第19章
Introduction to Java Programming, 8E

二进制I/O

学习目标

- 了解在Java中如何处理I/O（19.2节）。
- 区分文本I/O与二进制I/O的不同（19.3节）。
- 使用`FileInputStream`和`FileOutputStream`来读写字节（19.4.1节）。
- 使用基类`FilterInputStream`和`FilterOutputStream`来过滤数据（19.4.2节）。
- 使用`DataInputStream`或`DataOutputStream`来读写基本类型值和字符串（19.4.3节）。
- 使用`ObjectOutputStream`和`ObjectInputStream`实现对象的存储与恢复，理解如何序列化对象以及什么样的对象才可以序列化（19.6节）。
- 实现`Serializable`接口使对象可序列化（19.6.1节）。
- 序列化数组（19.6.2节）。
- 使用`RandomAccessFile`对文件进行读写（19.7节）。

19.1 引言

在文本文件（text file）中存储的数据是以我们能读懂的方式表示的。而在二进制文件（binary file）中存储的数据是用二进制形式表示的。我们是读不懂二进制文件的，因为它们是为让程序来读取而设计的。例如，Java源程序存储在文本文件中，可以使用文本编辑器阅读，但是，Java类存储在二进制文件中，可以被Java虚拟机阅读。二进制文件的优势在于它的处理效率比文本文件高。

尽管从技术上讲不怎么准确和正确，但是可以作这样一个比喻，文本文件是由字符序列构成的，而二进制文件是由位（bit）序列构成的。例如，十进制整数`199`在文本文件中是以三个字符序列`'1'`、`'9'`、`'9'`来存储的，而在二进制文件中它是以`byte`类型的值`C7`存储的，因为十进制数`199`等价的十六进制数是`C7`（$199 = 12 \times 16^1 + 7$）。

Java提供了许多实现文件输入/输出的类。这些类可以分类为文本I/O类（text I/O class）和二进制I/O类（binary I/O class）。在9.7节中已经介绍过使用`Scanner`和`PrintWriter`如何从/向文本文件读/写字符串和数字值。本节介绍实现二进制I/O的类。

19.2 在Java中如何处理输入/输出

回顾一下，`File`对象封装文件或路径属性，但是不包含从/向文件读/写数据的方法。为了进行I/O操作，需要使用正确的Java I/O类创建对象。这些对象包含从/向文件中读/写数据的方法。例如，为了将文本写入一个名为temp.txt的文件中，可以使用`PrintWriter`类按如下方式创建一个对象：

```
PrintWriter output = new PrintWriter("temp.txt");
```

现在，可以调用该对象的`print`方法向文件写入一个字符串。例如，下面的语句将`"Java 101"`写入这个文件中。

```
output.print("Java 101");
```

下面的语句关闭这个文件。

```
output.close();
```

Java有许多用于各种目的的I/O类。通常，可以将它们分类为输入类和输出类。输入类包括读数据的

方法，而输出类包含写数据的方法。PrintWriter是一个输出类的例子，而Scanner是一个输入类的例子。下面的代码为文件temp.txt创建一个输入对象，并从该文件中读取数据：

```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```

如果文件temp.txt中包含"Java 101"，那么input.nextLine()方法就会返回字符串"Java 101"。

图19-1描述了Java I/O程序设计。输入对象从文件中读取数据流，输出对象将数据流写入文件。输入对象也称作输入流（input stream）。同样，输出对象也称作输出流（output stream）。

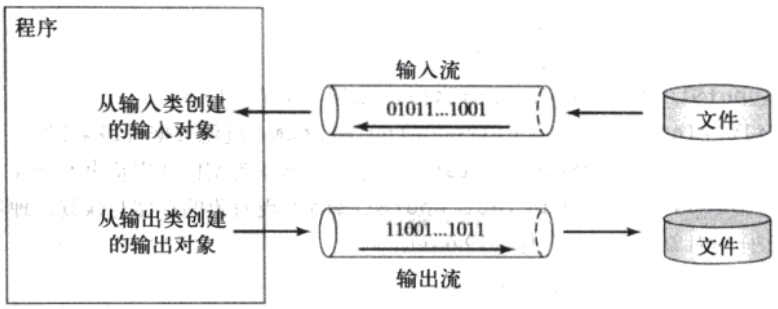


图19-1 程序通过输入对象接收数据，通过输出对象发送数据

19.3 文本I/O与二进制I/O

计算机并不区分二进制文件与文本文件。所有的文件都是以二进制形式来存储的，因此，从本质上说，所有的文件都是二进制文件。文本I/O建立在二进制I/O的基础之上，它能提供字符层次的编码和解码的抽象，如图19-2a所示。在文本I/O中自动进行编码和解码。在写入一个字符时，Java虚拟机会将统一码转化为文件指定的编码，而在读取字符时，将文件指定的编码转化为统一码。例如，假设使用文本I/O将字符串"199"写入文件，那么每个字符都会写入到文件中。由于字符'1'的统一码为0x0031，所以，会根据文件的编码方案将统一码0x0031转化成一个代码。（注意，前缀0x表示十六进制数。）在美国，Windows系统中文本文件的默认编码方案是ASCII码。字符'1'的ASCII码是49（十六进制数是0x31），而字符'9'的就是57（十六进制数是0x39）。所以，为了写入字符"199"，就应该将三个字节0x31、0x39和0x39发送到输出，如图19-2a所示。

注意 新版本的Java支持补充的统一码（supplementary Unicode）。但是，为了简单起见，本书只考虑从0到FFFF的原始统一码。

二进制I/O不需要转化。如果使用二进制I/O向文件写入一个数值，就是将内存中的确切值复制到文件中。例如，一个byte类型的数值199在内存中表示为0xC7（$199 = 12 \times 16^1 + 7$），并且在文件中实际显示的也是0xC7，如图19-2b所示。使用二进制I/O读取一个字节时，就会从输入流中读取一个字节的数值。

一般来说，对于文本编辑器或文本输出程序创建的文件，应该使用文本输入来读取，对于Java二进制输出程序创建的文件，应该使用二进制输入来读取。

由于二进制I/O不需要编码和解码，所以，它比文本I/O效率高。二进制文件与主机的编码方案无关，因此，它是可移植的。在任何机器上的Java程序可以读取Java程序所创建的二进制文件。这就是为什么Java的类文件存储为二进制文件的原因。Java类文件可以在任何具有Java虚拟机的机器上运行。

注意 为了保持一致性，本书使用扩展名.txt来命名文本文件，使用.dat来命名二进制文件。

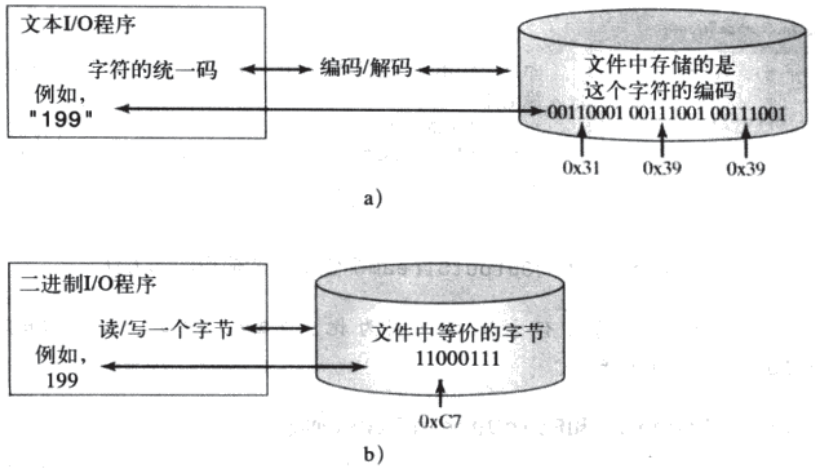


图19-2 文本I/O需要编码和解码，而二进制I/O不需要

19.4 二进制I/O类

Java I/O类的设计是一个很好的应用继承的例子，它们的公共操作是由父类生成的，而子类提供专门的操作。图19-3列出一些实现二进制I/O的类。`InputStream`类是二进制输入类的根类，而`OutputStream`类是二进制输出类的根类。图19-4和图19-5列出了`InputStream`类和`OutputStream`类的所有方法。

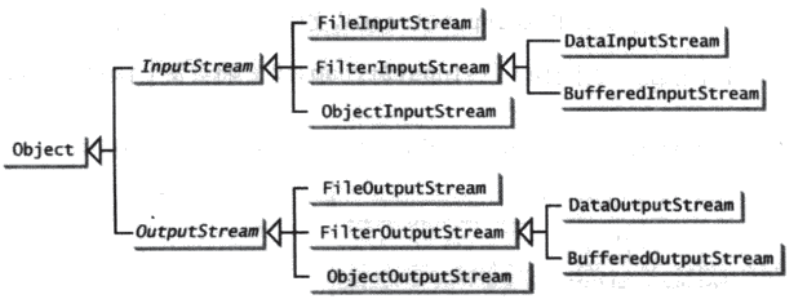


图19-3 用于二进制I/O的`InputStream`类、`OutputStream`类及其子类

java.io.InputStream

```
+read(): int

+read(b: byte[]): int

+read(b: byte[], off: int,
 len: int): int

+available(): int
+close(): void
+skip(n: long): long

+markSupported(): boolean
+mark(readlimit: int): void
+reset(): void
```

从输入流读取数据的下一个字节。返回的字节的值是一个在0到255之间的整数值。如果因为到达这个流的末尾而无字节可用，则返回值−1

从输入流读取大到`b.length`的字节给数组，然后返回读取的实际字节数。在流的末尾返回−1

从输入流读取字节，并将它们存储到`b[off]`、`b[off+1]`、...、`b[off+len−1]`。返回实际读取的字节数。在流的末尾返回−1

返回能从输入流读取的字节数的估计值

关闭这个输入流并释放它所占的系统资源

跳过和丢弃输入流的n字节数据。返回实际跳过的字节数

测试这个输入流是否支持`mark`和`reset`方法

标记这个输入流的当前位置

最后一次调用这个输入流上的`mark`方法时复位这个流

图19-4 抽象的`InputStream`类定义字节输入流的方法

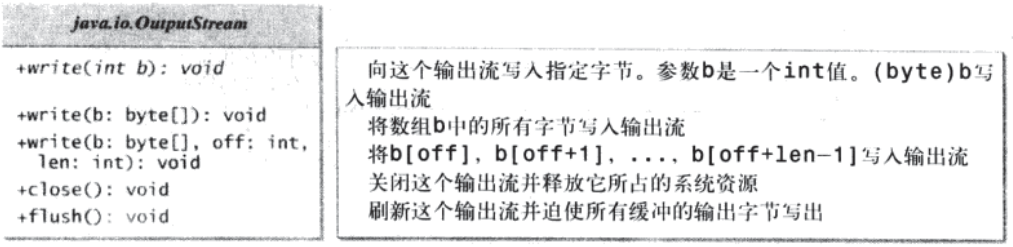


图19-5 抽象的OutputStream类定义字节输出流的方法

注意 二进制I/O类中的所有方法都声明为抛出java.io.IOException或java.io.IOException的子类。

19.4.1 FileInputStream类和FileOutputStream类

FileInputStream类和FileOutputStream类是为了从/向文件读取/写入字节。它们的所有方法都是从InputStream类和OutputStream类继承的。FileInputStream类和FileOutputStream类没有引入新的方法。为了构造一个FileInputStream对象，使用下面的构造方法，如图19-6所示。

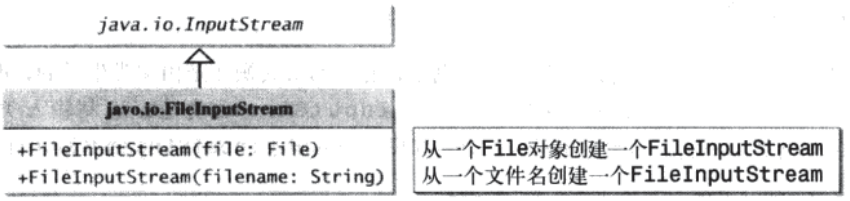


图19-6 FileInputStream从文件输入一个字节流

如果试图为一个不存在的文件创建FileInputStream对象，将会发生java.io.File NotFound-Exception异常。

要构造一个FileOutputStream对象，使用下面的构造方法，如图19-7所示。

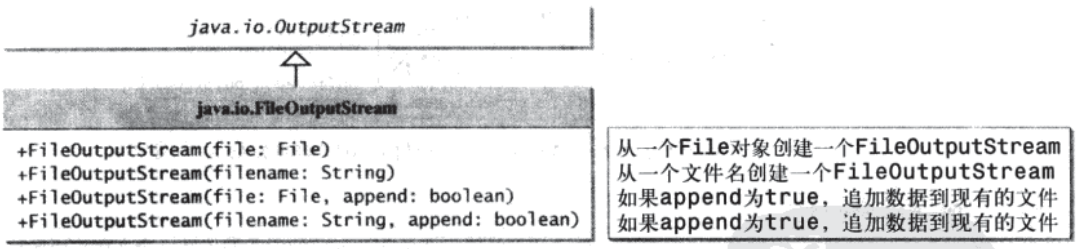


图19-7 FileOutputStream将一个字节流输出到文件中

如果这个文件不存在，就会创建一个新文件。如果这个文件已经存在，前两个构造方法将会删除文件的当前内容。为了既保留文件现有的内容又可以给文件追加新数据，将最后两个构造方法中的参数append设置为true。

几乎所有的I/O类中的方法都会抛出异常java.io.IOException。因此，必须在方法中声明会抛出java.io.IOException异常，或者将代码放到try-catch块中，如下所示：

在方法中声明异常

```
public static void main(String[] args)
    throws IOException {
  // Perform I/O operations
}
```

使用try-catch块

```
public static void main(String[] args) {
  try {
    // Perform I/O operations
  }
  catch (IOException ex) {
    ex.printStackTrace();
  }
}
```

程序清单19-1使用二进制I/O将从1到10的10个字节值写入一个名为temp.dat的文件，再把它们从文件中读出来。

程序清单19-1 TestFileStream.java

```
1 import java.io.*;
2
3 public class TestFileStream {
4   public static void main(String[] args) throws IOException {
5     // Create an output stream to the file
6     FileOutputStream output = new FileOutputStream("temp.dat");
7
8     // Output values to the file
9     for (int i = 1; i <= 10; i++)
10      output.write(i);
11
12    // Close the output stream
13    output.close();
14
15    // Create an input stream for the file
16    FileInputStream input = new FileInputStream("temp.dat");
17
18    // Read values from the file
19    int value;
20    while ((value = input.read()) != -1)
21      System.out.print(value + " ");
22
23    // Close the output stream
24    input.close();
25  }
26 }
```

```
1 2 3 4 5 6 7 8 9 10
```

第6行为文件temp.dat创建了一个FileOutputStream对象。for循环将10个字节值写入文件（第9～10行）。调用write(i)方法与调用write((byte)i)具有相同的功能。第13行关闭输出流。第16行给文件temp.dat创建一个FileInputStream对象。第19～21行从文件读取字节值并在控制台上显示出来。表达式((value = input.read())!=-1)（第20行）从input.read()中读取一个字节，然后将它赋值给value，并且检验它是否为-1。输入值为-1意味着文件的结束。

在这个例子中创建的文件temp.dat是一个二进制文件。可以从Java程序中读取它，但不能用文本编辑器阅读它，如图19-8所示。

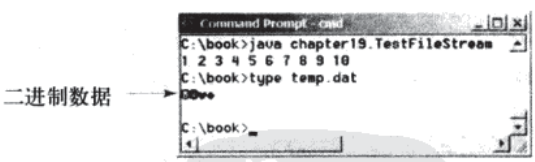


图19-8 二进制文件不能以文本模式显示

提示 当不再需要流时，总是使用close()方法将其关闭。不关闭流可能会在输出文件中造成数据受损，或导致其他的程序设计错误。

注意 这些文件的根目录是类路径的目录。对于本书中的例子，根目录是c:\book。因此，文件temp.dat放在c:\book中。如果希望将temp.dat放在特定的目录下，使用下面的语句替换第8行：

```
FileOutputStream output =
  new FileOutputStream("directory/temp.dat");
```

注意 FileInputStream类的实例可以作为参数去构造一个Scanner对象，而FileOutputStream类的实例可以作为参数构造一个PrinterWriter对象。可以使用

```
new PrintWriter(new FileOutputStream("temp.txt", true));
```

创建一个PrinterWriter对象来向文件中追加文本。

如果temp.txt不存在，就会创建这个文件。如果temp.txt文件已经存在，就将新数据追加到该文件中。

19.4.2 FilterInputStream类和FilterOutputStream类

过滤器数据流（filter stream）是为某种目的过滤字节的数据流。基本字节输入流提供的读取方法read只能用来读取字节。如果要读取整数值、双精度值或字符串，那就需要一个过滤器类来包装字节输入流。使用过滤器类就可以读取整数值、双精度值和字符串，而不是字节或字符。FilterInputStream类和FilterOutputStream类是过滤数据的基类。需要处理基本数值类型时，就使用DataInputStream类和DataOutputStream类来过滤字节。

19.4.3 DataInputStream类和DataOutputStream类

DataInputStream从数据流读取字节，并且将它们转换为正确的基本类型值或字符串。DataOutputStream将基本类型的值或字符串转换为字节，并且将字节输出到数据流。

DataInputStream类扩展FilterInputStream类，并实现DataInput接口，如图19-9所示。DataOutputStream类扩展FilterOutputStream类，并实现DataOutput接口，如图19-10所示。

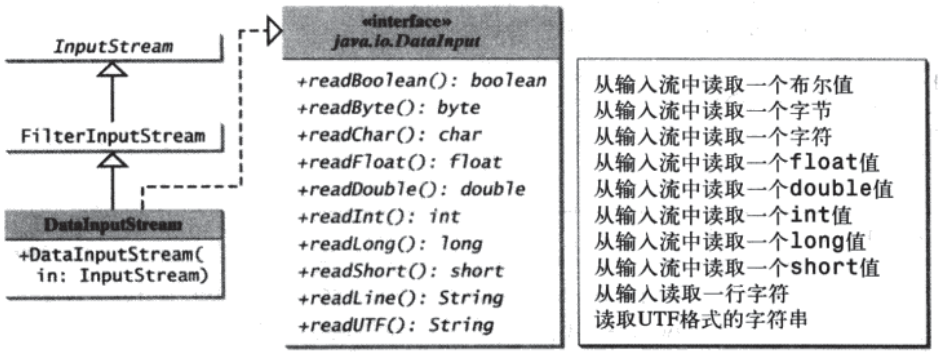


图19-9 DataInputStream过滤字节输入流并将其转化为基本类型值和字符串

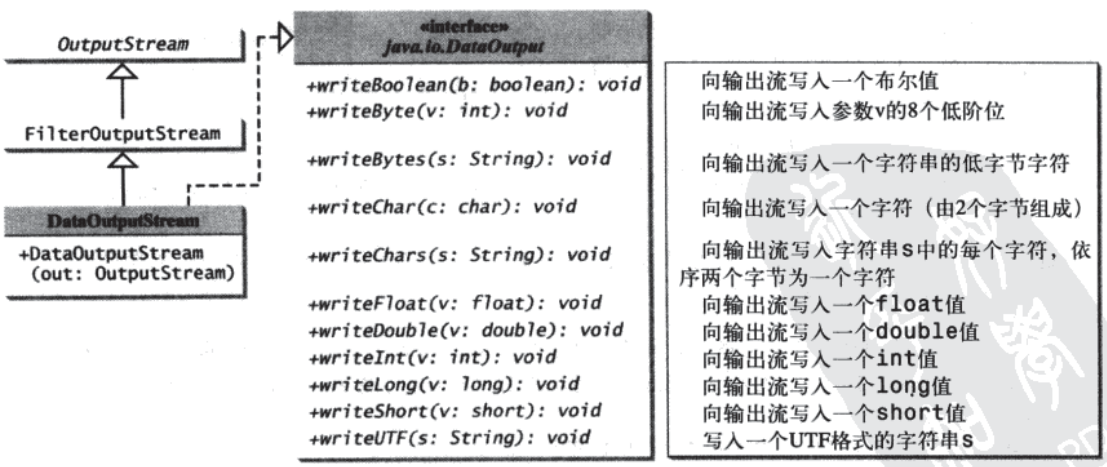


图19-10 DataOutputStream可以将基本数据类型的值和字符串写入输出流

DataInputStream实现了定义在DataInput接口中的方法来读取基本数据类型值和字符串。DataOutputStream实现了定义在DataOutput接口中的方法来写入基本数据类型值和字符串。基本类

型的值不需要做任何转化就可以从内存复制到输出数据流。字符串中的字符可以写成多种形式，这将在下面介绍。

1. 二进制I/O中的字符与字符串

一个统一码由两个字节构成。writerChar(char c)方法将字符c的统一码写入输出流。writerChars(String s)方法将字符串s中所有字符的统一码写到输出流中。writeBytes(String s)方法将字符串s中每个字符统一码的低字节写到输出流。统一码的高字节被丢弃。writeBytes方法适用于由ASCII码字符构成的字符串，因为ASCII码仅存储统一码的低字节。如果一个字符串包含非ASCII码的字符，必须使用writeChars方法实现写入这个字符串。

writeUTF(String s)方法将两个字节长度的信息写入输出流，后面紧跟着的是字符串s中每个字符的改进版UTF-8的形式。UTF-8是一种编码方案，它允许系统和统一码及ASCII码一起操作的编码方案。大多数操作系统使用ASCII码，Java使用统一码。ASCII码字符集是统一码字符集的子集。由于许多应用程序只需要ASCII码字符集，所以将8位的ASCII码转化为16位的统一码是很浪费的。UTF-8的修改版方案分别使用1字节、2字节或3字节来存储字符。如果字符的编码值小于或等于0x7F就将该字符编码为一个字节，如果字符的编码值大于0x7F而小于或等于0x7FF就将该字符编码为两个字节，如果该字符的编码值大于0x7FF就将该字符编码为三个字节。

UTF-8字符起始的几位表明这个字符是存储在一个字节、两个字节还是三个字节中。如果首位是0，那它就是一个字节的字符。如果前三位是110，那它就是两字节序列的第一个字节。如果前四位是1110，那它就是三字节序列的第一个字节。UTF-8字符之前的两个字节用来存储表明字符串中的字符个数的信息。例如，实际上，writeUTF("ABCDEF")写入文件的是8个字节（即00 06 41 42 43 44 45 46），因为头两个字节存储的是字符串中的字符个数。

writeUTF(String s)方法将字符串转化成UTF-8格式的一串字节，然后将它们写入一个二进制数据流。readUTF()方法读取一个使用writeUTF方法写入的字符串。

UTF-8格式具有以一个字节存储ASCII码的优势，因为一个统一码字符的存储需要两个字节，而在UTF-8格式中ASCII字符仅占一个字节。如果一个长字符串的大多数字符都是普通的ASCII字符，采用UTF-8格式存储的效率是很高的。

2. 使用DataInputStream类和DataOutputStream类

数据流用于对已经存在的输入/输出流进行包装，以便在原始流中过滤数据。可以使用下面的构造方法来创建它们（参见图19-9和图19-10）：

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

下面给出的语句会创建数据流。第一条语句为文件in.dat创建一个输入流；而第二条语句为文件out.dat创建一个输出流：

```
DataInputStream input =
  new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream output =
  new DataOutputStream(new FileOutputStream("out.dat"));
```

程序清单19-2将学生的名字和分数写入名为temp.dat的文件中，然后又将数据从这个文件中读出来。

程序清单19-2 TestDataStream.java

```
1  import java.io.*;
2
3  public class TestDataStream {
4    public static void main(String[] args) throws IOException {
5      // Create an output stream for file temp.dat
6      DataOutputStream output =
7        new DataOutputStream(new FileOutputStream("temp.dat"));
8
9      // Write student test scores to the file
10     output.writeUTF("John");
```

```
11      output.writeDouble(85.5);
12      output.writeUTF("Jim");
13      output.writeDouble(185.5);
14      output.writeUTF("George");
15      output.writeDouble(105.25);
16
17      // Close output stream
18      output.close();
19
20      // Create an input stream for file temp.dat
21      DataInputStream input =
22        new DataInputStream(new FileInputStream("temp.dat"));
23
24      // Read student test scores from the file
25      System.out.println(input.readUTF() + " " + input.readDouble());
26      System.out.println(input.readUTF() + " " + input.readDouble());
27      System.out.println(input.readUTF() + " " + input.readDouble());
28    }
29 }
```

```
John 85.5
Jim 185.5
George 105.25
```

第6～7行为文件temp.dat创建一个DataOutputStream对象。第10～15行将学生的名字和分数写入文件中。第18行关闭输出流。第21～22行为同一个文件创建DataInputStream。第25～27行将这个文件中的学生名字和分数读出，并显示在控制台上。

DataInputStream类和DataOutputStream类以同机器平台无关的方式读写Java基本类型值和字符串，因此，如果在一台机器上写好一个数据文件，可以在另一台具有不同操作系统或文件结构的机器上读取该文件。应用程序可以利用数据输出流写入数据，之后某个程序可以利用数据输入流读取这个数据。

警告 应该按存储的顺序和格式读取文件中的数据。例如，学生的姓名是用writeUTF方法以UTF-8格式写入的，所以，读取时必须使用readUTF方法。

3. 检测文件的末尾

如果到达InputStream的末尾之后还继续从中读取数据，就会发生EOFException异常。这个异常可以用来检查是否已经到达文件末尾，如程序清单19-3所示。

程序清单19-3 DetectEndOfFile.java

```
1 import java.io.*;
2
3 public class DetectEndOfFile {
4   public static void main(String[] args) {
5     try {
6       DataOutputStream output = new DataOutputStream
7         (new FileOutputStream("test.dat"));
8       output.writeDouble(4.5);
9       output.writeDouble(43.25);
10      output.writeDouble(3.2);
11      output.close();
12
13      DataInputStream input = new DataInputStream
14        (new FileInputStream("test.dat"));
15      while (true) {
16        System.out.println(input.readDouble());
17      }
18    }
19    catch (EOFException ex) {
20      System.out.println("All data read");
21    }
```

```
22      catch (IOException ex) {
23        ex.printStackTrace();
24      }
25    }
26 }
```

```
4.5
43.25
3.2
All data read
```

程序使用DataOutputStream向文件写入三个双精度值（第6～10行），然后使用DataInputStream读取这些数据（第13～17行）。当读取文件超过了文件末尾，就会抛出一个EOFException异常。该异常在第19行捕获。

19.4.4 BufferedInputStream类和BufferedOutputStream类

BufferedInputStream类和BufferedOutputStream类可以通过减少读写次数来提高输入和输出的速度。BufferedInputStream类和BufferedOutputStream类没有包含新的方法。BufferedInputStream类和BufferedOutputStream中的所有方法都是从InputStream类和OutputStream类继承而来的。BufferedInputStream类和BufferedOutputStream类为存储字节在流中添加一个缓冲区，以提高处理效率。

可以使用如图19-11和19-12所示的构造方法包装在任何一个InputStream类和OutputStream类上的BufferedInputStream类和BufferedOutputStream类。

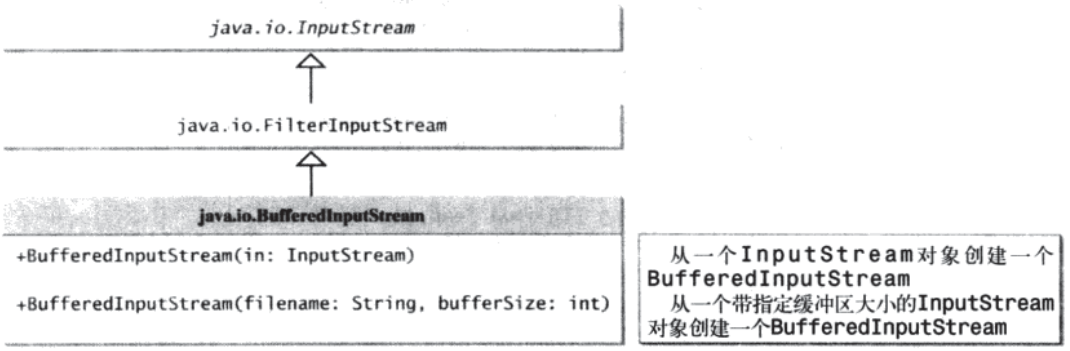


图19-11 BufferedInputStream缓冲输入流

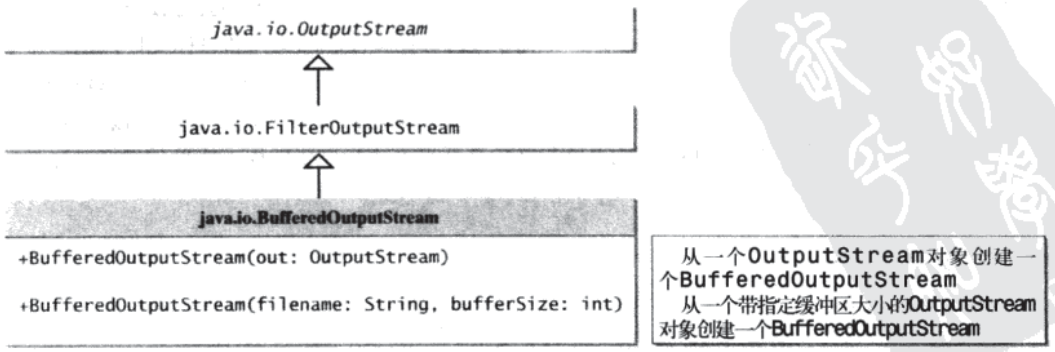


图19-12 BufferedOutputStream缓冲输出流

如果没有指定缓冲区大小，默认的大小是512个字节。缓冲区输入流会在每次读取调用中尽可能多地将数据读入缓冲区。相反地，只有当缓冲区已满或调用flush()方法时，缓冲输出流才会调用写入方法。

通过在第6～7行与第13～14行给数据流添加缓冲区，可以提高前面例子中TestDataStream程序的效率，如下所示：

```
DataOutputStream output = new DataOutputStream(
  new BufferedOutputStream(new FileOutputStream("temp.dat")));

DataInputStream input = new DataInputStream(
  new BufferedInputStream(new FileInputStream("temp.dat")));
```

提示 应该总是使用缓冲区IO来加速输入和输出。对于小文件，我们可能注意不到性能的提升。但是，对于超过100MB的大文件，我们将会看到使用缓冲的IO带来的实质性的性能提升。

19.5 问题：复制文件

本节开发一个复制文件的程序。用户需要提供一个源文件与一个目标文件作为命令行参数，所使用的命令如下：

```
java Copy 源文件名 目标文件名
```

该程序将源文件复制到目标文件，然后显示这个文件中的字节数。如果源文件不存在，就会告知用户找不到这个文件。如果目标文件已经存在，就告知用户目标文件存在。这个程序的一个运行示例如图19-13所示。

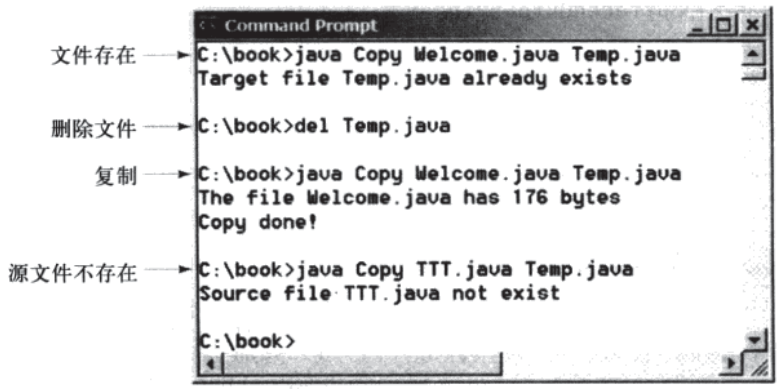


图19-13 程序复制一个文件

要把源文件的内容复制到目标文件，不管文件的内容如何，采用二进制流都会比较合适，使用二进制输入流从源文件读出字节，使用二进制输出流将字节写入目标文件。源文件和目标文件都是从命令行指定的。为源文件创建一个InputFileStream对象，为目标文件创建一个OutputFileStream对象。使用read()方法从输入流中读取一个字节，使用write(b)方法将一个字节写入输出流。使用BufferedInputStream类和BufferedOutputStream类来提高执行效率。程序清单19-4给出这个问题的解决方案。

程序清单19-4 Copy.java

```
1 import java.io.*;
2
3 public class Copy {
4   /** Main method
5      @param args[0] for sourcefile
6      @param args[1] for target file
7    */
8   public static void main(String[] args) throws IOException {
```

```
 9     // Check command-line parameter usage
10     if (args.length != 2) {
11       System.out.println(
12         "Usage: java Copy sourceFile targetfile");
13       System.exit(0);
14     }
15
16     // Check whether source file exists
17     File sourceFile = new File(args[0]);
18     if (!sourceFile.exists()) {
19        System.out.println("Source file " + args[0] + " not exist");
20        System.exit(0);
21     }
22
23     // Check whether target file exists
24     File targetFile = new File(args[1]);
25     if (targetFile.exists()) {
26       System.out.println("Target file " + args[1] + " already
27         exists");
28       System.exit(0);
29     }
30
31     // Create an input stream
32     BufferedInputStream input =
33       new BufferedInputStream(new FileInputStream(sourceFile));
34
35     // Create an output stream
36     BufferedOutputStream output =
37       new BufferedOutputStream(new FileOutputStream(targetFile));
38
39     // Continuously read a byte from input and write it to output
40     int r; int numberOfBytesCopied = 0;
41     while ((r = input.read()) != -1) {
42       output.write((byte)r);
43       numberOfBytesCopied++;
44     }
45
46     // Close streams
47     input.close();
48     output.close();
49
50     // Display the file size
51     System.out.println(numberOfBytesCopied + " bytes copied");
52   }
53 }
```

程序首先在第10～14行检查用户是否在命令行中传递了两个所需的参数。

程序使用File类检查源文件和目标文件是否存在。如果源文件不存在（第18～21行），或者目标文件已经存在，则退出这个程序。

在第32～33行，使用包装在FileInputStream类上的BufferedInputStream类来创建一个输入流，在第36～37行，使用包装在FileOutputStream类上的BufferedOutputStream类来创建一个输出流。

表达式((r = input.read())!=-1)（第41行）从input.read()读取一个字节，将该字节赋值给r，然后检查它是否为-1。输入值-1表示一个文件的结束。程序不断地从输入流读取字节，直到读取完所有的字节，然后将它们写入输出流。

19.6 对象的输入/输出

DataInputStream类和DataOutputStream类可以实现基本数据类型与字符串的输入和输出。而

ObjectInputStream类和ObjectOutputStream类除了可以实现基本数据类型与字符串的输入和输出之外，还可以实现对象的输入和输出。由于ObjectInputStream类和ObjectOutputStream类包含DataInputStream类和DataOutputStream类的所有功能，所以，完全可以用ObjectInputStream类和ObjectOutputStream类代替DataInputStream类和DataOutputStream类。

ObjectInputStream扩展InputStream类，并实现接口ObjectInput和ObjectStreamConstants，如图19-14所示。ObjectInput是DataInput的子接口。DataInput如图19-9所示。ObjectStreamConstants包含支持ObjectInputStream类和ObjectOutputStream类所用的常量。

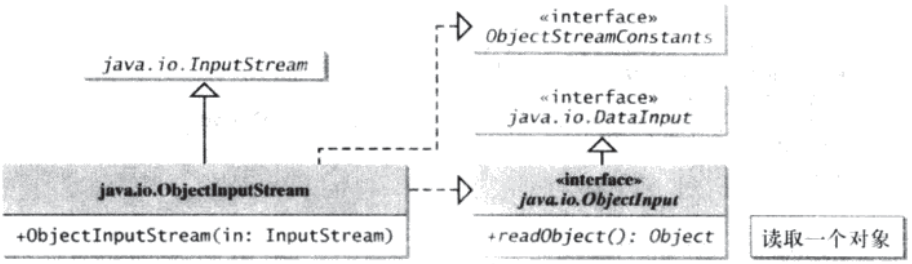


图19-14 ObjectInputStream可以读取对象、基本数据类型值和字符串

ObjectOutputStream扩展OutputStream类，并实现接口ObjectOutput与ObjectStreamConstants，如图19-15所示。ObjectOutput是DataOutput的子接口，DataOutput如图19-10所示。

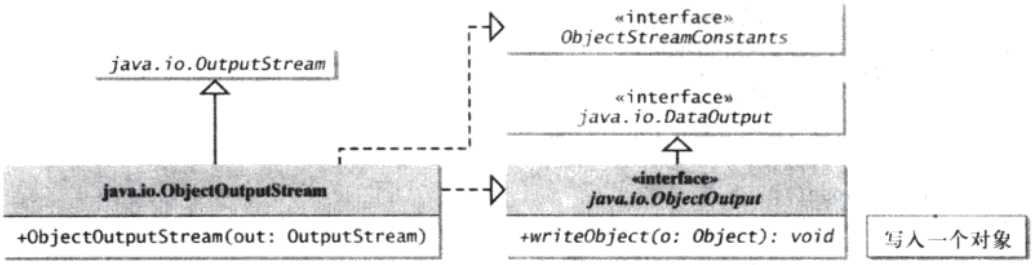


图19-15 ObjectOutputStream可以写对象、基本数据类型值和字符串

可以使用下面的构造方法包装任何一个InputStream和OutputStream上的ObjectInputStream和ObjectOutputStream：

```
// Create an ObjectInputStream
public ObjectInputStream(InputStream in)

// Create an ObjectOutputStream
public ObjectOutputStream(OutputStream out)
```

程序清单19-5将学生的姓名、分数和当前日期写入名为object.dat的文件中。

程序清单19-5 TestObjectOutputStream.java

```
1 import java.io.*;
2
3 public class TestObjectOutputStream {
4   public static void main(String[] args) throws IOException {
5     // Create an output stream for file object.dat
6     ObjectOutputStream output =
7       new ObjectOutputStream(new FileOutputStream("object.dat"));
8
9     // Write a string, double value, and object to the file
```

```
10      output.writeUTF("John");
11      output.writeDouble(85.5);
12      output.writeObject(new java.util.Date());
13
14      // Close output stream
15      output.close();
16    }
17 }
```

在第6～7行，创建一个ObjectOutputStream对象来将数据写入文件object.dat中。在第10～12行，将一个字符串、一个双精度值和一个对象写入这个文件。为了提高程序的性能，可以使用下面的语句替换第6～7行，以完成在数据流中添加一个缓冲区：

```
ObjectOutputStream output = new ObjectOutputStream(
  new BufferedOutputStream(new FileOutputStream("object.dat")));
```

可以向数据流中写入多个对象或基本类型数据。从对应的ObjectInputStream中读回这些对象时，必须与其写入时的类型和顺序相同。为了得到所需的类型，必须使用Java安全的类型转换。程序清单19-6从文件object.dat中读回数据。

程序清单19-6 TestObjectInputStream.java

```
1 import java.io.*;
2
3 public class TestObjectInputStream {
4   public static void main(String[] args)
5     throws ClassNotFoundException, IOException {
6     // Create an input stream for file object.dat
7     ObjectInputStream input =
8       new ObjectInputStream(new FileInputStream("object.dat"));
9
10    // Write a string, double value, and object to the file
11    String name = input.readUTF();
12    double score = input.readDouble();
13    java.util.Date date = (java.util.Date)(input.readObject());
14    System.out.println(name + " " + score + " " + date);
15
16    // Close output stream
17    input.close();
18  }
19 }
```

```
John 85.5 Mon Jun 26 17:17:29 EDT 2006
```

readObject()方法可能会抛出异常java.lang.ClassNotFoundException。这是因为Java虚拟机恢复一个对象时，如果没有加载该对象所在的类，就应该先加载这个类。因为ClassNotFoundException异常是一个必检异常，所以，在第5行main方法中声明要抛出它。第7～8行创建了一个ObjectInputStream对象以从文件object.dat中读取输入。必须以数据写入文件时的顺序和格式从文件中读取这些数据。第11～13行会读出一个字符串、一个双精度值和一个对象。由于readObject()方法返回一个Object对象，所以，在第13行将它转换为Date类型并且赋给一个Date型变量。

19.6.1 可序列化接口Serializable

并不是每一个对象都可以写到输出流。可以写入输出流中的对象称为可序列化的（serializable）。因为可序列化的对象是java.io.Serializable接口的实例，所以，可序列化对象的类必须实现Serializable接口。

Serializable接口是一种标记性接口。因为它没有方法，所以，不需要在类中为实现Serializable接口增加额外的代码。实现这个接口可以启动Java的序列化机制，自动完成存储对象和

数组的过程。

为了体会这个自动功能和理解对象是如何存储的，考虑一下不使用这一功能，储存一个对象需要做哪些工作。假设要存储一个JButton对象，为了完成这个任务，需要存储该对象所有属性（例如，颜色、字体、文本和对齐方式）的当前值。由于JButton是AbstractButton的一个子类，所以，必须把AbstractButton的属性值和AbstractButton所有父类的属性值都存储起来。如果某个属性是对象类型的（例如，background是Color类型的），存储它就要求存储这个对象的所有属性值。如你所见，这是一个非常烦琐冗长的过程。幸运的是，不必手工完成这个过程。Java提供一个内在机制自动完成写对象的过程。这个过程称为对象序列化（object serialization），它是在ObjectOutputStream中实现的。与此相反，读取对象的过程称作对象反序列化（object deserialization），它是在ObjectInputStream类中实现的。

许多Java API中的类都实现了Serializable接口。工具类如java.util.Date以及所有的Swing GUI组件类都实现了Serializable接口。试图存储一个不支持Serializable接口的对象会引起一个NotSerializableException异常。

当存储一个可序列化对象时，会对该对象的类进行编码。编码包括类名、类的签名、对象实例变量的值以及从初始对象引用的任何其他对象的闭包，但是不存储对象静态变量的值。

注意 非序列化的数据域 如果一个对象是Serializable的实例，但它包含的都是非序列化的数据域，那么可以序列化这个对象吗？答案是否定的。为了使该对象是可序列化的，需要给这些数据域加上关键字transient，告诉Java虚拟机将对象写入对象流时忽略这些数据域。考虑下面的类：

```
public class Foo implements java.io.Serializable {
  private int v1;
  private static double v2;
  private transient A v3 = new A();
}

class A { } // A is not serializable
```

当Foo类的一个对象进行序列化时，只需序列化变量v1。因为v2是一个静态变量，所以没有序列化。因为v3标记为transient，所以也没有序列化。如果v3没有标记为transient，将会发生异常java.io.NotSerializableException。

注意 复制对象 如果一个对象不止一次写入对象流，会存储对象的多份副本吗？答案是不会。第一次写入一个对象时，就会为它创建一个序列号。Java虚拟机将对象的所有内容和序列号一起写入对象流。以后每次存储时，如果再写入相同的对象，就只存储序列号。读出这些对象时，它们的引用相同，因为在内存中实际上存储的只是一个对象。

19.6.2 序列化数组

只有数组中的元素都是可序列化的，这个数组才是可序列化的。一个完整的数组可以用writeObject方法存入文件，随后用readObject方法恢复。程序清单19-7存储由五个int元素构成的数组和由三个字符串构成的数组，然后将它们从文件中读取出来显示在控制台上。

程序清单19-7 TestObjectStreamForArray.java

```
1 import java.io.*;
2
3 public class TestObjectStreamForArray {
4   public static void main(String[] args)
5       throws ClassNotFoundException, IOException {
6     int[] numbers = {1, 2, 3, 4, 5};
7     String[] strings = {"John", "Jim", "Jake"};
8
9     // Create an output stream for file array.dat
10    ObjectOutputStream output =
```

```
11        new ObjectOutputStream(new FileOutputStream
12          ("array.dat", true));
13
14      // Write arrays to the object output stream
15      output.writeObject(numbers);
16      output.writeObject(strings);
17
18      // Close the stream
19      output.close();
20
21      // Create an input stream for file array.dat
22      ObjectInputStream input =
23        new ObjectInputStream(new FileInputStream("array.dat"));
24
25      int[] newNumbers = (int[])(input.readObject());
26      String[] newStrings = (String[])(input.readObject());
27
28      // Display arrays
29      for (int i = 0; i < newNumbers.length; i++)
30        System.out.print(newNumbers[i] + " ");
31      System.out.println();
32
33      for (int i = 0; i < newStrings.length; i++)
34        System.out.print(newStrings[i] + " ");
35    }
36 }
```

```
1 2 3 4 5
John Jim Jake
```

第15～16行将两个数组写入文件array.dat中，第25～26行将这两个数组以存入时的顺序从文件中读取出来。由于`readObject()`方法返回`Object`对象，所以应该使用类型转换将其分别转换成`int[]`和`String[]`。

19.7 随机访问文件

到现在为止，所使用的所有数据流都是*只读的*（read.only）或*只写的*（write.only）。这些数据流的外部文件都是*顺序*（sequential）文件，如果不创建新文件就不能更新它们。经常需要修改文件或向文件中插入新记录。Java提供了`RandomAccessFile`类，允许在文件内的随机位置上进行读写。

`RandomAccessFile`类实现了`DataInput`和`DataOutput`接口，如图19-16所示。图19-9中显示的`DataInput`接口定义了读取基本数据类型和字符串的方法（例如，`readInt`、`readDouble`、`readChar`、`readBoolean`和`readUTF`）。图19-10中的`DataOutput`接口定义了输出基本数据类型和字符串的方法（例如，`writeInt`、`writeDouble`、`writeChar`、`writeBoolean`和`writeUTF`）。

当创建一个`RandomAccessFile`数据流时，可以指定两种模式（“`r`”或“`rw`”）之一。模式“`r`”表明这个数据流是只读的，模式“`rw`”表明这个数据流既允许读也允许写。例如，下面的语句创建一个新的数据流`raf`，它允许程序对文件test.dat进行读出和写入：

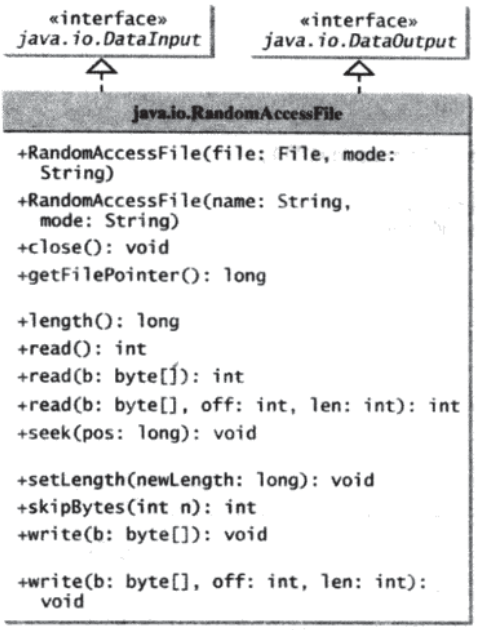
```
RandomAccessFile raf = new RandomAccessFile("test.dat", "rw");
```

如果文件test.dat已经存在，则创建`raf`以便访问这个文件；如果test.dat不存在，则创建一个名为test.dat的新文件，再创建`raf`来访问这个新文件。`raf.length()`方法返回在给定时刻文件test.dat中的字节数。如果向文件中追加新数据，`raf.length()`就会增大。

提示 如果不想改动文件，就将文件以`"r"`模式打开。这样做可以防止不经意中改动文件。

随机访问文件是由字节序列组成的。一个称为*文件指针*（file pointer）的特殊标记定位这些字节中

的某个字节的位置。文件的读写操作就是在文件指针所指的位置上进行的。打开文件时，文件指针置于文件的起始位置。在文件中进行读写数据后，文件指针就会向前移到下一个数据项。例如，如果使用readInt()方法读取一个int数据，Java虚拟机就会从文件指针处读取四个字节，现在，文件指针就会从它之前的位置向前移动四个字节，如图19-17所示。



创建一个带指定File对象和模式的RandomAccessFile流
创建一个带指定文件名、字符串和模式的RandomAccessFile流
关闭流并释放和这个流相关的资源
返回偏移量，以字节为单位，从文件开始位置到下一个read或write发生的地方
返回这个文件的长度
从这个文件读取一个字节的数据并在流的末尾处返回-1
从这个文件读取最大为b.length字节的数据给字节数组
从这个文件读取最大为len字节的数据给字节数组
设置从流的起始位置到下一个read或write发生的位置之间的偏移量（在pos中指定的字节数）
设置这个文件的新长度
跳过输入的n个字节
从当前文件指针开始，将指定字节数组的b.length个字节写入这个文件
向这个文件写入指定字节数组中的len个字节

图19-16 RandomAccessFile类实现DataInput和DataOutput接口，并且增加了支持随机访问的方法

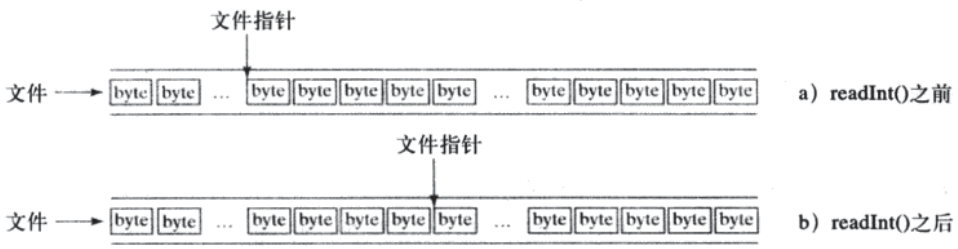


图19-17 读取一个int值之后，文件指针向前移动4个字节

设raf是RandomAccessFile的一个对象，可以调用raf.seek(position)方法将文件指针移到指定的位置。raf.seek(0)方法将文件指针移到文件的起始位置，而raf.seek(raf.length())方法则将文件指针移到文件的末尾。程序清单19-8演示RandomAccessFile类的使用。管理地址簿是一个学习使用RandomAccessFile的大型实例，这个例子在补充材料VII.B中给出。

程序清单19-8 TestRandomAccessFile.java

```
1 import java.io.*;
2
3 public class TestRandomAccessFile {
4   public static void main(String[] args) throws IOException {
5     // Create a random-access file
6     RandomAccessFile inout = new RandomAccessFile("inout.dat", "rw");
7
8     // Clear the file to destroy the old contents, if any
9     inout.setLength(0);
10
11    // Write new integers to the file
```

```
12      for (int i = 0; i < 200; i++)
13        inout.writeInt(i);
14
15      // Display the current length of the file
16      System.out.println("Current file length is " + inout.length());
17
18      // Retrieve the first number
19      inout.seek(0); // Move the file pointer to the beginning
20      System.out.println("The first number is " + inout.readInt());
21
22      // Retrieve the second number
23      inout.seek(1 * 4); // Move the file pointer to the second number
24      System.out.println("The second number is " + inout.readInt());
25
26      // Retrieve the tenth number
27      inout.seek(9 * 4); // Move the file pointer to the tenth number
28      System.out.println("The tenth number is " + inout.readInt());
29
30      // Modify the eleventh number
31      inout.writeInt(555);
32
33      // Append a new number
34      inout.seek(inout.length()); // Move the file pointer to the end
35      inout.writeInt(999);
36
37      // Display the new length
38      System.out.println("The new length is " + inout.length());
39
40      // Retrieve the new eleventh number
41      inout.seek(10 * 4); // Move the file pointer to the next number
42      System.out.println("The eleventh number is " + inout.readInt());
43
44      inout.close();
45    }
46 }
```

```
Current file length is 800
The first number is 0
The second number is 1
The tenth number is 9
The new length is 804
The eleventh number is 555
```

在第6行为名为inout.dat的文件创建了一个模式为“rw”的RandomAccessFile对象，允许进行读取和写入操作。

在第9行，inout.setLength(0)方法将文件长度设置为0。这样做的效果是将文件的原有内容删除。

在第12～13行，for循环将从0到199的200个int值存入文件。由于每个int值占4个字节，所以，inout.length()方法返回文件的现有总长度为800（第16行），如示例输出所示。

在第19行调用inout.seek(0)方法将文件指针设置到文件的起始位置。第20行中inout.readInt()方法读取文件的第一个数值，然后将文件指针移动到指向下一个数值。第23行读取第二个数值。

inout.seek(9*4)方法将文件指针指向第10个数值（第27行）。第28行中的inout.readInt()方法读取文件的第10个数值，然后将文件指针移动到指向文件的第11个数值。inout.write(555)方法在当前位置上写入新的第11个数值（第31行），原来的第11个数据被删除。

inout.seek(inout.length())方法将文件指针指向文件末尾（第34行），inout.writeInt(999)将数值999添加到文件中。现在文件的长度又增加了4，所以，inout.length()方法返回804（第38行）。

在第41行，inout.seek(10*4)方法将文件指针指向第11个数值。第42行显示新的第11个数值为555。

关键术语

binary I/O（二进制输入/输出）
deserialization（反序列化）
file pointer（文件指针）
random-access file（随机访问文件）
sequential-access file（顺序访问文件）
serialization（序列化）
stream（数据流）
text I/O（文本输入/输出）

本章小结

- I/O类可以分为文本I/O和二进制I/O。文本I/O将数据解释成字符序列，二进制I/O将数据解释成原始的二进制数值。文本在文件中如何存储依赖于文件的编码方式。Java自动完成对文本I/O的编码和解码。
- `InputStream`类和`OutputStream`类是所有二进制I/O类的根类。`FileInputStream`类和`FileOutputStream`类用于对文件实现二进制输入和输出。`BufferedInputStream`类和`BufferedOutputStream`类可以包装任何一个二进制输入/输出流以提高其性能。`DataInputStream`类和`DataOutputStream`类可以用来读写基本类型数据和字符串。
- `ObjectInputStream`类和`ObjectOutputStream`类除了可以读写基本类型数据值和字符串，还可以读写对象。为实现对象的可序列化，对象的定义类必须实现`java.io.Serializable`标记性接口。
- `RandomAccessFile`类允许对文件读写数据。可以打开一个模式为“`r`”的文件，这个模式表示文件是只读的，也可以打开一个模式为“`rw`”的文件，这个模式表示文件是可更新的。由于`RandomAccessFile`类实现了`DataInput`和`DataOutput`接口，所以，`RandomAccessFile`中的许多方法都与`DataInputStream`和`DataOutputStream`中的方法一样。

复习题

19.1～19.2节

19.1 什么是文本文件，什么是二进制文件？可以使用文本编辑器查看文本文件或二进制文件吗？

19.2 在Java中如何读写数据？什么是数据流？

19.3节

19.3 文本I/O与二进制I/O的区别是什么？

19.4 在Java中，字符在内存中是如何表示的，在文本文件中是如何表示的？

19.5 如果在一个ASCII码文本文件中写入字符串“`ABC`”，那么在文件中存储的是什么值？

19.6 如果在一个ASCII码文本文件中写入字符串“`100`”，那么在文件中存储的是什么值？如果使用二进制I/O写入字节类型数值`100`，那么文件中存储的又是什么值？

19.7 在Java程序中，表示一个字符使用的编码方案是什么？在默认情况下，Windows中文本文件的编码方案是什么？

19.4节

19.8 在Java IO程序中，为什么必须在方法中声明抛出异常`IOException`或者在try-catch块中处理该异常？

19.9 为什么总是要求关闭数据流？

19.10 `InputStream`可以读取字节。为什么`read()`方法返回`int`值而不是字节？找出`InputStream`和`OutputStream`中的抽象方法。

19.11 `FileInputStream`类和`FileOutputStream`类是否引入了新方法？如何创建`FileInputStream`和`FileOutputStream`对象？

19.12 如果视图为一个不存在的文件创建输入流，会发生什么？如果试图为一个已经存在的文件创建输出

流，会发生什么？能够将数据追加到一个已存在的文件中吗？

19.13 如何使用java.io.PrintWriter向一个已存在的文本文件中追加数据？

19.14 假如一个文件包含了未指定个数的double值，使用DataOutputStream的writeDouble方法将这些值写入文件。如何编写程序读取所有这些值？如何检测是否到达这个文件的末尾？

19.15 在FileOutputStream上使用writeByte(91)方法后，写入文件的是什么？

19.16 在二进制输入流（FileInputStream和DataInputStream）的文件中，如何判断是否已经到达文件末尾？

19.17 下面的代码有什么错误？

```
import java.io.*;

public class Test {
  public static void main(String[] args) {
    try {
      FileInputStream fis = new FileInputStream("test.dat");
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    catch (FileNotFoundException ex) {
      ex.printStackTrace();
    }
  }
}
```

19.18 假设使用默认的ASCII编码方案在Windows上运行程序，在程序结束后，文件t.txt中会有多少个字节？给出每个字节的内容。

```
public class Test {
  public static void main(String[] args)
      throws java.io.IOException {
    java.io.PrintWriter output =
      new java.io.PrintWriter("t.txt");
    output.printf("%s", "1234");
    output.printf("%s", "5678");
    output.close();
  }
}
```

19.19 下面的程序运行完成后，文件t.dat中会有多少个字节？给出每个字节的内容。

```
import java.io.*;

public class Test {
  public static void main(String[] args) throws IOException {
    DataOutputStream output = new DataOutputStream(
      new FileOutputStream("t.dat"));
    output.writeInt(1234);
    output.writeInt(5678);
    output.close();
  }
}
```

19.20 对于下面这些在DataOutputStream对象out上的语句，会有多少个字节发送到输出流？

```
output.writeChar('A');
output.writeChars("BC");
output.writeUTF("DEF");
```

19.21 使用缓冲流有什么好处？下面的语句是否正确？

```
BufferedInputStream input1 =
  new BufferedInputStream(new FileInputStream("t.dat"));

DataInputStream input2 = new DataInputStream(
```

```
    new BufferedInputStream(new FileInputStream("t.dat")));

ObjectInputStream input3 = new ObjectInputStream(
  new BufferedInputStream(new FileInputStream("t.dat")));
```

19.6节

19.22 使用ObjectOutputStream可以存储什么类型的对象？什么方法可以写入对象？什么方法可以读取对象？从ObjectInputStream读取对象的方法的返回类型是什么？

19.23 如果序列化两个同样类型的对象，它们占用的空间相同吗？如果不同，举一个例子。

19.24 是否java.io.Serializable中的任何实例都可以成功地实现序列化？对象的静态变量是否可序列化？如何标记才能避免一个实例变量序列化？

19.25 可以向ObjectOutputStream中写入一个数组吗？

19.26 在任何情况下，DataInputStream和DataOutputStream都可以用ObjectInputStream和ObjectOutputStream替换吗？

19.27 试图运行下面的代码时，会发生什么？

```
import java.io.*;

public class Test {
  public static void main(String[] args) throws IOException {
    ObjectOutputStream output =
      new ObjectOutputStream(new FileOutputStream("object.dat"));

    output.writeObject(new A());
  }
}

class A implements Serializable {
  B b = new B();
}

class B {
}
```

19.7节

19.28 RandomAccessFile流是否可以读写由DataOutputStream创建的数据文件？RandomAccessFile流是否可以读写对象？

19.29 为文件address.dat创建一个RandomAccessFile流，以便更新文件中的学生信息。为文件address.dat创建一个DataOutputStream流。解释这两条语句之间的差别。

19.30 如果文件test.dat不存在，那么试图编译运行下面的代码会出现什么情况？

```
import java.io.*;

public class Test {
  public static void main(String[] args) {
    try {
      RandomAccessFile raf =
        new RandomAccessFile("test.dat", "r");
      int i = raf.readInt();
    }
    catch (IOException ex) {
      System.out.println("IO exception");
    }
  }
}
```

编程练习题

19.3节

*19.1 （创建一个文本文件）编写一个程序，如果文件Exercise19_1.txt不存在，就创建一个名为Exercise19_1.txt的文件。向这个文件追加新数据。使用文本I/O将100个随机生成的整数写入这个文件。文件中的整数用空格分隔。

19.4节

*19.2 （创建二进制数据文件）编写一个程序，如果文件Exercise19_2.dat不存在，就创建一个名为Exercise19_2.txt的文件。向这个文件追加新数据。使用二进制I/O将100个随机生成的整数写入这个文件中。

*19.3 （对二进制数据文件中的所有整数求和）假设已经使用`DataOutputStream`中的`writeInt(int)`方法创建一个名为Exercise19_3.dat的二进制数据文件，文件包含数目不确定的整数，编写一个程序来计算这些整数的总和。

*19.4 （将文本文件转换为UTF格式）编写一个程序，每次从文本文件中读取多行字符，并将这些行字符以UTF-8字符串格式写入一个二进制文件中。显示文本文件和二进制文件的大小。使用下面的命令运行这个程序：

```
java Exercise19_4 Welcome.java Welcome.utf
```

19.6节

*19.5 （将对象和数组存储在文件中）编写一个程序，向一个名为Exercise19_5.dat的文件中存储一个含5个`int`值1，2，3，4，5的数组，存储一个表示当前时间的`Date`对象，存储一个`double`值5.5。

*19.6 （存储`Loan`对象）在程序清单10-2中的类`Loan`没有实现`Serializable`，改写类`Loan`使之实现`Serializable`。编写程序创建5个`Loan`对象，并且将它们存储在一个名为Exercise19_6.dat的文件中。

*19.7 （从文件中读取对象）假设已经用`ObjectOutputStream`创建了一个名为Exercise19_7.dat的文件。这个文件包含`Loan`对象。在程序清单10-2中的`Loan`类没有实现`Serializable`。改写`Loan`类实现`Serializable`。编写程序，从文件中读取`Loan`对象，并且计算总的贷款额。假定文件中`Loan`对象的个数未知。使用`EOFException`来结束这个循环。

19.7节

*19.8 （更新计数器）假设要追踪一个程序的运行次数。可以存储一个`int`值来对文件计数。程序每执行一次，计数器就加1。将程序命名为Exercise19_8，并且将计数器存储在文件Exercise19_8.dat中。

***19.9 （地址簿）补充材料VII.B给出使用随机访问文件来创建和操作一个地址簿的学习实例。通过添加一个如图19-18所示的更新按钮Update来修改这个学习实例，允许用户修改正在显示的地址。

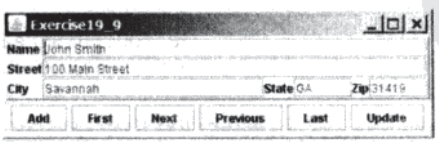


图19-18 这个应用程序可以存储、重新获取以及更新文件的地址

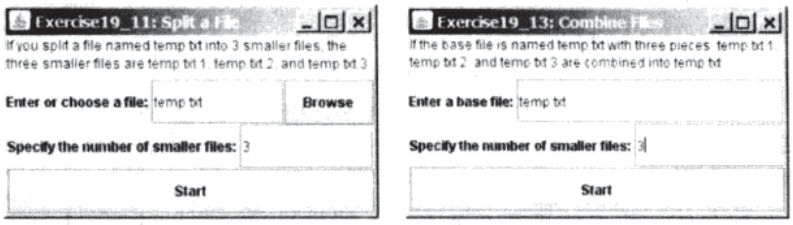
综合题

*19.10 （分割文件）假设希望在CD-R上备份一个大文件（例如，一个10GB的AVI文件）。可以将该文件分割为几个小一些的片段，然后独立备份这些小片段。编写一个工具程序，使用下面的命令将一个大文件分割为小一些的文件：

```
java Exercise19_10 SourceFile numberOfPieces
```

这个命令创建文件SourceFile.1，SourceFile.2，...，SourceFile.n，这里的n是numberOfPieces而输出文件的大小基本相同。

**19.11 （分割文件的GUI）改写练习题19.10使之带有GUI，如图19-19a所示。



a) b)

图19-19 a）程序分割一个文件；b）程序将文件组合成一个新文件

*19.12 （组合文件）编写一个工具程序，使它能够用下面的命令，将文件组合在一起构成一个新文件：

java Exercise19_12 SourceFile1 . . . SoureFilen TargetFile

这个命令将SourceFile1，...，SourceFilen合并为TargetFile。

*19.13 （组合文件的GUI）改写练习题19.12使之带有GUI，如图19-19b所示。

19.14 （加密文件）通过给文件中的每个字节加5来对文件编码。编写一个程序，提示用户输入一个输入文件名和一个输出文件名，然后将输入文件的加密版本存入输出文件。

19.15 （解密文件）假设文件是用练习题19.14中的编码方案加密的。编写一个程序，解码这个加密文件。程序应该提示用户输入一个输入文件名和一个输出文件名，然后将输入文件的解密版本存入输出文件。

19.16 （字符的频率）编写一个程序，提示用户输入一个ASCII文本文件名，然后显示文件中的每个字符出现的频率。

**19.17 （BitOutputStream）实现一个名为BitOutputStream的类，如图19-20所示，将比特写入一个输出流。方法writeBit(char bit)存储一个字节变量形式的比特。创建一个BitOutputStream时，该字节是空的。在调用writeBit('1')之后，这个字节就变成00000001。在调用writeBit("0101")之后，这个字节就变成00010101。前三个字节还没有填充。当字节填满后，就发送到输出流。现在，字节重置为空。必须调用close()方法关闭这个流。如果这个字节非空也非满，close()方法就会先填充0以使字节的8个比特都被填满，然后输出字节并关闭这个流。参见练习题4.46作为提示。编写一个测试程序，将比特0100001001000010011101发送给一个名为Exercise19_17.dat的文件。

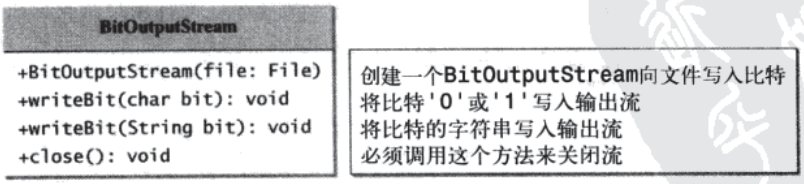


图19-20 BitOutputStream输出比特流给文件

*19.18 （查看比特）编写下面的方法，以显示一个整数的最后一个字节的比特表示：

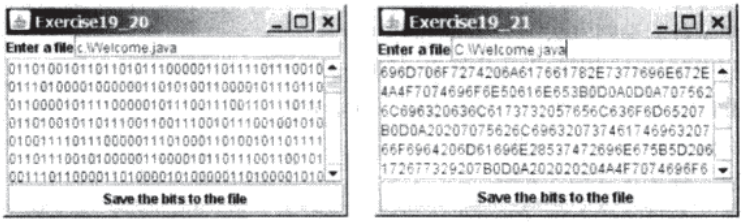
```
public static String getBits(int value)
```

参见练习题4.46作为提示。编写一个程序，提示用户输入一个文件名，从文件读取字节，然后显示每个字节的二进制表示形式。

*19.19 （查看十六进制）编写一个程序，提示用户输入文件名，从文件读取字节，然后显示每个字节的十六进制表示形式。

提示 可以先将字节值转换为一个8比特的字符串，然后再将比特字符串转换为一个两位的十六进制字符串。

**19.20 （二进制编辑器）编写一个GUI应用程序，让用户在文本域输入一个文件名，然后点击回车键，在文本区域显示它的二进制表示形式。用户也可以修改这个二进制代码，然后将它回存到这个文件中，如图19-21a所示。



a) b)

图19-21 该练习题允许用户操作二进制和十六进制形式的文件内容

**19.21 （十六进制编辑器）编写一个GUI应用程序，让用户在文本域输入一个文件名，然后按回车键，在文本域显示它的十六进制表达形式。用户也可以修改十六进制代码，然后将它回存到这个文件中，如图19-21b所示。