第 8 章 对象的容纳

“如果一个程序只含有数量固定的对象，而且已知它们的存在时间，那么这个程序可以说是相当简单的。”

通常，我们的程序需要根据程序运行时才知道的一些标准创建新对象。除非程序正式运行，否则我们根本不知道自己到底需要多少数量的对象，甚至不知道它们的准确类型。为了满足常规编程的需要，我们要求能在任何时候、任何地点创建任意数量的对象。所以不可依赖一个已命名的句柄来容纳自己的每一个对象，就象下面这样：

MyObject myHandle;

因为根本不知道自己实际需要多少这样的东西。

为解决这个非常关键的问题，Java 提供了容纳对象（或者对象的句柄）的多种方式。其中内建的类型是数组，我们之前已讨论过它，本章准备加深大家对它的认识。此外，Java 的工具（实用程序）库提供了一些“集合类”（亦称作“容器类”，但该术语已由 AWT 使用，所以这里仍采用“集合”这一称呼）。利用这些集合类，我们可以容纳乃至操纵自己的对象。本章的剩余部分会就此进行详细讨论。

8.1 数组

对数组的大多数必要的介绍已在第 4 章的最后一节进行。通过那里的学习，大家已知道自己该如何定义及初始化一个数组。对象的容纳是本章的重点，而数组只是容纳对象的一种方式。但由于还有其他大量方法可容纳数组，所以是哪些地方使数组显得如此特别呢？

有两方面的问题将数组与其他集合类型区分开来：效率和类型。对于 Java 来说，为保存和访问一系列对象（实际是对象的句柄）数组，最有效的方法莫过于数组。数组实际代表一个简单的线性序列，它使得元素的访问速度非常快，但我们却要为这种速度付出代价：创建一个数组对象时，它的大小是固定的，而且不可在那个数组对象的“存在时间”内发生改变。可创建特定大小的一个数组，然后假如用光了存储空间，就再创建一个新数组，将所有句柄从旧数组移到新数组。这属于“矢量”（Vector）类的行为，本章稍后还会详细讨论它。然而，由于为这种大小的灵活性要付出较大的代价，所以我们认为矢量的效率并没有数组高。

C++的矢量类知道自己容纳的是什么类型的对象，但同 Java 的数组相比，它却有一个明显的缺点：C++矢量类的 operator[]不能进行范围检查，所以很容易超出边界（然而，它可以查询 vector 有多大，而且 at()方法确实能进行范围检查）。在 Java 中，无论使用的是数组还是集合，都会进行范围检查——若超过边界，就会获得一个 RuntimeException（运行期违例）错误。正如大家在第 9 章会学到的那样，这类违例指出的是一个程序员错误，所以不需要在代码中检查它。在另一方面，由于 C++的 vector 不进行范围检查，所以访问速度较快——在 Java 中，由于对数组和集合都要进行范围检查，所以对性能有一定的影响。

本章还要学习另外几种常见的集合类：Vector（矢量）、Stack（堆栈）以及 Hashtable（散列表）。这些类都涉及对对象的处理——好象它们没有特定的类型。换言之，它们将其当作 Object 类型处理（Object 类型是 Java 中所有类的“根”类）。从某个角度看，这种处理方法是非常合理的：我们仅需构建一个集合，然后任何 Java 对象都可以进入那个集合（除基本数据类型外——可用 Java 的基本类型封装类将其作为常数置入集合，或者将其封装到自己的类内，作为可以变化的值使用）。这再一次反映了数组优于常规集合：创建一个数组时，可令其容纳一种特定的类型。这意味着可进行编译期类型检查，预防自己设置了错误的类型，或者错误指定了准备提取的类型。当然，在编译期或者运行期，Java 会防止我们将不当的消息发给一个对象。所以我们不必考虑自己的哪种做法更加危险，只要编译器能及时地指出错误，同时在运行期间加快速度，目的也就达到了。此外，用户很少会对一次违例事件感到非常惊讶的。

考虑到执行效率和类型检查，应尽可能地采用数组。然而，当我们试图解决一个更常规的问题时，数组的局限也可能显得非常明显。在研究过数组以后，本章剩余的部分将把重点放到 Java 提供的集合类身上。

8.1.1 数组和第一类对象

无论使用的数组属于什么类型，数组标识符实际都是指向真实对象的一个句柄。那些对象本身是在内存“堆”里创建的。堆对象既可“隐式”创建（即默认产生），亦可“显式”创建（即明确指定，用一个 new 表达式）。堆对象的一部分（实际是我们能访问的唯一字段或方法）是只读的 length（长度）成员，它告诉我们那个数组对象里最多能容纳多少元素。对于数组对象，“[]”语法是我们能采用的唯一另类访问方法。

下面这个例子展示了对数组进行初始化的不同方式，以及如何将数组句柄分配给不同的数组对象。它也揭示出对象数组和基本数据类型数组在使用方法上几乎是完全一致的。唯一的差别在于对象数组容纳的是句柄，而基本数据类型数组容纳的是具体的数值（若在执行此程序时遇到困难，请参考第 3 章的“赋值”小

节）：

325-327 页程序

其中，数组 a 只是初始化成一个 null 句柄。此时，编译器会禁止我们对这个句柄作任何实际操作，除非已正确地初始化了它。数组 b 被初始化成指向由 Weeble 句柄构成的一个数组，但那个数组里实际并未放置任何 Weeble 对象。然而，我们仍然可以查询那个数组的大小，因为 b 指向的是一个合法对象。这也为我们带来了一个难题：不可知道那个数组里实际包含了多少个元素，因为 length 只告诉我们可将多少元素置入那个数组。换言之，我们只知道数组对象的大小或容量，不知其实际容纳了多少个元素。尽管如此，由于数组对象在创建之初会自动初始化成 null，所以可检查它是否为 null，判断一个特定的数组“空位”是否容纳一个对象。类似地，由基本数据类型构成的数组会自动初始化成零（针对数值类型）、null（字符类型）或者 false（布尔类型）。

数组 c 显示出我们首先创建一个数组对象，再将 Weeble 对象赋给那个数组的所有“空位”。数组 d 揭示出“集合初始化”语法，从而创建数组对象（用 new 命令明确进行，类似于数组 c），然后用 Weeble 对象进行初始化，全部工作在一条语句里完成。

下面这个表达式：

a = d;

向我们展示了如何取得同一个数组对象连接的句柄，然后将其赋给另一个数组对象，就象我们针对对象句柄的其他任何类型做的那样。现在，a 和 d 都指向内存堆内同样的数组对象。

Java 1.1 加入了一种新的数组初始化语法，可将其想象成“动态集合初始化”。由 d 采用的 Java 1.0 集合初始化方法则必须在定义 d 的同时进行。但若采用 Java 1.1 的语法，却可以在任何地方创建和初始化一个数组对象。例如，假设 hide() 方法用于取得一个 Weeble 对象数组，那么调用它时传统的方法是：

hide(d);

但在 Java 1.1 中，亦可动态创建想作为参数传递的数组，如下所示：

hide(new Weeble[] {new Weeble(), new Weeble() });

这一新式语法使我们在某些场合下写代码更方便了。

上述例子的第二部分揭示出这样一个问题：对于由基本数据类型构成的数组，它们的运作方式与对象数组极为相似，只是前者直接包容了基本类型的数据值。

1. 基本数据类型集合

集合类只能容纳对象句柄。但对一个数组，却既可令其直接容纳基本类型的数据，亦可容纳指向对象的句柄。利用象 Integer、Double 之类的“封装器”类，可将基本数据类型的值置入一个集合里。但正如本章后面会在 WordCount.java 例子中讲到的那样，用于基本数据类型的封装器类只是在某些场合下才能发挥作用。无论将基本类型的数据置入数组，还是将其封装进入位于集合的一个类内，都涉及到执行效率的问题。显然，若能创建和访问一个基本数据类型数组，那么

比起访问一个封装数据的集合，前者的效率会高出许多。

当然，假如准备一种基本数据类型，同时又想要集合的灵活性（在需要的时候可自动扩展，腾出更多的空间），就不宜使用数组，必须使用由封装的数据构成的一个集合。大家或许认为针对每种基本数据类型，都应有一种特殊类型的Vector。但Java并未提供这一特性。某些形式的建模机制或许会在某一天帮助Java更好地解决这个问题（注释①）。

①：这儿是C++比Java做得好的一个地方，因为C++通过template关键字提供了对“参数化类型”的支持。

8.1.2 数组的返回

假定我们现在想写一个方法，同时不希望它仅仅返回一样东西，而是想返回一系列东西。此时，象C和C++这样的语言会使问题复杂化，因为我们不能返回一个数组，只能返回指向数组的一个指针。这样就非常麻烦，因为很难控制数组的“存在时间”，它很容易造成内存“漏洞”的出现。

Java采用的是类似的方法，但我们能“返回一个数组”。当然，此时返回的实际仍是指向数组的指针。但在Java里，我们永远不必担心那个数组的是否可用——只要需要，它就会自动存在。而且垃圾收集器会在我们完成后自动将其清除。

作为一个例子，请思考如何返回一个字串数组：

329-330 页程序

flavorSet()方法创建了一个名为results的String数组。该数组的大小为n——具体数值取决于我们传递给方法的自变量。随后，它从数组flav里随机挑选一些“香料”（Flavor），并将它们置入results里，并最终返回results。返回数组与返回其他任何对象没什么区别——最终返回的都是一个句柄。至于数组到底是在flavorSet()里创建的，还是在其他什么地方创建的，这个问题并不重要，因为反正返回的仅是一个句柄。一旦我们的操作完成，垃圾收集器会自动关照数组的清除工作。而且只要我们需要数组，它就会乖乖地听候调遣。

另一方面，注意当flavorSet()随机挑选香料的时候，它需要保证以前出现过的一次随机选择不会再次出现。为达到这个目的，它使用了一个无限while循环，不断地作出随机选择，直到发现未在picks数组里出现过的一个元素为止（当然，也可以进行字串比较，检查随机选择是否在results数组里出现过，但字串比较的效率比较低）。若成功，就添加这个元素，并中断循环（break），再查找下一个（i值会递增）。但假若t是一个已在picks里出现过的数组，就用标签式的continue往回跳两级，强制选择一个新t。用一个调试程序可以很清楚地看到这个过程。

main()能显示出20个完整的香料集合，所以我们看到flavorSet()每次都用一个随机顺序选择香料。为体会这一点，最简单的方法就是将输出重导向进入一个文件，然后直接观看这个文件的内容。

8.2 集合

现在总结一下我们前面学过的东西：为容纳一组对象，最适宜的选择应当是

数组。而且假如容纳的是一系列基本数据类型，更是必须采用数组。在本章剩下的部分，大家将接触到一些更常规的情况。当我们编写程序时，通常并不能确切地知道最终需要多少个对象。有些时候甚至想用更复杂的方式来保存对象。为解决这个问题，Java 提供了四种类型的“集合类”：Vector（矢量）、BitSet（位集）、Stack（堆栈）以及 Hashtable（散列表）。与拥有集合功能的其他语言相比，尽管这儿的数量显得相当少，但仍然能用它们解决数量惊人的实际问题。

这些集合类具有形形色色的特征。例如，Stack 实现了一个 LIFO（先入先出）序列，而 Hashtable 是一种“关联数组”，允许我们将任何对象关联起来。除此以外，所有 Java 集合类都能自动改变自身的大小。所以，我们在编程时可使用数量众多的对象，同时不必担心会将集合弄得有多大。

8.2.1 缺点：类型未知

使用 Java 集合的“缺点”是在将对象置入一个集合时丢失了类型信息。之所以会发生这种情况，是由于当初编写集合时，那个集合的程序员根本不知道用户到底想把什么类型置入集合。若指示某个集合只允许特定的类型，会妨碍它成为一个“常规用途”的工具，为用户带来麻烦。为解决这个问题，集合实际容纳的是类型为 Object 的一些对象的句柄。这种类型当然代表 Java 中的所有对象，因为它是所有类的根。当然，也要注意这并不包括基本数据类型，因为它们并不是从“任何东西”继承来的。这是一个很好的方案，只是不适用下述场合：

(1) 将一个对象句柄置入集合时，由于类型信息会被抛弃，所以任何类型的对象都可进入我们的集合——即便特别指示它只能容纳特定类型的对象。举个例子来说，虽然指示它只能容纳猫，但事实上任何人都可以把一条狗扔进来。

(2) 由于类型信息不复存在，所以集合能肯定的唯一事情就是自己容纳的是指向一个对象的句柄。正式使用它之前，必须对其进行造型，使其具有正确的类型。

值得欣慰的是，Java 不允许人们滥用置入集合的对象。假如将一条狗扔进一个猫的集合，那么仍会将集合内的所有东西都看作猫，所以在使用那条狗时会得到一个“违例”错误。在同样的意义上，假若试图将一条狗的句柄“造型”到一只猫，那么运行期间仍会得到一个“违例”错误。

下面是个例子：

332-333 页程序

可以看出，Vector 的使用是非常简单的：先创建一个，再用 addElement()置入对象，以后用 elementAt()取得那些对象（注意 Vector 有一个 size()方法，可使我们知道已添加了多少个元素，以便防止误超边界，造成违例错误）。

Cat 和 Dog 类都非常浅显——除了都是“对象”之外，它们并无特别之处（倘若不明确指出从什么类继承，就默认为从 Object 继承。所以我们不仅能用 Vector 方法将 Cat 对象置入这个集合，也能添加 Dog 对象，同时不会在编译期和运行期得到任何出错提示。用 Vector 方法 elementAt()获取原本认为是 Cat 的对象时，实际获得的是指向一个 Object 的句柄，必须将那个对象造型为 Cat。随后，需要将整个表达式用括号封闭起来，在为 Cat 调用 print()方法之前进行强制造型；否则就会出现一个语法错误。在运行期间，如果试图将 Dog 对象造型为 Cat，就会

得到一个违例。

这些处理的意义都非常深远。尽管显得有些麻烦，但却获得了安全上的保证。我们从此再难偶然造成一些隐藏得深的错误。若程序的一个部分（或几个部分）将对象插入一个集合，但我们只是通过一次违例在程序的某个部分发现一个错误的对象置入了集合，就必须找出插入错误的位置。当然，可通过检查代码达到这个目的，但这或许是最笨的调试工具。另一方面，我们可从一些标准化的集合类开始自己的编程。尽管它们在功能上存在一些不足，且显得有些笨拙，但却能保证没有隐藏的错误。

1. 错误有时并不显露出来

在某些情况下，程序似乎正确地工作，不造型回我们原来的类型。第一种情况是相当特殊的：String 类从编译器获得了额外的帮助，使其能够正常工作。只要编译器期待的是一个 String 对象，但它没有得到一个，就会自动调用在 Object 里定义、并且能够由任何 Java 类覆盖的 toString()方法。这个方法能生成满足要求的 String 对象，然后在我们需要的时候使用。

因此，为了让自己类的对象能显示出来，要做的全部事情就是覆盖 toString()方法，如下例所示：

334-335 页程序

可在 Mouse 里看到对 toString()的重定义代码。在 main()的第二个 for 循环中，可发现下述语句：

```
System.out.println("Free mouse: " +
mice.elementAt(i));
```

在“+”后，编译器预期看到的是一个 String 对象。elementAt()生成了一个 Object，所以为获得希望的 String，编译器会默认调用 toString()。但不幸的是，只有针对 String 才能得到象这样的结果；其他任何类型都不会进行这样的转换。

隐藏造型的第二种方法已在 Mousetrap 里得到了应用。caughtYa()方法接收的不是一个 Mouse，而是一个 Object。随后再将其造型为一个 Mouse。当然，这样做是非常冒失的，因为通过接收一个 Object，任何东西都可以传递给方法。然而，假若造型不正确——如果我们传递了错误的类型——就会在运行期间得到一个违例错误。这当然没有在编译期进行检查好，但仍然能防止问题的发生。注意在使用这个方法时毋需进行造型：

```
MouseTrap.caughtYa(mice.elementAt(i));
```

2. 生成能自动判别类型的 Vector

大家或许不想放弃刚才那个问题。一个更“健壮”的方案是用 Vector 创建一个新类，使其只接收我们指定的类型，也只生成我们希望的类型。如下所示：

335-336 页程序

这前一个例子类似，只是新的 GopherVector 类有一个类型为 Vector 的 private

成员（从 Vector 继承有些麻烦，理由稍后便知），而且方法也和 Vector 类似。然而，它不会接收和产生普通 Object，只对 Gopher 对象感兴趣。

由于 GopherVector 只接收一个 Gopher（地鼠），所以假如我们使用：

gophers.addElement(new Pigeon());

就会在编译期间获得一条出错消息。采用这种方式，尽管从编码的角度看显得更令人沉闷，但可以立即判断出是否使用了正确的类型。

注意在使用 elementAt()时不必进行造型——它肯定是一个 Gopher。

3. 参数化类型

这类问题并不是孤立的——我们许多时候都要在其他类型的基础上创建新类型。此时，在编译期间拥有特定的类型信息是非常有帮助的。这便是“参数化类型”的概念。在 C++中，它由语言通过“模板”获得了直接支持。至少，Java 保留了关键字 generic，期望有一天能够支持参数化类型。但我们现在无法确定这一天何时会来临。

8.3 枚举器（反复器）

在任何集合类中，必须通过某种方法在其中置入对象，再用另一种方法从中取得对象。毕竟，容纳各种各样的对象正是集合的首要任务。在 Vector 中，addElement()便是我们插入对象采用的方法，而 elementAt()是提取对象的唯一方法。Vector 非常灵活，我们可在任何时候选择任何东西，并可使用不同的索引选择多个元素。

若从更高的角度看这个问题，就会发现它的一个缺陷：需要事先知道集合的准确类型，否则无法使用。乍看来，这一点似乎没什么关系。但假若最开始决定使用 Vector，后来在程序中又决定（考虑执行效率的原因）改变成一个 List（属于 Java1.2 集合库的一部分），这时又该如何做呢？

可利用“反复器”（Iterator）的概念达到这个目的。它可以是一个对象，作用是遍历一系列对象，并选择那个序列中的每个对象，同时不让客户程序员知道或关注那个序列的基础结构。此外，我们通常认为反复器是一种“轻量级”对象；也就是说，创建它只需付出极少的代价。但也正是由于这个原因，我们常发现反复器存在一些似乎很奇怪的限制。例如，有些反复器只能朝一个方向移动。

Java 的 Enumeration（枚举，注释②）便是具有这些限制的一个反复器的例子。除下面这些外，不可再用它做其他任何事情：

(1) 用一个名为 elements()的方法要求集合为我们提供一个 Enumeration。我们首次调用它的 nextElement()时，这个 Enumeration 会返回序列中的第一个元素。

(2) 用 nextElement()获得下一个对象。

(3) 用 hasMoreElements()检查序列中是否还有更多的对象。

②：“反复器”这个词在 C++和 OOP 的其他地方是经常出现的，所以很难确定为什么 Java 的开发者采用了这样一个奇怪的名字。Java 1.2 的集合库修正了这个问题以及其他许多问题。

只可用 Enumeration 做这些事情，不能再有更多。它属于反复器一种简单的实现方式，但功能依然十分强大。为体会它的运作过程，让我们复习一下本章早些时候提到的 CatsAndDogs.java 程序。在原始版本中，elementAt()方法用于选择

每一个元素，但在下述修订版中，可看到使用了一个“枚举”：

338-339 页程序

我们看到唯一的改变就是最后几行。不再是：

```
for(int i = 0; i < cats.size(); i++)
((Cat)cats.elementAt(i)).print();
```

而是用一个 Enumeration 遍历整个序列：

```
while(e.hasMoreElements())
((Cat2)e.nextElement()).print();
```

使用 Enumeration，我们不必关心集合中的元素数量。所有工作均由 hasMoreElements()和 nextElement()自动照管了。

下面再看看另一个例子，让我们创建一个常规用途的打印方法：

339-340 页程序

仔细研究一下打印方法：

340 页上程序

注意其中没有与序列类型有关的信息。我们拥有的全部东西便是 Enumeration。为了解有关序列的情况，一个 Enumeration 便足够了：可取得下一个对象，亦可知道是否已抵达了末尾。取得一系列对象，然后在其中遍历，从而执行一个特定的操作——这是一个颇有价值的编程概念，本书许多地方都会沿用这一思路。

这个看似特殊的例子甚至可以更为通用，因为它使用了常规的 toString()方法（之所以称为常规，是由于它属于 Object 类的一部分）。下面是调用打印的另一个方法（尽管在效率上可能会差一些）：

```
System.out.println("" + e.nextElement());
```

它采用了封装到 Java 内部的“自动转换成字串”技术。一旦编译器碰到一个字串，后面跟随一个“+”，就会希望后面又跟随一个字串，并自动调用 toString()。在 Java 1.1 中，第一个字串是不必要的；所有对象都会转换成字串。亦可对此执行一次造型，获得与调用 toString()同样的效果：

```
System.out.println((String)e.nextElement())
```

但我们想做的事情通常并不仅仅是调用 Object 方法，所以会再度面临类型造型的问题。对于自己感兴趣的类型，必须假定自己已获得了一个 Enumeration，然后将结果对象造型成为那种类型（若操作错误，会得到运行期违例）。

8.4 集合的类型

标准 Java 1.0 和 1.1 库配套提供了非常少的一系列集合类。但对于自己的大

多数编程要求，它们基本上都能胜任。正如大家到本章末尾会看到的，Java 1.2 提供的是一套重新设计过的大型集合库。

8.4.1 Vector

Vector 的用法很简单，这已在前面的例子中得到了证明。尽管我们大多数时候只需用 addElement()插入对象，用 elementAt()一次提取一个对象，并用 elements()获得对序列的一个“枚举”。但仍有其他一系列方法是非常有用的。同我们对于 Java 库惯常的做法一样，在这里并不使用或讲述所有这些方法。但请务必阅读相应的电子文档，对它们的工作有一个大概的认识。

1. 崩溃 Java

Java 标准集合里包含了 toString()方法，所以它们能生成自己的 String 表达方式，包括它们容纳的对象。例如在 Vector 中，toString()会在 Vector 的各个元素中步进和遍历，并为每个元素调用 toString()。假定我们现在想打印出自己类的地址。看起来似乎简单地引用 this 即可（特别是 C++程序员有这样做的倾向）：

341 页下程序

若只是简单地创建一个 CrashJava 对象，并将其打印出来，就会得到无穷无尽的一系列违例错误。然而，假如将 CrashJava 对象置入一个 Vector，并象这里演示的那样打印 Vector，就不会出现什么错误提示，甚至连一个违例都不会出现。此时 Java 只是简单地崩溃（但至少它没有崩溃我的操作系统）。这已在 Java 1.1 中测试通过。

此时发生的是字串的自动类型转换。当我们使用下述语句时：

```
"CrashJava address: " + this
```

编译器就在一个字串后面发现了一个“+”以及好象并非字串的其他东西，所以它会试图将 this 转换成一个字串。转换时调用的是 toString()，后者会产生一个递归调用。若在一个 Vector 内出现这种事情，看起来堆栈就会溢出，同时违例控制机制根本没有机会作出响应。

若确实想在这种情况下打印出对象的地址，解决方案就是调用 Object 的 toString 方法。此时就不必加入 this，只需使用 super.toString()。当然，采取这种做法也有一个前提：我们必须从 Object 直接继承，或者没有一个父类覆盖了 toString 方法。

8.4.2 BitSet

BitSet 实际是由“二进制位”构成的一个 Vector。如果希望高效率地保存大量“开－关”信息，就应使用 BitSet。它只有从尺寸的角度看才有意义；如果希望的高效率的访问，那么它的速度会比使用一些固有类型的数组慢一些。

此外，BitSet 的最小长度是一个长整数（Long）的长度：64 位。这意味着假如我们准备保存比这更小的数据，如 8 位数据，那么 BitSet 就显得浪费了。所以最好创建自己的类，用它容纳自己的标志位。

在一个普通的 Vector 中，随我们加入越来越多的元素，集合也会自我膨胀。在某种程度上，BitSet 也不例外。也就是说，它有时会自行扩展，有时则不然。而且 Java 的 1.0 版本似乎在这方面做得最糟，它的 BitSet 表现十分差强人意

（Java1.1 已改正了这个问题）。下面这个例子展示了 BitSet 是如何运作的，同时演示了 1.0 版本的错误：

342-344 页程序

随机数字生成器用于创建一个随机的 byte、short 和 int。每一个都会转换成 BitSet 内相应的位模型。此时一切都很正常，因为 BitSet 是 64 位的，所以它们都不会造成最终尺寸的增大。但在 Java 1.0 中，一旦 BitSet 大于 64 位，就会出现一些令人迷惑不解的行为。假如我们设置一个只比 BitSet 当前分配存储空间大出 1 的一个位，它能够正常地扩展。但一旦试图在更高的位置设置位，同时不先接触边界，就会得到一个恼人的违例。这正是由于 BitSet 在 Java 1.0 里不能正确扩展造成的。本例创建了一个 512 位的 BitSet。构建器分配的存储空间是位数的两倍。所以假如设置位 1024 或更高的位，同时没有先设置位 1023，就会在 Java 1.0 里得到一个违例。但幸运的是，这个问题已在 Java 1.1 得到了改正。所以如果是为 Java 1.0 写代码，请尽量避免使用 BitSet。

8.4.3 Stack

Stack 有时也可以称为“后入先出”（LIFO）集合。换言之，我们在堆栈里最后“压入”的东西将是以后第一个“弹出”的。和其他所有 Java 集合一样，我们压入和弹出的都是“对象”，所以必须对自己弹出的东西进行“造型”。

一种很少见的做法是拒绝使用 Vector 作为一个 Stack 的基本构成元素，而是从 Vector 里“继承”一个 Stack。这样一来，它就拥有了一个 Vector 的所有特征及行为，另外加上一些额外的 Stack 行为。很难判断出设计者到底是明确想这样做，还是属于一种固有的设计。

下面是一个简单的堆栈示例，它能读入数组的每一行，同时将其作为字串压入堆栈。

345 页程序

months 数组的每一行都通过 push()继承进入堆栈，稍后用 pop()从堆栈的顶部将其取出。要声明的一点是，Vector 操作亦可针对 Stack 对象进行。这可能是由继承的特质决定的——Stack“属于”一种 Vector。因此，能对 Vector 进行的操作亦可针对 Stack 进行，例如 elementAt()方法。

8.4.4 Hashtable

Vector 允许我们用一个数字从一系列对象中作出选择，所以它实际是将数字同对象关联起来了。但假如我们想根据其他标准选择一系列对象呢？堆栈就是这样的一个例子：它的选择标准是“最后压入堆栈的东西”。这种“从一系列对象中选择”的概念亦可叫作一个“映射”、“字典”或者“关联数组”。从概念上讲，它看起来象一个 Vector，但却不是通过数字来查找对象，而是用另一个对象来查找它们！这通常都属于一个程序中的重要进程。

在 Java 中，这个概念具体反映到抽象类 Dictionary 身上。该类的接口是非常直观的 size()告诉我们其中包含了多少元素；isEmpty()判断是否包含了元素（是则为 true）；put(Object key, Object value)添加一个值（我们希望的东西），并将其

同一个键关联起来（想用于搜索它的东西）；get(Object key)获得与某个键对应的值；而 remove(Object Key)用于从列表中删除“键－值”对。还可以使用枚举技术：keys()产生对键的一个枚举（Enumeration）；而 elements()产生对所有值的一个枚举。这便是一个 Dictionary（字典）的全部。

Dictionary 的实现过程并不麻烦。下面列出一种简单的方法，它使用了两个 Vector，一个用于容纳键，另一个用来容纳值：

346-347 页程序

在对 AssocArray 的定义中，我们注意到的第一个问题是它“扩展”了字典。这意味着 AssocArray 属于 Dictionary 的一种类型，所以可对其发出与 Dictionary 一样的请求。如果想生成自己的 Dictionary，而且就在这里进行，那么要做的全部事情只是填充位于 Dictionary 内的所有方法（而且必须覆盖所有方法，因为它们——除构建器外——都是抽象的）。

Vector key 和 value 通过一个标准索引编号链接起来。也就是说，如果用“roof”的一个键以及“blue”的一个值调用 put()——假定我们准备将一个房子的各部分与它们的油漆颜色关联起来，而且 AssocArray 里已有 100 个元素，那么“roof”就会有 101 个键元素，而“blue”有 101 个值元素。而且要注意一下 get()，假如我们作为键传递“roof”，它就会产生与 keys.index.Of()的索引编号，然后用那个索引编号生成相关的值矢量内的值。

main()中进行的测试是非常简单的；它只是将小写字符转换成大写字符，这显然可用更有效的方式进行。但它向我们揭示出了 AssocArray 的强大功能。

标准 Java 库只包含 Dictionary 的一个变种，名为 Hashtable（散列表，注释③）。Java 的散列表具有与 AssocArray 相同的接口（因为两者都是从 Dictionary 继承来的）。但有一个方面却反映出了差别：执行效率。若仔细想想必须为一个 get()做的事情，就会发现在一个 Vector 里搜索键的速度要慢得多。但此时用散列表却可以加快不少速度。不必用冗长的线性搜索技术来查找一个键，而是用一个特殊的值，名为“散列码”。散列码可以获取对象中的信息，然后将其转换成那个对象“相对唯一”的整数（int）。所有对象都有一个散列码，而 hashCode()是根类 Object 的一个方法。Hashtable 获取对象的 hashCode()，然后用它快速查找键。这样可使性能得到大幅度提升（④）。散列表的具体工作原理已超出了本书的范围（⑤）——大家只需要知道散列表是一种快速的“字典”（Dictionary）即可，而字典是一种非常有用的工具。

③：如计划使用 RMI（在第 15 章详述），应注意将远程对象置入散列表时会遇到一个问题（参阅《Core Java》，作者 Conrell 和 Horstmann，Prentice-Hall 1997 年出版）

④：如这种速度的提升仍然不能满足你对性能的要求，甚至可以编写自己的散列表例程，从而进一步加快表格的检索过程。这样做可避免在与 Object 之间进行造型的时间延误，也可以避开由 Java 类库散列表例程内建的同步过程。

⑤：我的知道的最佳参考读物是《Practical Algorithms for Programmers》，作者为 Andrew Binstock 和 John Rex，Addison-Wesley 1995 年出版。

作为应用散列表的一个例子，可考虑用一个程序来检验 Java 的 Math.random()

方法的随机性到底如何。在理想情况下，它应该产生一系列完美的随机分布数字。但为了验证这一点，我们需要生成数量众多的随机数字，然后计算落在不同范围内的数字多少。散列表可以极大简化这一工作，因为它能将对象同对象关联起来（此时是将 Math.random()生成的值同那些值出现的次数关联起来）。如下所示：

348-349 页程序

在 main()中，每次产生一个随机数字，它都会封装到一个 Integer 对象里，使句柄能够随同散列表一起使用（不可对一个集合使用基本数据类型，只能使用对象句柄）。containKey()方法检查这个键是否已经在集合里（也就是说，那个数字以前发现过吗？）若已在集合里，则 get()方法获得那个键关联的值，此时是一个 Counter（计数器）对象。计数器内的值 i 随后会增加 1，表明这个特定的随机数字又出现了一次。

假如键以前尚未发现过，那么方法 put()仍然会在散列表内置入一个新的“键－值”对。在创建之初，Counter 会自己的变量 i 自动初始化为 1，它标志着该随机数字的第一次出现。

为显示散列表，只需把它简单地打印出来即可。Hashtable toString()方法能遍历所有键－值对，并为每一对都调用 toString()。Integer toString()是事先定义好的，可看到计数器使用的 toString。一次运行的结果（添加了一些换行）如下：

350 页上程序

大家或许会对 Counter 类是否必要感到疑惑，它看起来似乎根本没有封装类 Integer 的功能。为什么不用 int 或 Integer 呢？事实上，由于所有集合能容纳的仅有对象句柄，所以根本不可以使用整数。学过集合后，封装类的概念对大家来说就可能更容易理解了，因为不可以将任何基本数据类型置入集合里。然而，我们对 Java 封装器能做的唯一事情就是将其初始化成一个特定的值，然后读取那个值。也就是说，一旦封装器对象已经创建，就没有办法改变一个值。这使得 Integer 封装器对解决我们的问题毫无意义，所以不得不创建一个新类，用它来满足自己的要求。

1. 创建“关键”类

在前面的例子里，我们用一个标准库的类（Integer）作为 Hashtable 的一个键使用。作为一个键，它能很好地工作，因为它已经具备正确运行的所有条件。但在使用散列表的时候，一旦我们创建自己的类作为键使用，就会遇到一个很常见的问题。例如，假设一套天气预报系统将 Groundhog（土拔鼠）对象匹配成 Prediction（预报）。这看起来非常直观：我们创建两个类，然后将 Groundhog 作为键使用，而将 Prediction 作为值使用。如下所示：

350-351 页程序

每个 Groundhog 都具有一个标识号码，所以赤了在散列表中查找一个 Prediction，只需指示它“告诉我与 Groundhog 号码 3 相关的 Prediction”。Prediction 类包含了一个布尔值，用 Math.random()进行初始化，以及一个 toString()为我们

解释结果。在 main()中，用 Groundhog 以及与它们相关的 Prediction 填充一个散列表。散列表被打印出来，以便我们看到它们确实已被填充。随后，用标识号码为 3 的一个 Groundhog 查找与 Groundhog #3 对应的预报。

看起来似乎非常简单，但实际是不可行的。问题在于 Groundhog 是从通用的 Object 根类继承的（若当初未指定基础类，则所有类最终都是从 Object 继承的）。事实上是用 Object 的 hashCode()方法生成每个对象的散列码，而且默认情况下只使用它的对象的地址。所以，Groundhog(3)的第一个实例并不会产生与 Groundhog(3)第二个实例相等的散列码，而我们用第二个实例进行检索。

大家或许认为此时要做的全部事情就是正确地覆盖 hashCode()。但这样做依然行不能，除非再做另一件事情：覆盖也属于 Object 一部分的 equals()。当散列表试图判断我们的键是否等于表内的某个键时，就会用到这个方法。同样地，默认的 Object.equals()只是简单地比较对象地址，所以一个 Groundhog(3)并不等于另一个 Groundhog(3)。

因此，为了在散列表中将自己的类作为键使用，必须同时覆盖 hashCode()和 equals()，就象下面展示的那样：

352 页程序

注意这段代码使用了来自前一个例子的 Prediction，所以 SpringDetector.java 必须首先编译，否则就会在试图编译 SpringDetector2.java 时得到一个编译期错误。

Groundhog2.hashCode()将土拔鼠号码作为一个标识符返回（在这个例子中，程序员需要保证没有两个土拔鼠用同样的 ID 号码并存）。为了返回一个独一无二的标识符，并不需要 hashCode()，equals()方法必须能够严格判断两个对象是否相等。

equals()方法要进行两种检查：检查对象是否为 null；若不为 null，则继续检查是否为 Groundhog2 的一个实例（要用到 instanceof 关键字，第 11 章会详加论述）。即使为了继续执行 equals()，它也应该是一个 Groundhog2。正如大家看到的那样，这种比较建立在实际 ghNumber 的基础上。这一次一旦我们运行程序，就会看到它终于产生了正确的输出（许多 Java 库的类都覆盖了 hashcode()和 equals()方法，以便与自己提供的内容适应）。

2. 属性：Hashtable 的一种类型

在本书的第一个例子中，我们使用了一个名为 Properties（属性）的 Hashtable 类型。在那个例子中，下述程序行：

```
Properties p = System.getProperties();
p.list(System.out);
```

调用了一个名为 getProperties()的 static 方法，用于获得一个特殊的 Properties 对象，对系统的某些特征进行描述。list()属于 Properties 的一个方法，可将内容发给我们选择的任何流式输出。也有一个 save()方法，可用它将属性列表写入一个文件，以便日后用 load()方法读取。

尽管 Properties 类是从 Hashtable 继承的，但它也包含了一个散列表，用于容纳“默认”属性的列表。所以假如没有在主列表里找到一个属性，就会自动搜索默认属性。

Properties 类亦可在我们的程序中使用（第 17 章的 ClassScanner.java 便是一例）。在 Java 库的用户文档中，往往可以找到更多、更详细的说明。

8.4.5 再论枚举器

我们现在可以开始演示 Enumeration（枚举）的真正威力：将穿越一个序列的操作与那个序列的基础结构分隔开。在下面的例子里，PrintData 类用一个 Enumeration 在一个序列中移动，并为每个对象都调用 toString()方法。此时创建了两个不同类型的集合：一个 Vector 和一个 Hashtable。并且在它们里面分别填充 Mouse 和 Hamster 对象（本章早些时候已定义了这些类；注意必须先编译 HamsterMaze.java 和 WorksAnyway.java，否则下面的程序不能编译）。由于 Enumeration 隐藏了基层集合的结构，所以 PrintData 不知道或者不关心 Enumeration 来自于什么类型的集合：

354 页程序

注意 PrintData.print()利用了这些集合中的对象属于 Object 类这一事实，所以它调用了 toString()。但在解决自己的实际问题时，经常都要保证自己的 Enumeration 穿越某种特定类型的集合。例如，可能要求集合中的所有元素都是一个 Shape（几何形状），并含有 draw()方法。若出现这种情况，必须从 Enumeration.nextElement()返回的 Object 进行下溯造型，以便产生一个 Shape。

8.5 排序

Java 1.0 和 1.1 库都缺少的一样东西是算术运算，甚至没有最简单的排序运算方法。因此，我们最好创建一个 Vector，利用经典的 Quicksort（快速排序）方法对其自身进行排序。

编写通用的排序代码时，面临的一个问题是必须根据对象的实际类型来执行比较运算，从而实现正确的排序。当然，一个办法是为每种不同的类型都写一个不同的排序方法。然而，应认识到假若这样做，以后增加新类型时便不易实现代码的重复利用。

程序设计一个主要的目标就是“将发生变化的东西同保持不变的东西分隔开”。在这里，保持不变的代码是通用的排序算法，而每次使用时都要变化的是对象的实际比较方法。因此，我们不可将比较代码“硬编码”到多个不同的排序例程内，而是采用“回调”技术。利用回调，经常发生变化的那部分代码会封装到它自己的类内，而总是保持相同的代码则“回调”发生变化的代码。这样一来，不同的对象就可以表达不同的比较方式，同时向它们传递相同的排序代码。

下面这个“接口”（Interface）展示了如何比较两个对象，它将那些“要发生变化的东西”封装在内：

355 页中程序

对这两种方法来说，lhs 代表本次比较中的“左手”对象，而 rhs 代表“右手”对象。

可创建 Vector 的一个子类，通过 Compare 实现“快速排序”。对于这种算法，包括它的速度以及原理等等，在此不具体说明。欲知详情，可参考 Binstock 和

Rex 编著的《Practical Algorithms for Programmers》，由 Addison-Wesley 于 1995 年出版。

355-356 页程序

现在，大家可以明白“回调”一词的来历，这是由于 quickSort()方法“往回调用”了 Compare 中的方法。从中亦可理解这种技术如何生成通用的、可重复利用（再生）的代码。

为使用 SortVector，必须创建一个类，令其为我们准备排序的对象实现 Compare。此时内部类并不显得特别重要，但对于代码的组织却是有益的。下面是针对 String 对象的一个例子：

356-357 页程序

内部类是“静态”（Static）的，因为它毋需连接一个外部类即可工作。

大家可以看到，一旦设置好框架，就可以非常方便地重复使用象这样的一个设计——只需简单地写一个类，将“需要发生变化”的东西封装进去，然后将一个对象传给 SortVector 即可。

比较时将字串强制为小写形式，所以大写 A 会排列于小写 a 的旁边，而不会移动一个完全不同的地方。然而，该例也显示了这种方法的一个不足，因为上述测试代码按照出现顺序排列同一个字母的大写和小写形式：A a b B c C d D。但这通常不是一个大问题，因为经常处理的都是更长的字串，所以上述效果不会显露出来（Java 1.2 的集合提供了排序功能，已解决了这个问题）。

继承（extends）在这儿用于创建一种新类型的 Vector——也就是说，SortVector 属于一种 Vector，并带有一些附加的功能。继承在这里可发挥很大的作用，但了带来了问题。它使一些方法具有了 final 属性（已在第 7 章讲述），所以不能覆盖它们。如果想创建一个排好序的 Vector，令其只接收和生成 String 对象，就会遇到麻烦。因为 addElement()和 elementAt()都具有 final 属性，而且它们都是我们必须覆盖的方法，否则便无法实现只能接收和产生 String 对象。

但在另一方面，请考虑采用“合成”方法：将一个对象置入一个新类的内部。此时，不是改写上述代码来达到这个目的，而是在新类里简单地使用一个 SortVector。在这种情况下，用于实现 Compare 接口的内部类就可以“匿名”地创建。如下所示：

358-359 页程序

这样便可快速再生来自 SortVector 的代码，从而获得希望的功能。然而，并不是来自 SortVector 和 Vector 的所有 public 方法都能在 StrSortVector 中出现。若按这种形式再生代码，可在新类里为包含类内的每一个方法都生成一个定义。当然，也可以在刚开始时只添加少数几个，以后根据需要再添加更多的。新类的设计最终会稳定下来。

这种方法的好处在于它仍然只接纳 String 对象，也只产生 String 对象。而且相应的检查是在编译期间进行的，而非在运行期。当然，只有 addElement()和 elementAt()才具备这一特性；elements()仍然会产生一个 Enumeration（枚举），它

在编译期的类型是未定的。当然，对 Enumeration 以及在 StrSortVector 中的类型检查会照旧进行；如果真的有什么错误，运行期间会简单地产生一个违例。事实上，我们在编译或运行期间能保证一切都正确无误吗？（也就是说，“代码测试时也许不能保证”，以及“该程序的用户有可能做一些未经我们测试的事情”）。尽管存在其他选择和争论，使用继承都要容易得多，只是在造型时让人深感不便。同样地，一旦为 Java 加入参数化类型，就有望解决这个问题。

大家在这个类中可以看到有一个名为“sorted”的标志。每次调用 addElement()时，都可对 Vector 进行排序，而且将其连续保持在一个排好序的状态。但在开始读取之前，人们总是向一个 Vector 添加大量元素。所以与其在每个 addElement()后排序，不如一直等到有人想读取 Vector，再对其进行排序。后者的效率要高得多。这种除非绝对必要，否则就不采取行动的方法叫作“懒惰求值”（还有一种类似的技术叫作“懒惰初始化”——除非真的需要一个字段值，否则不进行初始化）。

8.6 通用集合库

通过本章的学习，大家已知道标准 Java 库提供了一些特别有用的集合，但距完整意义的集合尚远。除此之外，象排序这样的算法根本没有提供支持。C++出色的一个地方就是它的库，特别是“标准模板库”（STL）提供了一套相当完整的集合，以及许多象排序和检索这样的算法，可以非常方便地对那些集合进行操作。有感这一现状，并以这个模型为基础，ObjectSpace 公司设计了 Java 版本的“通用集合库”（从前叫作“Java 通用库”，即 JGL；但 JGL 这个缩写形式侵犯了 Sun 公司的版权——尽管本书仍然沿用这个简称）。这个库尽可能遵照 STL 的设计（照顾到两种语言间的差异）。JGL 实现了许多功能，可满足对一个集合库的大多数常规需求，它与 C++的模板机制非常相似。JGL 包括相互链接起来的列表、设置、队列、映射、堆栈、序列以及反复器，它们的功能比 Enumeration（枚举）强多了。同时提供了一套完整的算法，如检索和排序等。在某些方面，ObjectSpace 的设计也显得比 Sun 的库设计方案“智能”一些。举个例子来说，JGL 集合中的方法不会进入 final 状态，所以很容易继承和改写那些方法。

JGL 已包括到一些厂商发行的 Java 套件中，而且 ObjectSpace 公司自己也允许所有用户免费使用 JGL，包括商业性的使用。详细情况和软件下载可访问 http://www.ObjectSpace.com。与 JGL 配套提供的联机文档做得非常好，可作为自己的一个绝佳起点使用。

8.7 新集合

对我来说，集合类属于最强大的一种工具，特别适合在原创编程中使用。大家可能已感觉到我对 Java 1.1 提供的集合多少有点儿失望。因此，看到 Java 1.2 对集合重新引起了正确的注意后，确实令人非常愉快。这个版本的集合也得到了完全的重新设计（由 Sun 公司的 Joshua Bloch）。我认为新设计的集合是 Java 1.2 中两项最主要的特性之一（另一项是 Swing 库，将在第 13 章叙述），因为它们极大方便了我们的编程，也使 Java 变成一种更成熟的编程系统。

有些设计使得元素间的结合变得更紧密，也更容易让人理解。例如，许多名字都变得更短、更明确了，而且更易使用；类型同样如此。有些名字进行了修改，更接近于通俗：我感觉特别好的一个是用“反复器”（Inerator）代替了“枚举”（Enumeration）。

此次重新设计也加强了集合库的功能。现在新增的行为包括链接列表、队列以及撤消组队（即“双终点队列”）。

集合库的设计是相当困难的（会遇到大量库设计问题）。在 C++中，STL 用多个不同的类来覆盖基础。这种做法比起 STL 以前是个很大的进步，那时根本没做这方面的考虑。但仍然没有很好地转换到 Java 里面。结果就是一大堆特别容易混淆的类。在另一个极端，我曾发现一个集合库由单个类构成：colleciton，它同时作为 Vector 和 Hashtable 使用。新集合库的设计者则希望达到一种新的平衡：实现人们希望从一个成熟集合库上获得的完整功能，同时又要比 STL 和其他类似的集合库更易学习和使用。这样得到的结果在某些场合显得有些古怪。但和早期 Java 库的一些决策不同，这些古怪之处并非偶然出现的，而是以复杂性作为代价，在进行仔细权衡之后得到的结果。这样做也许会延长人们掌握一些库概念的时间，但很快就会发现自己很乐于使用那些新工具，而且变得越来越离不了它。

新的集合库考虑到了“容纳自己对象”的问题，并将其分割成两个明确的概念：

(1) 集合（Collection）：一组单独的元素，通常应用了某种规则。在这里，一个 List（列表）必须按特定的顺序容纳元素，而一个 Set（集）不可包含任何重复的元素。相反，“包”（Bag）的概念未在新的集合库中实现，因为“列表”已提供了类似的功能。

(2) 映射（Map）：一系列“键－值”对（这已在散列表身上得到了充分的体现）。从表面看，这似乎应该成为一个“键－值”对的“集合”，但假若试图按那种方式实现它，就会发现实现过程相当笨拙。这进一步证明了应该分离成单独的概念。另一方面，可以方便地查看 Map 的某个部分。只需创建一个集合，然后用它表示那一部分即可。这样一来，Map 就可以返回自己键的一个 Set、一个包含自己值的 List 或者包含自己“键－值”对的一个 List。和数组相似，Map 可方便扩充到多个“维”，毋需涉及任何新概念。只需简单地在一个 Map 里包含其他 Map（后者又可以包含更多的 Map，以此类推）。

Collection 和 Map 可通过多种形式实现，具体由编程要求决定。下面列出的是一个帮助大家理解的新集合示意图：

363 页图

这张图刚开始的时候可能让人有点儿摸不着头脑，但在通读了本章以后，相信大家会真正理解它实际只有三个集合组件：Map，List 和 Set。而且每个组件实际只有两、三种实现方式（注释⑥），而且通常都只有一种特别好的方式。只要看出了这一点，集合就不会再令人生畏。

⑥：写作本章时，Java 1.2 尚处于 β 测试阶段，所以这张示意图没有包括以后会加入的 TreeSet。

虚线框代表“接口”，点线框代表“抽象”类，而实线框代表普通（实际）类。点线箭头表示一个特定的类准备实现一个接口（在抽象类的情况下，则是“部分”实现一个接口）。双线箭头表示一个类可生成箭头指向的那个类的对象。例

如，任何集合都可以生成一个反复器（Iterator），而一个列表可以生成一个ListIterator（以及原始的反复器，因为列表是从集合继承的）。

致力于容纳对象的接口是 Collection，List，Set 和 Map。在传统情况下，我们需要写大量代码才能同这些接口打交道。而且为了指定自己想使用的准确类型，必须在创建之初进行设置。所以可能创建下面这样的一个 List：

List x = new LinkedList();

当然，也可以决定将 x 作为一个 LinkedList 使用（而不是一个普通的 List），并用 x 负载准确的类型信息。使用接口的好处就是一旦决定改变自己的实施细节，要做的全部事情就是在创建的时候改变它，就象下面这样：

List x = new ArrayList();

其余代码可以保持原封不动。

在类的分级结构中，可看到大量以“Abstract”（抽象）开头的类，这刚开始可能会使人感觉迷惑。它们实际上是一些工具，用于“部分”实现一个特定的接口。举个例子来说，假如想生成自己的 Set，就不是从 Set 接口开始，然后自行实现所有方法。相反，我们可以从 AbstractSet 继承，只需极少的工作即可得到自己的新类。尽管如此，新集合库仍然包含了足够的功能，可满足我们的几乎所有需求。所以考虑到我们的目的，可忽略所有以“Abstract”开头的类。

因此，在观看这张示意图时，真正需要关心的只有位于最顶部的“接口”以及普通（实际）类——均用实线方框包围。通常需要生成实际类的一个对象，将其上溯造型为对应的接口。以后即可在代码的任何地方使用那个接口。下面是一个简单的例子，它用 String 对象填充一个集合，然后打印出集合内的每一个元素：

364-365 页程序

新集合库的所有代码示例都置于子目录 newcollections 下，这样便可提醒自己这些工作只对于 Java 1.2 有效。这样一来，我们必须用下述代码来调用程序：

java c08.newcollections.SimpleCollection

采用的语法与其他程序是差不多的。

大家可以看到新集合属于 java.util 库的一部分，所以在使用时不需要再添加任何额外的 import 语句。

main()的第一行创建了一个 ArrayList 对象，然后将其上溯造型成为一个集合。由于这个例子只使用了 Collection 方法，所以从 Collection 继承的一个类的任何对象都可以正常工作。但 ArrayList 是一个典型的 Collection，它代替了 Vector 的位置。

显然，add()方法的作用是将一个新元素置入集合里。然而，用户文档谨慎地指出 add()“保证这个集合包含了指定的元素”。这一点是为 Set 作铺垫的，后者只有在元素不存在的前提下才会真的加入那个元素。对于 ArrayList 以及其他任何形式的 List，add()肯定意味着“直接加入”。

利用 iterator()方法，所有集合都能生成一个“反复器”（Iterator）。反复器其实就象一个“枚举”（Enumeration），是后者的一个替代物，只是：

(1) 它采用了一个历史上默认、而且早在 OOP 中得到广泛采纳的名字（反复器）。

(2) 采用了比 Enumeration 更短的名字：hasNext()代替了 hasMoreElement()，而 next()代替了 nextElement()。

(3) 添加了一个名为 remove()的新方法，可删除由 Iterator 生成的上一个元素。所以每次调用 next()的时候，只需调用 remove()一次。

在 SimpleCollection.java 中，大家可看到创建了一个反复器，并用它在集合里遍历，打印出每个元素。

8.7.1 使用 Collections

下面这张表格总结了用一个集合能做的所有事情（亦可对 Set 和 List 做同样的事情，尽管 List 还提供了一些额外的功能）。Map 不是从 Collection 继承的，所以要单独对待。

366-367 页表

boolean add(Object) ＊保证集合内包含了自变量。如果它没有添加自变量，就返回 false（假）

boolean addAll(Collection) ＊添加自变量内的所有元素。如果没有添加元素，则返回 true（真）

void clear() ＊删除集合内的所有元素

boolean contains(Object) 若集合包含自变量，就返回“真”

boolean containsAll(Collection) 若集合包含了自变量内的所有元素，就返回“真”

boolean isEmpty() 若集合内没有元素，就返回“真”

Iterator iterator() 返回一个反复器，以用它遍历集合的各元素

boolean remove(Object) ＊如自变量在集合里，就删除那个元素的一个实例。如果已进行了删除，就返回“真”

boolean removeAll(Collection) ＊删除自变量里的所有元素。如果已进行了任何删除，就返回“真”

boolean retainAll(Collection) ＊只保留包含在一个自变量里的元素（一个理论的“交集”）。如果已进行了任何改变，就返回“真”

int size() 返回集合内的元素数量

Object[] toArray() 返回包含了集合内所有元素的一个数组

＊这是一个“可选的”方法，有的集合可能并未实现它。若确实如此，该方法就会遇到一个 UnsupportedOperatiionException，即一个“操作不支持”违例，详见第 9 章。

下面这个例子向大家演示了所有方法。同样地，它们只对从集合继承的东西有效，一个 ArrayList 作为一种“不常用的分母”使用：

367-369 页程序

通过第一个方法，我们可用测试数据填充任何集合。在当前这种情况下，只是将 int 转换成 String。第二个方法将在本章其余的部分经常采用。

newCollection()的两个版本都创建了 ArrayList，用于包含不同的数据集，并将它们作为集合对象返回。所以很明显，除了 Collection 接口之外，不会再用到其他什么。

print()方法也会在本节经常用到。由于它用一个反复器（Iterator）在一个集合内遍历，而任何集合都可以产生这样的一个反复器，所以它适用于 List 和 Set，也适用于由一个 Map 生成的 Collection。

main()用简单的手段显示出了集合内的所有方法。

在后续的小节里，我们将比较 List，Set 和 Map 的不同实现方案，同时指出在各种情况下哪一种方案应成为首选（带有星号的那个）。大家会发现这里并未包括一些传统的类，如 Vector，Stack 以及 Hashtable 等。因为不管在什么情况下，新集合内都有自己首选的类。

8.7.2 使用 Lists

369-370 页表

List（接口） 顺序是 List 最重要的特性；它可保证元素按照规定的顺序排列。List 为 Collection 添加了大量方法，以便我们在 List 中部插入和删除元素（只推荐对 LinkedList 这样做）。List 也会生成一个 ListIterator（列表反复器），利用它可在一个列表里朝两个方向遍历，同时插入和删除位于列表中部的元素（同样地，只建议对 LinkedList 这样做）

ArrayList＊ 由一个数组后推得到的 List。作为一个常规用途的对象容器使用，用于替换原先的 Vector。允许我们快速访问元素，但在从列表中部插入和删除元素时，速度却嫌稍慢。一般只应该用 ListIterator 对一个 ArrayList 进行向前和向后遍历，不要用它删除和插入元素；与 LinkedList 相比，它的效率要低许多

LinkedList 提供优化的顺序访问性能，同时可以高效率地在列表中部进行插入和删除操作。但在进行随机访问时，速度却相当慢，此时应换用 ArrayList。也提供了 addFirst()，addLast()，getFirst()，getLast()，removeFirst()以及 removeLast()（未在任何接口或基础类中定义），以便将其作为一个规格、队列以及一个双向队列使用

下面这个例子中的方法每个都覆盖了一组不同的行为：每个列表都能做的事情（basicTest()），通过一个反复器遍历（iterMotion()）、用一个反复器改变某些东西（iterManipulation()）、体验列表处理的效果（testVisual()）以及只有 LinkedList 才能做的事情等：

370-373 页程序

在 basicTest()和 iterMotiion()中，只是简单地发出调用，以便揭示出正确的语法。而且尽管捕获了返回值，但是并未使用它。在某些情况下，之所以不捕获返回值，是由于它们没有什么特别的用处。在正式使用它们前，应仔细研究一下自己的联机文档，掌握这些方法完整、正确的用法。

8.7.3 使用 Sets

Set 拥有与 Collection 完全相同的接口，所以和两种不同的 List 不同，它没有什么额外的功能。相反，Set 完全就是一个 Collection，只是具有不同的行为（这是实例和多形性最理想的应用：用于表达不同的行为）。在这里，一个 Set 只允

许每个对象存在一个实例（正如大家以后会看到的那样，一个对象的“值”的构成是相当复杂的）。

374 页表

Set（接口） 添加到 Set 的每个元素都必须是独一无二的；否则 Set 就不会添加重复的元素。添加到 Set 里的对象必须定义 equals()，从而建立对象的唯一性。Set 拥有与 Collection 完全相同的接口。一个 Set 不能保证自己可按任何特定的顺序维持自己的元素

HashSet＊ 用于除非常小的以外的所有 Set。对象也必须定义 hashCode()

ArraySet 由一个数组后推得到的 Set。面向非常小的 Set 设计，特别是那些需要频繁创建和删除的。对于小 Set，与 HashSet 相比，ArraySet 创建和反复所需付出的代价都要小得多。但随着 Set 的增大，它的性能也会大打折扣。不需要 HashCode()

TreeSet 由一个“红黑树”后推得到的顺序 Set（注释⑦）。这样一来，我们就可以从一个 Set 里提到一个顺序集合

⑦：直至本书写作的时候，TreeSet 仍然只是宣布，尚未正式实现。所以这里没有提供使用 TreeSet 的例子。

下面这个例子并没有列出用一个 Set 能够做的全部事情，因为接口与 Collection 是相同的，前例已经练习过了。相反，我们要例示的重点在于使一个 Set 独一无二的行为：

374-375 页程序

重复的值被添加到 Set，但在打印的时候，我们会发现 Set 只接受每个值的一个实例。

运行这个程序时，会注意到由 HashSet 维持的顺序与 ArraySet 是不同的。这是由于它们采用了不同的方法来保存元素，以便它们以后的定位。ArraySet 保持着它们的顺序状态，而 HashSet 使用一个散列函数，这是特别为快速检索设计的）。创建自己的类型时，一定要注意 Set 需要通过一种方式来维持一种存储顺序，就象本章早些时候展示的“groundhog”（土拔鼠）例子那样。下面是一个例子：

375-376 页程序

对 equals()及 hashCode()的定义遵照“groundhog”例子已经给出的形式。在两种情况下都必须定义一个 equals()。但只有要把类置入一个 HashSet 的前提下，才有必要使用 hashCode()——这种情况是完全有可能的，因为通常应先选择作为一个 Set 实现。

8.7.4 使用 Maps

376-377 页表

Map（接口） 维持“键－值”对应关系（对），以便通过一个键查找相应的值

HashMap＊ 基于一个散列表实现（用它代替 Hashtable）。针对“键－值”对的插入和检索，这种形式具有最稳定的性能。可通过构建器对这一性能进行调整，以便设置散列表的“能力”和“装载因子”

ArrayMap 由一个 ArrayList 后推得到的 Map。对反复的顺序提供了精确的控制。面向非常小的 Map 设计，特别是那些需要经常创建和删除的。对于非常小的 Map，创建和反复所付出的代价要比 HashMap 低得多。但在 Map 变大以后，性能也会相应地大幅度降低

TreeMap 在一个“红－黑”树的基础上实现。查看键或者“键－值”对时，它们会按固定的顺序排列（取决于 Comparable 或 Comparator，稍后即会讲到）。TreeMap 最大的好处就是我们得到的是已排好序的结果。TreeMap 是含有 subMap()方法的唯一一种 Map，利用它可以返回树的一部分

下例包含了两套测试数据以及一个 fill()方法，利用该方法可以用任何两维数组（由 Object 构成）填充任何 Map。这些工具也会在其他 Map 例子中用到。

377-379 页程序

printKeys()，printValues()以及 print()方法并不只是有用的工具，它们也清楚地揭示了一个 Map 的 Collection“景象”的产生过程。keySet()方法会产生一个 Set，它由 Map 中的键后推得来。在这儿，它只被当作一个 Collection 对待。values()也得到了类似的对待，它的作用是产生一个 List，其中包含了 Map 中的所有值（注意键必须是独一无二的，而值可以有重复）。由于这些 Collection 是由 Map 后推得到的，所以一个 Collection 中的任何改变都会在相应的 Map 中反映出来。

print()方法的作用是收集由 entries 产生的 Iterator（反复器），并用它同时打印出每个“键－值”对的键和值。程序剩余的部分提供了每种 Map 操作的简单示例，并对每种类型的 Map 进行了测试。

当创建自己的类，将其作为 Map 中的一个键使用时，必须注意到和以前的 Set 相同的问题。

8.7.5 决定实施方案

从早些时候的那幅示意图可以看出，实际上只有三个集合组件：Map，List 和 Set。而且每个接口只有两种或三种实施方案。若需使用由一个特定的接口提供的功能，如何才能决定到底采取哪一种方案呢？

为理解这个问题，必须认识到每种不同的实施方案都有自己的特点、优点和缺点。比如在那张示意图中，可以看到 Hashtable，Vector 和 Stack 的“特点”是它们都属于“传统”类，所以不会干扰原有的代码。但在另一方面，应尽量避免为新的（Java 1.2）代码使用它们。

其他集合间的差异通常都可归纳为它们具体是由什么“后推”的。换言之，取决于物理意义上用于实施目标接口的数据结构是什么。例如，ArrayList，LinkedList 以及 Vector（大致等价于 ArrayList）都实现了 List 接口，所以无论选用哪一个，我们的程序都会得到类似的结果。然而，ArrayList（以及 Vector）是

由一个数组后推得到的；而 LinkedList 是根据常规的双重链接列表方式实现的，因为每个单独的对象都包含了数据以及指向列表内前后元素的句柄。正是由于这个原因，假如想在一个列表中部进行大量插入和删除操作，那么 LinkedList 无疑是最恰当的选择（LinkedList 还有一些额外的功能，建立于 AbstractSequentialList 中）。若非如此，就情愿选择 ArrayList，它的速度可能要快一些。

作为另一个例子，Set 既可作为一个 ArraySet 实现，亦可作为 HashSet 实现。ArraySet 是由一个 ArrayList 后推得到的，设计成只支持少量元素，特别适合要求创建和删除大量 Set 对象的场合使用。然而，一旦需要在自己的 Set 中容纳大量元素，ArraySet 的性能就会大打折扣。写一个需要 Set 的程序时，应默认选择 HashSet。而且只有在某些特殊情况下（对性能的提升有迫切的需求），才应切换到 ArraySet。

1. 决定使用何种 List

为体会各种 List 实施方案间的差异，最简便的方法就是进行一次性能测验。下述代码的作用是建立一个内部基础类，将其作为一个测试床使用。然后为每次测验都创建一个匿名内部类。每个这样的内部类都由一个 test()方法调用。利用这种方法，可以方便添加和删除测试项目。

380-382 页程序

内部类 Tester 是一个抽象类，用于为特定的测试提供一个基础类。它包含了一个要在测试开始时打印的字串、一个用于计算测试次数或元素数量的 size 参数、用于初始化字段的一个构建器以及一个抽象方法 test()。test()做的是最实际的测试工作。各种类型的测试都集中到一个地方：tests 数组。我们用继承于 Tester 的不同匿名内部类来初始化该数组。为添加或删除一个测试项目，只需在数组里简单地添加或移去一个内部类定义即可，其他所有工作都是自动进行的。

首先用元素填充传递给 test()的 List，然后对 tests 数组中的测试计时。由于测试用机器的不同，结果当然也会有所区别。这个程序的宗旨是揭示出不同集合类型的相对性能比较。下面是某一次运行得到的结果：

类型 获取 反复 插入 删除
ArrayList 110 270 1920 4780
LinkedList 1870 7580 170 110

可以看出，在 ArrayList 中进行随机访问（即 get()）以及循环反复是最划得来的；但对于 LinkedList 却是一个不小的开销。但另一方面，在列表中部进行插入和删除操作对于 LinkedList 来说却比 ArrayList 划算得多。我们最好的做法也许是先选择一个 ArrayList 作为自己的默认起点。以后若发现由于大量的插入和删除造成了性能的降低，再考虑换成 LinkedList 不迟。

2. 决定使用何种 Set

可在 ArraySet 以及 HashSet 间作出选择，具体取决于 Set 的大小（如果需要从一个 Set 中获得一个顺序列表，请用 TreeSet；注释⑧）。下面这个测试程序将有助于大家作出这方面的抉择：

383-384 页程序

⑧：TreeSet 在本书写作时尚未成为一个正式的特性，但在这个例子中可以很轻松地为其添加一个测试。

最后对 ArraySet 的测试只有 500 个元素，而不是 1000 个，因为它太慢了。

类型 测试大小 添加 包含 反复

384-385 页表略

进行 add()以及 contains()操作时，HashSet 显然要比 ArraySet 出色得多，而且性能明显与元素的多寡关系不大。一般编写程序的时候，几乎永远用不着使用 ArraySet。

3. 决定使用何种 Map

选择不同的 Map 实施方案时，注意 Map 的大小对于性能的影响是最大的，下面这个测试程序清楚地阐示了这一点：

385-387 页程序

由于 Map 的大小是最严重的问题，所以程序的计时测试按 Map 的大小（或容量）来分割时间，以便得到令人信服的测试结果。下面列出一系列结果（在你的机器上可能不同）：

类型 测试大小 置入 取出 反复

387 页表略

即使大小为 10，ArrayMap 的性能也要比 HashMap 差——除反复循环时以外。而在使用 Map 时，反复的作用通常并不重要（get()通常是我们时间花得最多的地方）。TreeMap 提供了出色的 put()以及反复时间，但 get()的性能并不佳。但是，我们为什么仍然需要使用 TreeMap 呢？这样一来，我们可以不把它作为 Map 使用，而作为创建顺序列表的一种途径。树的本质在于它总是顺序排列的，不必特别进行排序（它的排序方式马上就要讲到）。一旦填充了一个 TreeMap，就可以调用 keySet()来获得键的一个 Set“景象”。然后用 toArray()产生包含了那些键的一个数组。随后，可用 static 方法 Array.binarySearch()快速查找排好序的数组中的内容。当然，也许只有在 HashMap 的行为不可接受的时候，才需要采用这种做法。因为 HashMap 的设计宗旨就是进行快速的检索操作。最后，当我们使用 Map 时，首要的选择应该是 HashMap。只有在极少数情况下才需要考虑其他方法。

此外，在上面那张表里，有另一个性能问题没有反映出来。下述程序用于测试不同类型 Map 的创建速度：

387-388 页程序

在写这个程序期间，TreeMap 的创建速度比其他两种类型明显快得多（但你应亲自尝试一下，因为据说新版本可能会改善 ArrayMap 的性能）。考虑到这方面的原因，同时由于前述 TreeMap 出色的 put()性能，所以如果需要创建大量 Map，而且只有在以后才需要涉及大量检索操作，那么最佳的策略就是：创建和填充 TreeMap；以后检索量增大的时候，再将重要的 TreeMap 转换成 HashMap——使用 HashMap(Map)构建器。同样地，只有在事实证明确实存在性能瓶颈后，才应关心这些方面的问题——先用起来，再根据需要加快速度。

8.7.6 未支持的操作

利用 static（静态）数组 Arrays.toList()，也许能将一个数组转换成 List，如下所示：

388-389 页程序

从中可以看出，实际只实现了 Collection 和 List 接口的一部分。剩余的方法导致了不受欢迎的一种情况，名为 UnsupportedOperationException。在下一章里，我们会讲述违例的详细情况，但在这里有必要进行一下简单说明。这里的关键在于“集合接口”，以及新集合库内的另一些接口，它们都包含了“可选的”方法。在实现那些接口的集合类中，或者提供、或者没有提供对那些方法的支持。若调用一个未获支持的方法，就会导致一个 UnsupportedOperationException（操作未支持违例），这表明出现了一个编程错误。

大家或许会觉得奇怪，不是说“接口”和基础类最大的“卖点”就是它们许诺这些方法能产生一些有意义的行为吗？上述违例破坏了那个许诺——它调用的一部分方法不仅不能产生有意义的行为，而且还会中止程序的运行。在这些情况下，类型的所谓安全保证似乎显得一钱不值！但是，情况并没有想象的那么坏。通过 Collection，List，Set 或者 Map，编译器仍然限制我们只能调用那个接口中的方法，所以它和 Smalltalk 还是存在一些区别的（在 Smalltalk 中，可为任何对象调用任何方法，而且只有在运行程序时才知道这些调用是否可行）。除此以外，以 Collection 作为自变量的大多数方法只能从那个集合中读取数据——Collection 的所有“read”方法都不是可选的。

这样一来，系统就可避免在设计期间出现接口的冲突。而在集合库的其他设计方案中，最终经常都会得到数量过多的接口，用它们描述基本方案的每一种变化形式，所以学习和掌握显得非常困难。有些时候，甚至难于捕捉接口中的所有特殊情况，因为人们可能设计出任何新接口。但 Java 的“不支持的操作”方法却达到了新集合库的一个重要设计目标：易于学习和使用。但是，为了使这一方法真正有效，却需满足下述条件：

(1) UnsupportedOperationException 必须属于一种“非常”事件。也就是说，对于大多数类来说，所有操作都应是可行的。只有在一些特殊情况下，一、两个操作才可能未获支持。新集合库满足了这一条件，因为绝大多数时候用到的类——ArrayList，LinkedList，HashList 和 HashMap，以及其他集合方案——都提供了对所有操作的支持。但是，如果想新建一个集合，同时不想为集合接口中的所

有方法都提供有意义的定义，同时令其仍与现有库配合，这种设计方法也确实提供了一个“后门”可以利用。

(2) 若一个操作未获支持，那么 UnsupportedOperationException（未支持的操作违例）极有可能在实现期间出现，则不是在产品已交付给客户以后才会出现。它毕竟指出的是一个编程错误——不正确地使用了一个类。这一点不能十分确定，通过也可以看出这种方案的“试验”特征——只有经过多次试验，才能找出最理想的工作方式。

在上面的例子中，Arrays.toList()产生了一个 List（列表），该列表是由一个固定长度的数组后推出来的。因此唯一能够支持的就是那些不改变数组长度的操作。在另一方面，若请求一个新接口表达不同种类的行为（可能叫作“FixedSizeList”——固定长度列表），就有遭遇更大的复杂程度的危险。这样一来，以后试图使用库的时候，很快就会发现自己不知从何处下手。

对那些采用 Collection，List，Set 或者 Map 作为参数的方法，它们的文档应当指出哪些可选的方法是必须实现的。举个例子来说，排序要求实现 set()和 Iterator.set()方法，但不包括 add()和 remove()。

8.7.7 排序和搜索

Java 1.2 添加了自己的一套实用工具，可用来对数组或列表进行排列和搜索。这些工具都属于两个新类的“静态”方法。这两个类分别是用于排序和搜索数组的 Arrays，以及用于排序和搜索列表的 Collections。

1. 数组

Arrays 类为所有基本数据类型的数组提供了一个过载的 sort()和 binarySearch()，它们亦可用于 String 和 Object。下面这个例子显示出如何排序和搜索一个字节数组（其他所有基本数据类型都是类似的）以及一个 String 数组：

391-392 页程序

类的第一部分包含了用于产生随机字串对象的实用工具，可供选择的随机字母保存在一个字符数组中。randString()返回一个任意长度的字串；而 readStrings()创建随机字串的一个数组，同时给定每个字串的长度以及希望的数组大小。两个 print()方法简化了对示范数组的显示。在 main()中，Random.nextBytes()用随机选择的字节填充数组自变量（没有对应的 Random 方法用于创建其他基本数据类型的数组）。获得一个数组后，便可发现为了执行 sort()或者 binarySearch()，只需发出一次方法调用即可。与 binarySearch()有关的还有一个重要的警告：若在执行一次 binarySearch()之前不调用 sort()，便会发生不可预测的行为，其中甚至包括无限循环。

对 String 的排序以及搜索是相似的，但在运行程序的时候，我们会注意到一个有趣的现象：排序遵守的是字典顺序，亦即大写字母在字符集中位于小写字母的前面。因此，所有大写字母都位于列表的最前面，后面再跟上小写字母——Z 居然位于 a 的前面。似乎连电话簿也是这样排序的。

2. 可比较与比较器

但假若我们不满足这一排序方式，又该如何处理呢？例如本书后面的索引，如果必须对以 A 或 a 开头的词条分别到两处地方查看，那么肯定会使读者颇不耐烦。

若想对一个 Object 数组进行排序，那么必须解决一个问题。根据什么来判定两个 Object 的顺序呢？不幸的是，最初的 Java 设计者并不认为这是一个重要的问题，否则就已经在根类 Object 里定义它了。这样造成的一个后果便是：必须从外部进行 Object 的排序，而且新的集合库提供了实现这一操作的标准方式（最理想的是在 Object 里定义它）。

针对 Object 数组（以及 String，它当然属于 Object 的一种），可使用一个 sort()，并令其接纳另一个参数：实现了 Comparator 接口（即“比较器”接口，新集合库的一部分）的一个对象，并用它的单个 compare()方法进行比较。这个方法将两个准备比较的对象作为自己的参数使用——若第一个参数小于第二个，返回一个负整数；若相等，返回零；若第一个参数大于第二个，则返回正整数。基于这一规则，上述例子的 String 部分便可重新写过，令其进行真正按字母顺序的排序：

394 页上程序

通过造型为 String，compare()方法会进行“暗示”性的测试，保证自己操作的只能是 String 对象——运行期系统会捕获任何差错。将两个字串都强迫换成小写形式后，String.compareTo()方法会产生预期的结果。

若用自己的 Comparator 来进行一次 sort()，那么在使用 binarySearch()时必须使用那个相同的 Comparator。

Arrays 类提供了另一个 sort()方法，它会采用单个自变量：一个 Object 数组，但没有 Comparator。这个 sort()方法也必须用同样的方式来比较两个 Object。通过实现 Comparable 接口，它采用了赋予一个类的“自然比较方法”。这个接口含有单独一个方法——compareTo()，能分别根据它小于、等于或者大于自变量而返回负数、零或者正数，从而实现对象的比较。下面这个例子简单地阐示了这一点：

394-395 页程序

当然，我们的 compareTo()方法亦可根据实际情况增大复杂程度。

3. 列表

可用与数组相同的形式排序和搜索一个列表（List）。用于排序和搜索列表的静态方法包含在类 Collections 中，但它们拥有与 Arrays 中差不多的签名：sort(List)用于对一个实现了 Comparable 的对象列表进行排序；binarySearch(List,Object)用于查找列表中的某个对象；sort(List,Comparator)利用一个“比较器”对一个列表进行排序；而 binarySearch(List,Object,Comparator)则用于查找那个列表中的一个对象（注释⑨）。下面这个例子利用了预先定义好的 CompClass 和 AlphaComp 来示范 Collections 中的各种排序工具：

396 页程序

⑨：在本书写作时，已宣布了一个新的 Collections.stableSort()，可用它进行合并式排序，但还没有它的测试版问世。

这些方法的用法与在 Arrays 中的用法是完全一致的，只是用一个列表代替了数组。

TreeMap 也必须根据 Comparable 或者 Comparator 对自己的对象进行排序。

8.7.8 实用工具

Collections 类中含有其他大量有用的实用工具：

397 页表

enumeration(Collection) 为自变量产生原始风格的 Enumeration（枚举）

max(Collection)，min(Collection) 在自变量中用集合内对象的自然比较方法产生最大或最小元素

max(Collection,Comparator)，min(Collection,Comparator) 在集合内用比较器产生最大或最小元素

nCopies(int n, Object o) 返回长度为 n 的一个不可变列表，它的所有句柄均指向 o

subList(List,int min,int max) 返回由指定参数列表后推得到的一个新列表。可将这个列表想象成一个“窗口”，它自索引为 min 的地方开始，正好结束于 max 的前面

注意 min()和 max()都是随同 Collection 对象工作的，而非随同 List，所以不必担心 Collection 是否需要排序（就象早先指出的那样，在执行一次 binarySearch()——即二进制搜索——之前，必须对一个 List 或者一个数组执行 sort()）。

1. 使 Collection 或 Map 不可修改

通常，创建 Collection 或 Map 的一个“只读”版本显得更有利一些。Collections 类允许我们达到这个目标，方法是将原始容器传递进入一个方法，并令其传回一个只读版本。这个方法共有四种变化形式，分别用于 Collection（如果不想把集合当作一种更特殊的类型对待）、List、Set 以及 Map。下面这个例子演示了为它们分别构建只读版本的正确方法：

397-398 页程序

对于每种情况，在将其正式变为只读以前，都必须用有有效的数据填充容器。一旦载入成功，最佳的做法就是用“不可修改”调用产生的句柄替换现有的句柄。这样做可有效避免将其变成不可修改后不慎改变其中的内容。在另一方面，该工具也允许我们在一个类中将能够修改的容器保持为 private 状态，并可从一个方法调用中返回指向那个容器的一个只读句柄。这样一来，虽然我们可在类里修改它，但其他任何人都只能读。

为特定类型调用“不可修改”的方法不会造成编译期间的检查，但一旦发生

任何变化，对修改特定容器的方法的调用便会产生一个UnsupportedOperationException 违例。

2. Collection 或 Map 的同步

synchronized 关键字是“多线程”机制一个非常重要的部分。我们到第 14 章才会对这一机制作深入的探讨。在这儿，大家只需注意到 Collections 类提供了对整个容器进行自动同步的一种途径。它的语法与“不可修改”的方法是类似的：

399 页程序

在这种情况下，我们通过适当的“同步”方法直接传递新容器；这样做可避免不慎暴露出未同步的版本。

新集合也提供了能防止多个进程同时修改一个容器内容的机制。若在一个容器里反复，同时另一些进程介入，并在那个容器中插入、删除或修改一个对象，便会面临发生冲突的危险。我们可能已传递了那个对象，可能它位位于我们前面，可能容器的大小在我们调用 size()后已发生了收缩——我们面临各种各样可能的危险。针对这个问题，新的集合库集成了一套解决的机制，能查出除我们的进程自己需要负责的之外的、对容器的其他任何修改。若探测到有其他方面也准备修改容器，便会立即产生一个 ConcurrentModificationException（并发修改违例）。我们将这一机制称为“立即失败”——它并不用更复杂的算法在“以后”侦测问题，而是“立即”产生违例。

8.8 总结

下面复习一下由标准 Java（1.0 和 1.1）库提供的集合（BitSet 未包括在这里，因为它更象一种负有特殊使命的类）：

(1) 数组包含了对象的数字化索引。它容纳的是一种已知类型的对象，所以在查找一个对象时，不必对结果进行造型处理。数组可以是多维的，而且能够容纳基本数据类型。但是，一旦把它创建好以后，大小便不能变化了。

(2) Vector（矢量）也包含了对象的数字索引——可将数组和 Vector 想象成随机访问集合。当我们加入更多的元素时，Vector 能够自动改变自身的大小。但 Vector 只能容纳对象的句柄，所以它不可包含基本数据类型；而且将一个对象句柄从集合中取出来的时候，必须对结果进行造型处理。

(3) Hashtable（散列表）属于 Dictionary（字典）的一种类型，是一种将对象（而不是数字）同其他对象关联到一起的方式。散列表也支持对对象的随机访问，事实上，它的整个设计方案都在突出访问的“高速度”。

(4) Stack（堆栈）是一种“后入先出”（LIFO）的队列。

若你曾经熟悉数据结构，可能会疑惑为何没看到一套更大的集合。从功能的角度出发，你真的需要一套更大的集合吗？对于 Hashtable，可将任何东西置入其中，并以非常快的速度检索；对于 Enumeration（枚举），可遍历一个序列，并对其中的每个元素都采取一个特定的操作。那是一种功能足够强劲的工具。

但 Hashtable 没有“顺序”的概念。Vector 和数组为我们提供了一种线性顺序，但若要把一个元素插入它们任何一个的中部，一般都要付出“惨重”的代价。除此以外，队列、拆散队列、优先级队列以及树都涉及到元素的“排序”——并

非仅仅将它们置入，以便以后能按线性顺序查找或移动它们。这些数据结构也非常有用，这也正是标准 C++中包含了它们的原因。考虑到这个原因，只应将标准 Java 库的集合看作自己的一个起点。而且倘若必须使用 Java 1.0 或 1.1，则可在需要超越它们的时候使用 JGL。

如果能使用 Java 1.2，那么只使用新集合即可，它一般能满足我们的所有需要。注意本书在 Java 1.1 身上花了大量篇幅，所以书中用到的大量集合都是只能在 Java1.1 中用到的那些：Vector 和 Hashtable。就目前来看，这是一个不得以而为之的做法。但是，这样处理亦可提供与老 Java 代码更出色的向后兼容能力。若要用 Java1.2 写新代码，新的集合往往能更好地为你服务。

8.9 练习

(1) 新建一个名为 Gerbil 的类，在构建器中初始化一个 int gerbilNumber（类似本章的 Mouse 例子）。为其写一个名为 hop()的方法，用它打印出符合 hop()条件的 Gerbil 的编号。建一个 Vector，并为 Vector 添加一系列 Gerbil 对象。现在，用 elementAt()方法在 Vector 中遍历，并为每个 Gerbil 都调用 hop()。

(2) 修改练习 1，用 Enumeration 在调用 hop()的同时遍历 Vector。

(3) 在 AssocArray.java 中，修改这个例子，令其使用一个 Hashtable，而不是 AssocArray。

(4) 获取练习 1 用到的 Gerbil 类，改为把它置入一个 Hashtable，然后将 Gerbil 的名称作为一个 String（键）与置入表格的每个 Gerbil（值）都关联起来。获得用于 keys()的一个 Enumeration，并用它在 Hashtable 里遍历，查找每个键的 Gerbil，打印出键，然后将 gerbil 告诉给 hop()。

(5) 修改第 7 章的练习 1，用一个 Vector 容纳 Rodent（啮齿动物），并用 Enumeration 在 Rodent 序列中遍历。记住 Vector 只能容纳对象，所以在访问单独的 Rodent 时必须采用一个造型（如 RTTI）。

(6) 转到第 7 章的中间位置，找到那个 GreenhouseControls.java（温室控制）例子，该例应该由三个文件构成。在 Controller.java 中，类 EventSet 仅是一个集合。修改它的代码，用一个 Stack 代替 EventSet。当然，这时可能并不仅仅用 Stack 取代 EventSet 这样简单；也需要用一个 Enumeration 遍历事件集。可考虑在某些时候将集合当作 Stack 对待，另一些时候则当作 Vector 对待——这样或许能使事情变得更加简单。

(7) （有一定挑战性）在与所有 Java 发行包配套提供的 Java 源码库中找出用于 Vector 的源码。复制这些代码，制作名为 intVector 的一个特殊版本，只在其中包含 int 数据。思考是否能为所有基本数据类型都制作 Vector 的一个特殊版本。接下来，考虑假如制作一个链接列表类，令其能随同所有基本数据类型使用，那么会发生什么情况。若在 Java 中提供了参数化类型，利用它们便可自动完成这一工作（还有其他许多好处）。