第 16 章 设计范式

本章要向大家介绍重要但却并不是那么传统的“范式”（Pattern）程序设计方法。

在向面向对象程序设计的演化过程中，或许最重要的一步就是“设计范式”（Design Pattern）的问世。它在由 Gamma，Helm 和 Johnson 编著的《Design Patterns》一书中被定义成一个“里程碑”（该书由 Addison-Wesley 于 1995 年出版，注释①）。那本书列出了解决这个问题的 23 种不同的方法。在本章中，我们准备伴随几个例子揭示出设计范式的基本概念。这或许能激起您阅读《Design Pattern》一书的欲望。事实上，那本书现在已成为几乎所有 OOP 程序员都必备的参考书。

①：但警告大家：书中的例子是用 C++写的。

本章的后一部分包含了展示设计进化过程的一个例子，首先是比较原始的方案，经过逐渐发展和改进，慢慢成为更符合逻辑、更为恰当的设计。该程序（仿真垃圾分类）一直都在进化，可将这种进化作为自己设计方案的一个原型——先为特定的问题提出一个适当的方案，再逐步改善，使其成为解决那类问题一种最灵活的方案。

16.1 范式的概念

在最开始，可将范式想象成一种特别聪明、能够自我适应的手法，它可以解决特定类型的问题。也就是说，它类似一些需要全面认识某个问题的人。在了解了问题的方方面面以后，最后提出一套最通用、最灵活的解决方案。具体问题或

许是以前见到并解决过的。然而，从前的方案也许并不是最完善的，大家会看到它如何在一个范式里具体表达出来。

尽管我们称之为“设计范式”，但它们实际上并不局限于设计领域。思考“范式”时，应脱离传统意义上分析、设计以及实施的思考方式。相反，“范式”是在一个程序里具体表达一套完整的思想，所以它有时可能出现在分析阶段或者高级设计阶段。这一点是非常有趣的，因为范式具有以代码形式直接实现的形式，所以可能不希望它在低级设计或者具体实施以前显露出来（而且事实上，除非真正进入那些阶段，否则一般意识不到自己需要一个范式来解决问题）。

范式的基本概念亦可看成是程序设计的基本概念：添加一层新的抽象！只要我们抽象了某些东西，就相当于隔离了特定的细节。而且这后面最引人注目的动机就是“将保持不变的东西身上发生的变化孤立出来”。这样做的另一个原因是一旦发现程序的某部分由于这样或那样的原因可能发生变化，我们一般都想防止那些改变在代码内部繁衍出其他变化。这样做不仅可以降低代码的维护代价，也更便于我们理解（结果同样是降低开销）。

为设计出功能强大且易于维护的应用项目，通常最困难的部分就是找出我称之为“领头变化”的东西。这意味着需要找出造成系统改变的最重要的东西，或者换一个角度，找出付出代价最高、开销最大的那一部分。一旦发现了“领头变化”，就可以为自己定下一个焦点，围绕它展开自己的设计。

所以设计范式的最终目标就是将代码中变化的内容隔离开。如果从这个角度观察，就会发现本书实际已采用了一些设计范式。举个例子来说，继承可以想象成一种设计范式（类似一个由编译器实现的）。在都拥有同样接口（即保持不变的东西）的对象内部，它允许我们表达行为上的差异（即发生变化的东西）。合成亦可想象成一种范式，因为它允许我们修改——动态或静态——用于实现类的对象，所以也能修改类的运作方式。

在《Design Patterns》一书中，大家还能看到另一种范式：“继承器”（即 Iterator，Java 1.0 和 1.1 不负责任地把它叫作 Enumeration，即“枚举”；Java1.2 的集合则改回了“继承器”的称呼）。当我们在集合里遍历，逐个选择不同的元素时，继承器可将集合的实施细节有效地隐藏起来。利用继承器，可以编写出通用的代码，以便对一个序列里的所有元素采取某种操作，同时不必关心这个序列是如何构建的。这样一来，我们的通用代码即可伴随任何能产生继承器的集合使用。

16.1.1 单子

或许最简单的设计范式就是“单子”（Singleton），它能提供对象的一个（而且只有一个）实例。单子在 Java 库中得到了应用，但下面这个例子显得更直接一些：

909-910 页程序

创建单子的关键就是防止客户程序员采用除由我们提供的之外的任何一种方式来创建一个对象。必须将所有构建器都设为 private（私有），而且至少要创建一个构建器，以防止编译器帮我们自动同步一个默认构建器（它会自做聪明地创建成为“友好的”——friendly，而非 private）。

此时应决定如何创建自己的对象。在这儿，我们选择了静态创建的方式。但亦可选择等候客户程序员发出一个创建请求，然后根据他们的要求动态创建。不

管在哪种情况下，对象都应该保存为“私有”属性。我们通过公用方法提供访问途径。在这里，getHandle()会产生指向 Singleton 的一个句柄。剩下的接口（getValue()和 setValue()）属于普通的类接口。

Java 也允许通过克隆（Clone）方式来创建一个对象。在这个例子中，将类设为 final 可禁止克隆的发生。由于 Singleton 是从 Object 直接继承的，所以 clone()方法会保持 protected（受保护）属性，不能够使用它（强行使用会造成编译期错误）。然而，假如我们是从一个类结构中继承，那个结构已经过载了 clone()方法，使其具有 public 属性，并实现了 Cloneable，那么为了禁止克隆，需要过载 clone()，并掷出一个 CloneNotSupportedException（不支持克隆违例），就象第 12 章介绍的那样。亦可过载 clone()，并简单地返回 this。那样做会造成一定的混淆，因为客户程序员可能错误地认为对象尚未克隆，仍然操纵的是原来的那个。

注意我们并不限于只能创建一个对象。亦可利用该技术创建一个有限的对象池。但在那种情况下，可能需要解决池内对象的共享问题。如果不幸真的遇到这个问题，可以自己设计一套方案，实现共享对象的登记与撤消登记。

16.1.2 范式分类

《Design Patterns》一书讨论了 23 种不同的范式，并依据三个标准分类（所有标准都涉及那些可能发生变化的方面）。这三个标准是：

(1) 创建：对象的创建方式。这通常涉及对象创建细节的隔离，这样便不必依赖具体类型的对象，所以在新添一种对象类型时也不必改动代码。

(2) 结构：设计对象，满足特定的项目限制。这涉及对象与其他对象的连接方式，以保证系统内的改变不会影响到这些连接。

(3) 行为：对程序中特定类型的行动进行操纵的对象。这要求我们将希望采取的操作封装起来，比如解释一种语言、实现一个请求、在一个序列中遍历（就象在继承器中那样）或者实现一种算法。本章提供了“观察器”（Observer）和“访问器”（Visitor）的范式的例子。

《Design Patterns》为所有这 23 种范式都分别使用了一节，随附的还有大量示例，但大多是用 C++编写的，少数用 Smalltalk 编写（如看过这本书，就知道这实际并不是个大问题，因为很容易即可将基本概念从两种语言翻译到 Java 里）。现在这本书并不打算重复《Design Patterns》介绍的所有范式，因为那是一本独立的书，大家应该单独阅读。相反，本章只准备给出一些例子，让大家先对范式有个大致的印象，并理解它们的重要性到底在哪里。

16.2 观察器范式

观察器（Observer）范式解决的是一个相当普通的问题：由于某些对象的状态发生了改变，所以一组对象都需要更新，那么该如何解决？在 Smalltalk 的 MVC（模型－视图－控制器）的“模型－视图”部分中，或在几乎等价的“文档－视图结构”中，大家可以看到这个问题。现在我们有一些数据（“文档”）以及多个视图，假定为一张图（Plot）和一个文本视图。若改变了数据，两个视图必须知道对自己进行更新，而那正是“观察器”要负责的工作。这是一种十分常见的问题，它的解决方案已包括进标准的 java.util 库中。

在 Java 中，有两种类型的对象用来实现观察器范式。其中，Observable 类用于跟踪那些当发生一个改变时希望收到通知的所有个体——无论“状态”是否改

变。如果有人说“好了，所有人都要检查自己，并可能要进行更新”，那么Observable 类会执行这个任务——为列表中的每个“人”都调用 notifyObservers()方法。notifyObservers()方法属于基础类 Observable 的一部分。

在观察器范式中，实际有两个方面可能发生变化：观察对象的数量以及更新的方式。也就是说，观察器范式允许我们同时修改这两个方面，不会干扰围绕在它周围的其他代码。

下面这个例子类似于第 14 章的 ColorBoxes 示例。箱子（Boxes）置于一个屏幕网格中，每个都初始化一种随机的颜色。此外，每个箱子都“实现”（implement）了“观察器”（Observer）接口，而且随一个 Observable 对象进行了注册。若点击一个箱子，其他所有箱子都会收到一个通知，指出一个改变已经发生。这是由于 Observable 对象会自动调用每个 Observer 对象的 update()方法。在这个方法内，箱子会检查被点中的那个箱子是否与自己紧邻。若答案是肯定的，那么也修改自己的颜色，保持与点中那个箱子的协调。

912-914 页程序

如果是首次查阅 Observable 的联机帮助文档，可能会多少感到有些困惑，因为它似乎表明可以用一个原始的 Observable 对象来管理更新。但这种说法是不成立的；大家可自己试试——在 BoxObserver 中，创建一个 Observable 对象，替换 BoxObservable 对象，看看会有什么事情发生。事实上，什么事情也不会发生。为真正产生效果，必须从 Observable 继承，并在衍生类代码的某个地方调用 setChanged()。这个方法需要设置“changed”（已改变）标志，它意味着当我们调用 notifyObservers()的时候，所有观察器事实上都会收到通知。在上面的例子中，setChanged()只是简单地在 notifyObservers()中调用，大家可依据符合实际情况的任何标准决定何时调用 setChanged()。

BoxObserver 包含了单个 Observable 对象，名为 notifier。每次创建一个 OCBox 对象时，它都会同 notifier 联系到一起。在 OCBox 中，只要点击鼠标，就会发出对 notifyObservers()方法的调用，并将被点中的那个对象作为一个参数传递进去，使收到消息（用它们的 update()方法）的所有箱子都能知道谁被点中了，并据此判断自己是否也要变动。通过 notifyObservers()和 update()中的代码的结合，我们可以应付一些非常复杂的局面。

在 notifyObservers()方法中，表面上似乎观察器收到通知的方式必须在编译期间固定下来。然而，只要稍微仔细研究一下上面的代码，就会发现 BoxObserver 或 OCBox 中唯一需要留意是否使用 BoxObservable 的地方就是创建 Observable 对象的时候——从那时开始，所有东西都会使用基本的 Observable 接口。这意味着以后若想更改通知方式，可以继承其他 Observable 类，并在运行期间交换它们。

16.3 模拟垃圾回收站

这个问题的本质是若将垃圾丢进单个垃圾筒，事实上是未经分类的。但在以后，某些特殊的信息必须恢复，以便对垃圾正确地归类。在最开始的解决方案中，RTTI 扮演了关键的角色（详见第 11 章）。

这并不是一种普通的设计，因为它增加了一个新的限制。正是这个限制使问题变得非常有趣——它更象我们在工作中碰到的那些非常麻烦的问题。这个额外的限制是：垃圾抵达垃圾回收站时，它们全都是混合在一起的。程序必须为那些

垃圾的分类定出一个模型。这正是 RTTI 发挥作用的地方：我们有大量不知名的垃圾，程序将正确判断出它们所属的类型。

915-917 页程序

要注意的第一个地方是 package 语句：

package c16.recyclea;

这意味着在本书采用的源码目录中，这个文件会被置入从 c16（代表第 16 章的程序）分支出来的 recyclea 子目录中。第 17 章的解包工具会负责将其置入正确的子目录。之所以要这样做，是因为本章会多次改写这个特定的例子；它的每个版本都会置入自己的“包”（package）内，避免类名的冲突。

其中创建了几个 Vector 对象，用于容纳 Trash 句柄。当然，Vector 实际容纳的是 Object（对象），所以它们最终能够容纳任何东西。之所以要它们容纳 Trash（或者从 Trash 衍生出来的其他东西），唯一的理由是我们需要谨慎地避免放入除 Trash 以外的其他任何东西。如果真的把某些“错误”的东西置入 Vector，那么不会在编译期得到出错或警告提示——只能通过运行期的一个违例知道自己已经犯了错误。

Trash 句柄加入后，它们会丢失自己的特定标识信息，只会成为简单的 Object 句柄（上溯造型）。然而，由于存在多形性的因素，所以在我们通过 Enumeration sorter 调用动态绑定方法时，一旦结果 Object 已经造型回 Trash，仍然会发生正确的行为。sumValue()也用一个 Enumeration 对 Vector 中的每个对象进行操作。

表面上持，先把 Trash 的类型上溯造型到一个集合容纳基础类型的句柄，再回过头重新下溯造型，这似乎是一种非常愚蠢的做法。为什么不只是一开始就将垃圾置入适当的容器里呢？（事实上，这正是拨开“回收”一团迷雾的关键）。在这个程序中，我们很容易就可以换成这种做法，但在某些情况下，系统的结构及灵活性都能从下溯造型中得到极大的好处。

该程序已满足了设计的初衷：它能够正常工作！只要这是个一次性的方案，就会显得非常出色。但是，真正有用的程序应该能够在任何时候解决问题。所以必须问自己这样一个问题：“如果情况发生了变化，它还能工作吗？”举个例子来说，厚纸板现在是一种非常有价值的可回收物品，那么如何把它集成到系统中呢（特别是程序很大很复杂的时候）？由于前面在 switch 语句中的类型检查编码可能散布于整个程序，所以每次加入一种新类型时，都必须找到所有那些编码。若不慎遗漏一个，编译器除了指出存在一个错误之外，不能再提供任何有价值的帮助。

RTTI 在这里使用不当的关键是“每种类型都进行了测试”。如果由于类型的子集需要特殊的对待，所以只寻找那个子集，那么情况就会变得好一些。但假如在一个 switch 语句中查找每一种类型，那么很可能错过一个重点，使最终的代码很难维护。在下一节中，大家会学习如何逐步对这个程序进行改进，使其显得越来越灵活。这是在程序设计中一种非常有意义的例子。

16.4 改进设计

《Design Patterns》书内所有方案的组织都围绕“程序进化时会发生什么变化”这个问题展开。对于任何设计来说，这都可能是最重要的一个问题。若根据对这个问题的回答来构造自己的系统，就可以得到两个方面的结果：系统不仅更易维

护（而且更廉价），而且能产生一些能够重复使用的对象，进而使其他相关系统的构造也变得更廉价。这正是面向对象程序设计的优势所在，但这一优势并不是自动体现出来的。它要求对我们对需要解决的问题有全面而且深入的理解。在这一节中，我们准备在系统的逐步改进过程中向大家展示如何做到这一点。

就目前这个回收系统来说，对“什么会变化”这个问题的回答是非常普通的：更多的类型会加入系统。因此，设计的目标就是尽可能简化这种类型的添加。在回收程序中，我们准备把涉及特定类型信息的所有地方都封装起来。这样一来（如果没有别的原因），所有变化对那些封装来说都是在本地进行的。这种处理方式也使代码剩余的部分显得特别清爽。

16.4.1 “制作更多的对象”

这样便引出了面向对象程序设计时一条常规的准则，我最早是在 Grady Booch 那里听说的：“若设计过于复杂，就制作更多的对象”。尽管听起来有些暧昧，且简单得可笑，但这确实是我知道的最有用一条准则（大家以后会注意到“制作更多的对象”经常等同于“添加另一个层次的迂回”）。一般情况下，如果发现一个地方充斥着大量繁复的代码，就需要考虑什么类能使它显得清爽一些。用这种方式整理系统，往往会得到一个更好的结构，也使程序更加灵活。

首先考虑 Trash 对象首次创建的地方，这是 main()里的一个 switch 语句：

919-920 页程序

这些代码显然“过于复杂”，也是新类型加入时必须改动代码的场所之一。如果经常都要加入新类型，那么更好的方案就是建立一个独立的方法，用它获取所有必需的信息，并创建一个句柄，指向正确类型的一个对象——已经上溯造型到一个 Trash 对象。在《Design Patterns》中，它被粗略地称呼为“创建范式”。要在这里应用的特殊范式是 Factory 方法的一种变体。在这里，Factory 方法属于 Trash 的一名 static（静态）成员。但更常见的一种情况是：它属于衍生类中一个被过载的方法。

Factory 方法的基本原理是我们将创建对象所需的基本信息传递给它，然后返回并等候句柄（已经上溯造型至基础类型）作为返回值出现。从这时开始，就可以按多形性的方式对待对象了。因此，我们根本没必要知道所创建对象的准确类型是什么。事实上，Factory 方法会把自己隐藏起来，我们是看不见它的。这样做可防止不慎的误用。如果想在没有多形性的前提下使用对象，必须明确地使用 RTTI 和指定造型。

但仍然存在一个小问题，特别是在基础类中使用更复杂的方法（不是在这里展示的那种），且在衍生类里过载（覆盖）了它的前提下。如果在衍生类里请求的信息要求更多或者不同的参数，那么该怎么办呢？“创建更多的对象”解决了这个问题。为实现 Factory 方法，Trash 类使用了一个新的方法，名为 factory。为了将创建数据隐藏起来，我们用一个名为 Info 的新类包含 factory 方法创建适当的 Trash 对象时需要的全部信息。下面是 Info 一种简单的实现方式：

920-921 页程序

Info 对象唯一的任务就是容纳用于 factory()方法的信息。现在，假如出现了

一种特殊情况，factory()需要更多或者不同的信息来新建一种类型的 Trash 对象，那么再也不需要改动 factory()了。通过添加新的数据和构建器，我们可以修改 Info 类，或者采用子类处理更典型的面向对象形式。

用于这个简单示例的 factory()方法如下：

921 页上程序

在这里，对象的准确类型很容易即可判断出来。但我们可以设想一些更复杂的情况，factory()将采用一种复杂的算法。无论如何，现在的关键是它已隐藏到某个地方，而且我们在添加新类型时知道去那个地方。

新对象在 main()中的创建现在变得非常简单和清爽：

921 页下程序

我们在这里创建了一个 Info 对象，用于将数据传入 factory()；后者在内存堆中创建某种 Trash 对象，并返回添加到 Vector bin 内的句柄。当然，如果改变了参数的数量及类型，仍然需要修改这个语句。但假如 Info 对象的创建是自动进行的，也可以避免那个麻烦。例如，可将参数的一个 Vector 传递到 Info 对象的构建器中（或直接传入一个 factory()调用）。这要求在运行期间对参数（自变量）进行分析与检查，但确实提供了非常高的灵活程度。

大家从这个代码可看出 Factory 要负责解决的“领头变化”问题：如果向系统添加了新类型（发生了变化），唯一需要修改的代码在 Factory 内部，所以 Factory 将那种变化的影响隔离出来了。

16.4.2 用于原型创建的一个范式

上述设计方案的一个问题是仍然需要一个中心场所，必须在那里知道所有类型的对象：在 factory()方法内部。如果经常都要向系统添加新类型，factory()方法为每种新类型都要修改一遍。若确实对这个问题感到苦恼，可试试再深入一步，将与类型有关的所有信息——包括它的创建过程——都移入代表那种类型的类内部。这样一来，每次新添一种类型的时候，需要做的唯一事情就是从一个类继承。

为将涉及类型创建的信息移入特定类型的 Trash 里，必须使用“原型”（prototype）范式（来自《Design Patterns》那本书）。这里最基本的想法是我们有一个主控对象序列，为自己感兴趣的每种类型都制作一个。这个序列中的对象只能用于新对象的创建，采用的操作类似内建到 Java 根类 Object 内部的 clone()机制。在这种情况下，我们将克隆方法命名为 tClone()。准备创建一个新对象时，要事先收集好某种形式的信息，用它建立我们希望的对象类型。然后在主控序列中遍历，将手上的信息与主控序列中原型对象内任何适当的信息作对比。若找到一个符合自己需要的，就克隆它。

采用这种方案，我们不必用硬编码的方式植入任何创建信息。每个对象都知道如何揭示出适当的信息，以及如何对自身进行克隆。所以一种新类型加入系统的时候，factory()方法不需要任何改变。

为解决原型的创建问题，一个方法是添加大量方法，用它们支持新对象的创建。但在 Java 1.1 中，如果拥有指向 Class 对象的一个句柄，那么它已经提供了

对创建新对象的支持。利用 Java 1.1 的“反射”（已在第 11 章介绍）技术，即便我们只有指向 Class 对象的一个句柄，亦可正常地调用一个构建器。这对原型问题的解决无疑是个完美的方案。

原型列表将由指向所有想创建的 Class 对象的一个句柄列表间接地表示。除此之外，假如原型处理失败，则 factory()方法会认为由于一个特定的 Class 对象不在列表中，所以会尝试装载它。通过以这种方式动态装载原型，Trash 类根本不需要知道自己要操纵的是什么类型。因此，在我们添加新类型时不需要作出任何形式的修改。于是，我们可在本章剩余的部分方便地重复利用它。

923-925 页程序

基本 Trash 类和 sumValue()还是象往常一样。这个类剩下的部分支持原型范式。大家首先会看到两个内部类（被设为 static 属性，使其成为只为代码组织目的而存在的内部类），它们描述了可能出现的违例。在它后面跟随的是一个 Vector trashTypes，用于容纳 Class 句柄。

在 Trash.factory()中，Info 对象 id（Info 类的另一个版本，与前面讨论的不同）内部的 String 包含了要创建的那种 Trash 的类型名称。这个 String 会与列表中的 Class 名比较。若存在相符的，那便是要创建的对象。当然，还有很多方法可以决定我们想创建的对象。之所以要采用这种方法，是因为从一个文件读入的信息可以转换成对象。

发现自己要创建的 Trash（垃圾）种类后，接下来就轮到“反射”方法大显身手了。getConstructor()方法需要取得自己的参数——由 Class 句柄构成的一个数组。这个数组代表着不同的参数，并按它们正确的顺序排列，以便我们查找的构建器使用。在这儿，该数组是用 Java 1.1 的数组创建语法动态创建的：

new Class[] {double.class}

这个代码假定所有 Trash 类型都有一个需要 double 数值的构建器（注意 double.class 与 Double.class 是不同的）。若考虑一种更灵活的方案，亦可调用 getConstructors()，令其返回可用构建器的一个数组。

从 getConstructors()返回的是指向一个 Constructor 对象的句柄（该对象是 java.lang.reflect 的一部分）。我们用方法 newInstance()动态地调用构建器。该方法需要获取包含了实际参数的一个 Object 数组。这个数组同样是按 Java 1.1 的语法创建的：

new Object[] {new Double(info.data)}

在这种情况下，double 必须置入一个封装（容器）类的内部，使其真正成为这个对象数组的一部分。通过调用 newInstance()，会提取出 double，但大家可能会觉得稍微有些迷惑——参数既可能是 double，也可能是 Double，但在调用的时候必须用 Double 传递。幸运的是，这个问题只存在于基本数据类型中间。

理解了具体的过程后，再来创建一个新对象，并且只为它提供一个 Class 句柄，事情就变得非常简单了。就目前的情况来说，内部循环中的 return 永远不会执行，我们在终点就会退出。在这儿，程序动态装载 Class 对象，并把它加入 trashTypes（垃圾类型）列表，从而试图纠正这个问题。若仍然找不到真正有问题的地方，同时装载又是成功的，那么就重复调用 factory 方法，重新试一遍。

正如大家会看到的那样，这种设计方案最大的优点就是不需要改动代码。无论在什么情况下，它都能正常地使用（假定所有 Trash 子类都包含了一个构建器，

用以获取单个 double 参数）。

1. Trash 子类

为了与原型机制相适应，对 Trash 每个新子类唯一的要求就是在其中包含了一个构建器，指示它获取一个 double 参数。Java 1.1 的“反射”机制可负责剩下的所有工作。

下面是不同类型的 Trash，每种类型都有它们自己的文件里，但都属于 Trash 包的一部分（同样地，为了方便在本章内重复使用）：

926-927 页程序

下面是一种新的 Trash 类型：

927 页下程序

可以看出，除构建器以外，这些类根本没有什么特别的地方。

2. 从外部文件中解析出 Trash

与 Trash 对象有关的信息将从一个外部文件中读取。针对 Trash 的每个方面，文件内列出了所有必要的信息——每行都代表一个方面，采用“垃圾（废品）名称:值”的固定格式。例如：

928 页程序

注意在给定类名的时候，类路径必须包含在内，否则就找不到类。

为解析它，每一行内容都会读入，并用字串方法 indexOf()来建立“:”的一个索引。首先用字串方法 substring()取出垃圾的类型名称，接着用一个静态方法 Double.valueOf()取得相应的值，并转换成一个 double 值。trim()方法则用于删除字串两头的多余空格。

Trash 解析器置入单独的文件中，因为本章将不断地用到它。如下所示：

929-930 页程序

在 RecycleA.java 中，我们用一个 Vector 容纳 Trash 对象。然而，亦可考虑采用其他集合类型。为做到这一点，fillBin()的第一个版本将获取指向一个 Fillable 的句柄。后者是一个接口，用于支持一个名为 addTrash()的方法：

930 页上程序

支持该接口的所有东西都能伴随 fillBin 使用。当然，Vector 并未实现 Fillable，所以它不能工作。由于 Vector 将在大多数例子中应用，所以最好的做法是添加另一个过载的 fillBin()方法，令其以一个 Vector 作为参数。利用一个适配器（Adapter）类，这个 Vector 可作为一个 Fillable 对象使用：

930 页程序

可以看到，这个类唯一的任务就是负责将 Fillable 的 addTrash()同 Vector 的 addElement()方法连接起来。利用这个类，已过载的 fillBin()方法可在 ParseTrash.java 中伴随一个 Vector 使用：

930 页下程序

这种方案适用于任何频繁用到的集合类。除此以外，集合类还可提供它自己的适配器类，并实现 Fillable（稍后即可看到，在 DynaTrash.java 中）。

3. 原型机制的重复应用

现在，大家可以看到采用原型技术的、修订过的 RecycleA.java 版本了：

931 页程序

所有 Trash 对象——以及 ParseTrash 及支撑类——现在都成为名为 c16.trash 的一个包的一部分，所以它们可以简单地导入。

无论打开包含了 Trash 描述信息的数据文件，还是对那个文件进行解析，所有涉及到的操作均已封装到 static（静态）方法 ParseTrash.fillBin()里。所以它现在已经不是我们设计过程中要注意的一个重点。在本章剩余的部分，大家经常都会看到无论添加的是什么类型的新类，ParseTrash.fillBin()都会持续工作，不会发生改变，这无疑是一种优良的设计方案。

提到对象的创建，这一方案确实已将新类型加入系统所需的变动严格地“本地化”了。但在使用 RTTI 的过程中，却存在着一个严重的问题，这里已明确地显露出来。程序表面上工作得很好，但却永远侦测到不能“硬纸板”（Cardboard）这种新的废品类型——即使列表里确实有一个硬纸板类型！之所以会出现这种情况，完全是由于使用了 RTTI 的缘故。RTTI 只会查找那些我们告诉它查找的东西。RTTI 在这里错误的用法是“系统中的每种类型”都进行了测试，而不是仅测试一种类型或者一个类型子集。正如大家以后会看到的那样，在测试每一种类型时可换用其他方式来运用多形性特征。但假如以这种形式过多地使用 RTTI，而且又在自己的系统里添加了一种新类型，很容易就会忘记在程序里作出适当的改动，从而埋下以后难以发现的 Bug。因此，在这种情况下避免使用 RTTI 是很有必要的，这并不仅仅是为了表面好看——也是为了产生更易维护的代码。

16.5 抽象的应用

走到这一步，接下来该考虑一下设计方案剩下的部分了——在哪里使用类？既然归类到垃圾箱的办法非常不雅且过于暴露，为什么不隔离那个过程，把它隐藏到一个类里呢？这就是著名的“如果必须做不雅的事情，至少应将其本地化到一个类里”规则。看起来就象下面这样：

932 页图

现在，只要一种新类型的 Trash 加入方法，对 TrashSorter 对象的初始化就必

须变动。可以想象，TrashSorter 类看起来应该象下面这个样子：

```
class TrashSorter extends Vector {
void sort(Trash t) { /* ... */ }
}
```

也就是说，TrashSorter 是由一系列句柄构成的 Vector（系列），而那些句柄指向的又是由 Trash 句柄构成的 Vector；利用 addElement()，可以安装新的 TrashSorter，如下所示：

```
TrashSorter ts = new TrashSorter();
ts.addElement(new Vector());
```

但是现在，sort()却成为一个问题。用静态方式编码的方法如何应付一种新类型加入的事实呢？为解决这个问题，必须从 sort()里将类型信息删除，使其需要做的所有事情就是调用一个通用方法，用它照料涉及类型处理的所有细节。这当然是对一个动态绑定方法进行描述的另一种方式。所以 sort()会在序列中简单地遍历，并为每个 Vector 都调用一个动态绑定方法。由于这个方法的任务是收集它感兴趣的垃圾片，所以称之为 grab(Trash)。结构现在变成了下面这样：

933 页图

其中，TrashSorter 需要调用每个 grab()方法；然后根据当前 Vector 容纳的是什么类型，会获得一个不同的结果。也就是说，Vector 必须留意自己容纳的类型。解决这个问题的传统方法是创建一个基础“Trash bin”（垃圾筒）类，并为希望容纳的每个不同的类型都继承一个新的衍生类。若 Java 有一个参数化的类型机制，那就也许是最直接的方法。但对于这种机制应该为我们构建的各个类，我们不应该进行麻烦的手工编码，以后的“观察”方式提供了一种更好的编码方式。

OOP 设计一条基本的准则是“为状态的变化使用数据成员，为行为的变化使用多性形”。对于容纳 Paper（纸张）的 Vector，以及容纳 Glass（玻璃）的 Vector，大家最开始或许会认为分别用于它们的 grab()方法肯定会产生不同的行为。但具体如何却完全取决于类型，而不是其他什么东西。可将其解释成一种不同的状态，而且由于 Java 有一个类可表示类型（Class），所以可用它判断特定的 Tbin 要容纳什么类型的 Trash。

用于 Tbin 的构建器要求我们为其传递自己选择的一个 Class。这样做可告诉 Vector 它希望容纳的是什么类型。随后，grab()方法用 Class BinType 和 RTTI 来检查我们传递给它的 Trash 对象是否与它希望收集的类型相符。

下面列出完整的解决方案。设定为注释的编号（如*1*）便于大家对照程序后面列出的说明。

934-935 页程序

(1) TbinList 容纳一系列 Tbin 句柄，所以在查找与我们传递给它的 Trash 对象相符的情况时，sort()能通过 Tbin 继承。

(2) sortBin()允许我们将一个完整的 Tbin 传递进去，而且它会在 Tbin 里遍历，挑选出每种 Trash，并将其归类到特定的 Tbin 中。请注意这些代码的通用性：新类型加入时，它本身不需要任何改动。只要新类型加入（或发生其他事件）时大量代码都不需要变化，就表明我们设计的是一个容易扩展的系统。

(3) 现在可以体会添加新类型有多么容易了。为支持添加，只需要改动几行代码。如确实有必要，甚至可以进一步地改进设计，使更多的代码都保持“固定”。

(4) 一个方法调用使 bin 的内容归类到对应的、特定类型的垃圾筒里。

16.6 多重派遣

上述设计方案肯定是令人满意的。系统内新类型的加入涉及添加或修改不同的类，但没有必要在系统内对代码作大范围的改动。除此以外，RTTI 并不象它在 RecycleA.java 里那样被不当地使用。然而，我们仍然有可能更深入一步，以最“纯”的角度来看待 RTTI，考虑如何在垃圾分类系统中将它完全消灭。

为达到这个目标，首先必须认识到：对所有与不同类型有特殊关联的活动来说——比如侦测一种垃圾的具体类型，并把它置入适当的垃圾筒里——这些活动都应当通过多形性以及动态绑定加以控制。

以前的例子都是先按类型排序，再对属于某种特殊类型的一系列元素进行操作。现在一旦需要操作特定的类型，就请先停下来想一想。事实上，多形性（动态绑定的方法调用）整个的宗旨就是帮我们管理与不同类型有特殊关联的信息。既然如此，为什么还要自己去检查类型呢？

答案在于大家或许不以为然的一个道理：Java 只执行单一派遣。也就是说，假如对多个类型未知的对象执行某项操作，Java 只会为那些类型中的一种调用动态绑定机制。这当然不能解决问题，所以最后不得不人工判断某些类型，才能有效地产生自己的动态绑定行为。

为解决这个缺陷，我们需要用到“多重派遣”机制，这意味着需要建立一个配置，使单一方法调用能产生多个动态方法调用，从而在一次处理过程中正确判断出多种类型。为达到这个要求，需要对多个类型结构进行操作：每一次派遣都需要一个类型结构。下面的例子将对两个结构进行操作：现有的 Trash 系列以及由垃圾筒（Trash Bin）的类型构成的一个系列——不同的垃圾或废品将置入这些筒内。第二个分级结构并非绝对显然的。在这种情况下，我们需要人为地创建它，以执行多重派遣（由于本例只涉及两次派遣，所以称为“双重派遣”）。

16.6.1 实现双重派遣

记住多形性只能通过方法调用才能表现出来，所以假如想使双重派遣正确进行，必须执行两个方法调用：在每种结构中都用一个来判断其中的类型。在 Trash 结构中，将使用一个新的方法调用 addToBin()，它采用的参数是由 TypeBin 构成的一个数组。那个方法将在数组中遍历，尝试将自己加入适当的垃圾筒，这里正是双重派遣发生的地方。

937 页图

新建立的分级结构是 TypeBin，其中包含了它自己的一个方法，名为 add()，而且也应用了多形性。但要注意一个新特点：add()已进行了“过载”处理，可接受不同的垃圾类型作为参数。因此，双重满足机制的一个关键点是它也要涉及到过载。

程序的重新设计也带来了一个问题：现在的基础类 Trash 必须包含一个 addToBin()方法。为解决这个问题，一个最直接的办法是复制所有代码，并修改基础类。然而，假如没有对源码的控制权，那么还有另一个办法可以考虑：将

addToBin()方法置入一个接口内部，保持 Trash 不变，并继承新的、特殊的类型 Aluminum，Paper，Glass 以及 Cardboard。我们在这里准备采取后一个办法。

这个设计方案中用到的大多数类都必须设为 public（公用）属性，所以它们放置于自己的类内。下面列出接口代码：

938 页上程序

在 Aluminum，Paper，Glass 以及 Cardboard 每个特定的子类型内，都会实现接口 TypeBinMember 的 addToBin()方法，但每种情况下使用的代码“似乎”都是完全一样的：

938-940 页程序

每个 addToBin()内的代码会为数组中的每个 TypeBin 对象调用 add()。但请注意参数：this。对 Trash 的每个子类来说，this 的类型都是不同的，所以不能认为代码“完全”一样——尽管以后在 Java 里加入参数化类型机制后便可认为一样。这是双重派遣的第一个部分，因为一旦进入这个方法内部，便可知道到底是 Aluminum，Paper，还是其他什么垃圾类型。在对 add()的调用过程中，这种信息是通过 this 的类型传递的。编译器会分析出对 add()正确的过载版本的调用。但由于 tb[i]会产生指向基础类型 TypeBin 的一个句柄，所以最终会调用一个不同的方法——具体什么方法取决于当前选择的 TypeBin 的类型。那就是第二次派遣。

下面是 TypeBin 的基础类：

940 页程序

可以看到，过载的 add()方法全都会返回 false。如果未在衍生类里对方法进行过载，它就会一直返回 false，而且调用者（目前是 addToBin()）会认为当前 Trash 对象尚未成功加入一个集合，所以会继续查找正确的集合。

在 TypeBin 的每一个子类中，都只有一个过载的方法会被过载——具体取决于准备创建的是什么垃圾筒类型。举个例子来说，CardboardBin 会过载 add(DDCardboard)。过载的方法会将垃圾对象加入它的集合，并返回 true。而 CardboardBin 中剩余的所有 add()方法都会继续返回 false，因为它们尚未过载。事实上，假如在这里采用了参数化类型机制，Java 代码的自动创建就要方便得多（使用 C++的“模板”，我们不必费事地为子类编码，或者将 addToBin()方法置入 Trash 里；Java 在这方面尚有待改进）。

由于对这个例子来说，垃圾的类型已经定制并置入一个不同的目录，所以需要用一个不同的垃圾数据文件令其运转起来。下面是一个示范性的 DDTrash.dat：

941-942 页程序

下面列出程序剩余的部分：

942-943 页程序

其中，TrashBinSet 封装了各种不同类型的 TypeBin，同时还有 sortIntoBins()方法。所有双重派遣事件都会在那个方法里发生。可以看到，一旦设置好结构，再归类成各种 TypeBin 的工作就变得十分简单了。除此以外，两个动态方法调用的效率可能也比其他排序方法高一些。

注意这个系统的方便性主要体现在 main()中，同时还要注意到任何特定的类型信息在 main()中都是完全独立的。只与 Trash 基础类接口通信的其他所有方法都不会受到 Trash 类中发生的改变的干扰。

添加新类型需要作出的改动是完全孤立的：我们随同 addToBin()方法继承 Trash 的新类型，然后继承一个新的 TypeBin（这实际只是一个副本，可以简单地编辑），最后将一种新类型加入 TrashBinSet 的集合初化化过程。

16.7 访问器范式

接下来，让我们思考如何将具有完全不同目标的一个设计范式应用到垃圾归类系统。

对这个范式，我们不再关心在系统中加入新型 Trash 时的优化。事实上，这个范式使新型 Trash 的添加显得更加复杂。假定我们有一个基本类结构，它是固定不变的；它或许来自另一个开发者或公司，我们无权对那个结构进行任何修改。然而，我们又希望在那个结构里加入新的多形性方法。这意味着我们一般必须在基础类的接口里添加某些东西。因此，我们目前面临的困境是一方面需要向基础类添加方法，另一方面又不能改动基础类。怎样解决这个问题呢？

“访问器”（Visitor）范式使我们能扩展基本类型的接口，方法是创建类型为 Visitor 的一个独立的类结构，对以后需对基本类型采取的操作进行虚拟。基本类型的任务就是简单地“接收”访问器，然后调用访问器的动态绑定方法。看起来就象下面这样：

945 页图

现在，假如 v 是一个指向 Aluminum（铝制品）的 Visitable 句柄，那么下述代码：

```
PriceVisitor pv = new PriceVisitor();
v.accept(pv);
```

会造成两个多形性方法调用：第一个会选择 accept()的 Aluminum 版本；第二个则在 accept()里——用基础类 Visitor 句柄 v 动态调用 visit()的特定版本时。

这种配置意味着可采取 Visitor 的新子类的形式将新的功能添加到系统里，没必要接触 Trash 结构。这就是“访问器”范式最主要的优点：可为一个类结构添加新的多形性功能，同时不必改动结构——只要安装好了 accept()方法。注意这个优点在这儿是有用的，但并不一定是我们在任何情况下的首选方案。所以在最开始的时候，就要判断这到底是不是自己需要的方案。

现在注意一件没有做成的事情：访问器方案防止了从主控 Trash 序列向单独类型序列的归类。所以我们可将所有东西都留在单主控序列中，只需用适当的访问器通过那个序列传递，即可达到希望的目标。尽管这似乎并非访问器范式的本意，但确实让我们达到了很希望达到的一个目标（避免使用 RTTI）。

访问器范式中的双生派遣负责同时判断 Trash 以及 Visitor 的类型。在下面的例子中，大家可看到 Visitor 的两种实现方式：PriceVisitor 用于判断总计及价格，

而 WeightVisitor 用于跟踪重量。

可以看到，所有这些都是用回收程序一个新的、改进过的版本实现的。而且和 DoubleDispatch.java 一样，Trash 类被保持孤立，并创建一个新接口来添加 accept()方法：

946 页上程序

Aluminum，Paper，Glass 以及 Cardboard 的子类型实现了 accept()方法：

946-947 页程序

由于 Visitor 基础类没有什么需要实在的东西，可将其创建成一个接口：

947-948 页程序

程序剩余的部分将创建特定的 Visitor 类型，并通过一个 Trash 对象列表发送它们：

948-951 页程序

注意 main()的形状已再次发生了变化。现在只有一个垃圾（Trash）筒。两个 Visitor 对象被接收到序列中的每个元素内，它们会完成自己份内的工作。Visitor 跟踪它们自己的内部数据，计算出总重和价格。

最好，将东西从序列中取出的时候，除了不可避免地向 Trash 造型以外，再没有运行期的类型验证。若在 Java 里实现了参数化类型，甚至那个造型操作也可以避免。

对比之前介绍过的双重派遣方案，区分这两种方案的一个办法是：在双重派遣方案中，每个子类创建时只会过载其中的一个过载方法，即 add()。而在这里，每个过载的 visit()方法都必须在 Visitor 的每个子类中进行过载。

1. 更多的结合？

这里还有其他许多代码，Trash 结构和 Visitor 结构之间存在着明显的“结合”（Coupling）关系。然而，在它们所代表的类集内部，也存在着高度的凝聚力：都只做一件事情（Trash 描述垃圾或废品，而 Visitor 描述对垃圾采取什么行动）。作为一套优秀的设计方案，这无疑是个良好的开端。当然就目前的情况来说，只有在我们添加新的 Visitor 类型时才能体会到它的好处。但在添加新类型的 Trash 时，它却显得有些碍手碍脚。

类与类之间低度的结合与类内高度的凝聚无疑是一个重要的设计目标。但只要稍不留神，就可能妨碍我们得到一个本该更出色的设计。从表面看，有些类不可避免地相互间存在着一些“亲密”关系。这种关系通常是成对发生的，可以叫作“对联”（Couplet）——比如集合和继承器（Enumeration）。前面的 Trash-Visitor 对似乎也是这样的一种“对联”。

16.8 RTTI 真的有害吗

本章的各种设计方案都在努力避免使用 RTTI，这或许会给大家留下“RTTI 有害”的印象（还记得可怜的 goto 吗，由于给人印象不佳，根本就没有放到 Java 里来）。但实际情况并非绝对如此。正确地说，应该是 RTTI 使用不当才“有害”。我们之所以想避免 RTTI 的使用，是由于它的错误运用会造成扩展性受到损害。而我们事前提出的目标就是能向系统自由加入新类型，同时保证对周围的代码造成尽可能小的影响。由于 RTTI 常被滥用（让它查找系统中的每一种类型），会造成代码的扩展能力大打折扣——添加一种新类型时，必须找出使用了 RTTI 的所有代码。即使仅遗漏了其中的一个，也不能从编译器那里得到任何帮助。

然而，RTTI 本身并不会自动产生非扩展性的代码。让我们再来看一看前面提到的垃圾回收例子。这一次准备引入一种新工具，我把它叫作 TypeMap。其中包含了一个 Hashtable（散列表），其中容纳了多个 Vector，但接口非常简单：可以添加（add()）一个新对象，可以获得（get()）一个 Vector，其中包含了属于某种特定类型的所有对象。对于这个包含的散列表，它的关键在于对应的 Vector 里的类型。这种设计方案的优点（根据 Larry O'Brien 的建议）是在遇到一种新类型的时候，TypeMap 会动态加入一种新类型。所以不管什么时候，只要将一种新类型加入系统（即使在运行期间添加），它也会正确无误地得以接受。

我们的例子同样建立在 c16.Trash 这个“包”（Package）内的 Trash 类型结构的基础上（而且那儿使用的 Trash.dat 文件可以照搬到这里来）。

952-953 页程序

尽管功能很强，但对 TypeMap 的定义是非常简单的。它只是包含了一个散列表，同时 add()负担了大部分的工作。添加一个新类型时，那种类型的 Class 对象的句柄会被提取出来。随后，利用这个句柄判断容纳了那类对象的一个 Vector 是否已存在于散列表中。如答案是肯定的，就提取出那个 Vector，并将对象加入其中；反之，就将 Class 对象及新 Vector 作为一个“键－值”对加入。

利用 keys()，可以得到对所有 Class 对象的一个“枚举”（Enumeration），而且可用 get()，可通过 Class 对象获取对应的 Vector。

filler()方法非常有趣，因为它利用了 ParseTrash.fillBin()的设计——不仅能尝试填充一个 Vector，也能用它的 addTrash()方法试着填充实现了 Fillable（可填充）接口的任何东西。filter()需要做的全部事情就是将一个句柄返回给实现了 Fillable 的一个接口，然后将这个句柄作为参数传递给 fillBin()，就象下面这样：

```
ParseTrash.fillBin("Trash.dat", bin.filler());
```

为产生这个句柄，我们采用了一个“匿名内部类”（已在第 7 章讲述）。由于根本不需要用一个已命名的类来实现 Fillable，只需要属于那个类的一个对象的句柄即可，所以这里使用匿名内部类是非常恰当的。

对这个设计，要注意的一个地方是尽管没有设计成对归类加以控制，但在 fillBin()每次进行归类的时候，都会将一个 Trash 对象插入 bin。

通过前面那些例子的学习，DynaTrash 类的大多数部分都应当非常熟悉了。这一次，我们不再将新的 Trash 对象置入类型 Vector 的一个 bin 内。由于 bin 的类型为 TypeMap，所以将垃圾（Trash）丢进垃圾筒（Bin）的时候，TypeMap 的内部归类机制会立即进行适当的分类。在 TypeMap 里遍历并对每个独立的 Vector 进行操作，这是一件相当简单的事情：

就象大家看到的那样，新类型向系统的加入根本不会影响到这些代码，亦不会影响 TypeMap 中的代码。这显然是解决问题最圆满的方案。尽管它确实严重依赖 RTTI，但请注意散列表中的每个键－值对都只查找一种类型。除此以外，在我们增加一种新类型的时候，不会陷入“忘记”向系统加入正确代码的尴尬境地，因为根本就没有什么代码需要添加。

16.9 总结

从表面看，由于象 TrashVisitor.java 这样的设计包含了比早期设计数量更多的代码，所以会留下效率不高的印象。试图用各种设计方案达到什么目的应该是我们考虑的重点。设计范式特别适合“将发生变化的东西与保持不变的东西隔离开”。而“发生变化的东西”可以代表许多种变化。之所以发生变化，可能是由于程序进入一个新环境，或者由于当前环境的一些东西发生了变化（例如“用户希望在屏幕上当前显示的图示中添加一种新的几何形状”）。或者就象本章描述的那样，变化可能是对代码主体的不断改进。尽管废品分类以前的例子强调了新型 Trash 向系统的加入，但 TrashVisitor.java 允许我们方便地添加新功能，同时不会对 Trash 结构造成干扰。TrashVisitor.java 里确实多出了许多代码，但在 Visitor 里添加新功能只需要极小的代价。如果经常都要进行此类活动，那么多一些代码也是值得的。

变化序列的发现并非一件平常事；在程序的初始设计出台以前，那些分析家一般不可能预测到这种变化。除非进入项目设计的后期，否则一些必要的信息是不会显露出来的：有时只有进入设计或最终实现阶段，才能体会到对自己系统一个更深入或更不易察觉需要。添加新类型时（这是“回收”例子最主要的一个重点），可能会意识到只有自己进入维护阶段，而且开始扩充系统时，才需要一个特定的继承结构。

通过设计范式的学习，大家可体会到最重要的一件事情就是本书一直宣扬的一个观点——多形性是 OOP（面向对象程序设计）的全部——已发生了彻底的改变。换句话说，很难“获得”多形性；而一旦获得，就需要尝试将自己的所有设计都造型到一个特定的模子里去。

设计范式要表明的观点是“OOP 并不仅仅同多形性有关”。应当与 OOP 有关的是“将发生变化的东西同保持不变的东西分隔开来”。多形性是达到这一目的的特别重要的手段。而且假如编程语言直接支持多形性，那么它就显得尤其有用（由于直接支持，所以不必自己动手编写，从而节省大量的精力和时间）。但设计范式向我们揭示的却是达到基本目标的另一些常规途径。而且一旦熟悉并掌握了它的用法，就会发现自己可以做出更有创新性的设计。

由于《Design Patterns》这本书对程序员造成了如此重要的影响，所以他们纷纷开始寻找其他范式。随着的时间的推移，这类范式必然会越来越多。JimCoplien（http://www.bell-labs.com/~cope 主页作者）向我们推荐了这样的一些站点，上面有许多很有价值的范式说明：

http://st-www.cs.uiuc.edu/users/patterns

http://c2.com/cgi/wiki

http://c2.com/ppr

http://www.bell-labs.com/people/cope/Patterns/Process/index.html

http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns
http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic
http://www.cs.wustl.edu/~schmidt/patterns.html
http://www.espinc.com/patterns/overview.html

同时请留意每年都要召开一届权威性的设计范式会议，名为 PLOP。会议会出版许多学术论文，第三届已在 1997 年底召开过了，会议所有资料均由 Addison-Wesley 出版。

16.10 练习

(1) 将 SingletonPattern.java 作为起点，创建一个类，用它管理自己固定数量的对象。

(2) 为 TrashVisitor.java 添加一个名为 Plastic（塑料）的类。

(3) 为 DynaTrash.java 同样添加一个 Plastic（塑料）类。