

第2章 输入与输出

- ▲ 输入 / 输出流
- ▲ 文本输入与输出
- ▲ 读写二进制数据
- ▲ 对象输入 / 输出流与序列化

- ▲ 操作文件
- ▲ 内存映射文件
- ▲ 正则表达式

本章将介绍 Java 中用于输入和输出的各种应用编程接口 (Application Programming Interface, API)。你将要学习如何访问文件与目录，以及如何以二进制格式和文本格式来读写数据。本章还要向你展示对象序列化机制，它可以使存储对象像存储文本和数值数据一样容易。然后，我们将介绍使用文件和目录。最后，本章将讨论正则表达式，尽管这部分内容实际上与输入和输出并不相关，但是我们确实也找不到更合适的地方来处理这个话题。很明显，Java 设计团队在这个问题的处理上和我们一样，因为正则表达式 API 的规格说明隶属于“新 I/O”特性的规格说明。

2.1 输入 / 输出流

在 Java API 中，可以从其中读入一个字节序列的对象称做输入流，而可以向其中写入一个字节序列的对象称做输出流。这些字节序列的来源地和目的地可以是文件，而且通常都是文件，但是也可以是网络连接，甚至是内存块。抽象类 `InputStream` 和 `OutputStream` 构成了输入 / 输出 (I/O) 类层次结构的基础。

注意：这些输入 / 输出流与在前一章中看到的流没有任何关系。为了清楚起见，只要是讨论用于输入和输出的流，我们都将使用术语输入流、输出流或输入 / 输出流。

因为面向字节的流不便于处理以 Unicode 形式存储的信息（回忆一下，Unicode 中每个字符都使用了多个字节来表示），所以从抽象类 `Reader` 和 `Writer` 中继承出来了一个专门用于处理 Unicode 字符的单独的类层次结构。这些类拥有的读入和写出操作都是基于两字节的 Char 值的（即，Unicode 码元），而不是基于 byte 值的。

2.1.1 读写字节

`InputStream` 类有一个抽象方法：

```
abstract int read()
```

这个方法将读入一个字节，并返回读入的字节，或者在遇到输入源结尾时返回 -1。在设



计具体的输入流类时，必须覆盖这个方法以提供适用的功能，例如，在 `FileInputStream` 类中，这个方法将从某个文件中读入一个字节，而 `System.in`（它是 `InputStream` 的一个子类的预定义对象）却是从“标准输入”中读入信息，即控制台或重定向的文件。

`InputStream` 类还有若干个非抽象的方法，它们可以读入一个字节数组，或者跳过大量的字节。这些方法都要调用抽象的 `read` 方法，因此，各个子类都只需覆盖这一个方法。

与此类似，`OutputStream` 类定义了下面的抽象方法：

```
abstract void write(int b)
```

它可以向某个输出位置写出一个字节。

`read` 和 `write` 方法在执行时都将阻塞，直至字节确实被读入或写出。这就意味着如果流不能被立即访问（通常是因为网络连接忙），那么当前的线程将被阻塞。这使得在这两个方法等待指定的流变为可用的这段时间里，其他的线程就有机会去执行有用的工作。

`available` 方法使我们可以去检查当前可读入的字节数量，这意味着像下面这样的代码片段就不可能被阻塞：

```
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
    byte[] data = new byte[bytesAvailable];
    in.read(data);
}
```

当你完成对输入 / 输出流的读写时，应该通过调用 `close` 方法来关闭它，这个调用会释放掉十分有限的操作系统资源。如果一个应用程序打开了过多的输入 / 输出流而没有关闭，那么系统资源将被耗尽。关闭一个输出流的同时还会冲刷用于该输出流的缓冲区：所有被临时置于缓冲区中，以便用更大的包的形式传递的字节在关闭输出流时都将被送出。特别是，如果不关闭文件，那么写出字节的最后一个包可能将永远也得不到传递。当然，我们还可以用 `flush` 方法来人为地冲刷这些输出。

即使某个输入 / 输出流类提供了使用原生的 `read` 和 `write` 功能的某些具体方法，应用系统的程序员还是很少使用它们，因为大家感兴趣的数据可能包含数字、字符串和对象，而不是原生字节。

我们可以使用众多的从基本的 `InputStream` 和 `OutputStream` 类导出的某个输入 / 输出类，而不只是直接使用字节。

API `java.io.InputStream` 1.0

- `abstract int read()`

从数据中读入一个字节，并返回该字节。这个 `read` 方法在碰到输入流的结尾时返回 -1。

- `int read(byte[] b)`

读入一个字节数组，并返回实际读入的字节数，或者在碰到输入流的结尾时返回 -1。

这个 `read` 方法最多读入 `b.length` 个字节。

● **int read(byte[] b, int off, int len)**

读入一个字节数组。这个 `read` 方法返回实际读入的字节数，或者在碰到输入流的结尾时返回 -1。

参数: `b` 数据读入的数组

`off` 第一个读入字节应该被放置的位置在 `b` 中的偏移量

`len` 读入字节的最大数量

● **long skip(long n)**

在输入流中跳过 `n` 个字节，返回实际跳过的字节数（如果碰到输入流的结尾，则可能小于 `n`）。

● **int available()**

返回在不阻塞的情况下可获取的字节数（回忆一下，阻塞意味着当前线程将失去它对资源的占用）。

● **void close()**

关闭这个输入流。

● **void mark(int readlimit)**

在输入流的当前位置打一个标记（并非所有的流都支持这个特性）。如果从输入流中已经读入的字节多于 `readlimit` 个，则这个流允许忽略这个标记。

● **void reset()**

返回到最后一个标记，随后对 `read` 的调用将重新读入这些字节。如果当前没有任何标记，则这个流不被重置。

● **boolean markSupported()**

如果这个流支持打标记，则返回 `true`。

API java.io.OutputStream 1.0

● **abstract void write(int n)**

写出一个字节的数据。

● **void write(byte[] b)**

● **void write(byte[] b, int off, int len)**

写出所有字节或者某个范围的字节到数组 `b` 中。

参数: `b` 数据写出的数组

`off` 第一个写出字节在 `b` 中的偏移量

`len` 写出字节的最大数量

● **void close()**

冲刷并关闭输出流。

● **void flush()**

冲刷输出流，也就是将所有缓冲的数据发送到目的地。

2.1.2 完整的流家族

与 C 语言只有单一类型 FILE* 包打天下不同，Java 拥有一个流家族，包含各种输入 / 输出流类型，其数量超过 60 个！请参见图 2-1 和图 2-2。

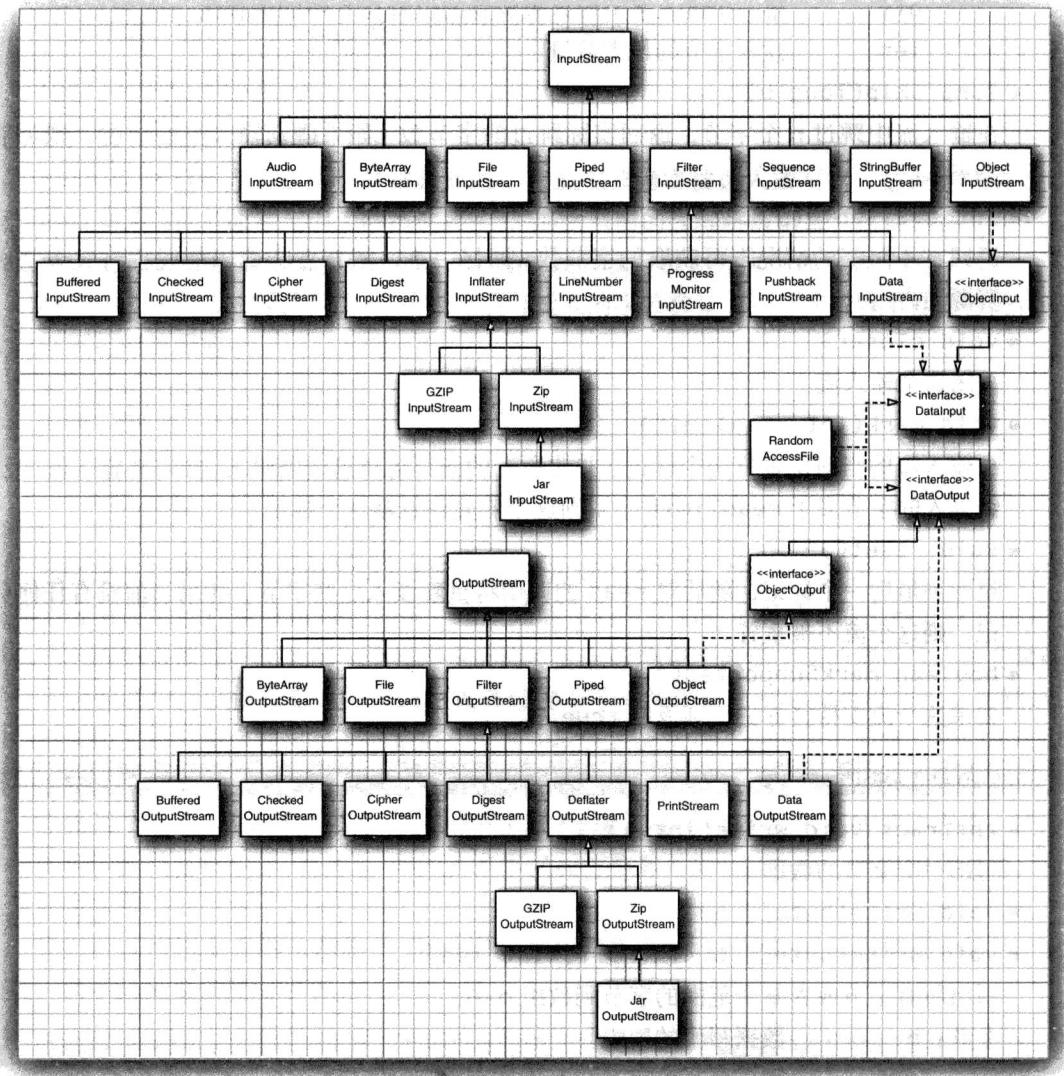


图 2-1 输入流与输出流的层次结构

让我们把输入 / 输出流家族中的成员按照它们的使用方法来进行划分，这样就形成了处理字节和字符的两个单独的层次结构。正如所见，`InputStream` 和 `OutputStream` 类可以读写单个字节或字节数组，这些类构成了图 2-1 所示的层次结构的基础。要想读写字符串和

数字，就需要功能更强大的子类，例如，`DataInputStream` 和 `DataOutputStream` 可以以二进制格式读写所有的基本 Java 类型。最后，还包含了多个很有用的输入 / 输出流，例如，`ZipInputStream` 和 `ZipOutputStream` 可以以常见的 ZIP 压缩格式读写文件。

另一方面，对于 Unicode 文本，可以使用抽象类 `Reader` 和 `Writer` 的子类（请参见图 2-2）。`Reader` 和 `Writer` 类的基本方法与 `InputStream` 和 `OutputStream` 中的方法类似。

```
abstract int read()
abstract void write(int c)
```

`read` 方法将返回一个 Unicode 码元（一个在 0 ~ 65535 之间的整数），或者在碰到文件结尾时返回 -1。`write` 方法在被调用时，需要传递一个 Unicode 码元（请查看卷 I 第 3 章有关 Unicode 码元的讨论）。

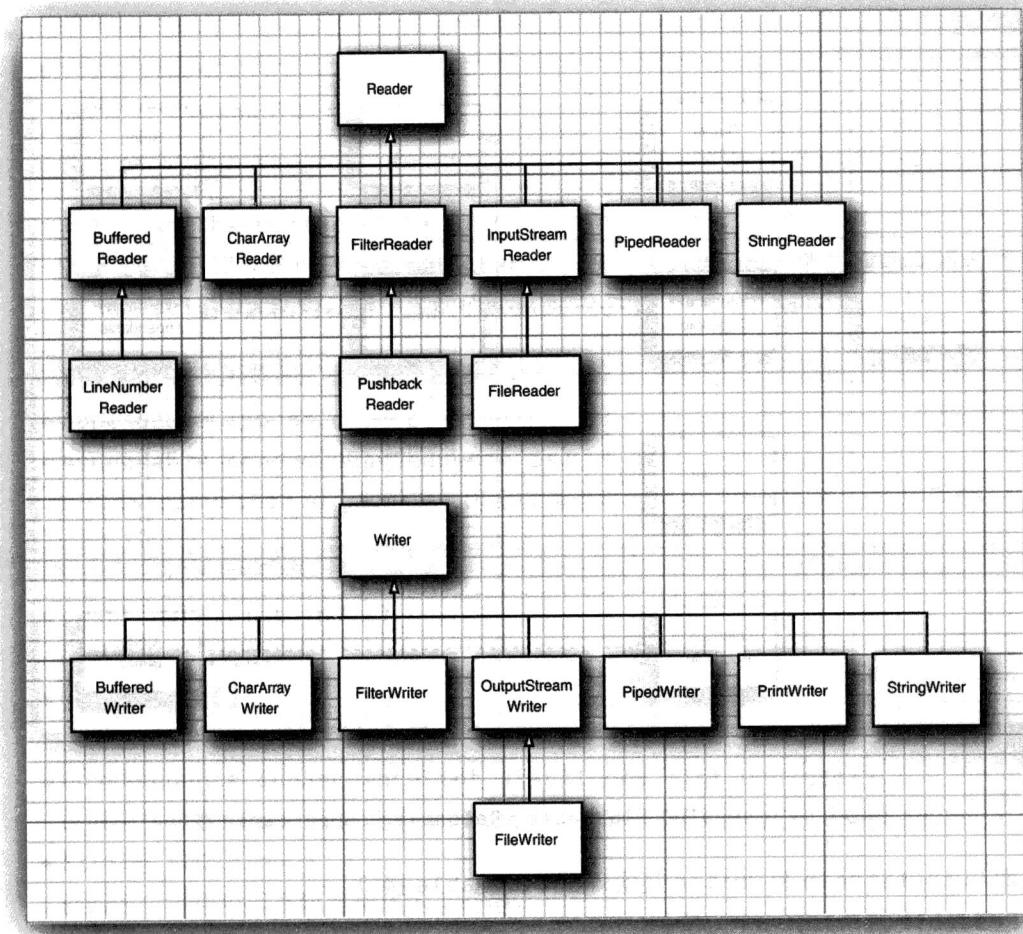


图 2-2 Reader 和 Writer 的层次结构

还有4个附加的接口：`Closeable`、`Flushable`、`Readable`和`Appendable`（请查看图2-3）。前两个接口非常简单，它们分别拥有下面的方法：

```
void close() throws IOException
```

和

```
void flush()
```

`InputStream`、`OutputStream`、`Reader`和`Writer`都实现了`Closeable`接口。

注意：`java.io.Closeable`接口扩展了`java.lang.AutoCloseable`接口。因此，对任何`Closeable`进行操作时，都可以使用try-with-resource语句（try-with-resource语句是指声明了一个或多个资源的try语句——译者注）。为什么要有两个接口呢？因为`Closeable`接口的`close`方法只抛出`IOException`，而`AutoCloseable.close`方法可以抛出任何异常。

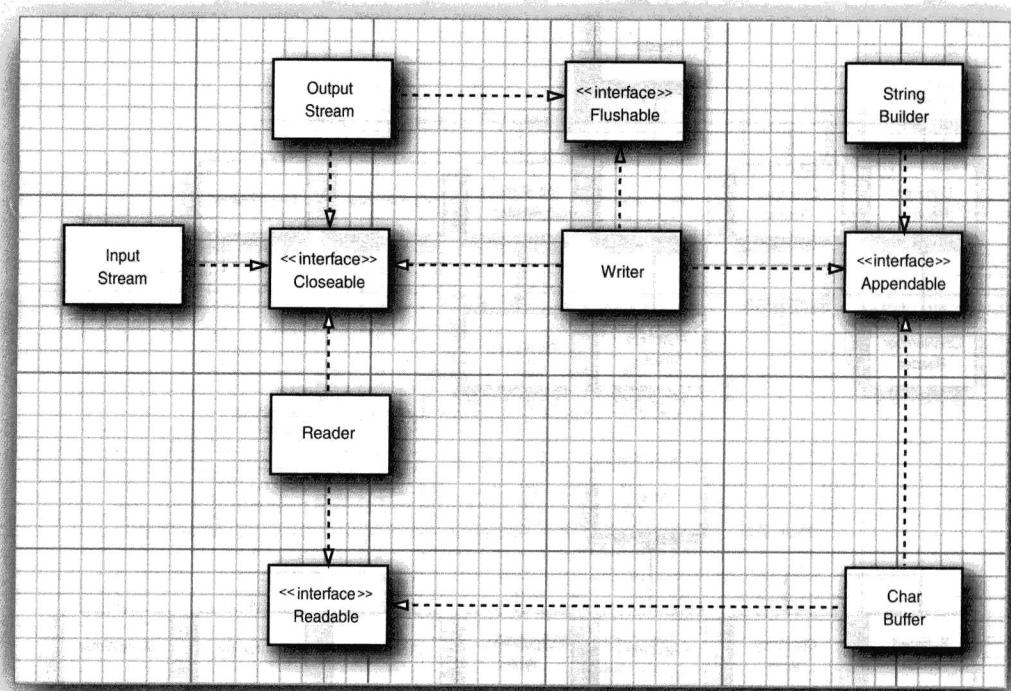


图2-3 `Closeable`、`Flushable`、`Readable`和`Appendable`接口

而`OutputStream`和`Writer`还实现了`Flushable`接口。

`Readable`接口只有一个方法：

```
int read(CharBuffer cb)
```

`CharBuffer`类拥有按顺序和随机地进行读写访问的方法，它表示一个内存中的缓冲区

或者一个内存映像的文件（请参见 2.6.2 节以了解细节）。

Appendable 接口有两个用于添加单个字符和字符序列的方法：

```
Appendable append(char c)
Appendable append(CharSequence s)
```

CharSequence 接口描述了一个 **char** 值序列的基本属性，**String**、**CharBuffer**、**StringBuilder** 和 **StringBuffer** 都实现了它。

在流类的家族中，只有 **Writer** 实现了 **Appendable**。

API **java.io.Closeable 5.0**

- **void close()**

关闭这个 **Closeable**，这个方法可能会抛出 **IOException**。

API **java.io.Flushable 5.0**

- **void flush()**

冲刷这个 **Flushable**。

API **java.lang.Readable 5.0**

- **int read(CharBuffer cb)**

尝试着向 **cb** 读入其可持有数量的 **char** 值。返回读入的 **char** 值的数量，或者当从这个 **Readable** 中无法再获得更多的值时返回 -1。

API **java.lang.Appendable 5.0**

- **Appendable append(char c)**

- **Appendable append(CharSequence cs)**

向这个 **Appendable** 中追加给定的码元或者给定的序列中的所有码元，返回 **this**。

API **java.lang.CharSequence 1.4**

- **char charAt(int index)**

返回给定索引处的码元。

- **int length()**

返回在这个序列中的码元的数量。

- **CharSequence subSequence(int startIndex, int endIndex)**

返回由存储在 **startIndex** 到 **endIndex-1** 处的所有码元构成的 **CharSequence**。

- **String toString()**

返回这个序列中所有码元构成的字符串。

2.1.3 组合输入 / 输出流过滤器

FileInputStream 和 **FileOutputStream** 可以提供附着在一个磁盘文件上的输入流和

输出流，而你只需向其构造器提供文件名或文件的完整路径名。例如：

```
FileInputStream fin = new FileInputStream("employee.dat");
```

这行代码可以查看在用户目录下名为“`employee.dat`”的文件。

 **提示：**所有在 `java.io` 中的类都将相对路径名解释为以用户工作目录开始，你可以通过调用 `System.getProperty("user.dir")` 来获得这个信息。

 **警告：**由于反斜杠字符在 Java 字符串中是转义字符，因此要确保在 Windows 风格的路径名中使用 `\`（例如，`C:\\Windows\\win.ini`）。在 Windows 中，还可以使用单斜杠字符（`C:/Windows/win.ini`），因为大部分 Windows 文件处理的系统调用都会将斜杠解释成文件分隔符。但是，并不推荐这样做，因为 Windows 系统函数的行为会因与时俱进而发生变化。因此，对于可移植的程序来说，应该使用程序所运行平台的文件分隔符，我们可以通过常量字符串 `java.io.File.separator` 获得它。

与抽象类 `InputStream` 和 `OutputStream` 一样，这些类只支持在字节级别上的读写。也就是说，我们只能从 `fin` 对象中读入字节和字节数组。

```
byte b = (byte) fin.read();
```

正如下节中看到的，如果我们只有 `DataInputStream`，那么我们就只能读入数值类型：

```
DataInputStream din = . . .;
double x = din.readDouble();
```

但是正如 `FileInputStream` 没有任何读入数值类型的方法一样，`DataInputStream` 也没有任何从文件中获取数据的方法。

Java 使用了一种灵巧的机制来分离这两种职责。某些输入流（例如 `FileInputStream` 和由 URL 类的 `openStream` 方法返回的输入流）可以从文件和其他更外部的位置上获取字节，而其他的输入流（例如 `DataInputStream`）可以将字节组装到更有用的数据类型中。Java 程序员必须对二者进行组合。例如，为了从文件中读入数字，首先需要创建一个 `FileInputStream`，然后将其传递给 `DataInputStream` 的构造器：

```
FileInputStream fin = new FileInputStream("employee.dat");
DataInputStream din = new DataInputStream(fin);
double x = din.readDouble();
```

如果再次查看图 2-1，你就会看到 `FilterInputStream` 和 `FilterOutputStream` 类。这些文件的子类用于向处理字节的输入 / 输出流添加额外的功能。

你可以通过嵌套过滤器来添加多重功能。例如，输入流在默认情况下是不被缓冲区缓存的，也就是说，每个对 `read` 的调用都会请求操作系统再分发一个字节。相比之下，请求一个数据块并将其置于缓冲区中会显得更加高效。如果我们想使用缓冲机制，以及用于文件的数据输入方法，那么就需要使用下面这种相当恐怖的构造器序列：

```
DataInputStream din = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
```



注意，我们把 `DataInputStream` 置于构造器链的最后，这是因为我们希望使用 `DataInputStream` 的方法，并且希望它们能够使用带缓冲机制的 `read` 方法。

有时当多个输入流链接在一起时，你需要跟踪各个中介输入流（intermediate input stream）。例如，当读入输入时，你经常需要预览下一个字节，以了解它是否是你想要的值。Java 提供了用于此目的的 `PushbackInputStream`：

```
PushbackInputStream pbin = new PushbackInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
```

现在你可以预读下一个字节：

```
int b = pbin.read();
```

并且在它并非你所期望的值时将其推回流中。

```
if (b != '<') pbin.unread(b);
```

但是读入和推回是可应用于可回推（pushback）输入流的仅有的方法。如果你希望能够预先浏览并且还可以读入数字，那么你就需要一个既是可回推输入流，又是一个数据输入流的引用。

```
DataInputStream din = new DataInputStream(
    pbin = new PushbackInputStream(
        new BufferedInputStream(
            new FileInputStream("employee.dat"))));
```

当然，在其他编程语言的输入 / 输出流类库中，诸如缓冲机制和预览等细节都是自动处理的。因此，相比较而言，Java 就有一点麻烦，它必须将多个流过滤器组合起来。但是，这种混合并匹配过滤器类以构建真正有用的输入 / 输出流序列的能力，将带来极大的灵活性，例如，你可以从一个 ZIP 压缩文件中通过使用下面的输入流序列来读入数字（请参见图 2-4）：

```
ZipInputStream zin = new ZipInputStream(new FileInputStream("employee.zip"));
DataInputStream din = new DataInputStream(zin);
```

（请查看 2.3.3 节以了解更多有关 Java 处理 ZIP 文件功能的知识。）

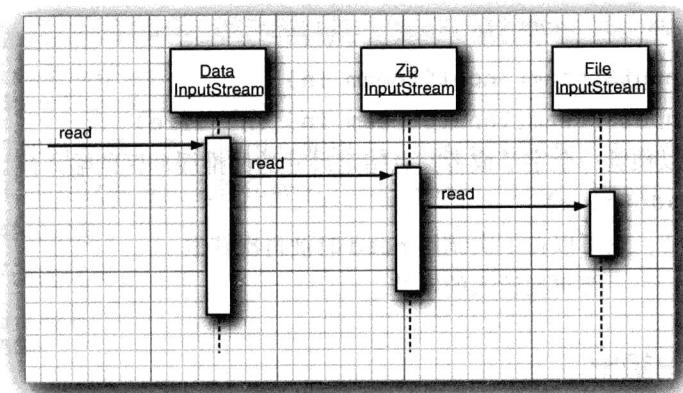


图 2-4 过滤器流序列

API `java.io.FileInputStream 1.0`

- `FileInputStream(String name)`
- `FileInputStream(File file)`

使用由 `name` 字符串或 `file` 对象指定路径名的文件创建一个新的文件输入流 (`File` 类在本章结尾处描述)。非绝对的路径名将按照相对于 VM 启动时所设置的工作目录来解析。

API `java.io.FileOutputStream 1.0`

- `FileOutputStream(String name)`
- `FileOutputStream(String name, boolean append)`
- `FileOutputStream(File file)`
- `FileOutputStream(File file, boolean append)`

使用由 `name` 字符串或 `file` 对象指定路径名的文件创建一个新的文件输出流 (`File` 类在本章结尾处描述)。如果 `append` 参数为 `true`, 那么数据将被添加到文件尾, 而具有相同名字的已有文件不会被删除; 否则, 这个方法会删除所有具有相同名字的已有文件。

API `java.io.BufferedInputStream 1.0`

- `BufferedInputStream(InputStream in)`

创建一个带缓冲区的输入流。带缓冲区的输入流在从流中读入字符时, 不会每次都对设备访问。当缓冲区为空时, 会向缓冲区中读入一个新的数据块。

API `java.io.BufferedOutputStream 1.0`

- `BufferedOutputStream(OutputStream out)`

创建一个带缓冲区的输出流。带缓冲区的输出流在收集要写出的字符时, 不会每次都对设备访问。当缓冲区填满或当流被冲刷时, 数据就被写出。

API `java.io.PushbackInputStream 1.0`

- `PushbackInputStream(InputStream in)`
- `PushbackInputStream(InputStream in, int size)`

构建一个可以预览一个字节或者具有指定尺寸的回推缓冲区的输入流。

- `void unread(int b)`

回推一个字节, 它可以在下次调用 `read` 时被再次获取。

参数: `b` 要再次读入的字节。

2.2 文本输入与输出

在保存数据时, 可以选择二进制格式或文本格式。例如, 整数 1234 存储成二进制数时,

它被写为由字节 00 00 04 D2 构成的序列（十六进制表示法），而存储成文本格式时，它被存成了字符串“1234”。尽管二进制格式的 I/O 高速且高效，但是不宜人来阅读。我们首先讨论文本格式的 I/O，然后在 2.3 节中讨论二进制格式的 I/O。

在存储文本字符串时，需要考虑字符编码（character encoding）方式。在 Java 内部使用的 UTF-16 编码方式中，字符串“1234”编码为 00 31 00 32 00 33 00 34（十六进制）。但是，许多程序都希望文本文件按照其他的编码方式编码。在 UTF-8 这种在互联网上最常用的编码方式中，这个字符串将写出为 4A 6F 73 C3 A9，其中并没有用于前 3 个字母的任何 0 字节，而字符 é 占用了两个字节。

`OutputStreamWriter` 类将使用选定的字符编码方式，把 Unicode 码元的输出流转换为字节流。而 `InputStreamReader` 类将包含字节（用某种字符编码方式表示的字符）的输入流转换为可以产生 Unicode 码元的读入器。

例如，下面的代码就展示了如何让一个输入读入器可以从控制台读入键盘敲击信息，并将其转换为 Unicode：

```
Reader in = new InputStreamReader(System.in);
```

这个输入流读入器会假定使用主机系统所使用的默认字符编码方式。在桌面操作系统中，它可能是像 Windows 1252 或 MacRoman 这样的古老的字符编码方式。你应该总是在 `InputStreamReader` 的构造器中选择一种具体的编码方式。例如，

```
Reader in = new InputStreamReader(new FileInputStream("data.txt"), StandardCharsets.UTF_8);
```

请查看 2.2.4 节以了解字符编码方式的更多信息。

2.2.1 如何写出文本输出

对于文本输出，可以使用 `PrintWriter`。这个类拥有以文本格式打印字符串和数字的方法，它还有一个将 `PrintWriter` 链接到 `FileWriter` 的便捷方法，下面的语句：

```
PrintWriter out = new PrintWriter("employee.txt", "UTF-8");
```

等同于：

```
PrintWriter out = new PrintWriter(
    new FileOutputStream("employee.txt"), "UTF-8");
```

为了输出到打印写出器，需要使用与使用 `System.out` 时相同的 `print`、`println` 和 `printf` 方法。你可以用这些方法来打印数字（`int`、`short`、`long`、`float`、`double`）、字符、`boolean` 值、字符串和对象。

例如，考虑下面的代码：

```
String name = "Harry Hacker";
double salary = 75000;
out.print(name);
out.print(' ');
out.println(salary);
```



它将把下面的字符：

```
Harry Hacker 75000.0
```

输出到写出器 `out`，之后这些字符将会被转换成字节并最终写入 `employee.txt` 中。

`println` 方法在行中添加了对目标系统来说恰当的行结束符（Windows 系统是 "`\r\n`"，UNIX 系统是 "`\n`"），也就是通过调用 `System.getProperty("line.separator")` 而获得的字符串。

如果写出器设置为自动冲刷模式，那么只要 `println` 被调用，缓冲区中的所有字符都会被发送到它们的目的地（打印写出器总是带缓冲区的）。默认情况下，自动冲刷机制是禁用的，你可以通过使用 `PrintWriter(Writer out, Boolean autoFlush)` 来启用或禁用自动冲刷机制：

```
PrintWriter out = new PrintWriter(
    new OutputStreamWriter(
        new FileOutputStream("employee.txt"), "UTF-8"),
    true); // autoflush
```

`print` 方法不抛出异常，你可以调用 `checkError` 方法来查看输出流是否出现了某些错误。

注意：Java 的老手们可能会很想知道 `PrintStream` 类和 `System.out` 底怎么了。在 Java 1.0 中，`PrintStream` 类只是通过将高字节丢弃的方式把所有 Unicode 字符截断成 ASCII 字符。（那时，Unicode 仍旧是 16 位编码方式）很明显，这并非一种干净利落和可移植的方式，这个问题在 Java 1.1 中通过引入读入器和写出器得到了修正。为了与已有的代码兼容，`System.in`、`System.out` 和 `System.err` 仍旧是输入/输出流而不是读入器和写出器。但是现在 `PrintStream` 类在内部采用与 `PrintWriter` 相同的方式将 Unicode 字符转换成了默认的主机编码方式。当你在使用 `print` 和 `println` 方法时，`PrintStream` 类型的对象的行为看起来确实很像打印写出器，但是与打印写出器不同的是，它们允许我们用 `write(int)` 和 `write(byte[])` 方法输出原生字节。

API java.io.PrintWriter 1.1

- `PrintWriter(Writer out)`
- `PrintWriter(Writer writer)`

创建一个向给定的写出器写出的新的 `PrintWriter`。

- `PrintWriter(String filename, String encoding)`
- `PrintWriter(File file, String encoding)`

创建一个使用给定的编码方式向给定的文件写出的新的 `PrintWriter`。

- `void print(Object obj)`
通过打印从 `toString` 产生的字符串来打印一个对象。
- `void print(String s)`

打印一个包含 Unicode 码元的字符串。

● `void println(String s)`

打印一个字符串，后面紧跟一个行终止符。如果这个流处于自动冲刷模式，那么就会冲刷这个流。

● `void print(char[] s)`

打印在给定的字符串中的所有 Unicode 码元。

● `void print(char c)`

打印一个 Unicode 码元。

● `void print(int i)`

● `void print(long l)`

● `void print(float f)`

● `void print(double d)`

● `void print(boolean b)`

以文本格式打印给定的值。

● `void printf(String format, Object... args)`

按照格式字符串指定的方式打印给定的值。请查看卷 I 第 3 章以了解格式化字符串的相关规范。

● `boolean checkError()`

如果产生格式化或输出错误，则返回 `true`。一旦这个流碰到了错误，它就受到了污染，并且所有对 `checkError` 的调用都将返回 `true`。

2.2.2 如何读入文本输入

最简单的处理任意文本的方式就是使用在卷 I 中我们广泛使用的 `Scanner` 类。我们可以从任何输入流中构建 `Scanner` 对象。

或者，我们也可以将短小的文本文件像下面这样读入到一个字符串中：

```
String content = new String(Files.readAllBytes(path), charset);
```

但是，如果想要将这个文件一行行地读入，那么可以调用：

```
List<String> lines = Files.readAllLines(path, charset);
```

如果文件太大，那么可以将惰性处理为一个 `Stream<String>` 对象：

```
try (Stream<String> lines = Files.lines(path, charset))
{
    ...
}
```

在早期的 Java 版本中，处理文本输入的唯一方式就是通过 `BufferedReader` 类。它的 `readLine` 方法会产生一行文本，或者在无法获得更多的输入时返回 `null`。典型的输入循环看起来像下面这样：



```

InputStream inputStream = . . .;
try (BufferedReader in = new BufferedReader(new InputStreamReader(inputStream,
                                                               StandardCharsets.UTF_8)))
{
    String line;
    while ((line = in.readLine()) != null)
    {
        do something with line
    }
}

```

如今，`BufferedReader` 类又有了一个 `lines` 方法，可以产生一个 `Stream<String>` 对象。但是，与 `Scanner` 不同，`BufferedReader` 没有用于任何读入数字的方法。

2.2.3 以文本格式存储对象

在本节，我们将带你领略一个示例程序，它将一个 `Employee` 记录数组存储成了一个文本文件，其中每条记录都保存成单独的一行，而实例字段彼此之间使用分隔符分离开，这里我们使用竖线 (|) 作为分隔符（冒号（:）是另一种流行的选择，有趣的是，每个人都会使用不同的分隔符）。因此，我们这里是在假设不会发生在要存储的字符串中存在 | 的情况。

下面是一个记录集的样本：

```

Harry Hacker|35500|1989-10-01
Carl Cracker|75000|1987-12-15
Tony Tester|38000|1990-03-15

```

写出记录相当简单，因为我们是要写出到一个文本文件中，所以我们使用 `PrintWriter` 类。我们直接写出所有的字段，每个字段后面跟着一个 |，而最后一个字段的后面跟着一个 \n。这项工作是在下面这个我们添加到 `Employee` 类中的 `writeEmployee` 方法里完成的：

```

public static void writeEmployee(PrintWriter out, Employee e)
{
    out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
}

```

为了读入记录，我们每次读入一行，然后分离所有的字段。我们使用一个扫描器来读入每一行，然后用 `String.split` 方法将这一行断开成一组标记。

```

public static Employee readEmployee(Scanner in)
{
    String line = in.nextLine();
    String[] tokens = line.split("\\|");
    String name = tokens[0];
    double salary = Double.parseDouble(tokens[1]);
    LocalDate hireDate = LocalDate.parse(tokens[2]);
    int year = hireDate.getYear();
    int month = hireDate.getMonthValue();
    int day = hireDate.getDayOfMonth();
    return new Employee(name, salary, year, month, day);
}

```

`split` 方法的参数是一个描述分隔符的正则表达式，我们在本章的末尾将详细讨论正则

表达式。碰巧的是，竖线在正则表达式中具有特殊的含义，因此需要用\字符来表示转义，而这个\又需要用另一个\来转义，这样就产生了“\\”表达式。

完整的程序如程序清单 2-1 所示。静态方法

```
void writeData(Employee[] e, PrintWriter out)
```

首先写出该数组的长度，然后写出每条记录。静态方法

```
Employee[] readData(BufferedReader in)
```

首先读入该数组的长度，然后读入每条记录。这显得稍微有点棘手：

```
int n = in.nextInt();
in.nextLine(); // consume newline
Employee[] employees = new Employee[n];
for (int i = 0; i < n; i++)
{
    employees[i] = new Employee();
    employees[i].readData(in);
}
```

对 `nextInt` 的调用读入的是数组长度，但不包括行尾的换行字符，我们必须处理掉这个换行符，这样，在调用 `nextLine` 方法后，`readData` 方法就可以获得下一行输入了。

程序清单 2-1 textFile/TextFileTest.java

```
1 package textFile;
2
3 import java.io.*;
4 import java.time.*;
5 import java.util.*;
6
7 /**
8 * @version 1.14 2016-07-11
9 * @author Cay Horstmann
10 */
11 public class TextFileTest
12 {
13     public static void main(String[] args) throws IOException
14     {
15         Employee[] staff = new Employee[3];
16
17         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
18         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
19         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
20
21         // save all employee records to the file employee.dat
22         try (PrintWriter out = new PrintWriter("employee.dat", "UTF-8"))
23         {
24             writeData(staff, out);
25         }
26
27         // retrieve all records into a new array
28         try (Scanner in = new Scanner(
```



```

29         new FileInputStream("employee.dat"), "UTF-8"))
30     {
31         Employee[] newStaff = readData(in);
32
33         // print the newly read employee records
34         for (Employee e : newStaff)
35             System.out.println(e);
36     }
37 }
38
39 /**
40 * Writes all employees in an array to a print writer
41 * @param employees an array of employees
42 * @param out a print writer
43 */
44 private static void writeData(Employee[] employees, PrintWriter out) throws IOException
45 {
46     // write number of employees
47     out.println(employees.length);
48
49     for (Employee e : employees)
50         writeEmployee(out, e);
51 }
52
53 /**
54 * Reads an array of employees from a scanner
55 * @param in the scanner
56 * @return the array of employees
57 */
58 private static Employee[] readData(Scanner in)
59 {
60     // retrieve the array size
61     int n = in.nextInt();
62     in.nextLine(); // consume newline
63
64     Employee[] employees = new Employee[n];
65     for (int i = 0; i < n; i++)
66     {
67         employees[i] = readEmployee(in);
68     }
69     return employees;
70 }
71
72 /**
73 * Writes employee data to a print writer
74 * @param out the print writer
75 */
76 public static void writeEmployee(PrintWriter out, Employee e)
77 {
78     out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
79 }
80
81 /**
82 * Reads employee data from a buffered reader

```



```

83     * @param in the scanner
84     */
85     public static Employee readEmployee(Scanner in)
86     {
87         String line = in.nextLine();
88         String[] tokens = line.split("\\|");
89         String name = tokens[0];
90         double salary = Double.parseDouble(tokens[1]);
91         LocalDate hireDate = LocalDate.parse(tokens[2]);
92         int year = hireDate.getYear();
93         int month = hireDate.getMonthValue();
94         int day = hireDate.getDayOfMonth();
95         return new Employee(name, salary, year, month, day);
96     }
97 }

```

2.2.4 字符编码方式

输入和输出流都是用于字节序列的，但是在许多情况下，我们希望操作的是文本，即字符序列。于是，字符如何编码成字节就成了问题。

Java 针对字符使用的是 Unicode 标准。每个字符或“编码点”都具有一个 21 位的整数。有多种不同的字符编码方式，也就是说，将这些 21 位数字包装成字节的方法有多种。

最常见的编码方式是 UTF-8，它会将每个 Unicode 编码点编码为 1 到 4 个字节的序列（请参阅表 2-1）。UTF-8 的好处是传统的包含了英语中用到的所有字符的 ASCII 字符集中的每个字符都只会占用一个字节。

表 2-1 UTF-8 编码方式

字符范围	编码方式
0...7F	0a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
80...7FF	110a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
800...FFFF	1110a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
10000...10FFFF	11110a ₂₀ a ₁₉ a ₁₈ 10a ₁₇ a ₁₆ a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀

另一种常见的编码方式是 UTF-16，它会将每个 Unicode 编码点编码为 1 个或 2 个 16 位值（请参阅表 2-2）。这是一种在 Java 字符串中使用的编码方式。实际上，有两种形式的 UTF-16，被称为“高位优先”和“低位优先”。考虑一下 16 位值 0x2122。在高位优先格式中，高位字节会先出现：0x21 后面跟着 0x22。但是在低位优先格式中，是另外一种排列方式：0x22 0x21。为了表示使用的是哪一种格式，文件可以以“字节顺序标记”开头，这个标记为 16 位数值 0xFEFF。读入器可以使用这个值来确定字节顺序，然后丢弃它。

表 2-2 UTF-16 编码方式

字符范围	编码方式
0...FFFF	a ₁₅ a ₁₄ a ₁₃ a ₁₂ a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
10000...10FFFF	110110b ₁₉ b ₁₈ b ₁₇ b ₁₆ a ₁₅ a ₁₄ a ₁₃ a ₁₂ a ₁₁ a ₁₀ 110111a ₉ a ₈ a ₇ a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀ 其中 b ₁₉ b ₁₈ b ₁₇ b ₁₆ = a ₂₀ a ₁₉ a ₁₈ a ₁₇ a ₁₆ - 1



◆ 警告：有些程序，包括 Microsoft Notepad（微软记事本）在内，都在 UTF-8 编码的文件开头处添加了一个字节顺序标记。很明显，这并不需要，因为在 UTF-8 中，并不存在字节顺序的问题。但是 Unicode 标准允许这样做，甚至认为这是一种好的做法，因为这样做可以使编码机制不留疑惑。遗憾的是，Java 并没有这么做，有关这个问题的缺陷报告最终是以“will not fix（不做修正）”关闭的。对你来说，最好的做法是将输入中发现的所有先导的 \uFEFF 都剥离掉。

除了 UTF 编码方式，还有一些编码方式，它们各自都覆盖了适用于特定用户人群的字符范围。例如，ISO 8859-1 是一种单字节编码，它包含了西欧各种语言中用到的带有重音符号的字符，而 Shift-JIS 是一种用于日文字符的可变长编码。大量的这些编码方式至今仍在被广泛使用。

不存在任何可靠的方式可以自动地探测出字节流中所使用的字符编码方式。某些 API 方法让我们使用“默认字符集”，即计算机的操作系统首选的字符编码方式。这种字符编码方式与我们的字节源中所使用的编码方式相同吗？字节源中的字节可能来自世界上的其他国家或地区，因此，你应该总是明确指定编码方式。例如，在编写网页时，应该检查 Content-Type 头信息。

◆ 注意：平台使用的编码方式可以由静态方法 `Charset.defaultCharset` 返回。静态方法 `Charset.availableCharsets` 会返回所有可用的 `Charset` 实例，返回结果是一个从字符集的规范名称到 `Charset` 对象的映射表。

◆ 警告：Oracle 的 Java 实现有一个用于覆盖平台默认值的系统属性 `file.encoding`。但是它并非官方支持的属性，并且 Java 库的 Oracle 实现的所有部分并非都以一致的方式处理该属性，因此，你不应该设置它。

`StandardCharsets` 类具有类型为 `Charset` 的静态变量，用于表示每种 Java 虚拟机都必须支持的字符编码方式：

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

为了获得另一种编码方式的 `Charset`，可以使用静态的 `forName` 方法：

```
Charset shiftJIS = Charset.forName("Shift-JIS");
```

在读入或写出文本时，应该使用 `Charset` 对象。例如，我们可以像下面这样将一个字节数组转换为字符串：

```
String str = new String(bytes, StandardCharsets.UTF_8);
```

◆ 提示：有些方法允许我们用一个 `Charset` 对象或字符串来指定字符编码方式。由于选择

的是 `StandardCharsets` 常量，所以无需担心拼写错误。例如，`new String(bytes, "UTF-8")` 就不可接受，并且会引发运行时错误。

! 警告：在不指定任何编码方式时，有些方法（例如 `String(byte[])` 构造器）会使用默认的平台编码方式，而其他方法（例如 `Files.readAllLines`）会使用 UTF-8。

2.3 读写二进制数据

文本格式对于测试和调试而言会显得很方便，因为它是人类可阅读的，但是它并不像以二进制格式传递数据那样高效。在下面的各小节中，你将会学习如何用二进制数据来完成输入和输出。

2.3.1 DataInput 和 DataOutput 接口

`DataOutput` 接口定义了下面用于以二进制格式写数组、字符、`boolean` 值和字符串的方法：

```
writeChars
writeByte
writeInt
writeShort
writeLong
writeFloat
writeDouble
writeChar
writeBoolean
writeUTF
```

例如，`writeInt` 总是将一个整数写出为 4 字节的二进制数量值，而不管它有多少位，`writeDouble` 总是将一个 `double` 值写出为 8 字节的二进制数量值。这样产生的结果并非人可阅读的，但是对于给定类型的每个值，所需的空间都是相同的，而且将其读回也比解析文本要更快。

! 注意：根据你所使用的处理器类型，在内存存储整数和浮点数有两种不同的方法。例如，假设你使用的是 4 字节的 `int`，如果有一个十进制数 1234，也就是十六进制的 4D2（ $1234 = 4 \times 256 + 13 \times 16 + 2$ ），那么它可以按照内存中 4 字节的第一个字节存储最高位字节的方式来存储为：00 00 04 D2，这就是所谓的高位在前顺序（MSB）；我们也可以从最低位字节开始：D2 04 00 00，这种方式自然就是所谓的低位在前顺序（LSB）。例如，SPARC 使用的是高位在前顺序，而 Pentium 使用的则是低位在前顺序。这就可能会带来问题，当存储 C 或者 C++ 文件时，数据会精确地按照处理器存储它们的方式来存储，这就使得即使是最简单的数据在从一个平台迁移到另一个平台上时也是一种挑战。在 Java 中，所有的值都按照高位在前的模式写出，不管使用何种处理器，这使得 Java 数据文件可以独立于平台。

`writeUTF` 方法使用修订版的 8 位 Unicode 转换格式写出字符串。这种方式与直接使用标准的 UTF-8 编码方式不同，其中，Unicode 码元序列首先用 UTF-16 表示，其结果之后使用 UTF-8 规则进行编码。修订后的编码方式对于编码大于 0xFFFF 的字符的处理有所不同，这是为了向后兼容在 Unicode 还没有超过 16 位时构建的虚拟机。

因为没有其他方法会使用 UTF-8 的这种修订，所以你应该只在写出用于 Java 虚拟机的字符串时才使用 `writeUTF` 方法，例如，当你需要编写一个生成字节码的程序时。对于其他场合，都应该使用 `writeChars` 方法。

为了读回数据，可以使用在 `DataInput` 接口中定义的下列方法：

```
readInt  
readShort  
readLong  
readFloat  
readDouble  
readChar  
readBoolean  
readUTF
```

`DataInputStream` 类实现了 `DataInput` 接口，为了从文件中读入二进制数据，可以将 `DataInputStream` 与某个字节源相组合，例如 `FileInputStream`：

```
DataInputStream in = new DataInputStream(new FileInputStream("employee.dat"));
```

与此类似，要想写出二进制数据，你可以使用实现了 `DataOutput` 接口的 `DataOutputStream` 类：

```
DataOutputStream out = new DataOutputStream(new FileOutputStream("employee.dat"));
```

API `java.io.DataInput 1.0`

- `boolean readBoolean()`
- `byte readByte()`
- `char readChar()`
- `double readDouble()`
- `float readFloat()`
- `int readInt()`
- `long readLong()`
- `short readShort()`

读入一个给定类型的值。

- `void readFully(byte[] b)`

将字节读入到数组 `b` 中，其间阻塞直至所有字节都读入。

参数：`b` 数据读入的缓冲区

- `void readFully(byte[] b, int off, int len)`

将字节读入到数组 `b` 中，其间阻塞直至所有字节都读入。



参数: b 数据读入的缓冲区
 off 数据起始位置的偏移量
 len 读入字节的最大数量

● **String readUTF()**

读入由“修订过的 UTF-8”格式的字符构成的字符串。

● **int skipBytes(int n)**

跳过n个字节,其间阻塞直至所有字节都被跳过。

参数: n 被跳过的字节数

API java.io.DataOutput 1.0

- **void writeBoolean(boolean b)**
写出一个给定类型的值。
- **void writeByte(int b)**
- **void writeChar(int c)**
- **void writeDouble(double d)**
- **void writeFloat(float f)**
- **void writeInt(int i)**
- **void writeLong(long l)**
- **void writeShort(int s)**
- **void writeChars(String s)**
写出字符串中的所有字符。
- **void writeUTF(String s)**
写出由“修订过的 UTF-8”格式的字符构成的字符串。

2.3.2 随机访问文件

RandomAccessFile类可以在文件中的任何位置查找或写入数据。磁盘文件都是随机访问的,但是与网络套接字通信的输入/输出流却不是。你可以打开一个随机访问文件,只用于读入或者同时用于读写,你可以通过使用字符串“r”(用于读入访问)或“rw”(用于读入/写出访问)作为构造器的第二个参数来指定这个选项。

```
RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
RandomAccessFile inOut = new RandomAccessFile("employee.dat", "rw");
```

当你将已有文件作为**RandomAccessFile**打开时,这个文件并不会被删除。

随机访问文件有一个表示下一个将被读入或写出的字节所处位置的文件指针,**seek**方法可以用来将这个文件指针设置到文件中的任意字节位置,**seek**的参数是一个long类型的整数,它的值位于0到文件按照字节来度量的长度之间。

getFilePointer方法将返回文件指针的当前位置。



`RandomAccessFile` 类同时实现了 `DataInput` 和 `DataOutput` 接口。为了读写随机访问文件，可以使用在前面小节中讨论过的诸如 `readInt/writeInt` 和 `readChar/writeChar` 之类的方法。

我们现在要剖析一个将雇员记录存储到随机访问文件中的示例程序，其中每条记录都拥有相同的大小，这样我们可以很容易地读入任何记录。假设你希望将文件指针置于第三条记录处，那么你只需将文件指针置于恰当的字节位置，然后就可以开始读入了。

```
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
Employee e = new Employee();
e.readData(in);
```

如果你希望修改记录，然后将其存回到相同的位置，那么请切记要将文件指针置回到这条记录的开始处：

```
in.seek((n - 1) * RECORD_SIZE);
e.writeData(out);
```

要确定文件中的字节总数，可以使用 `length` 方法，而记录的总数则是用字节总数除以每条记录的大小。

```
long nbytes = in.length(); // length in bytes
int nrecords = (int) (nbytes / RECORD_SIZE);
```

整数和浮点值在二进制格式中都具有固定的尺寸，但是在处理字符串时就有些麻烦了，因此我们提供了两个助手方法来读写具有固定尺寸的字符串。

`writeFixedString` 写出从字符串开头开始的指定数量的码元（如果码元过少，该方法将用 0 值来补齐字符串）。

```
public static void writeFixedString(String s, int size, DataOutput out)
throws IOException
{
    for (int i = 0; i < size; i++)
    {
        char ch = 0;
        if (i < s.length()) ch = s.charAt(i);
        out.writeChar(ch);
    }
}
```

`readFixedString` 方法从输入流中读入字符，直至读入 `size` 个码元，或者直至遇到具有 0 值的字符值，然后跳过输入字段中剩余的 0 值。为了提高效率，这个方法使用了 `StringBuilder` 类来读入字符串。

```
public static String readFixedString(int size, DataInput in)
throws IOException
{
    StringBuilder b = new StringBuilder(size);
    int i = 0;
    boolean more = true;
```

```

while (more && i < size)
{
    char ch = in.readChar();
    i++;
    if (ch == 0) more = false;
    else b.append(ch);
}
in.skipBytes(2 * (size - i));
return b.toString();
}

```

我们将 `writeFixedString` 和 `readFixedString` 方法放到了 `DataIO` 助手类的内部。

为了写出一条固定尺寸的记录，我们直接以二进制方式写出所有的字段：

```

DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
out.writeDouble(e.getSalary());
LocalDate hireDay = e.getHireDay();
out.writeInt(hireDay.getYear());
out.writeInt(hireDay.getMonthValue());
out.writeInt(hireDay.getDayOfMonth());

```

读回数据也很简单：

```

String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
double salary = in.readDouble();
int y = in.readInt();
int m = in.readInt();
int d = in.readInt();

```

让我们来计算每条记录的大小：我们将使用 40 个字符来表示姓名字符串，因此，每条记录包含 100 个字节：

- 40 字符 = 80 字节，用于姓名。
- 1 `double` = 8 字节，用于薪水。
- 3 `int` = 12 字节，用于日期。

程序清单 2-2 中所示的程序将三条记录写到了一个数据文件中，然后以逆序将它们从文件中读回。为了高效地执行，这里需要使用随机访问，因为我们需要首先读入第三条记录。

程序清单 2-2 randomAccess/RandomAccessTest.java

```

1 package randomAccess;
2
3 import java.io.*;
4 import java.util.*;
5 import java.time.*;
6
7 /**
8  * @version 1.13 2016-07-11
9  * @author Cay Horstmann
10 */
11 public class RandomAccessTest
12 {
13     public static void main(String[] args) throws IOException
14     {

```



```

15     Employee[] staff = new Employee[3];
16
17     staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
18     staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
19     staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
20
21     try (DataOutputStream out = new DataOutputStream(new FileOutputStream("employee.dat")))
22     {
23         // save all employee records to the file employee.dat
24         for (Employee e : staff)
25             writeData(out, e);
26     }
27
28     try (RandomAccessFile in = new RandomAccessFile("employee.dat", "r"))
29     {
30         // retrieve all records into a new array
31
32         // compute the array size
33         int n = (int)(in.length() / Employee.RECORD_SIZE);
34         Employee[] newStaff = new Employee[n];
35
36         // read employees in reverse order
37         for (int i = n - 1; i >= 0; i--)
38         {
39             newStaff[i] = new Employee();
40             in.seek(i * Employee.RECORD_SIZE);
41             newStaff[i] = readData(in);
42         }
43
44         // print the newly read employee records
45         for (Employee e : newStaff)
46             System.out.println(e);
47     }
48 }
49
50 /**
51 * Writes employee data to a data output
52 * @param out the data output
53 * @param e the employee
54 */
55 public static void writeData(DataOutput out, Employee e) throws IOException
56 {
57     DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
58     out.writeDouble(e.getSalary());
59
60     LocalDate hireDay = e.getHireDay();
61     out.writeInt(hireDay.getYear());
62     out.writeInt(hireDay.getMonthValue());
63     out.writeInt(hireDay.getDayOfMonth());
64 }
65
66 /**
67 * Reads employee data from a data input
68 * @param in the data input

```

```

69     * @return the employee
70     */
71    public static Employee readData(DataInput in) throws IOException
72    {
73        String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
74        double salary = in.readDouble();
75        int y = in.readInt();
76        int m = in.readInt();
77        int d = in.readInt();
78        return new Employee(name, salary, y, m - 1, d);
79    }
80 }

```

API java.io.RandomAccessFile 1.0

- `RandomAccessFile(String file, String mode)`

- `RandomAccessFile(File file, String mode)`

参数: `file` 要打开的文件

`mode` “`r`” 表示只读模式; “`rw`” 表示读 / 写模式; “`rws`” 表示每次更新时, 都对数据和元数据的写磁盘操作进行同步的读 / 写模式; “`rwd`” 表示每次更新时, 只对数据的写磁盘操作进行同步的读 / 写模式

- `long getFilePointer()`

返回文件指针的当前位置。

- `void seek(long pos)`

将文件指针设置到距文件开头 `pos` 个字节处。

- `long length()`

返回文件按照字节来度量的长度。

2.3.3 ZIP 文档

ZIP 文档 (通常) 以压缩格式存储了一个或多个文件, 每个 ZIP 文档都有一个头, 包含诸如每个文件名字和所使用的压缩方法等信息。在 Java 中, 可以使用 `ZipInputStream` 来读入 ZIP 文档。你可能需要浏览文档中每个单独的项, `getNextEntry` 方法就可以返回一个描述这些项的 `ZipEntry` 类型的对象。向 `ZipInputStream` 的 `getInputStream` 方法传递该项可以获取用于读取该项的输入流。然后调用 `closeEntry` 来读入下一项。下面是典型的通读 ZIP 文件的代码序列:

```

ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
    InputStream in = zin.getInputStream(entry);
    read the contents of in
    zin.closeEntry();
}

```



```

}
zin.close();

```

要写出到 ZIP 文件，可以使用 `ZipOutputStream`，而对于你希望放入到 ZIP 文件中的每一项，都应该创建一个 `ZipEntry` 对象，并将文件名传递给 `ZipEntry` 的构造器，它将设置其他诸如文件日期和解压缩方法等参数。如果需要，你可以覆盖这些设置。然后，你需要调用 `ZipOutputStream` 的 `putNextEntry` 方法来开始写出新文件，并将文件数据发送到 ZIP 输出流中。当完成时，需要调用 `closeEntry`。然后，你需要对所有你希望存储的文件都重复这个过程。下面是代码框架：

```

FileOutputStream fout = new FileOutputStream("test.zip");
ZipOutputStream zout = new ZipOutputStream(fout);
for all files
{
    ZipEntry ze = new ZipEntry(filename);
    zout.putNextEntry(ze);
    send data to zout
    zout.closeEntry();
}
zout.close();

```

注意：JAR 文件（在卷I第13章中讨论过）只是带有一个特殊项的 ZIP 文件，这个项称作清单。你可以使用 `JarInputStream` 和 `JarOutputStream` 类来读写清单项。

ZIP 输入流是一个能够展示流的抽象化的强大之处的实例。当你读入以压缩格式存储的数据时，不必担心边请求边解压数据的问题，而且 ZIP 格式的字节源并非必须是文件，也可以是来自网络连接的 ZIP 数据。事实上，当 Applet 的类加载器读入 JAR 文件时，它就是在读入和解压来自网络的数据。

注意：2.5.8 节将展示如何使用 Java SE7 的 `FileSystem` 类而无需特殊 API 来访问 ZIP 文档。

API java.util.zip.ZipInputStream 1.1

- `ZipInputStream(InputStream in)`

创建一个 `ZipInputStream`，使得我们可以从给定的 `InputStream` 向其中填充数据。

- `ZipEntry getNextEntry()`

为下一项返回 `ZipEntry` 对象，或者在没有更多的项时返回 `null`。

- `void closeEntry()`

关闭这个 ZIP 文件中当前打开的项。之后可以通过使用 `getNextEntry()` 读入下一项。

API java.util.zip.ZipOutputStream 1.1

- `ZipOutputStream(OutputStream out)`

创建一个将压缩数据写出到指定的 `OutputStream` 的 `ZipOutputStream`。

- `void putNextEntry(ZipEntry ze)`

将给定的 `ZipEntry` 中的信息写出到输出流中，并定位用于写出数据的流，然后这些数据可以通过 `write()` 写出到这个输出流中。

- `void closeEntry()`

关闭这个 ZIP 文件中当前打开的项。使用 `putNextEntry` 方法可以开始下一项。

- `void setLevel(int level)`

设置后续的各个 `DEFLATED` 项的默认压缩级别。这里默认值是 `Deflater.DEFAULT_COMPRESSION`。如果级别无效，则抛出 `IllegalArgumentException`。

参数: `level` 压缩级别，从 0(`NO_COMPRESSION`) 到 9(`BEST_COMPRESSION`)

- `void setMethod(int method)`

设置用于这个 `ZipOutputStream` 的默认压缩方法，这个压缩方法会作用于所有没有指定压缩方法的项上。

参数: `method` 压缩方法，`DEFLATED` 或 `STORED`

API `java.util.zip.ZipEntry 1.1`

- `ZipEntry(String name)`

用给定的名字构建一个 Zip 项。

参数: `name` 这一项的名字

- `long getCrc()`

返回用于这个 `ZipEntry` 的 CRC32 校验和的值。

- `String getName()`

返回这一项的名字。

- `long getSize()`

返回这一项未压缩的尺寸，或者在未压缩的尺寸不可知的情况下返回 -1。

- `boolean isDirectory()`

当这一项是目录时返回 `true`。

- `void setMethod(int method)`

参数: `method` 用于这一项的压缩方法，必须是 `DEFLATED` 或 `STORED`

- `void setSize(long size)`

设置这一项的尺寸，只有在压缩方法是 `STORED` 时才是必需的。

参数: `size` 这一项未压缩的尺寸

- `void setCrc(long crc)`

给这一项设置 CRC32 校验和，这个校验和是使用 CRC32 类计算的。只有在压缩方法是 `STORED` 时才是必需的。

参数: `crc` 这一项的校验和

API `java.util.zip.ZipFile 1.1`

- `ZipFile(String name)`

- `ZipFile(File file)`

创建一个 `ZipFile`, 用于从给定的字符串或 `File` 对象中读入数据。

- `Enumeration entries()`

返回一个 `Enumeration` 对象, 它枚举了描述这个 `ZipFile` 中各个项的 `ZipEntry` 对象。

- `ZipEntry getEntry(String name)`

返回给定名字所对应的项, 或者在没有对应项的时候返回 `null`。

参数: `name` 项名

- `InputStream getInputStream(ZipEntry ze)`

返回用于给定项的 `InputStream`。

参数: `ze` 这个 ZIP 文件中的一个 `ZipEntry`

- `String getName()`

返回这个 ZIP 文件的路径。

2.4 对象输入 / 输出流与序列化

当你需要存储相同类型的数据时, 使用固定长度的记录格式是一个不错的选择。但是, 在面向对象程序中创建的对象很少全部都具有相同的类型。例如, 你可能有一个称为 `staff` 的数组, 它名义上是一个 `Employee` 记录数组, 但是实际上却包含诸如 `Manager` 这样的子类实例。

我们当然可以自己设计出一种数据格式来存储这种多态集合, 但是幸运的是, 我们并不需要这么做。Java 语言支持一种称为对象序列化 (object serialization) 的非常通用的机制, 它可以将任何对象写出到输出流中, 并在之后将其读回。(你将在本章稍后看到“序列化”这个术语的出处。)

2.4.1 保存和加载序列化对象

为了保存对象数据, 首先需要打开一个 `ObjectOutputStream` 对象:

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("employee.dat"));
```

现在, 为了保存对象, 可以直接使用 `ObjectOutputStream` 的 `writeObject` 方法, 如下所示:

```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
out.writeObject(harry);
out.writeObject(boss);
```

为了将这些对象读回, 首先需要获得一个 `ObjectInputStream` 对象:

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("employee.dat"));
```

然后，用 `readObject` 方法以这些对象被写出时的顺序获得它们：

```
Employee e1 = (Employee) in.readObject();
Employee e2 = (Employee) in.readObject();
```

但是，对希望在对象输出流中存储或从对象输入流中恢复的所有类都应进行一下修改，这些类必须实现 `Serializable` 接口：

```
class Employee implements Serializable { . . . }
```

`Serializable` 接口没有任何方法，因此你不需要对这些类做任何改动。在这一点上，它与在卷 I 第 6 章中讨论过的 `Cloneable` 接口很相似。但是，为了使类可克隆，你仍旧需要覆盖 `Object` 类中的 `clone` 方法，而为了使类可序列化，你不需要做任何事。

■ 注意：你只有在写出对象时才能用 `writeObject/readObject` 方法，对于基本类型值，你需要使用诸如 `writeInt/readInt` 或 `writeDouble/readDouble` 这样的方法。（对象流类都实现了 `DataInput/DataOutput` 接口。）

在幕后，是 `ObjectOutputStream` 在浏览对象的所有域，并存储它们的内容。例如，当写出一个 `Employee` 对象时，其名字、日期和薪水域都会被写出到输出流中。

但是，有一种重要的情况需要考虑：当一个对象被多个对象共享，作为它们各自状态的一部分时，会发生什么呢？

为了说明这个问题，我们对 `Manager` 类稍微做些修改，假设每个经理都有一个秘书：

```
class Manager extends Employee
{
    private Employee secretary;
    . .
}
```

现在每个 `Manager` 对象都包含一个表示秘书的 `Employee` 对象的引用，当然，两个经理可以共用一个秘书，正如图 2-5 和下面的代码所示的那样：

```
harry = new Employee("Harry Hacker", . . .);
Manager carl = new Manager("Carl Cracker", . . .);
carl.setSecretary(harry);
Manager tony = new Manager("Tony Tester", . . .);
tony.setSecretary(harry);
```

保存这样的对象网络是一种挑战，在这里我们当然不能去保存和恢复秘书对象的内存地址，因为当对象被重新加载时，它可能占据的是与原来完全不同的内存地址。

与此不同的是，每个对象都是用一个序列号（serial number）保存的，这就是这种机制之所以称为对象序列化的原因。下面是其算法：

- 对你遇到的每一个对象引用都关联一个序列号（如图 2-6 所示）。
- 对于每个对象，当第一次遇到时，保存其对象数据到输出流中。
- 如果某个对象之前已经被保存过，那么只写出“与之前保存过的序列号为 x 的对象相同”。

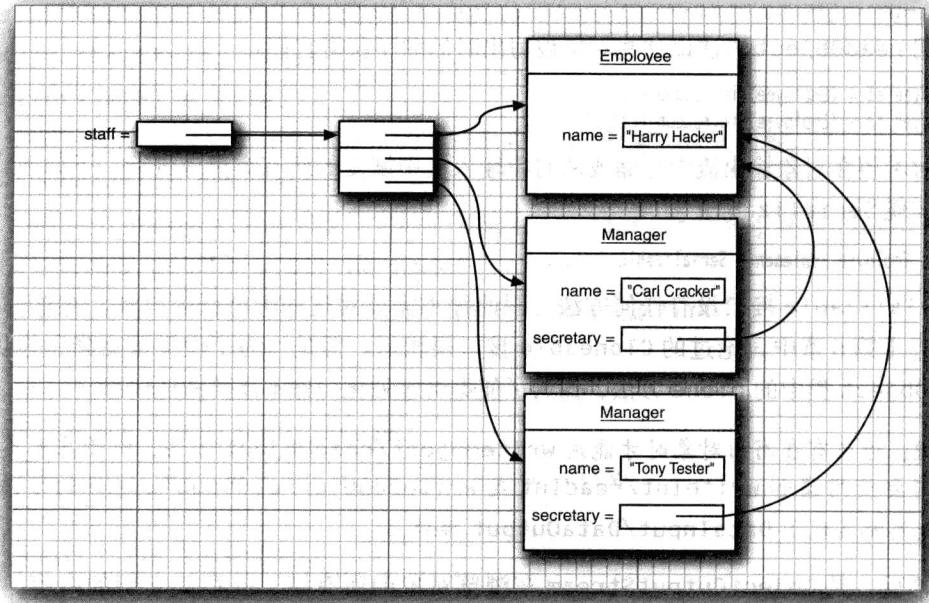


图 2-5 两个经理可以共用一个共有的雇员

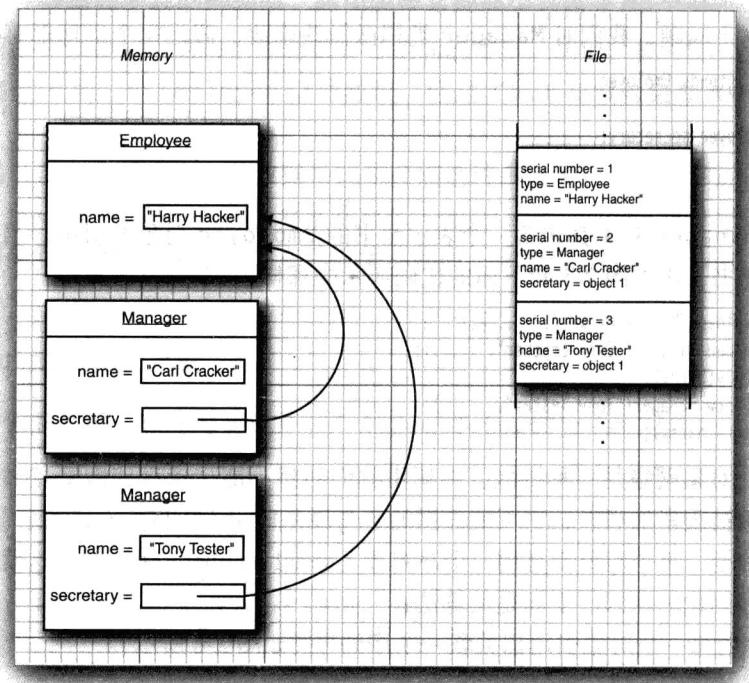


图 2-6 一个对象序列化的实例

在读回对象时，整个过程是反过来的。

- 对于对象输入流中的对象，在第一次遇到其序列号时，构建它，并使用流中数据来初始化它，然后记录这个顺序号和新对象之间的关联。
- 当遇到“与之前保存过的序列号为 x 的对象相同”标记时，获取与这个顺序号相关联的对象引用。

注意：在本章中，我们使用序列化将对象集合保存到磁盘文件中，并按照它们被存储的样子获取它们。序列化的另一种非常重要的应用是通过网络将对象集合传送到另一台计算机上。正如在文件中保存原生的内存地址毫无意义一样，这些地址对于在不同的处理器之间的通信也是毫无意义的。因为序列化用序列号代替了内存地址，所以它允许将对象集合从一台机器传送到另一台机器。

程序清单 2-3 是保存和重新加载 Employee 和 Manager 对象网络的代码（有些对象共享相同的表示秘书的雇员）。注意，秘书对象在重新加载之后是唯一的，当 newStaff[1] 被恢复时，它会反映到经理们的 secretary 域中。

程序清单 2-3 objectStream/ObjectStreamTest.java

```

1 package objectStream;
2
3 import java.io.*;
4
5 /**
6  * @version 1.10 17 Aug 1998
7  * @author Cay Horstmann
8  */
9 class ObjectStreamTest
10 {
11     public static void main(String[] args) throws IOException, ClassNotFoundException
12     {
13         Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
14         Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
15         carl.setSecretary(harry);
16         Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
17         tony.setSecretary(harry);
18
19         Employee[] staff = new Employee[3];
20         staff[0] = carl;
21         staff[1] = harry;
22         staff[2] = tony;
23
24         // save all employee records to the file employee.dat
25         try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("employee.dat")))
26         {
27             out.writeObject(staff);
28         }
29
30         try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("employee.dat")))
31         {

```



```

32  {
33      // retrieve all records into a new array
34
35      Employee[] newStaff = (Employee[]) in.readObject();
36
37      // raise secretary's salary
38      newStaff[1].raiseSalary(10);
39
40      // print the newly read employee records
41      for (Employee e : newStaff)
42          System.out.println(e);
43  }
44 }
45 }
```

API java.io.ObjectOutputStream 1.1

- **ObjectOutputStream(OutputStream out)**

创建一个 `ObjectOutputStream` 使得你可以将对象写出到指定的 `OutputStream`。

- **void writeObject(Object obj)**

写出指定的对象到 `ObjectOutputStream`，这个方法将存储指定对象的类、类的签名以及这个类及其超类中所有非静态和非瞬时的域的值。

API java.io.ObjectInputStream 1.1

- **ObjectInputStream(InputStream in)**

创建一个 `ObjectInputStream` 用于从指定的 `InputStream` 中读回对象信息。

- **Object readObject()**

从 `ObjectInputStream` 中读入一个对象。特别是，这个方法会读回对象的类、类的签名以及这个类及其超类中所有非静态和非瞬时的域的值。它执行的反序列化允许恢复多个对象引用。

2.4.2 理解对象序列化的文件格式

对象序列化是以特殊的文件格式存储对象数据的，当然，你不必了解文件中表示对象的确切字节序列，就可以使用 `writeObject/readObject` 方法。但是，我们发现研究这种数据格式对于洞察对象流化的处理过程非常有益。因为其细节显得有些专业，所以如果你对其实现不感兴趣，则可以跳过这一节。

每个文件都是以下面这两个字节的“魔幻数字”开始的

AC ED

后面紧跟着对象序列化格式的版本号，目前是

00 05

(我们在本节中统一使用十六进制数字来表示字节。) 然后，是它包含的对象序列，其顺序即它们存储的顺序。

字符串对象被存为

74 两字节表示的字符串长度 所有字符

例如，字符串“Harry”被存为

74 00 05 Harry

字符串中的 Unicode 字符被存储为修订过的 UTF-8 格式。

当存储一个对象时，这个对象所属的类也必须存储。这个类的描述包含

- 类名。
- 序列化的版本唯一的 ID，它是数据域类型和方法签名的指纹。
- 描述序列化方法的标志集。
- 对数据域的描述。

指纹是通过对类、超类、接口、域类型和方法签名按照规范方式排序，然后将安全散列算法 (SHA) 应用于这些数据而获得的。

SHA 是一种可以为较大的信息块提供指纹的快速算法，不论最初的数据块尺寸有多大，这种指纹总是 20 个字节的数据包。它是通过在数据上执行一个灵巧的位操作序列而创建的，这个序列在本质上可以百分之百地保证无论这些数据以何种方式发生变化，其指纹也都会跟着变化。(关于 SHA 的更多细节，可以查看一些参考资料，例如 William Stallings 所著的《Cryptography and Network Security: Principles and Practice》第 7 版 [Prentice Hall, 2016]。) 但是，序列化机制只使用了 SHA 码的前 8 个字节作为类的指纹。即便这样，当类的数据域或方法发生变化时，其指纹跟着变化的可能性还是非常大。

在读入一个对象时，会拿其指纹与它所属的类的当前指纹进行比对，如果它们不匹配，那么就说明这个类的定义在该对象被写出之后发生过变化，因此会产生一个异常。在实际情况下，类当然是会演化的，因此对于程序来说，读入较旧版本的对象可能是必需的。我们将在 2.4.5 节中讨论这个问题。

下面表示了类标识符是如何存储的：

- 72
- 2 字节的类名长度
- 类名
- 8 字节长的指纹
- 1 字节长的标志
- 2 字节长的数据域描述符的数量
- 数据域描述符
- 78 (结束标记)
- 超类类型 (如果没有就是 70)



标志字节是由在 `java.io.ObjectStreamConstants` 中定义的 3 位掩码构成的：

```
static final byte SC_WRITE_METHOD = 1;
    // class has a writeObject method that writes additional data
static final byte SC_SERIALIZABLE = 2;
    // class implements the Serializable interface
static final byte SC_EXTERNALIZABLE = 4;
    // class implements the Externalizable interface
```

我们会在本章稍后讨论 `Externalizable` 接口。可外部化的类提供了定制的接管其实例域输出的读写方法。我们要写出的这些类实现了 `Serializable` 接口，并且其标志值为 02，而可序列化的 `java.util.Date` 类定义了它自己的 `readObject/writeObject` 方法，并且其标志值为 03。

每个数据域描述符的格式如下：

- 1 字节长的类型编码
- 2 字节长的域名长度
- 域名
- 类名（如果域是对象）

其中类型编码是下列取值之一：

B byte
C char
D double
F float
I int
J long
L 对象
S short
Z boolean
[数组

当类型编码为 L 时，域名后面紧跟域的类型。类名和域名字符串不是以字符串编码 74 开头的，但域类型是。域类型使用的是与域名稍有不同的编码机制，即本地方法使用的格式。

例如，`Employee` 类的薪水域被编码为：

D 00 06 salary

下面是 `Employee` 类完整的类描述符：

72 00 08 Employee

E6 D2 86 7D AE AC 18 1B 02
00 03
D 00 06 salary
L 00 07 hireDay

指纹和标志
实例域的数量
实例域的类型和名字
实例域的类型和名字



74 00 10 Ljava/util/Date;	实例域的类名——Date
L 00 04 name	实例域的类型和名字
74 00 12 Ljava/lang/String;	实例域的类名——String
78	结束标记
70	无超类

这些描述符相当长，如果在文件中再次需要相同的类描述符，可以使用一种缩写版：

71 4 字节长的序列号

这个序列号将引用到前面已经描述过的类描述符，我们稍后将讨论编号模式。

对象将被存储为：

73 类描述符 对象数据

例如，下面展示的就是 Employee 对象如何存储：

40 E8 6A 00 00 00 00 00	salary 域的值——double
73	hireDate 域的值——新对象
71 00 7E 00 08	已有的类 java/util/Date
77 08 00 00 91 1B 4E B1 80 78	外部存储——稍后讨论细节
74 00 0C Harry Hacker	name 域的值——String

正如你所看见的，数据文件包含了足够的信息来恢复这个 Employee 对象。

数组总是被存储成下面的格式：

75 类描述符 4 字节长的数组项的数量 数组项

在类描述符中的数组类名的格式与本地方法中使用的格式相同（它与在其他的类描述符中的类名稍微有些差异）。在这种格式中，类名以 L 开头，以分号结束。

例如，3 个 Employee 对象构成的数组写出时就像下面一样：

75	数组
72 00 0B [LEmployee;	新类，字符串长度，类名 Employee[]
FC BF 36 11 C5 91 11 C7 02	指纹和标志
00 00	实例域的数量
78	结束标记
70	无超类
00 00 00 03	数组项的数量

注意，Employee 对象数组的指纹与 Employee 类自身的指纹并不相同。

所有对象（包含数组和字符串）和所有的类描述符在存储到输出文件时都被赋予了一个序列号，这个数字以 00 7E 00 00 开头。

我们已经看到过，任何给定的类其完整的类描述符只保存一次，后续的描述符将引用它。例如，在前面的示例中，对 Date 类的重复引用就被编码为：

71 00 7E 00 08



相同的机制还被用于对象。如果要写出一个对之前存储过的对象的引用，那么这个引用也会以完全相同的方式存储，即 71 后面跟随序列号，从上下文中可以很清楚地了解这个特殊的序列引用表示的是类描述符还是对象。

最后，空引用被存储为：

70

下面是前面小节中 `ObjectRefTest` 程序的带注释的输出。如果你喜欢，可以运行这个程序，然后查看其数据文件 `employee.dat` 的十六进制码，并将其与注释列表比较。在输出中接近结束部分的几行重要编码展示了对之前存储过的对象的引用。

AC ED 00 05	文件头
75	数组 <code>staff</code> (序列 #1)
72 00 0B [LEmployee;	新类、字符串长度、类名 <code>Employee[]</code> (序列 #0)
FC BF 36 11 C5 91 11 C7 02	指纹和标志
00 00	实例域的数量
78	结束标记
70	无超类
00 00 00 03	数组项的数量
73	<code>staff[0]</code> ——新对象 (序列 #7)
72 00 07 Manager	新类、字符串长度、类名 (序列 #2)
36 06 AE 13 63 8F 59 B7 02	指纹和标志
00 01	数据的数量
L 00 09 secretary	实例域的类型和名字
74 00 0A LEmployee;	实例域的类名—— <code>String</code> (序列 #3)
78	结束标记
72 00 08 Employee	超类——新类、字符串长度、类名 (序列 #4)
E6 D2 86 7D AE AC 18 1B 02	指纹和标志
00 03	实例域的数量
D 00 06 salary	实例域的类型和名字
L 00 07 hireDay	实例域的类型和名字
74 00 10 Ljava/util/Date;	实例域的类名—— <code>String</code> (序列 #5)
L 00 04 name	实例域的类型和名字
74 00 12 Ljava/lang/String;	实例域的类名—— <code>String</code> (序列 #6)
78	结束标记
70	无超类
40 F3 88 00 00 00 00 00	<code>salary</code> 域的值—— <code>double</code>
73	<code>hireDate</code> 域的值——新对象 (序列 #9)
72 00 0E java.util.Date	新类、字符串长度、类名 (序列 #8)
68 6A 81 01 4B 59 74 19 03	指纹和标志
00 00	无实例变量

78	结束标记
70	无超类
77 08	外部存储、字节的数量
00 00 00 83 E9 39 E0 00	日期
78	结束标记
74 00 0C Carl Cracker	name 域的值——String (序列 #10)
73	secretary 域的值——新对象 (序列 #11)
71 00 7E 00 04	已有的类 (使用序列 #4)
40 E8 6A 00 00 00 00 00	salary 域的值——double
73	hireDate 域的值——新对象 (序列 #12)
71 00 7E 00 08	已有的类 (使用序列 #8)
77 08	外部存储、字节的数量
00 00 00 91 1B 4E B1 80	日期
78	结束标记
74 00 0C Harry Hacker	name 域的值——String (序列 #13)
71 00 7E 00 08	staff[1] ——已有的对象 (使用序列 #11)
73	staff[2] ——新对象 (序列 #14)
71 00 7E 00 02	已有的类 (使用序列 #2)
40 E3 88 00 00 00 00 00	salary 域的值——double
73	hireDay 域的值——新对象 (序列 #15)
71 00 7E 00 08	已有的类 (使用序列 #8)
77 08	外部存储、字节的数量
00 00 00 94 6D 3E EC 00 00	日期
78	结束标记
74 00 0B Tony Tester	name 域的值——String (序列 #16)
71 00 7E 00 0B	secretary 域的值——已有的对象 (使用序列 #11)

当然，研究这些编码大概与阅读常用的电话号码簿一样枯燥。了解确切的文件格式确实不那么重要 (除非你试图通过修改数据来达到不可告人的目的)，但是对象流对其所包含的所有对象都有详细描述，并且这些充足的细节可以用来重构对象和对象数组，因此了解它还是大有益处的。

你应该记住：

- 对象流输出中包含所有对象的类型和数据域。
- 每个对象都被赋予一个序列号。
- 相同对象的重复出现将被存储为对这个对象的序列号的引用。

2.4.3 修改默认的序列化机制

某些数据域是不可以序列化的，例如，只对本地方法有意义的存储文件句柄或窗口句柄的整数值，这种信息在稍后重新加载对象或将其传送到其他机器上时都是没有用处的。事实

上，这种域的值如果不恰当，还会引起本地方法崩溃。Java 拥有一种很简单的机制来防止这种域被序列化，那就是将它们标记成是 `transient` 的。如果这些域属于不可序列化的类，你也需要将它们标记成 `transient` 的。瞬时的域在对象被序列化时总是被跳过的。

序列化机制为单个的类提供了一种方式，去向默认的读写行为添加验证或任何其他想要的行为。可序列化的类可以定义具有下列签名的方法：

```
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
    throws IOException;
```

之后，数据域就再也不会被自动序列化，取而代之的是调用这些方法。

下面是一个典型的示例。在 `java.awt.geom` 包中有大量的类都是不可序列化的，例如 `Point2D.Double`。现在假设你想要序列化一个 `LabeledPoint` 类，它存储了一个 `String` 和一个 `Point2D.Double`。首先，你需要将 `Point2D.Double` 标记成 `transient`，以避免抛出 `NotSerializableException`。

```
public class LabeledPoint implements Serializable
{
    private String label;
    private transient Point2D.Double point;
    ...
}
```

在 `writeObject` 方法中，我们首先通过调用 `defaultWriteObject` 方法写出对象描述符和 `String` 域 `label`，这是 `ObjectOutputStream` 类中的一个特殊的方法，它只能在可序列化类的 `writeObject` 方法中被调用。然后，我们使用标准的 `DataOutput` 调用写出点的坐标。

```
private void writeObject(ObjectOutputStream out)
    throws IOException
{
    out.defaultWriteObject();
    out.writeDouble(point.getX());
    out.writeDouble(point.getY());
}
```

在 `readObject` 方法中，我们反过来执行上述过程：

```
private void readObject(ObjectInputStream in)
    throws IOException
{
    in.defaultReadObject();
    double x = in.readDouble();
    double y = in.readDouble();
    point = new Point2D.Double(x, y);
}
```

另一个例子是 `java.util.Date` 类，它提供了自己的 `readObject` 和 `writeObject` 方法，这些方法将日期写出为从纪元（UTC 时间 1970 年 1 月 1 日 0 点）开始的毫秒数。`Date`

类有一个复杂的内部表示，为了优化查询，它存储了一个 `Calendar` 对象和一个毫秒计数值。`Calendar` 的状态是冗余的，因此并不需要保存。

`readObject` 和 `writeObject` 方法只需要保存和加载它们的数据域，而不需要关心超类数据和任何其他类的信息。

除了让序列化机制来保存和恢复对象数据，类还可以定义它自己的机制。为了做到这一点，这个类必须实现 `Externalizable` 接口，这需要它定义两个方法：

```
public void readExternal(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutputStream out)
    throws IOException;
```

与前面一节描述的 `readObject` 和 `writeObject` 不同，这些方法对包括超类数据在内的整个对象的存储和恢复负责。在写出对象时，序列化机制在输出流中仅仅只是记录该对象所属的类。在读入可外部化的类时，对象输入流将用无参构造器创建一个对象，然后调用 `readExternal` 方法。下面展示了如何为 `Employee` 类实现这些方法：

```
public void readExternal(ObjectInput s)
    throws IOException
{
    name = s.readUTF();
    salary = s.readDouble();
    hireDay = LocalDate.ofEpochDay(s.readLong());
}

public void writeExternal(ObjectOutput s)
    throws IOException
{
    s.writeUTF(name);
    s.writeDouble(salary);
    s.writeLong(hireDay.toEpochDay());
}
```

！ 警告：`readObject` 和 `writeObject` 方法是私有的，并且只能被序列化机制调用。与此不同的是，`readExternal` 和 `writeExternal` 方法是公共的。特别是，`readExternal` 还潜在地允许修改现有对象的状态。

2.4.4 序列化单例和类型安全的枚举

在序列化和反序列化时，如果目标对象是唯一的，那么你必须加倍当心，这通常会在实现单例和类型安全的枚举时发生。

如果你使用 Java 语言的 `enum` 结构，那么你就不必担心序列化，它能够正常工作。但是，假设你在维护遗留代码，其中包含下面这样的枚举类型：

```
public class Orientation
{
    public static final Orientation HORIZONTAL = new Orientation(1);
    public static final Orientation VERTICAL = new Orientation(2);
```



```

private int value;

private Orientation(int v) { value = v; }
}

```

这种风格在枚举被添加到 Java 语言中之前是很普遍的。注意，其构造器是私有的。因此，不可能创建出超出 `Orientation.HORIZONTAL` 和 `Orientation.VERTICAL` 之外的对象。特别是，你可以使用 `==` 操作符来测试对象的等同性：

```
if (orientation == Orientation.HORIZONTAL) . . .
```

当类型安全的枚举实现 `Serializable` 接口时，你必须牢记存在着一种重要的变化，此时，默认的序列化机制是不适用的。假设我们写出一个 `Orientation` 类型的值，并再次将其读回：

```

Orientation original = Orientation.HORIZONTAL;
ObjectOutputStream out = . . .;
out.write(original);
out.close();
ObjectInputStream in = . . .;
Orientation saved = (Orientation) in.read();

```

现在，下面的测试：

```
if (saved == Orientation.HORIZONTAL) . . .
```

将失败。事实上，`saved` 的值是 `Orientation` 类型的一个全新的对象，它与任何预定义的常量都不等同。即使构造器是私有的，序列化机制也可以创建新的对象！

为了解决这个问题，你需要定义另外一种称为 `readResolve` 的特殊序列化方法。如果定义了 `readResolve` 方法，在对象被序列化之后就会调用它。它必须返回一个对象，而该对象之后会成为 `readObject` 的返回值。在上面的情况下，`readResolve` 方法将检查 `value` 域并返回恰当的枚举常量：

```

protected Object readResolve() throws ObjectStreamException
{
    if (value == 1) return Orientation.HORIZONTAL;
    if (value == 2) return Orientation.VERTICAL;
    throw new ObjectStreamException(); // this shouldn't happen
}

```

请记住向遗留代码中所有类型安全的枚举以及向所有支持单例设计模式的类中添加 `readResolve` 方法。

2.4.5 版本管理

如果使用序列化来保存对象，就需要考虑在程序演化时会有什么问题。例如，1.1 版本可以读入旧文件吗？仍旧使用 1.0 版本的用户可以读入新版本产生的文件吗？显然，如果对象文件可以处理类的演化问题，那它正是我们想要的。

乍一看，这好像是不可能的。无论类的定义产生了什么样的变化，它的 SHA 指纹也会



跟着变化，而我们都知道对象输入流将拒绝读入具有不同指纹的对象。但是，类可以表明它对其早期版本保持兼容，要想这样做，就必须首先获得这个类的早期版本的指纹。我们可以使用 JDK 中的单机程序 `serialver` 来获得这个数字，例如，运行下面的命令

```
serialver Employee
```

将会打印出

```
Employee: static final long serialVersionUID = -1814239825517340645L;
```

如果在运行 `serialver` 程序时添加 `-show` 选项，那么这个程序就会产生下面的图形化对话框（参见图 2-7）。

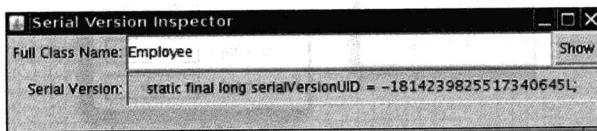


图 2-7 `serialver` 程序的图形化版本

这个类的所有较新的版本都必须把 `serialVersionUID` 常量定义为与最初版本的指纹相同。

```
class Employee implements Serializable // version 1.1
{
    ...
    public static final long serialVersionUID = -1814239825517340645L;
}
```

如果一个类具有名为 `serialVersionUID` 的静态数据成员，它就不再需要人工地计算其指纹，而只需直接使用这个值。

一旦这个静态数据成员被置于某个类的内部，那么序列化系统就可以读入这个类的对象的不同版本。

如果这个类只有方法产生了变化，那么在读入新对象数据时是不会有任何问题的。但是，如果数据域产生了变化，那么就可能会有问题。例如，旧文件对象可能比程序中的对象具有更多或更少的数据域，或者数据域的类型可能有所不同。在这些情况中，对象输入流将尽力将流对象转换成这个类当前的版本。

对象输入流会将这个类当前版本的数据域与被序列化的版本中的数据域进行比较，当然，对象流只会考虑非瞬时和非静态的数据域。如果这两部分数据域之间名字匹配而类型不匹配，那么对象输入流不会尝试将一种类型转换成另一种类型，因为这两个对象不兼容；如果被序列化的对象具有在当前版本中所没有的数据域，那么对象输入流会忽略这些额外的数据；如果当前版本具有在被序列化的对象中所没有的数据域，那么这些新添加的域将被设置成它们的默认值（如果是对象则是 `null`，如果是数字则为 0，如果是 `boolean` 值则是 `false`）。

下面是一个示例：假设我们已经用雇员类的最初版本（1.0）在磁盘上保存了大量的雇员记录，现在我们在 `Employee` 类中添加了称为 `department` 的数据域，从而将其演化到

了 2.0 版本。图 2-8 展示了将 1.0 版的对象读入到使用 2.0 版对象的程序中的情形，可以看到 `department` 域被设置成了 `null`。图 2-9 展示了相反的情况：一个使用 1.0 版对象的程序读入了 2.0 版的对象，可以看到额外的 `department` 域被忽略。

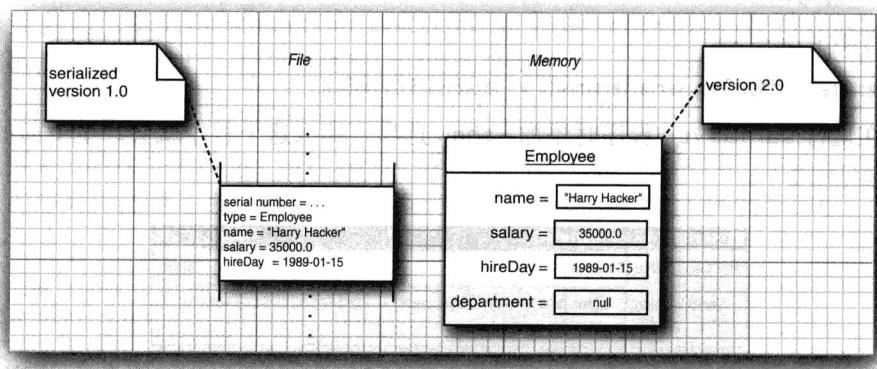


图 2-8 读入具有较少数据域的对象

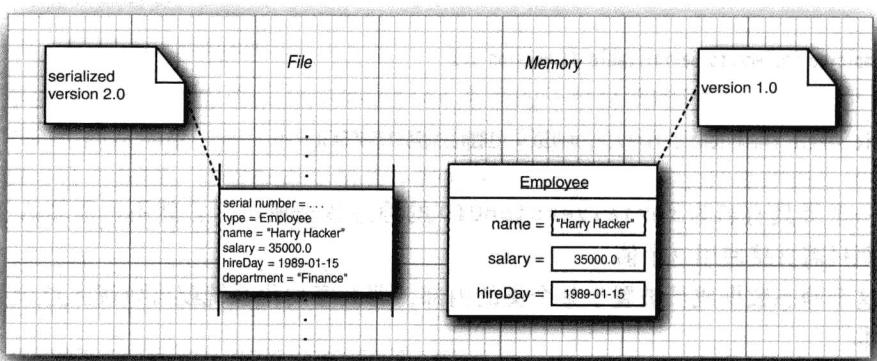


图 2-9 读入具有较多数据域的对象

这种处理是安全的吗？视情况而定。丢掉数据域看起来是无害的，因为接收者仍旧拥有它知道如何处理的所有数据，但是将数据域设置为 `null` 却有可能并不那么安全。许多类都费尽心思地在其所有的构造器中将所有的数据域都初始化为非 `null` 的值，以使得其各个方法都不必去处理 `null` 数据。因此，这个问题取决于类的设计者是否能够在 `readObject` 方法中实现额外的代码去订正版本不兼容问题，或者是否能够确保所有的方法在处理 `null` 数据时都足够健壮。

2.4.6 为克隆使用序列化

序列化机制有一种很有趣的用法：即提供了一种克隆对象的简便途径，只要对应的类是

可序列化的即可。其做法很简单：直接将对象序列化到输出流中，然后将其读回。这样产生的新对象是对现有对象的一个深拷贝（deep copy）。在此过程中，我们不必将对象写出到文件中，因为可以用 `ByteArrayOutputStream` 将数据保存到字节数组中。

正如程序清单 2-4 所示，要想得到 `clone` 方法，只需扩展 `Serializable` 类，这样就完事了。

程序清单 2-4 serialClone/SerialCloneTest.java

```

1 package serialClone;
2
3 /**
4  * @version 1.21 13 Jul 2016
5  * @author Cay Horstmann
6  */
7
8 import java.io.*;
9 import java.util.*;
10 import java.time.*;
11
12 public class SerialCloneTest
13 {
14     public static void main(String[] args) throws CloneNotSupportedException
15     {
16         Employee harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
17         // clone harry
18         Employee harry2 = (Employee) harry.clone();
19
20         // mutate harry
21         harry.raiseSalary(10);
22
23         // now harry and the clone are different
24         System.out.println(harry);
25         System.out.println(harry2);
26     }
27 }
28
29 /**
30 * A class whose clone method uses serialization.
31 */
32 class SerialCloneable implements Cloneable, Serializable
33 {
34     public Object clone() throws CloneNotSupportedException
35     {
36         try {
37             // save the object to a byte array
38             ByteArrayOutputStream bout = new ByteArrayOutputStream();
39             try (ObjectOutputStream out = new ObjectOutputStream(bout))
40             {
41                 out.writeObject(this);
42             }
43
44             // read a clone of the object from the byte array

```



```

45     try (InputStream bin = new ByteArrayInputStream(bout.toByteArray()))
46     {
47         ObjectInputStream in = new ObjectInputStream(bin);
48         return in.readObject();
49     }
50 }
51 catch (IOException | ClassNotFoundException e)
52 {
53     CloneNotSupportedException e2 = new CloneNotSupportedException();
54     e2.initCause(e);
55     throw e2;
56 }
57 }
58 }
59 /**
60 * The familiar Employee class, redefined to extend the
61 * Serializable class.
62 */
63
64 class Employee extends Serializable
65 {
66     private String name;
67     private double salary;
68     private LocalDate hireDay;
69
70     public Employee(String n, double s, int year, int month, int day)
71     {
72         name = n;
73         salary = s;
74         hireDay = LocalDate.of(year, month, day);
75     }
76
77     public String getName()
78     {
79         return name;
80     }
81
82     public double getSalary()
83     {
84         return salary;
85     }
86
87     public LocalDate getHireDay()
88     {
89         return hireDay;
90     }
91
92 /**
93 * Raises the salary of this employee.
94 * @byPercent the percentage of the raise
95 */
96     public void raiseSalary(double byPercent)
97     {
98         double raise = salary * byPercent / 100;

```

```

99     salary += raise;
100 }
101
102 public String toString()
103 {
104     return getClass().getName()
105     + "[name=" + name
106     + ",salary=" + salary
107     + ",hireDay=" + hireDay
108     + "]";
109 }
110 }

```

我们应该当心这个方法，尽管它很灵巧，但是它通常会比显式地构建新对象并复制或克隆数据域的克隆方法慢得多。

2.5 操作文件

你已经学习了如何从文件中读写数据，然而文件管理的内涵远远比读写要广。`Path` 和 `Files` 类封装了在用户机器上处理文件系统所需的所有功能。例如，`Files` 类可以用来移除或重命名文件，或者查询文件最后被修改的时间。换句话说，输入 / 输出流类关心的是文件的内容，而我们在此处要讨论的类关心的是在磁盘上如何存储文件。

`Path` 接口和 `Files` 类是在 Java SE 7 中新添加进来的，它们用起来比自 JDK 1.0 以来就一直使用的 `File` 类要方便得多。我们认为这两个类会在 Java 程序员中流行起来，因此在这里做深度讨论。

2.5.1 Path

`Path` 表示的是一个目录名序列，其后还可以跟着一个文件名。路径中的第一个部件可以是根部件，例如 `/` 或 `C:\`，而允许访问的根部件取决于文件系统。以根部件开始的路径是绝对路径；否则，就是相对路径。例如，我们要分别创建一个绝对路径和一个相对路径；其中，对于绝对路径，我们假设计算机运行的是类 Unix 的文件系统：

```

Path absolute = Paths.get("/home", "harry");
Path relative = Paths.get("myprog", "conf", "user.properties");

```

静态的 `Paths.get` 方法接受一个或多个字符串，并将它们用默认文件系统的路径分隔符（类 Unix 文件系统是 `/`，Windows 是 `\`）连接起来。然后它解析连接起来的结果，如果其表示的不是给定文件系统中的合法路径，那么就抛出 `InvalidPathException` 异常。这个连接起来的结果就是一个 `Path` 对象。

`get` 方法可以获取包含多个部件构成的单个字符串，例如，可以像下面这样从配置文件中读取路径：

