第 12 章 传递和返回对象

到目前为止，读者应对对象的“传递”有了一个较为深刻的认识，记住实际传递的只是一个句柄。

在许多程序设计语言中，我们可用语言的“普通”方式到处传递对象，而且大多数时候都不会遇到问题。但有些时候却不得不采取一些非常做法，使得情况突然变得稍微复杂起来（在 C++中则是变得非常复杂）。Java 亦不例外，我们十分有必要准确认识在对象传递和赋值时所发生的一切。这正是本章的宗旨。

若读者是从某些特殊的程序设计环境中转移过来的，那么一般都会问到：“Java 有指针吗？”有些人认为指针的操作很困难，而且十分危险，所以一厢情愿地认为它没有好处。同时由于 Java 有如此好的口碑，所以应该很轻易地免除自己以前编程中的麻烦，其中不可能夹带有指针这样的“危险品”。然而准确地说，Java 是有指针的！事实上，Java 中每个对象（除基本数据类型以外）的标识符都属于指针的一种。但它们的使用受到了严格的限制和防范，不仅编译器对它们有“戒心”，运行期系统也不例外。或者换从另一个角度说，Java 有指针，但没有传统指针的麻烦。我曾一度将这种指针叫做“句柄”，但你可以把它想像成“安全指针”。和预备学校为学生提供的安全剪刀类似——除非特别有意，否则不会伤着自己，只不过有时要慢慢来，要习惯一些沉闷的工作。

12.1 传递句柄

将句柄传递进入一个方法时，指向的仍然是相同的对象。一个简单的实验可以证明这一点（若执行这个程序时有麻烦，请参考第 3 章 3.1.2 小节“赋值”）：

toString 方法会在打印语句里自动调用，而 PassHandles 直接从 Object 继承，没有 toString 的重新定义。因此，这里会采用 toString 的 Object 版本，打印出对象的类，接着是那个对象所在的位置（不是句柄，而是对象的实际存储位置）。输出结果如下：

p inside main(): PassHandles@1653748

h inside f() : PassHandles@1653748

可以看到，无论 p 还是 h 引用的都是同一个对象。这比复制一个新的 PassHandles 对象有效多了，使我们能将一个参数发给一个方法。但这样做也带来了另一个重要的问题。

12.1.1 别名问题

“别名”意味着多个句柄都试图指向同一个对象，就象前面的例子展示的那样。若有人向那个对象里写入一点什么东西，就会产生别名问题。若其他句柄的所有者不希望那个对象改变，恐怕就要失望了。这可用下面这个简单的例子说明：

543 页程序

对下面这行：

Alias1 y = x; // Assign the handle

它会新建一个 Alias1 句柄，但不是把它分配给由 new 创建的一个新鲜对象，而是分配给一个现有的句柄。所以句柄 x 的内容——即对象 x 指向的地址——被分配给 y，所以无论 x 还是 y 都与相同的对象连接起来。这样一来，一旦 x 的 i 在下述语句中增值：

x.i++;

y 的 i 值也必然受到影响。从最终的输出就可以看出：

544 页上程序

此时最直接的一个解决办法就是干脆不这样做：不要有意将多个句柄指向同一个作用域内的同一个对象。这样做可使代码更易理解和调试。然而，一旦准备将句柄作为一个自变量或参数传递——这是 Java 设想的正常方法——别名问题就会自动出现，因为创建的本地句柄可能修改“外部对象”（在方法作用域之外创建的对象）。下面是一个例子：

544 页程序

输出如下：

x: 7

Calling f(x)

x: 8

方法改变了自己的参数——外部对象。一旦遇到这种情况，必须判断它是否合理，用户是否愿意这样，以及是不是会造成问题。

通常，我们调用一个方法是为了产生返回值，或者用它改变为其调用方法的那个对象的状态（方法其实就是我们向那个对象“发一条消息”的方式）。很少需要调用一个方法来处理它的参数；这叫作利用方法的“副作用”（Side Effect）。所以倘若创建一个会修改自己参数的方法，必须向用户明确地指出这一情况，并警告使用那个方法可能会有的后果以及它的潜在威胁。由于存在这些混淆和缺陷，所以应该尽量避免改变参数。

若需在一个方法调用期间修改一个参数，且不打算修改外部参数，就应在自己的方法内部制作一个副本，从而保护那个参数。本章的大多数内容都是围绕这个问题展开的。

12.2 制作本地副本

稍微总结一下：Java 中的所有自变量或参数传递都是通过传递句柄进行的。也就是说，当我们传递“一个对象”时，实际传递的只是指向位于方法外部的那个对象的“一个句柄”。所以一旦要对那个句柄进行任何修改，便相当于修改外部对象。此外：

■参数传递过程中会自动产生别名问题

■不存在本地对象，只有本地句柄

■句柄有自己的作用域，而对象没有

■对象的“存在时间”在 Java 里不是个问题

■没有语言上的支持（如常量）可防止对象被修改（以避免别名的副作用）

若只是从对象中读取信息，而不修改它，传递句柄便是自变量传递中最有效的一种形式。这种做非常恰当；默认的方法一般也是最有效的方法。然而，有时仍需将对象当作“本地的”对待，使我们作出的改变只影响一个本地副本，不会对外面的对象造成影响。许多程序设计语言都支持在方法内自动生成外部对象的一个本地副本（注释①）。尽管 Java 不具备这种能力，但允许我们达到同样的效果。

①：在 C 语言中，通常控制的是少量数据位，默认操作是按值传递。C++也必须遵照这一形式，但按值传递对象并非肯定是一种有效的方式。此外，在 C++中用于支持按值传递的代码也较难编写，是件让人头痛的事情。

12.2.1 按值传递

首先要解决术语的问题，最适合“按值传递”的看起来是自变量。“按值传递”以及它的含义取决于如何理解程序的运行方式。最常见的意思是获得要传递的任何东西的一个本地副本，但这里真正的问题是如何看待自己准备传递的东西。对于“按值传递”的含义，目前存在两种存在明显区别的见解：

(1) Java 按值传递任何东西。若将基本数据类型传递进入一个方法，会明确得到基本数据类型的一个副本。但若将一个句柄传递进入方法，得到的是句柄的副本。所以人们认为“一切”都按值传递。当然，这种说法也有一个前提：句柄肯定也会被传递。但 Java 的设计方案似乎有些超前，允许我们忽略（大多数时候）自己处理的是一个句柄。也就是说，它允许我们将句柄假想成“对象”，因为在发出方法调用时，系统会自动照管两者间的差异。

(2) Java 主要按值传递（无自变量），但对象却是按引用传递的。得到这个结论的前提是句柄只是对象的一个“别名”，所以不考虑传递句柄的问题，而是直

接指出“我准备传递对象”。由于将其传递进入一个方法时没有获得对象的一个本地副本，所以对象显然不是按值传递的。Sun 公司似乎在某种程度上支持这一见解，因为它“保留但未实现”的关键字之一便是 byvalue（按值）。但没人知道那个关键字什么时候可以发挥作用。

尽管存在两种不同的见解，但其间的分歧归根到底是由于对“句柄”的不同解释造成的。我打算在本书剩下的部分里回避这个问题。大家不久就会知道，这个问题争论下去其实是没有意义的——最重要的是理解一个句柄的传递会使调用者的对象发生意外的改变。

12.2.2 克隆对象

若需修改一个对象，同时不想改变调用者的对象，就要制作该对象的一个本地副本。这也是本地副本最常见的一种用途。若决定制作一个本地副本，只需简单地使用 clone()方法即可。Clone 是“克隆”的意思，即制作完全一模一样的副本。这个方法在基础类 Object 中定义成“protected”（受保护）模式。但在希望克隆的任何衍生类中，必须将其覆盖为“public”模式。例如，标准库类 Vector 覆盖了 clone()，所以能为 Vector 调用 clone()，如下所示：

547 页程序

clone()方法产生了一个 Object，后者必须立即重新造型为正确类型。这个例子指出 Vector 的 clone()方法不能自动尝试克隆 Vector 内包含的每个对象——由于别名问题，老的 Vector 和克隆的 Vector 都包含了相同的对象。我们通常把这种情况叫作“简单复制”或者“浅层复制”，因为它只复制了一个对象的“表面”部分。实际对象除包含这个“表面”以外，还包括句柄指向的所有对象，以及那些对象又指向的其他所有对象，由此类推。这便是“对象网”或“对象关系网”的由来。若能复制下所有这张网，便叫作“全面复制”或者“深层复制”。

在输出中可看到浅层复制的结果，注意对 v2 采取的行动也会影响到 v：

548 页上程序

一般来说，由于不敢保证 Vector 里包含的对象是“可以克隆”（注释②）的，所以最好不要试图克隆那些对象。

②：“可以克隆”用英语讲是 cloneable，请留意 Java 库中专门保留了这样的一个关键字。

12.2.3 使类具有克隆能力

尽管克隆方法是在所有类最基本的 Object 中定义的，但克隆仍然不会在每个类里自动进行。这似乎有些不可思议，因为基础类方法在衍生类里是肯定能用的。但 Java 确实有点儿反其道而行之；如果想在一个类里使用克隆方法，唯一的办法就是专门添加一些代码，以便保证克隆的正常进行。

1. 使用 protected 时的技巧

为避免我们创建的每个类都默认具有克隆能力，clone()方法在基础类 Object

里得到了“保留”（设为 protected）。这样造成的后果就是：对那些简单地使用一下这个类的客户程序员来说，他们不会默认地拥有这个方法；其次，我们不能利用指向基础类的一个句柄来调用 clone()（尽管那样做在某些情况下特别有用，比如用多形性的方式克隆一系列对象）。在编译期的时候，这实际是通知我们对象不可克隆的一种方式——而且最奇怪的是，Java 库中的大多数类都不能克隆。因此，假如我们执行下述代码：

```
Integer x = new Integer(l);
x = x.clone();
```

那么在编译期，就有一条讨厌的错误消息弹出，告诉我们不可访问 clone()——因为 Integer 并没有覆盖它，而且它对 protected 版本来说是默认的）。

但是，假若我们是在一个从 Object 衍生出来的类中（所有类都是从 Object 衍生的），就有权调用 Object.clone()，因为它是“protected”，而且我们在一个继承器中。基础类 clone()提供了一个有用的功能——它进行的是对衍生类对象的真正“按位”复制，所以相当于标准的克隆行动。然而，我们随后需要将自己的克隆操作设为 public，否则无法访问。总之，克隆时要注意的两个关键问题是：几乎肯定要调用 super.clone()，以及注意将克隆设为 public。

有时还想在更深层的衍生类中覆盖 clone()，否则就直接使用我们的 clone()（现在已成为 public），而那并不一定是我们所希望的（然而，由于 Object.clone()已制作了实际对象的一个副本，所以也有可能允许这种情况）。protected 的技巧在这里只能用一次：首次从一个不具备克隆能力的类继承，而且想使一个类变成“能够克隆”。而在从我们的类继承的任何场合，clone()方法都是可以使用的，因为 Java 不可能在衍生之后反而缩小方法的访问范围。换言之，一旦对象变得可以克隆，从它衍生的任何东西都是能够克隆的，除非使用特殊的机制（后面讨论）令其“关闭”克隆能力。

2. 实现 Cloneable 接口

为使一个对象的克隆能力功成圆满，还需要做另一件事情：实现 Cloneable 接口。这个接口使人稍觉奇怪，因为它是空的！

```
interface Cloneable {}
```

之所以要实现这个空接口，显然不是因为我们准备上溯造型成一个 Cloneable，以及调用它的某个方法。有些人认为在这里使用接口属于一种“欺骗”行为，因为它使用的特性打的是别的主意，而非原来的意思。Cloneable interface 的实现扮演了一个标记的角色，封装到类的类型中。

两方面的原因促成了 Cloneable interface 的存在。首先，可能有一个上溯造型句柄指向一个基础类型，而且不知道它是否真的能克隆那个对象。在这种情况下，可用 instanceof 关键字（第 11 章有介绍）调查句柄是否确实同一个能克隆的对象连接：

```
if(myHandle instanceof Cloneable) // ...
```

第二个原因是考虑到我们可能不愿所有对象类型都能克隆。所以 Object.clone()会验证一个类是否真的是实现了 Cloneable 接口。若答案是否定的，则“掷”出一个 CloneNotSupportedException 违例。所以在一般情况下，我们必须将“implement Cloneable”作为对克隆能力提供支持的一部分。

12.2.4 成功的克隆

理解了实现 clone()方法背后的所有细节后，便可创建出能方便复制的类，以便提供了一个本地副本：

550-551 页程序

不管怎样，clone()必须能够访问，所以必须将其设为 public（公共的）。其次，作为 clone()的初期行动，应调用 clone()的基础类版本。这里调用的 clone()是 Object 内部预先定义好的。之所以能调用它，是由于它具有 protected（受到保护的）属性，所以能在衍生的类里访问。

Object.clone()会检查原先的对象有多大，再为新对象腾出足够多的内存，将所有二进制位从原来的对象复制到新对象。这叫作“按位复制”，而且按一般的想法，这个工作应该是由 clone()方法来做的。但在 Object.clone()正式开始操作前，首先会检查一个类是否 Cloneable，即是否具有克隆能力——换言之，它是否实现了 Cloneable 接口。若未实现，Object.clone()就掷出一个 CloneNotSupportedException 违例，指出我们不能克隆它。因此，我们最好用一个 try-catch 块将对 super.clone()的调用代码包围（或封装）起来，试图捕获一个应当永不出现的违例（因为这里确实已实现了 Cloneable 接口）。

在 LocalCopy 中，两个方法 g()和 f()揭示出两种参数传递方法间的差异。其中，g()演示的是按引用传递，它会修改外部对象，并返回对那个外部对象的一个引用。而 f()是对自变量进行克隆，所以将其分离出来，并让原来的对象保持独立。随后，它继续做它希望的事情。甚至能返回指向这个新对象的一个句柄，而且不会对原来的对象产生任何副作用。注意下面这个多少有些古怪的语句：

v = (MyObject)v.clone();

它的作用正是创建一个本地副本。为避免被这样的一个语句搞混淆，记住这种相当奇怪的编码形式在 Java 中是完全允许的，因为有一个名字的所有东西实际都是一个句柄。所以句柄 v 用于克隆一个它所指向的副本，而且最终返回指向基础类型 Object 的一个句柄（因为它在 Object.clone()中是那样被定义的），随后必须将其造型为正确的类型。

在 main()中，两种不同参数传递方式的区别在于它们分别测试了一个不同的方法。输出结果如下：

552 页程序

大家要记住这样一个事实：Java 对“是否等价”的测试并不对所比较对象的内部进行检查，从而核实它们的值是否相同。==和!=运算符只是简单地对比句柄的内容。若句柄内的地址相同，就认为句柄指向同样的对象，所以认为它们是“等价”的。所以运算符真正检测的是“由于别名问题，句柄是否指向同一个对象？”

12.2.5 Object.clone()的效果

调用 Object.clone()时，实际发生的是什么事情呢？当我们在自己的类里覆盖 clone()时，什么东西对于 super.clone()来说是最关键的呢？根类中的 clone()方法负责建立正确的存储容量，并通过“按位复制”将二进制位从原始对象中复制到新对象的存储空间。也就是说，它并不只是预留存储空间以及复制一个对象——实际需要调查出欲复制之对象的准确大小，然后复制那个对象。由于所有这些工

作都是在由根类定义之 clone()方法的内部代码中进行的（根类并不知道要从自己这里继承出去什么），所以大家或许已经猜到，这个过程需要用 RTTI 判断欲克隆的对象的实际大小。采取这种方式，clone()方法便可建立起正确数量的存储空间，并对那个类型进行正确的按位复制。

不管我们要做什么，克隆过程的第一个部分通常都应该是调用 super.clone()。通过进行一次准确的复制，这样做可为后续的克隆进程建立起一个良好的基础。随后，可采取另一些必要的操作，以完成最终的克隆。

为确切了解其他操作是什么，首先要正确理解 Object.clone()为我们带来了什么。特别地，它会自动克隆所有句柄指向的目标吗？下面这个例子可完成这种形式的检测：

553-554 页程序

一条 Snake（蛇）由数段构成，每一段的类型都是 Snake。所以，这是一个一段段链接起来的列表。所有段都是以循环方式创建的，每做好一段，都会使第一个构建器参数的值递减，直至最终为零。而为给每段赋予一个独一无二的标记，第二个参数（一个 Char）的值在每次循环构建器调用时都会递增。

increment()方法的作用是循环递增每个标记，使我们能看到发生的变化；而 toString 则循环打印出每个标记。输出如下：

554 页中程序

这意味着只有第一段才是由 Object.clone()复制的，所以此时进行的是一种“浅层复制”。若希望复制整条蛇——即进行“深层复制”——必须在被覆盖的 clone()里采取附加的操作。

通常可在从一个能克隆的类里调用 super.clone()，以确保所有基础类行动（包括 Object.clone()）能够进行。随着是为对象内每个句柄都明确调用一个 clone()；否则那些句柄会别名变成原始对象的句柄。构建器的调用也大致相同——首先构造基础类，然后是下一个衍生的构建器……以此类推，直到位于最深层的衍生构建器。区别在于 clone()并不是个构建器，所以没有办法实现自动克隆。为了克隆，必须由自己明确进行。

12.2.6 克隆合成对象

试图深层复制合成对象时会遇到一个问题。必须假定成员对象中的 clone()方法也能依次对自己的句柄进行深层复制，以此类推。这使我们的操作变得复杂。为了能正常实现深层复制，必须对所有类中的代码进行控制，或者至少全面掌握深层复制中需要涉及的类，确保它们自己的深层复制能正确进行。

下面这个例子总结了面对一个合成对象进行深层复制时需要做哪些事情：

555-556 页程序

DepthReading 和 TemperatureReading 非常相似；它们都只包含了基本数据类型。所以 clone()方法能够非常简单：调用 super.clone()并返回结果即可。注意两个类使用的 clone()代码是完全一致的。

OceanReading 是由 DepthReading 和 TemperatureReading 对象合并而成的。为了对其进行深层复制，clone()必须同时克隆 OceanReading 内的句柄。为达到这个目标，super.clone()的结果必须造型成一个 OceanReading 对象（以便访问 depth 和 temperature 句柄）。

12.2.7 用 Vector 进行深层复制

下面让我们复习一下本章早些时候提出的 Vector 例子。这一次 Int2 类是可以克隆的，所以能对 Vector 进行深层复制：

557-558 页程序

Int3 自 Int2 继承而来，并添加了一个新的基本类型成员 int j。大家也许认为自己需要再次覆盖 clone()，以确保 j 得到复制，但实情并非如此。将 Int2 的 clone()当作 Int3 的 clone()调用时，它会调用 Object.clone()，判断出当前操作的是 Int3，并复制 Int3 内的所有二进制位。只要没有新增需要克隆的句柄，对 Object.clone()的一个调用就能完成所有必要的复制——无论 clone()是在层次结构多深的一级定义的。

至此，大家可以总结出对 Vector 进行深层复制的先决条件：在克隆了 Vector 后，必须在其中遍历，并克隆由 Vector 指向的每个对象。为了对 Hashtable（散列表）进行深层复制，也必须采取类似的处理。

这个例子剩余的部分显示出克隆已实际进行——证据就是在克隆了对象以后，可以自由改变它，而原来那个对象不受任何影响。

12.2.8 通过序列化进行深层复制

若研究一下第 10 章介绍的那个 Java 1.1 对象序列化示例，可能发现若在一个对象序列化以后再撤消对它的序列化，或者说进行装配，那么实际经历的正是一个“克隆”的过程。

那么为什么不用序列化进行深层复制呢？下面这个例子通过计算执行时间对比了这两种方法：

559-560 页程序

其中，Thing2 和 Thing4 包含了成员对象，所以需要进行一些深层复制。一个有趣的地方是尽管 Serializable 类很容易设置，但在复制它们时却要做多得多的工作。克隆涉及到大量的类设置工作，但实际的对象复制是相当简单的。结果很好地说明了一切。下面是几次运行分别得到的结果：

的确

561 页上程序

除了序列化和克隆之间巨大的时间差异以外，我们也注意到序列化技术的运行结果并不稳定，而克隆每一次花费的时间都是相同的。

12.2.9 使克隆具有更大的深度

若新建一个类，它的基础类会默认为 Object，并默认为不具备克隆能力（就

象在下一节会看到的那样）。只要不明确地添加克隆能力，这种能力便不会自动产生。但我们可以在任何层添加它，然后便可从那个层开始向下具有克隆能力。如下所示：

561-562 页程序

添加克隆能力之前，编译器会阻止我们的克隆尝试。一旦在 Scientist 里添加了克隆能力，那么 Scientist 以及它的所有“后裔”都可以克隆。

12.2.10 为什么有这个奇怪的设计

之所以感觉这个方案的奇特，因为它事实上的确如此。也许大家会奇怪它为什么要象这样运行，而该方案背后的真正含义是什么呢？后面讲述的是一个未获证实的故事——大概是由于围绕 Java 的许多买卖使其成为一种设计优良的语言——但确实要花许多口舌才能讲清楚这背后发生的所有事情。

最初，Java 只是作为一种用于控制硬件的语言而设计，与因特网并没有丝毫联系。象这样一类面向大众的语言一样，其意义在于程序员可以对任意一个对象进行克隆。这样一来，clone()就放置在根类 Object 里面，但因为它是一种公用方式，因而我们通常能够对任意一个对象进行克隆。看来这是最灵活的方式了，毕竟它不会带来任何害处。

正当 Java 看起来象一种终级因特网程序设计语言的时候，情况却发生了变化。突然地，人们提出了安全问题，而且理所当然，这些问题与使用对象有关，我们不愿望任何人克隆自己的保密对象。所以我们最后看到的是为原来那个简单、直观的方案添加的大量补丁：clone()在 Object 里被设置成“protected”。必须将其覆盖，并使用“implement Cloneable”，同时解决违例的问题。

只有在准备调用 Object 的 clone()方法时，才没有必要使用 Cloneable 接口，因为那个方法会在运行期间得到检查，以确保我们的类实现了 Cloneable。但为了保持连贯性（而且由于 Cloneable 无论如何都是空的），最好还是由自己实现 Cloneable。

12.3 克隆的控制

为消除克隆能力，大家也许认为只需将 clone()方法简单地设为 private（私有）即可，但这样是行不通的，因为不能采用一个基础类方法，并使其在衍生类中更“私有”。所以事情并没有这么简单。此外，我们有必要控制一个对象是否能够克隆。对于我们设计的一个类，实际有许多种方案都是可以采取的：

(1) 保持中立，不为克隆做任何事情。也就是说，尽管不可对我们的类克隆，但从它继承的一个类却可根据实际情况决定克隆。只有 Object.clone()要对类中的字段进行某些合理的操作时，才可以作这方面的决定。

(2) 支持 clone()，采用实现 Cloneable（可克隆）能力的标准操作，并覆盖 clone()。在被覆盖的 clone()中，可调用 super.clone()，并捕获所有违例（这样可使 clone()不“掷”出任何违例）。

(3) 有条件地支持克隆。若类容纳了其他对象的句柄，而那些对象也许能够克隆（集合类便是这样的一个例子），就可试着克隆拥有对方句柄的所有对象；如果它们“掷”出了违例，只需让这些违例通过即可。举个例子来说，假设有一个特殊的 Vector，它试图克隆自己容纳的所有对象。编写这样的一个 Vector 时，

并不知道客户程序员会把什么形式的对象置入这个 Vector 中，所以并不知道它们是否真的能够克隆。

(4) 不实现 Cloneable()，但是将 clone()覆盖成 protected，使任何字段都具有正确的复制行为。这样一来，从这个类继承的所有东西都能覆盖 clone()，并调用 super.clone()来产生正确的复制行为。注意在我们实现方案里，可以而且应该调用 super.clone()——即使那个方法本来预期的是一个 Cloneable 对象（否则会掷出一个违例），因为没有人会在我们这种类型的对象上直接调用它。它只有通过一个衍生类调用；对那个衍生类来说，如果要保证它正常工作，需实现 Cloneable。

(5) 不实现 Cloneable 来试着防止克隆，并覆盖 clone()，以产生一个违例。为使这一设想顺利实现，只有令从它衍生出来的任何类都调用重新定义后的 clone()里的 suepr.clone()。

(6) 将类设为 final，从而防止克隆。若 clone()尚未被我们的任何一个上级类覆盖，这一设想便不会成功。若已被覆盖，那么再一次覆盖它，并“掷”出一个 CloneNotSupportedException（克隆不支持）违例。为担保克隆被禁止，将类设为 final 是唯一的办法。除此以外，一旦涉及保密对象或者遇到想对创建的对象数量进行控制的其他情况，应该将所有构建器都设为 private，并提供一个或更多的特殊方法来创建对象。采用这种方式，这些方法就可以限制创建的对象数量以及它们的创建条件——一种特殊情况是第 16 章要介绍的 singleton（独子）方案。

下面这个例子总结了克隆的各种实现方法，然后在层次结构中将其“关闭”：

564-565 页程序

第一个类 Ordinary 代表着大家在本书各处最常见到的类：不支持克隆，但在它正式应用以后，却也不禁止对其克隆。但假如有一个指向 Ordinary 对象的句柄，而且那个对象可能是从一个更深的衍生类上溯造型来的，便不能判断它到底能不能克隆。

WrongClone 类揭示了实现克隆的一种不正确途径。它确实覆盖了 Object.clone()，并将那个方法设为 public，但却没有实现 Cloneable。所以一旦发出对 super.clone()的调用（由于对 Object.clone()的一个调用造成的），便会无情地掷出 CloneNotSupportedException 违例。

在 IsCloneable 中，大家看到的才是进行克隆的各种正确行动：先覆盖 clone()，并实现了 Cloneable。但是，这个 clone()方法以及本例的另外几个方法并不捕获 CloneNotSupportedException 违例，而是任由它通过，并传递给调用者。随后，调用者必须用一个 try-catch 代码块把它包围起来。在我们自己的 clone()方法中，通常需要在 clone()内部捕获 CloneNotSupportedException 违例，而不是任由它通过。正如大家以后会理解的那样，对这个例子来说，让它通过是最正确的做法。

类 NoMore 试图按照 Java 设计者打算的那样“关闭”克隆：在衍生类 clone()中，我们掷出 CloneNotSupportedException 违例。TryMore 类中的 clone()方法正确地调用 super.clone()，并解析成 NoMore.clone()，后者掷出一个违例并禁止克隆。

但在已被覆盖的 clone()方法中，假若程序员不遵守调用 super.clone()的“正确”方法，又会出现什么情况呢？在 BackOn 中，大家可看到实际会发生什么。这个类用一个独立的方法 duplicate()制作当前对象的一个副本，并在 clone()内部

调用这个方法，而不是调用 super.clone()。违例永远不会产生，而且新类是可以克隆的。因此，我们不能依赖“掷”出一个违例的方法来防止产生一个可克隆的类。唯一安全的方法在 ReallyNoMore 中得到了演示，它设为 final，所以不可继承。这意味着假如 clone()在 final 类中掷出了一个违例，便不能通过继承来进行修改，并可有效地禁止克隆（不能从一个拥有任意继承级数的类中明确调用 Object.clone()；只能调用 super.clone()，它只可访问直接基础类）。因此，只要制作一些涉及安全问题的对象，就最好把那些类设为 final。

在类 CheckCloneable 中，我们看到的第一个类是 tryToClone()，它能接纳任何 Ordinary 对象，并用 instanceof 检查它是否能够克隆。若答案是肯定的，就将对象造型成为一个 IsCloneable，调用 clone()，并将结果造型回 Ordinary，最后捕获有可能产生的任何违例。请注意用运行期类型鉴定（见第 11 章）打印出类名，使自己看到发生的一切情况。

在 main()中，我们创建了不同类型的 Ordinary 对象，并在数组定义中上溯造型成为 Ordinary。在这之后的头两行代码创建了一个纯粹的 Ordinary 对象，并试图对其克隆。然而，这些代码不会得到编译，因为 clone()是 Object 中的一个 protected（受到保护的）方法。代码剩余的部分将遍历数组，并试着克隆每个对象，分别报告它们的成功或失败。输出如下：

567-568 页程序

总之，如果希望一个类能够克隆，那么：
(1) 实现 Cloneable 接口
(2) 覆盖 clone()
(3) 在自己的 clone()中调用 super.clone()
(4) 在自己的 clone()中捕获违例
这一系列步骤能达到最理想的效果。

12.3.1 副本构建器

克隆看起来要求进行非常复杂的设置，似乎还该有另一种替代方案。一个办法是制作特殊的构建器，令其负责复制一个对象。在 C++中，这叫作“副本构建器”。刚开始的时候，这好象是一种非常显然的解决方案（如果你是 C++程序员，这个方法就更显亲切）。下面是一个实际的例子：

568-571 页程序

这个例子第一眼看上去显得有点奇怪。不同水果的质量肯定有所区别，但为什么只是把代表那些质量的数据成员直接置入 Fruit（水果）类？有两方面可能的原因。第一个是我们可能想简便地插入或修改质量。注意 Fruit 有一个 protected（受到保护的）addQualities()方法，它允许衍生类来进行这些插入或修改操作（大家或许会认为最合乎逻辑的做法是在 Fruit 中使用一个 protected 构建器，用它获取 FruitQualities 参数，但构建器不能继承，所以不可在第二级或级数更深的类中使用它）。通过将水果的质量置入一个独立的类，可以得到更大的灵活性，其中包括可以在特定 Fruit 对象的存在期间中途更改质量。

之所以将 FruitQualities 设为一个独立的对象，另一个原因是考虑到我们有时

希望添加新的质量，或者通过继承与多形性改变行为。注意对 GreenZebra 来说（这实际是西红柿的一类——我已栽种成功，它们简直令人难以置信），构建器会调用 addQualities()，并为其传递一个 ZebraQualities 对象。该对象是从 FruitQualities 衍生出来的，所以能与基础类中的 FruitQualities 句柄联系在一起。当然，一旦 GreenZebra 使用 FruitQualities，就必须将其下溯造型成为正确的类型（就象 evaluate()中展示的那样），但它肯定知道类型是 ZebraQualities。

大家也看到有一个 Seed（种子）类，Fruit（大家都知道，水果含有自己的种子）包含了一个 Seed 数组。

最后，注意每个类都有一个副本构建器，而且每个副本构建器都必须关心为基础类和成员对象调用副本构建器的问题，从而获得“深层复制”的效果。对副本构建器的测试是在 CopyConstructor 类内进行的。方法 ripen()需要获取一个 Tomato 参数，并对其执行副本构建工作，以便复制对象：

```
t = new Tomato(t);
```

而 slice()需要获取一个更常规的 Fruit 对象，而且对它进行复制：

```
f = new Fruit(f);
```

它们都在 main()中伴随不同种类的 Fruit 进行测试。下面是输出结果：

572 页上程序

从中可以看出一个问题。在 slice()内部对 Tomato 进行了副本构建工作以后，结果便不再是一个 Tomato 对象，而只是一个 Fruit。它已丢失了作为一个 Tomato（西红柿）的所有特征。此外，如果采用一个 GreenZebra，ripen()和 slice()会把它分别转换成一个 Tomato 和一个 Fruit。所以非常不幸，假如想制作对象的一个本地副本，Java 中的副本构建器便不是特别适合我们。

1. 为什么在 C++的作用比在 Java 中大？

副本构建器是 C++的一个基本构成部分，因为它能自动产生对象的一个本地副本。但前面的例子确实证明了它不适合在 Java 中使用，为什么呢？在 Java 中，我们操控的一切东西都是句柄，而在 C++中，却可以使用类似于句柄的东西，也能直接传递对象。这时便要用到 C++的副本构建器：只要想获得一个对象，并按值传递它，就可以复制对象。所以它在 C++里能很好地工作，但应注意这套机制在 Java 里是很不通的，所以不要用它。

12.4 只读类

尽管在一些特定的场合，由 clone()产生的本地副本能够获得我们希望的结果，但程序员（方法的作者）不得不亲自禁止别名处理的副作用。假如想制作一个库，令其具有常规用途，但却不能担保它肯定能在正确的类中得以克隆，这时又该怎么办呢？更有可能的一种情况是，假如我们想让别名发挥积极的作用——禁止不必要的对象复制——但却不希望看到由此造成的副作用，那么又该如何处理呢？

一个办法是创建“不变对象”，令其从属于只读类。可定义一个特殊的类，使其中没有任何方法能造成对象内部状态的改变。在这样的一个类中，别名处理是没有问题的。因为我们只能读取内部状态，所以当多处代码都读取相同的对象时，不会出现任何副作用。

作为“不变对象”一个简单例子，Java 的标准库包含了“封装器”（wrapper）类，可用于所有基本数据类型。大家可能已发现了这一点，如果想在一个象 Vector（只采用 Object 句柄）这样的集合里保存一个 int 数值，可以将这个 int 封装到标准库的 Integer 类内部。如下所示：

573 页中程序

Integer 类（以及基本的“封装器”类）用简单的形式实现了“不变性”：它们没有提供可以修改对象的方法。

若确实需要一个容纳了基本数据类型的对象，并想对基本数据类型进行修改，就必须亲自创建它们。幸运的是，操作非常简单：

573-574 页程序

注意 n 在这里简化了我们的编码。

若默认的初始化为零已经足够（便不需要构建器），而且不用考虑把它打印出来（便不需要 toString），那么 IntValue 甚至还能更加简单。如下所示：

class IntValue { int n; }

将元素取出来，再对其进行造型，这多少显得有些笨拙，但那是 Vector 的问题，不是 IntValue 的错。

12.4.1 创建只读类

完全可以创建自己的只读类，下面是个简单的例子：

574-575 页程序

所有数据都设为 private，可以看到没有任何 public 方法对数据作出修改。事实上，确实需要修改一个对象的方法是 quadruple()，但它的作用是新建一个 Immutable1 对象，初始对象则是原封未动的。

方法 f()需要取得一个 Immutable1 对象，并对其采取不同的操作，而 main()的输出显示出没有对 x 作任何修改。因此，x 对象可别名处理许多次，不会造成任何伤害，因为根据 Immutable1 类的设计，它能保证对象不被改动。

12.4.2 “一成不变”的弊端

从表面看，不变类的建立似乎是一个好方案。但是，一旦真的需要那种新类型的一个修改的对象，就必须辛苦地进行新对象的创建工作，同时还有可能涉及更频繁的垃圾收集。对有些类来说，这个问题并不是很大。但对其他类来说（比如 String 类），这一方案的代价显得太高了。

为解决这个问题，我们可以创建一个“同志”类，并使其能够修改。以后只要涉及大量的修改工作，就可换为使用能修改的同志类。完事以后，再切换回不可变的类。

因此，上例可改成下面这个样子：

575-577 页程序

和往常一样，Immutable2 包含的方法保留了对象不可变的特征，只要涉及修改，就创建新的对象。完成这些操作的是 add()和 multiply()方法。同志类叫作 Mutable，它也含有 add()和 multiply()方法。但这些方法能够修改 Mutable 对象，而不是新建一个。除此以外，Mutable 的一个方法可用它的数据产生一个 Immutable2 对象，反之亦然。

两个静态方法 modify1()和 modify2()揭示出获得同样结果的两种不同方法。在 modify1()中，所有工作都是在 Immutable2 类中完成的，我们可看到在进程中创建了四个新的 Immutable2 对象（而且每次重新分配了 val，前一个对象就成为垃圾）。

在方法 modify2()中，可看到它的第一个行动是获取 Immutable2 y，然后从中生成一个 Mutable（类似于前面对 clone()的调用，但这一次创建了一个不同类型的对象）。随后，用 Mutable 对象进行大量修改操作，同时用不着新建许多对象。最后，它切换回 Immutable2。在这里，我们只创建了两个新对象（Mutable 和 Immutable2 的结果），而不是四个。

这一方法特别适合在下述场合应用：

(1) 需要不可变的对象，而且

(2) 经常需要进行大量修改，或者

(3) 创建新的不变对象代价太高

12.4.3 不变字串

请观察下述代码：

577-578 页程序

q 传递进入 upcase()时，它实际是 q 的句柄的一个副本。该句柄连接的对象实际只在一个统一的物理位置处。句柄四处传递的时候，它的句柄会得到复制。

若观察对 upcase()的定义，会发现传递进入的句柄有一个名字 s，而且该名字只有在 upcase()执行期间才会存在。upcase()完成后，本地句柄 s 便会消失，而 upcase()返回结果——还是原来那个字串，只是所有字符都变成了大写。当然，它返回的实际是结果的一个句柄。但它返回的句柄最终是为一个新对象的，同时原来的 q 并未发生变化。所有这些是如何发生的呢？

1. 隐式常数

若使用下述语句：

```
String s = "asdf";
String x = Stringer.upcase(s);
```

那么真的希望 upcase()方法改变自变量或者参数吗？我们通常是不愿意的，因为作为提供给方法的一种信息，自变量一般是拿给代码的读者看的，而不是让他们修改。这是一个相当重要的保证，因为它使代码更易编写和理解。

为了在 C++中实现这一保证，需要一个特殊关键字的帮助：const。利用这个关键字，程序员可以保证一个句柄（C++叫“指针”或者“引用”）不会被用来修改原始的对象。但这样一来，C++程序员需要用心记住在所有地方都使用 const。这显然易使人混淆，也不容易记住。

2. 覆盖"+"和 StringBuffer

利用前面提到的技术，String 类的对象被设计成“不可变”。若查阅联机文档中关于 String 类的内容（本章稍后还要总结它），就会发现类中能够修改 String 的每个方法实际都创建和返回了一个崭新的 String 对象，新对象里包含了修改过的信息——原来的 String 是原封未动的。因此，Java 里没有与 C++的 const 对应的特性可用来让编译器支持对象的不可变能力。若想获得这一能力，可以自行设置，就象 String 那样。

由于 String 对象是不可变的，所以能够根据情况对一个特定的 String 进行多次别名处理。因为它是只读的，所以一个句柄不可能会改变一些会影响其他句柄的东西。因此，只读对象可以很好地解决别名问题。

通过修改产生对象的一个崭新版本，似乎可以解决修改对象时的所有问题，就象 String 那样。但对某些操作来讲，这种方法的效率并不高。一个典型的例子便是为 String 对象覆盖的运算符“+”。“覆盖”意味着在与一个特定的类使用时，它的含义已发生了变化（用于 String 的“+”和“+=”是 Java 中能被覆盖的唯一运算符，Java 不允许程序员覆盖其他任何运算符——注释④）。

④：C++允许程序员随意覆盖运算符。由于这通常是一个复杂的过程（参见《Thinking in C++》，Prentice-Hall 于 1995 年出版），所以 Java 的设计者认定它是一种“糟糕”的特性，决定不在 Java 中采用。但具有讽刺意味的是，运算符的覆盖在 Java 中要比在 C++中容易得多。

针对 String 对象使用时，“+”允许我们将不同的字串连接起来：

579 页中程序

可以想象出它“可能”是如何工作的：字串"abc"可以有一个方法 append()，它新建了一个字串，其中包含"abc"以及 foo 的内容；这个新字串然后再创建另一个新字串，在其中添加"def"；以此类推。

这一设想是行得通的，但它要求创建大量字串对象。尽管最终的目的只是获得包含了所有内容的一个新字串，但中间却要用到大量字串对象，而且要不断地进行垃圾收集。我怀疑 Java 的设计者是否先试过种方法（这是软件开发的一个教训——除非自己试试代码，并让某些东西运行起来，否则不可能真正了解系统）。我还怀疑他们是否早就发现这样做获得的性能是不能接受的。

解决的方法是象前面介绍的那样制作一个可变的同志类。对字串来说，这个同志类叫作 StringBuffer，编译器可以自动创建一个 StringBuffer，以便计算特定的表达式，特别是面向 String 对象应用覆盖过的运算符+和+=时。下面这个例子可以解决这个问题：

580 页程序

创建字串 s 时，编译器做的工作大致等价于后面使用 sb 的代码——创建一个 StringBuffer，并用 append()将新字符直接加入 StringBuffer 对象（而不是每次都产生新对象）。尽管这样做更有效，但不值得每次都创建象"abc"和"def"这样的引

号字串，编译器会把它们都转换成 String 对象。所以尽管 StringBuffer 提供了更高的效率，但会产生比我们希望的多得多的对象。

12.4.4 String 和 StringBuffer 类

这里总结一下同时适用于 String 和 StringBuffer 的方法，以便对它们相互间的沟通方式有一个印象。这些表格并未把每个单独的方法都包括进去，而是包含了与本次讨论有重要关系的方法。那些已被覆盖的方法用单独一行总结。

首先总结 String 类的各种方法：

方法 自变量，覆盖 用途

构建器 已被覆盖：默认，String，StringBuffer，char 数组，byte 数组 创建 String 对象

length() 无 String 中的字符数量

charAt() int Index 位于 String 内某个位置的 char

getChars()，getBytes 开始复制的起点和终点，要向其中复制内容的数组，对目标数组的一个索引 将 char 或 byte 复制到外部数组内部

toCharArray() 无 产生一个 char[]，其中包含了 String 内部的字符

equals()，equalsIgnoreCase() 用于对比的一个 String 对两个字串的内容进行等价性检查

compareTo() 用于对比的一个 String 结果为负、零或正，具体取决于 String 和自变量的字典顺序。注意大写和小写不是相等的！

regionMatches() 这个 String 以及其他 String 的位置偏移，以及要比较的区域长度。覆盖加入了“忽略大小写”的特性 一个布尔结果，指出要对比的区域是否相同

startsWith() 可能以它开头的 String。覆盖在自变量里加入了偏移 一个布尔结果，指出 String 是否以那个自变量开头

endsWith() 可能是这个 String 后缀的一个 String 一个布尔结果，指出自变量是不是一个后缀

indexOf(),lastIndexOf() 已覆盖：char，char 和起始索引，String，String 和起始索引 若自变量未在这个 String 里找到，则返回-1；否则返回自变量开始处的位置索引。lastIndexOf()可从终点开始回溯搜索

substring() 已覆盖：起始索引，起始索引和结束索引 返回一个新的 String 对象，其中包含了指定的字符子集

concat() 想连结的 String 返回一个新 String 对象，其中包含了原始 String 的字符，并在后面加上由自变量提供的字符

relpace() 要查找的老字符，要用它替换的新字符 返回一个新 String 对象，其中已完成了替换工作。若没有找到相符的搜索项，就沿用老字串

toLowerCase(),toUpperCase() 无 返回一个新 String 对象，其中所有字符的大小写形式都进行了统一。若不必修改，则沿用老字串

trim() 无 返回一个新的 String 对象，头尾空白均已删除。若毋需改动，则沿用老字串

valueOf() 已覆盖：object，char[]，char[]和偏移以及计数，boolean，char，int，long，float，double 返回一个 String，其中包含自变量的一个字符表现形式

Intern() 无 为每个独一无二的字符顺序都产生一个（而且只有一个）String 句柄

可以看到，一旦有必要改变原来的内容，每个 String 方法都小心地返回了一个新的 String 对象。另外要注意的一个问题是，若内容不需要改变，则方法只返回指向原来那个 String 的一个句柄。这样做可以节省存储空间和系统开销。

下面列出有关 StringBuffer（字串缓冲）类的方法：

方法 自变量，覆盖 用途

构建器 已覆盖：默认，要创建的缓冲区长度，要根据它创建的 String 新建一个 StringBuffer 对象

toString() 无 根据这个 StringBuffer 创建一个 String

length() 无 StringBuffer 中的字符数量

capacity() 无 返回目前分配的空间大小

ensureCapacity() 用于表示希望容量的一个整数 使 StringBuffer 容纳至少希望的空间大小

setLength() 用于指示缓冲区内字串新长度的一个整数 缩短或扩充前一个字符串。如果是扩充，则用 null 值填充空隙

charAt() 表示目标元素所在位置的一个整数 返回位于缓冲区指定位置处的 char

setCharAt() 代表目标元素位置的一个整数以及元素的一个新 char 值 修改指定位置处的值

getChars() 复制的起点和终点，要在其中复制的数组以及目标数组的一个索引 将 char 复制到一个外部数组。和 String 不同，这里没有 getBytes()可供使用

append() 已覆盖：Object，String，char[]，特定偏移和长度的 char[]，boolean，char，int，long，float，double 将自变量转换成一个字串，并将其追加到当前缓冲区的末尾。若有必要，同时增大缓冲区的长度

insert() 已覆盖，第一个自变量代表开始插入的位置：Object，String，char[]，boolean，char，int，long，float，double 第二个自变量转换成一个字串，并插入当前缓冲区。插入位置在偏移区域的起点处。若有必要，同时会增大缓冲区的长度

reverse() 无 反转缓冲内的字符顺序

最常用的一个方法是 append()。在计算包含了+和+=运算符的 String 表达式时，编译器便会用到这个方法。insert()方法采用类似的形式。这两个方法都能对缓冲区进行重要的操作，不需要另建新对象。

12.4.5 字串的特殊性

现在，大家已知道 String 类并非仅仅是 Java 提供的另一个类。String 里含有大量特殊的类。通过编译器和特殊的覆盖或过载运算符+和+=，可将引号字符串转换成一个 String。在本章中，大家已见识了剩下的一种特殊情况：用同志 StringBuffer 精心构造的“不可变”能力，以及编译器中出现的一些有趣现象。

12.5 总结

由于 Java 中的所有东西都是句柄，而且由于每个对象都是在内存堆中创建的——只有不再需要的时候，才会当作垃圾收集掉，所以对象的操作方式发生了变化，特别是在传递和返回对象的时候。举个例子来说，在 C 和 C++中，如果想在一个方法里初始化一些存储空间，可能需要请求用户将那片存储区域的地址传递进入方法。否则就必须考虑由谁负责清除那片区域。因此，这些方法的接口和对它们的理解就显得要复杂一些。但在 Java 中，根本不必关心由谁负责清除，也不必关心在需要一个对象的时候它是否仍然存在。因为系统会为我们照料一切。我们的程序可在需要的时候创建一个对象。而且更进一步地，根本不必担心那个对象的传输机制的细节：只需简单地传递句柄即可。有些时候，这种简化非常有价值，但另一些时候却显得有些多余。

可从两个方面认识这一机制的缺点：

(1) 肯定要为额外的内存管理付出效率上的损失（尽管损失不大），而且对于运行所需的时间，总是存在一丝不确定的因素（因为在内存不够时，垃圾收集器可能会被强制采取行动）。对大多数应用来说，优点显得比缺点重要，而且部分对时间要求非常苛刻的段落可以用 native 方法写成（参见附录 A）。

(2) 别名处理：有时会不慎获得指向同一个对象的两个句柄。只有在这两个句柄都假定指向一个“明确”的对象时，才有可能产生问题。对这个问题，必须加以足够的重视。而且应该尽可能地“克隆”一个对象，以防止另一个句柄被不希望的改动影响。除此以外，可考虑创建“不可变”对象，使它的操作能返回同种类型或不同种类型的一个新对象，从而提高程序的执行效率。但千万不要改变原始对象，使对那个对象别名的其他任何方面都感觉不出变化。

有些人认为 Java 的克隆是一个笨拙的家伙，所以他们实现了自己的克隆方案（注释⑤），永远杜绝调用 Object.clone()方法，从而消除了实现 Cloneable 和捕获 CloneNotSupportException 违例的需要。这一做法是合理的，而且由于 clone()在 Java 标准库中很少得以支持，所以这显然也是一种“安全”的方法。只要不调用 Object.clone()，就不必实现 Cloneable 或者捕获违例，所以那看起来也是能够接受的。

⑤：Doug Lea 特别重视这个问题，并把这个方法推荐给了我，他说只需为每个类都创建一个名为 duplicate()的函数即可。

Java 中一个有趣的关键字是 byvalue（按值），它属于那些“保留但未实现”的关键字之一。在理解了别名和克隆问题以后，大家可以想象 byvalue 最终有一天会在 Java 中用于实现一种自动化的本地副本。这样做可以解决更多复杂的克隆问题，并使这种情况下的编写的代码变得更加简单和健壮。

12.6 练习

(1) 创建一个 myString 类，在其中包含了一个 String 对象，以便用在构建器中用构建器的自变量对其进行初始化。添加一个 toString()方法以及一个 concatenate()方法，令其将一个 String 对象追加到我们的内部字串。在 myString 中实现 clone()。创建两个 static 方法，每个都取得一个 myString x 句柄作为自己的自变量，并调用 x.concatenate("test")。但在第二个方法中，请首先调用 clone()。

测试这两个方法，观察它们不同的结果。

(2) 创建一个名为 Battery（电池）的类，在其中包含一个 int，用它表示电池的编号（采用独一无二的标识符的形式）。接下来，创建一个名为 Toy 的类，其中包含了一个 Battery 数组以及一个 toString，用于打印出所有电池。为 Toy 写一个 clone()方法，令其自动关闭所有 Battery 对象。克隆 Toy 并打印出结果，完成对它的测试。

(3) 修改 CheckCloneable.java，使所有 clone() 方法都能捕获 CloneNotSupportException 违例，而不是把它直接传递给调用者。

(4) 修改 Compete.java，为 Thing2 和 Thing4 类添加更多的成员对象，看看自己是否能判断计时随复杂性变化的规律——是一种简单的线性关系，还是看起来更加复杂。

(5) 从 Snake.java 开始，创建 Snake 的一个深层复制版本。