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。

1.3 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。

1.4 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”。

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。

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。

1.5 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。

1.6 学习设计模式的一些常见问题

1. **模式有没有标准形式？**

   GoF为了描述每一种设计模式，给它们定义了四个基本要素，旨在模式使模式能够得到广泛传播，这并不代表GoF给出的模式就是标准模式。事实上，他们也没有给每一种模式给出标准，而是很多地方给出了它们可能的变化，比如像组合（Composite）模式，有透明的组合模式，也有安全的组合模式，还例如观察者（Observer）模式，有push和pull两种形式。

   这些设计模式只是为常见问题给出经验人士的一个参考方案，避免我们重复的发明轮子，而没有血统之分（哪个标准，哪个不标准），所以这个问题是个伪命题。

2. **设计模式和 OO 什么关系？**

   设计模式最初引入的时候，正处于面向对象设计高速发展时期，那些模式都是来自成功的面向对象系统的某些部分，所以这 23 个最基本的模式都是关于面向对象的，但这不表示只有面向对象才有模式。我们知道，模式是是某一上下文环境中一个问题的常用解决方案，那么，SOA（Service-Oriented Architecture）可以有模式，Agile 也可以有模式，OSGI（Open Services Gateway initiative）也有模式。

3. **能不能创造新的设计模式？**

   完全可以，只要你能给一个上下文的问题给出一个通用的解决方案，并能使用 GoF 的四个基本要素描述它们。但是要避免重复的发明轮子，你必须了解你创造的是否已经存在，或者是否是已存在模式的变种。

4. **学习了设计模式，就等于学会了设计？**

   学习设计模式只是借鉴 OO 专家的成功经验，要学会设计，还得向他们一样，学会使用 OO 的眼光看待问题，解决问题。这样，在解决问题的过程中，模式就会手到擒来，并且自然地变化它们以适应你的问题本身。

5. **设计模式是 OO 设计的根本吗？**

   解决问题才是根本，模式只是关于解决问题的经验总结。笔者最初学习设计模

式之后，以为优雅的设计就是尽可能的使用设计模式，所以在解决问题是刻意的套用它们，出现了一些拙劣的设计。后来读取了 Eric Evans 的《Domain-Driven Design: Tackling Complexity in the Heart of Software》一书，才意识到为领域建模的重要性，为解决问题，要为你的领域问题建立合理的模型，既然 OO 能够为软件编程带来巨大的变化，我们就要学会使用 OO 的眼光分析问题，享受 OO 给编程带来的莫大好处。

于是紧接着，在后来的一个软件开发中，我开始从问题本质入手，“忘却”模式，为问题提取模型，等设计开发完成之后，发现，我已经自然而然的使用了组合（Composite）模式和解释器（Interpreter）等模式。至此，我才意识到使用 OO 眼光分析问题的重要性，所以，这本书籍并不是单纯介绍模式的书籍，而是和大家一起探讨 OOP，分析 OOP 给编程带来的好处，希望阅读完本书的人都能开始使用 OO 的眼光分析问题。

6. 软件的核心是什么？

软件的核心是模型，为复杂领域问题提取精炼的模型是根本。我们要学会使用 OO 这把利器，借助它的眼光来分析问题解决问题，这样才能做出客户满意的软件。

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：

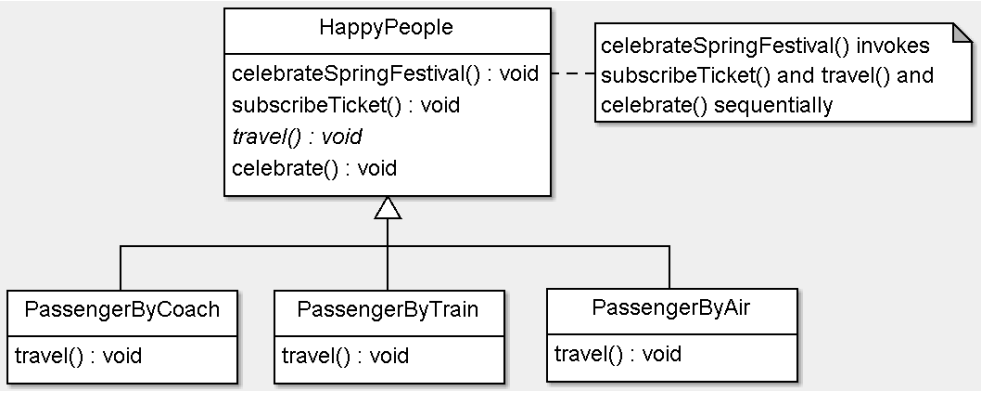
- 在重复代码中应用相同修改：为了增强功能或者修改错误（Bug），对于某一处代码的进行改动，在其他重复地方也可能需要修改。
- 开发成本增加：新的问题可能和旧问题非常相近，但是由于一些小细节的不同，导致了开发了不同的不能重用的代码。
- 不利于测试：如果相同的代码写在不同类里，导致测试代码的重复开发。
- 不利于阅读和维护：随着不断的开发维护，相同功能的实现在不同地方会变得不尽相同，有时甚至出现相同的需求实现逻辑却是不相同的，由于实现的不同，导致这些代码的健壮性也差异很大，很难维护这些代码。
- 代码重复还会引起性能等诸多问题，比如产生了很多重复的对象。

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）

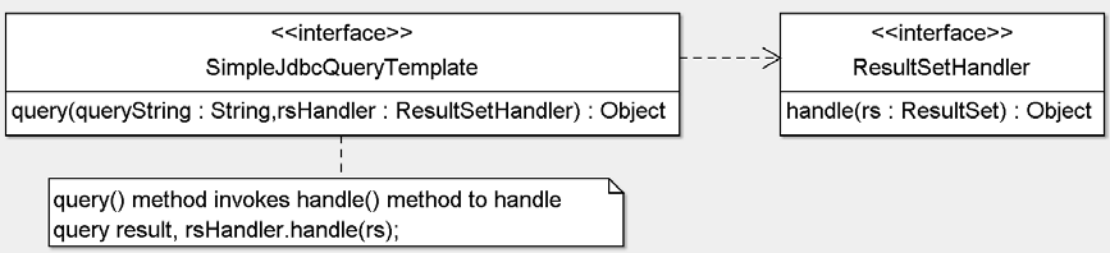
模板方法模式的应用很广泛，但过分地使用模板方法（Template Method）模式，往往会引起子类的泛滥。比如，我们有如下需求：查询数据库里的记录。首先我们需要得到数据库连接 Connection 对象，然后创建 Statement 实例并执行相关的查询语句，最后处理查询出来的结果并在整个执行过程中处理异常。研究这些步骤，不难发现，整个过程中第一步，第二步以及异常处理的逻辑对于每次查询来说，都是相同的，发生变化的部分主要是在对查询结果的处理上。据上分析，模板方法模式很适合处理这个问题，我们只需要抽象出这个处理查询结果的方法供不同的子类去延迟实现即可。

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

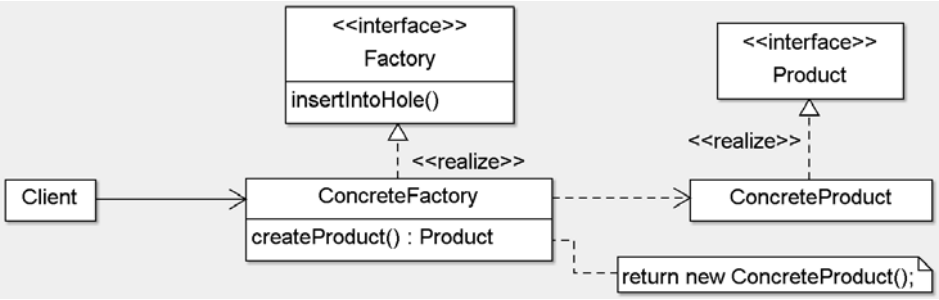
GoF 为工厂方法模式给出的定义如下：

> **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。

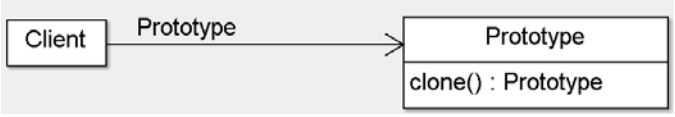
由于很多 OOP 语言都支持对象的克隆（拷贝）以方便复制对象，但这些方式并不那么完美，后述我们将会讨论。

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。

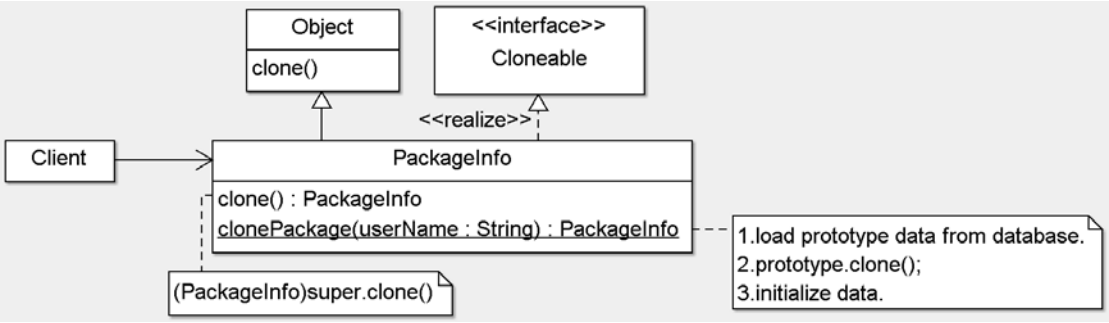
“和上次差不多一样，只是邮寄给另外一个地址，这里是邮寄地址……”顾客一边说一边把写有邮寄地址的纸条给你。

“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：

7. 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。

8. 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

6.2　使用工厂方法模式的问题

6.3　Inversion of Control（控制反转，IoC）

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

6.3.2　Service Locator（服务定位器）

6.3.3　Dependency Injection

6.3.3.1　Setter注入

6.3.3.2　Constructor注入

6.3.3.3　Annotation注入

6.3.3.4　Interface注入

6.3.3.5　Parameter注入

6.3.3.6　其他形式的注入

6.4　总结

第三篇 构建复杂结构

第7章　装饰器（Decorator）模式

第8章　代理（Proxy）模式

8.1　概述

8.2　代理（Proxy）模式

8.2.1　类图

8.2.2　访问分布式对象

8.3　J2SE动态代理

8.3.1　简介

8.3.2　类和接口

8.3.3　调用原理

8.3.4　实现同步

8.3.5　总结

8.4　代理（Proxy）模式与装饰器（Decorator）模式

8.5　总结

第9章　适配器（Adapter）模式

9.1　概述

9.2　打桩

9.3　其他适配器模式

9.3.1　类适配器

9.3.2　双向适配器

9.4　测试

9.5　适配器（Adapter）模式与代理（Proxy）模式

第10章　外观（Facade）模式

第11章　组合（Composite）模式

第四篇 行为模式

第12章　策略（Strategy）模式

第13章　状态（State）模式

13.1 电子颜料板

13.2 **switch-case**实现

13.3 如何封装变化

13.4 状态模式

13.5 使用**enum**类型

13.6 与策略（**Strategy**）模式的比较

第14章　观察者（Observer）模式

第五篇 终点还是起点

第15章　面向切面的编程（AOP）

第16章　面向对象开发

16.1 概述

16.2 写在面向对象设计之前

16.3 汲取知识

16.4 横看成岭侧成峰

16.5 提炼模型

16.6 应用设计模式

16.7 不能脱离实现技术

16.8 重构

16.9 过度的开发（**Over-engineering**）

16.10总结

第17章　结语

17.1 概述

17.2 面向对象的开发范式

17.3 一些原则

17.4 写在模式之后

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://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/