第 10 章 Java IO 系统

“对语言设计人员来说，创建好的输入 / 输出系统是一项特别困难的任务。”

由于存在大量不同的设计方案，所以该任务的困难性是很容易证明的。其中最大的挑战似乎是如何覆盖所有可能的因素。不仅有三种不同的种类的 I/O 需要考虑（文件、控制台、网络连接），而且需要通过大量不同的方式与它们通信（顺序、随机访问、二进制、字符、按行、按字等等）。

Java 库的设计者通过创建大量类来攻克这个难题。事实上，Java 的 I/O 系统采用了如此多的类，以致刚开始会产生不知从何处入手的感觉（具有讽刺意味的是，Java 的 I/O 设计初衷实际要求避免过多的类）。从 Java 1.0 升级到 Java 1.1 后，I/O 库的设计也发生了显著的变化。此时并非简单地用新库替换旧库，Sun 的设计人员对原来的库进行了大手笔的扩展，添加了大量新的内容。因此，我们有时不得不混合使用新库与旧库，产生令人无奈的复杂代码。

本章将帮助大家理解标准 Java 库内的各种 I/O 类，并学习如何使用它们。本章的第一部分将介绍“旧”的 Java 1.0 I/O 流库，因为现在有大量代码仍在使用那个库。本章剩下的部分将为大家引入 Java 1.1 I/O 库的一些新特性。注意若用 Java 1.1 编译器来编译本章第一部分介绍的部分代码，可能会得到一条“不建议使用该特性”（Deprecated feature）警告消息。代码仍然能够使用；编译器只是建议我们换用本章后面要讲述的一些新特性。但我们这样做是有价值的，因为可以更清楚地认识老方法与新方法之间的一些差异，从而加深我们的理解（并可顺利阅读为 Java 1.0 写的代码）。

10.1 输入和输出

可将 Java 库的 I/O 类分割为输入与输出两个部分，这一点在用 Web 浏览器阅读联机 Java 类文档时便可知道。通过继承，从 **InputStream**（输入流）衍生的所有类都拥有名为 **read()**的基本方法，用于读取**单个字节或者字节数组**。类似地，从 **OutputStream** 衍生的所有类都拥有基本方法 **write()**，用于写入单个字节或者字节数组。然而，我们通常不会用到这些方法；它们之所以存在，是因为更复杂的类可以利用它们，以便提供一个更有用的接口。因此，我们很少用单个类创建自己的系统对象。一般情况下，我们都是将多个对象重叠在一起，提供自己期望的功能。我们之所以感到 Java 的流库（Stream Library）异常复杂，正是由于为了创建单独一个结果流，却需要创建多个对象的缘故。

很有必要按照功能对类进行分类。库的设计者首先决定与输入有关的所有类都从 InputStream 继承，而与输出有关的所有类都从 OutputStream 继承。

10.1.1 InputStream 的类型

InputStream 的作用是标志那些从不同起源地产生输入的类。这些起源地包括（每个都有一个相关的 InputStream 子类）：

(1) 字节数组

(2) String 对象

(3) 文件

(4) “管道”，它的工作原理与现实生活中的管道类似：将一些东西置入一端，它们在另一端出来。 (5) 一系列其他流，以便我们将其统一收集到单独一个流内。

(6) 其他起源地，如 Internet 连接等（将在本书后面的部分讲述）。

除此以外，**FilterInputStream** 也属于 InputStream 的一种类型，用它可为“破坏器”类提供一个基础类，以便将属性或者有用的接口同输入流连接到一起。这将在以后讨论。

表 10.1 InputStream 的类型

类 功能 构建器参数 / 如何使用

ByteArrayInputStream 允许内存中的一个缓冲区作为 InputStream 使用 从中提取字节的缓冲区 / 作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口

StringBufferInputStream 将一个 String 转换成 InputStream 一个 String（字串）。基础的实施方案实际采用一个 StringBuffer（字串缓冲） / 作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口

FileInputStream 用于从文件读取信息 代表文件名的一个 String，或者一个 File 或 FileDescriptor 对象 / 作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口

PipedInputString 产生为相关的 PipedOutputStream 写的数据。实现了“管道化”的概念 PipedOutputStream / 作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口

SequenceInputStream 将两个或更多的 InputStream 对象转换成单个 InputStream 使用 两个 InputStream 对象或者一个 Enumeration，用于 InputStream 对象的一个容器 / 作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口

FilterInputStream 对作为破坏器接口使用的类进行抽象；那个破坏器为其他 InputStream 类提供了有用的功能。参见表 10.3 参见表 10.3 / 参见表 10.3

10.1.2 OutputStream 的类型

这一类别包括的类决定了我们的输入往何处去：一个字节数组（但没有 String；假定我们可用字节数组创建一个）；一个文件；或者一个“管道”。

除此以外，FilterOutputStream 为“破坏器”类提供了一个基础类，它将属性或者有用的接口同输出流连接起来。这将在以后讨论。

表 10.2 OutputStream 的类型

类 功能 构建器参数 / 如何使用

ByteArrayOutputStream 在内存中创建一个缓冲区。我们发送给流的所有数据都会置入这个缓冲区。 可选缓冲区的初始大小 / 用于指出数据的目的地。若将其同 FilterOutputStream 对象连接到一起，可提供一个有用的接口

FileOutputStream 将信息发给一个文件 用一个 String 代表文件名，或选用一个 File 或 FileDescriptor 对象 / 用于指出数据的目的地。若将其同 FilterOutputStream 对象连接到一起，可提供一个有用的接口

PipedOutputStream 我们写给它的任何信息都会自动成为相关的 PipedInputStream 的输出。实现了“管道化”的概念 PipedInputStream / 为多线程处理指出自己数据的目的地 / 将其同 FilterOutputStream 对象连接到一起，便可提供一个有用的接口

FilterOutputStream 对作为破坏器接口使用的类进行抽象处理；那个破坏器为其他 OutputStream 类提供了有用的功能。参见表 10.4 参见表 10.4 / 参见表 10.4

10.2 增添属性和有用的接口

利用层次化对象动态和透明地添加单个对象的能力的做法叫作“装饰器”（**Decorator**）方案——“方案”属于本书第 16 章的主题（注释①）。装饰器方案规定封装于初始化对象中的所有对象都拥有相同的接口，以便利用装饰器的“透明”性质——我们将相同的消息发给一个对象，无论它是否已被“装饰”。这正是在 Java I/O 库里存在“过滤器”（Filter）类的原因：抽象的“过滤器”类是所有装饰器的基础类（装饰器必须拥有与它装饰的那个对象相同的接口，但装饰器亦可对接口作出扩展，这种情况见诸于几个特殊的“过滤器”类中）。

子类处理要求大量子类对每种可能的组合提供支持时，便经常会用到装饰器——由于组合形式太多，造成子类处理变得不切实际。Java I/O 库要求许多不同的特性组合方案，这正是装饰器方案显得特别有用的原因。但是，装饰器方案也有自己的一个缺点。在我们写一个程序的时候，装饰器为我们提供了大得多的灵活性（因为可以方便地混合与匹配属性），但它们也使自己的代码变得更加复杂。

原因在于 Java I/O 库操作不便，我们必须创建许多类——“核心”I/O 类型加上所有装饰器——才能得到自己希望的单个 I/O 对象。

FilterInputStream 和 FilterOutputStream（这两个名字不十分直观）提供了相应的装饰器接口，用于控制一个特定的输入流（InputStream）或者输出流（OutputStream）。它们分别是从 InputStream 和 OutputStream 衍生出来的。此外，它们都属于抽象类，在理论上为我们与一个流的不同通信手段都提供了一个通用的接口。事实上，FilterInputStream 和 FilterOutputStream 只是简单地模仿了自己的基础类，它们是一个装饰器的基本要求。

10.2.1 通过 FilterInputStream 从 InputStream 里读入数据

FilterInputStream 类要完成两件全然不同的事情。其中，DataInputStream 允许我们读取不同的基本类型数据以及 String 对象（所有方法都以“read”开头，比如 readByte()，readFloat()等等）。伴随对应的 DataOutputStream，我们可通过数据“流”将基本类型的数据从一个地方搬到另一个地方。这些“地方”是由表 10.1 总结的那些类决定的。若读取块内的数据，并自己进行解析，就不需要用到 DataInputStream。但在其他许多情况下，我们一般都想用它对自己读入的数据进行自动格式化。

剩下的类用于修改 InputStream 的内部行为方式：是否进行缓冲，是否跟踪自己读入的数据行，以及是否能够推回一个字符等等。后两种类看起来特别象提供对构建一个编译器的支持（换言之，添加它们为了支持 Java 编译器的构建），所以在常规编程中一般都用不着它们。

也许几乎每次都要缓冲自己的输入，无论连接的是哪个 IO 设备。所以 IO 库最明智的做法就是将未缓冲输入作为一种特殊情况处理，同时将缓冲输入接纳为标准做法。

表 10.3 FilterInputStream 的类型

类 功能 构建器参数 / 如何使用

DataInputStream 与 DataOutputStream 联合使用，使自己能以机动方式读取一个流中的基本数据类型（int，char，long 等等） InputStream/包含了一个完整的接口，以便读取基本数据类型

BufferedInputStream 避免每次想要更多数据时都进行物理性的读取，告诉它“请先在缓冲区里找” InputStream，没有可选的缓冲区大小 / 本身并不能提供一个接口，只是发出使用缓冲区的要求。要求同一个接口对象连接到一起

LineNumberInputStream 跟踪输入流中的行号；可调用 getLineNumber()以及 setLineNumber(int) 只是添加对数据行编号的能力，所以可能需要同一个真正的接口对象连接

PushbackInputStream 有一个字节的后推缓冲区，以便后推读入的上一个字符 InputStream / 通常由编译器在扫描器中使用，因为 Java 编译器需要它。一般不在自己的代码中使用

10.2.2 通过 FilterOutputStream 向 OutputStream 里写入数据

与 DataInputStream 对应的是 DataOutputStream，后者对各个基本数据类型以

及 String 对象进行格式化，并将其置入一个数据“流”中，以便任何机器上的 DataInputStream 都能正常地读取它们。所有方法都以“wirte”开头，例如 writeByte()，writeFloat()等等。

若想进行一些真正的格式化输出，比如输出到控制台，请使用 PrintStream。利用它可以打印出所有基本数据类型以及 String 对象，并可采用一种易于查看的格式。这与 DataOutputStream 正好相反，后者的目标是将那些数据置入一个数据流中，以便 DataInputStream 能够方便地重新构造它们。System.out 静态对象是一个 PrintStream。

PrintStream 内两个重要的方法是 print()和 println()。它们已进行了覆盖处理，可打印出所有数据类型。print()和 println()之间的差异是后者在操作完毕后会自动添加一个新行。

BufferedOutputStream 属于一种“修改器”，用于指示数据流使用缓冲技术，使自己不必每次都向流内物理性地写入数据。通常都应将它应用于文件处理和控制器 IO。

表 10.4 FilterOutputStream 的类型

类 功能 构建器参数 / 如何使用

DataOutputStream 与 DataInputStream 配合使用，以便采用方便的形式将基本数据类型（int，char，long 等）写入一个数据流 OutputStream / 包含了完整接口，以便我们写入基本数据类型

PrintStream 用于产生格式化输出。DataOutputStream 控制的是数据的“存储”，而 PrintStream 控制的是“显示” OutputStream，可选一个布尔参数，指示缓冲区是否与每个新行一同刷新 / 对于自己的 OutputStream 对象，应该用“final”将其封闭在内。可能经常都要用到它

BufferedOutputStream 用它避免每次发出数据的时候都要进行物理性的写入，要求它“请先在缓冲区里找”。可调用 flush()，对缓冲区进行刷新 OutputStream，可选缓冲区大小 / 本身并不能提供一个接口，只是发出使用缓冲区的要求。需要同一个接口对象连接到一起

10.3 本身的缺陷：RandomAccessFile

RandomAccessFile 用于包含了已知长度记录的文件，以便我们能用 seek()从一条记录移至另一条；然后读取或修改那些记录。各记录的长度并不一定相同；只要知道它们有多大以及置于文件何处即可。

首先，我们有点难以相信 RandomAccessFile 不属于 InputStream 或者 OutputStream 分层结构的一部分。除了恰巧实现了 DataInput 以及 DataOutput（这两者亦由 DataInputStream 和 DataOutputStream 实现）接口之外，它们与那些分层结构并无什么关系。它甚至没有用到现有 InputStream 或 OutputStream 类的功能——采用的是一个完全不相干的类。该类属于全新的设计，含有自己的全部（大多数为固有）方法。之所以要这样做，是因为 RandomAccessFile 拥有与其他 IO 类型完全不同的行为，因为我们可在一个文件里向前或向后移动。不管在哪种情况下，它都是独立运作的，作为 Object 的一个“直接继承人”使用。

从根本上说，RandomAccessFile 类似 DataInputStream 和 DataOutputStream

的联合使用。其中，getFilePointer()用于了解当前在文件的什么地方，seek()用于移至文件内的一个新地点，而 length()用于判断文件的最大长度。此外，构建器要求使用另一个自变量（与 C 的 fopen()完全一样），指出自己只是随机读（"r"），还是读写兼施（"rw"）。这里没有提供对“只写文件”的支持。也就是说，假如是从 DataInputStream 继承的，那么 RandomAccessFile 也有可能能很好地工作。

还有更难对付的。很容易想象我们有时要在其他类型的数据流中搜索，比如一个 ByteArrayInputStream，但搜索方法只有 RandomAccessFile 才会提供。而后者只能针对文件才能操作，不能针对数据流操作。此时，BufferedInputStream 确实允许我们标记一个位置（使用 mark()，它的值容纳于单个内部变量中），并用 reset()重设那个位置。但这些做法都存在限制，并不是特别有用。

10.4 File 类

File 类有一个欺骗性的名字——通常会认为它对付的是一个文件，但实情并非如此。它既代表一个特定文件的名字，也代表目录内一系列文件的名字。若代表一个文件集，便可用 list()方法查询这个集，返回的是一个字串数组。之所以要返回一个数组，而非某个灵活的集合类，是因为元素的数量是固定的。而且若想得到一个不同的目录列表，只需创建一个不同的 File 对象即可。事实上，“FilePath”（文件路径）似乎是一个更好的名字。本节将向大家完整地例示如何使用这个类，其中包括相关的 FilenameFilter（文件名过滤器）接口。

10.4.1 目录列表器

现在假设我们想观看一个目录列表。可用两种方式列出 File 对象。若在不含自变量（参数）的情况下调用 list()，会获得 File 对象包含的一个完整列表。然而，若想对这个列表进行某些限制，就需要使用一个“目录过滤器”，该类的作用是指出应如何选择 File 对象来完成显示。

下面是用于这个例子的代码（或在执行该程序时遇到困难，请参考第 3 章 3.1.2 小节“赋值”）：

449-450 页程序

```
//: DirList.java
// Displays directory listing
package c10;
import java.io.*;
public class DirList {
public static void main(String[] args) {
try {
File path = new File(".");
String[] list;
if(args.length == 0)
list = path.list();
else
list = path.list(new DirFilter(args[0]));
for(int i = 0; i < list.length; i++)
```

```
System.out.println(list[i]);
} catch(Exception e) {
e.printStackTrace();
}
}
```

DirFilter 类“实现”了 interface FilenameFilter（关于接口的问题，已在第 7 章进行了详述）。下面让我们看看 FilenameFilter 接口有多么简单：

```
public interface FilenameFilter {
boolean accept(文件目录, 字串名);
}
```

它指出这种类型的所有对象都提供了一个名为 accept()的方法。之所以要创建这样的一个类，背后的全部原因就是把 accept()方法提供给 list()方法，使 list()能够“回调”accept()，从而判断应将哪些文件名包括到列表中。因此，通常将这种技术称为“回调”，有时也称为“算子”（也就是说，DirFilter 是一个算子，因为它唯一的作用就是容纳一个方法）。由于 list()采用一个 FilenameFilter 对象作为自己的自变量使用，所以我们能传递实现了 FilenameFilter 的任何类的一个对象，用它决定（甚至在运行期）list()方法的行为方式。回调的目的是在代码的行为上提供更大的灵活性。

通过 DirFilter，我们看出尽管一个“接口”只包含了一系列方法，但并不局限于只能写那些方法（但是，至少必须提供一个接口内所有方法的定义。在这种情况下，DirFilter 构建器也会创建）。

accept()方法必须接纳一个 File 对象，用它指示用于寻找一个特定文件的目录；并接纳一个 String，其中包含了要寻找之文件的名字。可决定使用或忽略这两个参数之一，但有时至少要使用文件名。记住 list()方法准备为目录对象中的每个文件名调用 accept()，核实哪个应包含在内——具体由 accept()返回的“布尔”结果决定。

为确定我们操作的只是文件名，其中没有包含路径信息，必须采用 String 对象，并在它的外部创建一个 File 对象。然后调用 getName()，它的作用是去除所有路径信息（采用与平台无关的方式）。随后，accept()用 String 类的 indexOf()方法检查文件名内部是否存在搜索字串"afn"。若在字串内找到 afn，那么返回值就是 afn 的起点索引；但假如没有找到，返回值就是-1。注意这只是一个简单的字串搜索例子，未使用常见的表达式“通配符”方案，比如"fo?.b?r*"；这种方案更难实现。

list()方法返回的是一个数组。可查询这个数组的长度，然后在其中遍历，选定数组元素。与 C 和 C++的类似行为相比，这种于方法内外方便游历数组的行为无疑是一个显著的进步。

1. 匿名内部类

下例用一个匿名内部类（已在第 7 章讲述）来重写显得非常理想。首先创建了一个 filter()方法，它返回指向 FilenameFilter 的一个句柄：

451-452 页程序

注意 filter()的自变量必须是 final。这一点是匿名内部类要求的，使其能使用来自本身作用域以外的一个对象。

之所以认为这样做更好，是由于 FilenameFilter 类现在同 DirList2 紧密地结合在一起。然而，我们可采取进一步的操作，将匿名内部类定义成 list()的一个参数，使其显得更加精简。如下所示：

452 页下程序

main()现在的自变量是 final，因为匿名内部类直接使用 args[0]。

这展示了如何利用匿名内部类快速创建精简的类，以便解决一些复杂的问题。由于 Java 中的所有东西都与类有关，所以它无疑是一种相当有用的编码技术。它的一个好处是将特定的问题隔离在一个地方统一解决。但在另一方面，这样生成的代码不是十分容易阅读，所以使用时必须慎重。

2. 顺序目录列表

经常都需要文件名以排好序的方式提供。由于 Java 1.0 和 Java 1.1 都没有提供对排序的支持（从 Java 1.2 开始提供），所以必须用第 8 章创建的 SortVector 将这一能力直接加入自己的程序。就象下面这样：

453-454 页程序

这里进行了另外少许改进。不再是将 path（路径）和 list（列表）创建为 main()的本地变量，它们变成了类的成员，使它们的值能在对象“生存”期间方便地访问。事实上，main()现在只是对类进行测试的一种方式。大家可以看到，一旦列表创建完毕，类的构建器就会自动开始对列表进行排序。

这种排序不要求区分大小写，所以最终不会得到一组全部单词都以大写字母开头的列表，跟着是全部以小写字母开头的列表。然而，我们注意到在以相同字母开头的一组文件名中，大写字母是排在前面的——这对标准的排序来说仍是一种不合格的行为。Java 1.2 已成功解决了这个问题。

10.4.2 检查与创建目录

File 类并不仅仅是对现有目录路径、文件或者文件组的一个表示。亦可用一个 File 对象新建一个目录，甚至创建一个完整的目录路径——假如它尚不存在的话。亦可用它了解文件的属性（长度、上一次修改日期、读 / 写属性等），检查一个 File 对象到底代表一个文件还是一个目录，以及删除一个文件等等。下列程序完整展示了如何运用 File 类剩下的这些方法：

454-456 页程序

在 fileData()中，可看到应用了各种文件调查方法来显示与文件或目录路径有关的信息。

main()应用的第一个方法是 renameTo()，利用它可以重命名（或移动）一个文件至一个全新的路径（该路径由参数决定），它属于另一个 File 对象。这也适

用于任何长度的目录。

若试验上述程序，就可发现自己能制作任意复杂程度的一个目录路径，因为mkdirs()会帮我们完成所有工作。在 Java 1.0 中，-d 标志报告目录虽然已被删除，但它依然存在；但在 Java 1.1 中，目录会被实际删除。

10.5 IO 流的典型应用

尽管库内存在大量 IO 流类，可通过多种不同的方式组合到一起，但实际上只有几种方式才会经常用到。然而，必须小心在意才能得到正确的组合。下面这个相当长的例子展示了典型 IO 配置的创建与使用，可在写自己的代码时将其作为一个参考使用。注意每个配置都以一个注释形式的编号起头，并提供了适当的解释信息。

457-459 页程序

10.5.1 输入流

当然，我们经常想做的一件事情是将格式化的输出打印到控制台，但那已在第 5 章创建的 com.bruceeckel.tools 中得到了简化。

第 1 到第 4 部分演示了输入流的创建与使用（尽管第 4 部分展示了将输出流作为一个测试工具的简单应用）。

1. 缓冲的输入文件

为打开一个文件以便输入，需要使用一个 FileInputStream，同时将一个 String 或 File 对象作为文件名使用。为提高速度，最好先对文件进行缓冲处理，从而获得用于一个 BufferedInputStream 的构建器的结果句柄。为了以格式化的形式读取输入数据，我们将那个结果句柄赋给用于一个 DataInputStream 的构建器。DataInputStream 是我们的最终（final）对象，并是我们进行读取操作的接口。

在这个例子中，只用到了 readLine()方法，但理所当然任何 DataInputStream 方法都可以采用。一旦抵达文件末尾，readLine()就会返回一个 null（空），以便中止并退出 while 循环。

“String s2”用于聚集完整的文件内容（包括必须添加的新行，因为 readLine()去除了那些行）。随后，在本程序的后面部分中使用 s2。最后，我们调用 close()，用它关闭文件。从技术上说，会在运行 finalize()时调用 close()。而且我们希望一旦程序退出，就发生这种情况（无论是否进行垃圾收集）。然而，Java 1.0 有一个非常突出的错误（Bug），造成这种情况不会发生。在 Java 1.1 中，必须明确调用 System.runFinalizersOnExit(true)，用它保证会为系统中的每个对象调用 finalize()。然而，最安全的方法还是为文件明确调用 close()。

2. 从内存输入

这一部分采用已经包含了完整文件内容的 String s2，并用它创建一个 StringBufferInputStream（字串缓冲输入流）——作为构建器的参数，要求使用一个 String，而非一个 StringBuffer）。随后，我们用 read()依次读取每个字符，并将其发送至控制台。注意 read()将下一个字节返回为 int，所以必须将其造型为一个 char，以便正确地打印。

3. 格式化内存输入

StringBufferInputStream 的接口是有限的，所以通常需要将其封装到一个 DataInputStream 内，从而增强它的能力。然而，若选择用 readByte()每次读出一个字符，那么所有值都是有效的，所以不可再用返回值来侦测何时结束输入。相反，可用 available()方法判断有多少字符可用。下面这个例子展示了如何从文件中一次读出一个字符：

461 页程序

注意取决于当前从什么媒体读入，avaiable()的工作方式也是有所区别的。它在字面上意味着“可以不受阻塞读取的字节数量”。对一个文件来说，它意味着整个文件。但对一个不同种类的数据流来说，它却可能有不同的含义。因此在使用时应考虑周全。

为了在这样的情况下侦测输入的结束，也可以通过捕获一个违例来实现。然而，若真的用违例来控制数据流，却显得有些大材小用。

4. 行的编号与文件输出

这个例子展示了如何 LineNumberInputStream 来跟踪输入行的编号。在这里，不可简单地将所有构建器都组合起来，因为必须保持 LineNumberInputStream 的一个句柄（注意这并非一种继承环境，所以不能简单地将 in4 造型到一个 LineNumberInputStream）。因此，li 容纳了指向 LineNumberInputStream 的句柄，然后在它的基础上创建一个 DataInputStream，以便读入数据。

这个例子也展示了如何将格式化数据写入一个文件。首先创建了一个 FileOutputStream，用它同一个文件连接。考虑到效率方面的原因，它生成了一个 BufferedOutputStream。这几乎肯定是我们一般的做法，但却必须明确地这样做。随后为了进行格式化，它转换成一个 PrintStream。用这种方式创建的数据文件可作为一个原始的文本文件读取。

标志 DataInputStream 何时结束的一个方法是 readLine()。一旦没有更多的字串可以读取，它就会返回 null。每个行都会伴随自己的行号打印到文件里。该行号可通过 li 查询。

可看到用于 out1 的、一个明确指定的 close()。若程序准备掉转头来，并再次读取相同的文件，这种做法就显得相当有用。然而，该程序直到结束也没有检查文件 IODemo.txt。正如以前指出的那样，如果不为自己的所有输出文件调用 close()，就可能发现缓冲区不会得到刷新，造成它们不完整。。

10.5.2 输出流

两类主要的输出流是按它们写入数据的方式划分的：一种按人的习惯写入，另一种为了以后由一个 DataInputStream 而写入。RandomAccessFile 是独立的，尽管它的数据格式兼容于 DataInputStream 和 DataOutputStream。

5. 保存与恢复数据

PrintStream 能格式化数据，使其能按我们的习惯阅读。但为了输出数据，以便由另一个数据流恢复，则需用一个 DataOutputStream 写入数据，并用一个 DataInputStream 恢复（获取）数据。当然，这些数据流可以是任何东西，但这里

采用的是一个文件，并进行了缓冲处理，以加快读写速度。

注意字串是用 writeBytes()写入的，而非 writeChars()。若使用后者，写入的就是 16 位 Unicode 字符。由于 DataInputStream 中没有补充的“readChars”方法，所以不得不用 readChar()每次取出一个字符。所以对 ASCII 来说，更方便的做法是将字符作为字节写入，在后面跟随一个新行；然后再用 readLine()将字符当作普通的 ASCII 行读回。

writeDouble()将 double 数字保存到数据流中，并用补充的 readDouble()恢复它。但为了保证任何读方法能够正常工作，必须知道数据项在流中的准确位置，因为既有可能将保存的 double 数据作为一个简单的字节序列读入，也有可能作为 char 或其他格式读入。所以必须要么为文件中的数据采用固定的格式，要么将额外的信息保存到文件中，以便正确判断数据的存放位置。

6. 读写随机访问文件

正如早先指出的那样，RandomAccessFile 与 IO 层次结构的剩余部分几乎是完全隔离的，尽管它也实现了 DataInput 和 DataOutput 接口。所以不可将其与 InputStream 及 OutputStream 子类的任何部分关联起来。尽管也许能将一个 ByteArrayInputStream 当作一个随机访问元素对待，但只能用 RandomAccessFile 打开一个文件。必须假定 RandomAccessFile 已得到了正确的缓冲，因为我们不能自行选择。

可以自行选择的是第二个构建器参数：可决定以“只读”（r）方式或“读写”（rw）方式打开一个 RandomAccessFile 文件。

使用 RandomAccessFile 的时候，类似于组合使用 DataInputStream 和 DataOutputStream（因为它实现了等同的接口）。除此以外，还可看到程序中使用了 seek()，以便在文件中到处移动，对某个值作出修改。

10.5.3 快捷文件处理

由于以前采用的一些典型形式都涉及到文件处理，所以大家也许会怀疑为什么要进行那么多的代码输入——这正是装饰器方案一个缺点。本部分将向大家展示如何创建和使用典型文件读取和写入配置的快捷版本。这些快捷版本均置入 packagecom.bruceeckel.tools 中（自第 5 章开始创建）。为了将每个类都添加到库内，只需将其置入适当的目录，并添加对应的 package 语句即可。

7. 快速文件输入

若想创建一个对象，用它从一个缓冲的 DataInputStream 中读取一个文件，可将这个过程封装到一个名为 InFile 的类内。如下所示：

463-464 页程序

无论构建器的 String 版本还是 File 版本都包括在内，用于共同创建一个 FileInputStream。

就象这个例子展示的那样，现在可以有效减少创建文件时由于重复强调造成的问题。

8. 快速输出格式化文件

亦可用同类型的方法创建一个 PrintStream，令其写入一个缓冲文件。下面是对 com.bruceeckel.tools 的扩展：

464-465 页程序

注意构建器不可能捕获一个由基础类构建器“掷”出的违例。

9. 快速输出数据文件

最后，利用类似的快捷方式可创建一个缓冲输出文件，用它保存数据（与由人观看的数据格式相反）：

465 页程序

非常奇怪的是（也非常不幸），Java 库的设计者居然没想到将这些便利措施直接作为他们的一部分标准提供。

10.5.4 从标准输入中读取数据

以 Unix 首先倡导的“标准输入”、“标准输出”以及“标准错误输出”概念为基础，Java 提供了相应的 System.in，System.out 以及 System.err。贯这一整本书，大家都会接触到如何用 System.out 进行标准输出，它已预封装成一个 PrintStream 对象。System.err 同样是一个 PrintStream，但 System.in 是一个原始的 InputStream，未进行任何封装处理。这意味着尽管能直接使用 System.out 和 System.err，但必须事先封装 System.in，否则不能从中读取数据。

典型情况下，我们希望用 readLine()每次读取一行输入信息，所以需要将 System.in 封装到一个 DataInputStream 中。这是 Java 1.0 进行行输入时采取的“老”办法。在本章稍后，大家还会看到 Java 1.1 的解决方案。下面是个简单的例子，作用是回应我们键入的每一行内容：

466 页程序

之所以要使用 try 块，是由于 readLine()可能“掷”出一个 IOException。注意同其他大多数流一样，也应对 System.in 进行缓冲。

由于在每个程序中都要将 System.in 封装到一个 DataInputStream 内，所以显得有点不方便。但采用这种设计方案，可以获得最大的灵活性。

10.5.5 管道数据流

本章已简要介绍了 PipedInputStream（管道输入流）和 PipedOutputStream（管道输出流）。尽管描述不十分详细，但并不是说它们作用不大。然而，只有在掌握了多线程处理的概念后，才可真正体会它们的价值所在。原因很简单，因为管道化的数据流就是用于线程之间的通信。这方面的问题将在第 14 章用一个示例说明。

10.6 StreamTokenizer

尽管 StreamTokenizer 并不是从 InputStream 或 OutputStream 衍生的，但它只

随同 InputStream 工作，所以十分恰当地包括在库的 IO 部分中。

StreamTokenizer 类用于将任何 InputStream 分割为一系列“记号”（Token）。这些记号实际是一些断续的文本块，中间用我们选择的任何东西分隔。例如，我们的记号可以是单词，中间用空白（空格）以及标点符号分隔。

下面是一个简单的程序，用于计算各个单词在文本文件中重复出现的次数：

467-469 页程序

最好将结果按排序格式输出，但由于 Java 1.0 和 Java 1.1 都没有提供任何排序方法，所以必须由自己动手。这个目标可用一个 StrSortVector 方便地达成（创建于第 8 章，属于那一章创建的软件包的一部分。记住本书所有子目录的起始目录都必须位于类路径中，否则程序将不能正确地编译）。

为打开文件，使用了一个 FileInputStream。而且为了将文件转换成单词，从 FileInputStream 中创建了一个 StreamTokenizer。在 StreamTokenizer 中，存在一个默认的分隔符列表，我们可用一系列方法加入更多的分隔符。在这里，我们用 ordinaryChar()指出“该字符没有特别重要的意义”，所以解析器不会把它当作自己创建的任何单词的一部分。例如，st.ordinaryChar('.')表示小数点不会成为解析出来的单词的一部分。在与 Java 配套提供的联机文档中，可以找到更多的相关信息。

在 countWords()中，每次从数据流中取出一个记号，而 ttype 信息的作用是判断对每个记号采取什么操作——因为记号可能代表一个行尾、一个数字、一个字串或者一个字符。

找到一个记号后，会查询 Hashtable counts，核实其中是否已经以“键”（Key）的形式包含了一个记号。若答案是肯定的，对应的 Counter（计数器）对象就会增值，指出已找到该单词的另一个实例。若答案为否，则新建一个 Counter——因为 Counter 构建器会将它的值初始化为 1，正是我们计算单词数量时的要求。

SortedWordCount 并不属于 Hashtable（散列表）的一种类型，所以它不会继承。它执行的一种特定类型的操作，所以尽管 keys()和 values()方法都必须重新揭示出来，但仍不表示应使用那个继承，因为大量 Hashtable 方法在这里都是不适当的。除此以外，对于另一些方法来说（比如 getCounter()——用于获得一个特定字串的计数器；又如 sortedKeys()——用于产生一个枚举），它们最终都改变了 SortedWordCount 接口的形式。

在 main()内，我们用 SortedWordCount 打开和计算文件中的单词数量——总共只用了两行代码。随后，我们为一个排好序的键（单词）列表提取出一个枚举。并用它获得每个键以及相关的 Count（计数）。注意必须调用 cleanup()，否则文件不能正常关闭。

采用了 StreamTokenizer 的第二个例子将在第 17 章提供。

10.6.1 StringTokenizer

尽管并不必要 IO 库的一部分，但 StringTokenizer 提供了与 StreamTokenizer 极相似的功能，所以在这里一并讲述。

StringTokenizer 的作用是每次返回字串内的一个记号。这些记号是一些由制表站、空格以及新行分隔的连续字符。因此，字串“Where is my cat?”的记号分别是“Where”、“is”、“my”和“cat?”。与 StreamTokenizer 类似，我们可以指示

StringTokenizer 按照我们的愿望分割输入。但对于 StringTokenizer，却需要向构建器传递另一个参数，即我们想使用的分隔字串。通常，如果想进行更复杂的操作，应使用 StreamTokenizer。

可用 nextToken()向 StringTokenizer 对象请求字串内的下一个记号。该方法要么返回一个记号，要么返回一个空字串（表示没有记号剩下）。

作为一个例子，下述程序将执行一个有限的句法分析，查询键短语序列，了解句子暗示的是快乐亦或悲伤的含义。

471-472 页程序

对于准备分析的每个字串，我们进入一个 while 循环，并将记号从那个字串中取出。请注意第一个 if 语句，假如记号既不是“I”，也不是“Are”，就会执行 continue（返回循环起点，再一次开始）。这意味着除非发现一个“I”或者“Are”，才会真正得到记号。大家可能想用==代替 equals()方法，但那样做会出现不正常的表现，因为==比较的是句柄值，而 equals()比较的是内容。

analyze()方法剩余部分的逻辑是搜索“I am sad”（我很忧伤、“I am nothappy”（我不快乐）或者“Are you sad?”（你悲伤吗？）这样的句法格式。若没有 break 语句，这方面的代码甚至可能更加散乱。大家应注意对一个典型的解析器来说，通常都有这些记号的一个表格，并能在读取新记号的时候用一小段代码在表格内移动。

无论如何，只应将 StringTokenizer 看作 StreamTokenizer 一种简单而且特殊的简化形式。然而，如果有一个字串需要进行记号处理，而且 StringTokenizer 的功能实在有限，那么应该做的全部事情就是用 StringBufferInputStream 将其转换到一个数据流里，再用它创建一个功能更强大的 StreamTokenizer。

10.7 Java 1.1 的 IO 流

到这个时候，大家或许会陷入一种困境之中，怀疑是否存在 IO 流的另一种设计方案，并可能要求更大的代码量。还有人能提出一种更古怪的设计吗？事实上，Java 1.1 对 IO 流库进行了一些重大的改进。看到 Reader 和 Writer 类时，大多数人的第一个印象（就象我一样）就是它们用来替换原来的 InputStream 和 OutputStream 类。但实情并非如此。尽管不建议使用原始数据流库的某些功能（如使用它们，会从编译器收到一条警告消息），但原来的数据流依然得到了保留，以便维持向后兼容，而且：

(1) 在老式层次结构里加入了新类，所以 Sun 公司明显不会放弃老式数据流。

(2) 在许多情况下，我们需要与新结构中的类联合使用老结构中的类。为达到这个目的，需要使用一些“桥”类：InputStreamReader 将一个 InputStream 转换成 Reader，OutputStreamWriter 将一个 OutputStream 转换成 Writer。

所以与原来的 IO 流库相比，经常都要对新 IO 流进行层次更多的封装。同样地，这也属于装饰器方案的一个缺点——需要为额外的灵活性付出代价。

之所以在 Java 1.1 里添加了 Reader 和 Writer 层次，最重要的原因便是国际化的需求。老式 IO 流层次结构只支持 8 位字节流，不能很好地控制 16 位 Unicode 字符。由于 Unicode 主要面向的是国际化支持（Java 内含的 char 是 16 位的 Unicode），所以添加了 Reader 和 Writer 层次，以提供对所有 IO 操作中的 Unicode 的支持。除此之外，新库也对速度进行了优化，可比旧库更快地运行。

与本书其他地方一样，我会试着提供对类的一个概述，但假定你会利用联机文档搞定所有的细节，比如方法的详尽列表等。

10.7.1 数据的发起与接收

Java 1.0 的几乎所有 IO 流类都有对应的 Java 1.1 类，用于提供内建的 Unicode 管理。似乎最容易的事情就是“全部使用新类，再也不要用旧的”，但实际情况并没有这么简单。有些时候，由于受到库设计的一些限制，我们不得不使用 Java 1.0 的 IO 流类。特别要指出的是，在旧流库的基础上新加了 java.util.zip 库，它们依赖旧的流组件。所以最明智的做法是“尝试性”地使用 Reader 和 Writer 类。若代码不能通过编译，便知道必须换回老式库。

下面这张表格分旧库与新库分别总结了信息发起与接收之间的对应关系。

发起＆接收：Java 1.0 类 对应的 Java 1.1 类

474 页表中内容略

我们发现即使不完全一致，但旧库组件中的接口与新接口通常也是类似的。

10.7.2 修改数据流的行为

在 Java 1.0 中，数据流通过 FilterInputStream 和 FilterOutputStream 的“装饰器”（Decorator）子类适应特定的需求。Java 1.1 的 IO 流沿用了这一思想，但没有继续采用所有装饰器都从相同“filter”（过滤器）基础类中衍生这一做法。若通过观察类的层次结构来理解它，这可能令人出现少许的困惑。

在下面这张表格中，对应关系比上一张表要粗糙一些。之所以会出现这个差别，是由类的组织造成的：尽管 BufferedOutputStream 是 FilterOutputStream 的一个子类，但是 BufferedWriter 并不是 FilterWriter 的子类（对后者来说，尽管它是一个抽象类，但没有自己的子类或者近似子类的东西，也没有一个“占位符”可用，所以不必费心地寻找）。然而，两个类的接口是非常相似的，而且不管在什么情况下，显然应该尽可能地使用新版本，而不应考虑旧版本（也就是说，除非在一些类中必须生成一个 Stream，不可生成 Reader 或者 Writer）。

过滤器：Java 1.0 类 对应的 Java 1.1 类

FilterInputStream FilterReader
FilterOutputStream FilterWriter（没有子类的抽象类）
BufferedInputStream BufferedReader（也有 readLine()）
BufferedOutputStream BufferedWriter
DataInputStream 使用 DataInputStream（除非要使用 readLine()，那时需要使用一个 BufferedReader）
PrintStream PrintWriter
LineNumberInputStream LineNumberReader
StreamTokenizer StreamTokenizer（用构建器取代 Reader）
PushBackInputStream PushBackReader

有一条规律是显然的：若想使用 readLine()，就不要再用一个 DataInputStream 来实现（否则会在编译期得到一条出错消息），而应使用一个 BufferedReader。但除这种情况以外，DataInputStream 仍是 Java 1.1 IO 库的“首选”成员。

为了将向 PrintWriter 的过渡变得更加自然，它提供了能采用任何 OutputStream 对象的构建器。PrintWriter 提供的格式化支持没有 PrintStream 那么多；但接口几乎是相同的。

10.7.3 未改变的类

显然，Java 库的设计人员觉得以前的一些类毫无问题，所以没有对它们作任何修改，可象以前那样继续使用它们：

没有对应 Java 1.1 类的 Java 1.0 类

DataOutputStream
File
RandomAccessFile
SequenceInputStream

特别未加改动的是 DataOutputStream，所以为了用一种可转移的格式保存和获取数据，必须沿用 InputStream 和 OutputStream 层次结构。

10.7.4 一个例子

为体验新类的效果，下面让我们看看如何修改 IOStreamDemo.java 示例的相应区域，以便使用 Reader 和 Writer 类：

476-478 页程序

大家一般看见的是转换过程非常直观，代码看起来也颇相似。但这些都不是重要的区别。最重要的是，由于随机访问文件已经改变，所以第 6 节未再重复。

第 1 节收缩了一点儿，因为假如要做的全部事情就是读取行输入，那么只需要将一个 FileReader 封装到 BufferedReader 之内即可。第 1b 节展示了封装 System.in，以便读取控制台输入的新方法。这里的代码量增多了一些，因为 System.in 是一个 DataInputStream，而且 BufferedReader 需要一个 Reader 参数，所以要用 InputStreamReader 来进行转换。

在 2 节，可以看到如果有一个字串，而且想从中读取数据，只需用一个 StringReader 替换 StringBufferInputStream，剩下的代码是完全相同的。

第 3 节揭示了新 IO 流库设计中的一个错误。如果有一个字串，而且想从中读取数据，那么不能再以任何形式使用 StringBufferInputStream。若编译一个涉及 StringBufferInputStream 的代码，会得到一条“反对”消息，告诉我们不要用它。此时最好换用一个 StringReader。但是，假如要象第 3 节这样进行格式化的内存输入，就必须使用 DataInputStream——没有什么“DataReader”可以代替它——而 DataInputStream 很不幸地要求用到一个 InputStream 参数。所以我们没有选择的余地，只好使用编译器不赞成的 StringBufferInputStream 类。编译器同样会发出反对信息，但我们对此束手无策（注释②）。

StringReader 替换 StringBufferInputStream，剩下的代码是完全相同的。

②：到你现在正式使用的时候，这个错误可能已经修正。

第 4 节明显是从老式数据流到新数据流的一个直接转换，没有需要特别指出的。在第 5 节中，我们被强迫使用所有的老式数据流，因为 DataOutputStream 和 DataInputStream 要求用到它们，而且没有可供替换的东西。然而，编译期间不会产生任何“反对”信息。若不赞成一种数据流，通常是由于它的构建器产生了一条反对消息，禁止我们使用整个类。但在 DataInputStream 的情况下，只有 readLine()是不赞成使用的，因为我们最好为 readLine()使用一个 BufferedReader（但为其他所有格式化输入都使用一个 DataInputStream）。

若比较第 5 节和 IOStreamDemo.java 中的那一小节，会注意到在这个版本中，数据是在文本之前写入的。那是由于 Java 1.1 本身存在一个错误，如下述代码所示：

479-480 页程序

看起来，我们在对一个 writeBytes()的调用之后写入的任何东西都不是能够恢复的。这是一个十分有限的错误，希望在你读到本书的时候已获得改正。为检测是否改正，请运行上述程序。若没有得到一个违例，而且值都能正确打印出来，就表明已经改正。

10.7.5 重导向标准 IO

Java 1.1 在 System 类中添加了特殊的方法，允许我们重新定向标准输入、输出以及错误 IO 流。此时要用到下述简单的静态方法调用：

setIn(InputStream)

setOut(PrintStream)

setErr(PrintStream)

如果突然要在屏幕上生成大量输出，而且滚动的速度快于人们的阅读速度，输出的重定向就显得特别有用。在一个命令行程序中，如果想重复测试一个特定的用户输入序列，输入的重定向也显得特别有价值。下面这个简单的例子展示了这些方法的使用：

481 页程序

这个程序的作用是将标准输入同一个文件连接起来，并将标准输出和错误重定向至另一个文件。

这是不可避免会遇到“反对”消息的另一个例子。用-deprecation 标志编译时得到的消息如下：

Note:The constructor java.io.PrintStream(java.io.OutputStream) has been deprecated.

注意：不推荐使用构建器 java.io.PrintStream（java.io.OutputStream）。

然而，无论 System.setOut()还是 System.setErr()都要求用一个 PrintStream 作为参数使用，所以必须调用 PrintStream 构建器。所以大家可能会觉得奇怪，既然 Java 1.1 通过反对构建器而反对了整个 PrintStream，为什么库的设计人员在添加这个反对的同时，依然为 System 添加了新方法，且指明要求用 PrintStream，而不是用 PrintWriter 呢？毕竟，后者是一个崭新和首选的替换措施呀？这真令人费解。

10.8 压缩

Java 1.1 也添加一个类，用以支持对压缩格式的数据流的读写。它们封装到现成的 IO 类中，以提供压缩功能。

此时 Java 1.1 的一个问题显得非常突出：它们不是从新的 Reader 和 Writer 类衍生出来的，而是属于 InputStream 和 OutputStream 层次结构的一部分。所以有时不得不混合使用两种类型的数据流（注意可用 InputStreamReader 和 OutputStreamWriter 在不同的类型间方便地进行转换）。

Java 1.1 压缩类 功能

CheckedInputStream GetCheckSum()为任何 InputStream 产生校验和（不仅是解压）

CheckedOutputStream GetCheckSum()为任何 OutputStream 产生校验和（不仅是解压）

DeflaterOutputStream 用于压缩类的基础类

ZipOutputStream 一个 DeflaterOutputStream，将数据压缩成 Zip 文件格式

GZIPOutputStream 一个 DeflaterOutputStream，将数据压缩成 GZIP 文件格式

InflaterInputStream 用于解压类的基础类

ZipInputStream 一个 DeflaterInputStream，解压用 Zip 文件格式保存的数据

GZIPInputStream 一个 DeflaterInputStream，解压用 GZIP 文件格式保存的数据

尽管存在许多种压缩算法，但是 Zip 和 GZIP 可能最常用的。所以能够很方便地用多种现成的工具来读写这些格式的压缩数据。

10.8.1 用 GZIP 进行简单压缩

GZIP 接口非常简单，所以如果只有单个数据流需要压缩（而不是一系列不同的数据），那么它就可能是最适当选择。下面是对单个文件进行压缩的例子：

483-484 页程序

压缩类的用法非常直观——只需将输出流封装到一个 GZIPOutputStream 或者 ZipOutputStream 内，并将输入流封装到 GZIPInputStream 或者 ZipInputStream 内即可。剩余的全部操作就是标准的 IO 读写。然而，这是一个很典型的例子，我们不得不混合使用新旧 IO 流：数据的输入使用 Reader 类，而 GZIPOutputStream 的构建器只能接收一个 OutputStream 对象，不能接收 Writer 对象。

10.8.2 用 Zip 进行多文件保存

提供了 Zip 支持的 Java 1.1 库显得更加全面。利用它可以方便地保存多个文件。甚至有一个独立的类来简化对 Zip 文件的读操作。这个库采采用的是标准 Zip 格式，所以能与当前因特网上使用的大量压缩、解压工具很好地协作。下面这个例子采取了与前例相同的形式，但能根据我们需要控制任意数量的命令行参数。除此之外，它展示了如何用 Checksum 类来计算和校验文件的“校验和”（Checksum）。可选用两种类型的 Checksum：Adler32（速度要快一些）和 CRC32（慢一些，但更准确）。

484－486 页程序

对于要加入压缩档的每一个文件，都必须调用 putNextEntry()，并将其传递给一个 ZipEntry 对象。ZipEntry 对象包含了一个功能全面的接口，利用它可以获取和设置 Zip 文件内那个特定的 Entry（入口）上能够接受的所有数据：名字、压缩后和压缩前的长度、日期、CRC 校验和、额外字段的数据、注释、压缩方法以及它是否一个目录入口等等。然而，虽然 Zip 格式提供了设置密码的方法，但 Java 的 Zip 库没有提供这方面的支持。而且尽管 CheckedInputStream 和 CheckedOutputStream 同时提供了对 Adler32 和 CRC32 校验和的支持，但是 ZipEntry 只支持 CRC 的接口。这虽然属于基层 Zip 格式的限制，但却限制了我们使用速度更快的 Adler32。

为解压文件，ZipInputStream 提供了一个 getNextEntry()方法，能在有的前提下返回下一个 ZipEntry。作为一个更简洁的方法，可以用 ZipFile 对象读取文件。该对象有一个 entries()方法，可以为 ZipEntry 返回一个 Enumeration（枚举）。

为读取校验和，必须多少拥有对关联的 Checksum 对象的访问权限。在这里保留了指向 CheckedOutputStream 和 CheckedInputStream 对象的一个句柄。但是，也可以只占有指向 Checksum 对象的一个句柄。

Zip 流中一个令人困惑的方法是 setComment()。正如前面展示的那样，我们可在写一个文件时设置注释内容，但却没有办法取出 ZipInputStream 内的注释。看起来，似乎只能通过 ZipEntry 逐个入口地提供对注释的完全支持。

当然，使用 GZIP 或 Zip 库时并不仅仅限于文件——可以压缩任何东西，包括要通过网络连接发送的数据。

10.8.3 Java 归档（jar）实用程序

Zip 格式亦在 Java 1.1 的 JAR（Java ARchive）文件格式中得到了采用。这种文件格式的作用是将一系列文件合并到单个压缩文件里，就象 Zip 那样。然而，同 Java 中其他任何东西一样，JAR 文件是跨平台的，所以不必关心涉及具体平台的问题。除了可以包括声音和图像文件以外，也可以在其中包括类文件。

涉及因特网应用时，JAR 文件显得特别有用。在 JAR 文件之前，Web 浏览器必须重复多次请求 Web 服务器，以便下载完构成一个“程序片”（Applet）的所有文件。除此以外，每个文件都是未经压缩的。但在将所有这些文件合并到一个 JAR 文件里以后，只需向远程服务器发出一次请求即可。同时，由于采用了压缩技术，所以可在更短的时间里获得全部数据。另外，JAR 文件里的每个入口（条目）都可以加上数字化签名（详情参考 Java 用户文档）。

一个 JAR 文件由一系列采用 Zip 压缩格式的文件构成，同时还有一张“详情

单”，对所有这些文件进行了描述（可创建自己的详情单文件；否则，jar 程序会为我们代劳）。在联机用户文档中，可以找到与 JAR 详情单更多的资料（详情单的英语是“Manifest”）。

jar 实用程序已与 Sun 的 JDK 配套提供，可以按我们的选择自动压缩文件。请在命令行调用它：

jar [选项] 说明 [详情单] 输入文件

其中，“选项”用一系列字母表示（不必输入连字号或其他任何指示符）。如下所示：

487 页表

c 创建新的或空的压缩档
t 列出目录表
x 解压所有文件
x file 解压指定文件
f 指出“我准备向你提供文件名”。若省略此参数，jar 会假定它的输入来自标准输入；或者在它创建文件时，输出会进入标准输出内
m 指出第一个参数将是用户自建的详情表文件的名字
v 产生详细输出，对 jar 做的工作进行巨细无遗的描述
O 只保存文件；不压缩文件（用于创建一个 JAR 文件，以便我们将其置入自己的类路径中）
M 不自动生成详情表文件

在准备进入 JAR 文件的文件中，若包括了一个子目录，那个子目录会自动添加，其中包括它自己的所有子目录，以此类推。路径信息也会得到保留。

下面是调用 jar 的一些典型方法：

jar cf myJarFile.jar *.class
用于创建一个名为 myJarFile.jar 的 JAR 文件，其中包含了当前目录中的所有类文件，同时还有自动产生的详情表文件。

jar cmf myJarFile.jar myManifestFile.mf *.class
与前例类似，但添加了一个名为 myManifestFile.mf 的用户自建详情表文件。

jar tf myJarFile.jar
生成 myJarFile.jar 内所有文件的一个目录表。

jar tvf myJarFile.jar
添加“verbose”（详尽）标志，提供与 myJarFile.jar 中的文件有关的、更详细的资料。

jar cvf myApp.jar audio classes image

假定 audio，classes 和 image 是子目录，这样便将所有子目录合并到文件 myApp.jar 中。其中也包括了“verbose”标志，可在 jar 程序工作时反馈更详尽的信息。

如果用 O 选项创建了一个 JAR 文件，那个文件就可置入自己的类路径（CLASSPATH）中：

CLASSPATH="lib1.jar;lib2.jar;"

Java 能在 lib1.jar 和 lib2.jar 中搜索目标类文件。

jar 工具的功能没有 zip 工具那么丰富。例如，不能够添加或更新一个现成 JAR 文件中的文件，只能从头开始新建一个 JAR 文件。此外，不能将文件移入一个 JAR 文件，并在移动后将它们删除。然而，在一种平台上创建的 JAR 文件可在其他任何平台上由 jar 工具毫无阻碍地读出（这个问题有时会困扰 zip 工具）。

正如大家在第 13 章会看到的那样，我们也用 JAR 为 Java Beans 打包。

10.9 对象序列化

Java 1.1 增添了一种有趣的特性，名为“对象序列化”（Object Serialization）。它面向那些实现了 Serializable 接口的对象，可将它们转换成一系列字节，并可在以后完全恢复回原来的样子。这一过程亦可通过网络进行。这意味着序列化机制能自动补偿操作系统间的差异。换句话说，可以先在 Windows 机器上创建一个对象，对其序列化，然后通过网络发给一台 Unix 机器，然后在那里准确无误地重新“装配”。不必关心数据在不同机器上如何表示，也不必关心字节的顺序或者其他任何细节。

就其本身来说，对象的序列化是非常有趣的，因为利用它可以实现“有限持久化”。请记住“持久化”意味着对象的“生存时间”并不取决于程序是否正在执行——它存在或“生存”于程序的每一次调用之间。通过序列化一个对象，将其写入磁盘，以后在程序重新调用时重新恢复那个对象，就能圆满实现一种“持久”效果。之所以称其为“有限”，是因为不能用某种“persistent”（持久）关键字简单地地定义一个对象，并让系统自动照看其他所有细节问题（尽管将来可能成为现实）。相反，必须在自己的程序中明确地序列化和组装对象。

语言里增加了对象序列化的概念后，可提供对两种主要特性的支持。Java 1.1 的“远程方法调用”（RMI）使本来存在于其他机器的对象可以表现出好象就在本地机器上的行为。将消息发给远程对象时，需要通过对象序列化来传输参数和返回值。RMI 将在第 15 章作具体讨论。

对象的序列化也是 Java Beans 必需的，后者由 Java 1.1 引入。使用一个 Bean 时，它的状态信息通常在设计期间配置好。程序启动以后，这种状态信息必须保存下来，以便程序启动以后恢复；具体工作由对象序列化完成。

对象的序列化处理非常简单，只需对象实现了 Serializable 接口即可（该接口仅是一个标记，没有方法）。在 Java 1.1 中，许多标准库类都发生了改变，以便能够序列化——其中包括用于基本数据类型的全部封装器、所有集合类以及其他许多东西。甚至 Class 对象也可以序列化（第 11 章讲述了具体实现过程）。

为序列化一个对象，首先要创建某些 OutputStream 对象，然后将其封装到 ObjectOutputStream 对象内。此时，只需调用 writeObject()即可完成对象的序列化，并将其发送给 OutputStream。相反的过程是将一个 InputStream 封装到

ObjectInputStream 内，然后调用 readObject()。和往常一样，我们最后获得的是指向一个上溯造型 Object 的句柄，所以必须下溯造型，以便能够直接设置。

对象序列化特别“聪明”的一个地方是它不仅保存了对象的“全景图”，而且能追踪对象内包含的所有句柄并保存那些对象；接着又能对每个对象内包含的句柄进行追踪；以此类推。我们有时将这种情况称为“对象网”，单个对象可与之建立连接。而且它还包含了对象的句柄数组以及成员对象。若必须自行操纵一套对象序列化机制，那么在代码里追踪所有这些链接时可能会显得非常麻烦。在另一方面，由于 Java 对象的序列化似乎找不出什么缺点，所以请尽量不要自己动手，让它用优化的算法自动维护整个对象网。下面这个例子对序列化机制进行了测试。它建立了许多链接对象的一个“Worm”（蠕虫），每个对象都与 Worm 中的下一段链接，同时又与属于不同类（Data）的对象句柄数组链接：

490-492 页程序

更有趣的是，Worm 内的 Data 对象数组是用随机数字初始化的（这样便不用怀疑编译器保留了某种原始信息）。每个 Worm 段都用一个 Char 标记。这个 Char 是在重复生成链接的 Worm 列表时自动产生的。创建一个 Worm 时，需告诉构建器希望它有多长。为产生下一个句柄（next），它总是用减去 1 的长度来调用 Worm 构建器。最后一个 next 句柄则保持为 null（空），表示已抵达 Worm 的尾部。

上面的所有操作都是为了加深事情的复杂程度，加大对象序列化的难度。然而，真正的序列化过程却是非常简单的。一旦从另外某个流里创建了 ObjectOutputStream，writeObject()就会序列化对象。注意也可以为一个 String 调用 writeObject()。亦可使用与 DataOutputStream 相同的方法写入所有基本数据类型（它们有相同的接口）。

有两个单独的 try 块看起来是类似的。第一个读写的是文件，而另一个读写的是一个 ByteArray（字节数组）。可利用对任何 DataInputStream 或者 DataOutputStream 的序列化来读写特定的对象；正如在关于连网的那一章会讲到的那样，这些对象甚至包括网络。一次循环后的输出结果如下：

492-493 页程序

可以看出，装配回原状的对象确实包含了原来那个对象里包含的所有链接。

注意在对一个 Serializable（可序列化）对象进行重新装配的过程中，不会调用任何构建器（甚至默认构建器）。整个对象都是通过从 InputStream 中取得数据恢复的。

作为 Java 1.1 特性的一种，我们注意到对象的序列化并不属于新的 Reader 和 Writer 层次结构的一部分，而是沿用老式的 InputStream 和 OutputStream 结构。所以在一些特殊的场合下，不得不混合使用两种类型的层次结构。

10.9.1 寻找类

读者或许会奇怪为什么需要一个对象从它的序列化状态中恢复。举个例子来说，假定我们序列化一个对象，并通过网络将其作为文件传送给另一台机器。此时，位于另一台机器的程序可以只用文件目录来重新构造这个对象吗？

回答这个问题的最好方法就是做一个实验。下面这个文件位于本章的子目录

下：

493 页中程序

用于创建和序列化一个 Alien 对象的文件位于相同的目录下：

493-494 页程序

该程序并不是捕获和控制违例，而是将违例简单、直接地传递到 main()外部，这样便能在命令行报告它们。

程序编译并运行后，将结果产生的 file.x 复制到名为 xfiles 的子目录，代码如下：

494 页中程序

该程序能打开文件，并成功读取 mystery 对象中的内容。然而，一旦尝试查找与对象有关的任何资料——这要求 Alien 的 Class 对象——Java 虚拟机（JVM）便找不到 Alien.class（除非它正好在类路径内，而本例理应相反）。这样就会得到一个名叫 ClassNotFoundException 的违例（同样地，若非能够校验 Alien 存在的证据，否则它等于消失）。

恢复了一个序列化的对象后，如果想对其做更多的事情，必须保证 JVM 能在本地类路径或者因特网的其他什么地方找到相关的.class 文件。

10.9.2 序列化的控制

正如大家看到的那样，默认的序列化机制并不难操纵。然而，假若有特殊要求又该怎么办呢？我们可能有特殊的安全问题，不希望对象的某一部分序列化；或者某一个子对象完全不必序列化，因为对象恢复以后，那一部分需要重新创建。

此时，通过实现 Externalizable 接口，用它代替 Serializable 接口，便可控制序列化的具体过程。这个 Externalizable 接口扩展了 Serializable，并增添了两个方法：writeExternal()和 readExternal()。在序列化和重新装配的过程中，会自动调用这两个方法，以便我们执行一些特殊操作。

下面这个例子展示了 Externalizable 接口方法的简单应用。注意 Blip1 和 Blip2 几乎完全一致，除了极微小的差别（自己研究一下代码，看看是否能发现）：

495-496 页程序

该程序输出如下：

496-497 页程序

未恢复 Blip2 对象的原因是那样做会导致一个违例。你找出了 Blip1 和 Blip2 之间的区别吗？Blip1 的构建器是“公共的”（public），Blip2 的构建器则不然，这样便会在恢复时造成违例。试试将 Blip2 的构建器属性变成“public”，然后删除//!注释标记，看看是否能得到正确的结果。

恢复 b1 后，会调用 Blip1 默认构建器。这与恢复一个 Serializable（可序列化）对象不同。在后者的情况下，对象完全以它保存下来的二进制位为基础恢复，不存在构建器调用。而对一个 Externalizable 对象，所有普通的默认构建行为都会发生（包括在字段定义时的初始化），而且会调用 readExternal()。必须注意这一事实——特别注意所有默认的构建行为都会进行——否则很难在自己的 Externalizable 对象中产生正确的行为。

下面这个例子揭示了保存和恢复一个 Externalizable 对象必须做的全部事情：

497-498 页程序

其中，字段 s 和 i 只在第二个构建器中初始化，不关默认构建器的事。这意味着假如不在 readExternal 中初始化 s 和 i，它们就会成为 null（因为在对象创建的第一步中已将对象的存储空间清除为 1）。若注释掉跟随于“You must do this”后面的两行代码，并运行程序，就会发现当对象恢复以后，s 是 null，而 i 是零。

若从一个 Externalizable 对象继承，通常需要调用 writeExternal() 和 readExternal()的基础类版本，以便正确地保存和恢复基础类组件。

所以为了让一切正常运作起来，千万不可仅在 writeExternal()方法执行期间写入对象的重要数据（没有默认的行为可用来为一个 Externalizable 对象写入所有成员对象）的，而是必须在 readExternal()方法中也恢复那些数据。初次操作时可能会有些不习惯，因为 Externalizable 对象的默认构建行为使其看起来似乎正在进行某种存储与恢复操作。但实情并非如此。

1. transient（临时）关键字

控制序列化过程时，可能有一个特定的子对象不愿让 Java 的序列化机制自动保存与恢复。一般地，若那个子对象包含了不想序列化的敏感信息（如密码），就会面临这种情况。即使那种信息在对象中具有“private”（私有）属性，但一旦经序列化处理，人们就可以通过读取一个文件，或者拦截网络传输得到它。

为防止对象的敏感部分被序列化，一个办法是将自己的类实现为 Externalizable，就象前面展示的那样。这样一来，没有任何东西可以自动序列化，只能在 writeExternal()明确序列化那些需要的部分。

然而，若操作的是一个 Serializable 对象，所有序列化操作都会自动进行。为解决这个问题，可以用 transient（临时）逐个字段地关闭序列化，它的意思是“不要麻烦你（指自动机制）保存或恢复它了——我会自己处理的”。

例如，假设一个 Login 对象包含了与一个特定的登录会话有关的信息。校验登录的合法性时，一般都想将数据保存下来，但不包括密码。为做到这一点，最简单的办法是实现 Serializable，并将 password 字段设为 transient。下面是具体的代码：

499-500 页程序

可以看到，其中的 date 和 username 字段保持原始状态（未设成 transient），所以会自动序列化。然而，password 被设为 transient，所以不会自动保存到磁盘；另外，自动序列化机制也不会作恢复它的尝试。输出如下：

501 页上程序

一旦对象恢复成原来的样子，password 字段就会变成 null。注意必须用 toString()检查 password 是否为 null，因为若用过载的“+”运算符来装配一个 String 对象，而且那个运算符遇到一个 null 句柄，就会造成一个名为 NullPointerException 的违例（新版 Java 可能会提供避免这个问题的代码）。

我们也发现 date 字段被保存到磁盘，并从磁盘恢复，没有重新生成。

由于 Externalizable 对象默认时不保存它的任何字段，所以 transient 关键字只能伴随 Serializable 使用。

2. Externalizable 的替代方法

若不是特别在意要实现 Externalizable 接口，还有另一种方法可供选用。我们可以实现 Serializable 接口，并添加（注意是“添加”，而非“覆盖”或者“实现”）名为 writeObject()和 readObject()的方法。一旦对象被序列化或者重新装配，就会分别调用那两个方法。也就是说，只要提供了这两个方法，就会优先使用它们，而不考虑默认的序列化机制。

这些方法必须含有下列准确的签名：

501 页下程序

从设计的角度出发，情况变得有些扑朔迷离。首先，大家可能认为这些方法不属于基础类或者 Serializable 接口的一部分，它们应该在自己的接口中得到定义。但请注意它们被定义成“private”，这意味着它们只能由这个类的其他成员调用。然而，我们实际并不从这个类的其他成员中调用它们，而是由 ObjectOutputStream 和 ObjectInputStream 的 writeObject()及 readObject()方法来调用我们对象的 writeObject()和 readObject()方法（注意我在这里用了很大的抑制力来避免使用相同的方法名——因为怕混淆）。大家可能奇怪 ObjectOutputStream 和 ObjectInputStream 如何有权访问我们的类的 private 方法——只能认为这是序列化机制玩的一个把戏。

在任何情况下，接口中的定义的任何东西都会自动具有 public 属性，所以假若 writeObject()和 readObject()必须为 private，那么它们不能成为接口（interface）的一部分。但由于我们准确地加上了签名，所以最终的效果实际与实现一个接口是相同的。

看起来似乎我们调用 ObjectOutputStream.writeObject()的时候，我们传递给它的 Serializable 对象似乎会被检查是否实现了自己的 writeObject()。若答案是肯定的是，便会跳过常规的序列化过程，并调用 writeObject()。readObject()也会遇到同样的情况。

还存在另一个问题。在我们的 writeObject()内部，可以调用 defaultWriteObject()，从而决定采取默认的 writeObject()行动。类似地，在 readObject()内部，可以调用 defaultReadObject()。下面这个简单的例子演示了如何对一个 Serializable 对象的存储与恢复进行控制：

502-503 页程序

在这个例子中，一个 String 保持原始状态，其他设为 transient（临时），以便证明非临时字段会被 defaultWriteObject()方法自动保存，而 transient 字段必须在程序中明确保存和恢复。字段是在构建器内部初始化的，而不是在定义的时候，这证明了它们不会在重新装配的时候被某些自动化机制初始化。

若准备通过默认机制写入对象的非 transient 部分，那么必须调用 defaultWriteObject()，令其作为 writeObject()中的第一个操作；并调用 defaultReadObject()，令其作为 readObject()的第一个操作。这些都是不常见的调用方法。举个例子来说，当我们为一个 ObjectOutputStream 调用 defaultWriteObject()的时候，而且没有为其传递参数，就需要采取这种操作，使其知道对象的句柄以及如何写入所有非 transient 的部分。这种做法非常不便。

transient 对象的存储与恢复采用了我们更熟悉的代码。现在考虑一下会发生一些什么事情。在 main()中会创建一个 SerialCtl 对象，随后会序列化到一个 ObjectOutputStream 里（注意这种情况下使用的是一个缓冲区，而非文件——与 ObjectOutputStream 完全一致）。正式的序列化操作是在下面这行代码里发生的：

o.writeObject(sc);

其中，writeObject()方法必须核查 sc，判断它是否有自己的 writeObject()方法（不是检查它的接口——它根本就没有，也不是检查类的类型，而是利用反射方法实际搜索方法）。若答案是肯定的，就使用那个方法。类似的情况也会在 readObject()上发生。或许这是解决问题唯一实际的方法，但确实显得有些古怪。

3. 版本问题

有时候可能想改变一个可序列化的类的版本（比如原始类的对象可能保存在数据库中）。尽管这种做法得到了支持，但一般只应在非常特殊的情况下才用它。此外，它要求操作者对背后的原理有一个比较深的认识，而我们在这里还不想达到这种深度。JDK 1.1 的 HTML 文档对这一主题进行了非常全面的论述（可从 Sun 公司下载，但可能也成了 Java 开发包联机文档的一部分）。

10.9.3 利用“持久性”

一个比较诱人的想法是用序列化技术保存程序的一些状态信息，从而将程序方便地恢复到以前的状态。但在具体实现以前，有些问题是必须解决的。如果两个对象都有指向第三个对象的句柄，该如何对这两个对象序列化呢？如果从两个对象序列化后的状态恢复它们，第三个对象的句柄只会出现在一个对象身上吗？如果将这两个对象序列化成独立的文件，然后在代码的不同部分重新装配它们，又会得到什么结果呢？

下面这个例子对上述问题进行了很好的说明：

504-506 页程序

这里一件有趣的事情是也许是能针对一个字节数组应用对象的序列化，从而实现对任何 Serializable（可序列化）对象的一个“全面复制”（全面复制意味着复制的是整个对象网，而不仅是基本对象和它的句柄）。复制问题将在第 12 章进行全面讲述。

Animal 对象包含了类型为 House 的字段。在 main()中，会创建这些 Animal 的一个 Vector，并对其序列化两次，分别送入两个不同的数据流内。这些数据重

新装配并打印出来后，可看到下面这样的结果（对象在每次运行时都会处在不同的内存位置，所以每次运行的结果有区别）：

506-507 页程序

当然，我们希望装配好的对象有与原来不同的地址。但注意在 animals1 和 animals2 中出现了相同的地址，其中包括共享的、对 House 对象的引用。在另一方面，当 animals3 恢复以后，系统没有办法知道另一个流内的对象是第一个流内对象的化身，所以会产生一个完全不同的对象网。

只要将所有东西都序列化到单独一个数据流里，就能恢复获得与以前写入时完全一样的对象网，不会不慎造成对象的重复。当然，在写第一个和最后一个对象的时间之间，可改变对象的状态，但那必须由我们明确采取操作——序列化时，对象会采用它们当时的任何状态（包括它们与其他对象的连接关系）写入。

若想保存系统状态，最安全的做法是当作一种“微观”操作序列化。如果序列化了某些东西，再去做其他一些工作，再来序列化更多的东西，以此类推，那么最终将无法安全地保存系统状态。相反，应将构成系统状态的所有对象都置入单个集合内，并在一次操作里完成那个集合的写入。这样一来，同样只需一次方法调用，即可成功恢复之。

下面这个例子是一套假想的计算机辅助设计（CAD）系统，对这一方法进行了很好的演示。此外，它还为我们引入了 static 字段的问题——如留意联机文档，就会发现 Class 是“Serializable”（可序列化）的，所以只需简单地序列化 Class 对象，就能实现 static 字段的保存。这无论如何都是一种明智的做法。

507-510 页程序

Shape（几何形状）类“实现了可序列化”（implements Serializable），所以从 Shape 继承的任何东西也都会自动“可序列化”。每个 Shape 都包含了数据，而且每个衍生的 Shape 类都包含了一个特殊的 static 字段，用于决定所有那些类型的 Shape 的颜色（如将一个 static 字段置入基础类，结果只会产生一个字段，因为 static 字段未在衍生类中复制）。可对基础类中的方法进行覆盖处理，以便为不同的类型设置颜色（static 方法不会动态绑定，所以这些都是普通的方法）。每次调用 randomFactory()方法时，它都会创建一个不同的 Shape（Shape 值采用随机值）。

Circle（圆）和 Square（矩形）属于对 Shape 的直接扩展；唯一的差别是 Circle 在定义时会初始化颜色，而 Square 在构建器中初始化。Line（直线）的问题将留到以后讨论。

在 main()中，一个 Vector 用于容纳 Class 对象，而另一个用于容纳形状。若不提供相应的命令行参数，就会创建 shapeTypes Vector，并添加 Class 对象。然后创建 shapes Vector，并添加 Shape 对象。接下来，所有 static color 值都会设成 GREEN，而且所有东西都会序列化到文件 CADState.out。

若提供了一个命令行参数（假设 CADState.out），便会打开那个文件，并用它恢复程序的状态。无论在哪种情况下，结果产生的 Shape 的 Vector 都会打印出来。下面列出它某一次运行的结果：

从中可以看出，xPos，yPos 以及 dim 的值都已成功保存和恢复出来。但在获取 static 信息时却出现了问题。所有“3”都已进入，但没有正常地出来。Circle 有一个 1 值（定义为 RED），而 Square 有一个 0 值（记住，它们是在构建器里初始化的）。看上去似乎 static 根本没有得到初始化！实情正是如此——尽管类 Class 是“可以序列化的”，但却不能按我们希望的工作。所以假如想序列化 static 值，必须亲自动手。

这正是 Line 中的 serializeStaticState()和 deserializeStaticState()两个 static 方法的用途。可以看到，这两个方法都是作为存储和恢复进程的一部分明确调用的（注意写入序列化文件和从中读回的顺序不能改变）。所以为了使 CADState.java 正确运行起来，必须采用下述三种方法之一：

(1) 为几何形状添加一个 serializeStaticState()和 deserializeStaticState()。

(2) 删除 Vector shapeTypes 以及与之有关的所有代码

(3) 在几何形状内添加对新序列化和撤消序列化静态方法的调用

要注意的另一个问题是安全，因为序列化处理也会将 private 数据保存下来。若有需要保密的字段，应将其标记成 transient。但在这之后，必须设计一种安全的信息保存方法。这样一来，一旦需要恢复，就可以重设那些 private 变量。

10.10 总结

Java IO 流库能满足我们的许多基本要求：可以通过控制台、文件、内存块甚至因特网（参见第 15 章）进行读写。可以创建新的输入和输出对象类型（通过从 InputStream 和 OutputStream 继承）。向一个本来预期为收到字串的方法传递一个对象时，由于 Java 已限制了“自动类型转换”，所以会自动调用 toString()方法。而我们可以重新定义这个 toString()，扩展一个数据流能接纳的对象种类。

在 IO 数据流库的联机文档和设计过程中，仍有些问题没有解决。比如当我们打开一个文件以便输出时，完全可以指定一旦有人试图覆盖该文件就“掷”出一个违例——有的编程系统允许我们自行指定想打开一个输出文件，但唯一的前提是它尚不存在。但在 Java 中，似乎必须用一个 File 对象来判断某个文件是否存在，因为假如将其作为 FileOutputStream 或者 FileWriter 打开，那么肯定会被覆盖。若同时指定文件和目录路径，File 类设计上的一个缺陷就会暴露出来，因为它会说“不要试图在单个类里做太多的事情”！

IO 流库易使我们混淆一些概念。它确实能做许多事情，而且也可以移植。但假如假如事先没有吃透装饰器方案的概念，那么所有的设计都多少带有一点盲目性质。所以不管学它还是教它，都要特别花一些功夫才行。而且它并不完整：没有提供对输出格式化的支持，而其他几乎所有语言的 IO 包都提供了这方面的支持（这一点没有在 Java 1.1 里得以纠正，它完全错失了改变库设计方案的机会，反而增添了更特殊的一些情况，使复杂程度进一步提高）。Java 1.1 转到那些尚未替换的 IO 库，而不是增加新库。而且库的设计人员似乎没有很好地指出哪些特性是不赞成的，哪些是首选的，造成库设计中经常都会出现一些令人恼火的反对消息。

然而，一旦掌握了装饰器方案，并开始在一些较为灵活的环境使用库，就会认识到这种设计的好处。到那个时候，为此多付出的代码行应该不至于使你觉得太生气。

10.11 练习

(1) 打开一个文本文件，每次读取一行内容。将每行作为一个 String 读入，并将那个 String 对象置入一个 Vector 里。按相反的顺序打印出 Vector 中的所有行。

(2) 修改练习 1，使读取那个文件的名字作为一个命令行参数提供。

(3) 修改练习 2，又打开一个文本文件，以便将文字写入其中。将 Vector 中的行随同行号一起写入文件。

(4) 修改练习 2，强迫 Vector 中的所有行都变成大写形式，将结果发给 System.out。

(5) 修改练习 2，在文件中查找指定的单词。打印出包含了欲找单词的所有文本行。

(6) 在 Blips.java 中复制文件，将其重命名为 BlipCheck.java。然后将类 Blip2 重命名为 BlipCheck（在进程中将其标记为 public）。删除文件中的//!记号，并执行程序。接下来，将 BlipCheck 的默认构建器变成注释信息。运行它，并解释为什么仍然能够工作。

(7) 在 Blip3.java 中，将接在“You must do this:”字样后的两行变成注释，然后运行程序。解释得到的结果为什么会与执行了那两行代码不同。

(8) 转换 SortedWordCount.java 程序，以便使用 Java 1.1 IO 流。

(9) 根据本章正文的说明修改程序 CADState.java。

(10) 在第 7 章（中间部分）找到 GreenhouseControls.java 示例，它应该由三个文件构成。在 GreenhouseControls.java 中，Restart()内部类有一个硬编码的事件集。请修改这个程序，使其能从一个文本文件里动态读取事件以及它们的相关时间。