译者序

提起设计模式，GoF 的《设计模式：可复用面向对象软件的基础》一书可谓是设计模式世界的的“圣经”，几乎无人不知，无人不晓。不过，一来该书实际上源自 4 位作者的博士论文，学术性较强，初学者很难透彻理解书中内容。二来，虽说设计模式只是设计思想，不依赖于任何编程语言，但是各种编程语言的特性终究是不同的，而该书中的示例代码又是基于 C++ 和 Smalltalk 的。因此，对于使用 Java 语言编程的开发者来说，当然还是最希望能够阅读通过 Java 语言的示例代码来讲解设计模式的图书。

本书是结城浩先生除《程序员的数学》《图解密码技术（第 3 版）》《数学女孩》系列之外的又一力作，初版于 2001 年 6 月发行。当时，日本还没有通俗易懂地讲解设计模式的图书。就这一点而言，本书堪称日本第一。许多日本 IT 工程师在攻读硕士和博士学位时都学习过本书。如今，15 年过去了，本书历经多次重印，仍位居销售排行榜前列，足见其在日本 IT 类图书中的地位。

当然，在这 15 年间，IT 界也发生了翻天覆地的变化，各种开源框架层出不穷，机器学习大兴其道。但是，在面向对象编程中，设计模式的重要性却不曾改变。与以前一样，在大规模的企业系统开发中，Java 和 C# 仍处于主导地位。在这种大规模系统的开发中，设计模式可以帮助我们实现系统结构化，很好地支撑起系统的稳定性和可扩展性。而本书内容经典，时至今日仍然适用，作为设计模式的入门图书，非常适合于初学设计模式的开发者。

本书特色如下：

- 讲解了 23 种设计模式

本书对 GoF 书中的 23 种设计模式全部进行了讲解。通过了解这些模式，我们可以知道在哪些情况下应当使用哪种设计模式。在编程时，如果能够预测到系统中的某处可能发生什么样的变化，然后提前在系统中使用合适的设计模式，就可以帮助我们以最少量的修改来应对需求变更。设计模式是由前人的知识和经验浓缩而成的，是帮助我们快速提高开发水平的捷径。

- 讲解了对接口的理解

接口的使用方法是 Java 等面向对象编程语言的重要部分，只是满足于知道接口的基本语法是不行的。本书可以帮助我们加深对接口的重要性和使用方法的理解。

- 讲解了可复用代码的写法

需求变更是令所有开发者都会感到头疼的问题。当发生需求变更时，我们总是希望需要修改的代码能尽量集中在一起，不想大范围地修改代码。另外，我们也经常希望在新系统中沿用之前已经测试过的代码。本书就将教我们如何编写可复用的代码。

不过，设计模式是一把双刃剑。正确地使用它可以提高系统的适应性，误用则会反过来降低系统的适应性。下面的学习方法有助于我们尽快地掌握设计模式：

1. 了解设计模式

首先通过阅读图书和文章了解设计模式。除了阅读本书以外，还可以参考本书附录中介绍的许多讲解和讨论设计模式的优秀图书和文章。

2. 动手体验设计模式

自己动手编写示例代码，观察代码运行结果。在这个过程中，注意用心去感受代码。

3. 在项目中实践

当认为时机成熟时，可以尝试在项目中运用设计模式。遇到阻力时，可以用书中的知识和自己的理解去说服其他开发人员和项目经理。

4. 总结经验教训

误用设计模式并不可怕，可怕的是一错再错。在每次误用设计模式后都应当总结经验教训，这样才能真正地提高对设计模式的理解。

5. 与其他开发者交流讨论

与其他开发人员，特别是与经验丰富的开发人员交流讨论是快速掌握设计模式的行之有效的方法之一。在讨论候选的几种设计模式到底哪种更好的过程中，时常会出现“一语惊醒梦中人”的情况。

在此衷心希望各位读者朋友们能够爱上设计模式。

杨文轩

2016 年 10 月

引言

大家好，我是结城浩。欢迎阅读《图解设计模式》。

想必大家在编写程序的时候，也曾遇到“咦，好像之前编写过类似的代码”这样的情况。随着开发经验的增加，大家都会在自己的脑海中积累起越来越多的“模式”，然后会将这些“模式”运用于下次开发中。

Eric Gamma、Richard Helm、Ralph Johnson、John Vlissides 等 4 人将开发人员的上述“体会”和“内在积累”整理成了“设计模式”。这 4 人被称为 the Gang of Four，简称 GoF。

GoF 为常用的 23 种模式赋予了“名字”，并按照类型对它们进行了整理，编写成了一本书，这本书就是《设计模式：可复用面向对象软件的基础》（请参见附录 E 中的 [GoF]）。

大家应当都知道，当多个模块组合在一起工作时，接口是非常重要的。其实，这条原则不仅仅适用于计算机，也适用于人。当多位开发人员一起工作的时候，“人”这个接口也非常重要，而这个接口的基础就是“语言”。特别是脱离具体代码、只讨论程序的大致结构时，语言和图示就显得尤其重要。比如，另外一位开发人员提出的改进方案与我的方案究竟是否相同？是不是大框架相同而细节不同呢？如果有无限的时间与耐力，这些问题都是可以通过反复讨论解答出来的。但是，如果借助设计模式的术语来表达想法，我们就可以更加轻松地比较两人的观点，进而使讨论进行得更加顺利。

设计模式为开发人员提供了有益且丰富的词汇，让开发人员可以更容易地理解对方所要表达的意思。

本书将对 GoF 的 23 种设计模式逐一进行讲解，让那些面向对象的初学者也可以很轻松地理解这些设计模式。本书并非仅仅给出枯燥的设计模式理论，还会用 Java 语言编写实现了设计模式的示例程序，并让程序真正地运行起来。我们学习设计模式，并不是为了遥远的将来而打算，而是将它当作是一种有益的技巧，因为它可以帮助我们从全新的角度审视我们每天所编写的代码，从而帮助我们开发出更易于复用和扩展的软件。

本书的特点

◆用 Java 语言编写可实际运行的程序

我们会编写 Java 程序代码来实现 GoF 的 23 种设计模式。为了方便大家通读这些代码，所有的代码都只有 100 行左右，非常精简。而且，所有的代码中都没有“以下代码省略”的部分，且都经过笔者自己编译并运行过。

◆模式名称的讲解

设计模式的名称原本不是汉语，而是英语。开发人员如果不精通英语，就无法由设计模式的名称直接联想到它的作用。因此，本书还会讲解各设计模式的名称是什么意思，以及怎样用汉语表达。这样一来，那些不擅长英语的开发人员也可以很轻松地掌握设计模式。

◆模式之间的关联与练习题

设计模式不需要死记硬背。要想掌握模式，必须要多练习，比如试着在阅读程序时识别出模式，在编写程序时运用模式。因此，必须了解模式之间的关联，并练习运用模式解决具体问题。本书为大家设计了用于学习设计模式的练习题和答案。

◆ Java 语言的相关信息

本书不仅会讲解设计模式，还会向读者展示一些信息以帮助大家深入理解 Java。带有Java符号的内容表示这部分是和 Java 语言相关的信息。

◆模式插图

如果只阅读文字讲解内容，很难掌握这些模式。在本书中，我们在每章的首页中都放了一张图片来直观地展示所要学习的模式，这样可以帮助大家更加轻松地掌握模式。

本书的读者

本书适合以下读者阅读。

- 对面向对象开发感兴趣的人
- 对设计模式感兴趣的人
  （特别是阅读了 GoF 的著作但是难以理解的人）
- 所有 Java 程序员
  （特别是对抽象类和接口的理解不充分的人）

阅读本书需要掌握 Java 语言的基本知识。具体而言，至少需要理解类和接口、字段和方法，并能够编译和运行 Java 源代码。

虽然本书讲解的是设计模式，但必要时也会对 Java 语言的功能进行补充说明，因此读者还可以在阅读本书的过程中加深对 Java 的理解。特别是对于那些对抽象类和接口的目的理解不充分的读者来说，本书具有很大的参考价值。

此外，即使不了解 Java 语言也没关系。如果了解 C++ 语言，同样可以轻松理解本书中的内容。

如果想从零开始学习 Java 语言，建议读者在阅读本书前，先阅读笔者的拙作《Java 语言编程教程（修订版）》[①]（请参见附录 E [Yuki03]）。

另外，建议学习完本书的读者再去学习一下《图解设计模式：多线程》[②]（请参见附录 E[Yuki02]）。

本书的结构

本书结构如下所示，各章基本上与 GoF 设计模式的章节相对应。但是笔者对设计模式的分类与 GoF 不同，因此章节划分也不尽相同。关于 GoF 对设计模式的分类，请参见附录 C。

- 在**第 1 部分“适应设计模式”**中，我们将学习一些比较容易理解的设计模式，并以此来适应

① 原书名为『改訂版　Java 言語プログラミングレッスン』，尚无中文版。——译者注

② 原书名为『Java 言語で学ぶデザインパターン入門　マルチスレッド編』，人民邮电出版社即将引进出版。——译者注

设计模式的概念。

- 在第 1 章“Iterator 模式——一个一个遍历”中，我们将要学习从含有多个元素的集合中将各个元素逐一取出来的 Iterator 模式。
- 在第 2 章“Adapter 模式——加个‘适配器’以便于复用”中，我们将要学习 Adapter 模式，它可以用来连接具有不同接口（API）的类。

- 在**第 2 部分“交给子类”**中，我们将学习与类的继承相关的设计模式。
  - 在第 3 章“Template Method 模式——将具体处理交给子类”中，我们将要学习在父类中定义处理框架，在子类中进行具体处理的 Template Method 模式。
  - 在第 4 章“Factory Method 模式——将实例的生成交给子类”中，我们将要学习在父类中定义生成接口的处理框架，在子类中进行具体处理的 Factory Method 模式。
- 在**第 3 部分“生成实例”**中，我们将学习与生成实例相关的设计模式。
  - 在第 5 章“Singleton 模式——只有一个实例”中，我们将要学习只允许生成一个实例的 Singleton 模式。
  - 在第 6 章“Prototype 模式——通过复制生成实例”中，我们将要学习复制原型接口并生成实例的 Prototype 模式。
  - 在第 7 章“Builder 模式——组装复杂的实例”中，我们将要学习通过各个阶段的处理以组装出复杂实例的 Builder 模式。
  - 在第 8 章“Abstract Factory 模式——将关联零件组装成产品”中，我们将要学习像在工厂中将各个零件组装成产品那样生成实例的 Abstract Factory 模式。
- 在**第 4 部分“分开考虑”**中，我们将学习分开考虑易变得杂乱无章的处理的设计模式。
  - 在第 9 章“Bridge 模式——将类的功能层次结构与实现层次结构分离”中，我们将要学习按照功能层次结构与实现层次结构把一个两种扩展（继承）混在一起的程序进行分离，并在它们之间搭建桥梁的 Bridge 模式。
  - 在第 10 章“Strategy 模式——整体地替换算法”中，我们将要学习 Strategy 模式，它可以帮助我们整体地替换算法，使我们可以更加轻松地改善算法。
- 在**第 5 部分“一致性”**中，我们将学习能够让两个看上去不同的对象的操作变得统一，以及在不改变处理方法的前提下增加功能的设计模式。另外，我们还要学习“委托”。
  - 在第 11 章“Composite 模式——容器与内容的一致性”中，我们将要学习让容器和内容具有一致性，从而构建递归结构的 Composite 模式。
  - 在第 12 章“Decorator 模式——装饰边框与被装饰物的一致性”中，我们将要学习让装饰边框与被装饰物具有一致性，并可以任意叠加装饰边框的 Decorator 模式。
- 在**第 6 部分“访问数据结构”**中，我们将学习能够漫步数据结构的设计模式。
  - 在第 13 章“Visitor 模式——访问数据结构并处理数据”中，我们将要学习在访问数据结构的同时重复套用相同操作的 Visitor 模式。
  - 在第 14 章“Chain of Responsibility 模式——推卸责任”中，我们将要学习可以处理连接在一起的多个对象中某个地方的 Chain of Responsibility 模式。
- 在**第 7 部分“简单化”**中，我们将学习可以让类关系简单的设计模式。
  - 在第 15 章“Facade 模式——简单窗口”中，我们将要学习 Facade 模式，该模式并不是单独地控制那些错综复杂地关联在一起的多个类，而是通过配置一个窗口类来改善系统整体的可操作性。

  - 在第 16 章“Mediator 模式——只有一个仲裁者”中，我们将要学习可以不与多个复杂的类打交道，而是准备一个窗口，然后通过与这个窗口打交道来简化程序的 Mediator 模式。
- 在**第 8 部分“管理状态”**中，我们将学习与状态相关的设计模式。
  - 在第 17 章“Observer 模式——发送状态变化通知”中，我们将要学习将状态发生变化的类和发送状态变化通知的类分开实现的 Observer 模式。
  - 在第 18 章“Memento 模式——保存对象状态”中，我们将要学习可以保存对象现在的状态，并可以根据情况撤销操作，将对象恢复到以前状态的 Memento 模式。
  - 在第 19 章“State 模式——用类表示状态”中，我们将要学习用类来表现状态，以减少 `switch` 语句的 State 模式。
- 在**第 9 部分“避免浪费”**中，我们将学习可以避免浪费、提高处理效率的设计模式。
  - 在第 20 章“Flyweight 模式——共享对象，避免浪费”中，我们将要学习当多个地方有重复对象时，通过共享对象来避免浪费的 Flyweight 模式。
  - 在第 21 章“Proxy 模式——只在必要时生成实例”中，我们将要学习除非必须“本人”处理，否则就只使用代理类来负责处理的 Proxy 模式。
- 在**第 10 部分“用类来表现”**中，我们将学习用类来表现特殊东西的设计模式。
  - 在第 22 章“Command 模式——命令也是类”中，我们将要学习用类来表现请求和命令的 Command 模式。
  - 在第 23 章“Interpreter 模式——语法规则也是类”中，我们将要学习用类来表现语法规则的 Interpreter 模式。

本书中的示例代码

示例代码的获取方法

本书的示例代码可以从以下网址下载（点击“随书下载”）：

```
http://www.ituring.com.cn/book/1811
```

详细信息请参见附录 B。

从 Main 类启动示例代码

在 Java 中，只要类中定义了以下方法，就可以将该类作为程序的起点：

```
public static void main(string[])
```

但是在本书中，为了使读者能够更容易理解代码，各章的示例程序都使用 `Main` 类作为程序的起点。

关于本书中术语的注意事项

接口和 API

接口这个术语有多个意思。

一般而言，在提到“某个类的接口”时，多是指该类所持有的方法的集合。当想要对该类进行某些操作时，需要调用这些方法。

但是在 Java 中，也将“使用关键字 `interface` 声明的代码”称为接口。

这两个“接口”的意思有些相似，在使用时容易混乱，因此本书中采用以下方式加以区分。

- 接口（API）：通常的意思（API 是 application programming interface 的缩写）
- 接口：使用关键字 `interface` 声明的代码

模式、类和角色

在本书中，**模式**这个词表示设计模式的意思。例如，“GoF 一共在书中整理了 23 种模式”指的就是“GoF 一共在书中整理了 23 种设计模式”。另外，我们会将名为 Memento 的设计模式简称为“Memento 模式”。

类是指 Java 中的类，即以 `class` 关键字定义的程序。例如，在书中会有“这段程序中定义的是 `Gamer` 类”这种描述；而“`Memento` 类”则是指在程序上用 `class Memento { ... }` 定义的代码。

角色是本书中特有的说法。它是指模式（设计模式）中出现的类、接口和实例在模式中所起的作用。例如，在书中会有“由 `Gamer` 类扮演 Originator 角色”这种描述。当然，也存在角色的名字与类和接口的名字不一致的情况。

此处的内容很繁琐，但是当大家阅读本书时，就会理解笔者在这里想要表达的意思了。

致谢

首先需要向整理出设计模式的 Eric Gamma、Richard Helm、Ralph Johnson、John Vlissides 这 4 人表示感谢。

然后，还要向阅读笔者拙作，包括图书、连载杂志和电子邮件杂志的读者们表示感谢。另外，还要向笔者 Web 主页上的朋友们表示感谢。

笔者在编写本书的原稿、程序以及图示的过程中，也同时将它们公布在了互联网上，以供大家评审。在互联网上招募的评审人员不限年龄、国籍、性别、住址、职业，所有交流都是通过电子邮件和网络进行的。在此，笔者要向参与本书评审的朋友们表示感谢，特别是对给予了我宝贵意见、改进方案，向我反馈错误以及一直鼓励我的以下各位表示我最真挚的感谢（按五十音图顺序排列）：

新真千惠、池田史子、石井胜、石田浩二、井芹义博、宇田川胜俊、川崎昌博、榊原知香子、砂生贵光、佐藤贵行、铃木健司、铃木信夫、竹井章、藤森知郎、前田恭男、前原正英、三宅喜义、谷内上智春、山城俊介。

此外，对其他参与了评审工作的人员也一并表示感谢。

另外，还要向软银出版股份有限公司的图书总编野泽喜美男和第一图书编辑部的松本香织表示感谢。当我们一起商量这本书的选题时，他们都表示“这一定会是一本好书”，这让我倍受鼓舞。

最后要感谢我最爱的妻子和两个儿子，以及总是精神满满地支持我的岳母大人。

结城浩

2001 年 3 月 于武藏野

写于“修订版”前

《图解设计模式》一书自 2001 年初版发行以来，承蒙各位读者的厚爱，在此再次向各位读者表达我最真挚的感谢。

在这次“修订版”中，笔者重新全面地审视了本书的内容和表述。在修订中，也参考了读者朋友们发送给我的无数反馈意见和建议，真心谢谢你们。希望本书也能在读者朋友的工作和学习中发挥些许作用。

结城浩

2004 年 6 月

关于本书官网

读者可以从以下网址获取本书的最新信息：

http://www.hyuki.com/dp

该网址是作者本人运营的网站之一。

问答 Web

请将您读完这本书后的感想及意见发送到以下网址。

http://www.ituring.com.cn/book/1811

关于 UML

UML

UML 是让系统可视化、让规格和设计文档化的表现方法，它是 Unified Modeling Language（统一建模语言）的简称。

本书使用 UML 来表现各种设计模式中类和接口的关系，所以我们在这里稍微了解一下 UML，以方便后面的阅读。但是请大家注意，在说明中我们使用的是 Java 语言的术语。例如讲解时我们会用 Java 中的“字段”（field）取代 UML 中的“属性”（attribute），用 Java 中的“方法”（method）取代 UML 中的“操作”（operation）。

UML 标准的内容非常多，本节只对书中使用到的 UML 内容进行讲解。如果想了解更多 UML 内容，请访问以下网站。UML 的规范书也可以从该网站下载。

- UML Resource Page

  `http://www.omg.org/uml/`

类图

UML 中的**类图**（Class Diagram）用于表示类、接口、实例等之间相互的静态关系。虽然名字叫作类图，但是图中并不仅仅只有类。

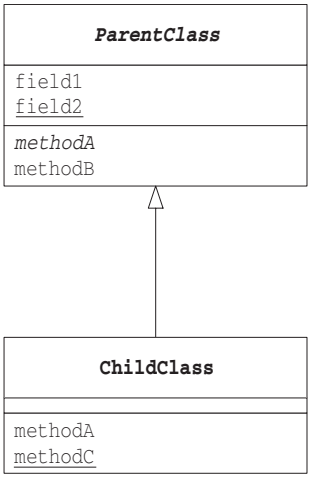
类与层次结构

图 0-1 展示了一段 Java 程序及其对应的类图。

图 0-1　展示类的层次关系的类图

```
abstract class ParentClass {
    int field1;
    static char field2;
    abstract void methodA();
    double methodB() {
        // ...
    }
}

class ChildClass extends ParentClass {
    void methodA() {
        // ...
    }
    static void methodC() {
        // ...
    }
}
```



该图展示了 ParentClass 和 ChildClass 两个类之间的关系，其中的空心箭头表明了两者之间的层次关系。箭头**由子类指向父类**，换言之，这是表示继承（extends）的箭头。

ParentClass 是 ChildClass 的父类，反过来说，ChildClass 是 ParentClass 的子类。父类也称为基类或超类，子类也称为派生类。

图中的长方形表示类，长方形内部被横线自上而下分为了如下 3 个区域。

- **类名**
- **字段名**
- **方法名**

有时，图中除了会写出类名、字段名和方法名等信息外，还会写出其他信息（可见性、方法的参数和类型等）。反之，有时图中也会省略所有不必要的项目（因此，我们无法确保一定可以根据类图生成源程序）。

abstract 类（抽象类）的名字以斜体方式显示。例如，在图 0-1 中 ParentClass 是抽象类，因此它的名字以斜体方式显示。

static 字段（静态字段）的名字带有下划线。例如，在图 0-1 中 field2 是静态字段，因此名字带有下划线。

abstract 方法（抽象方法）的名字以斜体方式显示。例如，在图 0-1 中 methodA 是抽象方法，因此它以斜体方式显示。

static 方法（静态方法）的名字以下划线显示。例如，在图 0-1 中 ChildClass 类的 methodC 是类的静态方法，因此它的名字带有下划线。

▶▶ 小知识：Java 术语与 C++ 术语

Java 术语跟 C++ 术语略有不同。Java 中的字段相当于 C++ 中的成员变量，而 Java 中的方法相当于 C++ 中的成员函数。

▶▶ 小知识：箭头的方向

UML 中规定的箭头方向是从子类指向父类。可能会有人认为子类是以父类为基础的，箭头从父类指向子类会更合理。

关于这一点，按照以下方法去理解有助于大家记住这条规则。在定义子类时需要通过 extends 关键字指定父类。因此，子类一定知道父类的定义，而反过来，父类并不知道子类的定义。只有在知道对方的信息时才能指向对方，因此箭头方向是从子类指向父类。

接口与实现

图 0-2 也是类图的示例。该图表示 PrintClass 类实现了 Printable 接口。为了强调接口与抽象类的相似性，本书的类图中会以斜体方式显示接口的名字。不过在其他书的类图中，接口名可能并非以斜体显示。空心箭头代表了接口与实现类的关系，箭头**从实现类指向接口**。换言之，这是表示实现（implements）的箭头。

UML 以 <<interface>> 表示 Java 的接口。

图 0-2 展示接口与实现类的类图

```
interface Printable {
    abstract void print();
    abstract void newPage();
}

class PrintClass implements Printable {
    void print() {
        // ...
    }
    void newPage() {
        // ...
    }
}
```

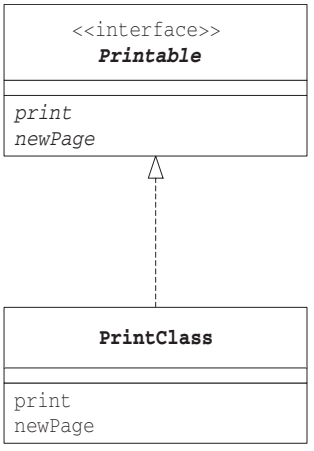
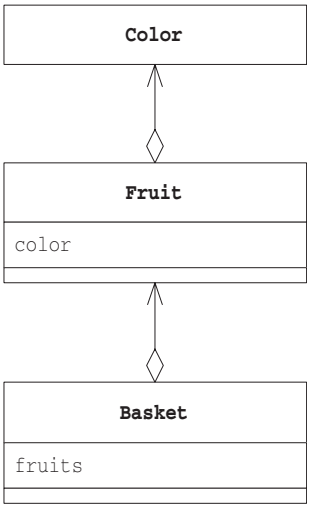


图 0-3 展示聚合关系的类图

```
class Color {
    // ...
}

class Fruit {
    Color color;
    // ...
}

class Basket {
    Fruit[] fruits;
    // ...
}
```



聚合

图 0-3 也是类图的示例。

该图展示了 Color（颜色）、Fruit（水果）、Basket（果篮）这 3 个类之间的关系。Basket 类中的 fruits 字段是可以存放 Fruit 类型数据的数组，在一个 Basket 类的实例中可以持有多个 Fruit 类的实例；Fruit 类中的 color 字段是 Color 类型，一个 Fruit 类实例中只能持有一个 Color 类的实例。通俗地说就是在篮子中可以放入多个水果，每个水果都有其自身的颜色。

我们将这种“持有”关系称为**聚合**（aggregation）。只要在一个类中持有另外一个类的实例——无论是一个还是多个——它们之间就是聚合关系。就程序上而言，无论是使用数组、java.util.Vector 或是其他实现方式，只要在一个类中持有另外一个类的实例，它们之间就是聚合关系。

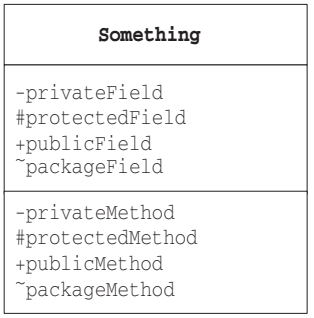
在 UML 中，我们使用带有空心菱形的实线表示聚合关系，因此可以进行联想记忆，将聚合关系想象为在空心菱形的器皿中装有其他物品。

可见性（访问控制）

图 0-4 也是类图的示例。

图 0-4　标识出了可见性的类图

```
class Something {
    private   int privateField;
    protected int protectedField;
    public    int publicField;
    int packageField;
    private   void privateMethod() {
    }
    protected void protectedMethod() {
    }
    public    void publicMethod() {
    }
    void packageMethod() {
    }
}
```



该图标识出了方法和字段的可见性。在 UML 中可以通过在方法名和字段名前面加上记号来表示可见性。

“+”表示 public 方法和字段，可以从类外部访问这些方法和字段。

“-”表示 private 方法和字段，无法从类外部访问这些方法和字段。

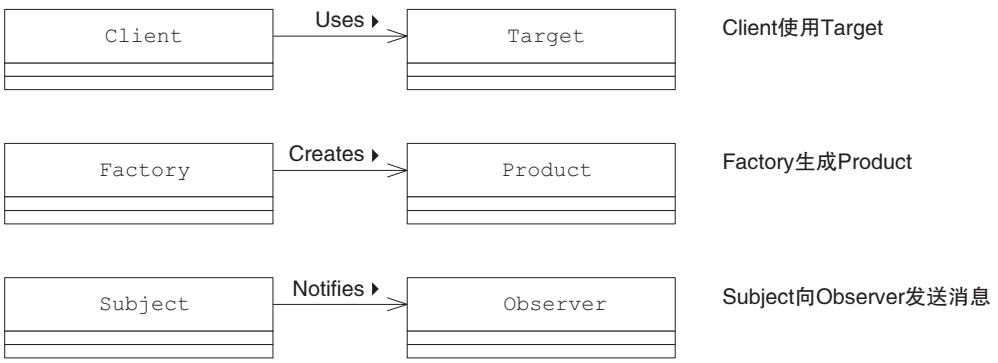
“#”表示 protect 方法和字段，能够访问这些方法和字段的只能是该类自身、该类的子类以及同一包中的类。

“~”表示只有同一包中的类才能访问的方法和字段。

类的关联

可以在类名前面加上黑三角表示类之间的关联关系，如图 0-5 所示。

图 0-5　类的关联



时序图

UML 的时序图（sequence diagram）用来表示程序在工作时其内部方法的调用顺序，以及事件的发生顺序。

类图中表示的是“不因时间流逝而发生变化的关系（静态关系）”，时序图则与之相反，表示的是“随时间发生变化的东西（动态行为）”。

处理流与对象间的协作

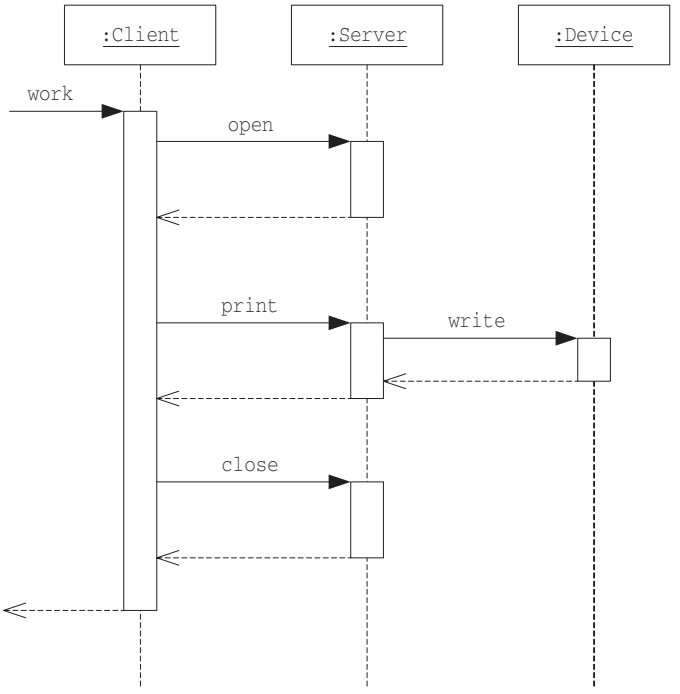
图 0-6 展示的是时序图的一个例子。

图 0-6　时序图示例（方法的调用）

```
class Client {
    Server server;
    void work() {
        server.open();
        server.print("Hello");
        server.close();
    }
    // ...
}

class Server {
    Device device;
    void open() {
        // ...
    }
    void print(String s) {
        device.write(s);
        // ...
    }
    void close() {
        // ...
    }
    // ...
}

class Device {
    void write(String s) {
        // ...
    }
}
```



在图 0-6 中，右侧是时序图示例，左侧是与之对应的代码片段。

该图中共有 3 个实例，如图中最上方的 3 个长方形所示。在长方形内部写有类名，类名跟在冒号（:）之后，并带有下划线，如 `:Client`、`:Server`、`:Device`，它们分别代表 `Client` 类、`Server` 类、`Device` 类的实例。

如果需要，还可以在冒号（:）之前表示出实例名，如 `server:Server`。

每个实例都带有一条向下延伸的虚线，我们称其为**生命线**。这里可以理解为时间从上向下流逝，上面是过去，下面是未来。生命线仅存在于实例的生命周期内。

在生命线上，有一些细长的长方形，它们表示实例处于某种活动中。

横方向上有许多箭头，请先看带有 `open` 字样的箭头。黑色实线箭头（—►）表示**方法的调用**，

这里表示 client 调用 server 的 open 方法。当 server 的 open 方法被调用后，server 实例处于活动中，因此在 open 箭头处画出了一个细长的长方形。

而在 open 箭头画出的长方形下方，还有一条指向 client 实例的虚线箭头（≪┈┈），它表示**返回 open 方法**。在上图中，我们画出了所有的返回箭头，但是有些时序图也会省略返回箭头。

由于程序控制已经返回至 client，所以表示 server 实例处于活动状态的长方形就此结束了。

接着，client 实例会调用 server 实例的 print 方法。不过这次不同的是在 print 方法中，server 会调用 device 实例的 write 方法。

这样，我们就将多个对象之间的行为用图示的方式展示出来了。**时序图的阅读顺序是沿着生命线从上至下阅读**。然后当遇到箭头时，我们可以顺着箭头所指的方向查看对象间的协作。

学习设计模式之前

在学习设计模式之前，我们先来了解几个小知识，以便更好地理解设计模式。

设计模式并非类库

为了方便地编写 Java 程序，我们会使用类库，但是设计模式并非类库。

与类库相比，设计模式是一个更为普遍的概念。类库是由程序组合而成的组件，而设计模式则用来表现内部组件是如何被组装的，以及每一个组件是如何通过相互关联来构成一个庞大系统的。

我们以白雪公主的故事为例来思考一下。在讲述故事梗概时，我们并不需要知道在演绎这个故事的电影中到底是谁扮演白雪公主、谁扮演王子。与介绍演员相比，讲述白雪公主与王子之间的“关系”更加重要。因为并非特定的演员扮演的“白雪公主”才是白雪公主，不论谁来扮演这个角色，只要是按照白雪公主的剧本进行演出，她们都是白雪公主。重要的是在这个故事中有哪些出场人物，他们之间是什么样的关系。

设计模式也是一样的。在回答“什么是 Abstract Factory 模式”时，阅读具体的示例代码有助于我们理解答案，但是并非只有这段特定的代码才是 Abstract Factory 模式。重要的是在这段代码中有哪些类和接口，它们之间是什么样的关系。

但是类库中使用了设计模式

设计模式并非类库，但是 Java 标准类库中使用了许多设计模式。掌握了设计模式可以帮助我们理解这些类库所扮演的角色。

典型的例子如下所示，在以后单独介绍各种设计模式的章节中，我们还会进一步学习。

- `java.util.Iterator` 是用于遍历元素集合的接口，这里使用了 Iterator 模式（第 1 章）
- `java.util.Observer` 是用于观察对象状态变化的接口，这里使用了 Observer 模式（第 17 章）
- 以下的方法中使用了 Factory Method 模式（第 4 章）
  `java.util.Calendar` 类的 `getInstance` 方法
  `java.secure.SecureRandom` 类的 `getInstance` 方法
  `java.text.NumberFormat` 类的 `getInstance` 方法
- `java.awt.Component` 和 `java.awt.Container` 这两个类中使用了 Composite 模式（第 11 章）

示例程序并非成品

设计模式的目标之一就是提高程序的可复用性；也就是说，设计模式考虑的是怎样才能将程序作为“组件”重复使用。因此，不应当将示例程序看作是成品，而应当将其作为扩展和变更的基础。

- **有哪些功能可以被扩展**
- **扩展功能时必须修改哪些地方**
- **有哪些类不需要修改**

从以上角度看待设计模式可以帮助我们加深对设计模式的理解。

不只是看图，还要理解图

本书以图解的方式讲解设计模式，其中主要使用的有类图和时序图（请参见前面“关于 UML”中的内容）。这些图并非只是简单的画，只瞥一眼是无法理解其中内容的。

在看类图时，首先看长方形（类），然后看它们里面的方法，并确认哪些是普通方法、哪些是抽象方法。接着确认类之间的箭头的指向，弄清究竟是哪个类实现了哪个接口。只有像这样循序渐进，一步一步对图中的内容刨根问底才能真正理解这幅图的主旨。

相比于类图，时序图理解起来更加容易一些。按照时间顺序自上而下一步一步确认哪个对象调用了哪个对象，就可以慢慢理解每个对象在模式中所扮演的角色。

只瞥一眼图是无法理解图中深藏的内容的，必须深入理解这些图。

自己思考案例

不要只是阅读书中的案例，还需要自己尝试着思考一些案例。

另外，我们还需要在自己进行设计和编程时思考一下学习过的设计模式是否适用于当前场景。

理解角色——谁扮演白雪公主

设计模式如同电影一样，类和接口这些“角色”之间进行各种各样的交互，共同演绎一部精彩的电影。在电影中，每个人都必须按照自己的角色做出相应的行为。主人公的行为必须像主人公，敌人则必须对抗主人公。女主角也会出场，将剧情推向高潮。

设计模式也一样。在每种设计模式中，类和接口被安排扮演各自的角色。各个类和接口如果不能理解自己所扮演的角色，就无法深入理解电影整体的剧情，无法扮演好自己的角色。这可能导致主人公屈服于敌方，或是女主角变成了坏人；又或是将喜剧演成了悲剧，将纪实片演成了虚构片。

在接下来的每章中，我们都将学习一种设计模式。同时我们也需要了解设计模式中出现的角色。请大家在阅读示例代码时，不要只盯着代码本身，而要将关注点转移到角色身上来，在阅读代码的时候，要思考各个类和接口到底在模式中扮演着什么角色。

如果模式相同，即使类名不同，它们所扮演的角色也是相同的。认清它们所扮演的角色有助于我们理解模式。这样，哪怕换了演员，我们依然可以正确地理解剧情。

如果我们现在看的是《白雪公主》，那么不论谁扮演白雪公主，王子都会爱上白雪公主。最后的结局也一定是白雪公主接受了王子的吻，苏醒了过来。

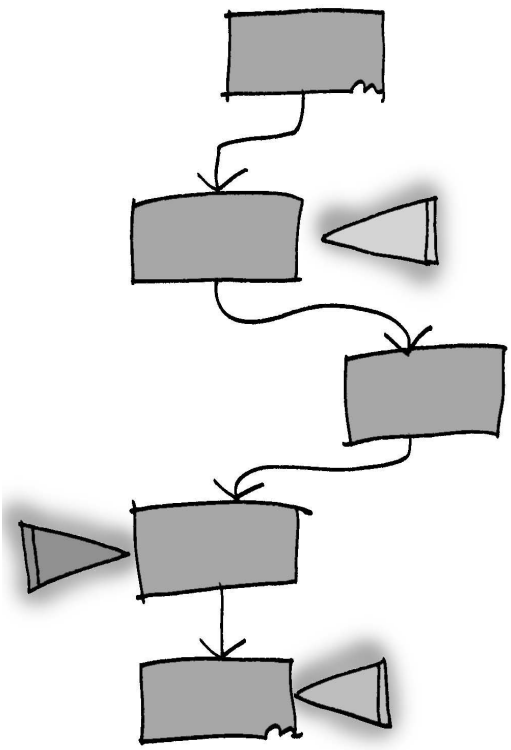
那么接下来，让我们赶快进入正题，逐个学习设计模式吧。

目 录

第 1 部分　适应设计模式

第 1 章　Iterator 模式

一个一个遍历



1.1 Iterator 模式

使用 Java 语言显示数组 arr 中的元素时，我们可以使用下面这样的 for 循环语句遍历数组。

```
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

请注意这段代码中的循环变量 i。该变量的初始值是 0，然后会递增为 1，2，3，...，程序则在每次 i 递增后都输出 arr[i]。我们在程序中经常会看到这样的 for 循环语句。

数组中保存了很多元素，通过指定数组下标，我们可以从中选择任意一个元素。

```
arr[0]  最开始的元素（第 0 个元素）
arr[1]  下一个元素（第 1 个元素）
          ⋮
arr[i]  （第 i 个元素）
          ⋮
arr[arr.length - 1]  最后一个元素
```

for 语句中的 i++ 的作用是让 i 的值在每次循环后自增 1，这样就可以访问数组中的下一个元素、下下一个元素、再下下一个元素，也就实现了从头至尾逐一遍历数组元素的功能。

将这里的循环变量 i 的作用抽象化、通用化后形成的模式，在设计模式中称为 **Iterator 模式**。

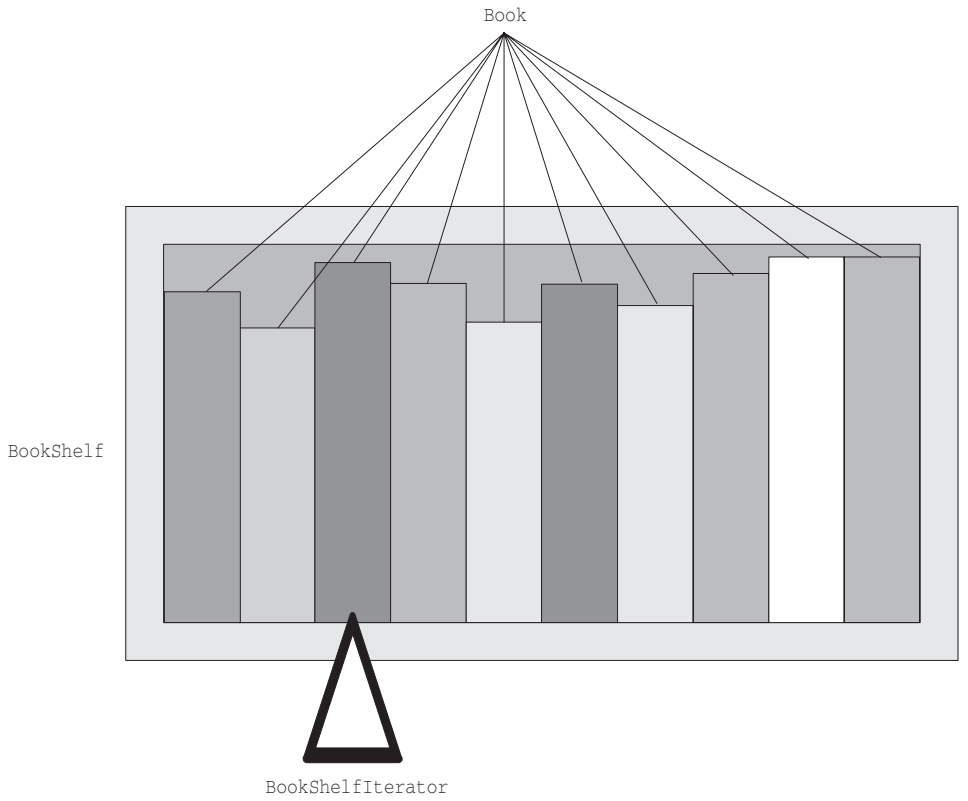
Iterator 模式用于在数据集合中按照顺序遍历集合。英语单词 Iterate 有反复做某件事情的意思，汉语称为“**迭代器**”。

我们将在本章中学习 Iterator 模式。

1.2 示例程序

首先，让我们来看一段实现了 Iterator 模式的示例程序。这段示例程序的作用是将书（Book）放置到书架（BookShelf）中，并将书的名字按顺序显示出来（图 1-1）。

图 1-1 示例程序的示意图



Aggregate 接口

`Aggregate` 接口（代码清单 1-1）是所要遍历的集合的接口。实现了该接口的类将成为一个可以保存多个元素的集合，就像数组一样。Aggregate 有“使聚集”“集合”的意思。

图 1-2 示例程序的类图

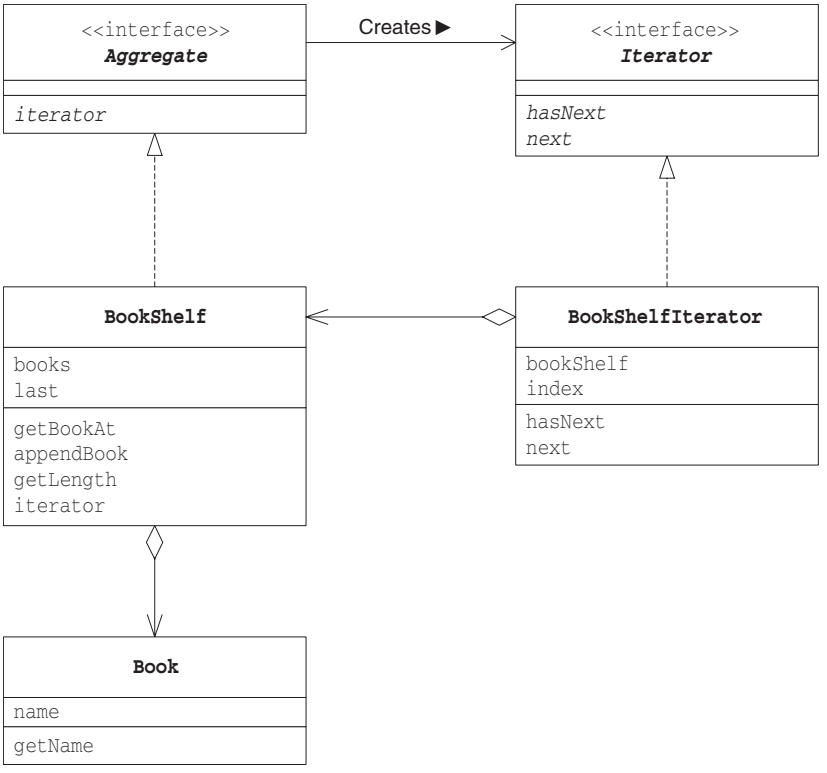


表 1-1 类和接口的一览表

名字	说明
Aggregate	表示集合的接口
Iterator	遍历集合的接口
Book	表示书的类
BookShelf	表示书架的类
BookShelfIterator	遍历书架的类
Main	测试程序行为的类

代码清单 1-1 Aggregate 接口（Aggregate.java）

```
public interface Aggregate {
    public abstract Iterator iterator();
}
```

在 Aggregate 接口中声明的方法只有一个——iterator 方法。该方法会生成一个用于遍历集合的迭代器。

想要遍历集合中的元素时，可以调用 iterator 方法来生成一个实现了 Iterator 接口的类的实例。

Iterator 接口

接下来我们看看 Iterator 接口（代码清单 1-2）。Iterator 接口用于遍历集合中的元素，其作用相当于循环语句中的循环变量。那么，在 Iterator 接口中需要有哪些方法呢？ Iterator 接口的定义方式有很多种，这里我们编写了最简单的 Iterator 接口。

代码清单 1-2　Iterator 接口（Iterator.java）

```
public interface Iterator {
    public abstract boolean hasNext();
    public abstract Object next();
}
```

这里我们声明了两个方法，即判断是否存在下一个元素的 hasNext 方法，和获取下一个元素的 next 方法。

hasNext 方法的返回值是 boolean 类型的，其原因很容易理解。当集合中存在下一个元素时，该方法返回 true；当集合中不存在下一个元素，即已经遍历至集合末尾时，该方法返回 false。hasNext 方法主要用于循环终止条件。

这里有必要说明一下 next 方法。该方法的返回类型是 Object，这表明该方法返回的是集合中的一个元素。但是，next 方法的作用并非仅仅如此。为了能够在下次调用 next 方法时正确地返回下一个元素，该方法中还隐含着将迭代器移动至下一个元素的处理。说“隐含”，是因为 Iterator 接口只知道方法名。想要知道 next 方法中到底进行了什么样的处理，还需要看一下实现了 Iterator 接口的类（BookShelfIterator）。这样，我们才能看懂 next 方法的作用。

Book 类

Book 类是表示书的类（代码清单 1-3）。但是这个类的作用有限，它可以做的事情只有一件——通过 getName 方法获取书的名字。书的名字是在外部调用 Book 类的构造函数并初始化 Book 类时，作为参数传递给 Book 类的。

代码清单 1-3　Book 类（Book.java）

```
public class Book {
    private String name;
    public Book(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

BookShelf 类

BookShelf 类是表示书架的类（代码清单 1-4）。由于需要将该类作为集合进行处理，因此它实现了 Aggregate 接口。代码中的 implements Aggregate 部分即表示这一点。此外，请注意在 BookShelf 类中还实现了 Aggregate 接口的 iterator 方法。

代码清单 1-4 BookShelf 类 (BookShelf.java)

```
public class BookShelf implements Aggregate {
    private Book[] books;
    private int last = 0;
    public BookShelf(int maxsize) {
        this.books = new Book[maxsize];
    }
    public Book getBookAt(int index) {
        return books[index];
    }
    public void appendBook(Book book) {
        this.books[last] = book;
        last++;
    }
    public int getLength() {
        return last;
    }
    public Iterator iterator() {
        return new BookShelfIterator(this);
    }
}
```

这个书架中定义了 books 字段，它是 Book 类型的数组。该数组的大小 (maxsize) 在生成 BookShelf 的实例时就被指定了。之所以将 books 字段的可见性设置为 private，是为了防止外部不小心改变了该字段的值。

接下来我们看看 iterator 方法。该方法会生成并返回 BookShelfIterator 类的实例作为 BookShelf 类对应的 Iterator。当外部想要遍历书架时，就会调用这个方法。

BookShelfIteraotr 类

接下来让我们看看用于遍历书架的 BookShelfIterator 类 (代码清单 1-5)。

代码清单 1-5 BookShelfIterator 类 (BookShelfIterator.java)

```
public class BookShelfIterator implements Iterator {
    private BookShelf bookShelf;
    private int index;
    public BookShelfIterator(BookShelf bookShelf) {
        this.bookShelf = bookShelf;
        this.index = 0;
    }
    public boolean hasNext() {
        if (index < bookShelf.getLength()) {
            return true;
        } else {
            return false;
        }
    }
    public Object next() {
        Book book = bookShelf.getBookAt(index);
        index++;
        return book;
    }
}
```

因为 BookShelfIterator 类需要发挥 Iterator 的作用，所以它实现了 Iterator 接口。

bookShelf 字段表示 BookShelfIterator 所要遍历的书架。index 字段表示迭代器当前所指向的书的下标。

构造函数会将接收到的 BookShelf 的实例保存在 bookShelf 字段中，并将 index 初始化为 0。

hasNext 方法是 Iterator 接口中所声明的方法。该方法将会判断书架中还有没有下一本书，如果有就返回 true，如果没有就返回 false。而要知道书架中有没有下一本书，可以通过比较 index 和书架中书的总册数（bookShelf.getLength() 的返回值）来判断。

next 方法会返回迭代器当前所指向的书（Book 的实例），并让迭代器指向下一本书。它也是 Iterator 接口中所声明的方法。next 方法稍微有些复杂，它首先取出 book 变量作为返回值，然后让 index 指向后面一本书。

如果与本章开头的 for 语句来对比，这里的“让 index 指向后面一本书”的处理相当于其中的 i++，它让循环变量指向下一个元素。

Main 类

至此，遍历书架的准备工作就完成了。接下来我们使用 Main 类（代码清单 1-6）来制作一个小书架。

代码清单 1-6　Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        BookShelf bookShelf = new BookShelf(4);
        bookShelf.appendBook(new Book("Around the World in 80 Days"));
        bookShelf.appendBook(new Book("Bible"));
        bookShelf.appendBook(new Book("Cinderella"));
        bookShelf.appendBook(new Book("Daddy-Long-Legs"));
        Iterator it = bookShelf.iterator();
        while (it.hasNext()) {
            Book book = (Book)it.next();
            System.out.println(book.getName());
        }
    }
}
```

这段程序首先设计了一个能容纳 4 本书的书架，然后按书名的英文字母顺序依次向书架中放入了下面这 4 本书。

Around the World in 80 Days（《环游世界 80 天》）
Bible（《圣经》）
Cinderella（《灰姑娘》）
Daddy Long Legs（《长腿爸爸》）

为了便于理解，笔者特意选了这 4 本首字母分别为 A、B、C、D 的书。

通过 bookShelf.iterator() 得到的 it 是用于遍历书架的 Iterator 实例。while 部分的条件当然就是 it.hasNext() 了。只要书架上有书，while 循环就不会停止。然后，程序会通过 it.next() 一本一本地遍历书架中的书。

图 1-3 展示了上面这段代码的运行结果。

图 1-3 运行结果

```
Around the World in 80 Days
Bible
Cinderella
Daddy-Long-Legs
```

1.3 Iterator 模式中的登场角色

读完示例程序，让我们来看看 Iterator 模式中的登场角色。

◆ Iterator（迭代器）

该角色负责定义按顺序逐个遍历元素的接口（API）。在示例程序中，由 `Iterator` 接口扮演这个角色，它定义了 `hasNext` 和 `next` 两个方法。其中，`hasNext` 方法用于判断是否存在下一个元素，`next` 方法则用于获取该元素。

◆ ConcreteIterator（具体的迭代器）

该角色负责实现 Iterator 角色所定义的接口（API）。在示例程序中，由 `BookShelfIterator` 类扮演这个角色。该角色中包含了遍历集合所必需的信息。在示例程序中，`BookShelf` 类的实例保存在 `bookShelf` 字段中，被指向的书的下标保存在 `index` 字段中。

◆ Aggregate（集合）

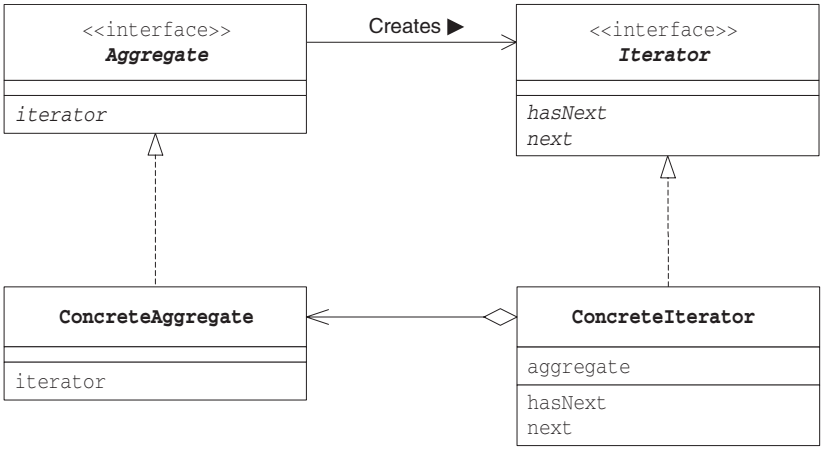
该角色负责定义创建 Iterator 角色的接口（API）。这个接口（API）是一个方法，会创建出“按顺序访问保存在我内部元素的人”。在示例程序中，由 `Aggregate` 接口扮演这个角色，它里面定义了 `iterator` 方法。

◆ ConcreteAggregate（具体的集合）

该角色负责实现 Aggregate 角色所定义的接口（API）。它会创建出具体的 Iterator 角色，即 ConcreteIterator 角色。在示例程序中，由 `BookShelf` 类扮演这个角色，它实现了 `iterator` 方法。

图 1-4 是展示了 Iterator 模式的类图。

图 1-4 Iterator 模式的类图



1.4 拓展思路的要点

不管实现如何变化，都可以使用 Iterator

为什么一定要考虑引入 Iterator 这种复杂的设计模式呢？如果是数组，直接使用 for 循环语句进行遍历处理不就可以了吗？为什么要在集合之外引入 Iterator 这个角色呢？

一个重要的理由是，引入 Iterator 后可以将遍历与实现分离开来。请看下面的代码。

```
while (it.hasNext()) {
    Book book = (Book)it.next();
    System.out.println(book.getName());
}
```

这里只使用了 Iterator 的 hasNext 方法和 next 方法，并没有调用 BookShelf 的方法。也就是说，**这里的 while 循环并不依赖于 BookShelf 的实现**。

如果编写 BookShelf 的开发人员决定放弃用数组来管理书本，而是用 java.util.Vector 取而代之，会怎样呢？不管 BookShelf 如何变化，只要 BookShelf 的 iterator 方法能正确地返回 Iterator 的实例（也就是说，返回的 Iterator 类的实例没有问题，hasNext 和 next 方法都可以正常工作），即使不对上面的 while 循环做任何修改，代码都可以正常工作。

这对于 BookShelf 的调用者来说真是太方便了。设计模式的作用就是帮助我们编写可复用的类。所谓“可复用”，就是指将类实现为“组件”，当一个组件发生改变时，不需要对其他的组件进行修改或是只需要很小的修改即可应对。

这样也就能理解为什么在示例程序中 iterator 方法的返回值不是 BookShelfIterator 类型而是 Iterator 类型了（代码清单 1-6）。这表明，这段程序就是要使用 Iterator 的方法进行编程，而不是 BookShelfIterator 的方法。

难以理解抽象类和接口

难以理解抽象类和接口的人常常使用 ConcreteAggregate 角色和 ConcreteIterator 角色编程，而不使用 Aggregate 接口和 Iterator 接口，他们总想用具体的类来解决所有的问题。

但是如果只使用具体的类来解决问题，很容易导致类之间的强耦合，这些类也难以作为组件被再次利用。为了弱化类之间的耦合，进而使得类更加容易作为组件被再次利用，我们需要引入抽象类和接口。

这也是贯穿本书的思想。即使大家现在无法完全理解，相信随着深入阅读本书，也一定能够逐渐理解。请大家将“不要只使用具体类来编程，要优先使用抽象类和接口来编程”印在脑海中。

Aggregate 和 Iterator 的对应

请大家仔细回忆一下我们是如何把 BookShelfIterator 类定义为 BookShelf 类的 ConcreteIterator 角色的。BookShelfIterator 类知道 BookShelf 是如何实现的。也正是因为如此，我们才能调用用来获取下一本书的 getBookAt 方法。

也就是说，如果 BookShelf 的实现发生了改变，即 getBookAt 方法这个接口（API）发生变

化时，我们必须修改 BookShelfIterator 类。

正如 Aggregate 和 Iterator 这两个接口是对应的一样，ConcreteAggregate 和 ConcreteIterator 这两个类也是对应的。

容易弄错“下一个”

在 Iterator 模式的实现中，很容易在 next 方法上出错。该方法的返回值到底是应该指向当前元素还是当前元素的下一个元素呢？更详细地讲，next 方法的名字应该是下面这样的。

```
returnCurrentElementAndAdvanceToNextPosition
```

也就是说，next 方法是“返回当前的元素，并指向下一个元素”。

还容易弄错“最后一个”

在 Iterator 模式中，不仅容易弄错“下一个”，还容易弄错“最后一个”。hasNext 方法在返回最后一个元素前会返回 true，当返回了最后一个元素后则返回 false。稍不注意，就会无法正确地返回“最后一个”元素。

请大家将 hasNext 方法理解成“确认接下来是否可以调用 next 方法”的方法就可以了。

多个 Iterator

“将遍历功能置于 Aggregate 角色之外”是 Iterator 模式的一个特征。根据这个特征，可以针对一个 ConcreteAggregate 角色编写多个 ConcreteIterator 角色。

迭代器的种类多种多样

在示例程序中展示的 Iterator 类只是很简单地从前向后遍历集合。其实，遍历的方法是多种多样的。

- **从最后开始向前遍历**
- **既可以从前向后遍历，也可以从后向前遍历（既有 next 方法也有 previous 方法）**
- **指定下标进行“跳跃式”遍历**

学到这里，相信大家应该可以根据需求编写出各种各样的 Iterator 类了。

不需要 deleteIterator

Java

在 Java 中，没有被使用的对象实例将会自动被删除（垃圾回收，GC）。因此，在 iterator 中不需要与其对应的 deleteIterator 方法。

1.5 相关的设计模式

◆ Visitor 模式（第 13 章）

Iterator 模式是从集合中一个一个取出元素进行遍历，但是并没有在 `Iterator` 接口中声明对取出的元素进行何种处理。

Visitor 模式则是在遍历元素集合的过程中，对元素进行相同的处理。

在遍历集合的过程中对元素进行固定的处理是常有的需求。Visitor 模式正是为了应对这种需求而出现的。在访问元素集合的过程中对元素进行相同的处理，这种模式就是 Visitor 模式。

◆ Composite 模式（第 11 章）

Composite 模式是具有递归结构的模式，在其中使用 Iterator 模式比较困难。

◆ Factory Method 模式（第 4 章）

在 `iterator` 方法中生成 `Iterator` 的实例时可能会使用 Factory Method 模式。

1.6 本章所学知识

在本章中，我们学习了按照统一的方法遍历集合中的元素的 Iterator 模式。

接下来让我们做一下练习题吧。

1.7 练习题

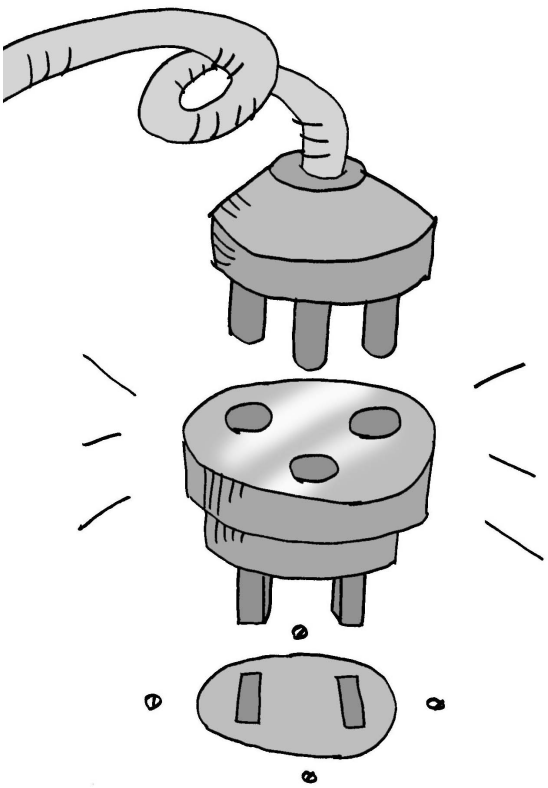
答案请参见附录 A（P.294）

● 习题 1-1

在示例程序的 `BookShelf` 类（代码清单 1-4）中，当书的数量超过最初指定的书架容量时，就无法继续向书架中添加书本了。请大家不使用数组，而是用 `java.util.ArrayList` 修改程序，确保当书的数量超过最初指定的书架容量时也能继续向书架中添加书本。

第 2 章　Adapter 模式

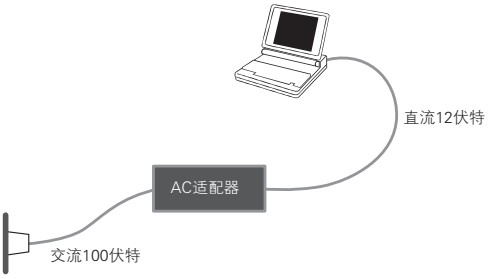
加个“适配器”以便于复用



2.1 Adapter 模式

如果想让额定工作电压是直流 12 伏特的笔记本电脑在交流 100 伏特[①] 的 AC 电源下工作，应该怎么做呢？通常，我们会使用 AC 适配器，将家庭用的交流 100 伏特电压转换成我们所需要的直流 12 伏特电压。这就是适配器的工作，它位于实际情况与需求之间，填补两者之间的差异。适配器的英文是 Adapter，意思是“使……相互适合的东西”。前面说的 AC 适配器的作用就是让工作于直流 12 伏特环境的笔记本电脑适合于交流 100 伏特的环境（图 2-1）。

图 2-1 适配器的角色



在程序世界中，经常会存在现有的程序无法直接使用，需要做适当的变换之后才能使用的情况。这种用于填补“现有的程序”和“所需的程序”之间差异的设计模式就是 **Adapter 模式**。

Adapter 模式也被称为 Wrapper 模式。Wrapper 有“包装器”的意思，就像用精美的包装纸将普通商品包装成礼物那样，替我们把某样东西包起来，使其能够用于其他用途的东西就被称为“包装器”或是“适配器”。

Adapter 模式有以下两种。

- **类适配器模式（使用继承的适配器）**
- **对象适配器模式（使用委托的适配器）**

本章将依次学习这两种 Adapter 模式。

2.2 示例程序（1）（使用继承的适配器）

首先，让我们来看一段使用继承的适配器的示例程序。这里的示例程序是一段会将输入的字符串显示为 `(Hello)` 或是 `*Hello*` 的简单程序。

目前在 `Banner` 类（Banner 有广告横幅的意思）中，有将字符串用括号括起来的 `showWithParen` 方法，和将字符串用 `*` 号括起来的 `showWithAster` 方法。我们假设这个 `Banner` 类是类似前文中的“交流 100 伏特电压”的“实际情况”。

假设 `Print` 接口中声明了两种方法，即弱化字符串显示（加括号）的 `printWeak`（weak 有弱化的意思）方法，和强调字符串显示（加 `*` 号）的 `printStrong`（strong 有强化的意思）方法。我们假设这个接口是类似于前文中的“直流 12 伏特电压”的“需求”。

① 日本的普通住宅区常用电压是 100 伏特，而国内居民区常用电压是 220 伏特，此处沿用了原文中的表述，故电压为 100 伏特。——译者注

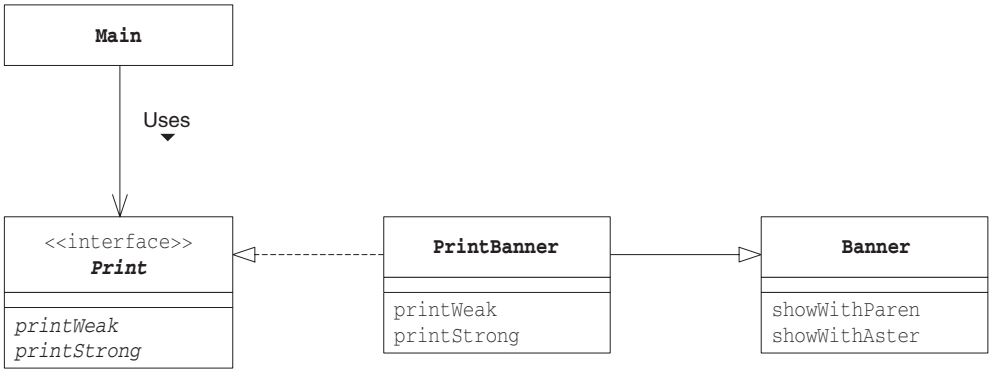
现在要做的事情是使用 Banner 类编写一个实现了 Print 接口的类，也就是说要做一个将“交流 100 伏特电压”转换成“直流 12 伏特电压”的适配器。

扮演适配器角色的是 PrintBanner 类。该类继承了 Banner 类并实现了“需求”——Print 接口。PrintBanner 类使用 showWithParen 方法实现了 printWeak，使用 showWithAster 方法实现了 printStrong。这样，PrintBanner 类就具有适配器的功能了。电源的比喻和示例程序的对应关系如表 2-1 所示。

表 2-1　电源的比喻和示例程序的对应关系

	电源的比喻	示例程序
实际情况	交流 100 伏特	Banner 类（showWithParen、showWithAster）
变换装置	适配器	PrintBanner 类
需求	直流 12 伏特	Print 接口（printWeak、printStrong）

图 2-2　使用了“类适配器模式”的示例程序的类图（使用继承）



Banner 类

假设 Banner 类（代码清单 2-1）是现在的实际情况。

代码清单 2-1　Banner 类（Banner.java）

```
public class Banner {
    private String string;
    public Banner(String string) {
        this.string = string;
    }
    public void showWithParen() {
        System.out.println("(" + string + ")");
    }
    public void showWithAster() {
        System.out.println("*" + string + "*");
    }
}
```

Print 接口

假设 Print 接口（代码清单 2-2）是“需求”的接口。

代码清单 2-2 Print 接口（Print.java）

```
public interface Print {
    public abstract void printWeak();
    public abstract void printStrong();
}
```

PrintBanner 类

PrintBanner 类（代码清单 2-3）扮演适配器的角色。它继承（extends）了 Banner 类，继承了 showWithParen 方法和 showWithAster 方法。同时，它又实现（implements）了 Print 接口，实现了 printWeak 方法和 printStrong 方法。

代码清单 2-3 PrintBanner 类（PrintBanner.java）

```
public class PrintBanner extends Banner implements Print {
    public PrintBanner(String string) {
        super(string);
    }
    public void printWeak() {
        showWithParen();
    }
    public void printStrong() {
        showWithAster();
    }
}
```

Main 类

Main 类（代码清单 2-4）的作用是通过扮演适配器角色的 PrintBanner 类来弱化（带括号）或是强化 Hello（带 * 号）字符串的显示。

代码清单 2-4 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        Print p = new PrintBanner("Hello");
        p.printWeak();
        p.printStrong();
    }
}
```

图 2-3 运行结果

```
(Hello)
*Hello*
```

请注意，这里我们将 PrintBanner 类的实例保存在了 Print 类型的变量中。在 Main 类中，

我们是使用 Print 接口（即调用 printWeak 方法和 printStrong 方法）来进行编程的。对 Main 类的代码而言，Banner 类、showWithParen 方法和 showWithAster 方法被完全隐藏起来了。这就好像笔记本电脑只要在直流 12 伏特电压下就能正常工作，但它并不知道这 12 伏特的电压是由适配器将 100 伏特交流电压转换而成的。

Main 类并不知道 PrintBanner 类是如何实现的，这样就可以在不用对 Main 类进行修改的情况下改变 PrintBanner 类的具体实现。

2.3 示例程序（2）（使用委托的示例程序）

之前的示例程序展示了类适配器模式。下面我们再来看看对象适配器模式。在之前的示例程序中，我们使用“继承”实现适配，而这次我们要使用“委托”来实现适配。

▶▶小知识：关于委托

“委托”这个词太过于正式了，说得通俗点就是“交给其他人”。比如，当我们无法出席重要会议时，可以写一份委任书，说明一下“我无法出席会议，安排佐藤代替我出席”。委托跟委任的意思是一样的。在 Java 语言中，委托就是指将某个方法中的实际处理交给其他实例的方法。

Main 类和 Banner 类与示例程序（1）中的内容完全相同，不过这里我们假设 Print 不是接口而是类（代码清单 2-5）。

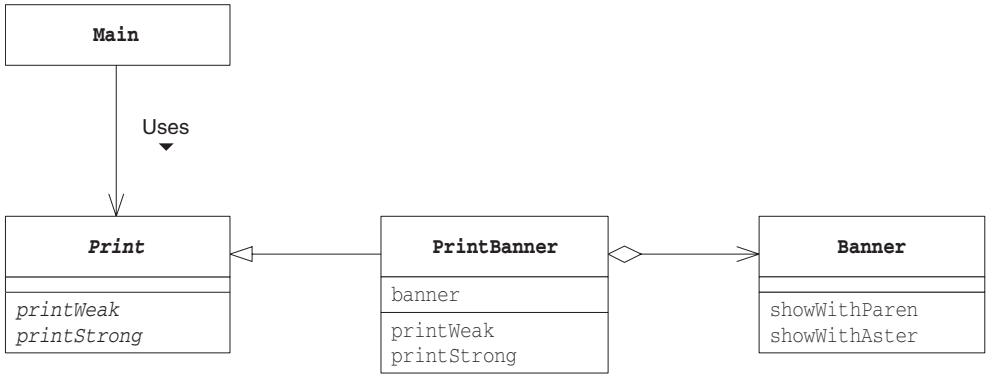
也就是说，我们打算利用 Banner 类实现一个类，该类的方法和 Print 类的方法相同。由于在 Java 中无法同时继承两个类（只能是单一继承），因此我们无法将 PrintBanner 类分别定义为 Print 类和 Banner 类的子类。

PrintBanner 类（代码清单 2-6）的 banner 字段中保存了 Banner 类的实例。该实例是在 PrintBanner 类的构造函数中生成的。然后，printWeak 方法和 printStrong 方法会通过 banner 字段调用 Banner 类的 showWithParen 和 showWithAster 方法。

与之前的示例代码中调用了从父类中继承的 showWithParen 方法和 showWithAster 方法不同，这次我们通过字段来调用这两个方法。

这样就形成了一种委托关系（图 2-4）。当 PrintBanner 类的 printWeak 被调用的时候，并不是 PrintBanner 类自己进行处理，而是将处理交给了其他实例（Banner 类的实例）的 showWithParen 方法。

图 2-4 使用了“对象适配器模式”的示例程序的类图（使用委托）



Print 类

代码清单 2-5 Print 类 (Print.java)

```
public abstract class Print {
    public abstract void printWeak();
    public abstract void printStrong();
}
```

PrintBanner 类

代码清单 2-6 PrintBanner 类 (PrintBanner.java)

```
public class PrintBanner extends Print {
    private Banner banner;
    public PrintBanner(String string) {
        this.banner = new Banner(string);
    }
    public void printWeak() {
        banner.showWithParen();
    }
    public void printStrong() {
        banner.showWithAster();
    }
}
```

2.4 Adapter 模式中的登场角色

在 Adapter 模式中有以下登场角色。

◆ Target (对象)

该角色负责定义所需的方法。以本章开头的例子来说，即让笔记本电脑正常工作所需的直流 12 伏特电源。在示例程序中，由 `Print` 接口 (使用继承时) 和 `Print` 类 (使用委托时) 扮演此角色。

◆ Client (请求者)

该角色负责使用 Target 角色所定义的方法进行具体处理。以本章开头的例子来说，即直流 12 伏特电源所驱动的笔记本电脑。在示例程序中，由 `Main` 类扮演此角色。

◆ Adaptee (被适配)

注意不是 Adapt-er (适配) 角色，而是 Adapt-ee (被适配) 角色。Adaptee 是一个持有既定方法的角色。以本章开头的例子来说，即交流 100 伏特电源。在示例程序中，由 `Banner` 类扮演此角色。

如果 Adaptee 角色中的方法与 Target 角色的方法相同 (也就是说家庭使用的电压就是 12 伏特直流电压)，就不需要接下来的 Adapter 角色了。

◆ Adapter (适配)

Adapter 模式的主人公。使用 Adaptee 角色的方法来满足 Target 角色的需求，这是 Adapter 模式的目的，也是 Adapter 角色的作用。以本章开头的例子来说，Adapter 角色就是将交流 100 伏特电

压转换为直流 12 伏特电压的适配器。在示例程序中，由 PrintBanner 类扮演这个角色。

在类适配器模式中，Adapter 角色通过继承来使用 Adaptee 角色，而在对象适配器模式中，Adapter 角色通过委托来使用 Adaptee 角色。

图 2-5 和图 2-6 展示了这两种 Adapter 模式的类图。

图 2-5 类适配器模式的类图（使用继承）

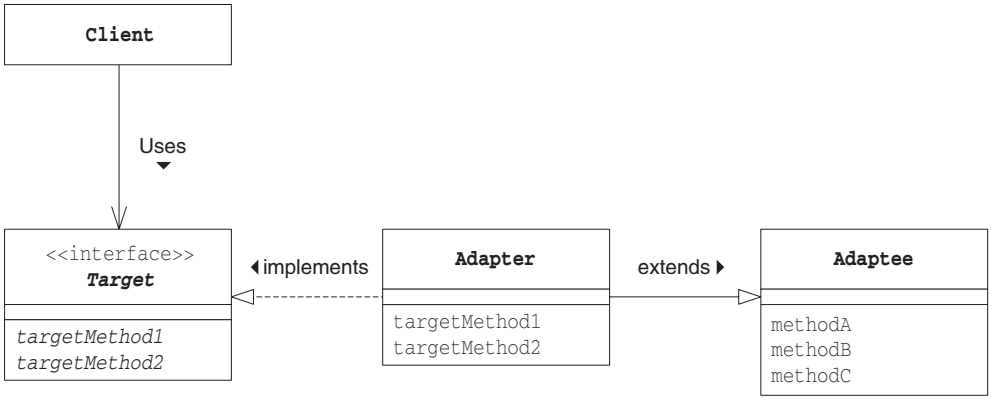
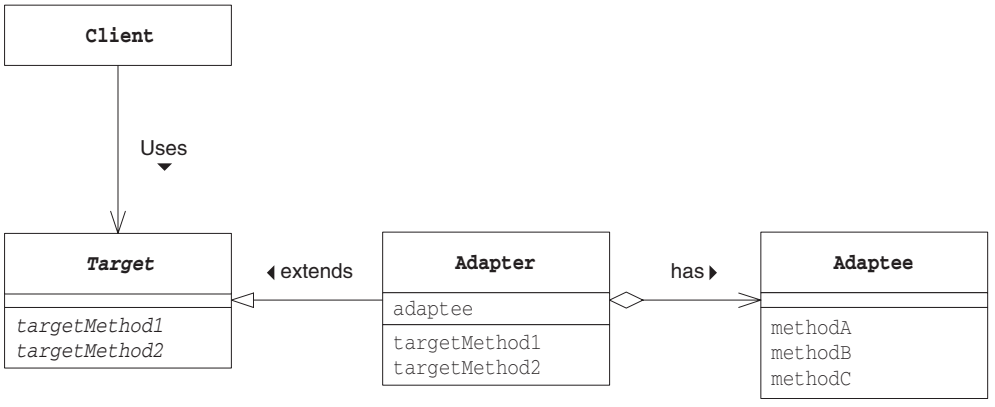


图 2-6 对象适配器模式的类图（使用委托）



2.5 拓展思路的要点

什么时候使用 Adapter 模式

一定会有读者认为“如果某个方法就是我们所需要的方法，那么直接在程序中使用不就可以了吗？为什么还要考虑使用 Adapter 模式呢？”那么，究竟应当在什么时候使用 Adapter 模式呢？

很多时候，我们并非从零开始编程，经常会用到现有的类。特别是当现有的类已经被充分测试过了，Bug 很少，而且已经被用于其他软件之中时，我们更愿意将这些类作为组件重复利用。

Adapter 模式会对现有的类进行适配，生成新的类。通过该模式可以很方便地创建我们需要的方法群。当出现 Bug 时，由于我们很明确地知道 Bug 不在现有的类（Adaptee 角色）中，所以只需调查扮演 Adapter 角色的类即可。这样一来，代码问题的排查就会变得非常简单。

如果没有现成的代码

让现有的类适配新的接口（API）时，使用 Adapter 模式似乎是理所当然的。不过实际上，我们在让现有的类适配新的接口时，常常会有“只要将这里稍微修改下就可以了”的想法，一不留神就会修改现有的代码。但是需要注意的是，如果要对已经测试完毕的现有代码进行修改，就必须在修改后重新进行测试。

使用 Adapter 模式可以在完全不改变现有代码的前提下使现有代码适配于新的接口（API）。此外，在 Adapter 模式中，并非一定需要现成的代码。只要知道现有类的功能，就可以编写出新的类。

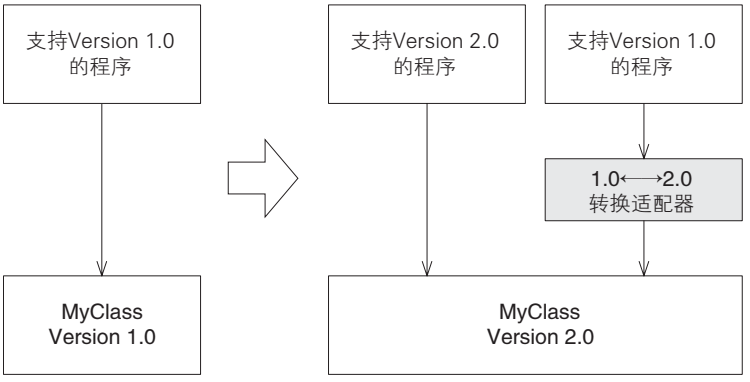
版本升级与兼容性

软件的生命周期总是伴随着版本的升级，而在版本升级的时候经常会出现“与旧版本的兼容性”问题。如果能够完全抛弃旧版本，那么软件的维护工作将会轻松得多，但是现实中往往无法这样做。这时，可以使用 Adapter 模式使新旧版本兼容，帮助我们轻松地同时维护新版本和旧版本。

例如，假设我们今后只想维护新版本。这时可以让新版本扮演 Adaptee 角色，旧版本扮演 Target 角色。接着编写一个扮演 Adapter 角色的类，让它使用新版本的类来实现旧版本的类中的方法。

图 2-7 展示了这些关系的类图（请注意它并非 UML 图）。

图 2-7 提高与旧版本软件的兼容性的 Adapter 模式



功能完全不同的类

当然，当 Adaptee 角色和 Target 角色的功能完全不同时，Adapter 模式是无法使用的。就如同我们无法用交流 100 伏特电压让自来水管出水一样。

2.6 相关的设计模式

◆ **Bridge 模式**（第 9 章）

Adapter 模式用于连接接口（API）不同的类，而 Bridge 模式则用于连接类的功能层次结构与实现层次结构。

◆ Decorator 模式（第 12 章）

Adapter 模式用于填补不同接口（API）之间的缝隙，而 Decorator 模式则是在不改变接口（API）的前提下增加功能。

2.7 本章所学知识

在本章中，我们学习了 Adapter 模式。Adapter 模式用于填补具有不同接口（API）的两个类之间的缝隙。此外，我们还学习了“使用继承”和“使用委托”这两种实现 Adapter 模式的方式和它们各自的特征。

现在大家应该对设计模式有些了解了，那么接下来让我们做两道练习题。

2.8 练习题

答案请参见附录 A（P.295）

● 习题 2-1

Java 在示例程序中生成 PrintBanner 类的实例时，我们采用了如下方法，即使用 Print 类型的变量来保存 PrintBanner 实例。

```
Print p = new PrintBanner("Hello");
```

请问我们为什么不像下面这样使用 PrintBanner 类型的变量来保存 PrintBanner 的实例呢？

```
PrintBannerp = new PrintBanner("Hello");
```

● 习题 2-2

在 java.util.Properties 类中，可以像下面这样管理键值对（属性）。

```
year=2004
month=4
day=21
```

java.util.Properties 类提供了以下方法，可以帮助我们方便地从流中取出属性或将属性写入流中。

```
void load(InputStream in) throws IOException
从 InputStream 中取出属性集合
void store(OutputStream out, String header) throws IOException
向 OutputStream 写入属性集合。header 是注释文字
```

请使用 Adapter 模式编写一个将属性集合保存至文件中的 FileProperties 类。

这里，我们假设在代码清单 2-7 中的 FileIO 接口（Target 角色）中声明了将属性集合保存至文件的方法，并假设 FileProperties 类会实现这个 FileIO 接口。

输入文件 file.txt 以及输出文件 newfile.txt 的内容请参见代码清单 2-9 和代码清单 2-10（以 # 开始的内容是 java.util.Properties 类自动附加的注释文字）。

当 FileProperties 类编写完成后，即使 FileProperties 类不了解 java.util.Properties 类的方法，只要知道 FileIO 接口的方法也可以对属性进行处理。还是以本章开头的电源的例子来说，java.util.Properties 类相当于现在家庭中使用的 100 伏特交流电压，FileIO 接口相当于所需要的直流 12 伏特电源，而 FileProperties 类则相当于适配器。

代码清单 2-7 FileIO 接口 (FileIO.java)

```
import java.io.*;

public interface FileIO {
    public void readFromFile(String filename) throws IOException;
    public void writeToFile(String filename) throws IOException;
    public void setValue(String key, String value);
    public String getValue(String key);
}
```

代码清单 2-8 Main 类 (Main.java)

```
import java.io.*;

public class Main {
    public static void main(String[] args) {
        FileIO f = new FileProperties();
        try {
            f.readFromFile("file.txt");
            f.setValue("year", "2004");
            f.setValue("month", "4");
            f.setValue("day", "21");
            f.writeToFile("newfile.txt");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

代码清单 2-9 输入文件 (file.txt)

```
year=1999
```

代码清单 2-10 输出文件 (newfile.txt)

```
#written by FileProperties
#Wed Apr 21 18:21:00 JST 2004
day=21
year=2004
month=4
```