Impl {
      public:
        templateImpl(const I1& xBegin,
                     const I1& xEnd,
                     const I2& yBegin);
        Real xMin() const;
        Real xMax() const;
      protected:
        Size locate(Real x) const;
        I1 xBegin_, xEnd_;
        I2 yBegin_;
    };
    boost::shared_ptr<Impl> impl_;
  public:
    typedef Real argument_type;
    typedef Real result_type;
    bool empty() const { return !impl_; }
    Real operator()(Real x, bool extrapolate = false) const {
        checkRange(x, extrapolate);
        return impl_->value(x);
    }
    Real primitive(Real x, bool extrapolate = false) const;
    Real derivative(Real x, bool extrapolate = false) const;
    Real xMin() const;
    Real xMax() const;
    void update();
  protected:
    void checkRange(Real x, bool extrapolate) const;
};
```

如你所见，`templateImpl` 不会拷贝 x 和 y 的值。相反，它只是通过存储两个序列的迭代器来提供一种视图。这就是使插值不安全的原因：一方面，我们必须确保 `Interpolation` 实例

的生命周期不超过底层序列的生命周期，以避免指向被破坏的对象；另一方面，任何存储 `interpolation` 实例的类都必须特别注意拷贝。

第一个要求不是一个大问题。内插很少单独使用，它通常作为其他类的数据成员与其底层数据一起存储。这样可以解决生命周期问题，因为插值和数据一起生存和死亡。

第二个也不是一个大问题：但是第一个问题通常是自动处理的，这个问题要求开发人员采取一些措施。正如我所说，通常的情况是将一个 `Interpolation` 实例与其数据一起存储在某个类中。编译器为容器类生成的拷贝构造函数将生成底层数据的新副本，这是正确的。但它也会制作一个新的插值副本，它仍然指向原始数据（因为它会存储原始迭代器的副本）。当然，这是不正确的。

为了避免这种情况，宿主类（host class）的开发人员需要编写一个用户定义的拷贝构造函数，它不仅拷贝数据，而且还重新生成插值，以便指向新的序列，这可能不那么简单。持有 `Interpolation` 实例的对象无法知道其确切类型（隐藏在 `Impl` 类中），因此无法重建它指向其他位置。

解决这个问题的一种方法是给插值提供某种类型的 `clone` 虚方法来返回相同类型的新实例，或者使用 `rebind` 虚方法来更改拷贝后的底层迭代器。然而，这并不是必要的，因为大多数时候我们已经有了插值特性。

你问那是什么？好吧，这是我在第 3 章 中解释插值期限结构时提到的那些 `Linear` 或 `LogLinear` 类。一个例子在下面的代码中，连同它的相应插值类。

`LinearInterpolation` 类的框架及其特性类。

```
template <class I1, class I2>
class LinearInterpolationImpl
    : public Interpolation::templateImpl<I1, I2> {
  public:
    LinearInterpolationImpl(const I1& xBegin,
                            const I1& xEnd,
                            const I2& yBegin)
        : Interpolation::templateImpl<I1, I2>(xBegin, xEnd, yBegin),
          primitiveConst_(xEnd - xBegin), s_(xEnd - xBegin) {
    }
    void update();
    Real value(Real x) const {
        Size i = this->locate(x);
        return this->yBegin_[i] + (x - this->xBegin_[i]) * s_[i];
    }
    Real primitive(Real x) const;
    Real derivative(Real x) const;
  private:
    std::vector<Real> primitiveConst_, s_;
};
class LinearInterpolation : public Interpolation {
  public:
    template <class I1, class I2>
```

```
    LinearInterpolation(const I1& xBegin,
                        const I1& xEnd,
                        const I2& yBegin) {
        impl_ = shared_ptr<Interpolation::Impl>(
            new LinearInterpolationImpl<I1, I2>(xBegin, xEnd, yBegin));
        impl_->update();
    }
};
class Linear {
  public:
    template <class I1, class I2>
    Interpolation interpolate(const I1& xBegin,
                              const I1& xEnd,
                              const I2& yBegin) const {
        return LinearInterpolation(xBegin, xEnd, yBegin);
    }
    static const bool global = false;
    static const Size requiredPoints = 2;
};
```

`LinearInterpolation` 类很简单：它只定义了一个模板构造函数（类本身不是模板），它实例化了正确的实现类。后者继承自 `templateImpl`，是执行繁重工作的类，实现了实际的插值公式（借助于在其基类中定义的 `locate` 等方法）。

`Linear` 特性类定义了一些静态信息，即我们至少需要两个点进行线性插值，而改变一个点只会影响局部插值，并且还定义了一个 `interpolate` 方法，该方法可以从 x 和 y 的一组迭代器中创建特定类型的插值。后一种方法是通过所有特性使用相同的接口实现的（例如样条插值需要更多的参数，它们被传递给特性构造函数并存储），并且它将有助于我们的拷贝问题。例如，如果你看看第 3 章 中 `InterpolatedZeroCurve` 类的代码，你会发现我们正在存储一个特性类的实例（它在那里被称为 `interpolator_`）以及插值和底层数据。如果我们在存储插值的任何类中都这样做，我们将能够使用特性在拷贝构造函数中创建一个新特性。

不幸的是，我们目前无法强制在存储插值的类中编写拷贝构造函数，因此其开发人员必须记住。我们没有办法，也就是说，不使 `Interpolation` 类不可拷贝，从而也阻止了有用的惯用法（比如像特性那样从一个方法返回插值）。在 C++11 中，我们通过使其不可拷贝和可移动来解决这个问题。

题外话：戈尔迪之结。[a]

在实现存储插值的类时，编写拷贝构造函数的替代方法是直接解决问题，并使类不可拷贝。它可能不像看起来那么严重：这些类通常是期限结构，而且通常它们在 `shared_ptr` 中传递，不需要拷贝。

（真实的故事：大多数曲线在 1.0 版之前已经不可拷贝了一段时间，并且没有人抱怨它。最后，我们重新引入拷贝作为方便，但我仍然不确定它是否必要。）

[a]译注：戈尔迪之结（Gordian knot），戈尔迪是古希腊神话传说中小亚细亚弗里基亚的国王, 他曾经打了个分辨不

> 出头尾的复杂绳结，并把它放在神庙里。神谕说能解开此结的人将能统治亚细亚。亚历山大大帝见到这个绳结之后，用剑将其劈为两半，解开了这个难题，这个神谕后来似乎正好应验。戈尔迪之结常被比喻作难以理清的问题，也引申为干净利落的解决难题。

最后一点：插值将迭代器存储到原始数据中，但这还不足以在任何数据更改时使其保持最新。当发生这种情况时，必须调用它的 `update` 方法，以便插值可以刷新其状态，这是包含插值和数据的类的责任（并且可能会将其注册为任何可变内容的观察者。）这也适用于那些可能直接读取数据的插值，例如线性插值：根据实现情况，它们可能会预先计算一些结果，并将其存储为要更新的状态。（当前的 `LinearInterpolation` 实现为点之间的斜率，以及节点处的原始值执行此操作。[18]根据数据更新的频率，这可能是优化或是劣化。）

13.4.2 一维求解器

求解器用于第 3 章 中描述的 bootstrap 过程，第 4 章 中提到的利率计算，以及需要计算值以匹配目标的任何代码。例如，在给定函数 f 的情况下，需要在给定精度内找到 x，使得 $f(x) = \xi$。

现有的求解器会找到 x，使得 $f(x) = 0$。当然，这并不会使它们变得不那么通用，但它要求你定义额外的辅助函数 $g(x) \equiv f(x) - \xi$。其中有一些算法，都取自《Numerical Recipes in C》(Press *et al*, 1992)，并且适当地重新实现。[19]

下面的代码显示了类模板 `Solver1D` 的接口，作为可用求解器的基类。

`Solver1D` 模板类以及一些派生类的接口。

```
template <class Impl>
class Solver1D : public CuriouslyRecurringTemplate<Impl> {
  public:
    template <class F>
    Real solve(const F& f,
               Real accuracy,
               Real guess,
               Real step) const;
    template <class F>
    Real solve(const F& f,
               Real accuracy,
               Real guess,
               Real xMin,
               Real xMax) const;
    void setMaxEvaluations(Size evaluations);
    void setLowerBound(Real lowerBound);
```

[18] `primitive` 和 `derivative` 方法的存在是一种实现泄漏（implementation leak）。它们是插值利率曲线所要求的，以便从零息利率推算远期利率，以及相反的过程。

[19] 这也适用于一些多维优化器。在这种情况下，除了显而易见的版权问题之外，我们还重写了它们，以便使用惯用的 C++，以及索引从 0 开始的数组。

```
    void setUpperBound(Real upperBound);
};
class Brent : public Solver1D<Brent> {
  public:
    template <class F>
    Real solveImpl(const F& f,
                   Real xAccuracy) const;
};
class Newton : public Solver1D<Newton> {
  public:
    template <class F>
    Real solveImpl(const F& f,
                   Real xAccuracy) const;
};
```

它提供了一些通用的样板代码：`solve` 方法的一个重载查找包含解 x 的下限和上限值，而另一个检查解实际上是否被传递来的最小值和最大值包括。在这两种情况下，实际计算都委托给派生类定义的 `solveImpl` 方法并实现特定的算法。其他方法允许你对探索的范围，或函数计算的次数设置约束。

委托到 `solveImpl` 是使用已经在第 7 章 中描述的奇异递归模板模式（CRTP）实现的。当我们编写这些类时，我们处于模板狂潮的高峰（我提到过我们甚至有表达式模板的实现吗？(Veldhuizen, 2000)），所以你可能会怀疑这个选择是由当时的时尚决定的。但是，不可能使用动态多态。我们希望求解器能够处理任何函数指针或函数对象，那是还没有 `boost::function`，这迫使我们使用模板方法。由于后者不能是虚拟，因此 CRTP 是将样板代码放在基类中，并让它调用派生类中定义的方法的唯一方式。

结束本小节的几点说明。第一：如果你想编写一个带求解器的函数，CRTP 的使用迫使你把它变成一个模板，这可能很尴尬。说实话，大多数时候我们都没有烦恼，只是硬编码了明确的求解器选择。如果你这样做我也不会怪你。第二：大多数求解器只会调用 `f(x)` 来使用 `f`，因此它们可以处理任何可以作为函数调用的东西，但是 `Newton` 和 `NewtonSafe` 还需要定义 `f.derivative(x)`。如果我们使用动态多态，这也可能很尴尬。第三个，也是最后一个：`Solver1D` 接口没有指定传递来的精度 ϵ 是否应该应用于 x（也就是说，如果返回的 $\tilde{x}$ 应该在真正根的 ϵ 范围内）或是 $f(x)$（即，如果 $f(\tilde{x})$ 应该在 0 的 ϵ 范围内）。但是，所有现有的求解器都将其视为 x 的精度。

13.4.3 优化器

多维优化器比一维求解器更复杂。简而言之，它们找到了一组变量 $\tilde{\mathbf{x}}$，使其成本函数 $f(\mathbf{x})$ 返回其最小值。但是还有更多内容，下面我们将看到成本函数的实现。

在这种情况下，我们没有使用模板。优化器继承自基类 `OptimizationMethod`，如下面的代码所示。

`OptimizationMethod` 类的接口。

```
class OptimizationMethod {
  public:
    virtual ~OptimizationMethod() {}
    virtual EndCriteria::Type minimize(
        Problem& P,
        const EndCriteria& endCriteria) = 0;
};
```

除了虚拟析构函数之外，它唯一的方法是 `minimize`。该方法接受一个 `Problem` 实例的引用，该实例又包含对需要最小化的函数的引用，以及相关任何约束，并执行计算。最后，它有返回过多内容的问题。除了最优解数组 $\tilde{\mathbf{x}}$ 之外，它必须至少返回退出计算的原因（最小化是否收敛，或者在达到最大计算次数后它是否返回的是最佳猜测?），并且它最好返回函数在 $\tilde{\mathbf{x}}$ 的函数值，因为它可能已经计算好了。

在当前的实现中，该方法返回退出的原因，并将其他结果存储在 `Problem` 实例中。这也是将问题作为非常引用传递的原因。另一种解决方案可能是单独保留 `Problem` 实例，并以结构体返回所有必需的值，但我看来这可能会更为麻烦。另一方面，我认为 `minimize` 本身没有理由成为非常量的：我的猜测是，这是我们的疏忽（稍后我会回到这里）。

转向 `Problem` 类，显示在下面的代码中。

`Problem` 类的接口。

```
class Problem {
  public:
    Problem(CostFunction& costFunction,
            Constraint& constraint,
            const Array& initialValue = Array());
    Real value(const Array& x);
    Disposable<Array> values(const Array& x);
    void gradient(Array& grad_f, const Array& x);
    // ... other calculations ...
    Constraint& constraint() const;
    // ... other inspectors ...
    const Array& currentValue();
    Real functionValue() const;
    void setCurrentValue(const Array& currentValue);
    Integer functionEvaluation() const;
    // ... other results ...
};
```

正如我所提到的，它将诸如要最小化的成本函数、约束和可选猜测之类的参数组合在一起。它提供了调用底层成本函数的方法，同时追踪计算的次数，我将在谈论成本函数时进行描述；一些组件的检查器；检索结果的方法（以及设置它们的方法，后者由优化器使用）。

问题（不是双关语）在于以非常引用接受和存储其组件。我稍后会谈到它们是否可以是常引用。事实上，引用本身就是一个问题，因为它将责任放在客户端代码上，以确保它们的生命周期至少与 Problem 实例的生命周期一样长。

在优化器将结果以结构体返回的替代实现中，问题可能没有实际意义：我们可能会废除 Problem 类，并将其组件直接传递给 minimize 方法。这将避开生命周期问题，因为它们不会被存储。缺点是每个优化器都必须追踪函数计算的次数，从而导致代码库中出现一些重复。

与一维求解器不同，成本函数不是最小化方法的模板参数。它需要从 CostFunction 类继承，如下面的代码所示。

CostFunction 类的接口。

```
class CostFunction {
  public:
    virtual ~CostFunction() {}
    virtual Real value(const Array& x) const = 0;
    virtual Array values(const Array& x) const = 0;
    virtual void gradient(Array& grad, const Array& x) const;
    virtual Real valueAndGradient(Array& grad,
                                  const Array& x) const;
    virtual void jacobian(Matrix& jac, const Array& x) const;
    virtual Array valuesAndJacobian(Matrix& jac,
                                    const Array& x) const;
};
```

不出所料，它的接口声明了 value 方法，它返回了给定参数数组的函数值。[20]但是它也声明了一个返回数组的 values 方法，这个有点令人惊讶，直到你记得优化器通常用于校准多个数值。而 value 返回要最小化的总误差（比如说误差的平方和，或类似的东西），values 返回每个数值上的误差集合，有些算法可以利用这些信息更快地收敛。

其他方法返回导数，供某些特定算法使用：gradient 计算每个变量的导数，并将其存储在作为第一个参数数组中；jacobian 执行与 values 相同的操作，用于填充矩阵；valueAndGradient 和 valuesAndJacobian 方法同时计算值和导数以提高效率。它们有一个默认实现，通过有限差分计算导数。然而计算量的代价是高昂的，如果你不能用解析计算覆盖方法，我不确定使用基于导数的算法是否值得。

注意：查看 CostFunction 的接口显示它的所有方法都被声明为常量的，因此将它作为非常引用传递给 Problem 构造函数可能是一件蠢事。将它更改为常量的会扩大对构造函数的约定，因此可能应该在不破坏向后兼容性的情况下实现。

最后是 Constraint 类，如下面的代码所示。

[20]虽然你可能会有点惊讶于它没有声明 operator()。

`Constraint` 类的接口。

```
class Constraint {
  protected:
    class Impl;
  public:
    bool test(const Array& p) const;
    Array upperBound(const Array& params) const;
    Array lowerBound(const Array& params) const;
    Real update(Array& p,
                const Array& direction,
                Real beta);
    Constraint(const shared_ptr<Impl>& impl = shared_ptr<Impl>());
};
```

它作为约束的基类，用于规定成本函数的定义域，QuantLib 还提供了一些预定义的约束（这里没有显示），以及一个 `CompositeConstraint` 类，可以用来将它们中的一些合并为一个约束。

它的主要方法是 `test`，它接受一个变量数组并返回它们是否满足约束条件。也就是说，判断数组属于我们指定为有效的定义域。它还定义了 `upperBound` 和 `lowerBound` 方法，理论上应该指定变量的最大值和最小值，但实际上并不能总是正确地指定它们。考虑到定义域是一个圆的情况，你会发现有些情况下 x 和 y 都在它们的上限和下限之间，但结果点在定义域之外。

还有几点。首先，`Constraint` 类还定义了 `update` 方法。它不是常量的，如果它更新了约束就有意义，没有更新就没有意义。它接受一系列变量和一个方向，并在给定方向上扩展原始数组，直到它满足约束。它应该是常量的，应该以不同的名称命名，并且正如孩子们说“我做不了”。[21]但这可能要被改正。其次，该类使用 pimpl 惯用法（参见上一节），默认构造函数也接受指向实现的可选指针。如果我今天要编写这个类，我会写一个不带参数的默认构造函数和一个额外的构造函数，它接受实现并声明为保护的，并且仅由派生类使用。

一些简短的最终想法，关于这些类常量声明的正确性。总结：它不是很好，有些方法可以改正，有些方法则无法改正。例如，更改 `minimize` 方法会破坏向后兼容性（因为它是一个虚方法，常量性是其签名的一部分），以及一些优化器，它们从 `minimize` 调用其他方法，并使用数据成员作为在方法之间传递信息的方式。[22]如果我们在版本 1.0 之前更加努力地检查代码，我们本可以避免这种情况。对于你们年轻的程序员来说，这是一个教训。

13.4.4 统计

统计类主要是为了从蒙特卡罗模拟中收集样本（你记得它们在第 6 章）。这部分的全部功能是作为一系列装饰器实现的，每个装饰器都添加了一层方法，而不是整体。据我所知，

[21]译注：但实际上能做到。

[22]其中一些声明这些数据成员是可变的，这表明这些方法过去可能都是常量的。在我写这篇文章时，我没有进一步研究过这个问题。

这是因为你可以在最基础的两个类中选择一个。基于底层的通用接口，分层设计使你可以一次性创建更高级的功能。

你可以选择的第一个类称为 IncrementalStatistics，如下所示，并且具有你或多或少预期的接口：它可以返回样本数量、它们的组合权重如果它们被赋予权重，以及一些统计结果。

IncrementalStatistics 类的接口。

```
class IncrementalStatistics {
  public:
    typedef Real value_type;
    IncrementalStatistics();
    Size samples() const;
    Real weightSum() const;
    Real mean() const;
    Real variance() const;
    Real standardDeviation() const;
    Real errorEstimate() const;
    // skewness, kurtosis, min, max...
    void add(Real value, Real weight = 1.0);
    template <class DataIterator>
    void addSequence(DataIterator begin, DataIterator end);
};
```

样本可以逐个添加，也可以作为由两个迭代器界定的序列添加。这个类的重点在于它不存储传递来的数据，而是在运行中更新统计数据。这个想法是为了节省原本用于存储的内存，相比于现在，在 2000 年（当时一台计算机可能只有 128 或 256 MB 的 RAM），这是一个更大的问题。实现在过去是自己编写的，现在它是用 Boost 的 accumulator 库编写的。

第二个类是 GeneralStatistics，实现了相同的接口，并添加了一些其他方法，这是因为它存储（因此可以返回）传递来的数据。例如，它可以返回分位数，或对其数据进行排序。它还提供了一个 expectationValue 模板方法，可用于定制计算。如果你感兴趣的话，在本节末尾的“题外话”还有更多内容。

GeneralStatistics 类的接口。

```
class GeneralStatistics {
  public:
    // ... same as IncrementalStatistics ...
    const std::vector<std::pair<Real, Real>>& data() const;
    template <class Func, class Predicate>
    std::pair<Real, Size>
    expectationValue(const Func& f,
                     const Predicate& inRange) const;
    Real percentile(Real y) const;
    // ... other inspectors ...
    void sort() const;
    void reserve(Size n) const;
};
```

接下来，外层。第一个，添加与风险相关的统计量，如预期损失或风险价值。问题是你通常需要所有样本。因此，在增量统计的情况下，我们必须放弃精确的计算并寻找近似值。一种可能性是采用样本的均值和方差，假设它们来自具有相同矩的高斯分布，并基于该假设得到解析结果。这就是 `GenericGaussianStatistics` 类所做的。

`GaussianStatistics` 类的接口。

```
template <class S>
class GenericGaussianStatistics : public S {
  public:
    typedef typename S::value_type value_type;
    GenericGaussianStatistics() {}
    GenericGaussianStatistics(const S& s) : S(s) {}
    Real gaussianDownsideVariance() const;
    Real gaussianDownsideDeviation() const;
    Real gaussianRegret(Real target) const;
    Real gaussianPercentile(Real percentile) const;
    Real gaussianValueAtRisk(Real percentile) const;
    Real gaussianExpectedShortfall(Real percentile) const;
    // ... other measures ...
};
typedef GenericGaussianStatistics<GeneralStatistics> GaussianStatistics;
```

正如我所提到和你所看到的，它是作为装饰者实现的。它接受要装饰的类作为模板参数，继承它以使它仍然具有其基类的所有方法，并添加新方法。QuantLib 提供了一个默认的实例化，`GaussianStatistics`，其中模板参数是 `GeneralStatistics`。是的，我也期望有增量版本，但这是有原因的，请宽容我一会儿。

当基类存储完整的样本集时，我们可以编写一个计算实际风险度量的装饰器，这将是 `GenericRiskStatistics` 类。至于高斯统计，我不会讨论实现（你可以在 QuantLib 中查找）。

`RiskStatistics` 类的接口。

```
template <class S>
class GenericRiskStatistics : public S {
  public:
    typedef typename S::value_type value_type;

    Real downsideVariance() const;
    Real downsideDeviation() const;
    Real regret(Real target) const;
    Real valueAtRisk(Real percentile) const;
    Real expectedShortfall(Real percentile) const;
    // ... other measures ...
};
typedef GenericRiskStatistics<GaussianStatistics> RiskStatistics;
```

如你所见，这些层可以组合在一起。QuantLib 提供的默认实例化，`RiskStatistics`，接受 `GaussianStatistics` 为基类，因此提供高斯和实际度量。这也是 `GeneralStatistics` 被用作后者基类的原因。

最重要的是，它可以有其他装饰者。QuantLib 提供了一些，但我不会在这里显示它们的代码。一个是 SequenceStatistics，当样本是一个数组而不是一个数字时可以使用，内部使用标量统计类实例的向量，并且还增加了样本元素之间的相关性和协方差的计算。例如，它用于 LIBOR 市场模型，其中每个样本通常在不同时间收集现金流。另外两个是 ConvergenceStatistics 和 DiscrepancyStatistics，它们提供了关于样本序列属性的信息，不会在 QuantLib 中的任何其他地方使用，但至少我们有礼貌地为这两个文件编写单元测试。

题外话：极端期望。

回顾一下 GeneralStatistics 类，我不确定我是应该为此感到骄傲还是感到羞耻，因为，哦，伙计，我真的花了大功夫来实现泛化。

这可能是数学。它始于一个简单的实现，但是看看平均值、方差、甚至是其他层中定义的一些更复杂的公式，我看到它们都可以写成（稍后给出或接受调整）

$$\frac{\sum_{x_i \in \mathcal{R}} f(x_i) w_i}{\sum_{x_i \in \mathcal{R}} w_i},$$

也就是说，$f(x)$ 在某个范围 $\mathcal{R}$ 上的期望值。对于平均值，$f(x)$ 将是恒等函数，$\mathcal{R}$ 将是全体样本；对于方差，$f(x)$ 为 $(x - \bar{x})^2$，范围相同，等等。结果是一个模板方法 expectationValue，该方法接受函数 f 和范围 $\mathcal{R}$，并返回相应的结果和范围内的样本数。大多数其他方法是通过使用相关输入调用它来实现的。如果你最初对于这样实现的意义有困惑

```
return expectationValue(identity<Real>(),
                        everywhere()).first;
```

那我不能怪你。顺便说一下，我当时一定在学习函数式编程。范围作为一个函数传递，它接受一个样本并返回 true 或 false，具体取决于它是否在范围内，并且上面的任何地方都是我为编写这种代码而添加的一些小的预定义辅助函数之一。在有人把 $(x - \bar{x})^2$ 写成下面的样子之前，这一切都很有趣

```
compose(square<Real>(),
        std::bind2nd(std::minus<Real>(),mean()))
```

自我嘲讽的题外话：上面是非常通用的，并且允许客户端代码创建新的计算，但可能数学倾向有点太多，并且可能会产生含糊的代码，所以我不确定我是否能够在这里达到正确的平衡。在 C++11 中用 lambda 替换绑定肯定会有所帮助，当我们开始使用 lambda 时我们将查看对这段代码产生的结果。另一方面，性能不应该是一个问题：expectationValue 是一个模板，上面的函数，就像 compose 和 everywhere，所以编译器可以看到它们的实现，并且可能内联它们。在这种情况下，结果可能是直接实现均值或方差公式的更简单的循环。

13.4.5 线性代数

我没有太多关于 Array 和 Matrix 类（如下面的代码所示）当前实现的文字。

Array 类和 Matrix 类的概要。

```
class Array {
  public:
    explicit Array(Size size = 0);
    // ... other constructors ...
    Array(const Array&);
    Array(const Disposable<Array>&);
    Array& operator=(const Array&);
    Array& operator=(const Disposable<Array>&);
    const Array& operator+=(const Array&);
    const Array& operator+=(Real);
    // ... other operators ...
    Real operator[](Size) const;
    Real& operator[](Size);
    void swap(Array&);
    // ... iterators and other utilities ...
  private:
    boost::scoped_array<Real> data_;
    Size n_;
};
Disposable<Array> operator+(const Array&, const Array&);
Disposable<Array> operator+(const Array&, Real);
// ... other operators and functions ...
class Matrix {
  public:
    Matrix(Size rows, Size columns);
    // ... other constructors, assignment operators etc. ...
    const_row_iterator operator[](Size) const;
    row_iterator operator[](Size);
    Real& operator()(Size i, Size j) const;
    // ... iterators and other utilities ...
};
Disposable<Matrix> operator+(const Matrix&, const Matrix&);
// ... other operators and functions ...
```

它们的接口是你所预期的：构造函数和赋值运算符、元素访问（Array 类提供 a[i] 语法；Matrix 类同时提供 m[i][j] 和 m(i, j) 语法，因为我们的目标是让使用者满意）、一堆算术运算符，全部是逐元素操作，[23]以及一些实用工具。没有用于调整大小或适用于容器的其他操作方法，因为这些类不应该这样使用，它们是数学工具。数据存储由一个简单的 scoped_ptr 提供，管理底层内存的生命周期。

在 Array 的情况下，我们还提供了一些函数，如 Abs、Log 等等。作为好公民，我们不会重载命名空间 std 中相应的函数，因为这是被标准禁止的。更复杂的功能（例如矩阵平方根或各种分解）可以在单独的模块中找到。

[23] a * b 返回元素间乘积而不是点积总是困扰我，但我似乎在程序员中是个另类。

简而言之，或多或少直接实现数组和矩阵。一个不明显的事情是 `Disposable` 类模板的存在，我将在本附录的另一部分中更详细地描述。暂时，我只想说这是前 C++11 时代一个关于移动语义（move semantic）的尝试。

这个想法是试图减少抽象惩罚。运算符重载非常方便，毕竟，`c = a + b` 比 `add(a, b, c)` 更容易理解，但不是免费的：声明加法为

```
Array operator + (const Array& a, const Array& b);
```

意味着运算符必须创建并返回一个新的 `Array` 实例，即必须分配并可能拷贝其内存。当操作数增加时，开销也增加。

在 QuantLib 的第一个版本中，我们尝试使用表达式模板来缓解此问题。这个想法（我只会粗略地描述，所以我建议你阅读 (Veldhuizen, 2000) 的细节）是运算符不返回数组，而是某种解析树保存对表达式项的引用。因此，例如，`2 * a + b` 实际上不会执行任何计算，只会创建一个包含相关信息的小结构。只有在赋值表达式时才会展开，并且在那时，编译器将检查整个表达式并生成单个循环，该循环既计算结果又将其拷贝到被分配的数组中。

这种技术今天仍然有重大作用（鉴于编译器技术的进步，可能更是如此），但我们在几年后放弃了它。它很难阅读和维护（相较 `operator +` 的当前声明

```
VectorialExpression<
    BinaryVectorialExpression<
        Array::const_iterator,
        Array::const_iterator,
        Add>>
operator+(const Array& v1, const Array& v2);
```

代码看起来是这个样子），而且并非所有编译器都能够处理它，这迫使我们同时维护表达式模板和更简单的实现。因此，当 C++ 社区开始谈论移动语义时，一些实现的想法开始出现，我们接受了提议示并切换到了 `Disposable`。

正如我所说，这些年来编译器取得了很大进步。如今，我猜它们都支持表达式模板实现，而且技术本身可能已经有所改进。但是，如果我今天编写代码（或者如果我开始改变那些东西），那么问题可能是是否要编写诸如 `Array` 或 `Matrix` 之类的类。至少，我会考虑用 `std::valarray` 来实现它们，它应该为这样的任务提供基础设施。但最后，我可能会选择一些现有的库，例如 uBLAS，它可以在 Boost 中找到，由数值计算领域真正的专家编写，我们已经在 QuantLib 的某些部分用于专门的计算。

13.5 全局配置

在下面列出的框架中 `Settings` 类是一个单体（见声明的后半部分），它保存整个 QuantLib 的全局信息。

`Settings` 类的概要。

```
class Settings : public Singleton<Settings> {
  private:
    class DateProxy : public ObservableValue<Date> {
        DateProxy();
        operator Date() const;
        // ...
    };
    // more implementation details
  public:
    DateProxy& evaluationDate();
    const DateProxy& evaluationDate() const;
    boost::optional<bool>& includeTodaysCashFlows();
    boost::optional<bool> includeTodaysCashFlows() const;
    // ...
};
```

它的大部分数据都是你可以在官方文档中查找的标志，或者你可以简单地放着不管。你需要管理的一条信息是估值日期，默认为今天的日期，用于金融工具的定价和任何其他量的确定。

这提出了一个挑战：当估值日期发生变化时，必须通知价值取决于估值日期的金融工具。这是通过间接返回相应的信息来完成的，即包装在代理类中，这可以从相关方法的签名中看出。代理继承自 `ObservableValue` 类模板（在下面的代码中概述），该模板可隐式转换为 `Observable`，并重载赋值运算符以通知任何更改。最后，它允许自动将代理类转换为包装值。

`ObservableValue` 类模板的框架。

```
template <class T>
class ObservableValue {
  public:
    // initialization and assignment
    ObservableValue(const T& t)
        : value(t), observable_(new Observable) {
    }
    ObservableValue<T>& operator=(const T& t) {
        value_ = t;
        observable_->notifyObservers();
        return *this;
    }
    // implicit conversions
    operator T() const { return value_; }
    operator boost::shared_ptr<Observable>() const {
        return observable_;
    }
  private:
    T value_;
    boost::shared_ptr<Observable> observable_;
};
```

这允许人们使用具有自然语法的工具。一方面，观察者可以注册估值日期，如下所示：

```
registerWith(Settings::instance().evaluationDate());
```

另一方面，可以将返回值当做 `Date` 实例使用，如下所示：

```
Date d2 = calendar.adjust(Settings::instance().evaluationDate());
```

这触发自动转换。on the gripping hand，[24]可以使用简单的赋值语法来设置估值日期，如下所示：

```
Settings::instance().evaluationDate() = d;
```

日期变化时将通知所有观察者。

当然，房间里的大象就是我们有一个全局估值日期的事实。显而易见的缺点是，不能执行使用两个不同估值日期的并行计算，至少在 QuantLib 默认配置中是这样。但事实确实如此，这也不是全部。一方面，有一个编译标志，允许程序每个线程有一个不同的 `Settings` 实例（用户方面要做一点儿工作），但正如我们所看到的，这并不能解决所有问题。另一方面，即使在单线程程序中，全局数据也可能导致不愉快：即使只想估值不同日期上的一个金融工具，当估值日期重新设置回原来的值时，变化将触发系统中每个其他金融工具的重新计算。

这清楚地指出（也就是说，在我们谈论它时很多聪明人有相同的想法）某些上下文类应该取代全局配置。但是如何为任何给定的计算选择上下文？

将 `setContext` 方法添加到 `Instrument` 类，并安排事情以便在计算过程中金融工具将上下文传送到其引擎，然后传送到需要它的任何期限结构，这样做将很有吸引力。但是，我认为这不容易实现。

首先，金融工具及其引擎并不总是知道计算中涉及的所有期限结构。例如，互换包含许多票息，其中任何一个可能会或可能不会引用预测曲线。除非我们将相关机制添加到所有相关类中，否则我们不会接触它们。我不确定我们是否要为票息设置上下文。

其次，更重要的是，为引擎设置上下文将是一种修改操作。在计算期间将它留在金融工具调用其 `NPV` 方法（该方法应该是常量的）的某个时刻执行它。这会使触发竞争条件变得太容易了，例如，看似无害的操作，对两个金融工具使用相同的贴现曲线并在不同的日期对它们估值。在并行编程方面经验最少的用户不会想到，例如，在两个并发线程中重新链接相同的句柄。但是当修改隐藏在常量方法中时，她可能不会意识到这一点。（可是，你会说等等。在调用 `NPV` 期间难道没有其他修改操作？接的好：请参阅本节末尾 的“题外话”。）

所以我们似乎必须在开始计算之前设置上下文。这排除了从金融工具中驱动整件事（因为，我们将隐藏这样一个事实，即为一个金融工具设置一个上下文可以撤消另一个上下文的工

[24]译注：“on the gripping hand”意为第三方面，再一方面。出自 Larry Niven 和 Jerry Pournelle 所著的科幻小说《The Mote in God's Eye》，小说中的外星人拥有三只手臂。

作，而两个上下文共享一个期限结构），并建议我们必须在几个期限结构上明确设置上下文。从好的方面来说，我们不再冒险去在不知不觉中试图为同一个对象设置相同的上下文。缺点是我们的设置变得更复杂，如果我们想要在不同的上下文中同时使用它们，我们必须拷贝曲线：例如，不同日期的两个并行计算意味着，两个用于贴现的隔夜曲线的副本。如果我们必须这样做，我们也可以管理每个线程的单体。

最后，我正在跳过上下文被传递但未被保存的场景。这将导致如下的方法调用，

```
termStructure->discount(t, context);
```

这将完全破坏缓存，会引起所有相关方的不安，如果我们想要这样的东西，我们会在 Haskell 中编写。

总结一下：我讨厌用不愉快的音符结束这一部分，但一切都不顺利。全局配置是一个缺陷，但我没有解决方案，更糟糕的是，可能的改变会增加复杂性。我们不仅要告诉第一次使用 Black-Scholes 公式的用户，她需要期限结构、报价、金融工具和定价引擎：我们还将上下文混在其中。这没什么帮助吧？

题外话：比 B 级片更多的变化。[a]

不幸的是，在调用所谓的常量方法 `Instrument::NPV` 时，事情已经发生了许多变化。

首先，引擎内部有 `arguments` 和 `results` 结构体，它们在计算过程中被读取和写入，从而防止同一引擎同时用于不同的金融工具。这可以通过向引擎添加锁（这将序列化计算），或是通过修改接口来改正，即引擎的 `calculate` 方法将 `arguments` 结构体作为参数，并返回 `results` 结构体。

然后，金融工具本身有可变数据成员，这些成员在计算结束时写入。这是否是一个问题取决于计算的类型。我猜想在并行线程中计算两次金融工具的值可能只会导致相同的值被写入两次。

想到的最后一个问题是隐藏的修改，它可能是最危险的。在计算过程中尝试使用期限结构可能会触发其 bootstrap 计算，并且两个并发的计算会破坏彼此。由于 bootstratp 计算的递归性质，我甚至不确定如何在它周围添加锁定。因此，如果你决定执行并行计算（小心，事先设置所有内容并使用相同的估值日期），请务必在启动之前触发曲线的完整 bootstrap 计算。

[a] 译注：B 级片（B-movie）即拍摄时间短且制作预算低的影片，通常伴随着离奇的故事和跳脱的剧情。

13.6 实用工具

在 QuantLib 中，有许多类和函数不对金融概念进行建模。它们是螺母和螺栓，用于为 QuantLib 的其余部分构建一些脚手架。本节专门介绍其中一些工具。

13.6.1 智能指针与句柄

运行时多态的使用决定了在堆上分配许多（即使不是大多数）对象。这引发了内存管理的问题，内置垃圾收集在其他语言中解决了这个问题，但是在 C++ 中留给了开发人员。

我不会详述内存管理中的许多问题，特别是因为它们现在已成为过去。任务的难度（特别是在存在异常的情况下）足以否定人工管理。因此，发明了自动化过程的方法。

C++ 社区中的首选武器是智能指针：类似于内置指针的类，但是当需要所指向对象时可以处理对象的生命周期，而当对象不再需要时它们会被破坏。这种类存在使用不同技术的若干实现。我们选择了 Boost 库中的智能指针（最值得注意的是 `shared_ptr`，现在包含在 ANSI/ISO C++ 标准中）。你可以浏览 Boost 网站以获取文档。在这里，我只想提一下它们在 QuantLib 中用于完全自动化的内存管理。在 QuantLib 所包含的数万行代码中，到处是对象的动态分配，但没有一个 `delete` 语句。

指针的指针（如果你需要快速复习，请参阅本节末尾 的“题外话”了解其目的和语义）也被智能等价物所取代。我们选择不简单地使用智能指针的智能指针，一方面，因为不得不写成

```
boost::shared_ptr<boost::shared_ptr<YieldTermStructure>>
```

即使在 Emacs 中，也很快就会令人厌烦。另一方面，因为内部的 `shared_ptr` 必须动态分配，这感觉不对劲，并且 on the gripping hand，因为它会使实现可观察性变得困难。相反，为此目的提供了一个名为 `Handle` 的类模板。它的实现，如下面的代码所示，依赖于一个名为 `Link` 的中间内部类，它存储一个智能指针。反过来，`Handle` 类存储一个指向 `Link` 实例的智能指针，用可以更容易使用它的方法进行装饰。由于给定句柄的所有副本共享相同的链接，因此当它们中的任何一个链接到新对象时，它们都被授予对新指针的访问权。

Handle 类模板的概要。

```
template <class Type>
class Handle {
  protected:
    class Link : public Observable, public Observer {
      public:
        explicit Link(const shared_ptr<Type>& h =
                            shared_ptr<Type>());
        void linkTo(const shared_ptr<Type>&);
        bool empty() const;
        void update() { notifyObservers(); }
      private:
        shared_ptr<Type> h_;
    };
    boost::shared_ptr<Link<Type>> link_;
  public:
    explicit Handle(
        const shared_ptr<Type>& h = shared_ptr<Type>());
    const shared_ptr<Type>& operator->() const;
    const shared_ptr<Type>& operator*() const;
    bool empty() const;
    operator boost::shared_ptr<Observable>() const;
};
template <class Type>
class RelinkableHandle : public Handle<Type> {
  public:
    explicit RelinkableHandle(
        const shared_ptr<Type>& h = shared_ptr<Type>());
    void linkTo(const boost::shared_ptr<Type>&);
};
```

包含的 shared_ptr<Link> 还为句柄提供了可以让其他类观察的方法。Link 类既是观察者又是被观察者。它接收来自其指针的通知，并将它们告知给自己的观察者，并在每次指向不同的指针时发送自己的通知。句柄通过定义自动转换为 shared_ptr<Observable> 来简单地返回包含的链接，从而利用了这种行为。因此，声明

```
registerWith(h);
```

是合法的，并按预期工作。注册观察者将接收来自链接和（间接）指向对象的通知。

你可能已经注意到，重新链接句柄的方法（即，使其所有副本指向一个不同的对象）没有被赋予 Handle 类本身，而是赋予派生的 RelinkableHandle 类。其基本原理是提供对可用于重新链接的句柄的控制，尤其是哪个句柄不能重新链接。在典型的用例中，Handle 实例将被实例化（例如，存储利率曲线），并传递给许多金融工具、定价引擎或其他对象，这些对象将存储句柄的副本，并在需要时使用它。重点是，不管是什么原因，都不得允许对象

（或客户端代码获取句柄，如果对象通过检查器公开它）重新链接它存储的句柄，这样做会影响许多其他对象。[25]

如果你愿意，只能从原始句柄（主句柄）更改链接。

鉴于人类的脆弱性，我们希望编译器强制执行此操作。使 `linkTo` 方法成为一个常量方法，并且从我们的检查器返回常量句柄是行不通的，客户端代码可以简单地创建一个副本来获取非常量句柄。因此，我们从 `Handle` 接口中删除了 `linkTo`，并将其添加到派生类中。类型系统很有利于我们。一方面，我们可以将主句柄实例化为 `RelinkableHandle`，并将其传递给任何需要 `Handle` 的对象，这将发生从派生类到基类的自动转换，使对象具有切片的但功能齐全的句柄。另一方面，当从检查器返回 `Handle` 实例的副本时，无法将其向下转换为 `RelinkableHandle`。

题外话：指针语义学。

在类实例中存储指针的副本使持有者可以访问指针对象的当前值，如下面的代码所示：

```
class Foo {
    int* p;

  public:
    Foo(int* p) : p(p) {}
    int value() { return *p; }
};
int i = 42;
int* p = &i;
Foo f(p);
cout << f.value();    // will print 42
i++;
cout << f.value();    // will print 43
```

但是，存储的指针（它是原始指针的副本）在外部指针修改时不会被修改。

```
int i = 42, j = 0;
int* p = &i;
Foo f(p);
cout << f.value(); // will print 42
p = &j;
cout << f.value(); // will still print 42
```

像往常一样，解决方案是添加另一个间接层。修改 `Foo` 以便它存储指向指针的指针，为类提供了两种可能性。

```
int i = 42, j = 0;
int* p = &i;
int** pp = &p;
Foo f(pp);
```

[25]这并不像看起来那么难以置信，我们曾被它坑过。

```
    cout << f.value(); // will print 42
    i++;
    cout << f.value(); // will print 43
    p = &j;
    cout << f.value(); // will print 0
```

13.6.2 错误报告

QuantLib 里有很多地方必须检查一些条件，而不是像这样做

```
if (i >= v.size())
    throw Error("index out of range");
```

我们想要更清楚地表达意图，使用类似的语法

```
require(i < v.size(), "index out of range");
```

一方面，我们写的条件是满足而不是相反；另一方面，诸如 `require`、`ensure` 或 `assert` 这样的术语在编程中具有某种规范意义，会告诉我们是在检查前置条件、后置条件还是程序员错误。

我们用宏提供了所需的语法。“Get behind thee”，[26]我就知道你会这么说。没错，宏有一个坏名声，实际上它们给我们带来了一两个问题，我们将在下面看到。但在这种情况下，函数有一个很大的缺点：它们会为所有参数赋值。很多时候，我们想要创建一个中等复杂的错误消息，例如

```
require(i < v.size(),
        "index " + to_string(i) + " out of range");
```

如果 `require` 是一个函数，则无论条件是否满足，都将构建消息，从而导致性能损失，这是不可接受的。使用宏，上面的文本替换为类似的东西

```
if (!(i < v.size()))
    throw Error("index " + to_string(i) + " out of range");
```

仅在违反条件时才构建消息。

下面的代码显示了其中一个宏的当前版本，即 `QL_REQUIRE`，其他宏以类似的方式定义。

[26]译注：可以理解为“恶灵退散”。出自钦定版圣经（King James Bible）中的路加福音（Luke）4:8，“And Jesus answered and said unto him, Get thee behind me, Satan: for it is written, Thou shalt worship the Lord thy God, and him only shalt thou serve.”

QL_REQUIRE 宏的定义。

```
#define QL_REQUIRE(condition, message)                \
    if (!(condition)) {                               \
        std::ostringstream _ql_msg_stream;            \
        _ql_msg_stream << message;                    \
        throw QuantLib::Error(__FILE__, __LINE__,     \
                              BOOST_CURRENT_FUNCTION, \
                              _ql_msg_stream.str());  \
    } else
```

它的定义有一些可能预期到的花哨玩意儿。首先，我们使用 ostringstream 来构建消息字符串。这允许人们使用类似的语法

```
QL_REQUIRE(i < v.size(),
           "index " << i << " out of range");
```

构建消息（你可以通过替换宏中的部分来了解它是如何工作的）。其次，当前函数的名称以及抛出错误的行和文件传递到 Error。根据编译标志，此信息可以包含在错误消息中以帮助开发人员。默认行为是不包含它，因为它对用户来说几乎没用。最后，你可能想知道为什么我们在宏的末尾添加了 else。这是由于宏的一个共同陷阱，即它缺乏词法范围（lexical scope）。需要 else 的代码如下

```
if (someCondition())
    QL_REQUIRE(i < v.size(), "index out of bounds");
else
    doSomethingElse();
```

如果没有宏中的 else，上面的内容将无法正常工作。相反，代码中的 else 将与宏中的 if 配对，代码将转换为

```
if (someCondition()) {
    if (!(i < v.size()))
        throw Error("index out of bounds");
    else
        doSomethingElse();
}
```

它有不同的行为。

最后，我必须描述这些宏的缺点。就像现在一样，它们抛出的异常只能返回它们包含的信息，没有为任何其他相关数据定义检查器。例如，虽然越界消息可能包含传递来的索引，但异常中没有其他方法将索引作为整数返回。因此，信息可以显示给用户，但是对于 catch 子句中的恢复代码是不可用的，除非有人解析消息，但这种努力不值得。目前还没有计划的解决方案，所以如果你有一条线方案，请给我们留言。

13.6.3 `Disposable` 对象

`Disposable` 类模板试图在 C++98 代码中实现移动语义。疑人不用，用人不疑，我们采用了 Andrei Alexandrescu 一篇文章 (Alexandrescu, 2003) 中的想法和技巧，描述了如何在返回临时对象时避免拷贝。

基本思想在那些年里就开始广为流传，并且在 C++11 中给出了它的最终形式：当传递一个临时对象时，将它拷贝到另一个对象的效率往往低于将其内容与目标对象的内容进行交换。你想移动一个临时向量？将指向内存的指针拷贝到新对象中，而不是分配新对象并拷贝元素。在现代 C++ 中，语言本身支持移动语义与 rvalue 引用的概念 (Hinnant *et al*, 2006)。编译器知道它何时处理临时对象，当我们想要将一个对象变成一个时，我们可以在少数情况下使用 `std::move`。在我们的实现中，如下面的代码所示，我们没有这样的支持，你会很快看到这种后果。

`Disposable` 类模板的实现。

```
template <class T>
class Disposable : public T {
  public:
    Disposable(T& t) {
        this->swap(t);
    }
    Disposable(const Disposable<T>& t) : T() {
        this->swap(const_cast<Disposable<T>&>(t));
    }
    Disposable<T>& operator=(const Disposable<T>& t) {
        this->swap(const_cast<Disposable<T>&>(t));
        return *this;
    }
};
```

这个类本身看起来内容并不多。它依赖于实现 `swap` 方法的模板参数，类中包含的任何资源在这里交换（希望以廉价的方式），而不是拷贝。构造函数和赋值运算符都使用它来移动没有副本的东西，有一点不同，这具体取决于传递来的内容。当从另一个构建一个 `Disposable` 实例时，我们以常引用接受参数，因为我们希望参数绑定到临时对象，这就是大多数一次性对象的用途。这迫使我们在内部使用 `const_cast`，当时需要调用 `swap` 并从一次性对象中获取资源。当从非一次性对象构建 `Disposable` 时，我们将其视为非常引用。这是为了防止我们自己触发不必要的破坏性转换，以及在我们想要有一个可用对象时却发现是个空壳。然而，这有一个缺点，我会很快提到它。

下面的代码显示了如何将 `Disposable` 改装为一个类，在这种情况下是 `Array`。

`Array` 类中 `Disposable` 类模板的使用。

```
Array::Array(const Disposable<Array>& from)
    : data_((Real*)(0)), n_(0) {
    swap(const_cast<Disposable<Array>&>(from));
}
Array& Array::operator=(const Disposable<Array>& from) {
    swap(const_cast<Disposable<Array>&>(from));
    return *this;
}
void Array::swap(Array& from) {
    data_.swap(from.data_);
    std::swap(n_, from.n_);
}
```

如你所见，我们需要添加一个构造函数和赋值运算符，它接受 `Disposable`（在 C++11 中，它们是移动构造函数和移动赋值运算符，接受 rvalue 引用），以及它们使用的 `swap` 方法。同样，构造函数通过常引用接受 `Disposable`，并在稍后将其强制转换，以便绑定到临时对象。我现在才想到，其实它们可以通过拷贝接受它，添加另一个廉价的交换。

最后，使用 `Disposable` 的方法是从函数返回它，如下面的代码：

```
Disposable<Array> ones(Size n) {
    Array result(n, 1.0);
    return result;
}
Array a = ones(10);
```

返回数组会导致它转换为 `Disposable`，并且赋值返回的对象会导致其内容与 `a` 互换。

现在，你可能还记得当我向你展示 `Disposable` 构造函数是安全的，并通过非常引用获取对象时，我谈到了一个缺点。它不能与临时对象结合，因此，上述函数不能简单地写成：

```
Disposable<Array> ones(Size n) {
    return Array(n, 1.0);
}
```

因为那不会编译成功。这迫使我们接受更冗长的路线，并给数组命名。[27]

当然，现在我们使用 rvalue 引用以及移动构造函数而忘记了上述所有内容。说实话，我有一种令人不安的怀疑，即 `Disposable` 可能会妨碍编译器且弊大于利。你是否知道编写上述代码的最佳方法并避免在现代 C++ 中的抽象惩罚？就是这个：

[27] 好吧，它实际上并没有强制我们，但是写出返回 `Disposable<Array>(Array(n, 10))` 比替代版本更丑陋。

```
Array ones(Size n) {
    return Array(n, 1.0);
}
Array a = ones(10);
```

在 C++17 中，返回数组和分配数组保证被省略时可能已经完成了拷贝（即，编译器将生成直接在我们分配的数组内构建返回数组的代码）。最近的编译器已经做了一段时间，没有等待标准绑定它们。这被称为 RVO（Return Value Optimization），并且使用 `Disposable` 可以阻止它，从而可能使代码变慢而不是更快。

13.7 设计模式

在 QuantLib 中实现了一些设计模式。你可以参考 Gang of Four 的书 (Gamma *et al*, 1995) 中对这些模式的描述。那我为什么要写这些呢？好吧，正如 G.K.Chesterton 所说，[28]

> [p]oets have been mysteriously silent on the subject of cheese
> （诗人们神秘兮兮地对奶酪三缄其口）

当你编写实际的实现时，Gang of Four 对于许多问题都保持沉默。请注意，他们没有过错。变化几乎是无限的，而他们只有四个人。

因此，我将使用最后一节来指出我们的实现，根据 QuantLib 的需求量身定制的几种模式。

13.7.1 观察者模式

在 QuantLib 库中使用观察者模式很普遍，你已经看到它在第 2 章 和第 3 章 中使用，让金融工具和期限结构追踪变化，并在需要时重新计算。

我们的模式版本（显示在下面的代码中）与 Gang of Four 一书中所描述的相近。但正如我所提到的那样，有些问题和难题没有在那里讨论过。

[28]译注：G.K.Chesterton，一位英国作家、文学评论家以及神学家。该句出自其文集《Alarms and Discursions》中的《Cheese》一文。

`Observable` 类和 `Observer` 类的概要。

```
class Observable {
    friend class Observer;
  public:
    void notifyObservers() {
        for (iterator i = observers_.begin();
             i != observers_.end();
             ++i) {
            try {
                (*i)->update();
            } catch (std::exception& e) {
                // store information for later
            }
        }
    }
  private:
    void registerObserver(Observer* o) {
        observers_.insert(o);
    }
    void unregisterObserver(Observer*);
    list<Observer*> observers_;
};
class Observer {
  public:
    virtual ~Observer() {
        for (iterator i = observables_.begin();
             i != observables_.end();
             ++i)
            (*i)->unregisterObserver(this);
    }
    void registerWith(const shared_ptr<Observable>& o) {
        o->registerObserver(this);
        observables_.insert(o);
    }
    void unregisterWith(const shared_ptr<Observable>&);
    virtual void update() = 0;
  private:
    list<shared_ptr<Observable>> observables_;
};
```

例如：我们应该在通知中包含哪些信息？在我们的实现中，我们追求极简主义——观察者所知道的一切就是发生了什么变化。可以提供更多信息（例如，通过使用 `update` 方法接受通知观察对象作为参数），以便观察者可以选择重新计算的内容，并保存一点儿计算，但我不认为这个函数值得增加复杂性。

另一个问题：如果观察者从其 `update` 方法引发异常会发生什么？当观察对象发送通知时会发生这种情况，即当观察对象迭代其观察者，并在每个观察者上调用 `update` 时。如果要传播异常，则循环将被中止，并且许多观察者将不会收到通知，这不好。我们的解决方案是，如果出现任何问题，则捕获任何此类异常，完成循环，并在最后抛出异常。这会导致原始异常丢失，这也不好。然而，我们觉得这是两害相权取其轻。

转到第三个问题：拷贝行为。当观察者或观察对象被拷贝时，应该发生什么并不是很清楚。目前，实现了一个合理的选择：一方面，拷贝一个观察对象带来没有任何观察者的副本；另一方面，拷贝观察者会导致副本注册与原始观察者相同的观察对象。但是，可能会考虑其他行为，事实上，正确的选择可能是完全禁止拷贝。

然而，最大的问题有两个。第一：我们显然必须确保观察者和观察对象的生命周期得到妥善管理，这意味着不应该向已经删除的对象发送通知。为此，我们让观察者存储了指向观察对象的共享指针，这确保在观察者完成之前不会删除任何可观察对象。观察者将在删除之前注销任何观察对象，这反过来使得观察对象可以安全地存储指向其观察者的原始指针列表。

但是，这只能保证在单线程设置中工作。我们正在将 QuantLib 绑定导出到 C# 和 Java，遗憾的是，垃圾收集器总在另一个线程忙着删除东西。每隔一段时间，这会导致随机崩溃，因为通知被发送到半删除的对象。一旦理解了这个问题，就可以修复（嗨，Klaus[29]）。但是，修复程序会降低代码的速度，因此默认情况下它处于非活动状态，并且可以通过编译开关启用。如果需要 C# 或 Java 绑定，请使用它。

就像 Jerry Lee Lewis 的歌[30]一样，第二个大问题是[31] “there’s a whole lotta notifyin’ going on”（有很多通知会继续发生）。更改日期很容易触发数十或数百个通知。即使大多数 `update` 方法只设置了一个标志并传递了调用，运行时间也会增加。

在计算时间至关重要的应用程序（例如 CVA/XVA）中使用 QuantLib 的人（嗨，Peter[32]）通过禁用通知，并且显式地重新计算来解决此问题。减少通知时间的一步是取消中间人，缩短通知链。然而，由于链条中普遍存在 `Handle` 类，这是不可能的。句柄可以重新链接，因此即使在构建对象之后，依赖链也可以改变。

简而言之，问题仍未解决。如果你有任何好主意，你知道在哪里可以找到我们。

13.7.2 单体模式

Gang of Four 将他们书中的第一部分用于创建型模式。虽然逻辑上合理，但这种选择结果却产生了不幸的副作用：时常是过度热心的程序员开始阅读这本书，并适当地将他们的代码用抽象工厂和单体来表达。毋庸置疑，这不仅仅是为了清晰的代码。

你可能怀疑在 QuantLib 中存在 `Singleton` 类模板的原因是相同的。（我可能要加一句，这是非常恶意地揣测，这是你的不是了。）幸运的是，我们可以根据版本控制日志进行辩护，比起观察者模式（一种行为型模式）或组合模式（一种结构型模式），这类被添加到库中稍晚一些。

我们的默认实现如下所示。

[29]译注：Klaus Spanderen，QuantLib 的一位重要贡献者，同时也是一位金融工程领域的专家。
[30]译注：Jerry Lee Lewis，是一位美国创作型歌手、音乐人和钢琴手，这里指的是他的歌《Whole Lotta Shakin’ Goin On》。
[31]注意，我没说曾经是。
[32]译注：Peter Caspers，QuantLib 的一位重要贡献者，同时也是一位金融工程领域的专家。

Singleton 类模板的接口。

```
template <class T>
class Singleton : private noncopyable {
  public:
    static T& instance();
  protected:
    Singleton();
};
#if defined(QL_ENABLE_SESSIONS)
// the definition must be provided by the user
Integer sessionId();
#endif
template <class T>
T& Singleton<T>::instance() {
    static map<Integer, shared_ptr<T>> instances_;
#if defined(QL_ENABLE_SESSIONS)
    Integer id = sessionId();
#else
    Integer id = 0;
#endif
    shared_ptr<T>& instance = instances_[id];
    if (!instance)
        instance = shared_ptr<T>(new T);
    return *instance;
}
```

它基于我在第 7 章中描述的奇异递归模板模式,要成为一个单体,C 类需要继承 Singleton<C>，提供一个接受参数的私有构造函数，并使 Singleton<C> 成为友元以便可以使用它。你可以在 Settings 类中看到一个例子。

正如 Scott Meyers (Meyers, 2005) 所建议的那样, 持有实例的映射（请耐心）被定义为 instance 方法中的静态变量。这可以防止所谓的静态初始化命令失败，其中变量在被定义之前使用，而在 C++11 中，它还有额外的保证，即初始化是线程安全的（尽管这不是我们看到的整个故事）。

现在，你可能会有几个问题。例如，为什么是一个实例的映射，这本应该是一个单体？嗯，这是因为拥有单个实例可能会受到限制。举个实例（没有双关语的意思），你可能希望在不同的估值日期执行同步计算。因此，我们试图通过在每个线程中允许一个 Singleton 实例来缓解这个问题。这是通过编译标志启用的，并使 instance 方法使用 #if 分支，在该分支中，它从 sessionId 函数获取 id，并使用它来索引到映射中。如果在每个线程上启用单体，则还必须提供一个函数，它可能会像这样

```
Integer sessionId() {
    return /* `some unique thread id from your system API` */ ;
}
```

在其中，你将使用操作系统（或某些线程库）提供的函数来识别线程，将标识符转换为唯一的整数，然后将其返回。反过来，这将导致 instance 方法为每个线程返回一个唯一的实

例。如果你不启用该功能，则 `id` 将始终为 0，并且你将始终获得相同的实例。在这种情况下，你可能根本不想使用线程，在另一种情况下，你显然可以这样做，但无论如何你必须小心：请参阅有关全局配置 部分中的讨论。

你可能也会问自己为什么我说这是我们的默认实现。好问题。还有其他一些实现，我不会在这里显示，并且由许多编译标志启用。一方面，事实证明，当在 .NET 框架下编译为托管 C++ 代码时，静态变量实现不起作用（至少在较旧的 Visual Studio 编译器上），因此在这种情况下我们将实现改成映射是一个静态类变量。另一方面，如果要在多线程设置中使用全局 `Singleton` 实例，则需要确保 `Singleton` 实例的初始化是线程安全的（我在谈论实例本身，而不是包含它的映射，它是执行 `new T` 的地方）。这需要锁、互斥和填充，我们不希望在默认的单线程设置中接近任何一个。因此，该代码位于另一个编译标志之后。如果你有兴趣，可以在 QuantLib 中查看。

你的最后一个问题可能是我们是否应该有一个 `Singleton` 类，而且这是一个严格意义上的。我再次建议你回到之前关于全局配置的讨论。在这个时候，就像温斯顿·丘吉尔所说的民主一样，除了所有其他方案之外，它似乎是最糟糕的解决方案。[33]

13.7.3 访问者模式

我们的实现（如下面的代码所示）遵循非循环访问者模式（Acyclic Visitor pattern）(Martin, 1997)，而不是 Gang of Four 书中的：我们定义了一个退化的 `AcyclicVisitor` 类用于接口，以及一个类模板 `Visitor` 为其模板参数定义纯虚的 `visit` 方法。

`AcyclicVisitor` 类和 `Visitor` 类模板的接口。

```
class AcyclicVisitor {
  public:
    virtual ~AcyclicVisitor() {}
};
template <class T>
class Visitor {
  public:
    virtual ~Visitor() {}
    virtual void visit(T&) = 0;
};
```

该模式还需要任何我们想要访问的类层次结构的支持，一个例子显示在下面的代码中。

[33]译注：出自《丘吉尔自传》：“Many forms of Government have been tried, and will be tried in this world of sin and woe. No one pretends that democracy is perfect or all-wise. Indeed it has been said that **democracy is the worst form of Government except for all those other forms that have been tried from time to time**.”

访问者模式在一个类层次结构中的实现。

```
void Event::accept(AcyclicVisitor& v) {
    Visitor<Event>* v1 = dynamic_cast<Visitor<Event>*>(&v);
    if (v1 != 0)
        v1->visit(*this);
    else
        QL_FAIL("not an event visitor");
}
void CashFlow::accept(AcyclicVisitor& v) {
    Visitor<CashFlow>* v1 =
        dynamic_cast<Visitor<CashFlow>*>(&v);
    if (v1 != 0)
        v1->visit(*this);
    else
        Event::accept(v);
}
void Coupon::accept(AcyclicVisitor& v) {
    Visitor<Coupon>* v1 = dynamic_cast<Visitor<Coupon>*>(&v);
    if (v1 != 0)
        v1->visit(*this);
    else
        CashFlow::accept(v);
}
```

层次结构中的每个类（或者至少是我们希望特别可访问的类）需要定义一个 accept 方法接受 AcyclicVisitor 的引用。每个方法都尝试将传递来的访问者转换为相应类的特定 Visitor 实例。成功的强制转换意味着访问者定义了一个接受这个特定类的 visit 方法，以便我们调用它。失败的强制转换意味着我们必须寻找回退。如果类不是层次结构的根（如代码中的 CashFlow 或 Coupon），我们可以调用 accept 的基类实现，这将尝试强制转换。如果我们处于根部，就像 Event 类一样，我们没有进一步的回退，我们引发异常。[34]

最后，一个访问者在第 4 章 中被实现为 BPSCalculator 类。它继承自 AcyclicVisitor，因此它可以被传递给各种 accept 方法，以及来自 Visitor 模板的实例化，其中每个类都将提供一个 visit 方法。它将被传递给某个实例的 accept 方法，该方法最终将调用其中一个 visit 方法或引发异常。

我已经在第 4 章 讨论了访问者模式的用处，所以我建议你回到那里（那取决于一份意料中的总结）。因此，我只会谈谈为什么选择非循环访问者模式。

简而言之，Gang of Four 版本的访问者可能会更快一点，但它更具侵入性。特别是，每次你向可访问的层次结构添加一个新类时，你也被迫去为每个现有的访问者添加相应的 visit 方法（其中“被迫”的意思是你的代码不能编译，如果你没有添加的话）。使用非循环访问者模式，你不需要这样做。你的新类中的 accept 方法将使强制转换失败，并回退到它的基类。[35]请注意，这很方便，但不一定是好事（就像生活中的很多事情，我可能会补充）：无论如何，你应该检查现有的访问者，检查回退的实现是否对你的类有意义，如果没有则添加

[34]另一个替代方案是什么都不做，但我们倾向于不要无声无息地失败。
[35]实际上，你甚至不需要定义 accept 方法，你可以继承它。但是，这会阻止访问者定位这个特定的类。

特定的实现。但是我认为没有编译器警告你的弊端抵不上不必为每个现金流类编写 `visit` 方法带来的好处，就像在 `BPSCalculator` 中那样，一对 `visit` 方法就足够了。

14. 附录 B：编码约定

每个程序员团队都有许多在编写代码时使用的约定。无论有哪些惯例（存在几个惯例，为无数次战争提供了方便的 Casus Belli[1]），重要的是它们必须被严格遵守，[2]因为它们使代码更容易理解。熟悉其用法的读者可以一目了然地区分宏和函数，或者在变量和类型名称之间进行区分。

以下代码简要说明了 QuantLib 中使用的约定。根据 Sutter 和 Alexandrescu 在 2004 年的建议 (Sutter and Alexandrescu, 2004)，我们试图将其数目降至最低，仅强制执行那些提高可读性的约定。[3]

QuantLib 编码约定示例。

```
#define SOME_MACRO
typedef double SomeType;
class SomeClass {
  public:
    typedef Real* iterator;
    typedef const Real* const_iterator;
};
class AnotherClass {
  public:
    void method();
    Real anotherMethod(Real x, Real y) const;
    Real member() const;
    // getter, no "get"
    void setMember(Real);    // setter
  private:
    Real member_;
    Integer anotherMember_;
};
struct SomeStruct {
    Real foo;
    Integer bar;
};
Size someFunction(Real parameter,
                  Real anotherParameter) {
    Real localVariable = 0.0;
    if (condition) {
        localVariable += 3.14159;
    } else {
        localVariable -= 2.71828;
```

[1]译注：Casus Belli，拉丁语，意思是“开战的借口”。

[2]然而，QuantLib 的开发者是人类。照此，他们有时候不能遵循我所描述的规则。

[3]译注：`return 42;` 的典故出自《银河系漫游指南》，42 是“生命、宇宙以及任何事情的终极答案”，本书的示例代码中多次出现这个数字。

```
    }
    return 42;
}
```

宏全部为大写，单词用下划线分隔。类型名称以大写字母开头，属于所谓的驼峰式写法，单词连接在一起，每个单词的首字母大写。这适用于类型声明，例如 `SomeType`，以及类的名称，例如 `SomeClass` 和 `AnotherClass`。但是，对模仿 C++ 标准库中的类型声明做了例外，这可以在 `SomeClass` 中的两种迭代器类型的声明中看到。内部类可能会出现相同的例外。

关于其他所有内容（变量、函数和方法名称以及参数）都是驼峰式的，并以小写字符开头。类的数据成员遵循相同的约定，但是以下划线结束，这使得更容易将它们与方法体中的局部变量区分开来（通常对公有数据成员进行例外处理，特别是在结构体或类似结构体的类中）。在方法中，进一步的约定用于 getter 和 setter。通过向成员名称添加前导词 `set`，并删除结尾的下划线来创建 setter 的名称。getter 的名称等于返回的数据成员的名称，没有结尾的下划线，没有添加前导词 `get`。这些约定在 `AnotherClass` 和 `SomeStruct` 中举例说明。

一个不那么严格的约定，在函数声明，`if`、`else`、`for`、`while` 或 `do` 关键字之后的左括号与前面的声明或关键字要在同一行，这在 `someFunction` 中显示。此外，`else` 关键字与前面的右括号位于同一行，这同样适用于结束 `do` 语句的 `while`。然而，这更多的是品味而非可读性。因此，如果开发人员无法忍受这种惯例，他们可以自由使用他们自己的约定。所示函数的例子说明了另一个旨在提高可读性的约定，即如果函数和方法参数不适合单行显示，则它们应垂直对齐。

最后但同样重要的是，代码中应该没有真正的制表符（tab），应该使用四个空格。如果你决定在这个集合中只保留一个约定（不过，它只是一个修辞手段），请保留这个。如果不这样做，很可能会破坏在另一个编辑器中查看代码的开发人员的缩进。无论你的编辑器是什么，它都可以配置（你是个程序员，对吧?），这样就可以强制执行这个约定，而不需要你做任何有意识的努力。

15. QuantLib license

該內容沒有包含在樣本書中。您可以在 Leanpub 上購買這本書，網址為 http://leanpub.com/implementingquantlib-cn

16. 参考文献

D.Abrahams, *Want Speed? Pass by Value*. In *C++ Next*, 2009.

D.Adams, *So Long, and Thanks for all the Fish*. 1984.

A.Alexandrescu, *Move Constructors*. In *C/C++ Users Journal*, February 2003.

F.Ametrano and M.Bianchetti, *Everything You Always Wanted to Know About Multiple Interest Rate Curve Bootstrapping but Were Afraid to Ask*. SSRN working papers series n.2219548, 2013.

J.Barton and L.R.Nackman, *Dimensional Analysis*. In *C++ Report*, January 1995.

T.Becker, *On the Tension Between Object-Oriented and Generic Programming in C++*. 2007.

Boost C++ libraries.http://boost.org.

M.K.Bowen and R.Smith, Derivative formulae and errors for non-uniformly spaced points. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 461, pages 1975–1997. The Royal Society, 2005.

D.Brigo and F.Mercurio, *Interest Rate Models —Theory and Practice*, 2nd edition. Springer, 2006.

Mel Brooks (director), *Young Frankenstein*. Twentieth Century Fox, 1974.

W.E.Brown, *Toward Opaque Typedefs for C++1Y, v2*. C++ Standards Committee Paper N3741, 2013.

L.Carroll, *The Hunting of the Snark*. 1876.

G.K.Chesterton, *Alarms and Discursions*. 1910.

M.P.Cline, G.Lomow and M.Girou, *C++ FAQs*, 2nd edition. Addison-Wesley, 1998.

J.O.Coplien, *A Curiously Recurring Template Pattern*. In S.B.Lippman, editor, *C++ Gems*. Cambridge University Press, 1996.

C.S.L.de Graaf. *Finite Difference Methods in Derivatives Pricing under Stochastic Volatility Models*. Master's thesis, Mathematisch Instituut, Universiteit Leiden, 2012.

C.Dickens, *Great Expectations*. 1860.

P.Dimov, H.E.Hinnant and D.Abrahams, *The Forwarding Problem: Arguments*. C++ Standards

Committee Paper N1385, 2002.

M.Dindal (director), *The Emperor's New Groove*. Walt Disney Pictures, 2000.

D.J.Duffy, *Finite Difference Methods in Financial Engineering: A Partial Differential Equation Approach*. John Wiley and Sons, 2006.

M.Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd edition. Addison-Wesley, 2003.

M.Fowler, *Fluent Interface*. 2005.

M.Fowler, K.Beck, J.Brant, W.Opdyke and D.Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

E.Gamma, R.Helm, R.Johnson and J.Vlissides, *Design Patterns: Element of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

P.Glasserman, *Monte Carlo Methods in Financial Engineering*. Springer, 2003.

D.Gregor, *A Brief Introduction to Variadic Templates*. C++ Standards Committee Paper N2087, 2006.

H.E.Hinnant, B.Stroustrup and B.Kozicki, *A Brief Introduction to Rvalue References*. C++ Standards Committee Paper N2027, 2006.

A.Hunt and D.Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

International Standards Organization, *Programming Languages —C++*. International Standard

ISO/IEC 14882:2014.

International Swaps and Derivatives Associations, *Financial products Markup Language*.

P.Jäckel, *Monte Carlo Methods in Finance*. John Wiley and Sons, 2002.

J.Kerievsky, *Refactoring to Patterns*. Addison-Wesley, 2004.

R.Kleiser (director), *Grease*. Paramount Pictures, 1978.

H.P.Lovecraft, *The Call of Cthulhu*. 1928.

R.C.Martin, *Acyclic Visitor*. In *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

S.Meyers, *Effective C++*, 3rd edition. Addison-Wesley, 2005.

N.C.Myers, *Traits: a new and useful template technique*. In The C++ Report, June 1995.

G.Orwell, *Animal Farm*. 1945.

W.H.Press, S.A.Teukolsky, W.T.Vetterling and B.P.Flannery, *Numerical Recipes in C*, 2nd edition. Cambridge University Press, 1992.

QuantLib. http://quantlib.org.

E.Queen, *The Roman Hat Mystery*. 1929.

V.Simonis and R.Weiss, *Exploring Template Template Parameters*. In *Perspectives of System Informatics*, number 2244 in Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001.

B.Stroustrup, *The C++ Programming Language*, 4th edition. Addison-Wesley, 2013.

H.Sutter, *You don't know const and mutable*. In Sutter's Mill, 2013.

H.Sutter and A.Alexandrescu, *C++ Coding Standards*. Addison-Wesley, 2004.

T.Veldhuizen, *Techniques for Scientific C++*. Indiana University Computer Science Technical Report TR542, 2000.

H.G.Wells, *The Shape of Things to Come*. 1933.

P.G.Wodehouse, *My Man Jeeves*. 1919.