Rambling On Design Patterns

漫谈设计模式

Co0der(库德) 著

网络书籍

前言

OOP（Object-Oriented Programming）早已不是一个新概念了，OOP 在最近的 20 多年里发展得异常迅猛，特别是最近的 10 多年里，OOP 相关技术层出不穷，当大家热衷于使用这些新技术时，却不会使用 OOP 进行软件设计，新的技术并没有为大家带来任何好处。

很多老的开发人员从过程式开发转向面向对象的开发过程中，由于他们习惯过程式思维的开发，尽管他们使用的是 OOP 语言，但这并没有给他们带来太多帮助，反而使他们更加厌倦 OOP 的软件开发，认为 OOP 没有想象中的那么便捷，很多地方没有使用过程式开发来的便捷，于是他们又退化为过程式的开发。

越来越多的新开发人员也加入了 OOP 的潮流，他们追求新的技术，学会使用各种工具和框架，却无暇顾及 OOP 进行开发设计的核心。虽然使用了新技术，代码的质量并未提高，反而事与愿违。

当他们沉浸在新技术的使用和业务逻辑的编码实现时，未料到这些拙劣的设计导致了他们的代码不易阅读，不易维护，不易扩展，不易测试，不易调试……。大家忙忙碌碌，但是项目进度缓慢，最终往往以失败而告终。归根结底，尽管使用了 OOP 语言开发和新的技术，但对 OOP 只限于粗浅的了解和相关语言语法使用上的理解，他们并不会真正使用 OOP 进行开发设计，以致使用时颠三倒四，未能真正享受到 OOP 和这些新技术带来的好处，有些新技术非但没有提供帮助，反而成为某些软件失败的罪魁祸首。那么，如何使用 OOP 进行开发设计呢？

OOP 开发新手由于没有这方面的设计经验，在遇到问题时，往往诉求于逻辑的实现，在维护性和扩展性没有考虑或者少有考虑，导致代码却乱七八糟，七零八散，随着开发的深入，最终在用户各种各样的需求面前无以应对。而有经验的 OOP 开发人员会灵活使用各种模式作出优秀的设计，编写的代码健壮性高，易于阅读、维护和扩展，可伸缩性强，开发成本也十分低廉。如果重用他们的开发经验，那么你就不需要在相同问题上重蹈覆辙，也能设计出优秀的软件。

市面上介绍设计模式的书籍非常多，它们一般仅仅给出 GoF 的 23 个最基本的设计模式的定义和一些简单的示例，大多数读者凭此充其量只能了解它们，在使用上大打折扣。本书精心筛选了一些我们经常在开发设计过程中使用到的模式，使用 OOP 的眼光分析它们，适时结合一些流行 J2EE 框架和技术，并从横向和纵向两方面扩展读者的思维，使读者对这些常用的模式有一个全面深刻的认识，也希望能够为正在使用这些框架和技术的读者带来帮助。

本书内容

开发人员之间交流最快莫过于代码了，本书给出了大量代码片段，在一些重要的地方使用黑体加粗的字体，并作了详细解释，希望能够抛砖引玉，帮助读者能够制作出更加出色的代码。另外，本书还添加了很多图片，希望图文并茂，使这本书更加容易阅读。

本书主要分为五篇：

- **第一篇：模式介绍**

  第一章讲述了面向对象与模式之间的关系和模式的简史；随后在第二章介绍了第一个简单的模式，模板方法模式，在这章，我们分析了代码重复所导致的“腐臭气味”，重复的代码是代码“臭味”中最糟糕的，在以后章节我们将会介绍各种模式来避免代码重复。

- **第二篇：创建对象**

  使用 OOP 语言的语法创建一个对象并不复杂，例如在 Java 语言中使用 new 即可。但是随着系统变得越来越复杂，使用 new 直接创建对象会给系统造成很高的耦合度。使用创建模式可以封装对象实例化的过程，把使用对象的功能和实例化对象解耦开来，从而降低了耦合度。

  在本篇最后，讨论了现在最流行的两个概念，IoC和 DI，这二者是目前流行的轻量级容器的基础。

- **第三篇：构建复杂结构**

  有时候，创建新的，更强功能的类并不需要重新编写代码，装配已有的类和对象要来地更加快捷，也更加灵活。该篇讨论了一些常用的组装对象的模式，你将发现，构建大的，功能更强的对象，不是只有多层继承才能实现，组合往往

是最有效的方式，本篇你将会看到如何使用继承和组合创建复杂的大结构。

- **第四篇：行为模式**

  我们在程序中经常需要封装一些对象的行为或者对象之间的通信，这篇将会讲述三个常用的行为模式：策略模式，状态模式和 观察者模式，加上第二章介绍的 模板方法模式，本书将一共讲述这四个常用的行为模式。

- **第五篇：终点还是起点**

  其实，在很早之前，就有人对面向对象思想的局限性就行了研究，提出了一种新的编程方法AOP，第 15 章将会介绍AOP的概念及其实现技术。

  尽管在前面篇章，我们学习了一些常用的模式和一些OOP设计的原则，并且了解了AOP方面的一些知识，但这些并不能表示我们已具备解决复杂领域问题的能力，在 第 16 章我们讨论了在实际开发过程中如何使用面向对象进行开发设计以及应该注意的一些问题。

  在本篇末尾，第 17 章，将回顾本书的内容，重新认识OOP设计范式。

本书读者

本书不是一本面向对象和 Java 语言的入门书籍，阅读对象主要是从事 Java 语言的软件开发人员。

希望读者了解这些 Java 基本技术：反射，内部类，序列化，线程，动态代理，垃圾回收，类加载器以及 Java 对象的引用类型等等。

本书中会使用到 UML 的一些图示，主要包括静态类图和时序图等。

希望读者了解或者使用过这些框架和技术：JDBC，Hibernate，Jpa，Spring，Ejb，Pico Container，Guice，XWork，Webwork，Struts和 EasyMock等等。由于在实际的J2EE开发过程中，我们经常使用到这些框架，笔者将在讲述过程中适时结合这些框架与技术，并会比较详细地介绍它们，用以消除因不理解这些技术而造成对模式理解上

产生问题。如果读者还想做进一步的了解，可以参看 附录A我给大家的相关推荐书籍和网站。

本书代码

要获取本书的示例代码，请登录 http://code.google.com/p/rambling-on-patterns/，点击进入Downloads标签页，选择最新的版本下载。也可以安装svn客户端下载文件最新文件，svn地址为：http://rambling-on-patterns.googlecode.com/svn/trunk/。为了能够让读者选择学习自己喜欢的章节并分别单独运行这些示例，作者尽量为每一种模式提供了相对独立和完整的代码。

注意，这些示例代码的运行环境是**Java 2 Platform Standard Edition SDK V6.0 及以上版本**，代码使用了一些Java 5 及以上的新语法和JDK的最新API，如果读者对它们还不熟悉，查阅 附录A的有关推荐书籍。

反馈

尽管笔者尽最大努力去避免正文和代码中出现的任何错误，但是人无完人，难免有纰漏错误之处。如果读者在阅读过程中发现拼写错误，代码错误，以及内容有混淆之处，希望能够及时反馈，或许它们能够节省其他读者很多宝贵时间，也有利于完善本书，反馈邮箱是：ramblingonpatterns@gmail.com。

写后感

为了完成此书，笔者增删了 5 次，尽管艰辛，但在写书的过程中也发现了不少乐趣，希望为读者在阅读过程中带来乐趣。

版权

本书目前是一本网络书籍，还没有打印版本，本书版权和示例代码版权是不相同的，代码版权遵守 Apache License2.0，而本书版权如下：

1. 任何人均可以在计算机，掌上设备，或其他电子设备上阅读该书。
2. 目前由于该书未出打印版本，任何人不得打印该书并以纸质形式进行传播。
3. 未经笔者的许可，任何人均不可以出版发行该书的纸版本。
4. 任何人都可以在 BBS，blog，或其他媒介上引用该书，但引用时必须注明出处。

目录

第一篇 模式介绍

模式被引入软件开发和 OOP 语言的流行是分不开的，在 80 年代末至 90 年代初，面向对象软件设计逐渐成熟，被计算机界广泛理解和接受，然而专业开发人员和非专业开发人员作出的设计差异巨大，为了让 OOP 开发人员能够使用 OOP 设计进行专业开发，经过多年的不懈努力，最终由 GoF 四人根据当时的一些成熟经验和解决方案归纳出了 23 条最基本的设计模式，以供 OOP 开发人员学习和使用。

该篇首先从面向对象谈起，回顾一下面向对象的一些基础概念，讲述OOP开发设计带来的好处，以及为什么要学习和使用设计模式。接着在第二章将为读者讲述第一个最简单也最容易理解的模式——模板方法（Template Method）模式。

第1章　谈面向对象和模式

1.1　什么是对象

也许你已经使用面向对象做开发好几年了，面向对象的概念也似乎不难理解，但是很多人并未认识面向对象编程的本质，继续偏向于使用 PP/FP（Procedural Programming/ Functional Programming）方式编程。在讲述模式之前，我们先回顾一下什么是面向对象。

在 OOP 世界里，任何事物，不管是无形的，还是有形的，都是对象。对象是包含一些行为和属性的一种组合体，它反映的是客观世界的任何事物。比方说，马有腿，耳朵和嘴巴等属性，它们会跑，也会嘶叫，这些是它们的行为。每个对象都归属于某一特定的类型，比如说，一匹汗血宝马的类型是马。

面向对象的语言一般都有三个基本特征：

- **封装**

  封装是面向对象最重要的特征之一，封装就是指隐藏。

  对象隐藏了数据（例如 Java 语言里 private 属性），避免了其他对象可以直接使用对象属性而造成程序之间的过度依赖，也可以阻止其他对象随意修改对象内部数据而引起对象状态的不一致。

  对象隐藏了实现细节：

  - 使用者只能使用公有的方法而不能使用那些受保护的或者私有的方法，你可以随意修改这些非公有的方法而不会影响使用者；
  - 可以隐藏具体类型，使用者不必知道对象真正的类型就可以使用它们（依赖于接口和抽象带来的好处）。
  - 使用者不需要知道与被使用者有关和使用者无关的那些对象，减少了耦合。

  只能通过公用接口和方法使用它们。这样，由于客户程序就不能使用那些受保护的方法（例如 Java 语言里的 private 方法和 protected 方法），而你可以随意修

改这些方法，并不会影响使用者，从而降低了耦合度。

- **继承**

  继承可以使不同类的对象具有相同的行为：为了使用其他类的方法，我们没有必要重新编写这些旧方法，只要这个类（子类）继承包含那些方法的类（父类）即可。从下往上看，继承可以重用父类的功能；从上往下看，继承可以扩展父类的功能。

- **多态**

  多态可以使我们以相同的方式处理不同类型的对象：我们可以使用同一段代码处理不同类型的对象，只要它们继承/实现了相同的类型。这样，我们没有必要为每一种类型的对象撰写相同的逻辑，极大地提高了代码重用程度。

1.2 面向对象的模块化

1.3 面向对象的好处

面向对象有很多优势，我们在这里总结了以下内容：

- 对象易于理解和抽象，例如马是一个类，一匹马是一个对象，跑是马的行为。正是由于这个特性，我们很容易把客观世界反映到计算机里，极大地方便了编程设计。

- 对象的粒度更大，模块化程度也更高：与方法（函数）和结构体相比，对象是一组方法和数据的单元，所以粒度更大，这样更方便控制和使用；而模块化程度越高，也越容易抽象。

- 更加容易重用代码：只要使用继承，就可以拥有父类的方法；只要创建这个对象，就可以使用它们的公有属性和方法；只要使用多态，就可以使用相同的逻辑处理不同类型的对象。

- 具有可扩充性和开放性：OOP 天生就具有扩展性和开放性。

- 代码易于阅读：阅读人员在阅读代码过程中，可以不去关注那些具体实现类，只要关注接口的约定即可，这样更容易侧重重点。

- 易于测试和调试：由于代码易于阅读，也方便测试；并且，由于模块化和抽象化程度高，越容易发现问题出在哪个模块，也就易于跟踪和调试；最后，测试过程中，有些对象只有在软件交付运行时才能使用（例如一个发送彩信的服务对象），由于对象可依赖于抽象/接口，我们在运行测试时可以使用假对象（Mock

对象）替换这些依赖的对象，使之不影响被测试的对象，这样便减少了测试的依赖程度。

- 代码容易维护：基于以上各种好处，不难想象代码会变得更加容易维护。

1.4 重用

回顾软件开发的历程，我们经历了从最初的二进制编程，然后到汇编语言的编程，最后到高级语言编程的过程，在这个过程中，我们不断地提高了语言指令的模块化：汇编语言模块化了机器指令，一条汇编指令可能是多条机器指令的组合；高级语言进一步模块化了汇编语言（例如 for 循环等）。**这样，使用高级语言编程的过程，其实就是重用这些大结构的模块指令的过程，效率得到大大提升**。它们解决了语言到机器的映射问题，却没有针对我们要解决的问题域（Problem domain/Problem space）思考问题，于是对问题域进行建模开始发展。

我们使用结构体和方法来建立模型，这些结构体粒度小，抽象程度不高，可重用程度也不高。随后出现了面向对象的编程，对象是方法和结构体的集合，抽象程度更高，我们可以通过重用对象功能协作解决更复杂的问题。随着计算机的发展，一个系统的代码也由原来的几百行增加至现在的动辄上百万行，如果从头到尾编写这些代码，那不仅是软件开发人员的噩梦，而是整个软件行业乃至所有使用软件的行业的噩梦。于是，**重用从方法、结构体、类的重用，上升到软件的重用**。

然而要使用OOP来设计可重用的软件并不容易，尽管面向对象的编程有如此之多的好处，但仍然有很多人倾向于使用PP/FP（Procedural Programming/ Functional Programming）进行编程，他们列举很多使用OOP的反面例子，例如最常见的一个是关于使用switch语句的例子，他们认为使用switch语句非常直观和简洁，而如果使用OOP的 状态模式（State Pattern）替换，便会产生了了大量文件和代码。其实这个驳斥只是溢于表面，没看到状态模式带来的易阅读易扩展等好处。另外，他们也喜欢列举继承所带来的坏处来证明使用OOP进行开发还不如使用PP/FP，其实在OOP开发过程中，我们更愿意使用 合成而非继承（Favor Composition over Inheritance），这些不足正是我们在OOP编程过程中应该避免的。

从种种问题来看，其原因归根结底是由于很多人理解了一些面向对象的基本概念和 OOP 相关语言的语法，却还没领会和掌握面向对象的开发设计方法。在开发设计过

程中，往往求助于以前使用的过程式开发设计和简单的面向对象的特征（例如继承性），因而并未享受面向对象编程所带来的模块化的好处，粗粒度的好处，封装的好处，易于抽象的好处，代码重用的好处……。

如何使用面向对象的编程方法进行软件开发便成了该问题的关键。其实，有经验的 OOP 开发人员并不会从头解决问题，他们往往会使用之前的成熟的解决方案来解决类似的问题。如果能够重用他们的经验和思想开发设计软件，那就好比站在前人的肩膀上解决问题，**模式能够让我们从思想上重用有经验的开发人员的解决方案来解决问题**。

我们可以看到，对于软件编程，重用非常重要。经过过去几十年的发展，重用包括指令集的重用，方法的重用，代码的重用，服务的重用，软件的重用，设计的重用，思想的重用等等，重用是指一切知识信息的重用。要实现重用并不那么简单，软件发展至今，开发人员一直在重用上摸索着前进，终于迈出了一步。笔者从开始学习编码至今，一直致力于把重用应用于软件开发，本书将会和读者一起探讨重用和如何编写可重用的软件。

1.5 模式简史

现在，模式被广泛地运用到软件设计和其他各个领域中，然而，**模式（Pattern）**这一词却最先在建筑学里被建筑师Christopher Alexander引入。在上个世纪 70 年代，Christopher Alexander等人写了一系列书籍描述建筑学上的模式，其中一本名为《A Pattern Language: Towns，Buildings，Construction》[1]的书中讲述了建筑领域的 253 个模式，并为模式的作出了定义，指出这些模式并不会随着时间消逝而褪色。

到 1987 年，Kent Beck 和Ward Cunningham开始尝试把模式设计引入编程世界，经过几年坚持不懈的努力和尝试，到了 1994 年，由 Erich Gamma，Richard Helm，Ralph Johnson and John Vlissides四人编写的书籍**《Design Patterns: Elements of Reusable Object-Oriented Software》**面世，这本书在后来设计模式学习和研究中影响非常广泛，是一本经典的设计模式参考书籍。在此书中，他们总结了多年来软件开发人员的实践经验和研究成果，收编了 23 个最常用的设计模式。时至今日，这 23 个设计

1 参见 Christopher Alexander, Sara Ishikawa and Murray Silverstein 编著的《A Pattern Language: Towns, Buildings, Construction》一书，出版社为 Oxford University Press，出版时间为 1977。他们为建筑学上的模式出版了一系列图书，这本书籍是其中的卷二，还有一本大家非常熟悉的书籍《The Timeless Way of Building》，是卷一。

模式仍然是最基本，最经典的模式，而这四个人往往被称为“Gang of Four（四人帮，其实是一种开玩笑的戏称）”，简称为“GoF”。

由于上世纪 80 年代，面向对象技术蓬勃发展，GoF 提出的这些模式都来自于面向对象开发设计的经验，即这些模式都是有关面向对象开发设计的。由于面向对象的强大生命力，随着越来越多的人加入面向对象的大军，面向对象的设计模式也得到极大地推广和发展，涌现出了很多出色的新模式。

1.6 什么是模式

我们刚才简要地介绍了模式的发展历史，但是，什么是模式呢？建筑师 Christopher Alexander 给出这样的定义：

> **Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.**
>
> **每一个模式都描述了一个在我们周围不断发生的问题，以及该问题的解决方案的核心，这样，你就能够无数次地使用该解决方案而不用按照同样的方式重做一遍。**

这个定义较长，一般情况下，模式被简单地定义为如下：

> **A pattern is a solution to a problem in a context.**
>
> **模式是某一上下文环境中一个问题的解决方案。**

是的，模式就是一个解决方案，一个模式解决了一类特定的问题，当我们再次遇到同样的问题时，我们仍然可以使用它解决同样的问题。

这个定义很容易方便初学者认识什么是模式，但是很多人对此定义提出了异议，在《Head First Design Patterns》[2]一书第 13 章中，作者就列举了一个例子驳斥了该概念：

[2] 参见 Eric T Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra 等人编写的《Head First Design Patterns》一书，出版社为 O'Reilly Media,，出版日期为 October 2004。

While an absent-minded person might lock his keys in the car often, breaking the car window doesn't qualify as a solution that can be applied over and over (or at least isn't likely to if we balance the goal with another constraint: cost).

一个健忘的人或许经常会把钥匙锁在汽车里，打破汽车的窗玻璃不是一个能够经常使用的解决方案（至少如果我们权衡另外一个限制条件——花费时，也极不可能使用）。

是的，不是任何一个解决方案都能成为模式。在 Christopher Alexander 给出的定义里，就明确指出了，如果一个解决方案称得上是模式，那它要能够被使用无数次，经得起时间的考验。GoF 为模式给出的定义如下所示：

The design patterns … are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

设计模式……描述了在一个特定上下文里，如何定制这些互相通信的对象和类来解决一个常见设计问题。

可以看出，GoF 给出的定义更加侧重于 OOP 编程设计。其实，我更愿意使用如下定义：

A pattern is a *general* solution to a problem in a context.

模式是某一上下文环境中一个问题的*常用*解决方案。

模式是一个**常用**的解决方案（**general solution**），而非仅仅是一个 **solution**，显然，打碎车窗玻璃并不能被看作是一个可以经常使用的解决方案。这个定义可以帮助读者简单但不失深刻地理解模式含义。

为了方便这 23 个经典模式的交流和传播，GoF 在书中为每个模式定义了 4 个基本要素，即模式名称（Pattern name）、问题（Problem）、解决方案（Solution）和效果（Consequence），全面地描述了在特定的上下文，相关的类和对象如何协作来解决这些常见的问题。而本书并不会从这些要素入手去重复 GoF 的工作，而是通过一些场景，讨论我们如何使用这些模式，并尽力纵横扩展我们的视野，希望以此启发读

者，为 OOP 开发设计打下坚实的基础。读者在阅读本书期间，可以参阅此书以作全面了解。

第2章　第一个模式

模板方法（Template Method）模式

2.1　从回家过年说起

春节是中国传统节日里最热闹的，对在外漂泊多时的游子而言，最幸福的事莫过于是回家过年。为了回家庆祝团圆，我们首先需要购买火车票，然后乘坐火车，最后才能和家人团聚。

我们这里编写一个简单的程序来模拟这个过程，最初的设计非常简单，我们编写了一个 HappyPeople 类，它有一个 celebrateSpringFestival()的方法，我们把买票，回家和家里庆祝的逻辑代码都写在这个方法里，代码大致如下所示：

```
public class HappyPeople {
    public void celebrateSpringFestival() {
        //Buying ticket…
        System.out.println("Buying ticket...");

        //Travelling by train…
        System.out.println("Travelling by train…");

        //Celebrating Chinese New Year…
        System.out.println("Happy Chinese New Year!");
    }
}
```

后来，我们逐渐发现，有人需要坐火车回家，有人需要坐飞机回家，而有人坐大巴回家就可以了。但是不管你乘坐哪种交通工具回家，都得先买票，然后才能和家人团聚。于是我们又创建了一个新类 PassengerByCoach 来实现坐汽车回家过年的逻辑，由于买票和在家庆祝的代码一样，我们把类 HappyPeople 的代码快速复制了一份，把乘坐交通工具的那部分代码替换成了坐大巴的逻辑，结果如下：

```
public class PassengerByCoach {
    public void celebrateSpringFestival() {
```

```
            //Buying a ticket…
            System.out.println("Buying ticket…");

            //Travelling by train…
            System.out.println("Travelling by train…");

            //Celebrating Chinese New Year…
            System.out.println("Happy Chinese New Year!");
        }
}
```

斜体加粗的部分便是我们替换的那部分代码，接着，我们同样使用**复制&粘贴（Copy&Paste）**方式编写了类 PassengerByAir 来实现表示坐飞机回家的那类人的需求。

复制&粘贴看起来非常实用，但是没过几周，我们慢慢发现情况不妙，这几个类的代码开始变得难以维护：如果买票逻辑有所改变，我们需要分别修改这三个类，但有的时候，马虎的工程师不会在所有类上做相应的修改；而且，由于这些类的功能发生了变化，相应类的测试代码也要做改变，这样修改这些类的测试代码和修改这些类一样，出现了相同的重复修改的问题；最后，我们开始变得越来越担心：因为随着交通工具的增多，势必我们需要开发更多类和测试类，这样维护就变得越来越麻烦。

最终我们停下来开始思考：**复制&粘贴**真的很好用吗？该不该编写重复的代码呢？让我们首先从 **DRY（Don't Repeat Yourself）原则**谈起吧。

2.2 DRY（Don't Repeat Yourself）

2.2.1 写在DRY之前

2.2.1.1 变化

在这个纷繁的世界上，你能否找到一样东西，它会永久不变？

谚语：

> **The Only Thing In The World That Doesn't Change Is Change Itself.**
>
> **世界上唯独不变的是变化本身。**

不管是日月星辰，还是软件开发，变化是永恒的。在软件开发过程中，一切都在变化：

- 需求变化：这也是软件开发中最主要的变化，我们经常听到的各种各样的抱怨，例如，之前拟定的需求突然又变化了；客户之前并不知道他们想要什么，现在还不能确定这些就是他们想要的；以前的需求根本是错误的；之前的需求并没有列出所有情形……，这都直接影响着软件开发设计。
- 技术变化：新技术不停地涌现，旧的技术被逐渐淘汰。
- 团队结构的变化：团队成员并不稳定，有来的，也有去的；此外团队组织结构的也在发生变化，主要体现在管理者的变化。
- 政治因素的变化：公司处于各种利益的考量，采取的一些行为干涉软件开发，这些行为往往是不可预料和没有回旋余地的。虽然这些行径经常被开发人员嗤之以鼻，但是这些变化真真实实地影响了软件的开发。
- 其他因素变化：自然灾害，航空灾难等等都可能影响软件开发。

总之，软件开发的最大的特征就是变化，变化总是在发生的，我们不能因为害怕变化而逃避它，否定它。谁能从容应对软件开发中的变化，谁就能最大程度上降低软件开发中的风险，成为这一方面的佼佼者。

2.2.1.2 开发还是维护

Andy Hunt和Dave Thomas 在《The Pragmatic Programmer》一书中认为，软件开发人员始终处于软件维护过程中，我个人非常认同这个观点，引用此书中的原话为[3]：

[3] 参见《The Pragmatic Programmer》一书，第二章：A Pragmatic Approach。

> **Programmers are constantly in maintenance mode. Our understanding changes day by day. New requirements arrive as we're designing or coding. Perhaps the environment changes. Whatever the reason, maintenance is not a discrete activity, but a routine part of the entire development process.**
>
> **程序员始一直处于维护状态，我们的理解每天都在发生变化，当我们设计和编码时，新的需求总是接踵而至，或许是由于环境的原因吧。不管是什么原因，维护不是一个离散的行为，而是整个日常软件开发的一部分。**

没错，不管是在应用发布之前还是之后，我们要么在修改程序错误（我们称为 Bug），要么为增强软件功能（我们称为 Enhancement）而修改代码，这些都属于软件维护的范畴。总而言之，软件维护始终贯穿于软件开发的始末。

2.2.2 DRY（Don't Repeat Yourself）

- **DRY（Don’t Repeat Yourself，不要复制自己）：**也叫DIE（Duplication Is Evil，即复制是魔鬼），这个原则在Andy Hunt和Dave Thomas所著的《The Pragmatic Programmer》一书中阐述为[4]：

  > **Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.**
  >
  > **每份知识在一个系统中必须存在唯一的，明确的，权威的表述。**

- **OAOO（Once and Only Once，仅此一次）：**OAOO 指的是要避免的代码重复，代码应该简洁，如果你发现代码“臭味”时，重复的代码是其中最严重的。

从以上定义我们不难发现，DRY 原则所涉及的范围其实要比 OAOO 宽泛的多，DRY 涉及的范围不仅包括代码，任何知识都算，例如逻辑，常量，标准，功能，服务等等，而 OAOO 指的是不能编写重复的代码，我们在本书主要将着力讨论如何使用设计模式避免代码的重复。

2.2.3 变化+重复，如何维护

我们已经知道，在软件开发过程中，变化时刻发生着，特别是需求变化，如果像前

[4] 参见《The Pragmatic Programmer》一书，第二章：A Pragmatic Approach。

述我们编写的重复的代码一样，维护这些重复的代码会给我们带来噩梦：

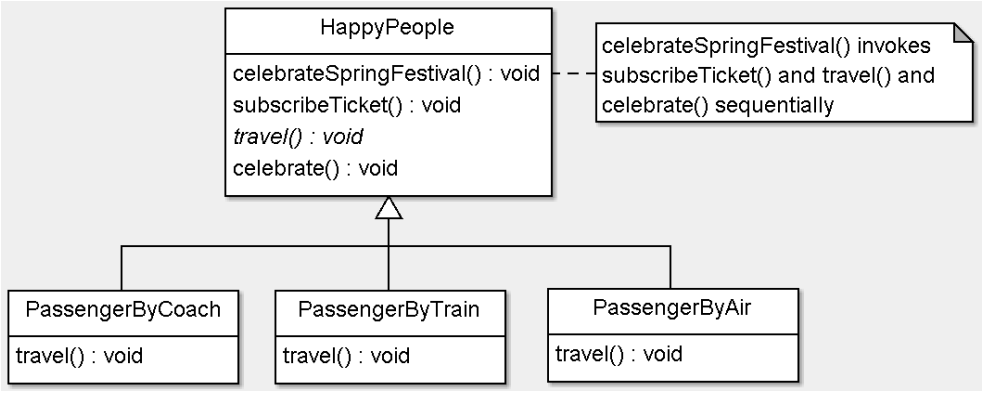
- 在重复代码中应用相同修改：为了增强功能或者修改错误（Bug），对于某一处代码的进行改动，在其他重复地方也可能需要修改。
- 开发成本增加：新的问题可能和旧问题非常相近，但是由于一些小细节的不同，导致了开发了不同的不能重用的代码。
- 不利于测试：如果相同的代码写在不同类里，导致测试代码的重复开发。
- 不利于阅读和维护：随着不断的开发维护，相同功能的实现在不同地方会变得不尽相同，有时甚至出现相同的需求实现逻辑却是不相同的，由于实现的不同，导致这些代码的健壮性也差异很大，很难维护这些代码。
- 代码重复还会引起性能等诸多问题，比如产生了很多重复的对象。

DRY 原则是我们软件开发里极其重要的一条原则，为了使我们的软件更加健壮，易于阅读和维护，我们应极力避免代码重复的“臭味”。下面让我们将讲述如何使用模板方法（Template Method）模式，避免上述丑陋的重复代码。

2.3 模板方法（Template Method）模式

2.3.1 使用继承

我们知道，为了重复使用代码，我们可以使用 OOP 一大特征——继承。既然 PassengerByCoach 类和 HappyPeople 类订票和庆祝团圆的逻辑是相同的，我们何不此抽象出一个父类，把这些相同的逻辑写在父类里？这样，我们为父类定义了 subscribeTicket()方法和 celebrate()方法，这些方法实现了不变的部分，订票和庆祝团圆。子类根据需要在 travel()方法里实现各自的回家方式。并且我们在父类里定义了 celebrateSpringFestival()方法供客户对象调用。我们先画出 UML 静态类图，如下所示：



我们重构了 HappyPeople 类为抽象父类，它包含一个抽象方法 ***travel()***供子类实现自定义的回家方式，celebrateSpringFestival()方法保证了 subscribeTicket()方法，travel()方法和 celebrate()方法按照顺序执行。

2.3.2 代码实现

我们首先实现父类 HappyPeople，代码大致如下所示：

```
public abstract class HappyPeople {
    public void celebrateSpringFestival() {
        subscribeTicket();
        travel();
        celebrate();
    }
    protected final void subscribeTicket() {
        //…
    }

    protected abstract void travel();

    protected final void celebrate() {
        //…
    }
}
```

现在，我们编写子类 PassengerByAir，它继承于 HappyPeople 类，travel()方法实现乘坐飞机回家的逻辑即可，代码如下所示：

```
public class PassengerByAir extends HappyPeople {
    @Override
    protected void travel() {
        //Traveling by Air…
        System.out.println("Travelling by Air...");
    }
```

```
}
```

这样客户对象使用PassengerByAir对象的celebrateSpringFestival()方法完成坐飞机回家过年的过程了。同样，PassengerByCoach子类和PassengerByTrain子类分别实现坐汽车回家和做火车回家的逻辑，代码在这里就不再赘述，详细请参见 示例代码。

我们编写如下代码以进行测试：

```
HappyPeople passengerByAir = new PassengerByAir();

HappyPeople passengerByCoach = new PassengerByCoach();

HappyPeople passengerByTrain = new PassengerByTrain();


System.out.println("Let's Go Home For A Grand Family Reunion...\n");


System.out.println("Tom is going home:");

passengerByAir.celebrateSpringFestival();


System.out.println("\nRoss is going home:");

passengerByCoach.celebrateSpringFestival();


System.out.println("\nCatherine is going home:");

passengerByTrain.celebrateSpringFestival();
```

执行结果如下所示：

```
Let's Go Home For A Grand Family Reunion...


Tom is going home:

Buying ticket...

Travelling by Air...

Happy Chinese New Year!


Ross is going home:
```

```
Buying ticket...
Travelling by Coach...
Happy Chinese New Year!


Catherine is going home:
Buying ticket...
Travelling by Train...
Happy Chinese New Year!
```

使用继承，子类中不需要实现那些重复的订票和庆祝团圆的代码了，避免了代码的重复；子类实现了不同回家的方法，把它栓入（hook）到父类中去，实现了完整的回家过年的逻辑。

2.3.3 模板方法模式

回顾上述代码，我们到底做了什么？父类中的方法 celebrateSpringFestival()是我们的一个模板方法（也叫框架方法），它把回家过年分为三步，其中方法 travel()是抽象部分，用于子类实现不同客户化逻辑，我们所使用的正是模板方法模式。GoF 给出的模板方法模式的定义是：

> **Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.**
>
> **定义了在一个操作中的一个算法框架，把一些步骤推迟到子类去实现。模板方法模式让子类不需要改变算法结构而重新定义特定的算法步骤。**

也就是说模板方法定义了一系列算法步骤，子类可以去实现/覆盖其中某些步骤，但不能改变这些步骤的执行顺序，模板方法有如下功能：

- 能够解决代码冗余问题。在此例中，PassengerByAir，PassengerByTrain 和 PassengerByCoach 等类没有必要去实现 subscribeTicket()和 celebrate()方法了。
- 把某些算法步骤延迟到子类，子类可以根据不同情况改变/实现这些方法，而子类的新方法不会引起既有父类的功能变化，例如上例中，我们加入 PassengerByAir 类，它没有影响 HappyPeople 类。

- 易于扩展。我们通过创建了新类，实现可定制化的方法就可以扩展功能。此例中，如果有人坐船回家，加入 PassengerByBoat 类实现父类的 travel()抽象方法即可。

- 父类提供了算法的框架，控制方法执行流程，而子类不能改变算法流程，子类方法的调用由父类模板方法决定。执行步骤的顺序有时候非常重要，我们在容器加载和初始化资源时，为避免子类执行错误的顺序，经常使用该模式限定子类方法的调用次序。此例中，你不能先回家再买票，也不能先庆祝再回家。

- 父类可以把那些重要的不允许改变的方法屏蔽掉，不让子类去覆写（Override/Overwrite）它们，比如在 Java 语言中，我们声明这些方法为 **final** 或者 **private** 即可。此例中，我们使用 **protected final** 关键字修饰 celebrate()和 subscribeTicket()方法，这样，子类不能覆写（Overwrite）它们；如果为了防止子类完全覆写 celebrateSpringFestival()方法，也可以修饰此方法为 final 的。

2.4 引入回调（Callback）

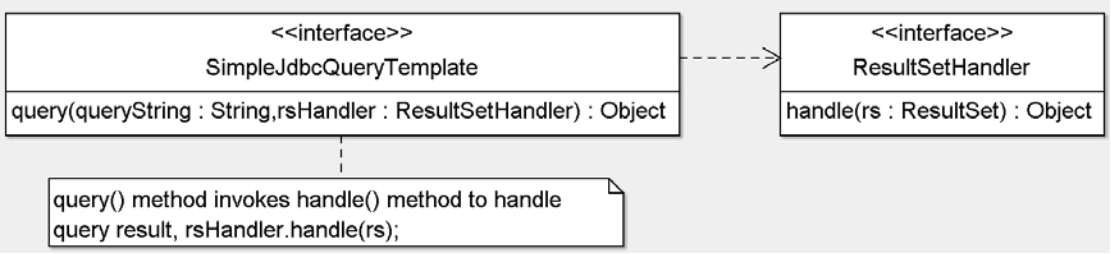
模板方法模式的应用很广泛，但过分地使用模板方法（Template Method）模式，往往会引起子类的泛滥。比如，我们有如下需求：查询数据库里的记录。首先我们需要得到数据库连接 Connection 对象，然后创建 Statement 实例并执行相关的查询语句，最后处理查询出来的结果并在整个执行过程中处理异常。研究这些步骤，不难发现，整个过程中第一步，第二步以及异常处理的逻辑对于每次查询来说，都是相同的，发生变化的部分主要是在对查询结果的处理上。据上分析，模板方法模式很适合处理这个问题，我们只需要抽象出这个处理查询结果的方法供不同的子类去延迟实现即可。

如果你正是这样做的，你就会慢慢发现，由于各种各样的查询太多，导致我们需要创建很多的子类来处理这些查询结果，引起了子类的泛滥。为了解决此问题，通常我们结合使用**回调**来处理这种问题。

回调表示一段可执行逻辑的引用（或者指针），我们把该引用（或者指针）传递到另外一段逻辑（或者方法）里供这段逻辑适时调用。回调在不同语言有不同的实现，例如，在 C 语言里我们经常使用函数指针实现回调，在 C#语言里我们使用代理（delegate）实现，而在 Java 语言里我们使用内部匿名类实现回调。这里我们还是以 Java 语言为例进行说明。

2.4.1 类图

为了实现上述数据库的查询，我们设计了 SimpleJdbcQueryTemplate 为模板方法类，ResultSetHandler 为回调接口，如下 UML 静态类图所示：



客户对象使用 SimpleJdbcQueryTemplate 类的 query(String queryString, ResultSetHandler rsHandler)方法时，需要把回调作为第二个参数传递给这个方法，query(…)方法会在做完查询后执行该回调的 handle(ResultSet rs)方法处理查询，返回最后的处理结果。

2.4.2 代码实现

定义ResultSetHandler接口，注意这里我们使用了Java新语法——泛型（Generics[5]），如下所示：

```
import java.sql.ResultSet;


public interface ResultSetHandler<T> {
    public T handle(ResultSet rs);
}
```

于是，SimpleJdbcQueryTemplate 类的代码片段如下所示：

```
import java.sql.*;


public class SimpleJdbcQueryTemplate {
    public <T> T query(String queryString, ResultSetHandler<T> rsHandler) {
        //define variables…
        try {
```

[5] 泛型是指能够在运行时动态得到对象类型的一种能力，这样，我们就没有必要每次都写强制类型转换语句了。其实 Java 是在编译时为生成的二进制代码加入强制类型转换的语句，并非真正在运行时得到对象的类型。

```
                connection = getCollection();//get a db connection.
                stmt = connection.prepareStatement(queryString);
                ResultSet rs = stmt.executeQuery();
                return rsHandler.handle(rs);
            } catch (SQLException ex) {
                //handle exceptions…
            } finally {
                //close the statement & connection…
            }
        }

        //other methods…
}
```

SimpleJdbcQueryTemplate 的 query(...)方法首先会获得一个数据库的连接，即这句，***connection = getConnection()***；接着创建了一个 PreparedStatement 实例，即 ***stmt = connection.prepareStatement(queryString)***准备查询；这句 ***ResultSet rs = stmt.executeQuery(queryString)***表示 ***stmt*** 对象去数据库执行该查询并返回结果；最后调用回调 rsHandler 的 handle(ResultSet rs)方法来处理处理查询结果并返回，即，***return rsHandler.handle(rs)***。

为了演示测试代码的执行，我们使用了EasyMock框架，EasyMock是一款非常流行的运行时生成Mock对象的软件，在单元测试中使用非常广泛。我们用它生成java.sql.Connection的Mock实例和java.sql.PreparedStatement实例去模拟真实的查询，这里仅给出测试的那部分代码片段，此例的详细代码请参见 示例代码，在这里就不再赘述，测试代码大致如下：

```
public void testTemplate() {
        boolean called = new SimpleJdbcQueryTemplate().
                query("select * from db",
                        new ResultSetHandler<Boolean>() {
                            public Boolean handle(ResultSet rs) {
                                //logical to resolve query result…
```

```
                    return Boolean.TRUE;
                }
            });

    //to verify result...
    assertTrue(called);
}
```

我们使用了 Java 的匿名类来会回调类处理查询结果，如上加粗斜体的部分所示。这样，即使有一千种不同的查询，我们也不需要增加一个专门的文件。

我们结合使用了模板方法模式和回调，避免了类的泛滥，此模式在 Spring框架里使用十分广泛，有兴趣的读者可以参看其关于ORM（Object-Relational Mapping）框架和Jdbc框架的源码以做进一步深入研究。

2.5 总结

这一章我们在介绍模板方法模式之前介绍了 DRY 原则，重复的代码会带来维护的噩梦，DRY 原则是一名优秀的软件开发人员必须恪守的原则之一。此原则看上去很简单，其实实现起来一点也不简单，在以下章节里，我们将继续碰到重复代码的“臭味”，我们会介绍更多的模式来防止代码重复。

模板方法模式非常简单，相信不少人在学习模式之前早就开始使用了。模板方法模式解决某些场景中的代码冗余问题，但也可能引入了类的泛滥问题，随后我们介绍了如何结合使用回调避免类的泛滥。使用回调可以避免类的泛滥，这并不是表示我们将使用带有回调的模板方法模式来替换所有的不带回调的模板方法模式，如果回调实现的接口较多，代码较为复杂时，把这些代码挤在一起会引起阅读问题。

在本章，为了解决回家过年的问题，我们使用了模板方法模式。在以下章节读者可以继续看到，如果需求变得更为复杂，我们就得需要更加灵活的设计，使用模板方法模式不能够成为新的复杂需求的解决方案，我们会在 策略模式等相关章节继续以该问题为例展开深入讨论。

第二篇 创建对象

创建一个对象并不难，但当我们不得不为每新添一种新的接口实现或者抽象类的具体子类而到处修改客户代码时，我们不得不思考直接使用 new 创建对象所带来的高耦合问题。

我们知道，如果类之间耦合度高，那么一个类里的一小处变化都可能导致其他类都发生未知的变化，这样的程序本身就充满了“腐臭”的气味，是不易重用的，脆弱的。OOP 带给我们的好处之一是封装，如果封装这些实例化的细节，即对客户对象隐藏实例化的过程，那么我们新添一个接口的实现或者抽象类的具体类，也不会影响客户代码了，降低了耦合度。

这篇我们将讲述如何创建对象而不会形成客户对象和服务对象之间高耦合的问题，这在提高代码的可移植性和重用性等方面都是非常有意义的：

- **第 3 章：单例（Singleton）模式**

  我们在一个系统里经常使用到单例对象，这章将讲述如何保证一个类在系统里只有一个对象被实例化。

- **第 4 章：工厂方法（Factory Method）模式**

  讲述如何实例化对象使得对象和使用者之间的耦合度降低。

- **第 5 章：原型（Prototype）模式**

  这章将讲述如何通过拷贝现有的对象快速地创建一个新的复杂对象。

- **第 6 章：控制反转（IoC）**

  我们介绍了时下最流行的两个概念，控制反转（IoC）和注入依赖（DI）。很多人对控制反转和注入依赖概念上混为一谈，这章将为读者讲述它们二者之间的关系，并介绍目前关于二者的一些常用技术和框架。

第3章　单例（Singleton）模式

3.1　概述

如果要保证系统里一个类最多只能存在一个实例时，我们就需要单例模式。这种情况在我们应用中经常碰到，例如缓存池，数据库连接池，线程池，一些应用服务实例等。在多线程环境中，为了保证实例的唯一性其实并不简单，这章将和读者一起探讨如何实现单例模式。

3.2　最简单的单例

为了限制该类的对象被随意地创建，我们保证该类构造方法是私有的，这样外部类就无法创建该类型的对象了；另外，为了给客户对象提供对此单例对象的使用，我们为它提供一个全局访问点，代码如下所示：

```
public class Singleton {
    private static Singleton instance = new Singleton();
    //other fields…

    private Singleton() {
    }

    public static Singleton getInstance() {
        return instance;
    }

    //other methods…
}
```

代码注解：

- Singleton 类的只有一个构造方法，它是被 ***private*** 修饰的，客户对象无法创建该类实例。
- 我们为此单例实现的全局访问点是 ***public static Singleton getInstance()***方法，注意，***instance*** 变量是私有的，外界无法访问的。

读者还可以定义 ***instance*** 变量是 public 的，这样把属性直接暴露给其他对象，就没必要实现 ***public static Singleton getInstance()***方法，但是可读性没有方法来的直接，而且把该实例变量的名字直接暴露给客户程序，增加了代码的耦合度，如果改变此变量名称，会引起客户类的改变。

还有一点，如果该实例需要比较复杂的初始化过程时，把这个过程应该写在static{…}代码块中。

- 此实现是线程安全的，当多个线程同时去访问该类的getInstance()方法时，不会初始化多个不同的对象，这是因为，JVM（Java Virtual Machine）在加载此类时，对于static属性的初始化只能由一个线程执行且仅一次[6]。

由于此单例提供了静态的公有方法，那么客户使用单例模式的代码也就非常简单了，如下所示：

```
Singleton singleton = Singleton.getInstance();
```

3.3 进阶

3.3.1 延迟创建

如果出于性能等的考虑，我们希望延迟实例化单例对象（Static 属性在加载类是就会被初始化），只有在第一次使用该类的实例时才去实例化，我们应该怎么办呢？

这个其实并不难做到，我们把单例的实例化过程移至 getInstance()方法，而不在加载类时预先创建。当访问此方法时，首先判断该实例是不是已经被实例化过了，如果已被初始化，则直接返回这个对象的引用；否则，创建这个实例并初始化，最后返回这个对象引用。代码片段如下所示：

```
public class UnThreadSafeSingelton {
    //variables and constructors…

    public static UnThreadSafeSingelton getInstance() {
        if(instatnce ==null){
```

[6] Static 属性和 Static 初始化块（Static Initializers）的初始化过程是串行的，这个由 JLS（Java Language Specification）保证，参见 James Gosling, Bill Joy, Guy Steele and Gilad Bracha 编写的《 The Java™ Language Specification Third Edition》一书的 12.4 一节。

```
            instatnce = new UnThreadSafeSingelton();
        }
        return instatnce;
    }
}
```

我们使用这句 ***if(instatnce ==null)*** 判断是否实例化完成了。此方法不是线程安全的，接下来我们将会讨论。

3.3.2 线程安全

上节我们创建了可延迟初始化的单例，然而不幸的是，在高并发的环境中，getInstance()方法返回了多个指向不同的该类实例，究竟是什么原因呢？我们针对此方法，给出两个线程并发访问 getInstance()方法时的一种情况，如下所示：

	Thread 1	**Thread 2**
1	if(instatnce ==null)	
2		if(instatnce ==null)
3	instatnce = new UnThreadSafeSingelton();	
4	return instatnce;	
5		instatnce = new UnThreadSafeSingelton();
6		return instatnce;

如果这两个线程按照上述步骤执行，不难发现，在时刻 **1** 和 **2**，由于还没有创建单例对象，Thread1 和 Thread2 都会进入创建单例实例的代码块分别创建实例。在时刻 3，Thread1 创建了一个实例对象，但是 Thread2 此时已无法知道，继续创建一个新的实例对象，于是这两个线程持有的实例并非为同一个。更为糟糕的是，在没有自动内存回收机制的语言平台上运行这样的单例模式，例如使用 C++编写此模式，因为我们认为创建了一个单例实例，忽略了其他线程所产生的对象，不会手动去回收它们，引起了内存泄露。

为了解决这个问题，我们给此方法添加 **synchronized** 关键字，代码如下：

```
public class ThreadSafeSingelton {
```

```
    //variables and constructors…


    public static synchronized ThreadSafeSingelton getInstance() {
        if(instatnce ==null){
            instatnce = new ThreadSafeSingelton();
        }
        return instatnce;
    }
}
```

这样，再多的线程访问都只会实例化一个单例对象。

3.3.3 Double-Check Locking

上述途径虽然实现了多线程的安全访问，但是在多线程高并发访问的情况下，给此方法加上 **synchronized** 关键字会使得性能大不如前。我们仔细分析一下不难发现，使用了 **synchronized** 关键字对整个 getInstance()方法进行同步是没有必要的：我们只要保证实例化这个对象的那段逻辑被一个线程执行就可以了，而返回引用的那段代码是没有必要同步的。按照这个想法，我们的代码片段大致如下所示：

```
public class DoubleCheckSingleton {
    private volatile static DoubleCheckSingleton instatnce = null;


    //constructors


    public static DoubleCheckSingleton getInstance() {
        if (instatnce == null) {      //check if it is created.
            synchronized (DoubleCheckSingleton.class) {    //synchronize creation block
                if (instatnce == null)      //double check if it is created
                    instatnce = new DoubleCheckSingleton();
            }
        }
        return instatnce;
```

```
    }
}
```

代码注解：

- 在 getInstance()方法里，我们首先判断此实例是否已经被创建了，如果还没有创建，首先使用 synchronized 同步实例化代码块。在同步代码块里，我们还需要再次检查是否已经创建了此类的实例，这是因为：如果没有第二次检查，这时有两个线程 Thread A 和 Thread B 同时进入该方法，它们都检测到 instatnce 为 null，不管哪一个线程先占据同步锁创建实例对象，都不会阻止另外一个线程继续进入实例化代码块重新创建实例对象，这样，同样会生成两个实例对象。所以，我们在同步的代码块里，进行第二次判断判断该对象是否已被创建。

  正是由于使用了两次的检查，我们称之为 double-checked locking 模式。

- 属性 instatnce 是被 volatile 修饰的，因为 volatile 具有 synchronized 的可见性特点，也就是说线程能够自动发现 volatile 变量的最新值。这样，如果 instatnce 实例化成功，其他线程便能立即发现。

注意：

此程序只有**在 JAVA 5 及以上版本才能正常运行**，在以前版本不能保证其正常运行。这是由于 Java 平台的内存模式容许 out-of-order writes 引起的，假定有两个线程，Thread 1 和 Thread 2，它们执行以下步骤：

1. Thread 1 发现 instatnce 没有被实例化，它获得锁并去实例化此对象，JVM 容许在没有完全实例化完成时，instance 变量就指向此实例，因为这些步骤可以是 out-of-order writes 的，此时 instance==null 为 false，之前的版本即使用 volatile 关键字修饰也无效。
2. 在初始化完成之前，Thread 2 进入此方法，发现 instance 已经不为 null 了，Thread 2 便认为该实例初始化完成了，使用这个未完全初始化的实例对象，则很可能引起系统的崩溃。

3.3.4 Initialization on demand holder

要使用线程安全的延迟的单例初始化，我们还有一种方法，称为 Initialization on demand holder 模式，代码如下所示：

```
public class LazyLoadedSingleton {
```

```
    private LazyLoadedSingleton() {
    }

    private static class LazyHolder {   //holds the singleton class
        private static final LazyLoadedSingleton singletonInstatnce = new
LazyLoadedSingleton();
    }

    public static LazyLoadedSingleton getInstance() {
        return LazyHolder.singletonInstatnce;
    }
}
```

当 JVM 加载 ***LazyLoadedSingleton*** 类时，由于该类没有 static 属性，所以加载完成后便即可返回。只有第一次调用 getInstance()方法时，JVM 才会加载 LazyHolder 类，由于它包含一个 static 属性 ***singletonInstatnce***，所以会首先初始化这个变量，根据前面的介绍，我们知道此过程并不会出现并发问题（JLS 保证），这样即实现了一个既线程安全又支持延迟加载的单例模式。

3.3.5 Singleton的序列化

如果单例类实现了 Serializable 接口，这时我们得特别注意，因为我们知道在默认情况下，每次反序列化（Desierialization）总会创建一个新的实例对象，这样一个系统会出现多个对象供使用。我们应该怎么办呢？

熟悉 Java 序列化的读者可能知道，我们需要在 ***readResolve()***方法里做文章，此方法在反序列化完成之前被执行，我们在此方法里替换掉反序列化出来的那个新的实例，让其指向内存中的那个单例对象即可，代码实现如下：

```
import java.io.Serializable;

public class SerialibleSingleton implements Serializable {
    private static final long serialVersionUID = -6099617126325157499L;
    static SerialibleSingleton singleton = new SerialibleSingleton();
```

```
    private SerialibleSingleton() {
    }

    // This method is called immediately after an object of this class is deserialized.
    // This method returns the singleton instance.
    private Object readResolve() {
        return singleton;
    }
}
```

方法 ***readResolve()***直接返回 singleton 单例，这样，我们在内存中始终保持了一个唯一的单例对象。

3.4 总结

通过这一章的学习，我相信大家对于基本的单例模式已经有了一个比较充分的认识。其实我们这章讨论的是在同一个 JVM 中，如何保证一个类只有一个单例，如果在分布式环境中，我们可能需要考虑如何保证在整个应用（可能分布在不同 JVM 上）只有一个实例，但这也超出本书范畴，在这里将不再做深入研究，有兴趣的读者可以查阅相关资料深入研究。

第4章　工厂方法（Factory Method）模式

4.1　概述

上一章我们讲述了如何创建单例对象，这章，我们将讲述如何使用工厂方法模式创建普通对象。工厂方法模式是我们常用的模式之一，我们经常在以下情景使用：

- 客户类不关心使用哪个具体类，只关心该接口所提供的功能。
- 创建过程比较复杂，例如需要初始化其他关联的资源类，读取配置文件等等。
- 接口有很多具体实现或者抽象类有很多具体子类时，你可能你需要为客户代码写一大串 if-else 逻辑来决定运行时使用哪个具体实现或者具体子类。
- 不希望给客户程序暴露过多此类的内部结构，隐藏这些细节可以降低耦合度。
- 优化性能，比如缓存大对象或者初始化比较耗时的对象。

4.2　工厂方法模式

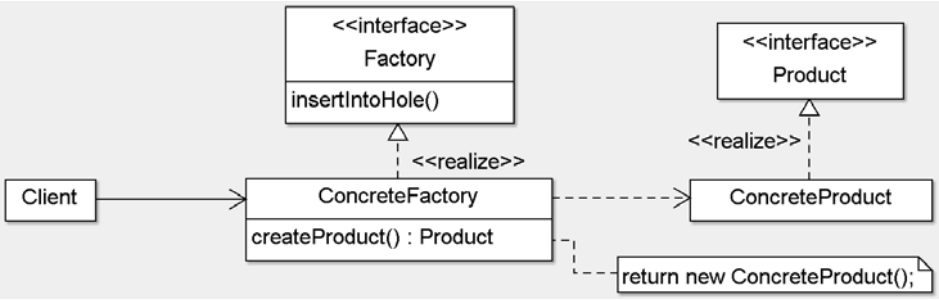
GoF 为工厂方法模式给出的定义如下：

> **Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.**
>
> **为创建对象定义一个接口，让子类决定实例化哪个类。工厂方法让一个类的实例化延迟至子类。**

工厂方法模式是对实例化过程进行封装而形成的，客户对象无需关心实例化这些类的细节，把它们交给工厂类，在 GoF 的定义中，为工厂类定义了一个工厂接口。该模式比较简单，我们从 UML 静态类图着手。

4.2.1　类图

我们有一个 Product 接口，其实现类为 ConcreteProduct，工厂负责创建 Product 对象，这样客户对象就不用关心使用哪个实现以及如何初始化该实例，以后 Product 实现类如果发生了改变，则不会引起客户代码的改动，只要维护相应的工厂类即可。此模型中，Factory 接口是工厂接口，其实现类是 ConcreteFactory，图示如下：



4.2.2 代码实现

根据上述的 UML 图示，不难写出实现代码，如下所示：

```
public interface Factory { //an interface mainly to create Product objects
    Product createProduct();
}

public class ConcreteFactory implements Factory {
    @Override
    public Product createProduct() {
        return new ConcreteProduct();
    }
}

public interface Product {
}

public class ConcreteProduct implements Product{
}
```

代码注解：

- Factory 接口定义了 createProduct()方法来返回 Product 类型的实例对象，ConcreteFactory 类实现了该方法，每次调用都会实例化一个新的 ConcreteProduct 对象返回。

工厂方法模式使用起来也非常简单，代码如下所示：

```
public class Client {
    private Factory factory;

    public Client(Factory factory) {
        this.factory = factory;
    }

    public void doSomething(){
        Product product= factory.createProduct();
        //to do something...
    }

    public static void main(String[] args) {
        Client client = new Client(new ConcreteFactory());
        client.doSomething();
    }
}
```

如果具体实现类较多，我们可以定义一个参数化的工厂方法，根据不同的输入返回不同的实现子类，代码片段如下：

```
public Product createProduct(String type){
    if("type1".equals (type)){
        return a type1 instance…
    }else "type2".equals (type){
        return a type2 instance…
    } else{
        ….
    }
}
```

客户对象可以根据需要传入不同的参数获得不同类型的对象。

4.2.3 创建数据库连接对象

有时候实例化对象可能会非常消耗资源，我们需要考虑缓存这些实例，于是我们工厂对象里创建一个缓存池缓存这些实例对象。在 J2EE Web 应用中，我们经常使用 ThreadLocal 缓存数据连接对象。可能有些读者还不熟悉 ThreadLocal 类，在实现之前，我们首先讲解一下这个类。

ThreadLocal 类可以保证在同一个线程里持有同一个对象的引用，即用当前的线程绑定一个实例。由于使用了弱引用（Weak Reference），在使用完后，JVM 会自动销毁这些绑定的对象。

也许有人使用 Java 很多年了，还不了解弱引用，我们就 JVM 支持的引用类型做个简单介绍。从 Java2 开始，引用被分为 4 个级别，我们一般使用到强引用，只要没有引用指向该对象时，这个对象便会被垃圾回收器回收。还有其他 3 种特别的引用，分别是软引用（Soft Reference），弱引用（Weak Reference）和幻影引用（Phantom Reference），它们的级别依次减弱，并都弱于强引用：

- 软引用：如果一个对象没有强引用只有软引用时，当 JVM 发现内存不够时，垃圾回收器便会回收这些对象。
- 弱引用：如果一个对象只具有弱引用，在每次回收垃圾时，该对象都会被回收。
- 幻影引用：如果一个对象仅持有幻影引用，那么它就和没有引用指向它一样，在任何时候该对象都可能被垃圾回收。

幻影引用和软引用与弱引用的区别在于：虚引用**必须**和引用队列（ReferenceQueue）一起使用。幻影引用可以用来跟踪对象被回收的活动，因为当垃圾回收器准备回收一个对象时，如果发现它还有幻影引用，就会在回收之前，把这个幻影引用加入到与之关联的引用队列（ReferenceQueue）中去。这样，程序可以通过判断引用队列中是否已经加入了幻影引用，来了解被引用的对象是否将要被垃圾回收，如果发现某个幻影引用已经被加入到引用队列，那么就可以在所引用的对象被回收之前采取必要的行动。

软引用和弱引用非常适合做缓存，关于它们更详细介绍，请参见 java.lang.ref 包。

ThreadLocal 类使用到弱引用把对象绑定到当前线程，为每一个线程提供一个对象的拷贝。如果没有强引用或者软引用指向该对象时，每次垃圾回收器启动时便会回收该对象。

在 J2EE Web 应用中，每接收到一个 HTTP 请求时，就会启用一个线程来处理这个请求，使用 ThreadLocal 类这样很容易实现在处理同一个请求的整个过程中，尽可能使用同一个数据库连接对象，使用完成后，JVM 总会自动清理该数据库连接对象。代码片段如下：

```
import java.sql.Connection;


public class ConnectionFactory {
    private final ThreadLocal<Connection> connections = new ThreadLocal<Connection>();
    //other variables and methods...


    public Connection currentConnection() {
        Connection conn = connections.get();
        if (conn == null) {//if no connection binds to this thread, create a new one


            //create a new connection.
            conn = createConnection();


            //bind this connction to current thread
            connections.set(conn);
        }
        return conn;
    }
}
```

代码注解：

- 定义了一个 ThreadLocal 对象来做数据连接的缓冲池，即这句，***private static final ThreadLocal<Connection> connections = new ThreadLocal<Connection>()***。

- 如果当前线程没有绑定的 Connection 对象（第一次获取数据连接或者已被回收），则 ***connections.get()***返回 null，这时我们就创建一个新的 java.sql.Connection 对象，并把它绑定到当前线程，即这句，***connection.set(conn)***。以后只要没有启动垃圾回收，当前请求总会得到刚创建的数据连接对象。

- 虽然我们这里为每个 HTTP 请求缓存 Connection 对象，但在实际应用中，我们并不是直接去创建一个数据连接对象，我们往往从另外的数据连接池获得缓存的 Connection 对象，试想如果在一个小型应用中，对 10000 次 HTTP 请求创建 10000 个数据连接，往往会引起不必要的数据库瓶颈问题。数据连接池提供多个线程可以重用的数据库连接对象，它们不会随着线程消亡而被关闭或回收，数据库连接池并非本书重点，在这里就不再深入探讨。

4.3 静态工厂方法

4.3.1 概述

工厂模式非常实用，但是为每一个类创建一个工厂方法方法类会引起工厂类的泛滥，此时，我们可以使用静态工厂方法可以来避免，我们在每个类里实现一个静态的工厂方法，就不需要额外的工厂类了。静态工厂方法在《Effective Java》[7]一书中有详细的介绍，我们也经常使用它们。例如，在Java1.5 版本里，为创建基本类型Integer，Long，Boolean对象都提供了静态工厂方法，以Integer类为例，它的静态工厂方法如下所示：

```
public static Integer valueOf(int i) {
    if(i >= -128 && i <= IntegerCache.high)
        return IntegerCache.cache[i + 128];
    else
        return new Integer(i);
}
```

代码注解：

- java.lang.Integer.IntegerCache.high 默认值是 127，可以通过设置 VM 的启动参数的值来设定（大于 127 才有意义）。如果在[-128, IntegerCache.high]之间，则返回 IntegerCache 类缓存的 Integer 对象，否则创建一个新的 Integer 对象。

- 这里需要注意的是，由于 Integer 是不可变（immutable）对象，所以这些缓存对

[7] 参见该书的 Item 1: Consider providing static factory methods instead of constructors。

象不会引起计算上的错误。

4.3.2 静态工厂模式的优缺点

- **优点：**

1. 可以为静态工厂选择合适的命名，提高程序的可读性。一段优秀的代码具有可读性，并不一定需要冗长的注释，有时候根据类名，方法名，变量名的良好命名更容易使读者读懂程序。
2. 静态工厂和工厂模式一样，可以封装复杂的初始化过程，实现实例的缓存。
3. 还可以根据不同的输入返回不同实现类/具体类对象。

- **缺点：**

1. 一般为了强迫使用工厂方法，不直接使用构造方法来构造实例，我们会强迫类只含有私有构造方法，这样，会导致此类不能被子类化。
2. 如果添加了一个新的该类的子类（该类有非私有的构造方法），此静态工厂方法可能需要重写，以加入该子类的实例化过程，扩展性不强。
3. 静态方法没有面向对象的特征，比如继承，动态多态等，不可被覆写（Overwritten）。

第5章　原型（Prototype）模式

5.1　概述

谈到原型模式，学过 Java 的人可能会想到 java.lang.Cloneable 这个接口，以为 Java 的原型模式描述的就是 java.lang.Cloneable 接口的使用，这就大错特错了。其实，原型模式在我们日常生活中经常可以看到，比如你刚给你的客厅做了装修，你朋友正好也希望给他的客厅做装修，那么，他可能会把你家的装修方案拿过来改改就成，你的装修方案就是原型。

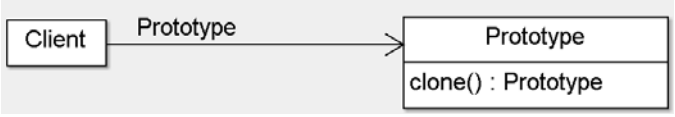
由于很多 OOP 语言都支持对象的克隆（拷贝）以方便复制对象，但这些方式并不那么完美，后述我们将会讨论。

5.2　原型模式

当创建这些对象（一般情况是一些大对象）非常耗时，或者创建过程非常复杂时，非常有用，GoF 给出的原型模式定义如下：

> **Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.**
>
> **使用原型实例指定将要创建的对象类型，通过拷贝这个实例创建新的对象。**

原型模式的静态类图非常简单，如下所示：



Client 使用 Prototype 的 clone()方法得到这个对象的拷贝，其实拷贝原型对象不一定是指从内存中进行拷贝，我们的原型数据可能保存在数据库里。

一般情况下，OOP 语言都提供了内存中对象的复制，Java 语言提供了对象的浅拷贝（Shallow copy），也就是说复制一个对象时，如果它的一个属性是引用，则复制这个引用，使之指向内存中同一个对象；但如果为此属性创建了一个新对象，让其引用指向它，即是深拷贝（Deep copy）。

5.3 寄个快递

下面是一个邮寄快递的场景：

“给我寄个快递。”顾客说。

“寄往什么地方？寄给……？”你问。

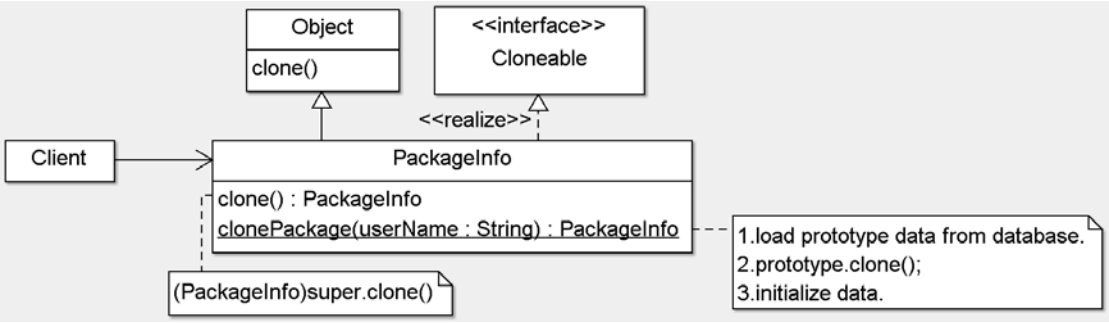
“和上次差不多一样，只是邮寄给另外一个地址，这里是邮寄地址……”顾客一边说一边把写有邮寄地址的纸条给你。

“好！”你愉快地答应，因为你保存了用户的以前邮寄信息，只要复制这些数据，然后通过简单的修改就可以快速地创建新的快递数据了。

5.4 实现

我们在复制新的数据时，需要特别注意的是，我们不能把所有数据都复制过来，例如，当对象包含主键时，不能使用原型数据的主键，必须创建一个新的主键。我们这里提供一个静态工厂方法，来获得原型数据，然后拷贝这些数据，最后做相应的初始化。为了操作安全起见，我们不直接使用原型数据，而是使用 clone()方法从内存克隆这条原型数据做后续操作。我们使用 Java 提供 java.lang.Cloneable 接口克隆数据，它实现对象的浅拷贝，关于 java.lang.Cloneable 接口的使用请看后续介绍。

5.4.1 UML静态类图



Client 使用 PackageInfo 提供的静态工厂方法 clonePackage(String userName)创建一个新对象：首先根据 userName 加载一条用户以前的数据作为原型数据（可以是数据库，可以是其他任何你保存数据的地方），然后在内存中克隆这条数据，最后初始化该数据并返回。

5.4.2 代码实现

```
public class PackageInfo implements Cloneable {
    //getters, setters and other methods...

    public PackageInfo clone() {
        try {
            return (PackageInfo)super.clone();
        } catch (CloneNotSupportedException e) {
            System.out.println("Cloning not allowed.");
            return null;
        }
    }

    public static PackageInfo clonePackage(String userName) {
        //load package as prototype data from db...
        PackageInfo prototype = loadPackageInfo(userName);

        //clone information ...
        prototype = prototype.clone();

        //initialize copied data ...
        prototype.setId(null);

        return prototype;
    }
}
```

代码注解：

- Java 的 java.lang.Object 方法里就提供了克隆方法 clone()，原则上似乎所有类都拥有此功能，但其实不然，关于它的使用有如下限制：

- 要实现克隆，必须实现 ***java.lang.Cloneable*** 接口，否则在运行时调用 clone()方法，会抛 CloneNotSupportedException 异常。
- 返回的是 Object 类型的对象，所以使用时可能需要强制类型转换。
- 该方法是 protected 的，如果想让外部对象使用它，必须在子类重写该方法，设定其访问范围是 public 的，参见 PackageInfo 的 ***clone()***方法。
- Object 的 clone()方法的复制是采用逐字节的方式从复制内存数据，复制了属性的引用，而属性所指向的对象本身没有被复制，因此所复制的引用指向了相同的对象。由此可见，这种方式拷贝对象是浅拷贝，不是深拷贝。

- 静态工厂方法 ***public static PackageInfo clonePackage(String userName)***方法根据原型创建一份拷贝：首先拿出用户以前的一条数据，即这句 ***PackageInfo prototype = loadPackageInfo(userName)***，然后调用方法它的 clone()方法完成内存拷贝，即 ***prototype.clone()***，最后我们初始化这条新数据，比如使 id 为空等。

现在来看看我们的测试代码，如下所示：

```
public class PackageInfoTestDrive {
    public static void main(String[] args) {
        PackageInfo currentInfo = PackageInfo.clonePackage("John");
        System.out.println("Original package information:");
        display(currentInfo);

        currentInfo.setId(10000l);
        currentInfo.setReceiverName("Ryan");
        currentInfo.setReceiverAddress("People Square, Shanghai");

        System.out.println("\nNew package information:");
        display(currentInfo);
    }
    //other methods…
}
```

我们通过这句，***PackageInfo currentInfo = PackageInfo.clonePackage("John")***，拷贝了一份快递信息出来，通过设置 currentInfo.setReceiverName("Ryan")和

currentInfo.setReceiverAddress("People Square, Shanghai")，便完成了第二个包裹的信息录入，测试结果如下：

```
Original package information:
Package id: null
Receiver name: John
Receiver address: People Square,Shanghai
Sender name: William
Sender Phone No.: 12345678901

New package information:
Package id: 10000
Receiver name: Ryan
Receiver address: People Square, Shanghai
Sender name: William
Sender Phone No.: 12345678901
```

在实际的应用中，使用原型模式创建对象图[8]（Object Graph）非常便捷。

5.5 深拷贝（Deep Copy）

通过上述学习，我们知道 Java 提供了浅拷贝的方法，那么，如何实现一个深拷贝呢？一般情况下，我们有两种方式来实现：

1. 拷贝对象时，递归地调用属性对象的克隆方法完成。读者可以根据具体的类，撰写出实现特定类型的深拷贝方法。

   一般地，我们很难实现一个一般性的方法来完成任何类型对象的深拷贝。有人根据反射得到属性的类型，然后依照它的类型构造对象，但前提是，这些属性的类型必须含有一个公有的默认构造方法，否则作为一个一般性的方法，很难确定传递给非默认构造方法的参数值；此外，如果属性类型是接口或者抽象类型，必须提供查找到相关的具体类方法，作为一个一般性的方法，这个也很难办到。

[8] 对象图不是一个单个对象，而是一组聚合的对象，该组对象有一个根对象。

2. 如果类实现了 java.io.Serializable 接口，把原型对象序列化，然后反序列化后得到的对象，其实就是一个新的深拷贝对象。

我们整理给出第二种方法的实现，代码片段大致如下所示：

```
import java.io.Serializable;
//other imports…

public class DeepCopyBean implements Serializable {
    private String objectField;
    private int primitiveField;

    //getters and setters …

    public DeepCopyBean deepCopy() {
        try {
            ByteArrayOutputStream buf = new ByteArrayOutputStream();
            ObjectOutputStream o = new ObjectOutputStream(buf);
            o.writeObject(this);

            ObjectInputStream in = new ObjectInputStream(new
ByteArrayInputStream(buf.toByteArray()));
            return (DeepCopyBean) in.readObject();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

代码注解：

- DeepCopyBean 实现了 ***java.io.Serializable*** 接口，它含有一个原始类型（Primitive Type）的属性 primitiveField 和对象属性 objectField。
- 此类的 ***deepCopy()***方法首先序列化自己到流中，然后从流中反序列化，得到的对象便是一个新的深拷贝。

为了验证是不是实现了深拷贝，我们编写了如下测试代码：

```
DeepCopyBean originalBean = new DeepCopyBean();


//create a String object in jvm heap not jvm string pool
originalBean.setObjectField(new String("123456"));
originalBean.setPrimitiveField(2);


//clone this bean
DeepCopyBean newBean = originalBean.deepCopy();


System.out.println("Primitive ==? " + (newBean.getPrimitiveField() ==
originalBean.getPrimitiveField()));
System.out.println("Object ==? " + (newBean.getObjectField() ==
originalBean.getObjectField());
System.out.println("Object equal? " +
(newBean.getObjectField().equals(originalBean.getObjectField()));
```

注意：

这句，originalBean.setObjectField(***new String("123456")***)和 originalBean.setObjectField(***"123456"***)是不一样的，前者创建了两个 String 对象，其中一个是在 JVM 的字符串池（String pool）里，另外一个在堆中，并且属性引用指向的对象在堆里；后者属性引用指向了 JVM 字符串池中的***"123456"***对象。

如果是浅拷贝，即引用指向同一内存地址，则 ***newBean.getObjectField() == originalBean.getObjectField()***为 ***true***，如果是深拷贝，则创建了不同对象，引用指向的地址肯定不一样，即此值应为 ***false***。但是这两种方式，使用这句，***newBean.getObjectField().equals(originalBean.getObjectField())***，进行比较，其结果必须为 ***true***，测试结果如下所示：

```
Primitive ==? true
Object ==? false
Object equal? true
```

和我们预想的结果一样，原始类型的使用==进行比较，结果相等，而引用类型使用==比较，结果显示未指向相同的地址，但是使用 equals()方法比较的结果为 ***true***，即证明我们实现了深拷贝。

使用这种方式进行深拷贝，一方面，它只能拷贝实现 Serializable 接口类型的对象，其属性也是可序列化的；另一方面，序列化和反序列化比较耗时。选用此方式实现深拷贝时需要做这两方面的权衡。

5.6 总结

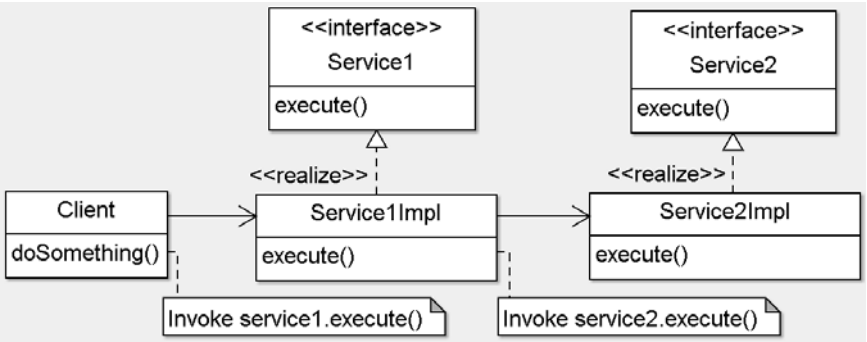
我们以前使用 ***java.lang.Cloneable*** 的一个很大原因是使用 new 创建对象的速度相对来说比较慢，如今，随着 JVM 性能的提升，new 的速度已经很接近 Object 的 clone()方法的速度了，然而这并没有使原型模式使用失去多少光泽，使用原型模式有以下优点：

- 创建大的聚合对象图时，没必要为每个层次的子对象创建相应层次的工厂类。
- 方便实例化，只要复制对象，然后初始化对象，就可以得到你想要的对象，并不不需要过多的编程。

第6章　控制反转（IoC）

6.1　从创建对象谈起

我们前述讲解了使用 工厂方法模式创建对象，比如现在有这样的场景：Client对象需要使用Service1 的execute()方法完成特定功能，而Service1 的实现Service1Impl类又依赖于Service2 的实现类Service2Impl，UML静态类图如下：



为了减少依赖，我们为 Service1 和 Service2 对象的实例化分别提供工厂方法类的实现，代码大致如下：

```
public interface Service1 {
    void execute();
}
public class Service1Impl implements Service1 {
    private Service2 service2;

    public Service1Impl(Service2 service2) {
        this.service2 = service2;
    }

    public void execute() {
        System.out.println("Service1 is doing something.");
        service2.execute();
    }
}
```

```
public interface Service2 {
    void execute();
}
public class Service2Impl implements Service2 {
    public void execute() {
        System.out.println("Service2 is doing something.");
    }
}
```

于是，接口 Service1 和 Service2 工厂类的实现代码如下所示：

```
public class Service2Factory {
    public static Service2 getService2Instance() {
        return new Service2Impl();
    }
}

public class Service1Factory {
    public static Service1 getService1Instance() {
        Service2 service2 = Service2Factory.getService2Instance();
        return new Service1Impl(service2);
    }
}
```

这样，Client 类的实现如下：

```
public class Client {
    public void doSomething() {
        Service1 service = Service1Factory.getService1Instance();
        service.execute();
    }
}
```

我们的测试代码非常简单，如下所示：

```
new Client().doSomething();
```

执行结果如下：

```
Service1 is doing something.
Service2 is doing something.
```

6.2 使用工厂方法模式的问题

- 工厂类泛滥：在实际应用中，经常出现几百个像这样的服务类，如果使用工厂方法模式的话，就可能出现几百个这样的工厂类。我们不能使用静态工厂模式的理由也很简单，因为你不会知道将来会产生多少子类扩展系统。
- 依赖关系复杂：往往这几百个 Service 对象之间存在复杂的依赖关系，工厂类的装配逻辑也随之变得十分复杂。我们为装配这些服务对象花费了很多精力，而不能很好专注于业务功能的开发。
- 不易单元测试：我们需要创建一些 Mock 对象来进行单元测试，以本例说明，我们要测试 Client 的 doSomething()方法，需要使用一个 Service1 的 Mock 对象，一般情况下，我们在测试时修改 getService1Instance()方法，测试完成后，再改回来，这样做非常麻烦，极易出错。

总之，大规模地使用工厂方法模式，会引起诸多问题，为了解决这些问题，我们先从控制反转（IoC，Inversion of Control）说起。

6.3 Inversion of Control（控制反转，IoC）

大家可能已经知道**好莱坞（Hollywood）原则**：

> **Don’t call us, we’ll call you.**
>
> **不要找我们，我们会找你。**

好莱坞原则在软件开发领域中极受追捧：我们经常把控制逻辑写在其他地方（例如 Framework）而非客户化的代码里，我们就更专注于客户化的逻辑，也就是说，外部逻辑负责调用我们客户化的逻辑。在软件开发领域，我们又给它取了一个名字叫**控**

制反转（IoC）[9]。

控制反转概念的涉及面十分广泛，有人认为它是一个模式，但是我更倾向于认为它是一个原则（Principle）。很多模式都实现了控制反转（例如模板方法模式），例如，我们 第 2 章讲解的 模板方法模式就是控制反转的一个很好应用，父类的模板方法控制子类的方法调用；还有，使用 回调的方法都是控制反转的很好应用。

再如，在 Java 标准库里，我们常用到查找和排序的这两个方法，binarySearch(...)和 sort(...)方法，它们在执行过程中会调用对象的 compareTo()方法（如果这些对象实现了 java.lang.Comparable 接口的话），或者调用我们所传递的回调接口 java.util.Comparator 对象的 compare()方法来比较大小，最终完成查找/排序操作，这些都是控制反转的例子。

此外，我们经常提到的框架（Framework），它最典型的特点其实就是控制反转：框架抽象了通用流程，我们可以通过框架提供的规范（比如子类化抽象类或者实现相关的接口，实现相关的交互协议和接口等等）就可以把客户化的代码植入流程中，完成各种定制的需求。框架和工具库（Library）的区别是：如果框架没有实现控制反转，那么框架就会退化为工具库。也就是说，使用工具库，你必须撰写调用它们的逻辑，每次调用它们都会完成相应的工作，并把控制权返回给调用者；而框架不需要客户程序撰写控制调用的逻辑，由框架专门负责。

以我们时下非常流行的 MVC 框架 Struts 为例，我们只需要实现相关的 Action，Form 类以及相关 View，并配置好 Action-Form-View 的映射关系，这样每次客户端提交请求，该框架都会选择相应的 Action 去处理它，并根据返回的执行结果选择相应的视图。这些控制逻辑由 Struts 框架为我们完成了，不需要我们实现如何调用相应 Action 执行的逻辑，也不需要实现如何选择 View 显示的逻辑。

6.3.1 IoC和DI（Dependency Injection，依赖注入）

后来，随着轻量级容器/框架不停地涌现，控制反转的概念越来越被开发人员提及，很多轻量级容器/框架标榜使用到的控制反转，其实通过上述介绍我们知道，任何容器/框架都实现了控制反转。它们所说的控制反转指的是对服务/对象/组件的实例化

[9] 这个短语首先在 Johnson and Foote 撰写的论文 Designing Reusable Classes 使用到。

和查找实现了控制反转，这只是控制反转的一种而已。关于这方面的控制反转主要有两种形式：Service Locator（服务定位器）和 Dependency Injection（依赖注入，简写为 DI），下面我们就这两种形式展开讨论。

6.3.2 Service Locator（服务定位器）

服务定位器封装了服务/对象/组件查找，为它们提供一个全局的入口。例如在 EJB 中，为了查找所需要的组件/服务，我们使用 javax.naming.InitialContext 对象去查找：

```
javax.naming.Context ctx = new javax.naming.InitialContext();
YourService yourService = (YourService) ctx.lookup(Source Name);
```

为了直观说明，我们这里为上述例子实现了一个非常简单的服务定位器。首先我们构建一个 ServiceLocator 类，代码如下所示：

```
import java.util.HashMap;
import java.util.Map;
public class ServiceLocator {
    private static ServiceLocator locator;
    private Map<String,Object> services = new HashMap<String,Object>();

    public static void configure() {
        load(new ServiceLocator());

        Service2 service2 = new Service2Impl();
        locator.services.put("service2", service2);
        Service1 service1 = new Service1Impl(service2);
        locator.services.put("service1", service1);
    }

    private static void load(ServiceLocator serviceLocator) {
        locator = serviceLocator;
    }
```

```
    public static Object lookup(String serviceName) {
        return locator.services.get(serviceName);
    }

    public static void registerService(String name,Object service){
        locator.services.put(name, service);
    }
}
```

从上述实现可以看到，我们创建了一个 **HashMap<String,Object>**对象来持有这些服务对象，Service1 和 Service2 的对象其实是按照单例创建的。这样我们的 Client 类的实现如下所示：

```
public class Client {
    public void doSomething() {
        Service1 service1 = (Service1) ServiceLocator.lookup("service1");
        service1.execute();
    }
}
```

那么，使用之前，我们需要使用 ***ServiceLocator.configure()***初始化，以下是我们的测试代码：

```
public static void main(String[] args) {
    //Initialize the services
    ServiceLocator.configure();

    new Client().doSomething();
}
```

服务定位器封装了查找逻辑，隐藏了组件/服务/对象之间的依赖关系，客户对象只要依赖于它就能获取你想要的组件/服务/对象。工厂方法模式和服务定位器的区别是：服务定位器为整个应用组件/服务/对象的获取提供了单一的入口，而一个工厂只提供特定类型的实例化，如果一个工厂能提供所有组件/服务/对象的装配和实例化，那它就被进化为服务定位器。

使用服务定位器时，容器/框架侵入了你的代码，降低了代码移植性，单元测试也相对比较麻烦。以上述场景为例，为了测试 Client 的方法，我们需要替换掉 Service1 服务对象，代码如下所示：

```
public void testDoSomething() {
    //setup env to test this method
    ServiceLocator.configure();
    Service1 service1 = (Service1) ServiceLocator.lookup("service1");
    MockService1 mockService1 = new MockService1();
    ServiceLocator.registerService("service1",mockService1);

    try {
        new Client().doSomething();
        assertEquals(mockService1.isExecuted(),true,"Test failed");
    } finally {
        ServiceLocator.registerService("service1",service1);
    }
}
```

我们在正式测试之前，用 ***MockService1*** 服务对象替换掉原来的 Service1 服务对象，即 ***ServiceLocator.registerService("service1",mockService1)***；在测试结束之后，我们要把原来的服务对象注册回去，否则很可能影响其他测试，于是，在最后，为了避免测试过程中发生异常中断，我们把重新注册回原来 service1 服务对象的逻辑放在 finally 代码块里。由于每个测试都要做类似的清理操作，给单元测试带来极大的不便。

6.3.3 Dependency Injection

外部程序把服务对象通过某种方式注入到客户对象供其使用的方法称之为**注入依赖**。

根据注入方式的不同，我在这里把依赖注入分为 6 类：Setter注入，Constructor注入，Annotation注入，Interface注入，Parameter注入和 其他形式的注入。目前实现依赖注入的框架有很多，下面我们就结合现在非常流行的一些轻量级容器/框架来作介绍。

6.3.3.1 Setter注入

Spring框架是时下被广泛使用的J2EE开源框架，它不只是一款轻量级的依赖注射容器，还包括持久化框架，事务管理框架，Web应用框架（MVC框架），AOP等框架。它实现了Setter注入等多种注入方式，这里以其为例来介绍Setter方式的注入。Setter注入是指外部程序通过调用setting方法为客户对象注入所依赖的对象。

为了给 Client 注入所依赖的 Service1 服务对象，我们为该属性提供了一个 setting 方法，代码如下：

```
public class Client {
    private Service1 service1;

    public void setService1(Service1 service1) {
        this.service1 = service1;
    }

    //other methods…
}
```

以同样方式，我们为 Service1Impl 类的依赖对象 service2 提供一个 setting 方法，由于非常简单，代码就不再赘述。

接下来我们需要在 Spring 配置文件里配置它们之间的依赖关系，如下所示：

```
<bean id="client" class="pattern.part2.chapter6.setter.Client">
    <property name="service1" ref="service1"/>
</bean>

<bean id="service1" class="pattern.part2.chapter6.setter.Service1Impl">
    <property name="service2" ref="service2"/>
</bean>
```

```
<bean id="service2" class="pattern.part2.chapter6.setter.Service2Impl"/>
```

配置文件里，节点***<bean id="client" class="pattern.part2.chapter6.setter.Client ">…</bean>***定义了一个名字为***"client"***的单例服务对象，其实现类是***pattern.part2.chapter6.setter.Client***，子节点***<property name="service1" ref="service1"/>***表示对象 **client** 的属性 **service1** 含有一个 setting 方法，并且依赖指向了名字为***"service1"***的对象。同样在***"service1"***服务对象的配置里，属性 **service2** 的依赖指向***"service2"***对象。

这样，我们就完成了它们之间的依赖关系，为了测试注入效果，我们编写的代码大致如下：

```
public static void main(String[] args) {
    ApplicationContext ctx = new
ClassPathXmlApplicationContext("pattern/part2/chapter6/setter/spring-beans.xml");
    Client client = (Client) ctx.getBean("client");

    client.doSomething();
}
```

ApplicationContext ctx = new ClassPathXmlApplicationContext(…)加载配置文件，并根据 XML 配置文件定义好的依赖关系初始化所有实例对象，这样只要我们从 Spring 容器得到的对象就是通过 setting 方法装配好依赖的对象，为了演示 ***client*** 对象的***doSomething()***方法调用，我们首先得到***"client"***对象，即 ***Client client = (Client) ctx.getBean("client")***。看看执行效果如何，结果如下所示：

```
Service1 is doing something.
Service2 is doing something.
```

由于使用了 setting 方法注入依赖，在单元测试时，我们也非常方便的使用其注入 Mock 对象，于是 Client 单元测试代码如下：

```
public void testDoSomething() {
    Client client = new Client();
    MockService1 mockService1 = new MockService1();
    client.setService1(mockService1);
```

```
    client.doSomething();
    assertEquals(mockService1.isExecuted(), true, "Test failed");
}
```

在单元测试时，我们手动调用setting方法注入依赖对象，Spring框架是使用反射机制调用setting方法注入依赖的对象，它们之间没有什么本质区别。

因为我们的 Mock 对象并未替换容器里的对象，在单元测试结束之后不必要撰写清理逻辑。

6.3.3.2 Constructor注入

Constructor注入，顾名思义，就是通过带参数的构造方法注入依赖对象，Pico Container也是一款非常流行的轻量级DI容器，我们以其为例进行说明。为了给Client对象注入Service1 的对象，我们为其创建一个带有参数的构造方法，代码如下所示：

```
public class Client {
    private Service1 service1;

    public Client(Service1 service1) {
        this.service1 = service1;
    }
    //other methods…
}
```

同样我们也为 Service1Impl 类实现一个带参数的构造函数，如下所示：

```
public class Service1Impl implements Service1 {
    private Service2 service2;

    public Service1Impl(Service2 service2) {
        this.service2 = service2;
    }
    //other methods…
}
```

Pico Container可以根据构造函数参数的类型，在注册的服务类里查找相应的实现类，这样我们就可以省去一些简单的依赖配置了。我这里写了一个配置方法来注册服务类，如下所示：

```
private static MutablePicoContainer configure() {
    MutablePicoContainer pico = new DefaultPicoContainer();
    pico.addComponent(Service1Impl.class);//register an implementation of Service1
    pico.addComponent(Service2Impl.class); //register an implementation of Service2
    pico.addComponent(Client.class);
    return pico;
}
```

由于 Pico Container没有提供像 Spring框架通过配置文件定义这些组件类和依赖关系的功能，需要我们手工实现，在实际使用中，大家可以根据各自喜好，可以选择编程配置，也可以写个程序从配置文件读取这些定义。

我们在使用这个容器之前，调用这个配置方法，测试代码如下：

```
public static void main(String[] args) {
    MutablePicoContainer pico = configure();
    Client client = pico.getComponent(Client.class);
    client.doSomething();
}
```

由于 Pico Container支持泛型，这里也不需要做强制类型转换就能得到Client对象，得到Client对象的这句代码是***Client client = pico.getComponent(Client.class)***。测试运行结果如下：

```
Service1 is doing something.
Service2 is doing something.
```

同样，在单元测试时，我们根据带参数的构造函数，手工注入 Service1 的 Mock 对象即可，在测试完成之后，我们也不需要做额外的清理工作，代码如下所示：

```
public void testDoSomething() {
    MockService1 mockService1 = new MockService1();
    Client client = new Client(mockService1);
```

```
        client.doSomething();
        assertEquals(mockService1.isExecuted(), true, "Test failed");
}
```

6.3.3.3 Annotation注入

在Java版本 5 即以上做过开发的人都应该了解**注解（Annotation）**[10]。我们知道，**Annotation**可以注解各种类型，例如属性，类型，方法，本地变量，Annotation，构造方法等等。只要这些Annotation是被***@Retention(RUNTIME)***注解的，那么，此类型的Annotation注解信息就可以在运行时通过反射读取。自从Java引入Annotation，Java的世界就变得更加热闹了。

如果我们把实例化信息和对象之间的依赖关系信息使用Annotation注解，那么，只要在运行时得到它们，就知道如何初始化服务对象了。Guice是Google旗下的一款非常出色的轻量级DI容器，由于使用了Annotation实现注入依赖，加之其优良的性能，得到广大编程爱好者的一致好评，这里我们就以其为例来作说明。

Guice使用***@ com.google.inject.ImplementedBy***指定此类型的具体实现，例如接口Service1 实现类是Service1Impl，那么注解的代码就如下所示：

```
import com.google.inject.ImplementedBy;

@ImplementedBy(Service1Impl.class)
public interface Service1 {
        void execute();
}
```

同样，我们可以注解接口 Service2，这里不再赘述。

Guice提供Annotation***@com.google.inject.Inject***表示在实例化该类型的对象时，需要

[10] Java 5 及以上支持的特性，Annotation 能够修饰方法，类，参数，变量，构造方法，包和类型（类，接口，枚举等）的声明。可以在编译，加载，运行过程中读取这些有用信息，根据这些信息执行相应的操作。具体使用请参见附录 A 推荐的 Java 相关书籍。

为被注解目标注入依赖，此Annotation可以注解方法，构造函数和属性 3 种类型。

我们使用它分别注解 Service1Impl 类的 setting 方法和 Client 类的构造函数，代码如下所示：

```
import com.google.inject.Inject;

public class Service1Impl implements Service1 {
    private Service2 service2;

    @Inject
    public void setService2(Service2 service2) {
        this.service2 = service2;
    }

    //other methods…
}
```

```
import com.google.inject.Inject;

public class Client {
    private Service1 service1;

    @Inject
    public Client(Service1 service1) {
        this.service1 = service1;
    }

    //other methods…
}
```

由于依赖关系比较简单，我们不需要额外的配置。现在只要从 ***Injector*** 查找我们需要的对象，它就会根据此类型的定义读取注解，实例化我们想要的对象。测试使用 Annotation 注入的代码如下：

```
public static void main(String[] args) {
    Injector injector = Guice.createInjector();
    Client client = injector.getInstance(Client.class);

    client.doSomething();
}
```

注意，同样由于 Guice 支持泛型（Generics），我们在使用 Injector 的 getInstance(…) 方法时不需要进行强制类型转化。

由于使用***@com.google.inject.Inject*** 注解了 Client 构造函数，我们照样可以使用该构造函数注入我们的 Mock 对象进行单元测试，代码如下所示：

```
public void testDoSomething() {
    MockService1 mockService1 = new MockService1();
    Client client = new Client(mockService1);

    client.doSomething();
    assertEquals(mockService1.isExecuted(), true, "Test failed");
}
```

需要注意的是，如果对私有属性进行注解，虽然实现了更加精准的注入，但是给单元测试造成了一定的麻烦，因为私有属性在其他类里是不可见的，所以不能通过直接赋值的方式注入 Mock 对象（框架使用反射机制，在给私有属性赋值之前，调用 field.setAccessible(true)，除非你也是用反射初始化你的类），建议大家尽量避免使用。

6.3.3.4 Interface注入

客户程序通过实现容器/框架所规范的某些特殊接口，在为客户对象返回这些依赖的对象之前，容器回调这些接口的方法，注入所依赖的服务对象。例如很久以前Apache的 Avalon 框架就是使用这种方式注入。现在，Struts2 也使用了这种方式注入一些依赖的对象，比如，如果你的Action实现了接口ServletRequestAware，SessionAware等，

只要Action类实现这些接口，Struts框架会回调这些接口的相应方法，注入HttpServletRequest，HttpSession对象。我这里给出一个简单的示例，演示如何实现接口注入。

首先我们定义一个 ServiceAware 接口，如下所示：

```
public interface ServiceAware {
    void injectService(Service service);
}
```

Service 接口及其实现类的代码如下：

```
public interface Service {
    void execute();
}

public class ServiceImpl implements Service {
    public void execute() {
        System.out.println("Service is doing something...");
    }
}
```

只要服务类实现这个接口，容器就会注入 Service 对象，我们定义一个 Client 类实现该接口，代码如下：

```
public class Client implements ServiceAware {
    private Service service;

    @Override
    public void injectService(Service service) {
        this.service = service;
    }

    public void doSomething() {
```

```
            service.execute();
        }

        //other methods…
}
```

我们下一步来创建这个简单的接口注入容器，代码如下所示：

```
import java.util.HashMap;
import java.util.Map;

public class InterfaceInjector {
        private static InterfaceInjector injector;
        private Map<Class, Object> services = new HashMap<Class, Object>();

        public static void configure() {
                load(new InterfaceInjector());
        }

        public static <T> T getInstance(Class<T> clazz) {
                return injector.loadService(clazz);
        }

        private static void load(InterfaceInjector container) {
                InterfaceInjector.injector = container;
        }

        @SuppressWarnings("unchecked")
        private <T> T loadService(Class<T> clazz) {
                Object service = injector.services.get(clazz);
                if (service != null) {
                        return (T) service;
```

```
            }


            try {
                service = clazz.newInstance();
                if (service instanceof ServiceAware) {
                    ((ServiceAware) service).injectService(new ServiceImpl());
                }
                injector.services.put(clazz, service);
            } catch (InstantiationException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
            return (T) service;
        }
}
```

我们使用根据 Class 类型查找服务对象，第一次查找时采用反射机制使用默认的构造函数生成对象。代码的核心部分是这句，***if (service instanceof ServiceAware){...}***，通过接口装配依赖的对象：如果对象是 ***ServiceAware*** 的实例，则调用此接口的方法注入 ServiceAware 对象，即这句***((ServiceAware) service).injectService(new ServiceImpl())***，我们为每个 ***ServiceAware*** 实例注入 ***ServiceImpl*** 对象。

现在，我们看看如何使用，代码如下：

```
public static void main(String[] args) {
    InterfaceInjector.configure();
    Client client = InterfaceInjector.getInstance(Client.class);


    client.doSomething();
}
```

执行效果如下：

```
Service is doing something...
```

我们根据接口方法，也可以轻松地手工注入 Service 的 Mock 对象完成对 Client 类的单元测试，同样未影响容器里的对象，在测试结束时也不必要做额外的清理操作，代码如下所示：

```
public void testDoSomething() {
    MockService mockService = new MockService();
    Client client = new Client();
    client.injectService(mockService);

    client.doSomething();
    assertEquals(mockService.isExecuted(), true, "Test failed");
}
```

这种方式不够灵活，容器必须预先定义一些接口实现注入，适合实现少数特定类型的对象注入。

6.3.3.5 Parameter注入

外部程序可以通过函数参数，给客户程序注入所依赖的服务对象，非常简单。Client 类的代码如下所示：

```
public class Client {
    public void doSomething(Service service) {
        service.execute();
    }
}
```

方法 doSomething(Service service)使用外部传递 Service 对象，至于 Service 如何是实例化的，它一无所知。

如何使用呢？其实非常简单，代码如下：

```
public static void main(String[] args) {
    //create a service
    Service service = new ServiceImpl();
```

```
        //put new-created service instance by input parameter
        new Client().doSomething(service);
}
```

外部程序实例化一个 Service 实例，传递给 Client 的 doSomething(Service service)方法使用。

单元测试代码非常简单，和使用一样，在测试调用方法时传入的是 Mock 对象，测试结束时当然也不需要做额外的清理操作，代码如下所示：

```
public void testDoSomething() {
        MockService mockService = new MockService();
        Client client = new Client();

        client.doSomething(mockService);
        assertEquals(mockService.isExecuted(), true, "Test failed");
}
```

注意，这种形式的注入比较特殊，Client类的doSomething(Service service)方法不光完成了依赖的装配，而且执行了Service回调的方法execute()，完成了其他逻辑。如果是一个提供Parameter注入的框架程序，在没有特殊需求的情况下，在实例化的过程中，应该使用Parameter注入完成服务对象的装配逻辑，这里如此举例的目的只是为了简要说明这种方式。其实，可以看出，Setter注入是一种特殊的参数注入，它规定只能使用setting方法注入依赖，同样，Constructor注入也是一种特殊的参数注入，只能对构造函数实现参数注入。

一些流行DI框架在使用参数注入实例化对象时，往往结合 Annotation注入，Interface注入等方式一起使用，但是明显的一条，这些实现方法不应包含除依赖装配之外的其他逻辑。

6.3.3.6 其他形式的注入

很多框架还有其他特点的注入，这些注入形式往往和框架有关。例如，Pico Container 和 Guice的使用Providers的注入方式，Spring框架的lookup方法注入方式，这些都是

这些容器/框架所提供的一些高级用法，主要用来解决一些特殊问题，读者以后有机会可以慢慢实践，这里不再赘述。

6.4 总结

IoC 使得客户化专注于客户化的逻辑设计，把程序流程的控制交给外部代码，实现高内聚低耦合目标。

同样，我们把实例化的过程交给框架/容器来处理，这样我们就更专注于业务逻辑的开发，这也是轻量级容器迅速发展的一个重要原因。Ioc实现实例化的方式主要有两种：一种是使用服务定位器，另外一种是使用依赖注入。依赖注入的 Setter注入和 Constructor注入是最常见的注入方式，代码往往不会受到容器/框架的入侵，可以在多种轻量级容器上移植，而其他的方式或多或少都受到容器/框架代码的入侵。

关于 IoC/DI这方面流行的轻量级框架有 Spring框架，Guice，Picocontainer等，前面也使用它们编写了一些简单的实例代码进行说明，感兴趣的读者可以上相关网站作更深入研究。

Martin Fowler有一篇精彩的文章讲述 IoC和 DI的关系，链接地址为 http://martinfowler.com/articles/injection.html。

第三篇 构建复杂结构

我们知道，一个类是不能完成一个系统的全部功能，为了完成某些复杂的功能，我们往往需要多个对象组成更大更复杂的结构协作。使用面向对象来合成更大单元的方式，主要有合成，继承，还有像 Ruby语言的Mixin[11]等，我们这里主要讨论合成和继承两种。

通常情况下，为了使用现有类的功能，避免代码重复，我们可以继承该类，从而具有父类的性质；或者我们使用合成，执行它的功能。到底那种方式更好呢？或者是否需要结合二者呢？这章会给您提供建议。

[11] Mixin 是一种复用代码的方式，可以把不同模块的代码通过 include 关键字引入。

第7章　装饰器（Decorator）模式

7.1　概述

提到装饰器模式，使用过 Java IO 编程的人应该非常熟悉，Java IO 包就是使用装饰器模式设计的。譬如，为了快速地从 InputStream 流读取数据，我们使用 BufferedInputStream 装饰该流，被它装饰过的流便增加了缓冲数据的功能。装饰器模式可以让我们在运行时动态地增强对象的功能，而不影响客户程序的使用。

7.2　记录历史修改

几年前，我曾接到过一个任务，需要开发一个数据访问层（Data Access Layer），提供对持久化数据的访问。我们当时使用的是关系数据库（Relational Database），所以选用了 Hibernate这款非常著名的ORM[12]（Object-Relational Mapping）工具来实现持久化。

刚开始，我编写了一个非常简单的接口GenericRepository用于完成CRUD操作 [13]，代码如下所示：

```
public interface GenericRepository<T> {
    void save(T o);

    void update(T o);

    //other methods...
}
```

幸亏有 Hibernate工具，我们的实现变得异常简单了：使用Hibernate Session对象的save(T o)方法可以根据ORM映射关系把对象持久化到数据库，它的update(T o)方法可以更新持久化的数据，其他方法就不再一一叙述，Hibernate官方文档非常全面，感兴趣读者可以登录它的网站下载相关文档进行学习。

[12] 对象关系映射（Object-Relational Mapping），由于关系数据库和面向对象的数据之间存在不匹配现象，为了像使用面向对象数据库那样使用关系数据库的数据，我们使用这方面技术为面向对象编程虚拟出对象数据库以供数据访问。

[13] 是指四个最基础的持久化操作：Create，Read，Update 和 Delete 操作，缩写为 CRUD。

我们的代码大致如下所示：

```
import org.hibernate.Session;
//other imports…

public class GenericRepositoryHibernate<T> implements GenericRepository<T> {
    @Override
    public void save(T o) {
        if (o == null) {
            return;
        }
        Session session = getCurrentSession();//get hibernate session
        session.save(o);
    }

    @Override
    public void update(T o) {
        if (o == null) {
            return;
        }

        Session session = getCurrentSession();//get hibernate session
        session.update(o);
    }

    //other fields and methods...
}
```

我们的getCurrentSession()方法用于得到Hibernate的Session对象，然后调用Session对象的相应方法就可以创建和更新数据了。在 4.2.3 节，我们讲述了如何在Web应用使用ThreadLocal类给当前线程绑定一个Connection对象，我们这里的getCurrentSession()方法就是使用ThreadLocal给当前线程绑定一个Session对象，关于此类的使用，请参

见 ThreadLocal类的介绍。

随着开发的深入，几个星期之后，用户希望对某些业务操作进行回滚，为了实现回滚，我们要对修改前的数据记录日志，这样才能完成回滚。比方说，他们的这些操作会影响用户账户状态，我们就为用户账户信息记录日志，我们把数据存储在 user 这张表中，把日志记录在另外一张日志表 user_log 中。

在刚开始的讨论中，很多人想到了直接修改 GenericRepositoryHibernate 这个类，这样，update(T o)方法的代码便会被修改为如下所示：

```
public void update(T o) {
    Session session = getCurrentSession();//get hibernate session

    session.update(o);
    if(isLog(o)){
        saveLog(o);
    }
}
```

这种方式对现有类进行了修改，但对之前健壮的代码进行修改，很可能导致这些代码不再健壮。为了保证以前的代码能够正常执行，我们必须对该类重新做单元测试，更麻烦的是，由于此方法非常基础，为了保证不影响到其他功能，我们还要为所有受影响的其他类和方法以及它们的单元测试代码进行修改，这一工作量相当大。能否不修改这些健壮的代码又能扩展它的功能呢？我们首先看看一条设计原则——**OCP 原则（Open-Close Principle）**。

7.3 Open-Closed Principle（开放——封闭原则，OCP）

Software Entities (Classes, Modules, Functions, etc.) Should Be Open For Extension, But Closed For Modification.

软件实体（类，模块，功能等等）应当对扩展开放，但是对修改关闭。

即就是说，使用 OOP 进行开发，我们只能修改已有的类的错误（Bugs），若是扩展它们的功能，我们不能修改它们，而是要通过添加新类（实体）来解决，这即**开放——封闭原则**。为了不破坏 OCP 原则，我们下面尝试了两种方案。

方案一：

对于上述问题，为了避免增加修改已有 GenericRepositoryHibernate 类的代码，有人提议创建一个新类实现 GenericRepository 接口，代码片段大致如下：

```
public class LogRepositoryHibernate<T> implements GenericRepository<T> {
    public void update(T o) {
        Session session = getCurrentSession();//get hibernate session


        session.update(o);
        saveLog(o);
    }
    //other fields and methods…
}
```

问题：代码重复

- 不难看出，上述代码斜体加粗的部分与 GenericRepositoryHibernate 类的 update(Object o)方法的代码重复，虽然未违反 OCP 原则，却违反了 DRY 原则。

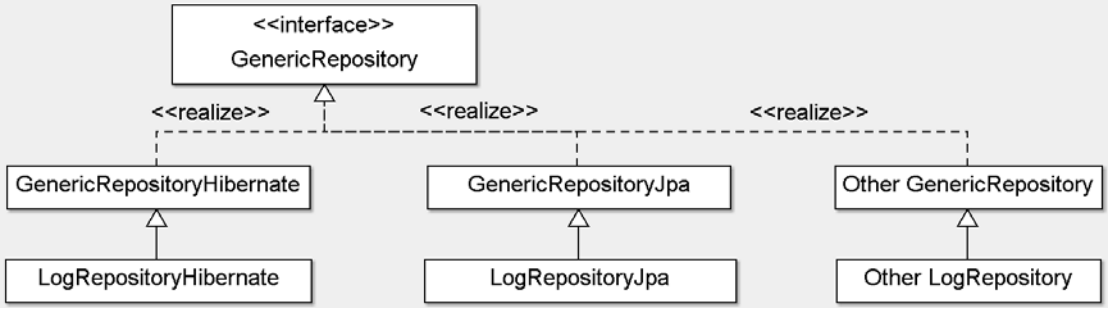
方案二：

我们想到，为了重用代码，可以使用继承，我们可以让 LogRepositoryHibernate 继承于 GenericRepositoryHibernate 类，不错，我们看看如何实现，代码大致如下所示：

```
public class LogRepositoryHibernate<T> extends GenericRepositoryHibernate<T> {
    public void update(T o) {
        super.update(o);
        saveLog(o);
    }
    //other fields and methods…
}
```

问题：扩展性不高，代码重复仍然未能避免

- 上述解决方法一定程度避免了代码重复，如果我们使用 JPA 来实现持久化，该怎么办呢？我们就必须得增加新持久化实现，设计可能如下 UML 图示：

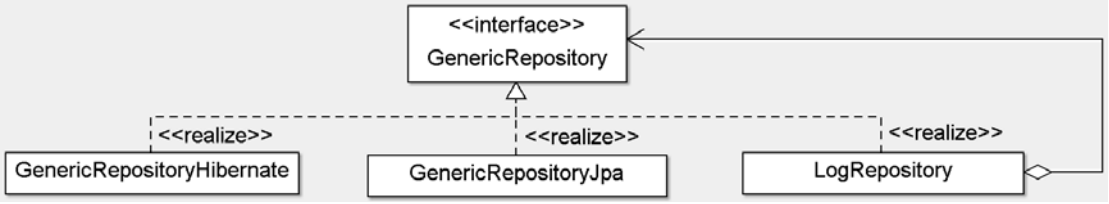


这样明显会造成类的膨胀，我们每使用一种新持久化方法，就需要一个新的 LogRepository 实现类，有没有更好的方式呢？答案是有的，请看下节。

7.4 装饰器（Decorator）模式

我们知道，要解决代码重复问题，我们还可以使用合成；另外，为了保证改变对客户对象透明，我们新添的类型应该保持同样接口。如果按照这个想法来设计，这会是一种什么样的结构呢？让我们先画出 UML 静态类图。

7.4.1 类图



LogRepository 对象聚合了一个 GenericRepository 的对象，并且 LogRepository 实现了 GenericRepository 接口，它备份完之前的数据状态，然后把请求转发给 GenericRepository 的对象去做更新操作。

7.4.2 实现

为了记录日志，我这里给出一个简单实现：在更新之前把数据的全部信息备份在另外一张日志表 user_log 中，我们那些没有修改的信息，我们也保存在日志表中。

我们给出创建 user 表和 user_log 表结构的 DDL 语句如下所示：

```
create table user (id number(20), name varchar(250), primary key (id))
```

```
create table user_log (log_id number(20), id number(20), log_time timestamp, name varchar(250), primary key (log_id))
```

这样，我们在更新操作之前在 user 表里查找到这条记录，然后把这条旧记录备份到日志表 user_log，最后把新记录持久化到日志表即可。

我们的 User 类的代码大致如下所示：

```
//imports...

@Entity
@Table(name = "user")
public class User {
    private Long id;
    private String name;
    //other fields...

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Column(name = "name", columnDefinition = "varchar(250)")
    public String getName() {
        return name;
```

```
    }

    public void setName(String name) {
        this.name = name;
    }

    //other getters, setters and other methods...
}
```

为了记录日志，我们为日志类增加了一个接口 Loggable，如下：

```
public interface Loggable {
    Date getLogTime();

    void setLogTime(Date logTime);
}
```

那么，UserLog 类如下所示：

```
//imports...

@Entity
@Table(name = "user_log")
public class UserLog implements Loggable {
    private Long logId;
    private Long id;
    private String name;
    private Date logTime;
    //other fields...

    @Id
    @Column(name = "log_id")
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
public Long getLogId() {
        return logId;
}

public void setLogId(Long logId) {
        this.logId = logId;
}

@Column(name = "id")
public Long getId() {
        return id;
}

public void setId(Long id) {
        this.id = id;
}

@Column(name = "name", columnDefinition = "varchar(250)")
public String getName() {
        return name;
}

public void setName(String name) {
        this.name = name;
}

@Column(name = "log_time")
public Date getLogTime() {
        return logTime;
}
```

```
    public void setLogTime(Date logTime) {
        this.logTime = logTime;
    }

    //other getters, setters and other methods...
}
```

我们上述使用Hibernate提供的Annotation配置ORM关系，由于这不是本章重点，大家只要知道有了上述Annotation注解，就可以把User对象及其日志对象持久化到数据库即可，感兴趣的读者可以登录 Hibernate官方网站作更深一层次的了解。

现在，我们编写 LogRepository 类代码，在 update(Object o)方法中，我们首先执行 saveLog(Object o)方法把这条记录前一个状态持久化到日志表中，之后我们调用被修饰对象 ***genericRepository*** 的 update(Object o)方法更新改变，代码大致如下所示：

```
public class LogRepository<T> implements GenericRepository<T> {
    private GenericRepository genericRepository;
    //other fields…

    public LogRepository(GenericRepository genericRepository) {
        this.genericRepository = genericRepository;
    }

    @Override
    public void update(T o) {
        saveLog(o); //log this entity's previous state
        genericRepository.update(o);
    }

    //other methods...
}
```

7.4.3 一点变化

备份之前记录的状态需要重新从数据库加载，性能会受到一定影响。为使其达到同样的效果，我们在每次做**创建**和**更新**记录操作时，把这条内存中的记录同时持久化到 user 表和 user_log 表里，虽然日志表里的那条最新的记录永远是多余的，占用了存储空间，但是能保证日志表里跟踪了记录的所有变化，提高了性能（以空间换性能）。

按照这个想法，我们实际没有按照上述给出的代码编写，而是撰写了如下代码：

```
public class LogRepository<T> implements GenericRepository<T> {
    private GenericRepository genericRepository;

    public LogRepository(GenericRepository genericRepository) {
        this.genericRepository = genericRepository;
    }

    @Override
    public void save(T o) {
        genericRepository.save(o);
        saveLog(o);//log this entity's state
    }

    @Override
    public void update(T o) {
        genericRepository.update(o);
        saveLog(o); //log this entity's state
    }

    //other methods...
}
```

这样，我们既扩展了 GenericRepositoryHibernate 和 GenericRepositoryJpa 对象的行为，又没有对它们的类进行任何修改，这就不会影响既有的健壮代码，符合 OCP 原则。

7.4.4 如何使用

现在，要使用记录日志的持久化功能，我们只要运行时装饰 **GenericRepository** 对象即可，代码如下：

```
//instatiate a repository which can log current state
GenericRepository<User> genericRepository = new GenericRepositoryHibernate<User> ();
genericRepository = new LogRepository<User> (genericRepository);
```

对被修饰过的 **GenericRepository** 对象使用起来和以前一样，调用 save(Object o)和 update(Object o)方法就能完成相应的持久化操作。

7.4.5 测试

这里给出save操作的测试代码，其他部分请参见 示例代码：

```
public void testSave() {
    decoratorTest.setup(this);

    //create a new user
    User user = new User();
    user.setName("Jason");
    genericrepository.save(user);
    decoratorTest.commitAndRenewTx();

    //to verify…

    decoratorTest.tearDown();
}
```

这里简单介绍一下测试步骤：

1. 我们首先建立测试环境，即这句 ***decoratorTest.setup(this)***：如果我们使用 Jpa，那么我们会创建 GenericRepositoryJpa 对象并使用 LogRepository 修饰，然后调

用 Jpa 的 API 设置一个事务的保存点（savepoint）；使用 Hibernate 一样。

2. 创建一条 user 数据并提交，即 ***genericrepository.save(user)***，接着我们提交事务并设置下一个保存点，即 ***decoratorTest.commitAndRenewTx()***。

3. 在完成测试验证之后，我们提交数据并撤销资源，即这句 ***decoratorTest.tearDown()***。

已经熟悉 策略（Strategy）模式的读者阅读完代码之后不难发现，我们正是采用了 策略模式撰写了此测试，DecoratorTestDrive类承担Context角色，***DecoratorTest.HBTEST***和***DecoratorTest.JPATEST***是其中的两个具体策略（Concrete Strategy）：在Hibernate的上下文中，使用Hibernate的API设置保存点，提交数据和释放连接资源等，在Jpa的上下文中，使用Jpa的API设置保存点，提交数据和释放连接资源等。对 策略模式有兴趣的读者可以参见本书第 12 章。

7.5 总结

装饰器模式有很多优点：

- 比静态继承更灵活：上述例子中，被 LogRepository 修饰过的任何 GenericRepository 对象都可以记录日志，相比之下，使用静态继承则需要创建两个子类分别实现 Jpa 和 Hibernate 的日志记录功能。
- 接口一致，不影响客户对象的使用：上例中，不管是 save 操作还是 update 操作，使用起来和没有添加功能之前的对象是一样。
- 可以产生叠加效果：可以重复修饰同一个对象，也可以使用不同的装饰器修饰同一个对象，产生叠加的效果。
- 可以作为一种AOP的简单实现方式（参见 AOP的章节），因为我们可以给这类对象的所有方法之前或者之后添加新的功能并控制该方法的调用，但是局限性也比较明显，我们只能动态修改实现该接口的类对象，不能修改其他类型的对象。

它的缺点是：

- 产生了很多看上去很类似的小对象，不宜学习。
- 对象功能与如何动态地修饰这些对象有关，从它们的类型不易判断，不宜排错和调试。
- 一个被修饰过的对象和之前的对象是不同对象，如果有必要，要特别注意覆写 equals(Object object)和 hashCode()方法，最好不要依赖于对象的标识符。

7.6 我们学到了什么

这章我们介绍了 OCP 原则，我们都想做可扩展的应用，因为很容易定制客户的需求，但在开发过程中，很多人只沉浸于逻辑的实现，以为实现这些逻辑的开发就万事大吉，结果为了扩展新功能破坏了以前的健壮代码，导致以前的健壮代码不能正常运行。在持续的开发中，由于扩展性差，添加新功变得能越来越困难，到最后，大家累得筋疲力尽，而软件不是失败就是延期。如果一开始大家就考略可扩展性，就不会出现这样的局面。

在软件开发中，逻辑的实现往往不是最主要的，最主要的可能是我们应该把这些逻辑怎么封装，封装在哪里。

第8章　代理（Proxy）模式

8.1　概述

有时候，需要对实际对象的访问进行控制，我们把这层访问控制封装成一个新的代理对象来代替实际对象，交由客户对象直接访问。

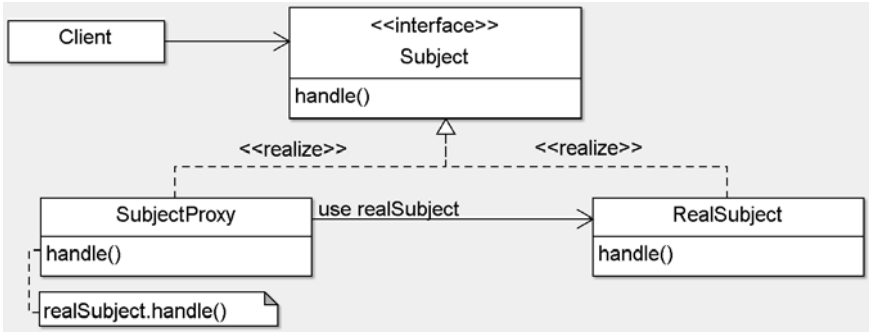
8.2　代理（Proxy）模式

代理（Proxy）模式经常在以下情景中使用：

- 代理一些开销很大的对象，这样便能能迅速返回，进行其他操作，只有在真正需要时才去实例化。
- 安全控制，同步控制，缓存处理结果，缓存初始化开销很大的对象，统计对象的使用以及异常处理等方面。
- 分布式对象访问控制，使得客户对象像使用本地对象一样使用分布式对象。

8.2.1　类图

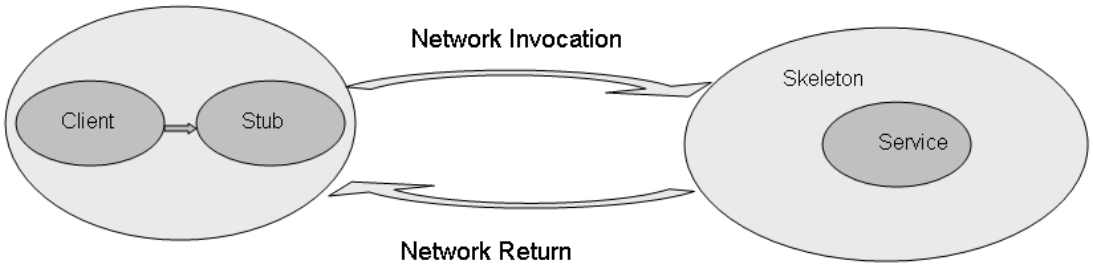
为了保证客户对象透明地使用代理对象，代理类和实际类具有一致的接口，UML 静态类图如下所示：



8.2.2　访问分布式对象

熟悉 EJB 的读者可能非常了解分布式对象通信机制，EJB 的 Stub 正是使用代理模式的好例子，为了不增加学习该模式的难度，我们并不真实打算使用 EJB 来介绍代理模式，而是仿照其原理来创建我们自己的分布式对象进行说明。

我们首先看看分布式对象的实现原理，如下图所示：



Client 对象使用 Stub 代理对象进行远程访问，对 Client 对象来说，由于接口一致，操作远程对象和本地对象并没有什么不同。Stub 对象转发这些操作的请求给远程对象，Skeleton 接收请求返回 Service 对象执行结果，这即是一次远程调用执行的全过程。

我们这里有一个实现了 Service 接口的 ServiceImpl 分布式对象。Service 接口包含一个 hello()方法：

```
public interface Service {
    String hello();
}
```

实现类 ServiceImpl 的 hello()方法非常简单，仅返回"Server says hello!"字符串。代码如下所示：

```
public class ServiceImpl implements Service {
    @Override
    public String hello() {
        return "Server says hello!";
    }
}
```

我们引入了 Service_Stub 代理对象，实现 Service 接口。它将 hello()方法的调用请求发给给远程 Skeleton 对象，Skeleton 将服务对象的方法执行结果写回给 Service_Stub 代理，这样 Service_Stub 对象为客户对象隐藏了远程访问。Stub 的代码片段大致如下所示：

```
public class Service_Stub implements Service {
    @Override
    public String hello() {
```

```
        //variables…
        try {
            //connect to skeleton server…
            socketChannel.register(sel, SelectionKey.OP_WRITE);
            //…
            while (true) {
                //select keys...
                while (it.hasNext()) {//handle each key
                    //remove the key and prepare for next...
                    if (key.isReadable()) {//read the result back from skeleton server
                        if (socketChannel.read((ByteBuffer) byteBuffer.clear()) > 0)
{//read the result
                            byteBuffer.flip();
                            socketChannel.close();//close the channel and return
                            return cs.decode(byteBuffer).toString();//return result
                        }
                    } else {//is writable, write "hello" to skeleton server
                        //write "hello" to skeleton…
                        socketChannel.write(ByteBuffer.wrap("hello".getBytes()));
                        //register a read operatioin to wait for the response…
                        socketChannel.register(sel, SelectionKey.OP_READ);
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (socketChannel != null) socketChannel.close();
                if (sel != null) sel.close();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            return null;
        }
}
```

我们在该例中使用Java NIO实现远程通信，如果有人对NIO不够熟悉，可以参看我们下面的详细解说，有兴趣想深入学习的读者可以参考 附录A的推荐书籍。

Service_Stub通讯过程中，我们用到两种SelectionKey，一种关联一个读操作，即***key.isReadable()==true***，我们把读操作的处理写在***if (key.isReadable()){…}***代码块里。另外一种关联一个写操作，即***key. isWritable()==true***，我们写在***else{…}***代码块里。

Stub 通讯主要分为 3 个步骤：

1. 建立连接：和 Skeleton 服务建立通信连接，并注册一个写操作，即***socketChannel.register(sel, SelectionKey.OP_WRITE)***，这样，接下来 Stub 就可以向服务端发送请求了。
2. 发送请求，即写操作：Stub 向 Skeleton 发送"hello"字符串，然后注册一个读操作，这样 Stub 就开始准备从通信通道 ***socketChannel*** 等待执行结果。
3. 接收返回，即读操作：读取的远程调用返回的结果，关闭此次的通信通道（***socketChannel***），最后返回这次执行的结果。

Service_Skeleton 类继承于 Thread 类，让我们来看看它如何实现的，代码片段大致如下所示：

```
public class Service_Skeleton extends Thread {
    //other fields and methods…

    @Override
    public void run() {
        //...
        try {
```

```
                ***//open a new server socket…***
                ***serverSocketChannel.register(sel, SelectionKey.OP_ACCEPT);***
                //***…***
                while (!finished) {
                    ***//select keys...***
                    while (it.hasNext()) {
                        ***//remove the key and prepare for next...***
                        ***if (skey.isAcceptable())*** {//select a channel to read and write
                            ***ch = serverSocketChannel.accept();***
                            ***ch.configureBlocking(false);***
                            ***//register a read operatioin***
                            ***ch.register(sel, SelectionKey.OP_READ);***
                        } ***else if (skey.isReadable())*** {
                            ch = (SocketChannel) skey.channel();
                            ***//read info from this channel***
                            ***if (ch.read((ByteBuffer) buffer.clear()) > 0)*** {
                                buffer.flip();
                                String method = cs.decode(buffer).toString();
                                ***//if it says "hello", register a write operation***
                                ***if ("hello".equals(method)) {***
                                    ***ch.register(sel, SelectionKey.OP_WRITE);***
                                ***}***
                            }
                        } else {// is writable
                            ***ch = (SocketChannel) skey.channel();***
                            ***// invoke the service.hello method and write the result into the
channel***
                            ***ch.write(ByteBuffer.wrap(service.hello().getBytes()));***
                            ***ch.close();//close the channel***
                            ***finished=true;//quit server***
```

```
                        }
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                try {
                    if (ch != null) ch.close();
                    if (serverSocketChannel != null) serverSocketChannel.close();
                    if (sel != null) sel.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
}
```

Service_Skeleton 循环等待接收客户端发送来的请求，直至处理完成一次 Stub 的服务请求，它的通讯主要分 4 步：

1. 建立 Server 端：监听指定的服务端口，然后注册连接操作，即这句 ***serverSocketChannel.register(sel, SelectionKey.OP_ACCEPT)***，等待 Stub 端的访问。
2. 和客户端建立连接：监听到 Stub 发送的连接请求，建立连接，为此次通信通道注册读操作。
3. 接收请求，即读操作：接收 Stub 发送过来的请求数据，如果接收到的字符串是"hello"，则注册一个写操作准备向客户端发送执行结果。

- 发送结果，即写操作：执行服务对象的 hello()方法，把执行结果写回客户端，关闭此通信通道，最后设置 ***finished=true*** 以关闭服务端。

到此为止，我们已经完成了分布式调用的所有机制。为了等待 Stub 对象的调用，我们必须先启动 Service_Skeleton 服务，启动代码如下：

```
public static void main(String[] args) {
    Service_Skeleton Service_Skeleton = new Service_Skeleton(new ServiceImpl());
```

```
        Service_Skeleton.start();
}
```

现在，让我们看看客户端如何使用，代码如下：

```
public class Client {
        public static void main(String[] args) {
                Service service = new Service_Stub();
                String result = service.hello();
                System.out.println(result);
        }
}
```

我们可以看到，客户端实例化了一个 Srvice_Stub 对象，调用它的 hello()方法，但是它并不知道这个 Stub 对象只是远程对象的一个网络代理，即客户端对远程对象的访问具有透明性，运行结果如下：

```
Server says hello!
```

实际应用中的 EJB 分布式调用比这个复杂，这里只是以其为例介绍代理模式。

8.3 J2SE动态代理

8.3.1 简介

上节介绍了代理模式，其实我们创建了静态代理[14]Service_Stub类，Java还可以支持在运行时创建代理类，即所谓的动态代理。

8.3.2 类和接口

Java 版本自从 1.3 版本开始，就引入了动态代理。在介绍如何使用之前，我们首先讲述一些相关的类和接口：

- **Proxy 类**

  Proxy 类提供了用于创建动态代理类和实例的静态方法，并且，它是你所创建的所有动态代理类的父类。我们看看这个类两个主要方法：

  - public static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces)

[14] 相对于动态代理而言，即在编译期就就产生了 class 文件，决定了运行时的功能。

返回代理类的 java.lang.Class 对象，需要提供类加载器和指定代理的接口数组作为参数。

- public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)

  返回代理对象，需要指定类加载器，指定需要代理的接口数组和 InvocationHandler 对象作为参数。其实这个方法逻辑相当于下面这句：

```
Proxy.getProxyClass(loader, interfaces)
        .getConstructor(new Class[] { InvocationHandler.class })
        .newInstance(new Object[] { handler });
```

- **InvocationHandler 接口**

  InvocationHandler 对象集中控制所有代理的方法，这个接口只有一个方法需要实现：

  - public Object invoke(Object proxy, Method method, Object[] args) throws Throwable

    其中，第一个参数 proxy 是代理类的实例；第二个参数 method 是调用代理实例的 Method 方法实例；第三个参数 args 是调用代理实例的接口方法时所传递的参数数组，如果调用的方法无参数，则传递值为 null，如果是基本类型（Primitive）的参数，会被转化为适当基本包装器类的实例（例如 int 变量，会转化为一个 Integer 对象）。

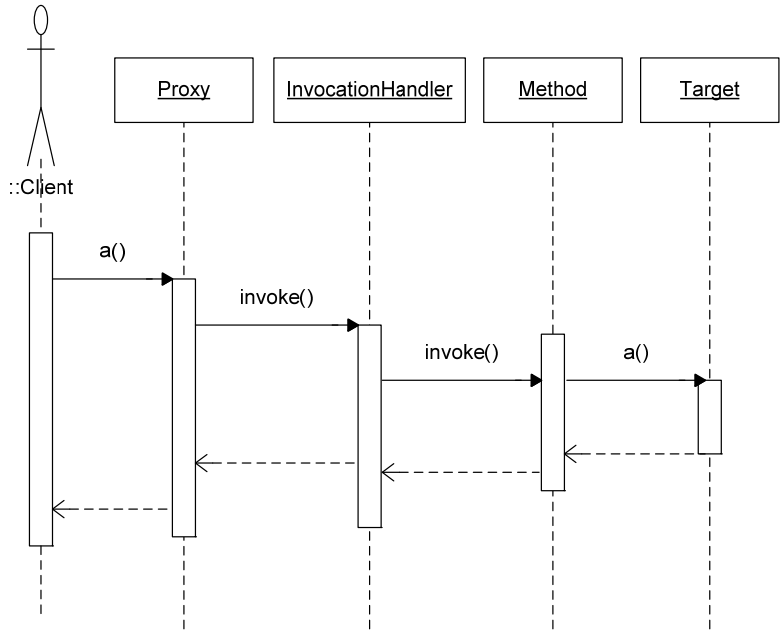
  - 生成的动态代理会首先把请求统一转发给 InvocationHandler 对象的 invoke()方法，如果想要直接调用目标对象 target 的相应方法，则代码如下所示：

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    return method.invoke(target, args);
}
```

8.3.3 调用原理

动态代理类实现了一组运行时确定的接口，客户对象在调用动态代理对象的方法时，它把请求转发给 InvocationHandler 对象的 invoke(Object proxy, Method method, Object[] args)方法，所有方法都被拦截在 InvocationHandler 对象，由 invoke(…)方法里统一处理。

假如调用的接口方法是 a()，则序列图示如下：



8.3.4 实现同步

我们这里有一个简单的接口 Increasable，该接口含有一个 increase(int delta)方法，IncreaseImpl 实现了这个接口，按照传递参数 delta，以其大小从 0 开始增长，增长 max 次之后，我们判断结果是否还是 delta 的整数倍，即这句 count % delta != 0，如果不是，则抛出 IllegalStateException("Count state is illegal!")的异常。Increasable 接口和实现 IncreaseImpl 类的代码如下所示：

```
public interface Increasable {
    void increase(int delta);
}

public class IncreaseImpl implements Increasable {
    private static final int max = 100000;
    private long count;

    @Override
    public void increase(int delta) {
        count = 0;
```

```
            for (int i = 0; i < max; i++) {
                count += delta;
            }

            if (count % delta != 0) {
                throw new IllegalStateException("Count state is illegal!");
            }
        }
}
```

顺序执行同一 IncreaseImpl 对象的方法，是不会有异常抛出的，但是让多个线程，以不同的 delta 参数并发增长，则很可能会抛出异常。我们为此撰写的测试代码如下所示：

```
private void increase(final Increasable increasable) throws Throwable {
    ExecutorService pool = Executors.newFixedThreadPool(2);

    Future f1 = pool.submit(new Runnable() {
        @Override
        public void run() {
            increasable.increase(13);
        }
    });

    Future f2 = pool.submit(new Runnable() {
            @Override
            public void run() {
                increasable.increase(19);
            }
        });

    try {
```

```
            f1.get();
            f2.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            throw e.getCause();
        } finally {
            pool.shutdown();
        }
}
```

我们使用 Executors.newFixedThreadPool(2)生成一个可以包含两个线程的线程池；然后提交两个 FutureTask 对象给线程池，即这句 pool.submit(new Runnable(){…})；接着等待这两个 FutureTask 对象执行完成；FutureTask 对象调用 Increase 对象的 increase(int delta)方法。

由于两个线程使用同一个对象，如果 increase(int delta)方法不是线程安全的，那么很容易抛出 IllegalStateException 异常。这样，我们在此测试方法里就会捕捉到 ExecutionException 异常，该异常的 cause 应该 increase(int delta)方法方法刚刚抛出来 IllegalStateException 异常；反之，如果 increasable 是线程安全的，则不会抛出 ExecutionException 异常。

非线程安全的测试方法如下：

```
public void testIncrease() {
    boolean illegalState = false;
    while (!illegalState) {
        try {
            Increasable increasable = new IncreaseImpl();
            increase(increasable);
        } catch (Throwable e) {
            //assert expected exception message
```

```
                assertEquals("Count state is illegal!", e.getMessage());
                illegalState = true;
            }
        }
        assertEquals(illegalState, true);
}
```

IncreaseImpl 对象是非线程安全的，那么最终必然会抛出异常消息为***"Count state is illegal!"***的异常。由于能否抛出异常与执行有很大关系，我们使用循环多运行几次，直至抛出异常。

现在，我们使用动态代理为 IncreaseImpl 对象创建一个运行时安全的代理对象，我们只要在调用 increase(int delta)方法时为其加上这个对象为同步锁，代码如下所示：

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import static java.lang.reflect.Proxy.newProxyInstance;

public class SynchronizedHandler implements InvocationHandler {
    private Increasable delegate;

    public static Increasable newProxy(Increasable delegate) {
        Increasable o = (Increasable) newProxyInstance(
                                        delegate.getClass().getClassLoader(),
                                        delegate.getClass().getInterfaces(),
                                        new SynchronizedHandler(delegate));
        return o;
    }

    public SynchronizedHandler(Increasable delegate) {
        this.delegate = delegate;
    }
```

```
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        synchronized (delegate) {
            return method.invoke(delegate, args);
        }
    }
}
```

当调用目标对象的 increase(int delta)方法时，我们添加了 ***synchronized (delegate){…}*** 同步代码块。

为了验证我们的想法，我们编写了方法 testSynchronizedIncrease()，把线程安全的代理对象传递给 increase(final Increasable increasable)方法，循环 **max** 次。测试代码大致如下所示：

```
public void testSynchronizedIncrease() {
    Increasable increasable = newProxy(new IncreaseImpl());
    for (int i = 0; i < max; i++) {
        try {
            increase(increasable);
        } catch (Throwable e) {
            //fail to test, throw a new exception
            throw new RuntimeException("Synchronized Test Failed", e);
        }
    }
}
```

为启动我们线程安全的和非线程安全的测试，我们的代码如下：

```
public static void main(String[] args) {
    TestDrive test = new TestDrive(157);//run synchronizedIncrease max=157 times
    System.out.println("Test non-thread-safe increment...");
    test.testIncrease();
```

```
        System.out.println("Test non-thread-safe increment successful.");


        System.out.println("Test thread-safe increment...");
        test.testSynchronizedIncrease();
        System.out.println("Test thread-safe increment successful.");
}
```

其实不管 **max** 值为多少，线程安全的和非线程安全的测试执行结果都一样，如下所示：

```
Test non-thread-safe increment...
Test non-thread-safe increment successful.
Test thread-safe increment...
Test thread-safe increment successful.
```

8.3.5 总结

Java 动态代理只能代理接口。感兴趣的读者可以借助如下窍门反编译动态生成代理类的二进制代码了解动态生成的代理类：

- 小窍门

  如果读者使用 Sun 公司 JDK 做开发，则可以使用 sun.misc.ProxyGenerator.generateProxyClass(String s, Class aclass[])方法生成代理类的二进制 class 代码，然后反编译这个 class 文件一窥究竟。

其实创建动态代理方式不只有Jdk提供的这一种方式，很多动态字节码工具都可以生成动态代理。例如，使用 CGLib就可以，它不仅可以动态代理接口，还可以动态代理具体的类。由于在运行时创建代理类非常灵活，动态代理在事务管理，AOP（Aspect-Oriented Programming，切面编程）实现等方面发挥了出色的作用。

8.4 代理（Proxy）模式与装饰器（Decorator）模式

代理模式和装饰器模式非常相似，很多人甚至把这两者看做是同一种模式，的确，这二者区别并不太明显，他们都提供了到目的对象方法的间接过渡，但它们的设计目的不同：

- 代理模式并不实现和目标对象类似的业务功能，而是一些与目标对象功能联系不太紧密的职责，也就是说，二者处理的是不同的问题域。比如权限控制，连

接服务端，处理异常，缓存目标对象等，这些与目标对象的功能不是同一类的；而装饰器对象处理的问题与目标对象处理的问题都是同一问题域，目的是增加目标对象那方面的能力，例如 Java IO 的 BufferedInputStream 和修饰对象 InputStream 都是为实现流读取设计的，它使得 InputStream 可以缓冲读取数据。

- 装饰器动态地为目标对象添加功能或者减少功能，还可以递归地装饰目标对象，产生叠加的效果，而代理模式不做此用途。

8.5 总结

代理模式在安全性控制，线程同步控制，缓存执行结果，代理创建一些开销很大的对象（例如，Hibernate 使用延迟加载创建一些代理对象）等方面应用广泛，由于没有破坏现有的类而扩展了它们的功能，是 OCP 原则的又一成功应用。

第9章　适配器（Adapter）模式

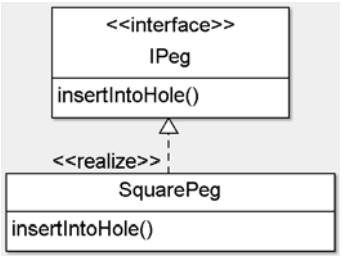
9.1　概述

如果你去美国出差，你带的笔记本电脑插头是中国标准的，就没办法直接连接他们的电源插孔。一般，你不会为你的笔记本电脑再去买一个新的可以连接美国插座的电源，往往我们买一个插座转换器，让这个转换器连接美国插座，而你的插座连接这个转换器就可以了。

现实世界中，我们有数不尽的适配器，例如交直流转换器，PS2-USB 的鼠标/键盘接口转换线，还有在汽车上用于给手机、ipod 等充电用的电源转换器。在编程世界，我们也经常碰到类似的情况：我们需要把一个接口转换成另外接口，以此使得客户对象能够继续使用这类对象而不需任何改变，我们把这个转换类就叫做适配器。我们这里以打桩为场景来介绍适配器模式。

9.2　打桩

我们需要建造一幢高楼，需要打方形桩，这里抽象出一个 IPeg 接口，SquarePeg 是它的实现类，如下 UML 静态类图所示：

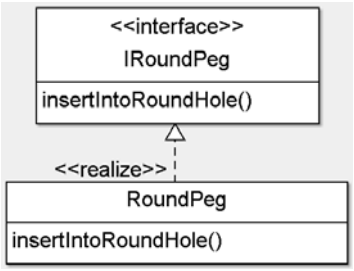


我们来编写它们的模拟代码，如下所示：

```
public interface IPeg {
    void insertIntoHole();
}

public class SquarePeg implements IPeg {

    @Override
```

```
    public void insertIntoHole() {
        System.out.println("I'm inserting into square hole...");
        //other logic…
    }
}
```

现在我们还需要打圆形桩，实现圆形桩不是一件容易的事情，所幸的是，我找到了一个 RoundPeg 类，它实现了 IRoundPeg 接口的 insertIntoRoundHole()的方法。UML 静态类图如下所示：



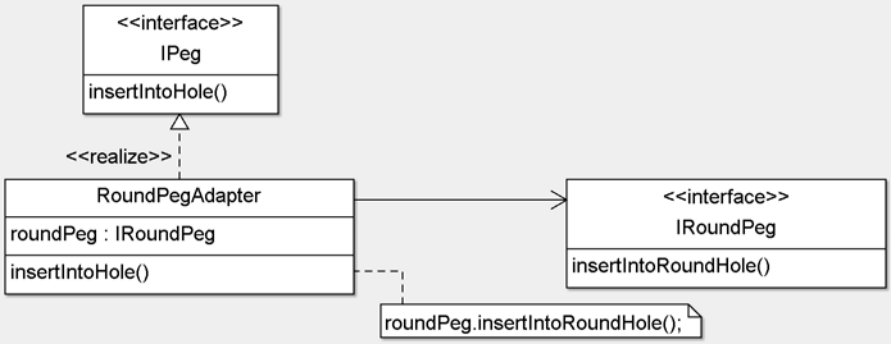
模拟代码如下所示：

```
public interface IRoundPeg {
    void insertIntoRoundHole();
}

public class RoundPeg implements IRoundPeg {
    @Override
    public void insertIntoRoundHole() {
        System.out.println("I'm inserting into round hole...");
        //other logic…
    }
}
```

尽管 RoundPeg 类实现了 insertIntoRoundHole()的逻辑，但是客户对象不能像使用 SquarePeg 对象一样直接使用 RoundPeg 对象，因为它们的接口不一致，RoundPeg 的接口不是 IPeg。

现在，轮到适配器模式上场了，我们为此撰写一个适配器类，使它实现 IPeg 接口，

但是它除过把 insertIntoHole()请求转发给 IRoundPeg 的 insertIntoRoundHole()方法处理之外，自己不做任何额外的逻辑，不难画出 UML 静态类图，如下所示：



IPeg 接口是我们的 Target（目标）接口，IRoundPeg 是 Adaptee 接口（被适配的接口），而 RoundPegAdapter 就是 Adapter 类（适配器类）。

适配器的代码非常简单，如下所示：

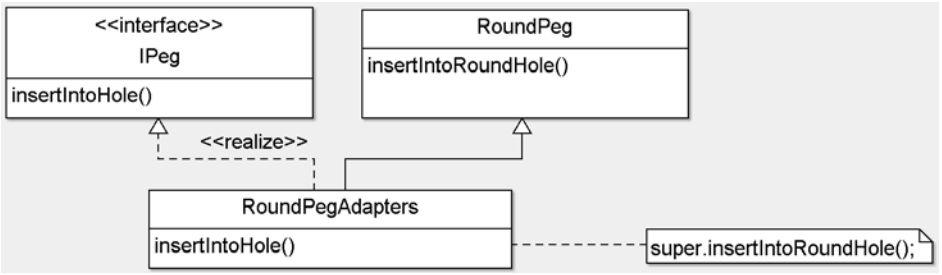
```
public class RoundPegAdapter implements IPeg {
    private IRoundPeg roundPeg;

    public RoundPegAdapter(IRoundPeg roundPeg) {
        this.roundPeg = roundPeg;
    }

    @Override
    public void insertIntoHole() {
        roundPeg.insertIntoRoundHole();
    }
}
```

9.3 其他适配器模式

9.3.1 类适配器

上面我们实现的适配器称为**对象适配器**（**Object Adapter**），RoundPegAdapter 适配器依赖于 IRoundPeg 对象，可以说，我们是使用了合成的方式重用了 IRoundPeg 的

功能，除了使用合成外，我们还可以使用继承的方式实现适配器，UML 静态类图如下所示：



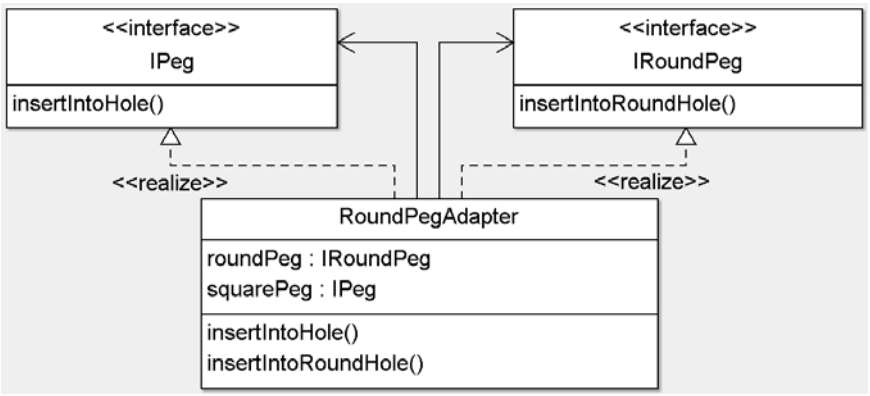
RoundPegAdapter2 类继承于 RoundPeg 类并实现 IPeg 接口，而 insertIntoHole()方法把请求转发给给父类的 insertIntoRoundHole()方法，即，

```
public class RoundPegAdapter2 extends RoundPeg implements IPeg {
    @Override
    public void insertIntoHole() {
        super.insertIntoRoundHole();
    }
}
```

我们把使用继承方式实现的适配器称之为**类适配器**（**Class Adapter**），这种情况在一些支持多继承的语言中（比如 C++，Python 等），应用的比较多，在不支持多继承的语言里应用的并不是很广泛。

9.3.2 双向适配器

有时候，我们希望能够实现一个相互转换的适配器，即**双向适配器**（**Two-way Adapter**），双向适配器要同时实现这两个接口，以满足不同的客户处理方式，于是 UML 静态类图如下所示：



不难写出如下代码：

```
public class TwoWayPegAdapter implements IRoundPeg, IPeg {
    private IPeg squarePeg;
    private IRoundPeg roundPeg;

    public TwoWayPegAdapter(IPeg squarePeg) {
        this.squarePeg = squarePeg;
    }

    public TwoWayPegAdapter(IRoundPeg roundPeg) {
        this.roundPeg = roundPeg;
    }

    @Override
    public void insertIntoRoundHole() {
        squarePeg.insertIntoHole();
    }

    @Override
    public void insertIntoHole() {
        roundPeg.insertIntoRoundHole();
    }
}
```

9.4 测试

终于到我们测试的时刻了，测试代码如下所示：

```
public class TestDrive {
    public static void main(String[] args) {
        TestDrive test = new TestDrive();
```

```
        IPeg squarePeg = new SquarePeg();
        IRoundPeg roundPeg = new RoundPeg();
        RoundPegAdapter adpater = new RoundPegAdapter(roundPeg);

        System.out.println("Testing square peg...");
        test.testPeg(squarePeg);
        System.out.println("\nTesting square adapter peg...");
        test.testPeg(adpater);

        TwoWayPegAdapter roundPeg2 = new TwoWayPegAdapter(roundPeg);
        TwoWayPegAdapter squarePeg2 = new TwoWayPegAdapter(squarePeg);
        System.out.println("\nTesting a 2-way square adapter peg...");
        test.testPeg(roundPeg2);
        System.out.println("\nTesting    2-way round adapter peg...");
        test.testRoundPeg(squarePeg2);
    }

    private void testPeg(IPeg peg) {
        peg.insertIntoHole();
    }

    private void testRoundPeg(IRoundPeg peg) {
        peg.insertIntoRoundHole();
    }
}
```

测试结果如下所示：

```
Testing square peg...
I'm inserting into square hole...

Testing square adapter peg...
```

```
I'm inserting into round hole...


Testing 2-way square adapter peg...
I'm inserting into round hole...


Testing 2-way round adapter peg...
I'm inserting into square hole...
```

9.5 适配器（Adapter）模式与代理（Proxy）模式

适配器模式和代理模式都是把请求转发给目标对象来处理，但是实现的接口不一样：代理模式的代理类和被代理类（目标类）都实现了相同的接口，而适配器实现新的满足客户类的接口（目标接口），实现兼容客户类的目的。

第10章　外观（Facade）模式

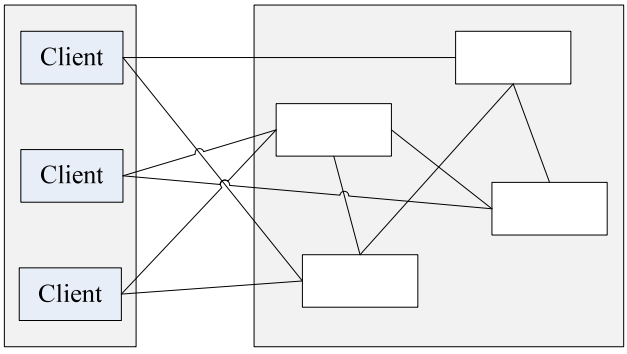
10.1 概述

外观模式在现实世界中比比皆是，比如你想吃蛋糕，你没有必要去买面粉，鸡蛋，奶油等，然后和面加工等，你只要拿着钱去蛋糕店订做一个就好了；另外，以开机为例，你没必要在打开电源之后，输入一些指令，使 CPU 开始运行，然后检测内存和硬盘，加载硬盘数据，检测鼠标键盘设备……，只要你按一下开关，操作系统就会为你做完了这一切。蛋糕店和操作系统都为我们提供了外观，使我们我们没必要了解蛋糕是怎样制作的，电脑是按照哪些步骤启动的，我们直接使用它们就可以了。

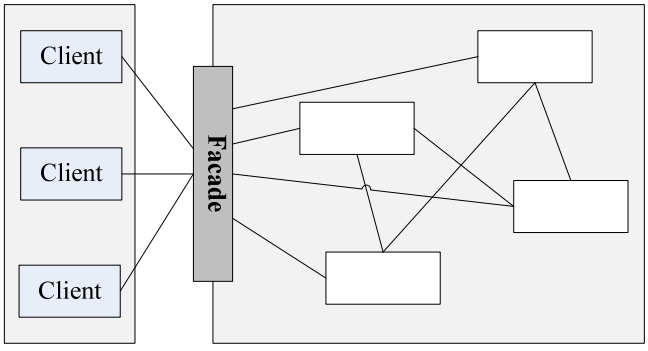
在软件设计里，如果软件系统和一些子系统有非常繁复的交互，你可能希望减少和它们之间的交互达到简化的目的，这就使用到了外观模式。外观模式是我们最常使用的设计模式之一，非常直观和简单，读者可能已经在使用了，只不过可能还不知道它就是外观模式。

10.2 外观（Facade）模式

一个系统在使用外观模式之前，客户类和子系统的类之间的交互关系如下图所示：



引入外观类之后，它们之间的交互关系就变为下图所示：



10.3 Least Knowledge Principle（最少知识原则）

外观模式用到了一条非常有用的设计原则：最少知识原则（Least Knowledge Principle），也称为迪米特原则（Law of Demeter）：

> **Only talk to your immediate friends, don't talk to strangers.**
>
> **只和你的朋友说话，不要和陌生人说话。**

如果你的应用有很多类，并且它们有复杂的依赖关系，那么，你的应用将很难维护和扩展，因为对其中一处的改动，会影响很多使用者。迪米特原则让我们减少和其他类的依赖关系，这样，一处改动只影响少很小的范围，降低了风险。

在外观模式里，我们添加了 Facade 类，高层次的类和 Facade 类直接打交道，不和低层次的类直接交互，这样，低层次类的变化不会影响高层次的类，从而降低了客户类和子系统类之间的耦合度。

10.4 懒惰的老板请客

这里给出一个老板请客吃饭的场景，老板发现大家最近干活很努力，准备犒劳一下大家，老板想了一下整个过程，大致需要做如下事情：

> **预定桌子：subscribe()**
>
> **点菜：waitForAnOrder()**
>
> **做菜：cookDish()**
>
> **老板讲话：address()**
>
> **上菜：serveDishes()**
>
> **结账：check()**

“原来请人吃饭也要这么麻烦”，老板嘀咕着，抬起头正好看见了助理，他就索性把请客的事情全部交给助理去办了。

这一个是虚拟的场景，或许你已经是老板，让助理处理这些事情也早已习以为常了，那么，这里提到的助理正是充当了 Façade 的角色。有了助理，老板请客吃饭就简单了许多：

助理准备晚餐：prepareDinner()

老板发表讲话：address()

助理结束晚餐：endDinner()

助理的模拟代码片段大致如下所示：

```
public class Assistant {
        //other methods…
    public void prepareDinner() {
        hotelReceptionist.subscribe();
        waitress.waitForAnOrder();
        cook.cookDish();
    }

    public void endDinner() {
        waitress.serveDishes();
        cashier.check();
    }
}
```

在 ***prepareDinner()***方法中，助理预定位置，点菜并吩咐厨师做饭；在 ***endDinner()*** 方法中，助理负责让服务员上菜，最后结账。这样，模拟老板的代码如下所示：

```
public class Boss {
    private Assistant assistant;

    public void treat() {
        assistant.prepareDinner();
        address();
```

```
            assistant.endDinner();
        }
        //constructors and methods…
}
```

老板除了自己发言之外，其他事情都交给助理去处理了，我们的测试程序如下所示：

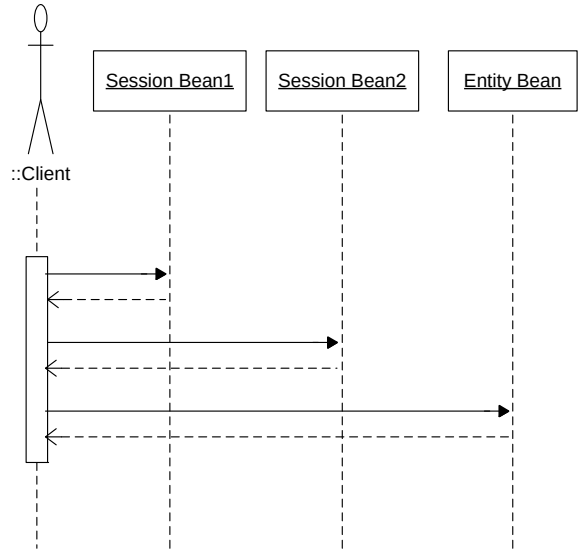
```
public static void main(String[] args) {
        Cashier cashier = new Cashier();
        Cook cook = new Cook();
        HotelReceptionist hotelReceptionist = new HotelReceptionist();
        Waitress waitress = new Waitress();
        Assistant assistant = new Assistant(hotelReceptionist, cook, waitress, cashier);

        Boss boss = new Boss(assistant);

        boss.treat();
}
```

测试的执行结果如下所示：

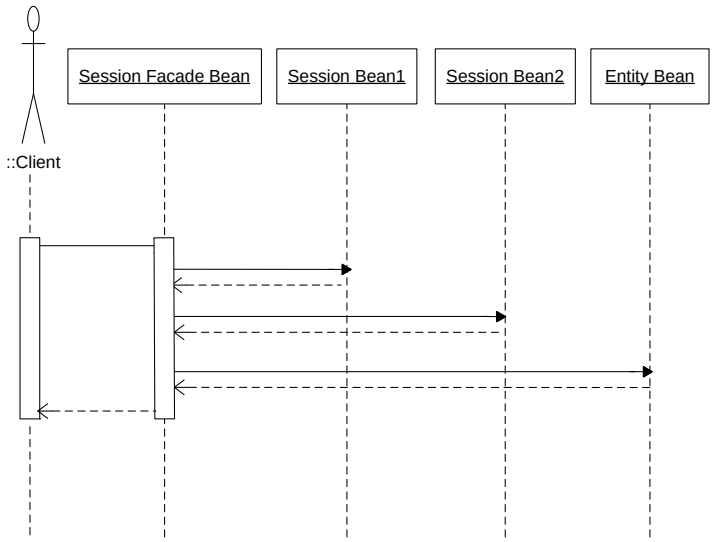
```
Subscribe a table...
Waiting for the order...
Cooking dishes...
Boss is bitching : "Tomorrow is going to be better, we will make blah blah..."
Serving dishes...
Check the bill...
```

10.5 EJB里的外观模式

使用过 EJB 做开发的人应该知道**会话外观（Session Facade）模式**，比如，Client 对象需要调用远程对象 Session Bean1，Session Bean2 和 Entity Bean 来完成特定功能，使用会话外观模式之前的序列图如下所示：



由于 Session Bean1，Session Bean2 和 Entity Bean 都是远程对象，Client 对象要完成一个操作需要三次远程调用。这三次调用都要消耗网络带宽，如果引入我们刚才讲的外观模式，在远程服务端创建一个会话对象，Session Facade Bean，我们就可以减少与远程对象的通信次数，节省了带宽。引入之后，序列图如下所示：



Session Facade Bean 负责与 Session Bean1，Session Bean2 和 Entity Bean 交互，Client 对象不直接和它们交互，这样，Client 对象完成这个操作，和 Session Facade Bean 只进行一次远程通讯就可以了，大大的减少了网络通讯的代价。

除了提高网络通信效率外，和基础的外观模式一样，会话外观模式减少了客户对象

和服务对象之间的依赖程度，把低层服务对象的接口和它们之间的复杂依赖和交互隐藏起来通过 Session Facade bean 隐藏起来，降低了客户端和服务端的耦合性。

由于会话外观模式和普通的外观模式没有太大区别，是其在 EJB 里的一个应用而已，这里就不再撰写代码演示如何使用了，有兴趣的读者可以自己实现。

10.6 总结

外观封装了子系统之间复杂的交互和依赖关系，为客户对象提供了单一简单的界面，降低了系统的复杂性。在讲解外观模式同时，介绍了 最少知识原则，我们知道，类之间耦合度越低，越易扩展和实现重用。

第11章　组合（Composite）模式

11.1 概述

我们在开发过程中经常使用到树形结构，它分为叶子节点和和分支节点两种，客户对象在使用这两种对象时候经常要对它们加以区别，增加了代码的复杂度，也非常容易出错。组合模式为这两种类型提供了统一的接口，可以让我们像操作叶子节点那样方便地操作分支节点。

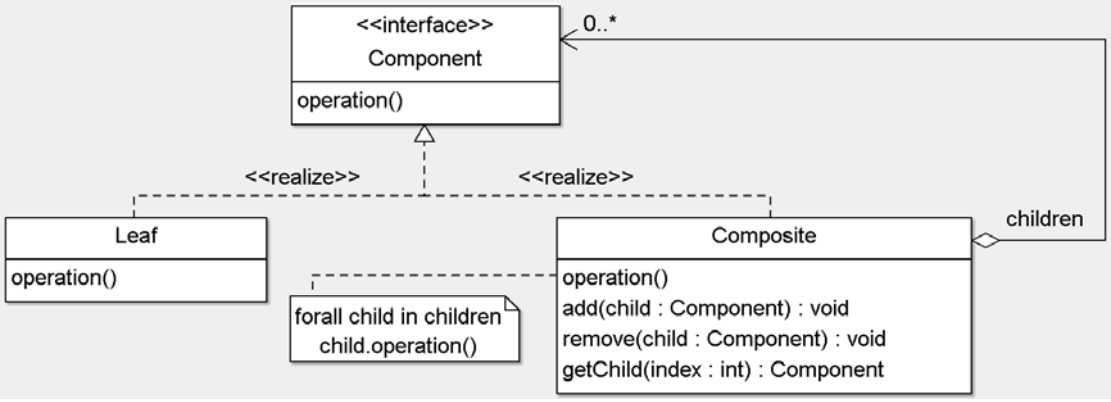
11.2 组合模式

关于组合模式，GoF 给出的定义是：

> **Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.**
>
> **将对象组合成树形结构来表现部分-整体的层次关系。组合使得客户一致地使用单个对象和组合对象。**

要使客户程序对分支节点和叶子节点进行一样的操作，那么它们外形必须一样，即得有相同的接口隐藏具体实现，下面我们先从 UML 静态类图入手。

11.2.1 类图



叶子类 Leaf 和分支类 Composite 都实现了 Component 接口，这样就保证了访问的一致性，此外，还要注意以下两点：

- Component 接口的 operation()方法的实现逻辑一般是：如果这个节点如果是叶子结构，则执行自己的 operation()逻辑；如果是分支结构，除过执行自己的那部分

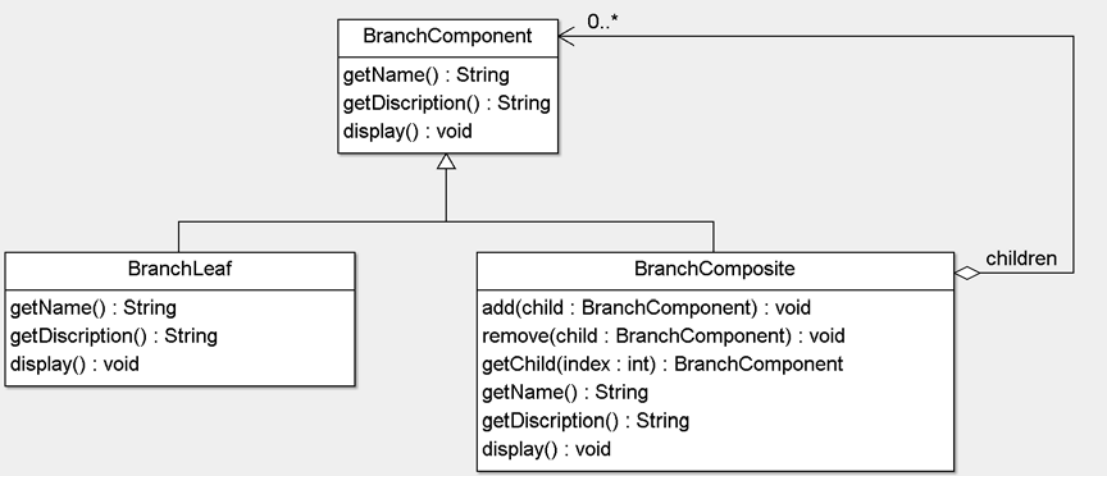
逻辑外，还要递归地调用所有子节点的 operation()方法。

- Composite 除了 operation()操作外，还要提供管理子节点的操作，例如 add，remove，getChild 等操作。

11.2.2 使用组合（Composite）模式

银行一般有很多分行，这些分行分布在不同的地方，一个大的分行下面还有其他分支机构。为了显示银行的组织结构，我们使用组合模式，首先为抽象父类 BranchComponent 定义一个 display()方法。它还包括 getName()和 getDiscription()两个方法分别用以得到银行的名称和描述。

UML 静态类图如下所示：



BranchComponent 类即为我们的 Component 类（这里未使用接口），供分支类 BranchLeaf 和叶子类 BranchComposite 继承。

BranchComponent 类的代码如下所示：

```
public abstract class BranchComponent {
    public String getName() {
        throw new UnsupportedOperationException();
    }

    public String getDiscription() {
```

```
        throw new UnsupportedOperationException();
    }

    public void display() {
        throw new UnsupportedOperationException();
    }
}
```

分支类 BranchComposite 的代码片段大致如下所示：

```
public class BranchComposite extends BranchComponent {
    public BranchComposite(String name, String discription) {
        //…
    }
    // other field and methods…

    public void display() {
        System.out.printf("%s: %s\n", name, discription);
        for (BranchComponent child : childrenBranch) {
            child.display();
        }
    }

    public void add(BranchComponent child) {
        childrenBranch.add(child);
    }

    public void remove(BranchComponent child) {
        childrenBranch.remove(child);
    }

    public BranchComponent getChild(int i) {
```

```
            return childrenBranch.get(i);
        }
}
```

BranchComposite类的display()方法除了显示自己的信息外，还要递归地显示所有子节点的信息，我们使用for-each循环[15]显示子节点信息：

```
for (BranchComponent child : childrenBranch) {
        child.display();
}
```

它还包含管理子节点的方法：add(BranchComponent child)，remove(BranchComponent child)和 getChild(int i)。

叶子类 BranchLeaf 的代码片段大致如下所示：

```
public class BranchLeaf extends BranchComponent {
        //fields and constructors…

        public void display() {
                System.out.printf("\t%s: %s\n", name, discription);
        }

        //other methods…
}
```

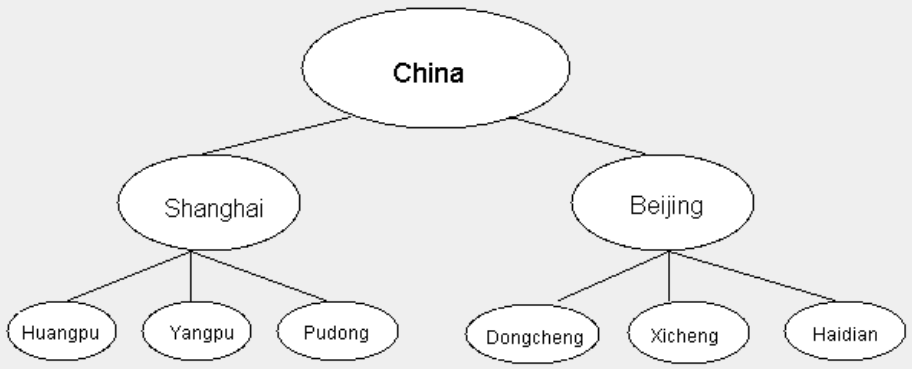
11.2.3 测试

这样，显示分支分行结构和显示叶子分行分行的操作一样，我们编写如下方法用以显示银行结构：

```
private static void display(BranchComponent branch){
    branch.display();
}
```

[15] Java 5 及以上版本的新的语法，参见附录 A 的推荐。

现在假定某银行的组织结构如下所示：



我们编写测试代码表示上述结构，代码如下：

```
public class TestDrive {

    public static void main(String[] args) {

        BranchComposite china = new BranchComposite("CN", "China Branch");


        BranchComposite shanghai = new BranchComposite("Sh", "Shanghai
Branch");
        BranchLeaf huangpu = new BranchLeaf("Hp", "Huangpu Branch");
        BranchLeaf yangpu = new BranchLeaf("Yp", "Yangpu Branch");
        BranchLeaf pudong = new BranchLeaf("Pd", "Pudong Branch");


        BranchComposite beijing = new BranchComposite("Bj", "Beijing Branch");
        BranchLeaf dongcheng = new BranchLeaf("Dc", "Dongcheng Branch");
        BranchLeaf xicheng = new BranchLeaf("Xc", "Xicheng Branch");
        BranchLeaf haidian = new BranchLeaf("Hd", "Haidian Branch");


        shanghai.add(huangpu);
        shanghai.add(yangpu);
```

```
            shanghai.add(pudong);


            beijing.add(dongcheng);
            beijing.add(xicheng);
            beijing.add(haidian);


            china.add(shanghai);
            china.add(beijing);


            System.out.println("Displaying the head bank information");
            display(china);


            System.out.println("\nDisplaying Shanghai bank branch information");
            display(shanghai);


            System.out.println("\nDisplaying Pudong bank branch information in
Shanghai");
            display(pudong);
        }


        //other methods…
}
```

我们显示中国总行，上海分行和浦东子行的结构，看看执行效果如何，如下所示：

```
Displaying the head bank information
CN: China Branch
Displaying the head bank information
```

```
CN: China Branch
Sh: Shanghai Branch
    Hp: Huangpu Branch
    Yp: Yangpu Branch
    Pd: Pudong Branch
Bj: Beijing Branch
    Dc: Dongcheng Branch
    Xc: Xicheng Branch
    Hd: Haidian Branch


Displaying Shanghai bank branch information
Sh: Shanghai Branch
    Hp: Huangpu Branch
    Yp: Yangpu Branch
    Pd: Pudong Branch


Displaying Pudong bank branch information in Shanghai
    Pd: Pudong Branch
```

11.3 透明的组合模式

在上述结构中，我们对于子节点的管理操作（add，remove 和 getChild 等操作）放在了分支类 BranchComposite 类里，叶子类 BranchLeaf 类看不到这些操作的定义。我们还有一种方式，把管理子节点的操作放在 Component，然后让叶子类的这些操作在运行时失效（可以不做任何处理，也可以抛出异常等），这样对客户对象来说，这种方式更加透明。我们把前者称为安全的组合模式，因为叶子节点没有 add，remove 和 getChild 这样的操作，如果尝试调用这些方法，在编译时就会报错；后者在运行过程中才可能被发现，但是这种方式具有良好的透明性，我们称之为透明的组合模式。

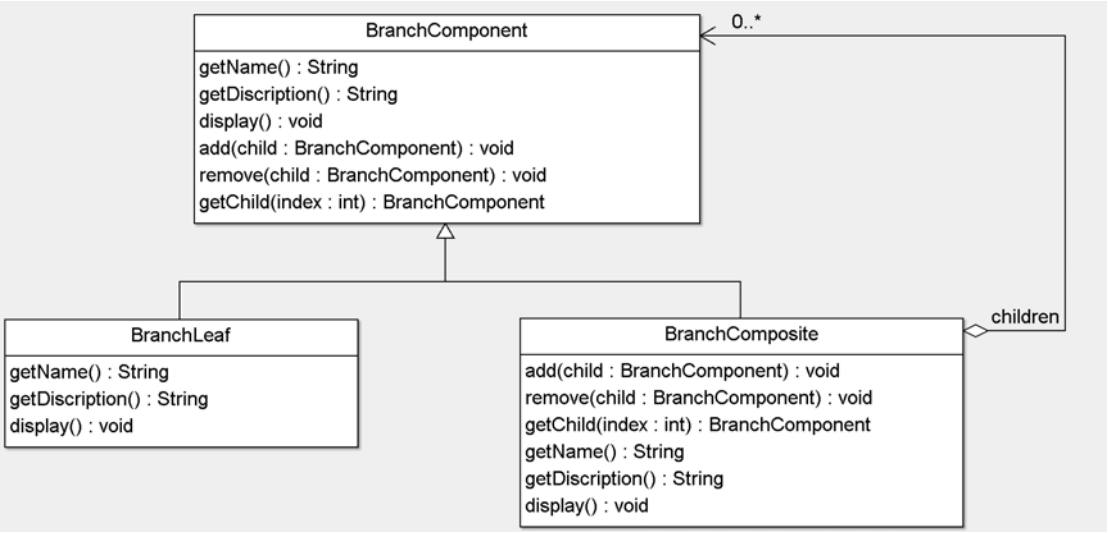
透明的组合模式对叶子节点调用 add 等操作的处理有两种方式：运行时报错，例如抛出一个异常等；忽略这些无意义的操作，不做任何逻辑处理，使用没有自动内存

回收机制的语言来说，使用这种方式要特别注意垃圾回收。

我们现在就为上述场景重构透明的组合模式，首先我们需要把以前子类 BranchComposite 类的 add，remove 和 getChild 操作放上移至父类 BranchComponent。为了让叶子节点的这些操作失效，我们有 2 种选择：

1. 父类 Component 中声明这些方法为 abstract，然后分别在 Composite 子类和 leaf 子类中实现这些操作，但是 Leaf 类在这些操作中只能实现失效的逻辑。
2. 父类中就实现这些失效的操作，那么只在 Composite 子类中覆写这些操作。

我们这里选用第 2 种方式，使用 BranchComponent 类的 add，remove 和 getChild 操作会抛出异常，这样前述的 BranchComposite 类和 BranchLeaf 类的代码则不需要发生任何改变。UML 图示如下：



BranchComponent 类的代码大致如下：

```
public abstract class BranchComponent {

    //other methods…


    public void add(BranchComponent child) {

        throw new UnsupportedOperationException();

    }


    public void remove(BranchComponent child) {
```

```
            throw new UnsupportedOperationException();
        }


        public BranchComponent getChild(int i) {
            throw new UnsupportedOperationException();
        }
}
```

测试代码和执行非常简单，这里就不再赘述，详细请参见 示例代码。

11.4 安全的组合模式VS透明的组合模式

安全的组合模式编译时就可以发现为叶子节点执行 add 等无意义的管理子节点的操作，但造成了 Leaf 和 Composite 接口不一致，损失了透明性。透明的组合模式只有在运行时才**可能**（如果不忽略的这些无意义调用的话）暴露错误地使用叶子节点的操作，牺牲了安全性，具有良好的透明性。这二者各有优缺点，读者在实际编程中需要自己权衡来决定应该使用哪种方式。

11.5 还需要注意什么

- 执行顺序：如果子节点的执行是有序的，特别要注意对子节点管理和访问的实现；另外还要注意父节点操作和子节点操作的执行顺序，例如上述实现中，我们总是先显示父行信息，然后依次显示子行信息。
- 存储结构：存储子节点的结构可以是 Hash 表，数组，链表，树等任何集合结构，读者在实际操作中，可以自行选择效率最高的结构。
- 缓存：Composite 可以按需提供高速缓存等方式，以提升对子节点的遍历等相关操作的性能。但是注意，如果子节点发生了变化，父节点关于子节点的缓存也随之失效，在执行这些操作时，要提供一种方式告诉父节点缓存已经失效，一般我们可以为子节点定义指向一个父节点引用，这样便于使用父节点缓存失效的相关操作。

第四篇 行为模式

这篇将介绍一些常用的行为模式，它们描述如何控制执行的流转以及对象之间如何交互。在行为模式中，我们把行为中变化的部分进行了封装。我们前面提到的模板方法模式就是行为模式的一种，它把算法进行了分解，子类封装了那些可变的步骤。

在这篇，我将继续为大家讲解三个常用的行为模式，带领大家一起领悟 OOP 给我们带来的好处，特别是封装带来的好处。

第12章　策略（Strategy）模式

12.1 既要坐飞机又要坐大巴

还记得我们第 2 章给出了的回家过年的例子吗？我们使用模板方法模式解决了代码重复的问题，我们现在又有新的需求了，如果Jennifer先要坐飞机，然后坐大巴才能回到家，考略使用模板方法模式解决这个问题，我们必须增加了一个新类PassengerByAirAndCoach，表示既要坐飞机又要坐大巴回家的这类人，那么，代码片段如下所示：

```
public class PassengerByAirAndCoach extends HappyPeople…
    @Override
    protected void travel() {
        //Traveling by Air…
        System.out.println("Travelling by Air...");

        //Travel by Coach…
        System.out.println("Travelling by Coach...");
    }
}
```

让我们看看黑体加粗的部分，不难发现，该死的代码重复又摆在我们面前了，在类 PassengerByAir 和类 PassengerByCoach 你会发现这些黑体加粗的代码。这或许只是麻烦的开始，试想如果我们还需要实现坐火车再坐大巴回家的这类人，坐大巴再坐火车回家的这类人，坐大巴再坐大巴……的人，别再想下去了，重复的噩梦已经开始了。

继承虽然解决了第 2 章提到的简单需求，避免了代码重复，但是对于现在的需求而言，它仍然带来了“代码重复”的臭味。

12.2 封装变化

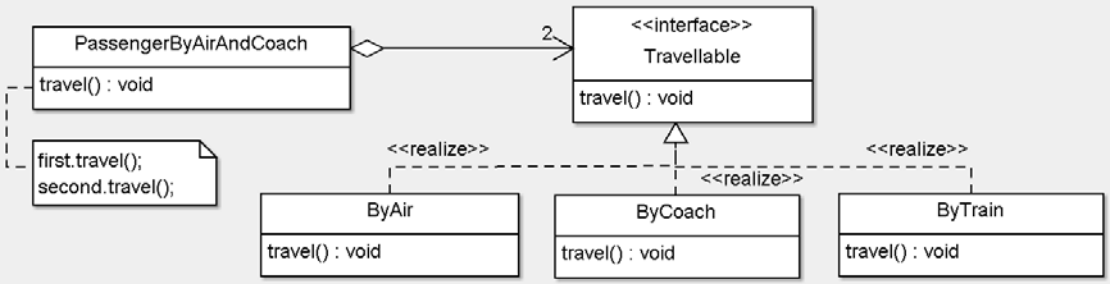
我们先给出一条 OOP 关于封装变化的设计原则，如下所示：

Identify the aspects of your application that vary and separate them from what stays the same.

找出你的应用程序变化的那些方面，并把它们和那些不变的部分分开。

封装这些变化，这样，你可以改变或者扩展那些可变的部分，而不去改变那些不变的部分，改变和扩展可变的部分就不会影响那些不变的部分。

由于回家的方式会发生变化，我们之前按照回家过年人的种类把它进行了封装，但是这样种类太多了，其实我们发现更合理的方法是把回家的方式进行封装，而非按照人的种类封装。于是我们抽象出 Travellable 接口，把坐飞机，坐大巴等回家的方式封装在各自的实现类里，于是，UML 静态类图示如下：



ByAir 类，ByCoach 类和 ByTrain 类实现了 Travellable 接口，PassengerByAirAndCoach 对象含有 2 个 Travellable 对象：ByAir 对象和 ByCoach 对象。

Travellable 接口及其实现类的代码如下：

```
public interface Travellable {
    void travel();
}

public class ByAir implements Travellable {
    @Override
    public void travel() {
        //travel logic...
        System.out.println("Travelling by Air...");
```

```
    }
}

public class ByCoach implements Travellable {
    @Override
    public void travel() {
        //travel logic...
        System.out.println("Travelling by Coach...");
    }
}

public class ByTrain implements Travellable {
    @Override
    public void travel() {
        //travel logic...
        System.out.println("Travelling by Train...");
    }
}
```

接下来，我们来看看 PassengerByAirAndCoach 类如何实现，代码如下：

```
public class PassengerByAirAndCoach extends HappyPeople {
    private Travellable first;
    private Travellable second;

    public PassengerByAirAndCoach() {
        first = new ByAir();
        second = new ByCoach();
    }

    @Override
    protected void travel() {
```

```
            first.travel();
            second.travel();
        }
}
```

PassengerByAirAndCoach 类包含 ByAir 类的对象 first 和 ByCoach 类的对象 second。于是，我们的测试代码如下：

```
public class HappyPeopleTestDrive {
    public static void main(String[] args) {
            HappyPeople passengerByAirAndCoach = new PassengerByAirAndCoach();

            System.out.println("Let's Go Home For A Grand Family Reunion...\n");
            System.out.println("Jennifer is going home:");
            passengerByAirAndCoach.celebrateSpringFestival();
    }
}
```

最后，执行结果如下：

```
Let's Go Home For A Grand Family Reunion...

Jennifer is going home:
Buying ticket...
Travelling by Air...
Travelling by Coach...
Happy Chinese New Year!
```

12.3 策略模式

我们上述封装了具体的回家方式，实际上是使用了策略模式，GoF 给出的定义是：

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

定义了一组算法，对每一种进行封装，使它们可以相互交换。此模式可以使算法独立于使用它的客户程序而变化。

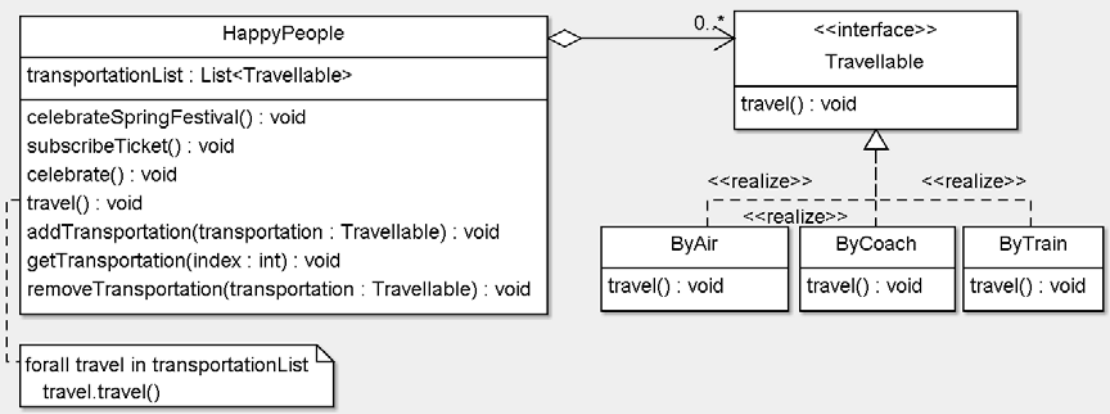
Travellable 接口是策略接口，我们的三个实现类，ByAir 类，ByCoach 类和 ByTrain 类都实现了各自不同的具体回家方式；PassengerByAirAndCoach 类和 PassengerByAir 等类都包含了不同的回家算法，它们在此模式里被称为 Context，维护指向具体策略类的对象引用。

细心的读者可能会回想起我们在 第 7 章所讲的那段测试代码？我们在测试代码中使用了策略模式，枚举类型（enum）DecoratorTest提供了抽象的算法接口，枚举值***HBTEST***和***JPATEST***是具体算法的实现，关于枚举类型的介绍，请参见 13.5 使用enum类型一节。DecoratorTestDrive是相应的Context类，我们在运行时可以交换选择Hibernate环境和Jpa环境进行测试。

12.4 还需要继承吗

上述我们尽管抽象出了 Travellable 接口，我们依然在使用继承，为了创建先坐大巴，然后再转大巴回家的这类人，我们还得创建一个新类，虽然避免了代码重复，但又引起了类的泛滥，既然策略模式可以让我们在运行时交换算法，如果我们运行时决定他们的回家方式，那我们现在还需要 HappyPeople 的子类吗？

既然这样，我们没必要使用静态的方式创建这些不同回家方式的人，我们在运行时就可以决定他们的回家行为。由于 HappyPeople 和 Travellable 之间是一对多的关系，我们为 HappyPeople 添加了一个链表来持有 Travellable 对象，在运行时动态地把回家的方式添加到这个链表就行；方法 travel()依次执行每个 Travellable 对象的 travel()方法；最后，为了管理 Travellable 对象，我们还增加 addTransportation(Travellable transportation)，getTransportation(int index)等方法。于是，UML 静态类图示如下：



有了上述结构，不管乘坐什么交通工具回家，乘坐多少次交通工具，我们都可以实现，而且没有了重复的代码。

我们看看 HappyPeople 的代码如何实现的：

```
public class HappyPeople {
    private List<Travellable> transportationList;

    public HappyPeople() {
        transportationList = new LinkedList<Travellable>();
    }

    public void celebrateSpringFestival() {
        subscribeTicket();
        travel();
        celebrate();
    }

    public void travel() {
        for (Travellable travel : transportationList) {
            travel.travel();
        }
    }
```

```
    //other methods…
}
```

其实我们这里又使用到了一条 OOP 的原则，**优先使用合成而非继承**，即：

> **Favor Composition over Inheritance.**
>
> **优先使用对象组合，而不是继承。**

OOP 的一个重要特点是继承，继承提供了一条避免代码重复的途径，尽管如此，我们还是优先考虑合成，HappyPeople 对象合成了 Travellable 的对象，这比继承来的更加灵活。

虽然在重用方面，继承比合成来的更为简单，但其有如下缺点：

- 子类和父类之间是高耦合的，这体现在三个方面：
  - 子类可以看到父类的一些非公有的细节（protected 属性和方法），破坏了封装性。
  - 子类的行为和父类的私有性质之间的协作也是直接的，父类可以直接控制子类的行为，父类私有行为的改变有时会给子类的运行带来不可预知的错误。
  - 父类公有性质的改变，直接影响了子类性质的改变。

  如果使用合成，只能依赖于它的公有性质，它们之间不会轻易改变这些公有的约定。
- 子类的行为在运行时无法改变的，这些行为都是编译时决定的，是静态的。如果使用合成，只要依赖于接口或者抽象，运行时就可以动态替换这些对象，享受到了依赖于接口/抽象的好处。

在实际的开发中，我们不一定孤立地使用它们，有时候往往结合它们，比如我们上述讲解的 装饰器模式，组合模式等都是结合使用它们很好的示例。

12.5 总结

封装变化是行为模式的一个重要主题：把独立变化的部分抽象出来，使得不变的部

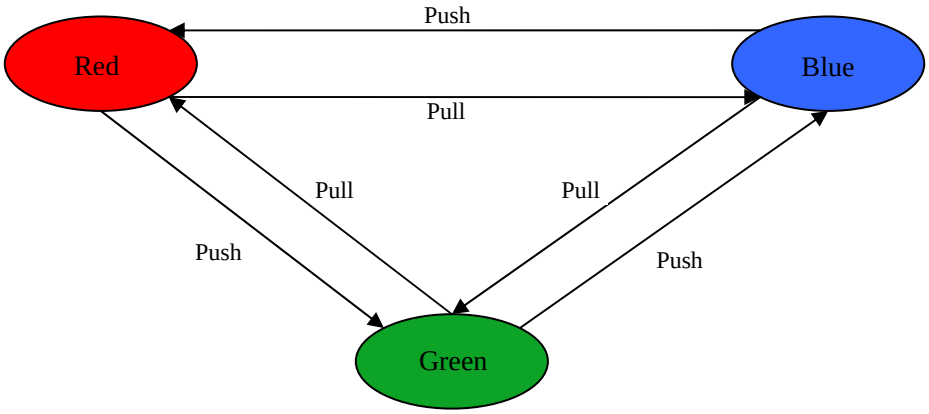
分不受变化的那部分影响，这样非常便于测试和调式。策略模式变化的那方面是策略那方面，不变的是 Context 那方面。

从上例可以看到，并不是所有的模式拿过来使用都能解决当前的问题。我们之前使用的模板方法模式，并不能解决后来面临的复杂问题，于是选择了策略模式，这并不意味着模板方法模式比策略模式好，只是在此场景使用策略模式会更好地帮助我们解决问题。**其实模式本身没有错，错就错在我们没在正确的场合合理地使用它们**。

第13章　状态（State）模式

13.1 电子颜料板

有这样一块电子颜料板，它上面有个手柄，可以拉（pull）也可以推（push），每次操作，都会改变颜料板的颜色，它们的变化关系如下图所示：



13.2 switch-case实现

这又是一个关于变化的例子，在讲解如何封装变化之前，我们首先给出使用switch-case语句解决这类问题的代码。关于push操作的代码片段，大致如下：

```
public void push() {
    switch (state) {
        case RED:
            state = Color.GREEN;
            break;
        case BLUE:
            state = Color.RED;
            break;
        case GREEN:
            state = Color.BLUE;
            break;
        default:
```

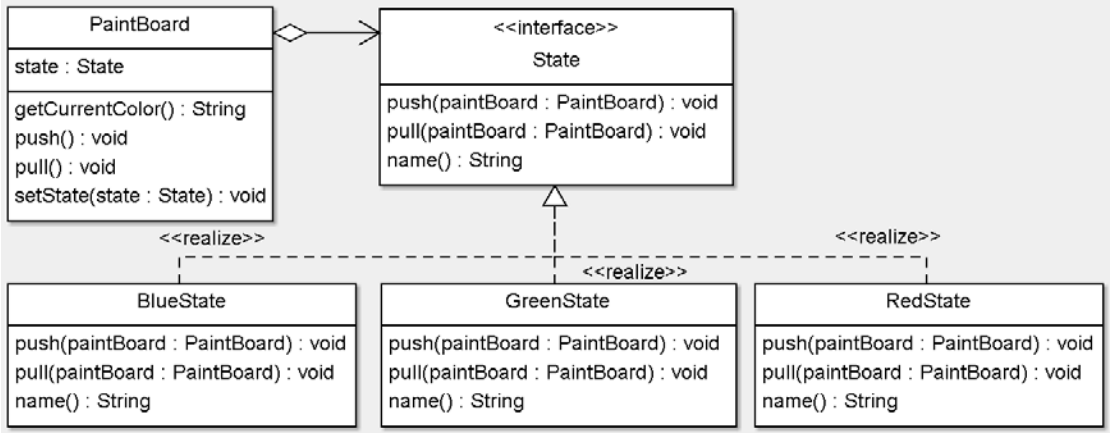
```
                throw new RuntimeException("Invalid state when pushing");
        }
}
```

这段代码看似并没有什么不妥。我们知道，需求总是不停变化的，如果又有新的状态需要加入时，你就得在 push()方法和 pull()方法里添加新的 case 分支语句，破坏了 OCP 原则。另外，如果逻辑比较复杂，一个 case 分支语句的代码甚至会超过几百行，代码会变得很难阅读和维护。

13.3 如何封装变化

从上面的状态变化图我们知道，在不同的颜色状态下 push 操作和 pull 操作是不一样的。譬如，当前状态是红色，push 操作会使状态变成绿色；如果状态是绿色，则会变成蓝色。不同的状态关联不同的行为，这就是我们该场景的变化，为此，我们就抽象出一个状态的接口，它定义了与状态相关的方法，于是，具体的状态就可以封装与之相关的行为了。

现在就让我们来构造它们：State 是状态接口，含有 push(PaintBoard paintBoard)和 pull(PaintBoard paintBoard)两个方法；PaintBoard 类包含 push()和 pull()两个方法，它把这两个操作委托给当前状态完成。那么，UML 静态类图如下所示：



我们看看 PaintBoard 类的代码，大致如下：

```
public class PaintBoard {

    private State state = new RedState();

    //other methods and fields…
```

```
    public void setState(State state) {
        this.state = state;
    }

    public void push(){
        state.push(this);
    }

    public void pull(){
        state.pull(this);
    }
}
```

每个具体状态的 push 操作和 pull 操作都会回调 PaintBoard 的方法 setState(State state) 来设置 PaintBoard 的新状态。State 接口和 RedState 实现类的代码如下所示：

```
public interface State {
    void push(PaintBoard paintBoard);

    void pull(PaintBoard paintBoard);

    String name();
}

public class RedState implements State{
    @Override
    public void push(PaintBoard paintBoard) {
        paintBoard.setState(new GreenState());
    }

    @Override
    public void pull(PaintBoard paintBoard) {
```

```
            paintBoard.setState(new BlueState());
        }

        @Override
        public String name() {
            return "RED";
        }
}
```

其他具体状态类与RedState类相似，具体代码我们就不再赘述，请参见 示例代码。

下一步我们创建测试代码，如下所示：

```
public class PaintBoardTestDrive {
    public static void main(String[] args) {
        PaintBoardTestDrive test = new PaintBoardTestDrive();
        PaintBoard paintBoard = new PaintBoard();

        System.out.println("Push Test:");
        System.out.println("Paint board current color:" + paintBoard.getCurrentColor());
        test.pushTest(paintBoard);

        System.out.println("\nPull Test:");
        System.out.println("Paint board current color:" + paintBoard.getCurrentColor());
        test.pullTest(paintBoard);
    }

    public void pushTest(PaintBoard paintBoard) {
        for (int i = 0; i < 3; i++) {
            System.out.printf("%s ---push---> ", paintBoard.getCurrentColor());
            paintBoard.push();
```

```
                System.out.println(paintBoard.getCurrentColor());
            }
        }
        public void pullTest(PaintBoard paintBoard) {
            for (int i = 0; i < 3; i++) {
                System.out.printf("%s ---pull---> ", paintBoard.getCurrentColor());
                paintBoard.pull();
                System.out.println(paintBoard.getCurrentColor());
            }
        }
}
```

最后，我们看看测试效果，如下所示：

```
Push Test:
Paint board current color:RED
RED ---push---> GREEN
GREEN ---push---> BLUE
BLUE ---push---> RED

Pull Test:
Paint board current color:RED
RED ---pull---> BLUE
BLUE ---pull---> GREEN
GREEN ---pull---> RED
```

当前状态代理了 PaintBoard 的 push 和 pull 操作，这样，PaintBoard 类的代码里再也找不到“丑陋”的条件分支语句，代码也就更加容易阅读。如果需要添加更多的状态，只要添加新的 State 接口实现即可，符合 OCP 原则。

13.4 状态模式

我们为变化的那部分抽象了 State 接口，把不同的行为封装在相应的状态类中，PaintBoard 在执行与状态相关的行为时，把请求转发给当前状态类去执行，这正是

状态模式。GoF 为状态模式给出的定义是：

> **Allow an object to alter its behavior when its internal state changes.The object will appear to change its class.**
>
> **允许一个对象在其内部状态发生变化时改变自己的行为，该对象看起来好像修改了它的的类型。**

此例中，PaintBoard 类是担任了 Context 角色，客户对象直接操作此类对象，State 是状态接口，BlueState，GreenState 和 RedState 这些都是具体状态类。

13.5 使用enum类型

我们把不同状态下的行为封装在不同的 State 实现类里，每一种状态就是一个独立的类，其实我们还可以使用枚举类型（enum），每一种状态是一个枚举值。

枚举类型是从 JDK1.5 版本开始引入的一种新类型，它包含了几种特定的值供程序选择。Java 语言的枚举类型和 C 语言的枚举类型有点不同，C 语言的枚举值其实是一个整型常量，而 Java 语言的枚举值是一个对象，所以我们可以在其中封装方法和属性。

我们枚举类型仍然实现了 State 接口，每种枚举值封装了 push 和 pull 操作。枚举类型 Color 的实现代码大致如下：

```
public enum Color implements State {
    //enum values...

    @Override
    public void push(PaintBoard paintBoard) {
        throw new UnsupportedOperationException("Invalid push!");
    }

    @Override
```

```
    public void pull(PaintBoard paintBoard) {
        throw new UnsupportedOperationException("Invalid pull!");
    }
}
```

这里每种枚举值都必须实现 State 接口的这些方法，否则会抛出 UnsupportedOperationException 异常。

我们看看 RED 枚举值是如何实现的，代码如下所示：

```
public enum Color implements State {
    RED {
        @Override
        public void push(PaintBoard paintBoard) {
            paintBoard.setState(GREEN);
        }

        @Override
        public void pull(PaintBoard paintBoard) {
            paintBoard.setState(BLUE);
        }
    },

    //other enum values…
}
```

RED枚举值覆写了push(PaintBoard paintBoard)方法和pull(PaintBoard paintBoard)方法，和之前具体类的实现没太大区别，其他枚举类型的实现与RED类似，请参见 示例代码。

其实我们甚至还可以省略 State 接口，在 enum 类型 Color 里定义这两个方法，push(PaintBoard paintBoard)方法和 pull(PaintBoard paintBoard)方法。使用接口的好处是，如果你需要增加了新的状态时，就没必要为增加新的枚举值而修改此枚举类

型，你可以创建新的具体实现类，不影响既有的代码。

如果你用的是Jdk 1.4 及以下版本，读者可以使用《Effective Java》一书中提出的**Typesafe Enum模式** [16]来实现。其实Java的枚举类型沿袭了《Effective Java》一书中提出的Typesafe Enum模式，编译器在编译enum类型时，会生成一个真正的类代替枚举类型。其实Java没有真正的枚举类型，这个新生成的类继承于java.lang.Enum类，每个枚举值的类型都是这个新类的子类。为了方便理解，这里给出此例反编译出来的代码以供参考，如下所示：

```
public class Color extends Enum implements State {
    public static final Color RED;
    public static final Color GREEN;
    public static final Color BLUE;
    private static final Color $VALUES[];

    private Color(String s, int i) {
        super(s, i);
    }

    public static Color[] values(){
        return (Color[])$VALUES.clone();
    }

    public static Color valueOf(String name){
        return (Color)Enum.valueOf(pattern.part4.chapter13.tiger.Color, name);
    }

    public void push(PaintBoard paintBoard){
        throw new UnsupportedOperationException("Invalid push!");
    }
```

[16] 参见《Effective Java》一书的第五章：Chapter 5. Substitutes for C Constructs, Item 21: Replace enum constructs with classes。

```
    public void pull(PaintBoard paintBoard){
        throw new UnsupportedOperationException("Invalid pull!");
    }

    //push(), pull() and other methods …
    static {
        RED = new Color("RED", 0) {
            public void push(PaintBoard paintBoard) {
                paintBoard.setState(GREEN);
            }
            public void pull(PaintBoard paintBoard) {
                paintBoard.setState(BLUE);
            }
        };

        //other states…
        $VALUES = (new Color[]{RED, GREEN, BLUE});
    }
}
```

我们看到，RED 是这个新生成的 Color 类的一个内部类对象，RED、GREEN 和 BLUE 都是 Color 类的成员常量，有兴趣的读者可以自己动手反编译生成的相关 class 二进制文件来做更进一步的了解。

我们这里给出了应用 Java 枚举类型实现状态模式的方法，在一些操作不是特别复杂的应用中，可以避免类的膨胀，但如果需要封装一些复杂的操作时，不同状态的方法挤在一起，代码也不容易阅读。为了能够扩展新的状态，笔者建议建议大家仍然为枚举类型创建 State 接口，否则，势必只能修改该枚举类型以添加新的状态，不符合 OCP 原则。

13.6 与策略（Strategy）模式的比较

- 都把行为封装到具体的类中。策略模式把不同的算法封装到不同的子类，状态模式把不同状态下的行为封装到与之对应的状态子类中。
- 它们都是使用了合成，代理了上下文的相关操作。
- 一个状态对象封装了与状态相关的行为，并控制状态之间的转换。策略对象封装了算法，算法之间不存在切换。
- 状态模式隐藏了状态接口和子类：策略模式中，客户对象可以在运行时选择不同的算法，策略接口和具体实现对它是可见的；但是状态模式中，客户对象只能操作 Context 对象，状态接口和子类对它是不可见的。

第14章　观察者（Observer）模式

14.1 股票价格变了多少

小 D 喜欢炒股，经过严格的筛选，终于看中了一只股票。为了及时地了解这只股票价格的信息，他每天都要上网查询好几次。刚开始，由于新鲜感的驱使，小 D 并没感到太多麻烦，但是几个星期后，他逐渐觉得这样做十分不便：有时候，几天过去了，所关注的股票几乎没有波动；有时候，这只股票的价格会在一天之内上下大幅波动好几次，小 D 为此错失了好多机会。如果你是小 D，你该怎么办？

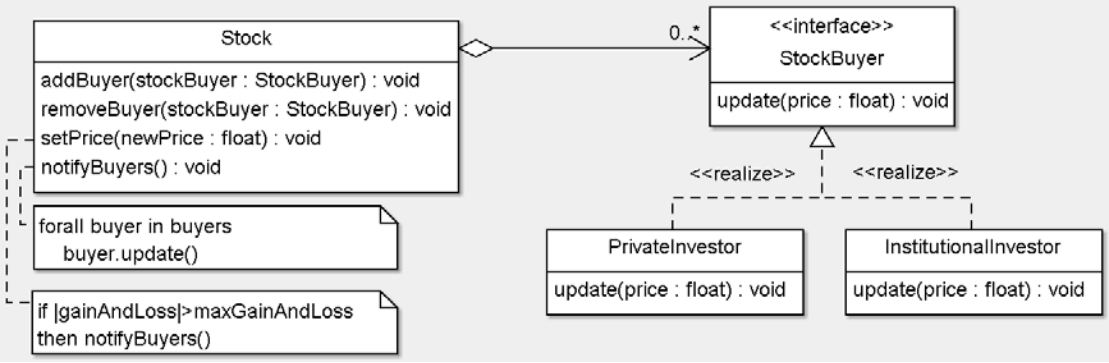
我不知道你是否有更好的方法，小 D 是这么想的，如果这只股票的价格变化波动到一定程度时能通知自己，那他就不必要不停地去上网查询了。于是，他决定设计一个工具能够自动地把股票信息发送给自己。

14.2 观察者模式

小 D 的这种想法涉及了两个角色：一个是观察者，即小 D，另外一个是被观察的对象，即股票。当股票的价格发生一定改变时，股票去通知观察者小 D，这样小 D 就会根据这些变化并作出相应的反应。其实，不光像小 D 这样的散户希望收到通知，还有一些大型机构投资者也希望收到通知，这些机构也是观察者。

14.2.1 如何实现

为了发送通知，主题应该知道观察者是谁，我们应该提供注册和删除观察者的接口方法；当主题发生变化时或者变化到一定程度时，主题就通知这些观察者。我们首先给出 UML 实现类图，如下所示：



Stock 类可以通过 ***addBuyers(StockBuyer buyer)***和 ***removeBuyers(StockBuyer buyer)*** 来注册和移除观察者，股票投资者和投资机构；方法 ***setPrice(float newPrice)***设置股

票的价格，当股票价格的波动超过一定范围（***maxGainAndLoss***）时，便会触发***notifyBuyers()***方法通知所有股票投资者。

观察者接口 StockBuyer 有一个 ***update(float price)***方法，其实现类定制股票价格发生变化时各自具体的行为，接口代码如下：

```
public interface StockBuyer {
    void update(float price);
}
```

接下来，我们来看看 Stock 类的代码实现，大致如下所示：

```
import static java.lang.Math.abs;
//other imports…

public class Stock {
    //other fields and methods

    public void notifyBuyer() {
        for (StockBuyer buyer : buyers) {
            buyer.update(price);
        }
    }

    public void setPrice(float newPrice) {
        if (newPrice < 0) {
            throw new IllegalArgumentException("Price can not be negative!");
        }

        //update price and calculate change…
        float oldPrice = price;
        price = newPrice;
```

```
            float gainAndLoss = (newPrice - oldPrice) / oldPrice;//calculate change
            System.out.printf("Previous price: %g. Current price: %g. Loss/Gain: %g%%. \n",
oldPrice, newPrice, gainAndLoss);


            //if change beyond maxGainAndLoss, notify stock buyers
            if (abs(gainAndLoss) > maxGainAndLoss) {
                notifyBuyer();
            }
        }
}
```

当股票价格变化超过一定范围时，方法 ***notifyBuyers()***方法调用每一个观察者的 ***update(float price)***方法。

InstitutionalInvestor（机构投资者）实现了 StockBuyer 接口，当股价高于 **maxPrice** 时，机构投资者会卖出 100000 股，当股价低于 **minPrice** 时，会买进 20000 股。模拟代码大致如下：

```
public class InstitutionalInvestor implements StockBuyer {
    private String name;
    private float maxPrice;
    private float minPrice;

    public InstitutionalInvestor(String name, float maxPrice, float minPrice, Stock stock) {
        this.name = name;
        this.maxPrice = maxPrice;
        this.minPrice = minPrice;
        stock.addBuyers(this);
    }

    @Override
    public void update(float price) {
```

```
            if (price > maxPrice) {
                System.out.printf("%s is selling 100000 stocks...\n", name);
            }

            if (price < minPrice) {
                System.out.printf("%s is buying 20000 shares...\n", name);
            }
        }
}
```

同样，PrivateInvestor（散户）也实现 StockBuyer 接口，当股价高于 **maxPrice** 时，散户将会买进 500 股，当股价低于 **minPrice** 时，会卖出 1000 股。模拟代码大致如下：

```
public class PrivateInvestor implements StockBuyer {
    private String name;
    private float maxPrice;
    private float minPrice;

    public PrivateInvestor(String name, float maxPrice, float minPrice, Stock stock) {
        this.name = name;
        this.maxPrice = maxPrice;
        this.minPrice = minPrice;
        stock.addBuyers(this);
    }

    @Override
    public void update(float price) {
        if (price > maxPrice) {
            System.out.printf("%s is buying 500 shares...\n", name);
        }
```

```
            if (price < minPrice) {
                System.out.printf("%s is selling 1000 stocks...\n", name);
            }
        }
}
```

我们来编写测试代码，如下所示：

```
public class TestDrive {
    public static void main(String[] args) {
        Stock stock = new Stock(19f);
        InstitutionalInvestor institutionalInvestor = new InstitutionalInvestor("Company E", 20f, 18.5f, stock);
        PrivateInvestor privateInvestor = new PrivateInvestor("Xiao D", 20f, 18.9f, stock);

        stock.setPrice(19.0224f);
        System.out.println();

        stock.setPrice(20.923f);
        System.out.println();

        stock.setPrice(18.8938f);
        System.out.println();

        stock.setPrice(19.9938f);
    }
}
```

机构投资者 Company E 的 **maxPrice** 是 **20f**，**minPrice** 是 **18.5f**；散户小 D 的 **maxPrice** 是 **20f**，**minPrice** 是 **18.9f**。

最后，我们看看执行结果如何，如下所示：

```
Previous price: 19.0000. Current price: 19.0224. Loss/Gain: 0.117894%.
```

```
Previous price: 19.0224. Current price: 20.9230. Loss/Gain: 9.99138%.
Company E is selling 100000 stocks...
Xiao D is buying 500 shares...

Previous price: 20.9230. Current price: 18.8938. Loss/Gain: -9.69842%.
Xiao D is selling 1000 stocks...

Previous price: 18.8938. Current price: 19.9938. Loss/Gain: 5.82201%.
```

14.2.2 观察者模式

在这个场景中，股票和投资者之间的关系是 1:N 的关系，当价格发生一定变化时，股票以广播的方式通知投资者，这正是观察者模式，GoF 给出的定义如下：

> **Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.**
>
> **定义了对象之间的一种一对多的依赖关系，这样，当一个对象的状态发生变化时，所有依赖于对象都被通知并自动更新。**

我们把被观察的对象也叫做主题（Subject），这里的主题即是 Stock 类，StockBuyer 便是观察者接口。当有多个主题时，你可以抽象出一个 Subject 接口，这样，就可以独立地改变主题和观察者这两方面而不会相互影响。

我们上述实现的模式是 push 方式的观察者模式，即股票把观察者需要的所有信息都推给了它们，这里是指把股票价格推给了目标。我们还有一种方式，我们称之为 pull 方式，即股票在通知观察者时并不传递股票信息，而由观察者显式地查询股票信息。使用前者时，如果观察者需要获得更多的信息，可能导致观察者接口不能重用；使用后者时，观察者可能不容易知道哪些信息发生了改变。

14.2.3 Java标准库的观察者模式

观察者模式的应用极其广泛，Java 标准库中有很多地方用到了观察者模式，特别是在 Java GUI 编程中。Java Uitl 包中还提供了类 java.util.Observable（主题）和

java.util.Observer（观察者）接口以方便 Java 开发人员在编程过程中快速地使用观察者模式，我们先来介绍这两个类型。

- java.util.Observer 接口

  Observer 接口的代码如下所示：

```
package java.util;

public interface Observer {

    void update(Observable o, Object arg);

}
```

  和我们的 StockBuyer 接口一样都含有一个方法，用于主题发生变化时去告知该观察者。执行这个方法 ***update(Observable o, Object arg)***时，能得到主题 Observable 对象，它的第二个参数用于传递主题和观察者之间的变化数据，例如，上例中的股票价格，如果不需要传递数据，则可传入 null 值。这种方式把 push 方式和 pull 方式结合了起来，读者在使用过程中可以灵活实现。

- java.util.Observable 类

  - 方法 addObserver(Observer o)/ deleteObserver(Observer o)用于注册/删除观察者。

  - 方法 setChanged()和 clearChanged()设置状态变化情况：在调用 notifyObservers()/notifyObservers(Object arg)方法之前，一定要都要调用 setChanged()方法设置状态发生了改变，否则不会触发通知；clearChanged()方法清除状态变化，这两个方法都是 protected 的，代码如下所示：

```
protected synchronized void setChanged() {
changed = true;
}

protected synchronized void clearChanged() {
changed = false;
}
```

  - 方法 notifyObservers()和 notifyObservers(Object arg)用于通知观察者，相当于我们前述 Stock 类的 notifyBuyer()方法，这两个方法的区别是，***notifyObservers()***方法相当于 ***notifyObservers(null)***。让我们看看

notifyObservers(Object arg)方法是如何实现的，代码如下：

```
public void notifyObservers(Object arg) {
    Object[] arrLocal;

    synchronized (this) {
        if (!changed)
            return;
        arrLocal = obs.toArray();
        clearChanged();
    }

    for (int i = arrLocal.length-1; i>=0; i--)
        ((Observer)arrLocal[i]).update(this, arg);
}
```

此方法首先判断 Subject 状态是否发生了改变，如果没有改变则返回，否则准备通知观察者，并调用 clearChanged()方法清除这次状态变化。最后依次调用每个观察者的 updateupdate(Observable o, Object arg)的方法。可以看出，如果调用 notifyObservers(Object arg)方法，则是使用了 pull 和 push 结合的方式，而调用 notifyObservers()方法，则是选择了 pull 的方式。最后，还需注意的是，它采用了倒序的方式通知所有的观察者，即这句 ***for (int i = arrLocal.length-1; i>=0; i--)***。

14.2.3.1 如何使用

有了上述学习之后，我们来使用它们重写代码，如下所示：

```
import java.util.Observable;
import static java.lang.Math.abs;

public class Stock extends Observable {
    private static final float maxGainAndLoss=0.05f;//5%
    private float price;
```

```
    public Stock(float price) {
        super();
        this.price = price;
    }

    public void setPrice(float newPrice) {
        if (newPrice < 0) {
            throw new IllegalArgumentException("Price can not be negative!");
        }

        //update price and calculate change...
        float oldPrice = price;
        price = newPrice;
        float gainAndLoss = (newPrice - oldPrice) / oldPrice;//calculate change
        System.out.printf("Previous price: %g. Current price: %g. Loss/Gain: %g%%.\n",
oldPrice, newPrice, gainAndLoss*100);

        //if change beyond maxGainAndLoss, notify stock buyers
        if (abs(gainAndLoss) > maxGainAndLoss) {
            setChanged();
            notifyObservers(price);
        }
    }
}
```

注意：

在调用 ***notifyObservers(Object arg)***方法之前，必须调用 ***setChanged()***方法，否则据上分析，它会忽略此次改变。

接着，我们让 InstitutionalInvestor 类实现 java.util.Observer 接口，代码如下所示：

```
import java.util.Observable;
```

```
import java.util.Observer;


public class InstitutionalInvestor implements Observer {
	private String name;
	private float maxPrice;
	private float minPrice;


	public InstitutionalInvestor(String name, float maxPrice, float minPrice, Stock stock) {
		this.name = name;
		this.maxPrice = maxPrice;
		this.minPrice = minPrice;
		stock.addObserver(this);
	}


	@Override
	public void update(Observable o, Object arg) {
		float price = (Float) arg;
		if (price > maxPrice) {
			System.out.printf("%s is selling 100000 stocks.\n", name);
		}


		if (price < minPrice) {
			System.out.printf("%s is buying 20000 shares.\n", name);
		}
	}
}
```

构造函数里，我们使用stock.addObserver(this)替换stock.addBuyers(this)，来为股票添加此观察者，update (float price)的方法签名也换成update(Observable o, Object arg)。PrivateInvestor类与此类似，在这里不再赘述，具体请参见 示例代码。

最后，让我们验证一下执行效果，测试代码和以前一样，执行结果如下所示：

```
Previous price: 19.0000. Current price: 19.0224. Loss/Gain: 0.117894%.


Previous price: 19.0224. Current price: 20.9230. Loss/Gain: 9.99138%.
Xiao D is buying 500 shares...
Company E is selling 100000 stocks...


Previous price: 20.9230. Current price: 18.8938. Loss/Gain: -9.69842%.
Xiao D is selling 1000 stocks...


Previous price: 18.8938. Current price: 19.9938. Loss/Gain: 5.82201%.
```

注意蓝色的部分，由于采用倒序的方法通知观察者，所以执行结果中，小 D 先于机构投资者 Company E 得知股票变化。

14.2.3.2 还应注意

- java.util.Observable 是具体类，而不是接口，对于 Java 这样的单继承语言，不能让其同时继承 Observable 类和其他父类。这里为大家提供一种思路来解决此问题：你可以使该新类依赖与的 java.util.Observable 子类（必须是子类，随后你将看到），当对象的状态发生变化时调用 java.util.Observable 子类的通知方法，来达到通知的效果；必须注意的是，你一定要在调用通知之前修改状态变化的标志，即调用 setChanged()方法，但是此方法在 java.util.Observable 类中是 protected 的，为了能够使用，必须在该 java.util.Observable 子类中覆写此方法，使其变为 public 的，另外 clearChanged()方法在必要时也需要覆写。

- 正如我们前面所提到的，观察者的通知顺序和我们之前的实现不一样，一般地我们希望观察者的通知顺序是无关的，如果你的通知必须要按照特定的顺序执行，可以专门维护一个 ChangeManager 类来处理这种问题，而且，还可以在 ChangeManager 里还可以实现一些复杂的通知策略。

14.3 总结

如果你还记得 好莱坞原则（不要找我们，我们会找你）的话，你一定发现了，观察者模式又是一个它的应用。观察者模式降低了主题和观察者的耦合度，使主题依赖

于观察者的接口，而不去关心哪个观察者会关注该主题，在图形化编程和分布式事件处理系统中，此模式被广泛使用，也是在JDK中使用最多的模式之一。

第五篇 终点还是起点

世界上，任何事物都不是完美的，OOP也一样，它不能解决所有的编程问题，OOP在纵向分解问题方面表现十分出色，但对于水平问题，有点无能为力，AOP能够弥补这方面的不足，我们在 第 15 章将介绍时下最流行的AOP技术。

了解和使用模式对于开发来说，还远远不够，软件的核心是模型，开发大型的复杂业务的应用，需要精炼的模型，如果模型不够精炼，也不能享受到设计模式带来的好处，在 第 16 章我们将讲解简单介绍如何提炼模型解决复杂的领域问题。

在本篇末尾，第 17 章，我们将回顾本书的内容。

第15章　面向切面的编程（AOP）

15.1 简介

OOP 目前已经是软件开发的主流了，我们非常方便地把领域问题进行分解，封装成对象，然而和过程式编程（Procedural Programming，PP）或者函数式编程语言（Functional Programming，FP）等传统编程技术一样，还有一类问题它们都无法优雅地解决，产生这些问题的原因正是由于它们的共同优点，把问题分解为小的单元——对象，方法，结构体等，引起的，这章我们首先给出一个例子引入这类问题的讨论。

15.2 记录时间

我们依然采用我们大家熟悉的回家过年的例子来描述该问题，以第 2 章的示例代码为基础，假如我们现在需要记录travel()方法所耗费的时间，我们可以使用模板方法模式来解决，重构代码大致如下所示：

```
public abstract class HappyPeopleAop {
    private final Stopwatch stopwatch= new Stopwatch();

    protected final void travel() {
        stopwatch.reset();
        onTravel();
        stopwatch.info("Method travel() used: ");
    }

    protected abstract void onTravel();

    //other methods …
}
```

代码注解：

- 我们使用 Stopwatch 类记录 travel()的运行时间。Stopwatch（秒表）类正如其名，它从调用 reset()方法时开始计时，当调用 info()方法时会打印开始计时到当前操作这段执行时间的长度。

- 为了记录 travel()方法的执行时间，我们把这个方法分解为三个部分，分离出的抽象方法 onTravel()方法代替之前 travel()方法，执行之前该方法的全部逻辑；在执行 onTravel()方法之前设定计时时刻，在执行该方法之后打印出这段时间的长度。

那么以 PassengerByAirAop 类为例，onTravel()方法代码大致如下：

```
@Override
protected void onTravel() {
    //Travel by Air…
    System.out.println("Travelling by Air...");
    try {//try to sleep for a while...
        TimeUnit.MILLISECONDS.sleep(200l);
    } catch (InterruptedException e) {
        //do nothing
    }
}
```

为了方便我们计算时间，我们让线程休息 200 毫秒，即这句 ***TimeUnit.MILLISECONDS.sleep(200l)***，其他子类与之类似，我们在这里将不再赘述。

我们撰写的测试代码如下所示：

```
public static void main(String[] args) {
    HappyPeopleAop passengerByAir = new PassengerByAirAop();
    HappyPeopleAop passengerByCoach = new PassengerByCoachAop();
    HappyPeopleAop passengerByTrain = new PassengerByTrainAop();

    System.out.println("Let's Go Home For A Grand Family Reunion...\n");

    System.out.println("Tom is going home:");
    passengerByAir.celebrateSpringFestival();

    System.out.println("\nRoss is going home:");
```

```
        passengerByCoach.celebrateSpringFestival();


        System.out.println("\nCatherine is going home:");
        passengerByTrain.celebrateSpringFestival();
}
```

某一次的测试执行结果如下所示：

```
Let's Go Home For A Grand Family Reunion...


Tom is going home:
Buying a ticket...
Travelling by Air...
Method travel() used: 188ms
Happy Chinese New Year!


Ross is going home:
Buying a ticket...
Travelling by Coach...
Method travel() used: 593ms
Happy Chinese New Year!


Catherine is going home:
Buying a ticket...
Travelling by Train...
Method travel() used: 407ms
Happy Chinese New Year!
```

模板方法模式解决了我们对 onTravel()方法运行时间的统计，如果我们想更进一步对 subscribeTicket()和 celebrate()所耗的时间进行统计，那该怎么办了呢？我们只能为每个方法添加了如下斜体加粗部分的代码，如下所示：

```
stopwatch.reset();
XXX();    //method
```

stopwatch.info(…); //print how long method XXX() takes

冗余代码的“臭味”再度摆在我们面前了，到目前为止，还没有一个已经介绍的模式能够解决这类重复的代码？使用模板方法模式对少数几个方法进行这样的处理，似乎问题不会太严重，如果对几十个，甚至上千个方法进行同样处理，重复代码的“臭味”将会弥漫在整个程序中。有没有更加优雅的方案把这些零散的代码模块化在一处呢？答案是有的，这即是我们下节要介绍的 AOP 技术。

15.3 AOP（Aspect-Oriented Programming）

我们在软件设计时，会把系统分割为小的单独的单元来处理，我们把这种设计方法称之为 SoC（即 Separation of Concerns），一个关注点（Concern）就是一个问题。编程过程中，我们把这些关注点模块化在编程语言所提供的单元里，例如，在传统的编程语言中（这里指过程式编程/函数式语言），会把系统分割成数据结构和方法，在 OOP 语言中，我们通常把它们分解为类。

SoC一方面分解了问题：我们把系统表示为一堆软件实体和实体之间的交互，把复杂的问题按照功能分解成了小粒度的简单问题，主要是纵向分解了问题，然后各个击破。但另外一方面，正是由于分解产生了很多小单元，自然而然地出现了一些横跨这些单元的相同逻辑，即在水平方向上出现了相同的逻辑。在上述例子中，记录travel()的执行时间和记录celebrate()的执行时间，都是相同的问题。使用传统的软件编程和OOP，我们都不容易解决这种重复问题，例如使用 模板方法模式仍然导致在这两个方法分别添加相同逻辑的代码。为了把这种水平方向上的逻辑模块化，随之便产生了AOP编程。

AOP 可以在水平方向上补充 OOP 的这一缺点，它可以把这些离散的，跨模块的，混入其他关注点的逻辑放在一个独立的模块里，即从横向模块化了问题。对于上述计时问题，我们可以把记录这些方法执行时间的逻辑模块化在同一个单元里，这个新的模块化单元就称之为**切面**（Aspect）。

我们编程过程中有很多水平问题，例如事务管理，异常处理，日志记录，安全管理，性能优化，透明的持久化等等，这些问题也是 J2EE 关注的重要问题，由于 AOP 在这方面发挥出色, AOP 也成了时下很多 J2EE 流行框架的重要部分，我们就从一些基础概念入手了解 AOP。

15.3.1 一些重要概念

下面我们介绍一些 AOP 的重要概念和术语：

- 切面（Aspect）：横切关注点（Concerns）的模块化。在传统的软件编程方法里，我们把关注点模块化为函数，过程，类等里，而在 AOP 编程里，我们把关注点模块化为切面。
- 连接点（Joinpoint）：程序执行过程中的可被识别某个点。连接点可以是某个方法的调用，也可以是某个赋值操作，是 AOP 里最基础的概念。例如上例中 travel()方法调用 onTravel()方法的位置就是一个连接点。
- 增强（Advice，有的地方也翻译为通知）：连接点处执行的逻辑。分为前增强，后增强，环绕增强，异常增强和最终增强。
  - 前增强（Before advice）：在连接点之前执行的增强。
  - 后增强（After returning advice）：在连接点正常执行完成后执行的增强。
  - 异常增强（After throwing advice）：捕捉连接点抛出特定异常后执行的增强。
  - 最终增强（After/finally advice）：调用连接点之后始终执行的增强，不管此连接点是异常退出还是正常退出。
  - 环绕增强（Around Advice）：包围一个连接点的增强。它控制连接点的执行，是最强大的一种增强，它在连接点之前和之后都完成一些操作，负责调用连接点，甚至可以不执行连接点，直接抛出异常。
- 切入点（Pointcut）：选择一组特定连接点（Joinpoint）的逻辑。根据切入点我们判断哪些连接点（Joinpoint）需要增强，哪些不需要。
- 织入（Weaving）：把这些水平的切面连接到执行过程以组合成完整流程的行为称之为织入。织入可以发生在编辑期，编译期，类加载期和运行期。

15.3.2 OOP实现横切

我们知道 OOP 在解决横切的问题时，有很大局限性，但并不表示不能使用。我们可以使用它解决一些简单问题或者特定的问题，在某时候是非常有价值的，因为引入新的软件和技术支持 AOP 总是有风险的，没有必要因为一些简单问题而引入太大的风险。下面列举了一些常用的使用面向对象编程解决这类问题的方法：

- 模板方法（Template Method）模式：上例中我们使用了模板方法模式计算 onTravel()方法的执行时间，我们发现，它实现了环绕增强。对于极少数方法进

行横切是可行的，因为它只能完成对所有子类的onTravel()方法执行时间的计算，如果要针对其他方法也进行计算，需要加入冗余的代码。

- 装饰器（Decorator）模式：其实和模板方法模式一样，解决得也不够彻底，例如我们在讲解装饰器模式时所举的记录log的例子，虽然对update操作记录了日志，然而同样如果对其他方法，如delete()方法也要记录日志，我们仍然要加入重复的代码，而且它只能对具有相同接口的对象织入增强。
- 代理（Proxy）模式：和装饰器模式在这方面没有区别。
- 其他模式：如 Observer模式等，都只能完成了对部分连接点的增强，水平跨度不大，我们在这里就不再赘述。
- 拦截器框架：这个功能比较强大，但是作为一个框架，实现起来并不太容易，具体请参见下节介绍。

OOP 可以解决一些 AOP 问题，但是使用它们终究不够优雅和全面，下面我们来介绍一些常见 AOP 实现技术。

15.3.3 AOP实现技术

目前 Java 的 AOP 实现技术主要有以下几种，了解这些技术对读者使用和深入了解这些框架是非常有帮助的，笔者在这里将为大家一一介绍。

15.3.3.1 J2SE动态代理

我们在 代理模式这章为大家介绍了 J2SE动态代理技术，它可以动态代理所有指定的接口方法，这为AOP的实现提供了一个契机：动态代理把所有代理接口的方法转发给InvocationHandler的invoke(…)方法统一处理，我们在运行时就可以很轻松地根据切入点为这些方法织入增强，由于接口一致，客户对象在使用时并不会察觉到不同之处。

我们看看如何使用动态代理解决这个计时问题。由于动态代理只能代理接口，所以我们还需要提供一个 IHappyPeople 接口，如下所示：

```
public interface IHappyPeople {
    void celebrateSpringFestival();

    void subscribeTicket();
```

```
    void travel();

    void celebrate();
}
```

HappyPeopleHandler 类实现了 InvocationHandler 接口，在 invoke(Object proxy, Method method, Object[] args)方法里，我们实现此环绕增强的逻辑，代码大致如下所示：

```
public class HappyPeopleHandler implements InvocationHandler {
    private HappyPeople delegate;
    private final Stopwatch stopwatch = new Stopwatch();
    private static Set<String> methodNames;

    static {
        methodNames = new HashSet<String>();
        methodNames.add("subscribeTicket");
        methodNames.add("travel");
        methodNames.add("celebrate");
        methodNames = Collections.<String>unmodifiableSet(methodNames);
    }

    public static IHappyPeople newProxy(HappyPeople delegate) {
        IHappyPeople o = (IHappyPeople) newProxyInstance(
                delegate.getClass().getClassLoader(),
                delegate.getClass().getSuperclass().getInterfaces(),
                new HappyPeopleHandler(delegate));

        return o;
    }
```

```
public HappyPeopleHandler(HappyPeople delegate) {
        this.delegate = delegate;
}

//logic of pointcut, to check which methods should be weaved
private boolean isMethodQualified(Method method) {
        if (method == null)return false;
        return methodNames.contains(method.getName());
}

@Override
public Object invoke(Object proxy, Method method, Object[] args)
                throws Throwable {
        Object result = null;
        try {
                boolean intercept = isMethodQualified(method);

                if (intercept) {
                        stopwatch.reset();
                }

                result = method.invoke(delegate, args);

                if (intercept) {
                        stopwatch.info("Method "+method.getName()+"() used");
                }
        } catch (Exception e) { //handle exceptions...
                e.printStackTrace();
        }
        return result;
```

```
    }
}
```

其中 isMethodQualified(Method method)方法判断该代理方法是否需要增强，它封装了切入点的逻辑。在调用代理方法之前，如果满足切入点条件，则织入开始计时的逻辑，即 ***stopwatch.reset()***；同样，执行完代理方法之后，如果满足条件则织入这句，***stopwatch.info("Method "+method.getName()+"() used")***打印时间。这样，一个简单的 AOP 环绕增强的例子完成了。对于切入点逻辑，可以使用更为通用的方式实现，譬如可以根据模式字符串的来判断是否需要增强，这样，上例中切入点的逻辑可以写为类似于这样的模式匹配字符串“**.subscribeTicket()||**.travel()||**.celebrate()”。

接下来我们写出测试代码，如下所示：

```
import static pattern.chapter5.dynamicproxy.HappyPeopleHandler.newProxy;

public class HappyPeopleTestDrive {
    public static void main(String[] args) {
        IHappyPeople passengerByAir = newProxy(new PassengerByAir());
        IHappyPeople passengerByCoach = newProxy(new PassengerByCoach());
        IHappyPeople passengerByTrain = newProxy(new PassengerByTrain());

        System.out.println("Let's Go Home For A Grand Family Reunion...\n");

        System.out.println("Tom is going home:");
        passengerByAir.subscribeTicket();
        passengerByAir.travel();
        passengerByAir.celebrate();

        System.out.println("\nRoss is going home:");
        passengerByCoach.subscribeTicket();
        passengerByCoach.travel();
        passengerByCoach.celebrate();
```

```
            System.out.println("\nCatherine is going home:");
            passengerByTrain.subscribeTicket();
            passengerByTrain.travel();
            passengerByTrain.celebrate();
        }
}
```

以下是某一次的执行结果：

```
Let's Go Home For A Grand Family Reunion...

Tom is going home:
Buying a ticket...
Method subscribeTicket() used: 109ms
Travelling by Air...
Method travel() used: 203ms
Happy Chinese New Year!
Method celebrate() used: 297ms

Ross is going home:
Buying a ticket...
Method subscribeTicket() used: 109ms
Travelling by Coach...
Method travel() used: 594ms
Happy Chinese New Year!
Method celebrate() used: 296ms

Catherine is going home:
Buying a ticket...
Method subscribeTicket() used: 110ms
Travelling by Train...
```

```
Method travel() used: 390ms
Happy Chinese New Year!
Method celebrate() used: 313ms
```

这样，我们把记录时间的逻辑封装在了 HappyPeopleHandler 类里，你再也看不到重复的代码。

注意：

细心的读者可能发现了，我们在测试代码中未直接调用 HappyPeople 的 celebrateSpringFestival()方法，而是依次调用了它的 subscribeTicket()，travel()和 celebrate()这三个方法。如果直接调用 celebrateSpringFestival()方法，你就会发现并未执行计时逻辑，这是什么原因呢？

动态代理把方法请求交给了 HappyPeopleHandler 的 invoke(Object proxy, Method method, Object[] args)方法，它会调用目标对象的 celebrateSpringFestival()方法，即 HappyPeople 对象的 celebrateSpringFestival()方法，目标对象调用自己的这三个未织入增强的方法，而不是调用代理对象的那三个可以织入增强的方法，所以没有打印出执行时间。

如何解决这个问题呢？其实方法很多，我们把庆祝团圆封装成一个独立的 Celebration 类，它的方法为 celebrateSpringFestival(IHappyPeople ppl)，代码大致如下所示：

```
public class Celebration {
    public void celebrateSpringFestival(IHappyPeople ppl) {
        ppl.subscribeTicket();
        ppl.travel();
        ppl.celebrate();
    }
}
```

这样，客户对象传入织入增强的代理对象，执行 ***Celebration*** 对象的 ***celebrateSpringFestival(IHappyPeople ppl)***方法即可。

其实聪明的读者可能想到了在 HappyPeople 使用如下方法：

```
public class HappyPeople{
    public static void celebrateSpringFestival(IHappyPeople ppl) {
        ppl.subscribeTicket();
        ppl.travel();
        ppl.celebrate();
    }
}
```

这种方法虽然可以解决代理对象未彻底代理所有方法引起的问题，但是从模型上来考虑，它是欠妥的。***HappyPeople***类的职责会变得不明确——为什么不是自己而是别人来庆祝团圆？并且使用static方法会丧失面向对象的所带来的继承和多态等特性，不易扩展。一个优秀的软件开发人员，不仅要熟悉各种模式，还需要为问题抽象出精炼的模型，关于如何提炼模型，请参见 下章。

因为 Java 语言自动支持动态代理技术，所以不需要依赖于任何第三方软件和库，移植性和兼容性得到了保证。但是，一方面动态代理只能局限于接口的代理，不能对类进行代理；另一方面，通过上述示例，我们看到由于 Java 动态代理的实现机制问题——把方法转发给了目标对象，逻辑织入得不够彻底。

动态代理作为AOP的一种实现技术，被广泛应用到了诸多框架之中，例如 Nanning，Spring框架，Pico Container等等。而这些成熟的AOP框架往往结合一些其他AOP技术（如 动态字节码，拦截器框架等AOP实现技术），以实现更强大的AOP框架。

15.3.3.2 动态字节码

由于 Java 语言的开放性，涌现出了很多动态字节码工具，像 Apache 的 BCEL，Jboss 的 Javassist，ObjectWeb 的 ASM 和开源工具 CGLib 等，它们可以在运行时分析、生成和改变 Java Class 文件，同样，在运行时可以非常方便地织入增强，是 AOP 实现的另一条途径。这方面比较流行的工具是 CGLib，它曾被 Hibernate 早期版本成功地应用于延迟加载等方面，我们这里就以其为例作介绍。

CGLib 的 ***MethodInterceptor*** 和 InvocationHandler 一样，都能统一控制被代理的方法，

使用二者的区别不大，代码如下所示：

```
import net.sf.cglib.proxy.MethodInterceptor;
//other imports…

public class HappyPeopleHandler implements MethodInterceptor {

    @Override
    public Object intercept(Object o, Method method, Object[] objects, MethodProxy
methodProxy) throws Throwable {
        Object result = null;
        try {
            boolean intercept = isMethodQualified(method);

            if (intercept) {
                stopwatch.reset();
            }

            result = methodProxy.invokeSuper(o, objects);

            if (intercept) {
                stopwatch.info("Method " + method.getName() + "() used");
            }
        } catch (Exception e) {//handle exceptions...
            e.printStackTrace();
        }
        return result;
    }
}
```

可以看到，除过 ***intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy)***的方法签名不同之外，代码并未和 Java 动态代理的 InvocationHandler

实现类有很大区别。

CGLib 不仅可以代理接口，还可以代理类，所以这里我们不再创建接口。以下代码显示如何使用 CGLib 生成代理类的对象：

```
public static HappyPeople newProxy(HappyPeople delegate) {
    Enhancer enhancer = new Enhancer();
    enhancer.setSuperclass(delegate.getClass());
    enhancer.setCallback(new HappyPeopleHandler());

    HappyPeople o = (HappyPeople)enhancer.create();
    return o;
}
```

我们的测试代码如下所示：

```
import static pattern.chapter5.bytecode.HappyPeopleHandler.newProxy;

public class HappyPeopleTestDrive {
    public static void main(String[] args) {
        HappyPeople passengerByAir = newProxy(new PassengerByAir());
        HappyPeople passengerByCoach = newProxy(new PassengerByCoach());
        HappyPeople passengerByTrain = newProxy(new PassengerByTrain());

        System.out.println("Let's Go Home For A Grand Family Reunion...\n");

        System.out.println("Tom is going home:");
        passengerByAir.celebrateSpringFestival();

        System.out.println("\nRoss is going home:");
        passengerByCoach.celebrateSpringFestival();
```

```
            System.out.println("\nCatherine is going home:");
            passengerByTrain.celebrateSpringFestival();
        }
}
```

这里给出了测试的某一次执行结果，如下所示：

```
Let's Go Home For A Grand Family Reunion...


Tom is going home:
Buying a ticket...
Method subscribeTicket() used: 109ms
Travelling by Air...
Method travel() used: 203ms
Happy Chinese New Year!
Method celebrate() used: 297ms


Ross is going home:
Buying a ticket...
Method subscribeTicket() used: 94ms
Travelling by Coach...
Method travel() used: 609ms
Happy Chinese New Year!
Method celebrate() used: 297ms


Catherine is going home:
Buying a ticket...
Method subscribeTicket() used: 94ms
Travelling by Train...
Method travel() used: 406ms
Happy Chinese New Year!
Method celebrate() used: 297ms
```

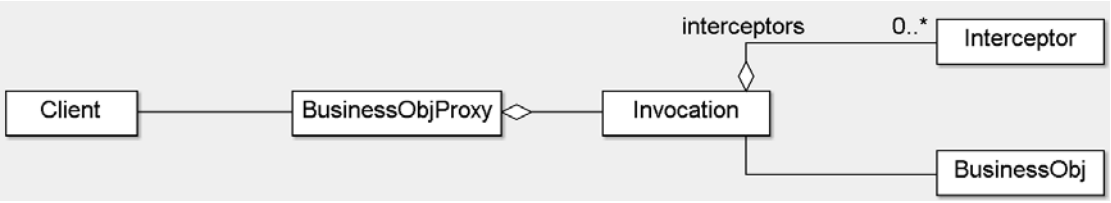
细心的读者可能早就发现，我们这次直接调用 HappyPeople 类的 celebrateSpringFestival()方法而完成了执行时间的计算，这是由于 CGLib 生成代理类的方式和 J2SE 动态代理生成代理类的方式不同引起的：J2SE 动态代理生成代理类继承于 java.lang.reflect.Proxy 类，所有方法指向了 InvocationHandler 实现对象的 invoke(…)方法，而 invoke(…)方法把请求再次转发给目标对象处理；CGLib 生成的代理类继承于被代理类/目标类，这样，上述执行代理对象的 celebrateSpringFestival()方法时，而不会把请求直接转发给目标对象，而是执行自己被织入增强的方法，也正是由于此缘故——使用继承的方式创建代理类，CGLib 不能代理任何 final 的方法和类。

通过CGLib的介绍，旨在希望大家对Java字节码工具在AOP方面的实现上有个大体的认识，由于这种工具很多，限于篇幅，不再赘述。关于CGLib的详细使用，有兴趣的读者可以登录 http://cglib.sourceforge.net/。

15.3.3.3 拦截器（Interceptor）框架

拦截器（Interceptor）一般在 Command 框架里都有实现，它能围绕被执行的业务对象织入增强，我们把切面都封装在拦截器的具体实现类里。当然拦截器框架是使用 OOP 语言设计的一个框架，其中往往结合使用了 Java 动态代理，字节码工具和 Java 反射（Reflection）等技术。

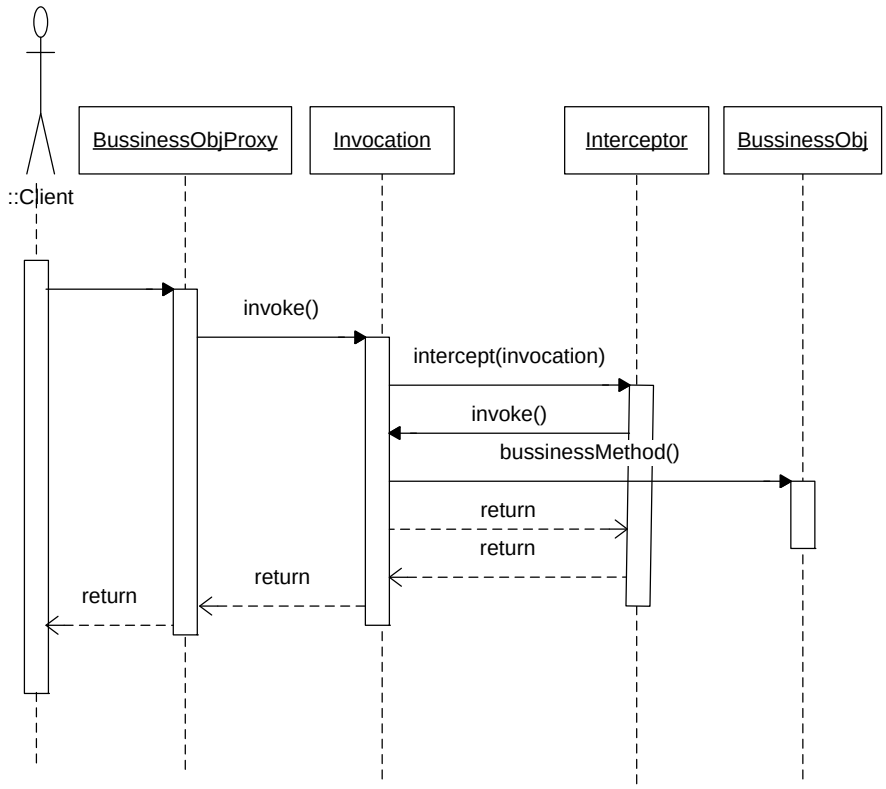
一般地，拦截器框架的结构如下所示：



- 客户对象直接使用的是代理对象 BusinessObjProxy，它持有控制执行逻辑的 Invocation 对象。
- Invocation 持有 Interceptor 链和真正被代理的 BusinessObj 对象，和前述 invoke(…)方法的功能一样，负责织入增强，即依次调用拦截器进行拦截，最终执行 BusinessObj 对象的方法。
- Interceptor 实现类便是我们的切面。

- BusinessObj 是目标对象，即要被织入增强的对象。

Invocation 负责 BusinessObjProxy 的执行，在执行过程中它负责 Interceptor 链的每个拦截器方法被依次执行，以及被代理对象 BusinessObj 的执行。为了便于描述，我们给出拦截器的序列图，如下所示：



Client 调用 BusinessObjProxy 对象的 invoke()方法，BusinessObjProxy 把请求转给 Invocation 对象，Invocation 执行完拦截器链，接着执行 BusinessObj 的方法，最后执行拦截器链未执行完的堆栈逻辑，并返回执行结果。

我们在这里实现一个非常简单的拦截器框架，来说明拦截器框架的工作原理。

首先实现定义一个 Invocation 接口，代码如下：

```
public interface Invocation {

    void init(BusinessObjectProxy proxy);


    void addInterceptor(Interceptor interceptor);
```

```
    void removeInterceptor(Interceptor interceptor);

    Object invoke() throws Exception;

    BusinessObjectProxy getProxy();
}
```

接下来我们给出一个简单的 Invocation 实现，经上述介绍，我们知道，拦截器框架织入逻辑会在此类中实现，如下所示：

```
import java.lang.reflect.InvocationTargetException;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class DefaultInvocation implements Invocation {
    private List<Interceptor> interceptorList = new LinkedList<Interceptor>();
    private Iterator<Interceptor> interceptors;
    private Object businessObject;
    private BusinessObjectProxy proxy;

    public DefaultInvocation(Object businessObject) {
        this.businessObject = businessObject;

    }

    @Override
    public void init(BusinessObjectProxy proxy) {
        this.proxy = proxy;
        interceptors = interceptorList.iterator();
    }
```

```
@Override
public void addInterceptor(Interceptor interceptor) {
    interceptorList.add(interceptor);
}



@Override
public void removeInterceptor(Interceptor interceptor) {
    interceptorList.remove(interceptor);
}



@Override
public Object invoke() throws Exception {
    //other logic...
    Object result = null;
    //Traverse all interceptors
    try {
        if (interceptors.hasNext()) {
            Interceptor interceptor = interceptors.next();
            interceptor.intercept(this);
        } else {
            //To invoke real business object's method
            result = invokeBusinessObjectMethod();
        }
    } catch (Exception e) {
        throw e;
    }
```

```
            //other logic...
            return result;
        }

        @Override
        public BusinessObjectProxy getProxy() {
            return proxy;    //To change body of implemented methods use File | Settings | File Templates.
        }

        private Object invokeBusinessObjectMethod() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {
            String methodName = proxy.getConfig().getMethodName();
            businessObject.getClass().getMethod(methodName).invoke(businessObject);
            return null;
        }
}
```

我们的织入逻辑在 ***invoke()***方法里：首先判断拦截器链是否未执行完毕，即 ***if (interceptors.hasNext())***，如果没有，则取出下一个拦截器继续执行；如果拦截器链执行完毕，这才去执行真正被代理对象的方法，即 ***result = invokeBusinessObjectMethod()***。

invokeBusinessObjectMethod()方法会调用目标方法，我们这里的实现非常简单，使用反射得到配置的方法并执行。

ProxyFactory 工厂类用于创建 BusinessObjectProxy 对象，代码如下：

```
public class ProxyFactory {

    public static BusinessObjectProxy createProxy(Invocation invocation, Config config) {
        BusinessObjectProxy proxy = new BusinessObjectProxy(invocation,
```

```
config);
            invocation.init(proxy);
            return proxy;
        }
}
```

Interceptor 接口的代码如下所示：

```
public interface Interceptor {
    Object intercept(Invocation invocation) throws Exception;
}
```

我们这里实现记录时间的拦截器 StopWatchInterceptor，代码如下：

```
public class StopWatchInterceptor implements Interceptor {
        private Stopwatch stopwatch = new Stopwatch();

        @Override
        public Object intercept(Invocation invocation) throws Exception {
            stopwatch.reset();

            //invoke
            Object result = invocation.invoke();

            stopwatch.info("Method " +
invocation.getProxy().getConfig().getMethodName() + "() used");
            return result;
        }
}
```

我们在 **Object result = invocation.invoke()**之前加入这句 **stopwatch.reset()**开始计时，

Object result = invocation.invoke()这句非常重要，保证了 Invocation 能够继续执行拦截器链上其他还未执行的拦截器以及目标对象的被拦截方法。最后，由于方法的堆栈调用，Invocation 仍然要返回该拦截器，执行连接点 ***invocation.invoke()***之后的逻辑，即打印出所消耗的时间，即 ***stopwatch.info("...")***。

我们这里做了一个非常简单的配置类 Config，用于描述对象的哪个方法需要拦截，当然你可以撰写更为复杂的配置支持你的框架，代码如下所示：

```
public class Config {
    private String methodName;

    public Config(String methodName) {
        this.methodName = methodName;
    }

    public String getMethodName() {
        return methodName;
    }
}
```

在使用之前，和其他框架一样，我们需要建立上下文环境，我这里写了一个简单的 NaiveInterceptorFrameWork 类初始化环境上下文，如下所示：

```
public class NaiveInterceptorFrameWork {
    private static NaiveInterceptorFrameWork context;
    private Map<String, BusinessObjectProxy> services = new HashMap<String, BusinessObjectProxy>();

    public static void configure() {
        load(new NaiveInterceptorFrameWork());
```

```
        PassengerByAir passengerByAir = new PassengerByAir();
        Invocation subscribeTicketInvocation = new
DefaultInvocation(passengerByAir);
        subscribeTicketInvocation.addInterceptor(new StopWatchInterceptor());
        BusinessObjectProxy subscribeTicket =
ProxyFactory.createProxy(subscribeTicketInvocation, new
Config("subscribeTicket"));
        context.services.put("subscribeTicket", subscribeTicket);

        Invocation travelInvocation = new DefaultInvocation(passengerByAir);
        travelInvocation.addInterceptor(new StopWatchInterceptor());
        BusinessObjectProxy travel = ProxyFactory.createProxy(travelInvocation,
new Config("travel"));
        context.services.put("travel", travel);

        Invocation celebrateInvocation = new DefaultInvocation(passengerByAir);
        celebrateInvocation.addInterceptor(new StopWatchInterceptor());
        BusinessObjectProxy celebrate =
ProxyFactory.createProxy(celebrateInvocation, new Config("celebrate"));
        context.services.put("celebrate", celebrate);
    }

    private static void load(NaiveInterceptorFrameWork naiveInterceptorFrameWork)
{
        context = naiveInterceptorFrameWork;
    }
```

```
    public static BusinessObjectProxy lookup(String proxyName) {
        return context.services.get(proxyName);
    }
}
```

我们为 ***PassengerByAir*** 对象的三个方法，subscribeTicket()、travel()和 celebrate()，配置了 ***StopWatchInterceptor*** 拦截器，这样，我们测试代码如下所示：

```
public class InterceptorTestDrive {
    public static void main(String[] args) throws Exception {
        NaiveInterceptorFrameWork.configure();

        BusinessObjectProxy subscribeTicket =
NaiveInterceptorFrameWork.lookup("subscribeTicket");
        BusinessObjectProxy travel = NaiveInterceptorFrameWork.lookup("travel");
        BusinessObjectProxy celebrate =
NaiveInterceptorFrameWork.lookup("celebrate");

        subscribeTicket.execute();
        travel.execute();
        celebrate.execute();
    }
}
```

执行结果如下所示：

```
Buying a ticket...
Method subscribeTicket() used: 140ms
Travelling by Air...
```

```
Method travel() used: 203ms

Happy Chinese New Year!

Method celebrate() used: 312ms
```

优点：

- 我们不需要在编译和加载类时做任何额外的操作，方便编译和发布。
- 从上述实现我们知道，可以对任何对象进行拦截。
- 从上述例子知道，我们可以对同一类型的不同对象可以织入不同的逻辑，只要为它们配置不同的拦截器即可。
- 拦截器的代码不会侵入目标的类和方法。

缺点：

- 框架的不同，拦截器的能力强弱差别比较大。
- 框架的一次执行只能拦截一个目标方法，要统计三个方法的执行时间，我们要配置三个拦截器链。

拦截器在很多框架中均有实现，最常见的有Jboss的拦截器，xwork框架[17]的拦截器，和Spring AOP的拦截器等等，这些框架往往结合其他AOP技术，例如J2SE动态代理和动态字节码。

15.3.3.4 源代码生成

在 EJB 早期实现里，我们经常使用这种方式生成织入增强的 Java 源代码，这种方式随着 Java 动态代理技术和字节码工具的出现已经开始逐渐退出潮流。

15.3.3.5 在编译时织入二进制代码

我们可以使用特定的编译器在编译时织入增强，这种方式的织入，运行性能总比运行时织入要优越一些[18]。它的运行速度和Java源代码生成可以媲美，这二者都是静态织入，目标对象被织入的功能在编译时就决定了。

---

[17] xwork 框架是一款非常出色的 command 框架软件，著名的 struts2 和 webwork 框架就是使用其实现的，它的拦截器实现了对 command/action 的动态拦截。

[18] 其实运行时织入并没有想象中的那么慢，现在由于 JVM 性能的提升，对动态代理等做了很大的优化，那些字节码工具也在运行时表现十分出色。

AspectJ 和 AspectWerkz 都支持这种策略，Cobertura 是一款统计测试代码覆盖率的软件，它就是在编译时把统计覆盖率的逻辑织入已编译好的 Java 二进制代码里，然后运行这些二进制文件，完成了覆盖率的统计。

15.3.3.6 定制类加载器（Class Loader）

我们还有一种织入方式，那就是在类加载器加载类时织入增强，由于 Java 类加载器的可扩展性，我们可以定制自己的类加载器，让其在加载类时织入增强。AspectWerkz（称之为在线织入，Online Weaving）和 AspectJ 都支持这种方式。

但是这种方式可能在某些 J2EE 服务器上并不能被使用，因为这些服务器一般有一套自制的完整类加载结构体系，这种方式可能导致不能定制我们自己的类加载器，否则会引起冲突。

15.4 AOP框架介绍

这里介绍一些 AOP 流行框架，有兴趣的读者可以登录它们的网站作更深入的了解：

- AspectJ

  AspectJ 是目前最流行也最完善的 AOP 实现，它给自己的定义是：

  > **AspectJ is a seamless aspect-oriented extension to the Java™ programming language, Java platform compatible, easy to learn and use.**
  >
  > **AspectJ 是一个 Java™ 编程语言在面向方面上的无缝扩展，并且与 Java 平台兼容，容易学习和使用。**

  AspectJ 支持编译时，类加载时和运行时织入。其中，编译时入不仅支持源代码的生成（其称之为 Compile Weaving），也支持对已编译好的二进制文件（例如 jar 包，class 文件）织入增强（其称之为 Post-compile Weaving，也叫二进制织入，Binary Weaving）。正如定义所说，它是 Java 的扩展，这也体现在语法上面，增加了初学者的难度。

- Nanning

  Nanning 是一款非常小巧的 AOP 框架，名字源于中国广西南宁。它使用了 Java 反射技术实现了拦截器框架，可以实现基于方法的拦截。该项目最后发布日期

是 2003 年 8 月。

- AspectWerkz

  AspectWerkz 是一款非常优秀的 AOP 框架，它使用了字节码工具，可以在编译时、类加载时和运行时修改二进制代码。该项目最后发布日期是 2005 年 3 月。

- Spring AOP框架

  Spring 现在不仅仅包含 Spring 框架，在 Webflow，SOA，Andriod，云计算等各个方面均有出色的贡献。Spring 框架本身也包含一个 AOP 框架，但并未和 Spring 框架强行绑定，读者可以在使用时自由选择。

  Spring AOP 框架最开始使用了 Java 动态代理和动态字节码工具，其本身只支持方法级别上的拦截和增强，后来由于实现了和 AspectJ 的集成，所以可以完成属性（field）的访问拦截等 AspectJ 才有的功能。并且由于 Spring AOP 和其 IoC 框架实现了无缝集成，使其成为使用 Spring 框架做项目的上选 AOP 方案。

- JBoss AOP

  JBoss AOP 的框架目前已经支持属性（field），方法，构造方法上的拦截，但是它是一个 EJB 的容器，所以其 AOP 框架和 EJB 服务器实现了绑定，读者选择时需要在移植性方面作考量。

15.5 AOP 联盟（AOP Alliance）

AOP联盟为Java和AOP爱好者建立了一个AOP的规范，它不提供具体实现，而是在AOP联盟成员之间达成标准，旨在使你的代码在这些成员的软件之间可以互相移植。由于AOP联盟对这方面的发展一直持谨慎态度，目前只能支持对构造方法和成员方法的拦截。

15.6 使用AOP编程的风险

虽然 AOP 弥补了 OOP 水平方向进行模块化的不足，避免代码的重复，但是我们对 AOP 编程一直持谨慎的态度，这是因为：

- 一个类如果被多个切面横切，可能会产生叠加的效果，有些叠加的效果正是你所需要的，但有些意外的叠加效果可能会带来严重的，无法预知的错误。特别是在使用 JPM（Joint Point Model）时，由于很容易加入切入点的规则，所以很容易造成意想不到的结果。

为了避免因叠加效果而产生的无法预料的问题，AspectWerkz 推荐使用正交的连接模型（Orthogonal weaving model），即切面可以以任一顺序组合使用。然而这往往并不现实，有时候我们希望多个切面之间存在先后顺序，比如，我们总是希望验证用户安全性的增强在其他操作之前执行，而且很多框架都支持顺序执行，例如拦截器框架中，按照配置顺序执行拦截器。

- 由于增强逻辑可以横切多个模块，这样，很难让各个模块的开发人员都知道我们为某些连接点织入了增强，由此可能造成相关开发人员不知道究竟执行了哪些逻辑代码。

在使用 AOP 的编程过程中，应该规范团队开发，对相关代码进行有效地约束和管理，譬如所有切入点的定义遵从一定的命名规范，或者在指定的目录下，这样方便开发人员查阅代码。并且在开发过程中，可以完善相关文档，对软件中哪些地方使用了 AOP 技术一目了然。

- 不容易测试和调试，如果你选择生成织入增强的源代码，或者使用编译器织入，或者在类加载期织入，这些代码都不易调试，排错困难。

15.7 OOP还是AOP

我们使用 SoC 方法把系统分解成小单元时，就不可避免地遇到需要把一些洒落在零散小单元里的相同逻辑模块化为同一单元的问题。不管是过程式编程/函数式编程，还是 OOP 编程，只要你把问题由大到小进行分割，都会出现相同的问题。由于近年来 OOP 编程的迅猛发展，我们往往把这个诟病归归结于 OOP，当出现 AOP 编程技术时，自然而然产生了 AOP 编程是否能像 OOP 在某些开发领域取代过程式编程/函数式编程一样取代 OOP 编程的疑问。其实纵向切割系统是非常有用的，大多数问题在纵向上分解之后非常容易得到解决，如果横向分解系统，那就会产生大量的切面，大量切面产生的叠加效果往往是难以估计的，灾难性的。可见，单纯的使用水平切割系统和垂直切割系统都会引入问题，只有把二者结合起来才能帮助我们更加完善地模块化问题，现在大多数软件开发人员都认为 AOP 是 OOP 的重要补充。

15.8 总结

AOP 经过 10 多年的发展，在企业级软件开发方面发挥了举足轻重的作用，我们在享受 AOP 编程带来方便的同时，也要注意 AOP 编程带来的负面因素，特别是切面过多所产生的不可控的叠加效果。

OOP 和 AOP 在模块化和代码重用方面功不可没，重复并没有因此而消失。不管怎样，它们二者模块化的最小单元都是类，粒度相对来说还是比较小的，有时候我们需要更为宏观的重用，例如重用服务，系统等这些更大粒度的单元。

比如，公司有一个日常收发邮件的系统，现在正在开发一个新的财务系统需要为一些紧急事件发邮件通知，如果为此财务系统再开发一个新的邮件系统，这将会是巨大的浪费：对于邮件系统来说，发送一封通知和发送一封普通邮件并无很大区别。为此，为了重用这些软件服务，出现了面向服务的架构（SOA，Service-Oriented Architecture），面向服务的架构提供更大粒度[19]的单元封装，便于服务之间的重用，它的比较流行的实现是Web Service。

SOA 让我们在重用方面迈出了更远的一步，但并不表示重复的结束，其实 DRY 涉及的内容非常广泛，并非一两种技术就能完全实现，它涉及的领域同样十分广泛，不仅在软件开发领域，其他领域都会出现重复的知识，并且这些知识的粒度同样有大有小。

[19] 面向服务的架构是粗粒度的架构，尽管有人会提出粒度可大可小，但是实际都是一些粗粒度地服务/组件接口，只是在这些粗粒度的服务/组件里，粒度还能根据大小进行分类。举一个极端的例子，如果你为isNull(Object o)这样小的方法也建立一个 Service 的话，那将是笑话。

第16章　面向对象开发

16.1 概述

懂得设计模式并不等于已经学会了使用面向对象进行开发，虽然熟悉这些模式有助于使用这些成功的经验解决问题，但是由于问题本身的复杂性和隐藏性等特点，如何使用面向对象的方法建立合理的模型成为了关键。本章，笔者将和大家一起简单地讨论如何建立 OOP 模型。

16.2 写在面向对象设计之前

我是一个 Agile 和 Domain-Driven Design（简称 DDD）的爱好者。在以下各节的讲述中，所提到的一些开发设计方法很大部分和它们有关，如果你是它们的反对者，可以忽略此章。这里首先简单介绍一下这两个概念。

Agile兴起于上世纪 90 年代，像大家非常熟悉的 Scrum（1995），E**x**treme **P**rogramming（极限开发，简称XP，1996）等轻量级软件开发方法现在都被认为是Agile的实现方法。在 2001 年 2 月，17 个软件开发大师在美国犹他州（Utah）的一处滑雪胜地度假，期间讨论了各种轻量级软件开发的方式，最终一起发表了题为《Manifesto for Agile Software Development》的软件开发宣言。在此宣言中，提出了 12 条软件开发原则，由此便形成了我们以后知道的敏捷开发和敏捷联盟。

Domain-Driven Design（DDD），是一门面向对象设计的哲学[20]，在解决复杂领域问题方面，卓有成效。Eric Evans在《Domain-Driven Design: Tackling Complexity in the Heart of Software》一书，详细地讲述了如何进行DDD的开发，此书极具影响力，也是笔者最喜欢的软件设计图书之一。对于极限开发（XP），缺乏经验的新手往往会带来过于简单的开发，而一些经验丰富的老手在开发设计中，往往沉浸于技术和设计，带来了过度的开发（Over-engineering），DDD开发介于过度开发和过于简单的开发之间，能为复杂的领域迅速地迭代提炼出出色的模型。

我们有很多人已经使用模式好长时间了，是否享受到面向对象设计的好处，并不见

[20] 参见 Domain-Driven Design: Tackling Complexity in the Heart of Software , Eric Evans, 2003, 在序言中，作者原话为：“… it has never been formulated clearly, a philosophy has emerged as an undercurrent in the object community, a philosophy I call domain-driven design”。

得。软件的核心是模型，如果在不能表达丰富功能需求的模型上应用模式，将不会有太大成效，往往因为误使了模式而带来开发的泥沼。例如，如果使用 模板方法模式解决 第 12 章提出的问题时，则会弄巧成拙。本章我将给读者分享一些我在面向对象开发设计过程中的经验，虽然这不是本书的重点，但对读者进行OOP开发设计有重要意义，其中有些部分虽然超越了OOP的范畴，但都属于软件开发设计的范畴。

16.3 汲取知识

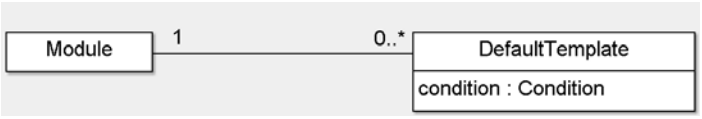
前不久，我们承接了一个项目，其中一个模块需要做两个系统的集成。一开始，做该集成开发的工作人员有两个，后来，过了一个月，其中一位离开了公司。又过了一个月，集成仍然迟迟不能发布，于是经理派我去着手解决这个问题。由于我只了解其中一个系统，对另一系统，除名字之外我一无所知，我并没有立即打开繁冗的技术文档，而是向之前的开发人员了解需求。开发人员告诉我，没有正式的需求文档，只有一封潦草的邮件，其中关于集成需求的描述只有一句话：系统 A 和系统 B 的集成。

由于领域专家不在，我就坐下来和该开发人员进行了半个小时左右的沟通，慢慢地我发现该开发人员对需求并不太理解，我提到的很多细节问题他不能如愿作答。此时，他对需求的理解就是对把系统 A 的一条记录发送到系统 B，然后生成一条新记录，由于数据格式不匹配，二者之间需要定义一个映射。他在其上花了很多时间做尝试，最后都被领域专家否定，他也不知道领域专家需要什么。

所幸的是，一天后，领域专家赶了回来，我直接找他了解需求，在长达 2 个小时的讨论中，我发现之前的交流存在一些问题：领域专家未能完全表达他的想法，也不知道开发人员能否按照自己的想法做出功能，而软件开发人员也不知道真正的需求是什么，双方并未完全沟通自己的想法。根据这次探讨，我了解到：由于系统 A 和系统 B 之前就有集成，后来系统 B 中的一个模块发生了变化，主要是为这模块的模型新添了一些属性，导致使用之前的映射不能创建一条完整的记录，为了在不修改系统 A 的前提下，要使之前的集成正常运行，我们要给那些属性配置一些默认值。

这次讨论之后，我看了下系统 A 和 B 集成的技术文档，然后画出一个简单的设计草图，你也许发现，它简单到了极点：

需求比较简单，但在动笔之前，我拿着设计草案又和领域专家进行了一次讨论，发现我们可以根据记录的某些属性，给出更为具体的默认值，比如，如果记录的一个属性**优先级**如果值为**高**，那么新添加的这个属性**影响**我们最好设置为**重要**。而之前没有深入讨论这个的原因是，他觉得这样可能难于实现，所以自己也未朝这个方面多想。经过这次讨论之后，我们确定了设计方案图，如下所示：



Condition 表示如果 Module 记录信息满足该条件，那么就是用此模板提供的默认值给记录赋值。真实的设计草图当然比这个复杂，这里为了讨论的方便，我们就此为止不再给出细节。

经过这两次的讨论，我和领域专家敲定了我们的需求，他也理解了我们能为用户做些什么，我们也知道他们想要什么。后来的事情就相对简单了许多，经过我 2 天多的苦力编程，终于顺利完成了开发。这样，2 个月未做完的需求我一共花了不到 4 天就做完了（包括讨论需求），这并不是我的技术能力比之前的开发人员好很多，原因在于：开发人员在编写软件时，对需求和领域知识总是未引起足够重视，只专注于学习二者如何集成的一些技术问题，越是简单的需求越是不重视，导致做了很多无用的工作；领域专家并非对领域的每一方面有着极其深入的了解，只有和他们进行深入讨论，他才会明白你能做什么，不能做什么，也才能开发出比他们当初想象更为强大的功能。

能够高效建模的人是具有超强的消化领域知识的能力，也能帮助用户分析用户到底想要什么，消化知识需要的不是独立的单方面的行为，而需要开发人员去主导，开发团队和需求专家一起分析信息，消化知识，才能建立有效的模型。

16.4 横看成岭侧成峰

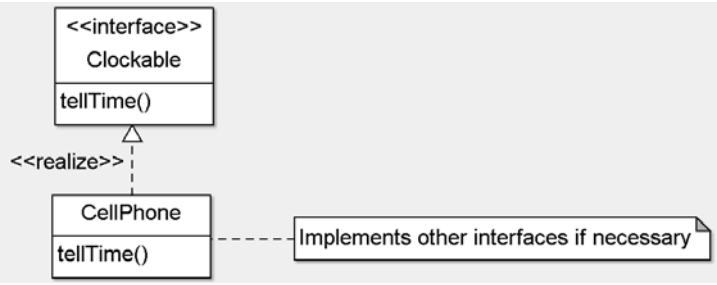
我们很多人都有表，它们形形色色，各种各样，有的是用于装饰的，有的是可以在水里显示时间的，也有的是专门记录体育比赛时间的……，我自己也有一只表，让我来说说它的功能吧。

我的这只表有闹钟的功能，可以提醒我上班不要迟到；能播放 MP3，我可以用它听音乐放松；使用它还能查阅各种文件，还能照相留念；这些还不够，我的这只表还提供上网浏览网页和收发电子邮件的功能，让我随时随地都能工作（我不是工作狂，千万别凭此判断）；这些还不够绝，更绝的是，我能用它收发短信，接听电话。

全世界有数以亿计这样的手表，我相信你也有块这样的手表，只是在日常生活中，我们大部分人把它们叫做**手机**。自从我有了手机后，我再也没有用过手表了，每当有人向我询问几点时，我都会拿它出看时间。它到底是手表还是手机呢？

苏东坡当年在游览庐山时，被美景所迷，“往来山南北十余日”后，终于做出“横看成岭侧成峰，远近高低各不同”的诗句。当然他在那里的寓意和我这里的用意并不相同，我在这里要强调的是，站在不同角度分析事物时，事物所呈现的形态不尽相同。从某种角度来认识手机，它就是“手表”。

这么说的目的并不是让我们不去认识必要的手机的特征来完善一个合理的模型，而是强调让那些不相干的信息别去干扰我们对模型的认识。那么我们在抽象一个时钟系统的模型时，手机的通话功能对我们来说是无用的，不需要去提取一个通话模型，所以，而手机和时钟之间的关系可能如下图所示：



在软件设计过程中，即使对于同一事物建模，也不一定能够使用相同的模型，因为你站的角度不同。譬如，在一个简单的工资系统里，经理的角色和其他员工的角色没太大的区别，每个月都需要给他们账户发钱，工资系统不需要了解经理下属有多少职工，而在职工管理系统里，我们需要知道经理下属有多少职工，所以二者模型并不相同。甚至在同一个应用里，相同事物的模型也不一定完全相同。Eric Evans 在《Domain-Driven Design: Tackling Complexity in the Heart of Software》[21]对此进行

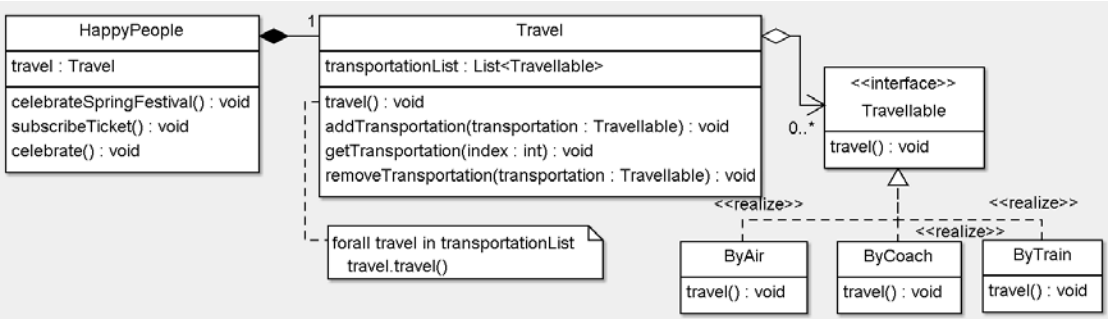
[21] 参见《Domain-Driven Design: Tackling Complexity in the Heart of Software》，作者 Eric Evans，在 Part IV: Strategic Design 的 Chapter Fourteen. Maintaining Model Integrity 一章，指出一个复杂的应用里，实现相同的领域模型非常困难，往往存在多个领域模型。

了更为深刻的探讨。

16.5 提炼模型

我们软件设计的其中一个目标是实现模块内部的高内聚，要实现这个目标，对象应该职责分明，如果对象承担了应该由其他对象应该承担的职责时，它是低内聚的，换句话说，高内聚需要限制对象的职责，尽可能为它们分解单一的不重复的职责。

在 第 12 章，我们讲述了 策略模式，从模型来看，其实它还不算是精炼的。让我们再仔细思看看这个模型，我们为HappyPeople类添加了addTransportation(Travellable transportation)，getTransportation(int index)和removeTransportation(Travellable transportation)等方法，添加这些方法的意图无外乎就是管理Travellable对象。随着业务的深入，或许需要更多的方法来管理它们，这样HappyPeople多出了很多与Travellable管理相关的职责。其实更进一步，我们应该分解这个类，把管理Travellable对象的职责封装在一个新类里，这样便形成了如下结构：



这样，类之间的结构便是高内聚的，对于 Travellable 对象的处理我们都在 Travel 对象里，HappyPeople 类便能更加关注自己的职责了。

我们看看 Travel 类的实现代码，大致如下所示：

```
public class Travel {
    private List<Travellable> transportationList;
    //other methods…

    public void travel() {
        for (Travellable travellable : transportationList) {
            travellable.travel();
```

```
        }
    }
}
```

这样，HappyPeople 类的代码如下所示：

```
public class HappyPeople {
    private Travel travel;

    public void celebrateSpringFestival() {
        subscribeTicket();
        travel.travel();
        celebrate();
    }

    //other methods…
}
```

于是，我们撰写测试的代码大致如下所示：

```
public class HappyPeopleTestDrive {
    public static void main(String[] args) {
        Travel travelByAir = new Travel();
        travelByAir.addTransportation(new ByAir());
        HappyPeople passengerByAir = new HappyPeople(travelByAir);

        Travel travelByCoach = new Travel();
        travelByCoach.addTransportation(new ByCoach());
        HappyPeople passengerByCoach = new HappyPeople(travelByCoach);

        Travel travelByTrain = new Travel();
```

```
            travelByTrain.addTransportation(new ByTrain());
            HappyPeople passengerByTrain = new HappyPeople(travelByTrain);

            Travel travelByAirAndCoach = new Travel();
            travelByAirAndCoach.addTransportation(new ByAir());
            travelByAirAndCoach.addTransportation(new ByCoach());
            HappyPeople passengerByAirAndCoach = new HappyPeople(travelByAirAndCoach);

            System.out.println("Let's Go Home For A Grand Family Reunion...");
            System.out.println("Tom is going home:");
            passengerByAir.celebrateSpringFestival();
            System.out.println("\nRoss is going home:");
            passengerByCoach.celebrateSpringFestival();
            System.out.println("\nCatherine is going home:");
            passengerByTrain.celebrateSpringFestival();
            System.out.println("\nJennifer is going home:");
            passengerByAirAndCoach.celebrateSpringFestival();
        }
}
```

最后，执行结果如下所示：

```
Let's Go Home For A Grand Family Reunion...
Tom is going home:
Buying ticket...
Travelling by Air...
Happy Chinese New Year!
```

```
Ross is going home:
Buying ticket...
Travelling by Coach...
Happy Chinese New Year!

Catherine is going home:
Buying ticket...
Travelling by Train...
Happy Chinese New Year!

Jennifer is going home:
Buying ticket...
Travelling by Air...
Travelling by Coach...
Happy Chinese New Year!
```

提炼模型不是在开发中的某一个阶段应该做的事情，而是一个不停迭代的过程，随着我们对领域知识的不断消化，我们也就能提炼出更为精准的模型。提炼模型比使用技术更为重要，一个好的软件架构能够使我们选用不同的实现技术（所谓的Reversibility[22]），而如果模型粗糙、拙劣，不管使用多么先进多么流行的技术，都是无法应对复杂的领域问题。软件的核心不是技术，也不是模式，而是模型。

16.6 应用设计模式

使用设计模式和提炼模型有时候是分不开的，有些设计模式可以直接拿过来做模型，例如像状态模式，策略模式和组合模式等等。在一个完整的软件里，不仅需要使用多个模式，可能需要把多个模式结合起来，比如，使用 代理模式时，可以结合工厂模式，让 工厂模式实例化代理对象；又如，在使用 组合模式时，为了使用没有实现

[22] 参见《The Pragmatic Programmer》一书的第二章：A Pragmatic Approach。

Component接口的既有类，可以使用 适配器模式达到此目的。

在挖掘领域问题时，应用这些设计模式的经验，可以把那些隐含的，深层次的概念找出来，提炼出更为有用的模型。

16.7 不能脱离实现技术

提炼模型时，如果只专注于业务模型，而忽略了实现技术，这样有可能导致因无法实现而最终搁浅。

三年前，我曾经开发了一款应用软件，当时为模型设计了回滚处理，由于使用了组合模式，当时并没有考虑到嵌套事务的回滚问题，我们当时的问题是：在一个树形对象图中，如果子对象的事务回滚了，我们需要对其父对象的所有子对象的事务回滚，并且父对象的事务也要回滚。比方说，对象a有三个子对象b，c和d，当b和c提交了操作，而d对象的事务需要回滚时，那么已经提交事务的父对象a，和兄弟对象b和c都需要回滚。由于当时只能使用简单的JDBC事务，这种需求是不能被支持的，虽然JTA（Java Transaction API）[23]事务可以实现（执行子对象提交之前，在父对象a设置回滚点，如果b，c提交成功，d失败，可以回滚到父对象a设置的回滚点），但是由于其中种种原因，我们并不能使用JTA。

当时的开发工作已经完成了 3/4，幸运的是，当时碰巧读到一篇文章，提到了 Ebay 电子商务网站的应用并没有使用数据库事务管理，由此大受到启发。我开始思考不是使用传统的数据库回滚，而是实现了业务操作的回滚，即如果 d 操作需要回滚，我让 a，b 和 c 对象执行业务上的回滚操作。试想，如果不是受这篇文章的启发，可能此项目将会失败。

我这里要强调的是，在任何时间，都要联系到实现技术。有时候即使技术能够实现，对你的团队来说，可能由于大家都未曾考虑过这些问题，当最终摆在面前时，大家便会手足无措。实现技术是多种多样的，即便你联系到的实现技术不是最好的，但它至少能解决你的问题；倘若刚开始你没有这样做，即便有最简单的技术实现，也可能因为你当时的无知而失败。

[23] 是 Java EE 分布式事务处理的 APIs，是关于跨越多个 XA 资源的分布式事务的 Java EE 标准。

此外，联系到技术有利于把握时间，成本等其他因素，降低其中的风险。上述例子之所以可以考虑实现业务上的回滚，是因为我们的应用从一开始就支持那些业务的回滚操作，如果不支持，那么重新开发将是极其复杂的事情，因为会引起核心的大修改，很可能导致该软件半途而废。

16.8 重构

什么是重构呢？Martin Fowler在《Refactoring: Improving the Design of Existing Code》一书对重构定义[24]如下：

> **Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.**
>
> **重构是采用这样一种方式改变软件系统的一个过程：它不改变代码的外部行为，但是提高了内部结构的质量。**

重构是一个持续的过程，提炼模型也属于重构，重构不仅能够提高代码质量，还能偿还技术债务[25]，为以后新功能的扩展提供了可能性。关于如何进行重构，大家可以好好研读Martin Fowler的这本书籍，它详细讲述了重构代码的过程。

但是，重构绝对不等于推翻了重做，有相当一部分初学者甚至一些所谓的“资深”开发人员可能由于追求完美的“偏执”所致，他们都喜欢推翻了重做。重新开发虽然可以规避之前的一些拙劣的设计，但是在重用之前的健壮代码上大打折扣；而且，由于之前的拙劣设计，代码也不易阅读，为移植和重写这些拙劣的代码带来了问题，往往因为不理解其中的“特殊含义”而重写的逻辑不能很好运行；另外，和一般的重构一样，新的代码总会引入新的未知问题。总之，重构并不等于推翻了重做，它也不是重构的最好办法，做推翻重做决定时一定要慎重。

一个良好的项目，开发步骤往往是由慢变快的，起步比较慢，在往后的迭代开发过程中，会变得越来越快；而一个急功近利型的项目往往一开始就快速启动，在随后的开发中，过分地强调了快速发布，导致开发了很多不可扩展代码，随着新的功能不断加入，开发设计变得越来越拙劣，项目也变得越来越慢。如果你的项目目前处

[24] 参见该书的前言部分对重构的定义。
[25] Ward Cunningham 在 1992 年在自己的一篇 Experience Report 里引入这个比喻。Vikas Hazrati 在这个链接里对技术债务进行了归纳和讨论：http://www.infoq.com/news/2009/10/dissecting-technical-debt。

境是这样，那么最好停下脚步，考虑是否要花点时间对它进行重构。

我们既要避免轻率地“推翻了重做”，也要避免因为安于现状而导致项目陷入泥沼。

16.9 过度的开发（Over-engineering）

几年前，曾参与了一个产品重构的开发，该产品有 4 个模块，为了避免风险起见，我们从一个最简单也最基本的模块入手，原计划是在半年内完成重构。然而经过了一年的开发仍未结束，设计人员总是觉得某些方面可以做得更好，特别是技术方面，他们经常发现某些技术上的一些小的以前未使用的功能，他们拼命地把这些功能加入软件，并且随着时间的发展，新的技术、新的软件和新版本的发布带来了更好的功能，这些最终让他们看花了眼，导致不断的推翻之前使用的技术，而未提炼模型，最终软件迟迟不能发布，功能却没有得到明显的增强。

一年之后，我便离开了那家公司。又过了两年，我碰巧遇到一位以前参与开发的工程师，他告诉我那个简单的模块还没有完成重构，其他模块也没有开始重构。

这是一个真实的故事，我至今还记得他们为了使用 Hibernate 的一个小小功能而大肆修改，导致其他使用该产品的项目停滞等待情景。虽然他们在不停地完善软件，但只注重了使用技术，而没有花时间去提炼完善的模型，殊不知最重要的是模型。其间往往导致不仅功能没有增强，反而因为模型的粗糙经常推翻以前所使用的新技术。

虽然导致三年后仍未完成一个模块重构的原因是多方面的，但是过度的开发是其失败的一个重要原因。从某种程度上来说，一个软件在任何时候都需要重构，但是完美的软件是不存在的，你必须首先得承认这一点，然后才能开发出成功的软件。过度开发不仅浪费了时间和金钱，还造成了代码过度复杂臃肿，产生很多无用的接口和配置，为了增加一个小小的功能必须得大动干戈，给维护造成很大的麻烦，而且可配置的功能太多往往造成使用上的不方便。

在软件开发中，我们经常使用到一条原则叫**Pareto原则** [26]（也称为 **80-20 原则**），这

[26] Pareto 法则最初在 1906 年被 Vilfredo Pareto 发现，当时是为了说明经济现象：意大利 80%的土地掌握在 20%的人手里。这条原则被广泛地使用到各个领域，例如在商业上：80%的销售来自 20%的客户。

条原则在各个领域都经常被使用到，大致是：

> **20%的工作需要花费 80%的精力。**

为了避免过度开发，我们在这里也可以应用该原则：

> **你的开发可以让 80%的常用功能非常容易实现，而剩下 20%的功能可以实现但需要花费较多的精力。**

这样，你就可以没必要为那些不常用的功能和需求而花费大量的精力，导致过度的设计。使用 XP（极限开发）可以由浅入深地开发需求，一定程度上避免过度的开发，但是经验并不丰富的开发人员开发过程中往往造成了过于简单和粗糙的设计，而不易扩展和完成必要的功能。想既避免过度开发，又能设计出扩展性良好的软件，绝非一朝一夕就能够做到的，Eric Evans 撰写了一本书籍《Domain-Driven Design: Tackling Complexity in the Heart of Software》，详细地讲述了如何使用 DDD 解决复杂领域问题，使用 DDD 开发能够在开发中平衡这二者，有兴趣的读者可以阅读此书以作更深入的了解。

16.10总结

客观世界要反映在计算机里，需要我们为其归纳出合理的模型，为问题建模是软件开发最关心的问题。如果建立的模型本身有问题，哪怕你是设计模式的专家，也不能够开发出有用的软件。在本节，我们讨论了关于 OOP 设计应该注意的一些问题，以上提到的所有过程都是不停迭代的过程，它们之间并没有绝对的先后顺序。**软件设计的关键是建立能够表达丰富功能需求的领域模型。**

真的 OOP 开发设计高手要有优秀的抽象概括能力，不断地汲取领域知识，选择合适的设计模式，甚至创造你所需要的模式，为问题领域迭代提炼出完善的模型，结合相关的实现技术，才能开发出成功的软件。

第17章　结语

17.1 概述

笔者认为，尽管面向对象的开发早已不再新鲜了，但是很多人仍然倾向于使用面向过程的开发，这样，虽然使用了 OOP 语言开发设计，却没有享受到面向对象所带来的简单性，高质量，可维护性，易扩展性，高性能，可重用性，可伸缩性等方面的好处。笔者撰写此书的目的就是试图通过一些常用的设计模式和设计原则来介绍如何使用面向对象视角来分析和设计问题，分享它给我们带来的设计上的好处。

笔者发现，如果站在为了使用设计模式而学习设计模式的角度考虑问题，不能给你的设计带来如愿以偿的效果，我们要学会使用 OOP 的眼光分析这些模式，才能明白这些模式的本质，才能明白有经验的 OOP 开发者如何使用 OOP 的眼光开发软件。本书重点结合面向对象开发范式向读者介绍这些模式，中间穿插一些软件设计原则，这些原则不一定只是关于 OOP 的，但是对软件设计至关重要的。在本书的第 15 章，笔者讲述了 OOP 开发设计的缺陷，即也是 AOP 能为之发展的最主要原因。下面我们对这些模式做一下简单的回顾。

17.2 面向对象的开发范式

对象是包含了方法和数据的结构体，这给开发者带来了新的视角去分析问题：

- 面向对象的最大好处就是封装，让我们再回顾一下这些设计模式都封装了哪些方面：

创建型模式	创建型模式封装了实例化的过程，客户对象不再关心这些创建的细节：应该使用哪个具体类，如何初始化和组装实例。这样，也为实例化提供了很大的灵活性：可以使用使用克隆的方式加快创建过程和简化创建细节，也可以根据配置确定实例化的具体类型，还可以根据需要创建单例对象。
模板方法模式	封装了那些不变的算法步骤，把变化那些部分交给子类去封装，对使用者隐藏了具体的子类型。
装饰器模式	装饰器模式隐藏了被装饰对象；并且，由于装饰对象和被

	装饰的对象具有相同的接口，客户对象在使用被装饰过的对象和未装饰过的对象时，不需要对它们区别对待，隐藏了装饰器类型。
代理模式	和装饰器模式一样，隐藏了被代理对象（目标对象），也实现了代理类型的隐藏。
适配器模式	隐藏了被适配的接口/类，客户对象并不知道请求会转发给被适配的对象。
外观模式	隐藏了子系统，封装了外观和子系统之间的复杂交互。
组合模式	实现了叶子类和分支子类的隐藏，客户对象操作叶子对象和分支对象时不需要区别对待。
策略模式	每一具体策略都封装了一个具体的实现算法。
状态模式	每一状态都封装了与一个特定状态相关的行为，Context 隐藏了状态接口和实现，客户对象不知道它们的存在。
观察者模式	不同的观察者对变化的处理是不同的，把这些变化封装在不同类型的观察者类型里，由于它们有相同的接口，观察者就能独立于主题而变化。另外如果为主题也抽象了接口，这样观察者和主题两方面就能独立变化与重用，而不会影响对方。

- 对象模块化的粒度超过了单个的方法或者结构体，这样，我们更容易把系统分割成易于处理的抽象单元，我们也可以重用和组合这些功能更强的单元，并隐藏更多的细节，减少程序之间的耦合。
- 多态可以让我们关心在什么时间去做，而不去关心如何去做，因为如何去做可以在后续开发中扩展。多态强调以相同的方式处理不同类型的对象，而这些对象有什么样的行为却在运行时决定。
- OOP 强调面向接口和抽象的编程，隐藏了具体的实现，减少了耦合。

17.3 一些原则

我们这里简要总结一下前面讨论过的一些软件设计原则：

DRY原则	尽量避免重复，如果你能在你的代码里发现重复，尽快的重构它。
好莱坞原则/控制反转	这个原则让我们可以很轻松的把可扩展的那部分代码插入到控制程序中去，以完成一个完整的流程，这样，我们就更专注于需要扩展的那部分逻辑开发了。
OCP原则	软件实体要对修改关闭，对扩展开放，软件开发发展至今，在这方面取得了一些成就。
最少知识原则	知道的最少，耦合性最弱。
封装变化	封装那些变化的部分供客户化，让使用者依赖于抽象而非具体，在不影响使用者的前提下扩展系统功能。
优先使用合成而非继承	尽管继承是一个非常好的重用父类功能、扩展父类的功能的方法，但是我们尽量使用合成，有时候二者结合。
Pareto原则	这条原则不仅仅是我上述讲述的几种表达方式，可以广泛运用到软件开发的其他方面。

17.4 写在模式之后

很久之前，和朋友去一家快餐店吃饭，我点了一个汉堡，一包薯条和几包番茄酱，我留下了一包番茄酱，把其他所有番茄酱都挤到汉堡中了，我的朋友瞪大眼睛看着我，然后一本正经地对我说，番茄酱是用来吃薯条的，不是吃汉堡的。我顿时哑然失语。

模式不是你把它们强加于你的设计，你的设计就会变得优雅而成功了；最重要的事情是了解你的领域问题本身，你如果能够清楚认识你的问题本质而非臆想它们时，你就能在设计时候自然而然地选择它们，使用它们。当你需要番茄酱作为调味剂的任何场合，只要你需要它，你就可以添加它，不会再有“番茄酱是用来吃薯条”的束缚。

我们要避免刻意地为使用模式而拼装设计，这不会带来优雅的设计。而是要站在你的问题上，使用 OOP 眼光，分析你所遇到问题的本质，只要到那一步，不管问题有多复杂，我们总能够找到需要的模式（不管是别人的还是自己创造的）把我们领入设计的道路。也许那时，你就真正地懂得了 OOP 设计，至于设计模式，也就没有那么重要了。

我相信，读者通过这本书的介绍，得到的不仅仅是了解模式，而是学会使用 OOP 的眼光分析问题，解决问题。

第六篇 附录

A. 本书推荐

本书是一本主要是关于设计模式介绍性和实践性的书籍，希望本书能给读者对面向对象的开发带来启发和更有价值的思考。在本书的写作过程中，作者参考了大量书籍，其中很多书籍值得一读，希望能给有这方面需要的读者给出建议。最后，作者给出一些一些网站和论坛，希望读者能够实时关注有关开发的新话题。

Java语言相关学习的书籍

- Bruce Eckel. Thinking in Java, 3rd Edition. Prentice-Hall, December 2002

  这本书籍已经出第四版了，新版本书里介绍了 Java 5 的一些新的语法。它是公认的 Java 语言学习的权威书籍，不仅讲述了 Java 语言的语法，还涵盖了许多面向对象的思想，如果你想学习使用 Java 语言进行面向对象的编程与设计，此书非常值得一读。

- Ron Hitchens. Java NIO. O'Reilly, 2002

  本书讲述了 Java NIO 的编程技术（特别是网络 IO 的编程）。

- James Gosling, Bill Joy, Guy Steele and Gilad Bracha. The Java™ Language Specification Third Edition. ADDISON-WESLEY, 2005.

  这是一本介绍 Java 语言规范的书籍，大多数语法介绍书籍不会全面介绍 JVM 如何加载和初始化类，线程等一些相关技术和概念，这些都可以在此书中找到，并且它是 Java 语言的官方书籍，权威性不言而喻。

- Joshua Bloch. Effective Java, Second Edition. Addison-Wesley, 2008.

  如果你已经使用 Java 开发好几年了，但是你还想知道那些专业的 Java 开发人员如何编写高效的代码的，这本书绝对值得一读。

- David Flanagan and Brett McLaughlin. Java 1.5 Tiger: A Developer's Notebook. O'Reilly, 2004.

  如果你不熟悉 Java 5 的新语法，可以参考本书和《Think in Java》第四版相关章节。

J2EE技术相关书籍

- Rod Johnson. Expert One-on-One J2EE Design and Development. Wiley Publishing, Inc, 2003.

  这本书籍介绍了一些 J2EE 的常用技术，深入探讨了 J2EE 编程中经常出现的问题和风险，帮助读者创建高效的 J2EE 应用。

- Rod Johnson and Juergen Hoeller. Expert One-on-One J2EE Development without EJB. Wiley Publishing, Inc, 2003.

  这本书籍可以说是上面书籍的续篇，它颠覆了一些传统的 J2EE 观点，审视了 EJB 所带来巨大复杂性。现在阅读本书可能当时那么震撼，因为读者对不使用 EJB 来创建 J2EE 应用已经习以为常了，很多 Java 架构采用 SSH（Struts+Spring+Hibernate）等技术创建应用。但它详细讲述了 Spring 核心框架的实现技术，对正在使用 Spring 框架或者对其实现技术感兴趣的读者，此书值得一读。

- Deepak Alur, John Crupi and Dan Malks. Core J2EE Patterns: Best Practices and Design Strategies, Second Edition. Prentice Hall PTR, 2003.

  这本书籍主要讲述了一些非常重要的 J2EE 模式，J2EE 架构师和开发人员值得一读。

面向对象设计相关书籍

- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

  这是影响最大的设计模式的经典书籍，读者在使用相关模式时，都可以拿来翻一翻。

- Andrew Hunt and David Thomas. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, 1999.

  这本书讲述了如何高效开发出高质量软件的方法，讲述过程中穿插了很多寓言故事，深入浅出，是一本有经验的软件开发人员继续“修炼”的哲学书籍。

- Alan Shalloway and James R. Trott. Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd Edition. Addison-Wesley, 2004.

这本书籍从面向对象的视角分析设计模式，是一本学习模式的好书籍。

- Eric T Freeman, Elisabeth Robson, Bert Bates and Kathy Sierra. Head First Design Patterns. O'Reilly Media, October 2004.

  这本书籍非常适合初学者学习设计模式，由于使用了 Head First 的写作风格，通俗易懂。

- Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

  这本书籍讲述了如何重构你的代码，重构代码是一个复杂的过程，很容易引起各种各样的问题，这本书籍教你重构的整个过程，书写风格也十分流畅，非常易于阅读。

- Martin Fowler. Analysis Patterns: Reusable Object Models. Addison-Wesley, 1997.

  做金融开发的人员都有必要阅读这本书籍，在医药领域，金融领域，测量领域，贸易等领域使用书中所提到的分析模式建模有莫大的帮助，当然这些分析模式不局限于这些领域。在此书，Martin Fowler 把自己丰富的对象建模经验与读者分享，如果你想为复杂领域建模，但是没有足够把握，强烈推荐你学习此书。

- Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, August, 2003

  解决复杂领域问题的关键是有精炼的模型，这本书籍讲解了如何使用领域驱动设计迅速提炼有用的模型，本人强烈推荐此书。

给Agile（敏捷）开发人员推荐的书籍

- Kent Beck and Cynthia Andres. Extreme Programming Explained: Embrace Change. Addison-Wesley, 2004.

  这是敏捷开发人员必读的一本书籍，它为大家消除了很多开发上的错误观念。

- Henrik Kniberg. Scrum and XP from the Trenches (Enterprise Software Development). Lulu.com, 2007.

  这本书是一本非常浅显易读的Scrum书籍，作者把一年来实施Scrum过程和经验进行分享，没有高深的理论，只有故事和实践。这本书的电子版本在InfoQ网站上有下载：http://www.infoq.com/minibooks/scrum-xp-from-the-trenches。

这里给出的英文书籍都是英文原版，由于有些书籍的中文翻译版本较多，在这里就

不会一一列举，读者有需要可以购买相应的中文译本。

网站和论坛

读者可以从以下网站了解一些 J2EE 前言技术和话题：

- http://www.theserverside.com/：这个大家最熟悉不过了，是讨论J2EE技术的最大社区，不用多做介绍了。
- http://www.infoq.com/：企业级软件开发的一个很活跃的社区之一，这里讨论一些最流行开发技术和方法，包括SOA，Agile，Rubby等等。此网站有很多软件开发名人视频采访。它的中文站点为 http://www.infoq.com/cn/，其上不仅翻译英文网上的文章，还有一些国内软件开发牛人的一些采访视频和文章，还制作有免费的电子期刊，值得软件开发爱好者收藏浏览。
- http://www.javaeye.com/：是国内最大的Java社区，但是里面也对Rubby，Python等语言进行讨论和交流版块，这是能在国内看到的为数不多的比较专业的技术网站之一。
- http://picocontainer.org/：Pico Container官方网站。
- http://code.google.com/p/google-guice/：Guice官方网站。
- http://www.springsource.org/：Spring的门户网站。
- http://www.hibernate.org/：Hibernate的官方网站。
- http://easymock.org/：EasyMock组织的网站。
- http://www.opensymphony.com/xwork/：xwork框架的官方网站。
- http://www.opensymphony.com/webwork/：webwork框架的官方网站。
- http://struts.apache.org/：struts框架的官方网站。

B. 本书参考

[1] Gregor Kiczales, John Lamping, Anurag Mendhekar et al. Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, 1997.

[2] Aspect-Oriented Software Development Community & Conference. Web site: http://www.aosd.net

[3] Ramnivas Laddad. AspectJ in Action: Practical Aspect-Oriented Programming. Manning, 2003.

[4] Hafedh Mili, Amel Elkharraz and Hamid Mcheick.Understanding separation of concerns. In Proceedings of the 3rd Workshop on Early Aspects, 3rd International Conference on Aspect-OrientedSoftware Development. Lancaster, 2004.

[5] Eduardo Kessler Piveta and Luiz Carlos Zancanella. Observer Pattern usingAspect-Oriented Programming. In Proceeds of the 3rd Latin American Conferenceon Pattern Languages of Programming. Porto de Galinhas, PE, Brazil, August 2003.

[6] Bill Burke and Adrian Brock. Aspect-Oriented Programming and JBoss. Web site: http://onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html?page=1. May 2003.

[7] Bruce Eckel. Thinking in Java, 3rd Edition. Prentice-Hall, December 2002.

[8] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[9] Alan Shalloway and James R. Trott. Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd Edition. Addison-Wesley, 2004.

[10] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, August, 2003.

[11] Joshua Bloch. Effective Java: Programming Language Guide. Addison-Wesley, 2001.

[12] Andrew Hunt and David Thomas. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, 1999.

[13] Patrick Lightbody and Jason Carreira. WebWork in Action. Manning, 2006.

[14] Rod Johnson. Expert One-on-One J2EE Design and Development. Wiley Publishing, Inc, 2003.

[15] Rod Johnson and Juergen Hoeller. Expert One-on-One J2EE Development without EJB. Wiley Publishing, Inc, 2003.

[16] Martin Fowler. Analysis Patterns: Reusable Object Models. Addison-Wesley, 1997.

[17] Eric T Freeman, Elisabeth Robson, Bert Bates and Kathy Sierra. Head First Design Patterns. O'Reilly Media, October 2004.

[18] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. Department of Computer Science, June/July 1988, Volume 1, Number 2, pages 22-35.

[19] Christopher Alexander, Sara Ishikawa and Murray Silverstein. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, 1977.

[20] Alexander Christopher. The Timeless Way of Building. Oxford University Press, 1979.

[21] Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[22] Richard Monson-Haefel. Enterprise JavaBeans, Second Edition. O'Reilly, 2001.

[23] Ron Hitchens. Java NIO. O'Reilly, 2002.

[24] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. The Java™ Language Specification Third Edition. Addison-Wesley, 2005.

[25] Joshua Bloch. Effective Java, Second Edition. Addison-Wesley, 2008.

[26] Deepak Alur, John Crupi and Dan Malks. Core J2EE Patterns: Best Practices and Design Strategies, Second Edition. Prentice Hall PTR, 2003.

[27] Kent Beck and Cynthia Andres. Extreme Programming Explained: Embrace Change. Addison-Wesley, 2004.

[28] Martin Fowler. TechnicalDebt. Web site: http://martinfowler.com/bliki/TechnicalDebt.html. 2004.

[29] Martin Fowler. InversionOfControl. Web site: http://martinfowler.com/bliki/InversionOfControl.html. 2005.

[30] Martin Fowler. Inversion of Control Containers and the Dependency Injection

pattern. Web site: http://martinfowler.com/articles/injection.html. 2004.

[31] Vikas Hazrati. Dissecting Technical Debt. Web site: http://www.infoq.com/news/2009/10/dissecting-technical-debt. Oct, 2009.

[32] Dirk Riehle. Framework Design: A Role Modeling Approach. Web site: http://dirkriehle.com/computer-science/research/dissertation/index.html. 2000.

[33] Spring Framework. Web Site: http://www.springsource.org/.

[34] Guice. Web Site: http://code.google.com/p/google-guice/.

[35] Pico Container. Web Site: http://picocontainer.org/.

[36] Hiberante. Web Site: http://www.hibernate.org/.

[37] CGLib. Web Site: http://cglib.sourceforge.net/.

[38] XWork. Web Site: http://www.opensymphony.com/xwork/.

[39] Avalon. Web Site: http://avalon.apache.org/.

[40] Struts. Web Site: http://struts.apache.org/.

[41] Webwork. Web Site: http://www.opensymphony.com/webwork/.

[42] EasyMock. Web Site: http://easymock.org/.

[43] CGLib. Web Site: http://cglib.sourceforge.net/