第 6 章 类再生

“Java 引人注目的一项特性是代码的重复使用或者再生。但最具革命意义的是，除代码的复制和修改以外，我们还能做多得多的其他事情。”

在象 C 那样的程序化语言里，代码的重复使用早已可行，但效果不是特别显著。与 Java 的其他地方一样，这个方案解决的也是与类有关的问题。我们通过创建新类来重复使用代码，但却用不着重新创建，可以直接使用别人已建好并调试好的现成类。

但这样做必须保证不会干扰原有的代码。在这一章里，我们将介绍两个达到这一目标的方法。第一个最简单：在新类里简单地创建原有类的对象。我们把这种方法叫作**“合成”**，因为新类由现有类的对象合并而成。我们只是简单地重复利用代码的功能，而不是采用它的形式。

第二种方法则显得稍微有些技巧。它创建一个新类，将其作为现有类的一个“类型”。我们可以原样采取现有类的形式，并在其中加入新代码，同时不会对现有的类产生影响。这种魔术般的行为叫作“**继承**”（Inheritance），涉及的大多数工作都是由编译器完成的。对于面向对象的程序设计，“继承”是最重要的基础概念之一。它对我们下一章要讲述的内容会产生一些额外的影响。

对于合成与继承这两种方法，大多数语法和行为都是类似的（因为它们都要根据现有的类型生成新类型）。在本章，我们将深入学习这些代码再生或者重复使用的机制。

6.1 合成的语法

就以前的学习情况来看，事实上已进行了多次“合成”操作。为进行合成，

我们只需在新类里简单地置入对象句柄即可。举个例子来说，假定需要在一个对象里容纳几个 String 对象、两种基本数据类型以及属于另一个类的一个对象。对于非基本类型的对象来说，只需将句柄置于新类即可；而对于基本数据类型来说，则需在自己的类中定义它们。如下所示（若执行该程序时有麻烦，请参见第 3 章 3.1.2 小节“赋值”）：

218-219 页程序

```
//: c06:SprinklerSystem.java
// Composition for code reuse.

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  WaterSource source;
  int i;
  float f;
  void print() {
    System.out.println("valve1 = " + valve1);
    System.out.println("valve2 = " + valve2);
    System.out.println("valve3 = " + valve3);
    System.out.println("valve4 = " + valve4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("source = " + source);
  }
  public static void main(String[] args) {
    SprinklerSystem x = new SprinklerSystem();
    x.print();
  }
} ///:~
```

WaterSource 内定义的一个方法是比较特别的：toString()。大家不久就会知道，每种非基本类型的对象都有一个 toString()方法。若编译器本来希望一个 String，但却获得某个这样的对象，就会调用这个方法。所以在下面这个表达式中：

```
System.out.println("source = " + source) ;
```

编译器会发现我们试图向一个 WaterSource 添加一个 String 对象（"source ="）。

这对它来说是不可接受的，因为我们只能将一个字串“添加”到另一个字串，所以它会说：“我要调用 toString()，把 source 转换成字串！”经这样处理后，它就能编译两个字串，并将结果字串传递给一个 System.out.println()。每次随同自己创建的一个类允许这种行为的时候，都只需要写一个 toString()方法。

如果不深究，可能会草率地认为编译器会为上述代码中的每个句柄都自动构造对象（由于 Java 的安全和谨慎的形象）。例如，可能以为它会为 WaterSource 调用默认构建器，以便初始化 source。打印语句的输出事实上是：

219 页下程序

valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = null

在类内作为字段使用的基本数据会初始化成零，就象第 2 章指出的那样。但对象句柄会初始化成 null。而且假若试图为它们中的任何一个调用方法，就会产生一次“违例”。这种结果实际是相当好的（而且很有用），我们可在不丢弃一次违例的前提下，仍然把它们打印出来。

编译器并不只是为每个句柄创建一个默认对象，因为那样会在许多情况下招致不必要的开销。如希望句柄得到初始化，可在下面这些地方进行：

(1) 在对象定义的时候。这意味着它们在构建器调用之前肯定能得到初始化。

(2) 在那个类的构建器中。

(3) 紧靠在要求实际使用那个对象之前。这样做可减少不必要的开销——假如对象并不需要创建的话。

下面向大家展示了所有这三种方法：

220-221 页程序

请注意在 Bath 构建器中，在所有初始化开始之前执行了一个语句。如果不在定义时进行初始化，仍然不能保证能在将一条消息发给一个对象句柄之前会执行任何初始化——除非出现不可避免的运行期违例。

下面是该程序的输出：

221 页中程序

调用 print()时，它会填充 s4，使所有字段在使用之前都获得正确的初始化。

6.2 继承的语法

继承与 Java（以及其他 OOP 语言）非常紧密地结合在一起。我们早在第 1 章就为大家引入了继承的概念，并在那章之后到本章之前的各章里不时用到，因

为一些特殊的场合要求必须使用继承。除此以外，创建一个类时肯定会进行继承，因为若非如此，会从 Java 的标准根类 Object 中继承。

用于合成的语法是非常简单且直观的。但为了进行继承，必须采用一种全然不同的形式。需要继承的时候，我们会说："这个新类和那个旧类差不多。"为了在代码里表面这一观念，需要给出类名。但在类主体的起始花括号之前，需要放置一个关键字 extends，在后面跟随"基础类"的名字。若采取这种做法，就可自动获得基础类的所有数据成员以及方法。下面是一个例子：

222 页程序

这个例子向大家展示了大量特性。首先，在 Cleanser append()方法里，字串同一个 s 连接起来。这是用"+="运算符实现的。同"+"一样，"+="被 Java 用于对字串进行"过载"处理。

其次，无论 Cleanser 还是 Detergent 都包含了一个 main()方法。我们可为自己的每个类都创建一个 main()。通常建议大家象这样进行编写代码，使自己的测试代码能够封装到类内。即便在程序中含有数量众多的类，但对于在命令行请求的 public 类，只有 main()才会得到调用。所以在这种情况下，当我们使用"java Detergent"的时候，调用的是 Degergent.main()——即使 Cleanser 并非一个 public 类。采用这种将 main()置入每个类的做法，可方便地为每个类都进行单元测试。而且在完成测试以后，毋需将 main()删去；可把它保留下来，用于以后的测试。

在这里，大家可看到 Deteregent.main()对 Cleanser.main()的调用是明确进行的。

需要着重强调的是 Cleanser 中的所有类都是 public 属性。请记住，倘若省略所有访问指示符，则成员默认为"友好的"。这样一来，就只允许对包成员进行访问。在这个包内，任何人都可使用那些没有访问指示符的方法。例如，Detergent 将不会遇到任何麻烦。然而，假设来自另外某个包的类准备继承 Cleanser，它就只能访问那些 public 成员。所以在计划继承的时候，一个比较好的规则是将所有字段都设为 private，并将所有方法都设为 public（protected 成员也允许衍生出来的类访问它；以后还会深入探讨这一问题）。当然，在一些特殊的场合，我们仍然必须作出一些调整，但这并不是一个好的做法。

注意 Cleanser 在它的接口中含有一系列方法：append()，dilute()，apply()，scrub()以及 print()。由于 Detergent 是从 Cleanser 衍生出来的（通过 extends 关键字），所以它会自动获得接口内的所有这些方法——即使我们在 Detergent 里并未看到对它们的明确定义。这样一来，就可将继承想象成"对接口的重复利用"或者"接口的再生"（以后的实施细节可以自由设置，但那并非我们强调的重点）。

正如在 scrub()里看到的那样，可以获得在基础类里定义的一个方法，并对其进行修改。在这种情况下，我们通常想在新版本里调用来自基础类的方法。但在 scrub()里，不可只是简单地发出对 scrub()的调用。那样便造成了递归调用，我们不愿看到这一情况。为解决这个问题，Java 提供了一个 super 关键字，它引用当前类已从中继承的一个"超类"（Superclass）。所以表达式 super.scrub()调用的是方法 scrub()的基础类版本。

进行继承时，我们并不限于只能使用基础类的方法。亦可在衍生出来的类里加入自己的新方法。这时采取的做法与在普通类里添加其他任何方法是完全一样的：只需简单地定义它即可。extends 关键字提醒我们准备将新方法加入基础类

的接口里，对其进行“扩展”。foam()便是这种做法的一个产物。

在 Detergent.main()里，我们可看到对于 Detergent 对象，可调用 Cleanser 以及 Detergent 内所有可用的方法（如 foam()）。

6.2.1 初始化基础类

由于这儿涉及到两个类——基础类及衍生类，而不再是以前的一个，所以在想象衍生类的结果对象时，可能会产生一些迷惑。从外部看，似乎新类拥有与基础类相同的接口，而且可包含一些额外的方法和字段。但继承并非仅仅简单地复制基础类的接口了事。创建衍生类的一个对象时，它在其中包含了基础类的一个“子对象”。这个子对象就象我们根据基础类本身创建了它的一个对象。从外部看，基础类的子对象已封装到衍生类的对象里了。

当然，基础类子对象应该正确地初始化，而且只有一种方法能保证这一点：在构建器中执行初始化，通过调用基础类构建器，后者有足够的能力和权限来执行对基础类的初始化。在衍生类的构建器中，Java 会自动插入对基础类构建器的调用。下面这个例子向大家展示了对这种三级继承的应用：

224-225 页程序

该程序的输出显示了自动调用：

```
Art constructor
Drawing constructor
Cartoon constructor
```

可以看出，构建是在基础类的“外部”进行的，所以基础类会在衍生类访问它之前得到正确的初始化。

即使没有为 Cartoon()创建一个构建器，编译器也会为我们自动合成一个默认构建器，并发出对基础类构建器的调用。

1. 含有自变量的构建器

上述例子有自己默认的构建器；也就是说，它们不含任何自变量。编译器可以很容易地调用它们，因为不存在具体传递什么自变量的问题。如果类没有默认的自变量，或者想调用含有一个自变量的某个基础类构建器，必须明确地编写对基础类的调用代码。这是用 super 关键字以及适当的自变量列表实现的，如下所示：

225-226 页程序

如果不调用 BoardGames()内的基础类构建器，编译器就会报告自己找不到 Games()形式的一个构建器。除此以外，在衍生类构建器中，对基础类构建器的调用是必须做的第一件事情（如操作失当，编译器会向我们指出）。

2. 捕获基本构建器的违例

正如刚才指出的那样，编译器会强迫我们在衍生类构建器的主体中首先设置

对基础类构建器的调用。这意味着在它之前不能出现任何东西。正如大家在第 9 章会看到的那样，这同时也会防止衍生类构建器捕获来自一个基础类的任何违例事件。显然，这有时会为我们造成不便。

6.3 合成与继承的结合

许多时候都要求将合成与继承两种技术结合起来使用。下面这个例子展示了如何同时采用继承与合成技术，从而创建一个更复杂的类，同时进行必要的构建器初始化工作：

226-228 页程序

尽管编译器会强迫我们对基础类进行初始化，并要求我们在构建器最开头做这一工作，但它并不会监视我们是否正确初始化了成员对象。所以对此必须特别加以留意。

6.3.1 确保正确的清除

Java 不具备象 C++的“破坏器”那样的概念。在 C++中，一旦破坏（清除）一个对象，就会自动调用破坏器方法。之所以将其省略，大概是由于在 Java 中只需简单地忘记对象，不需强行破坏它们。垃圾收集器会在必要的时候自动回收内存。

垃圾收集器大多数时候都能很好地工作，但在某些情况下，我们的类可能在自己的存在时期采取一些行动，而这些行动要求必须进行明确的清除工作。正如第 4 章已经指出的那样，我们并不知道垃圾收集器什么时候才会显身，或者说不知它何时会调用。所以一旦希望为一个类清除什么东西，必须写一个特别的方法，明确、专门地来做这件事情。同时，还要让客户程序员知道他们必须调用这个方法。而在所有这一切的后面，就如第 9 章（违例控制）要详细解释的那样，必须将这样的清除代码置于一个 finally 从句中，从而防范任何可能出现的违例事件。

下面介绍的是一个计算机辅助设计系统的例子，它能在屏幕上描绘图形：

229-230 页程序

这个系统中的所有东西都属于某种 Shape（几何形状）。Shape 本身是一种 Object（对象），因为它是从根类明确继承的。每个类都重新定义了 Shape 的 cleanup()方法，同时还要用 super 调用那个方法的基础类版本。尽管对象存在期间调用的所有方法都可负责做一些要求清除的工作，但对于特定的 Shape 类——Circle（圆）、Triangle（三角形）以及 Line（直线），它们都拥有自己的构建器，能完成“作图”（draw）任务。每个类都有它们自己的 cleanup()方法，用于将非内存的东西恢复回对象存在之前的景象。

在 main()中，可看到两个新关键字：try 和 finally。我们要到第 9 章才会向大家正式引荐它们。其中，try 关键字指出后面跟随的块（由花括号定界）是一个“警戒区”。也就是说，它会受到特别的待遇。其中一种待遇就是：该警戒区后面跟随的 finally 从句的代码肯定会得以执行——不管 try 块到底存不存在（通过违例控制技术，try 块可有多种不寻常的应用）。在这里，finally 从句的意思是“总是为 x 调用 cleanup()，无论会发生什么事情”。这些关键字将在第 9 章进行全面、

完整的解释。

在自己的清除方法中，必须注意对基础类以及成员对象清除方法的调用顺序——假若一个子对象要以另一个为基础。通常，应采取与 C++编译器对它的“破坏器”采取的同样的形式：首先完成与类有关的所有特殊工作（可能要求基础类元素仍然可见），然后调用基础类清除方法，就象这儿演示的那样。

许多情况下，清除可能并不是个问题；只需让垃圾收集器尽它的职责即可。但一旦必须由自己明确清除，就必须特别谨慎，并要求周全的考虑。

1. 垃圾收集的顺序

不能指望自己能确切知道何时会开始垃圾收集。垃圾收集器可能永远不会得到调用。即使得到调用，它也可能以自己愿意的任何顺序回收对象。除此以外，Java 1.0 实现的垃圾收集器机制通常不会调用 finalize()方法。除内存的回收以外，其他任何东西都最好不要依赖垃圾收集器进行回收。若想明确地清除什么，请制作自己的清除方法，而且不要依赖 finalize()。然而正如以前指出的那样，可强迫 Java1.1 调用所有收尾模块（Finalizer）。

6.3.2 名字的隐藏

只有 C++程序员可能才会惊讶于名字的隐藏，因为它的工作原理与在 C++里是完全不同的。如果 Java 基础类有一个方法名被“过载”使用多次，在衍生类里对那个方法名的重新定义就不会隐藏任何基础类的版本。所以无论方法在这一级还是在一个基础类中定义，过载都会生效：

232 页程序

正如下一章会讲到的那样，很少会用与基础类里完全一致的签名和返回类型来覆盖同名的方法，否则会使人感到迷惑（这正是 C++不允许那样做的原因，所以能够防止产生一些不必要的错误）。

6.4 到底选择合成还是继承

无论合成还是继承，都允许我们将子对象置于自己的新类中。大家或许会奇怪两者间的差异，以及到底该如何选择。

如果想利用新类内部一个现有类的特性，而不想使用它的接口，通常应选择合成。也就是说，我们可嵌入一个对象，使自己能用它实现新类的特性。但新类的用户会看到我们已定义的接口，而不是来自嵌入对象的接口。考虑到这种效果，我们需在新类里嵌入现有类的 private 对象。

有些时候，我们想让类用户直接访问新类的合成。也就是说，需要将成员对象的属性变为 public。成员对象会将自身隐藏起来，所以这是一种安全的做法。而且在用户知道我们准备合成一系列组件时，接口就更容易理解。car（汽车）对象便是一个很好的例子：

233-234 页程序

由于汽车的装配是故障分析时需要考虑的一项因素（并非只是基础设计简单的一部分），所以有助于客户程序员理解如何使用类，而且类创建者的编程复杂

程度也会大幅度降低。

如选择继承，就需要取得一个现成的类，并制作它的一个特殊版本。通常，这意味着我们准备使用一个常规用途的类，并根据特定的需求对其进行定制。只需稍加想象，就知道自己不能用一个车辆对象来合成一辆汽车——汽车并不“包含”车辆；相反，它“属于”车辆的一种类别。“属于”关系是用继承来表达的，而“包含”关系是用合成来表达的。

6.5 protected

现在我们已理解了继承的概念，protected 这个关键字最后终于有了意义。在理想情况下，private 成员随时都是“私有”的，任何人不得访问。但在实际应用中，经常想把某些东西深深地藏起来，但同时允许访问衍生类的成员。protected 关键字可帮助我们做到这一点。它的意思是“它本身是私有的，但可由从这个类继承的任何东西或者同一个包内的其他任何东西访问”。也就是说，Java 中的 protected 会成为进入“友好”状态。

我们采取的最好的做法是保持成员的 private 状态——无论如何都应保留对基 础的实施细节进行修改的权利。在这一前提下，可通过 protected 方法允许类的继承者进行受到控制的访问：

235 页程序

可以看到，change()拥有对 set()的访问权限，因为它的属性是 protected（受到保护的）。

6.6 累积开发

继承的一个好处是它支持“累积开发”，允许我们引入新的代码，同时不会为现有代码造成错误。这样可将新错误隔离到新代码里。通过从一个现成的、功能性的类继承，同时增添成员新的数据成员及方法（并重新定义现有方法），我们可保持现有代码原封不动（另外有人也许仍在使用它），不会为其引入自己的编程错误。一旦出现错误，就知道它肯定是由于自己的新代码造成的。这样一来，与修改现有代码的主体相比，改正错误所需的时间和精力就可以少很多。

类的隔离效果非常好，这是许多程序员事先没有预料到的。甚至不需要方法的源代码来实现代码的再生。最多只需要导入一个包（这对于继承和合并都是成立的）。

大家要记住这样一个重点：程序开发是一个不断递增或者累积的过程，就象人们学习知识一样。当然可根据要求进行尽可能多的分析，但在一个项目的设计之初，谁都不可能提前获知所有的答案。如果能将自己的项目看作一个有机的、能不断进步的生物，从而不断地发展和改进它，就有望获得更大的成功以及更直接的反馈。

尽管继承是一种非常有用的技术，但在某些情况下，特别是在项目稳定下来以后，仍然需要从新的角度考察自己的类结构，将其收缩成一个更灵活的结构。请记住，继承是对一种特殊关系的表达，意味着“这个新类属于那个旧类的一种类型”。我们的程序不应纠缠于一些细树末节，而应着眼于创建和操作各种类型的对象，用它们表达出来自“问题空间”的一个模型。

6.7 上溯造型

继承最值得注意的地方就是它没有为新类提供方法。继承是对新类和基础类之间的关系的一种表达。可这样总结该关系：“新类属于现有类的一种类型”。

这种表达并不仅仅是对继承的一种形象化解释，继承是直接由语言提供支持的。作为一个例子，大家可考虑一个名为 Instrument 的基础类，它用于表示乐器；另一个衍生类叫作 Wind。由于继承意味着基础类的所有方法亦可在衍生出来的类中使用，所以我们发给基础类的任何消息亦可发给衍生类。若 Instrument 类有一个 play()方法，则 Wind 设备也会有这个方法。这意味着我们能肯定地认为一个 Wind 对象也是 Instrument 的一种类型。下面这个例子揭示出编译器如何提供对这一概念的支持：

236-237 页程序

这个例子中最有趣的无疑是 tune()方法，它能接受一个 Instrument 句柄。但在 Wind.main()中，tune()方法是通过为其赋予一个 Wind 句柄来调用的。由于 Java 对类型检查特别严格，所以大家可能会感到很奇怪，为什么接收一种类型的方法也能接收另一种类型呢？但是，我们一定要认识到一个 Wind 对象也是一个 Instrument 对象。而且对于不在 Wind 中的一个 Instrument（乐器），没有方法可以由 tune()调用。在 tune()中，代码适用于 Instrument 以及从 Instrument 衍生出来的任何东西。在这里，我们将从一个 Wind 句柄转换成一个 Instrument 句柄的行为叫作“上溯造型”。

6.7.1 何谓“上溯造型”？

之所以叫作这个名字，除了有一定的历史原因外，也是由于在传统意义上，类继承图的画法是根位于最顶部，再逐渐向下扩展（当然，可根据自己的习惯用任何方法描绘这种图）。因素，Wind.java 的继承图就象下面这个样子：

237 页图

由于造型的方向是从衍生类到基础类，箭头朝上，所以通常把它叫作“上溯造型”，即 Upcasting。上溯造型肯定是安全的，因为我们是从一个更特殊的类型到一个更常规的类型。换言之，衍生类是基础类的一个超集。它可以包含比基础类更多的方法，但它至少包含了基础类的方法。进行上溯造型的时候，类接口可能出现的唯一一个问题是它可能丢失方法，而不是赢得这些方法。这便是在没有任何明确的造型或者其他特殊标注的情况下，编译器为什么允许上溯造型的原因所在。

也可以执行下溯造型，但这时会面临第 11 章要详细讲述的一种困境。

1. 再论合成与继承

在面向对象的程序设计中，创建和使用代码最可能采取的一种做法是：将数据和方法统一封装到一个类里，并且使用那个类的对象。有些时候，需通过“合成”技术用现成的类来构造新类。而继承是最少见的一种做法。因此，尽管继承在学习 OOP 的过程中得到了大量的强调，但并不意味着应该尽可能地到处使用它。相反，使用它时要特别慎重。只有在清楚知道继承在所有方法中最有效的前提下，才可考虑它。为判断自己到底应该选用合成还是继承，一个最简单的办法

就是考虑是否需要从新类上溯造型回基础类。若必须上溯，就需要继承。但如果不需要上溯造型，就应提醒自己防止继承的滥用。在下一章里（多形性），会向大家介绍必须进行上溯造型的一种场合。但只要记住经常问自己“我真的需要上溯造型吗”，对于合成还是继承的选择就不应该是个太大的问题。

6.8 final 关键字

由于语境（应用环境）不同，final 关键字的含义可能会稍微产生一些差异。但它最一般的意思就是声明“这个东西不能改变”。之所以要禁止改变，可能是考虑到两方面的因素：设计或效率。由于这两个原因颇有些区别，所以也许会造成 final 关键字的误用。

在接下去的小节里，我们将讨论 final 关键字的三种应用场合：数据、方法以及类。

6.8.1 final 数据

许多程序设计语言都有自己的办法告诉编译器某个数据是“常数”。常数主要应用于下述两个方面：

(1) 编译期常数，它永远不会改变

(2) 在运行期初始化的一个值，我们不希望它发生变化

对于编译期的常数，编译器（程序）可将常数值“封装”到需要的计算过程里。也就是说，计算可在编译期间提前执行，从而节省运行时的一些开销。在 Java 中，这些形式的常数必须属于基本数据类型（Primitives），而且要用 final 关键字进行表达。在对这样的一个常数进行定义的时候，必须给出一个值。

无论 static 还是 final 字段，都只能存储一个数据，而且不得改变。

若随同对象句柄使用 final，而不是基本数据类型，它的含义就稍微让人有点儿迷糊了。对于基本数据类型，final 会将值变成一个常数；但对于对象句柄，final 会将句柄变成一个常数。进行声明时，必须将句柄初始化到一个具体的对象。而且永远不能将句柄变成指向另一个对象。然而，对象本身是可以修改的。Java 对此未提供任何手段，可将一个对象直接变成一个常数（但是，我们可自己编写一个类，使其中的对象具有“常数”效果）。这一限制也适用于数组，它也属于对象。

下面是演示 final 字段用法的一个例子：

29-240 页程序

由于 i1 和 I2 都是具有 final 属性的基本数据类型，并含有编译期的值，所以它们除了能作为编译期的常数使用外，在任何导入方式中也不会出现任何不同。I3 是我们体验此类常数定义时更典型的一种方式：public 表示它们可在包外使用；Static 强调它们只有一个；而 final 表明它是一个常数。注意对于含有固定初始化值（即编译期常数）的 fianl static 基本数据类型，它们的名字根据规则要全部采用大写。也要注意 i5 在编译期间是未知的，所以它没有大写。

不能由于某样东西的属性是 final，就认定它的值能在编译时期知道。i4 和 i5 向大家证明了这一点。它们在运行期间使用随机生成的数字。例子的这一部分也向大家揭示出将 final 值设为 static 和非 static 之间的差异。只有当值在运行期间初始化的前提下，这种差异才会揭示出来。因为编译期间的值被编译器认为是相

同的。这种差异可从输出结果中看出：

240-241 页程序

注意对于 fd1 和 fd2 来说，i4 的值是唯一的，但 i5 的值不会由于创建了另一个 FinalData 对象而发生改变。那是因为它的属性是 static，而且在载入时初始化，而非每创建一个对象时初始化。

从 v1 到 v4 的变量向我们揭示出 final 句柄的含义。正如大家在 main()中看到的那样，并不能认为由于 v2 属于 final，所以就不能再改变它的值。然而，我们确实不能再将 v2 绑定到一个新对象，因为它的属性是 final。这便是 final 对于一个句柄的确切含义。我们会发现同样的含义亦适用于数组，后者只不过是另一种类型的句柄而已。将句柄变成 final 看起来似乎不如将基本数据类型变成 final 那么有用。

2. 空白 final

Java 1.1 允许我们创建“空白 final”，它们属于一些特殊的字段。尽管被声明成 final，但却未得到一个初始值。无论在哪种情况下，空白 final 都必须在实际使用前得到正确的初始化。而且编译器会主动保证这一规定得以贯彻。然而，对于 final 关键字的各种应用，空白 final 具有最大的灵活性。举个例子来说，位于类内部的一个 final 字段现在对每个对象都可以有所不同，同时依然保持其“不变”的本质。下面列出一个例子：

241-242 页程序

现在强行要求我们对 final 进行赋值处理——要么在定义字段时使用一个表达 式，要么在每个构建器中。这样就可以确保 final 字段在使用前获得正确的初始化。

3. final 自变量

Java 1.1 允许我们将自变量设成 final 属性，方法是在自变量列表中对它们进行适当的声明。这意味着在一个方法的内部，我们不能改变自变量句柄指向的东西。如下所示：

242 页程序

注意此时仍然能为 final 自变量分配一个 null（空）句柄，同时编译器不会捕获它。这与我们对非 final 自变量采取的操作是一样的。

方法 f()和 g()向我们展示出基本类型的自变量为 final 时会发生什么情况：我们只能读取自变量，不可改变它。

6.8.2 final 方法

之所以要使用 final 方法，可能是出于对两方面理由的考虑。第一个是为方法“上锁”，防止任何继承类改变它的本来含义。设计程序时，若希望一个方法的行为在继承期间保持不变，而且不可被覆盖或改写，就可以采取这种做法。

采用 final 方法的第二个理由是程序执行的效率。将一个方法设成 final 后，编译器就可以把对那个方法的所有调用都置入“嵌入”调用里。只要编译器发现一个 final 方法调用，就会（根据它自己的判断）忽略为执行方法调用机制而采取的常规代码插入方法（将自变量压入堆栈；跳至方法代码并执行它；跳回来；清除堆栈自变量；最后对返回值进行处理）。相反，它会用方法主体内实际代码的一个副本来替换方法调用。这样做可避免方法调用时的系统开销。当然，若方法体积太大，那么程序也会变得臃肿，可能受到到不到嵌入代码所带来的任何性能提升。因为任何提升都被花在方法内部的时间抵消了。Java 编译器能自动侦测这些情况，并颇为“明智”地决定是否嵌入一个 final 方法。然而，最好还是不要完全相信编译器能正确地作出所有判断。通常，只有在方法的代码量非常少，或者想明确禁止方法被覆盖的时候，才应考虑将一个方法设为 final。

类内所有 private 方法都自动成为 final。由于我们不能访问一个 private 方法，所以它绝对不会被其他方法覆盖（若强行这样做，编译器会给出错误提示）。可为一个 private 方法添加 final 指示符，但却不能为那个方法提供任何额外的含义。

6.8.3 final 类

如果说整个类都是 final（在它的定义前冠以 final 关键字），就表明自己不希望从这个类继承，或者不允许其他任何人采取这种操作。换言之，出于这样或那样的原因，我们的类肯定不需要进行任何改变；或者出于安全方面的理由，我们不希望进行子类化（子类处理）。

除此以外，我们或许还考虑到执行效率的问题，并想确保涉及这个类各对象的所有行动都要尽可能地有效。如下所示：

244 页程序

注意数据成员既可以是 final，也可以不是，取决于我们具体选择。应用于 final 的规则同样适用于数据成员，无论类是否被定义成 final。将类定义成 final 后，结果只是禁止进行继承——没有更多的限制。然而，由于它禁止了继承，所以一个 final 类中的所有方法都默认为 final。因为此时再也无法覆盖它们。所以与我们将一个方法明确声明为 final 一样，编译器此时有相同的效率选择。

可为 final 类内的一个方法添加 final 指示符，但这样做没有任何意义。

6.8.4 final 的注意事项

设计一个类时，往往需要考虑是否将一个方法设为 final。可能会觉得使用自己的类时执行效率非常重要，没有人想覆盖自己的方法。这种想法在某些时候是正确的。

但要慎重作出自己的假定。通常，我们很难预测一个类以后会以什么样的形式再生或重复利用。常规用途的类尤其如此。若将一个方法定义成 final，就可能杜绝了在其他程序员的项目中对自己的类进行继承的途径，因为我们根本没有想到它会象那样使用。

标准 Java 库是阐述这一观点的最好例子。其中特别常用的一个类是 Vector。如果我们考虑代码的执行效率，就会发现只有不把任何方法设为 final，才能使其发挥更大的作用。我们很容易就会想到自己应继承和覆盖如此有用的一个类，但它的设计者却否定了我们的想法。但我们至少可以用两个理由来反驳他们。首先，

Stack（堆栈）是从 Vector 继承来的，亦即 Stack“是”一个 Vector，这种说法是不确切的。其次，对于 Vector 许多重要的方法，如 addElement()以及 elementAt()等，它们都变成了 synchronized（同步的）。正如在第 14 章要讲到的那样，这会造成显著的性能开销，可能会把 final 提供的性能改善抵销得一干二净。因此，程序员不得不猜测到底应该在哪里进行优化。在标准库里居然采用了如此笨拙的设计，真不敢想象会在程序员里引发什么样的情绪。

另一个值得注意的是 Hashtable（散列表），它是另一个重要的标准类。该类没有采用任何 final 方法。正如我们在本书其他地方提到的那样，显然一些类的设计人员与其他设计人员有着全然不同的素质（注意比较 Hashtable 极短的方法名与 Vecor 的方法名）。对类库的用户来说，这显然是不应该如此轻易就能看出的。一个产品的设计变得不一致后，会加大用户的工作量。这也从另一个侧面强调了代码设计与检查时需要很强的责任心。

6.9 初始化和类装载

在许多传统语言里，程序都是作为启动过程的一部分一次性载入的。随后进行的是初始化，再是正式执行程序。在这些语言中，必须对初始化过程进行慎重的控制，保证 static 数据的初始化不会带来麻烦。比如在一个 static 数据获得初始化之前，就有另一个 static 数据希望它是一个有效值，那么在 C++中就会造成问题。

Java 则没有这样的问题，因为它采用了不同的装载方法。由于 Java 中的一切东西都是对象，所以许多活动变得更加简单，这个问题便是其中的一例。正如下一章会讲到的那样，每个对象的代码都存在于独立的文件中。除非真的需要代码，否则那个文件是不会载入的。通常，我们可认为除非那个类的一个对象构造完毕，否则代码不会真的载入。由于 static 方法存在一些细微的歧义，所以也能认为“类代码在首次使用的时候载入”。

首次使用的地方也是 static 初始化发生的地方。装载的时候，所有 static 对象和 static 代码块都会按照本来的顺序初始化（亦即它们在类定义代码里写入的顺序）。当然，static 数据只会初始化一次。

6.9.1 继承初始化

我们有必要对整个初始化过程有所认识，其中包括继承，对这个过程中发生的事情有一个整体性的概念。请观察下述代码：

246-247 页程序

该程序的输出如下：

247 页中程序

对 Beetle 运行 Java 时，发生的第一件事情是装载程序到外面找到那个类。在装载过程中，装载程序注意它有一个基础类（即 extends 关键字要表达的意思），所以随之将其载入。无论是否准备生成那个基础类的一个对象，这个过程都会发生（请试着将对象的创建代码当作注释标注出来，自己去证实）。

若基础类含有另一个基础类，则另一个基础类随即也会载入，以此类推。接

下来，会在根基础类（此时是 Insect）执行 static 初始化，再在下一个衍生类执行，以此类推。保证这个顺序是非常关键的，因为衍生类的初始化可能要依赖于对基础类成员的正确初始化。

此时，必要的类已全部装载完毕，所以能够创建对象。首先，这个对象中的所有基本数据类型都会设成它们的默认值，而将对象句柄设为 null。随后会调用基础类构建器。在这种情况下，调用是自动进行的。但也完全可以用 super 来自行指定构建器调用（就象在 Beetle()构建器中的第一个操作一样）。基础类的构建采用与衍生类构建器完全相同的处理过程。基础顺构建器完成以后，实例变量会按本来的顺序得以初始化。最后，执行构建器剩余的主体部分。

6.10 总结

无论继承还是合成，我们都可以在现有类型的基础上创建一个新类型。但在典型情况下，我们通过合成来实现现有类型的“再生”或“重复使用”，将其作为新类型基础实施过程的一部分使用。但如果想实现接口的“再生”，就应使用继承。由于衍生或派生出来的类拥有基础类的接口，所以能够将其“上溯造型”为基础类。对于下一章要讲述的多形性问题，这一点是至关重要的。

尽管继承在面向对象的程序设计中得到了特别的强调，但在实际启动一个设计时，最好还是先考虑采用合成技术。只有在特别必要的时候，才应考虑采用继承技术（下一章还会讲到这个问题）。合成显得更加灵活。但是，通过对自己的成员类型应用一些继承技巧，可在运行期准确改变那些成员对象的类型，由此可改变它们的行为。

尽管对于快速项目开发来说，通过合成和继承实现的代码再生具有很大的帮助作用。但在允许其他程序员完全依赖它之前，一般都希望能重新设计自己的类结构。我们理想的类结构应该是每个类都有自己特定的用途。它们不能过大（如集成的功能太多，则很难实现它的再生），也不能过小（造成不能由自己使用，或者不能增添新功能）。最终实现的类应该能够方便地再生。

6.11 练习

(1) 用默认构建器（空自变量列表）创建两个类：A 和 B，令它们自己声明自己。从 A 继承一个名为 C 的新类，并在 C 内创建一个成员 B。不要为 C 创建一个构建器。创建类 C 的一个对象，并观察结果。

(2) 修改练习 1，使 A 和 B 都有含有自变量的构建器，则不是采用默认构建器。为 C 写一个构建器，并在 C 的构建器中执行所有初始化工作。

(3) 使用文件 Cartoon.java，将 Cartoon 类的构建器代码变成注释内容标注出去。解释会发生什么事情。

(4) 使用文件 Chess.java，将 Chess 类的构建器代码作为注释标注出去。同样解释会发生什么。