

Chp14 I/O 框架

在程序运行的过程当中，JVM 的内存中必然会存放很多数据，包括基本类型和对象类型。但是当程序结束，JVM 关闭的时候，这些数据必然会随之消失。我们可能希望通过某种方式，让这些数据能够保存下来，以备在此使用。因此我们会把数据存入文件，或通过网络发送出去，或存入数据库。反之，我们当然也需要用某种方式，把保存的数据重新读回 JVM。这些，都涉及到 JVM 与外部进行数据交换。

将 JVM 中的数据写出去，我们称为数据的输出。反之，将数据读入 JVM，我们称之为数据的输入。因此，Java 中解决这部分问题的 API 被称为 I/O。（I 是英语 Input 的首字母，表示输入，O 是英语 Output 的首字母，表示输出）

1 Java I/O 概览

1.1 文件系统和 File 类

首先介绍一下 Java 中的 File 类。这个类在 java.io 包中，对于一个 File 对象来说，它能够代表硬盘上的一个文件或者文件夹。在这句描述中，有两个要点值得注意：

- 1、File 对象不仅能够代表一个文件，还能够代表一个文件夹。
- 2、File 对象是“代表”一个文件或者文件夹。怎么来理解“代表”两个字呢？

首先，当我们创建一个 File 对象时，指的是在内存中分配了一块数据区域，也就是说，创建一个 File 对象并不会在系统中真的创建一个文件或者文件夹，而只是在 JVM 的内存中创建了一个对象。通过这个对象能够跟磁盘打交道，从而操作底层的文件。

其次，既然 File 对象是“代表”磁盘中的文件，因此并不要求 File 对象所代表的文件或者文件夹在磁盘中一定存在。也就是说，File 对象所代表的文件或者文件夹可能不存在。

下面我们来结合 JDK 的文档，来看一下 File 类中都有哪些值得注意的方法，以及如何使用 File 类。

首先是 File 类的构造方法。File 对象有四个构造方法，其中三个构造方法比较常用。罗列如下：

`File(String pathname)`：利用字符串作为参数，表示一个路径名。用来创建一个代表给定参数的文件或者文件夹。

`File(String parent, String child)`：parent 表示父目录，child 表示在 parent 目录下的路径。这种写法用来代表名字为 parent/child 的文件或者文件夹。

`File(File parent, String child)`：同样用 parent 表示父目录，只不过这个 parent 是用 File 类型来表示。

需要注意的是，在创建 File 对象的时候，需要指定文件的路径，指定的时候，可以用绝对路径，也可以用相对路径。

另外，要注意路径分隔符的问题。在 Windows 中，路径分隔符使用的是反斜杠“\”，而在 Java 中反斜杠是用来转义的，因此如果要使用反斜杠的话，必须使用“\\”来表示一个反斜杠。

例如，如果要表示 D 盘的 abc 目录，则在 Java 中的字符串应当这样写：

```
"D:\\abc"
```

另外，也可以用一个正斜杠用来做 Windows 中的路径分隔，这样就不需要转义。因此，也可以使用下面的表示方式：

"D:/abc"

这两种路径的写法都是正确的。

其次是 `File` 类中的一些基本操作。

`createNewFile()`：这个方法可以用来创建一个新文件。需要注意的是，如果这个文件在系统中已经存在，`createNewFile` 方法不会覆盖原有文件。

`mkdir()` / `makedirs()`：这两个方法都可以用来创建文件夹。所不同的是，`mkdir` 只能创建一层文件夹，而 `makedirs` 能够创建多层文件夹。

例如，如果当前目录下已经存在一个 `abc` 目录，则下面的代码能够成功创建一个目录：

```
File dir = new File("abc/def");
```

```
dir.mkdir();
```

由于已经有了 `abc` 目录，所以需要创建的仅仅是一层 `def` 目录而已。此时可以调用 `mkdir()` 方法来创建目录。

而如果当前目录下不存在 `abc` 目录，创建时需要先创建一个 `abc` 目录，然后创建 `def` 目录，属于创建两层目录。因此，这就需要调用 `makedirs()` 方法来创建多层文件夹。

`delete()`：这个方法能够删除 `File` 所代表的文件或者文件夹。

`deleteOnExit()`：这个方法也能用来删除文件或者文件夹。所不同的是，`delete()` 方法被调用时，这个文件或者文件夹会被立刻删除，而 `deleteOnExit()` 方法被调用后，文件或者文件夹并不会立刻被删除，而会等到程序退出以后再删除。

`getPath()`：返回路径

`getName()`：返回文件名

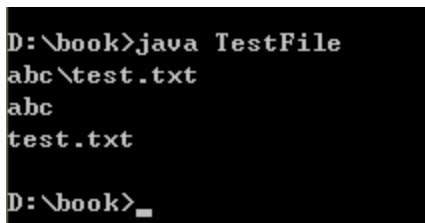
`getParent()`：返回所在的文件夹

对于上面这三个方法，我们用一段代码来说明。我们创建一个 `File` 对象，表示当前目录的 `abc` 目录下的 `test.txt` 文件，并分别调用上述三个方法。代码如下：

```
import java.io.*;

public class TestFile{
    public static void main(String args[]){
        File f = new File("abc/test.txt");
        System.out.println(f.getPath());
        System.out.println(f.getParent());
        System.out.println(f.getName());
    }
}
```

输出结果如下：



```
D:\book>java TestFile
abc\test.txt
abc
test.txt
D:\book>
```

可以看出，`getPath()` 返回的是我们给出的路径，`getParent()` 返回的所在的父文件夹，而 `getName()` 返回的是文件名。

`getAbsolutePath()`：返回绝对路径。

`getCanonicalPath()`：返回规格化以后的路径。需要注意的是这个方法会抛出一个异常。

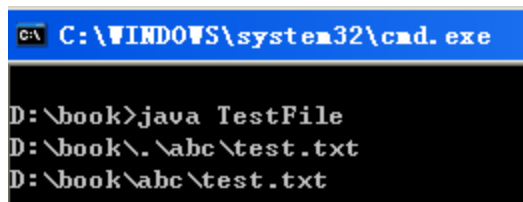
我们同样可以使用一个程序来说明这两个方法。代码如下：

```
import java.io.*;

public class TestFile{

    public static void main(String args[]) throws Exception{
        File f = new File("./abc/test.txt");
        System.out.println(f.getAbsolutePath());
        System.out.println(f.getCanonicalPath());
    }
}
```

上述代码运行结果如下：



```
C:\WINDOWS\system32\cmd.exe

D:\book>java TestFile
D:\book\.\abc\test.txt
D:\book\abc\test.txt
```

`getAbsolutePath` 和 `getCanonicalPath` 都会返回绝对路径，只不过，如果在路径中存在“.”以及“..”这两个符号的话，则 `getCanonicalPath` 会修正路径，而不让返回值中存在“.”和“..”这两个符号。

下面是一些判断：

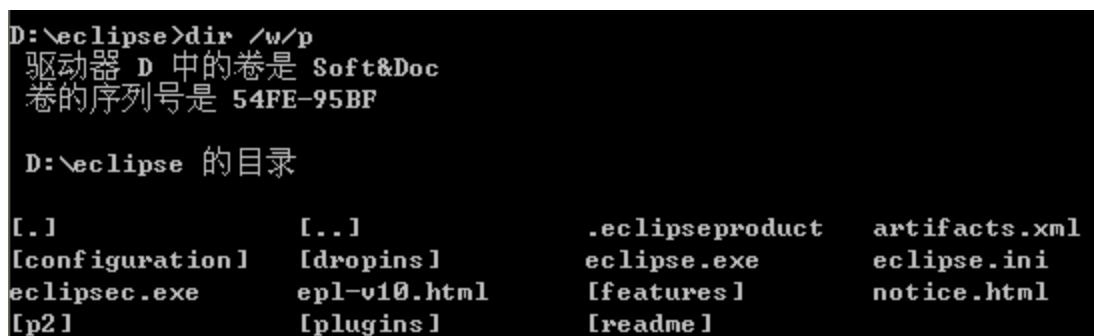
`exists()`：这个方法用来判断 `File` 对象表示的文件或者文件夹是否存在

`isFile()`：判断 `File` 对象是否代表的是一个文件

`isDirectory()`：判断 `File` 对象是否代表的是一个文件夹

最后，是一个文件夹特有的操作：

`listFiles()`：这个方法能够返回一个 `File` 数组，这个数组表示 `File` 对象所代表文件夹下所有的内容。例如，如果 `D:\eclipse` 目录下的内容如下图：



```
D:\eclipse>dir /w/p
驱动器 D 中的卷是 Soft&Doc
卷的序列号是 54FE-95BF

D:\eclipse 的目录

[.]          [...]          .eclipseproduct  artifacts.xml
[configuration] [dropins]      eclipse.exe      eclipse.ini
eclipse.exe   epl-v10.html   [features]      notice.html
[p2]          [plugins]      [readme]
```

我们可以创建一个表示这个目录的 `File` 对象，并对其调用 `listFiles` 方法，代码如下：

```
import java.io.*;

public class TestListFiles{

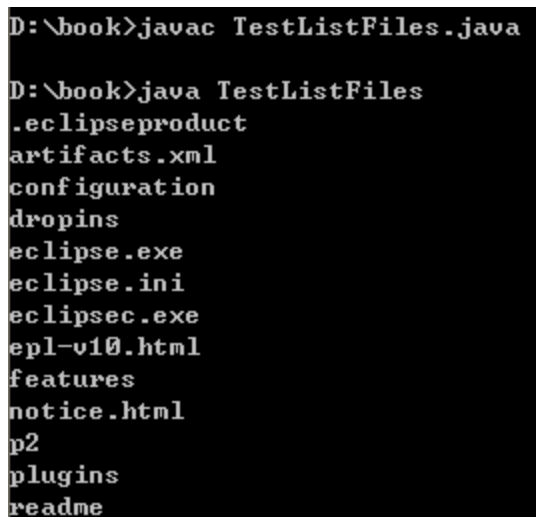
    public static void main(String args[]){
```

```

        File f = new File("D:/eclipse");
        File[] fs = f.listFiles();
        for(int i = 0; i<fs.length; i++){
            System.out.println(fs[i].getName());
        }
    }
}

```

结果如下：



```

D:\book>javac TestListFiles.java

D:\book>java TestListFiles
.eclipseproduct
artifacts.xml
configuration
dropins
eclipse.exe
eclipse.ini
eclipseec.exe
epl-v10.html
features
notice.html
p2
plugins
readme

```

上面就是 File 对象的使用。请在继续下一部分的学习之前，巩固一下 File 对象的一些基本操作。

1.2 I/O 分类

对于 Java 来说，进行 I/O 操作需要使用“流”对象。所谓的流，指的是：用来传输数据的对象。例如，在生活中，电线就是一种流，这个对象用来传输电力；水管是一种流，这个对象是用来传输水；输油管也是流，用来传输石油，等等。

对于 Java 中的流来说，有三种分类的方式：按照流的方向分，按照流的数据单位分，按照流的功能来分。

首先，流可以按照方向分类，分为输入流和输出流。这里，需要解决的问题是，哪个方向是输入流，哪个方向是输出流。例如，如果从硬盘上读取一个文件，算输入还是输出呢？

所谓输入、输出的方向，都是相对于 JVM 而言的。所谓读取文件，指的是从硬盘中的文件里读取数据，然后这些数据就会传入 JVM 中。这个过程，就是数据从虚拟机的外部“进入”JVM 的过程，这就是“输入”。而写文件，就是把 JVM 中的数据保存到文件中，是数据从 JVM “输出”到文件中，这就是“输出”。

从生活上来说，流也都是有方向的。我们的楼房中，有上水管和下水管，上水管就如同是输入流，而下水管就如同是输出流。

其次，流可以按照数据单位分类，分为字节流和字符流。顾名思义，字节流传输的单位是字节，而字符流传输的单位是字符。

首先，对于任何系统中的所有文件来说，底层都是 0 和 1 组成的，这就是二进制的位(bit)的概念。而 8 个 bit 组成一个字节，这也就是计算机中处理数据的最小单位。换言之，由

于任何数据都是 bit 组成的，而我们可以每次都传输 8 个 bit 形成一个字节，也就是说，任何数据都可以按照字节的方式进行传输。

因此，字节流可以用来传输任何一种文件类型，包括 mp3、电影、图片、网页、文本……等等。

而系统中，有一种文件比较特殊：文本文件。这种文件大量存在于系统中，例如源代码、html 源码、xml 配置文件，等等。我们在进行 I/O 的时候可能会频繁跟文本文件打交道。字节流同样可以处理文本文件，但是会有一些小问题。例如，大部分中文的文本，一个汉字可能占用的空间不止一个字节。假设一个汉字需要占用两个字节的空間，如果要用字节流处理文本的话，就需要读入两个字节，然后再把这两个字节拼成一个完整的汉字。更有可能在传输错误的时候，产生只保存了“半个汉字”这种问题。为了解决这种问题，我们提供了字符流。

字符流传输数据的单位是字符。这种流专门用于处理文本，能够方便的处理字符编码的问题。关于字符编码的问题，在后面关于字符流的介绍中再进行详细阐述。

最后，流可以按照功能分类，分为节点流和过滤流。什么是节点流呢？这指的是：真正能够完成传输功能的流。而相对的，过滤流并不能完成真正的数据传输，过滤流是用来为其他流增强功能。

如何来理解呢？在输电线中，真正能够传输电力的，是输电线中的金属丝，这就相当于节点流。而输电线往往会包一层绝缘的胶布，这层胶布并不能用来传输电力，而是为节点流增强功能（增加了绝缘保护的功能），这一层绝缘胶布就被称之为过滤流。

在节点流和过滤流的设计上，I/O 框架中使用了一种设计模式，这种设计模式被称为“装饰模式”。

1.3 初识装饰模式

首先，我们简单介绍一下设计模式。所谓的设计模式，我们可以把它理解为设计的“套路”。在面向对象编程的发展过程中，有一些程序员总结出了在设计中有可能经常要遇到的一些问题，为了解决这些问题，在业界有一些套路，这些套路就是所谓的设计模式。在学习设计模式的时候，学习模式的编码和实现固然重要，但是不能忽略这种模式用来解决什么问题。

下面我们就来提出一个问题。

假设我们要设计一种第一人称设计游戏。这种设计游戏，当角色刚刚产生时，手里的枪是一把默认的枪，一把单发的威力较小的枪。

之后，在游戏过程中，有可能遇到一些字母。例如，如果遇到英文字母 S，此时，手里的枪就会变成一把散弹枪，一按扳机会同时发出 5 发子弹；如果遇到英文字母 L，手里的枪则会变成一把激光枪；如果遇到英文字母 F，手里的枪就会变成一把火焰枪。等等

当然，作为第一人称射击游戏，我们的设计和经典游戏魂斗罗还是有区别的。例如，如果先遇到 S 后遇到 L，则此时，我们手中的枪会变成一把散弹激光枪，也就是说，扣动扳机之后会有 5 道激光同时从枪中发出。

此外，除了基本的英文字母之外，还可能遇到一些其他的道具，例如，可能可以捡到瞄准镜，或者消声器等道具。

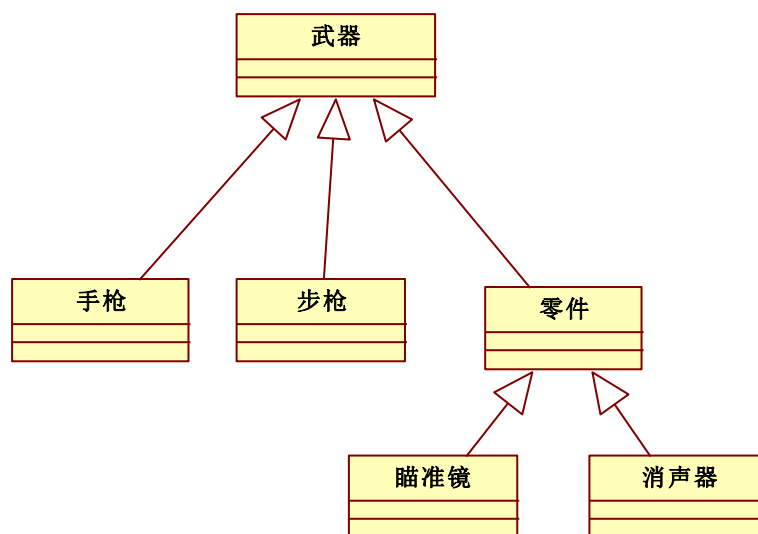
下面我们就要为这种游戏来设计枪械类。如果按照一般的设计的话，则对于每一种不同的枪都要设计一个特定的类。

例如：散弹枪、激光枪、火焰枪；散弹激光枪、散弹火焰枪、激光火焰枪；散弹激光火焰枪。还得加上道具：带瞄准镜的火焰枪，带瞄准镜的激光枪，带瞄准镜的散弹枪（或许，

散弹枪不需要瞄准镜？)；带瞄准镜的散弹火焰枪……；带消声器的散弹枪，带消声器的火焰枪，带消声器的……

目前仅仅是两个零件而已。如果随着游戏的更改，又增加了新的道具，比如枪榴弹、刺刀等等，我们就需要写带刺刀的散弹火焰枪、带枪榴弹的散弹激光枪……等等大量的类。每当增加一种新道具，我们所需要写的类都是成几何级数增长的！

怎么解决这个问题呢？为此，我们引入了装饰模式，采用下面的类关系来设计整个武器的体系：



我们可以看到，我们写一个父类武器，是所有武器的父类。此外，这个父类有一些子类，例如手枪、步枪等等。而在子类中，有一个子类是零件，零件的子类就包括标准镜、消声器等等。

从类的设计上来说，除了武器类之外，手枪、步枪等类都是真正完成射击功能的类，也就是真正成功能的类。在 I/O 框架中对应的概念就是节点流。而零件的那些子类，是为枪械增强功能的，本身并不能完成射击的功能，因此对应于 I/O 的概念，就是过滤流。

2 字节流

介绍完流的分类之后，我们首先来介绍一下字节流。字节流的特点是传输的数据单位是字节，也意味着字节流能够处理任何一种文件。

2.1 InputStream/OutputStream 的基本操作

首先，介绍一下所有字节流的父类，也就是在装饰模式中扮演“武器”这个角色的类。所有输入字节流的父类是 `InputStream`，所有输出字节流的父类是 `OutputStream`，他们都处于 `java.io` 包下。要学习这两个类，就需要创建这两个类的对象。但是，查阅 JDK 文档，我们可以知道这两个类都是抽象类，无法创建对象。因此，为了学习这两个类的用法，我们需要先获得这两个类的某个子类。

我们将使用 `FileInputStream` 和 `FileOutputStream` 这两个类来学习 `InputStream` 和 `OutputStream`。`FileInputStream` 是文件输入流，从功能上说，这是一个节点流，能够读取硬

盘上的文件；而 `FileOutputStream` 是文件输出流，能够写入文件。

下面，我们就以 `FileInputStream` / `FileOutputStream` 为例，来学习一下字节流。

2.1.1 FileInputStream

首先我们研究一下 `FileInputStream` 这个类。这个类由若干个构造方法，其中两个比较常用，罗列如下：

`FileInputStream(String filename)`：通过文件路径，获得文件输入流

`FileInputStream(File file)`：通过文件对象，获得文件输入流。

需要注意的是，`FileInputStream` 在创建对象的时候，当要读取的文件不存在时，就会抛出异常：`FileNotFoundException`。由于这是一个已检查异常，因此必须要处理。

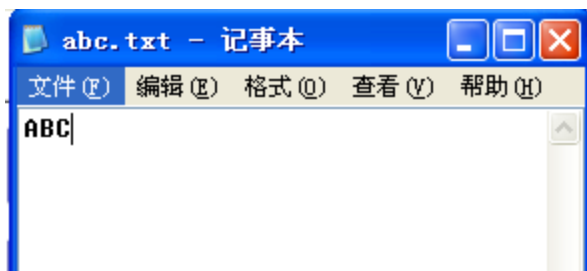
除此之外，`FileInputStream` 还有一些值得注意的方法：

`close()`：这个方法顾名思义，可以用来关闭文件流，释放资源。当我们结束输入操作时，应该调用该方法来关闭流。

`int read()`：无参的 `read` 方法。这个方法每次都从文件中读取一个字节，并且把读到的内容返回。要强调的是，虽然返回值是一个 `int` 类型，但是每次读文件时只读取一个字节，这个字节作为 `int` 四个字节中的最低位返回。

而当读到流末尾时，返回-1。

我们可以结合代码来理解 `read` 方法。假设我们在当前目录下准备一个文件 `abc.txt`，这个文件中保存了三个英文字母：ABC。如下所示：



保存这个文件，本质上是保存了三个字节，这三个字节分别代表了三个英文字母 A、B、C 的底层编码。根据 ASCII 编码的规范，这三个字母的编码分别为 65、66、67。

然后有下面的代码：

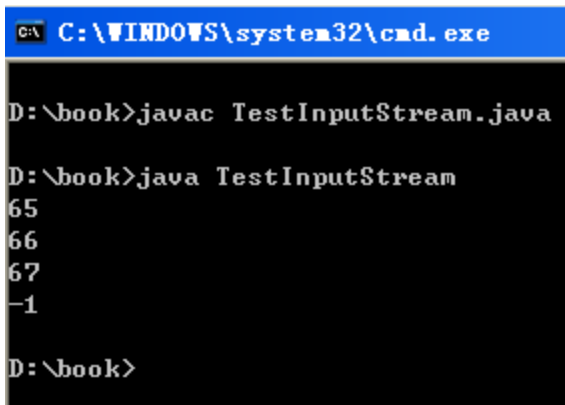
```
import java.io.*;

public class TestInputStream{
    public static void main(String args[]) throws Exception{
        FileInputStream fin = new FileInputStream("abc.txt");
        System.out.println(fin.read());
        System.out.println(fin.read());
        System.out.println(fin.read());
        System.out.println(fin.read());
        fin.close();
    }
}
```

我们总共调用了四次 `read` 方法。

当我们调用第一次 `read` 方法时，会从文件中读取一个字节，因此会输出 65；第二次时，会读取下一个字节，输出 66，第三次输出 67；当第四次调用 `read` 方法时，由于此时已经读

到了流的末尾，因此此时 read 方法返回值为-1。运行结果如下：



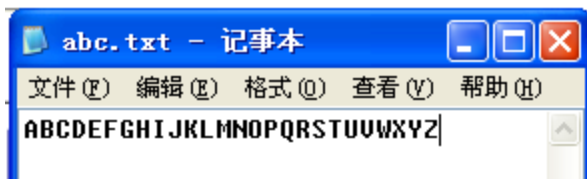
```
C:\WINDOWS\system32\cmd.exe

D:\book>javac TestInputStream.java

D:\book>java TestInputStream
65
66
67
-1

D:\book>
```

下面，我们把 abc.txt 增加内容，让这个文件中保存 26 个英文字母，如下：

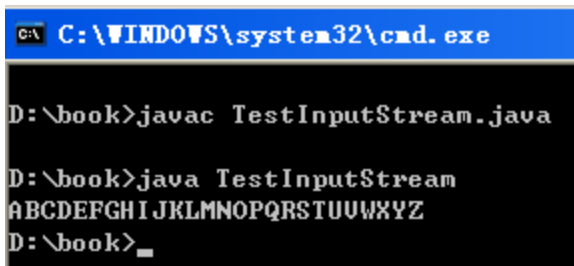


如果我们要读取这个文件中的所有内容，此时再一次一次调用 read 方法显然不现实，我们应当用 while 循环来完成。同时，由于 read 方法返回值为 int 类型，为了能够输出正常的字母，我们需要把 read 方法的返回值强制转换为 char 类型。典型代码如下：

```
import java.io.*;

public class TestInputStream{
    public static void main(String args[]) throws Exception{
        FileInputStream fin = new FileInputStream("abc.txt");
        int ch = 0;
        while( (ch=fin.read()) != -1){
            System.out.print((char)ch);
        }
        fin.close();
    }
}
```

这个代码利用 while 循环读取文件内容，并且把文件中所有内容都输出。运行结果如下：



```
C:\WINDOWS\system32\cmd.exe

D:\book>javac TestInputStream.java

D:\book>java TestInputStream
ABCDEFGHIJKLMNOPQRSTUVWXYZ

D:\book>
```

介绍完无参的 read 方法之后，下面我们来介绍有参的 read 方法。首先是带 byte 数组参数的 read 方法：

int read(byte[] bs)：这个方法的特点是，每次调用这个方法的时候，会把读取到的数据

放入 `bs` 数组中，一次调用尽量读取 `bs.length` 个字节。为什么说尽量读取 `bs.length` 个字节呢？假设 `bs.length` 长度为 6，如果流中剩余的字节数超过 6 个，那么这时候，调用这个 `read` 方法会把 `bs` 数组读满；而如果流中剩余的字节数不到 6 个的时候，那么调用 `read` 方法会把流中剩下所有的数据都读入到数组中。由于剩余的数据少于 6 个字节，因此数组不会被读满。

我们可以这么来理解：`read` 方法就好像是去食堂打饭一样。无参的 `read` 方法，相当于每次去食堂都只打一粒饭。而对于 `read(byte[])` 来说，就相当于每次去食堂打饭时都带着一个饭盒。比如这个饭盒能打 6 两饭，每次调用 `read` 方法，都尽量能够把这饭盒打满。如果食堂里面剩下的饭超过 6 两，那么要去打满一盒饭没有问题；而如果去食堂的时候已经比较晚了，食堂里只剩下 3 两饭了，这个时候，那只能打三两饭，即使饭盒不满也没办法。

理解了上面的问题，返回值也就好理解了：`int read(byte[] bs)` 这个方法的返回值，返回的是读到的字节数。对于大多数情况来说，读到的字节数和 `bs.length` 相同；而对于流中剩下的字节数少于 `bs.length` 的情况，`read` 方法返回的就是实际读到的字节数。

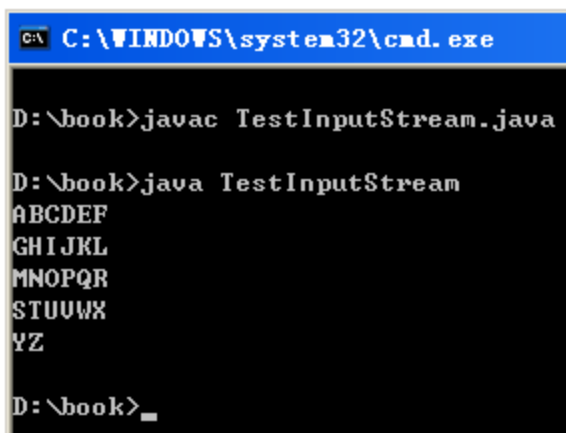
特别要注意的是，当读到流末尾时，返回 -1。

我们可以利用这个方法，改写一下之前的 `TestInputStream` 程序。修改过的代码和运行结果如下：

```
import java.io.*;

public class TestInputStream{

    public static void main(String args[]) throws Exception{
        FileInputStream fin = new FileInputStream("abc.txt");
        byte[] bs = new byte[6];
        int len = 0;
        while( (len=fin.read(bs)) != -1){
            for(int i = 0; i < len; i++){
                System.out.print((char)bs[i]);
            }
            System.out.println();
        }
        fin.close();
    }
}
```



```
C:\WINDOWS\system32\cmd.exe

D:\book>javac TestInputStream.java

D:\book>java TestInputStream
ABCDEF
GHIJKL
MNOPQR
STUUVX
YZ
D:\book>
```

需要注意的是，上面的代码我们使用了一个 `len` 变量用来保存每次 `read` 方法的返回值。读者可以思考一下为什么。

最后，简单介绍一下带多个参数的 `read` 方法：

`int read(byte[] bs, int off, int len)`：这个方法同样也是把数据读入数组中，只不过这个方法读取数据的时候，会让数据在数组中以下标为 `off` 的地方开始，并且最多只读取 `len` 个。也就是说，这个方法并没有使用整个 `bs` 数组来存放读到的数据，而是使用了数组的一部分。究竟是哪部分，是由 `off` 和 `len` 这两个参数决定的。这就好比我们去食堂打饭的时候，使用的是分过格子的饭盒，我们要打的也不是一盒饭，而是一格饭。

返回值是真正读取的字节数，当读到流末尾时，返回-1。

2.1.2 FileOutputStream

相对于 `FileInputStream`，`FileOutputStream` 更简单一些。首先同样是研究一下 `FileOutputStream` 的构造方法，罗列如下：

`FileOutputStream(String path)`：根据路径创建文件输出流

`FileOutputStream(File file)`：根据文件对象创建文件输出流

还有其他的构造方法，会在后面的内容中再次提到。

此外，下面是一些基本操作：

`close()`：关闭流

`FileOutputStream` 最重要的是 `write` 方法，其 `write` 方法罗列如下：

`void write(int v)`：这个 `write` 方法每次调用时写入一个字节。注意，虽然这个 `write` 方法接受的参数类型是 `int` 类型，虽然 `int` 类型有 4 个字节，但是这个 `write` 方法每次只写入 `int` 类型中最后的那个字节。

`void write(byte[] bs)`：写入一个 `byte` 数组。

`void write(byte[] bs, int off, int len)`：同样是写入一个 `byte` 数组，所不同的是，写入这个 `byte` 数组的时候，数据内容从数组下标 `off` 的位置开始，写入 `len` 个字节。

下面我们演示一下如何使用 `write` 方法。假设我们要写入一个字符串：**Hello World**，由于 `FileOutputStream` 中没有一个类能够接受一个字符串作为参数，因此我们必须要把字符串转化为 `byte` 数组。转换的方式也很简单，`String` 类中有一个方法叫做 `getBytes()`，签名如下：

```
public byte[] getBytes()
```

这个方法没有参数，并且返回一个 `byte` 数组，这就是字符串转为 `byte` 数组的方法。

我们利用这个方法，编写代码如下：

```
import java.io.*;

public class TestOutputStream{

    public static void main(String args[]) throws Exception{
        String hello = "Hello World";
        byte[] bs = hello.getBytes();
        FileOutputStream fout= new FileOutputStream("test.txt");
        fout.write(bs);
        fout.close();
    }

}
```

运行这个代码之前，当前目录下并不存在 `test.txt` 文件。

运行之后，在当前目录下产生 `test.txt` 文件，其中的内容为：**Hello World**。这也就提示

我们：如果文件不存在，`FileOutputStream` 会创建新文件。

那么文件已存在会怎么样呢？修改 `test.txt` 文件的内容并保存，然后再次运行程序，你会发现 `test.txt` 文件中的内容依然是 `Hello World`。

这说明：默认情况下，当文件已存在时，`FileOutputStream` 会覆盖同名文件。

我们也可以使用 `FileOutputStream` 另外两个构造方法：

`FileOutputStream(String path, boolean append)`

`FileOutputStream(File file, boolean append)`

这两个构造方法与原来的构造方法相比，增加了一个 `boolean` 参数：`append`。这个参数表示什么含义呢？

当这个参数为 `false` 的时候，这两个构造方法与只有一个参数的构造方法用起来相同：如果文件不存在，则 `FileOutputStream` 会创建新文件；如果文件已存在，则覆盖原文件。

如果这个参数为 `true`，则：当如果文件不存在，依然会创建新文件；而如果文件已存在，则会用追加的方式写文件。

我们可以写一个实验代码：

```
import java.io.*;

public class TestOutputStream{
    public static void main(String args[]) throws Exception{
        String hello = "Hello World";
        byte[] bs = hello.getBytes();
        FileOutputStream fout
            = new FileOutputStream("test.txt", true);
        fout.write(bs);
        fout.close();
    }
}
```

把上面的例子进行修改，在创建 `FileOutputStream` 的时候增加一个新的参数 `true`。如果 `test.txt` 文件不存在，第一次运行时，会创建这个文件，文件内容为：`Hello World`；反复运行这个程序多次，就会出现多个 `Hello World`。这就是用追加的方式写文件。

2.1.3 处理异常

之前，我们的程序处理异常的时候，都是使用 `throws` 的方式处理。下面，我们对 `TestInputStream` 程序进行修改，使用相对比较积极的 `try-catch` 的方式进行处理。

首先，在 `FileInputStream` 这个类中，有下面这些部分是需要进行异常处理的：

1、创建对象时

2、调用 `read` 方法时

3、调用 `close` 方法时

另外，要注意的是，`close` 方法属于释放资源的代码，应当写在 `finally` 代码块中。

修改之后的 `TestInputStream` 代码如下：

```
import java.io.*;

public class TestInputStream{
    public static void main(String args[]) {
        FileInputStream fin = null;
        try{
            fin = new FileInputStream("abc.txt");
```

```

        byte[] bs = new byte[6];
        int len = 0;
        while( (len=fin.read(bs))!=-1){
            for(int i = 0; i<len; i++){
                System.out.print((char)bs[i]);
            }
            System.out.println();
        }
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        if(fin!=null)
            try{
                fin.close();
            }catch(IOException e){
                e.printStackTrace();
            }
    }
}
}

```

有这样几个要点需要注意：

- 1、在 try 块之外定义 `FileInputStream` 变量 `fin`。原因在于：如果在 try 块内定义这个变量的话，则这个变量的作用范围就是 try 块内部，这样这个变量就无法在 finally 中使用。
- 2、给 `fin` 赋初值 `null`。原因在于，在 try 块内的代码被 Java 认为有可能不执行，如果不给 `fin` 赋初值的话，在 finally 中对 `fin` 的使用有可能没有赋初值，违反了对局部变量先赋值后使用的原则。
- 3、在 finally 中，调用 `close` 方法之前，需要判断一下 `fin` 变量是否为 `null`。

请读者根据上面的例子，修改 `TestOutputStream` 程序，使用 `try-catch-finally` 处理异常。

2.2 过滤流基础

介绍完节点流之后，下面我们开始介绍字节流的过滤流。首先介绍一下过滤流的基本使用。在学习过滤流的时候，关键的关键在于，一定要搞清楚过滤流增强了什么功能，以及在什么情况之下要使用过滤流的这种功能。

2.2.1 Data Stream

首先我们来介绍一对过滤流：`DataInputStream` 和 `DataOutputStream`。这两个类有什么作用呢？首先来思考下面的需求：

假设，要把一个 `double` 类型的数据写入文件中（例如 3.14），应当怎么做呢？由于 `FileOutputStream` 没有一个接受 `double` 类型作为参数的 `write` 方法，因此必须要想别的方法。

一个 `double` 变量占据 8 个字节，因此写一个 `double` 类型的时候，比较直观的方法应当是把这个 `double` 类型的数拆分成 8 个字节，然后把这 8 个字节写入到流中去。当然，把 `double` 类型进行拆分是一件比较麻烦的事情。

幸运的是，拆分 `double` 这件事情不需要程序员自己去做，而可以使用 Java 中现成的类。Sun 公司提供了 `DataInputStream` 以及 `DataOutputStream` 这两个类，这两个类为流增强的功能就是：增强了读写八种基本类型和字符串的功能。

我们可以看一下 `DataOutputStream` 的方法：除了有 `OutputStream` 中有的几个 `write` 方法之外，还有 `writeBoolean`, `writeByte`, `writeShort` ... 等一系列方法，这些方法接受某一种基本类型，把基本类型写入到流中。

需要注意的是，有一个 `writeInt(int n)` 方法，这个方法接受一个 `int` 类型的参数。这个方法与 `write(int v)` 方法不同。`writeInt` 方法是 `DataOutputStream` 特有的方法，这个方法一次写入参数 `n` 的四个字节。而 `write` 方法则一次写入参数 `v` 的最后一个字节。

与之对应的，`DataInputStream` 的方法中，除了有几个 `read` 方法之外，还有 `readBoolean`, `readByte`, `readInt` 等一系列方法，这些方法能够读入若干字节，然后拼成所需要的数据。例如 `readDouble` 方法，就会一次读入 8 个字节，然后把这 8 个字节拼接成一个 `double` 类型。

最后要提示的是，`DataXXXStream` 中有 `readUTF` 和 `writeUTF` 这两个方法用来读写字符串，但是一般来说，我们读写字符串的时候几乎不使用 Data 流。Data 流主要是用在 8 种基本类型的读写上。

下面，我们来写一个程序，在一个文件中存入 3.14 这个 `double` 值，然后再从文件中把这个值读取出来。在写代码之前，我们先来研究一下过滤流的使用。

2.2.1.1 过滤流使用的基本步骤

首先，过滤流的构造方法中，一般都会有一个构造方法，接受其他类型的流。例如，在 `DataInputStream` 的构造方法中，唯一的构造方法如下：

```
DataInputStream(InputStream is)
```

而 `DataOutputStream` 类唯一的构造方法如下：

```
DataOutputStream(OutputStream os)
```

这个参数表示什么呢？可以这么来理解：过滤流使用来为其他流增强功能的，而构造方法中的这个参数，表明的是过滤流为哪一个流增强功能。

过滤流的使用分为下面四个步骤：

- 1、创建节点流。这个步骤是使用过滤流的先决条件，由于过滤流无法直接实现数据传输功能，因此必须先有一个节点流，才能够进行数据传输。

- 2、封装过滤流。所谓的“封装”，指的是创建过滤流的时候，必须以其他的流作为构造方法的参数。需要注意的是，可以为一个节点流封装多个过滤流。

- 3、读/写数据。

- 4、关闭外层流。这指的是，关闭流的时候，只需要关闭最外层的过滤流即可，内层流会随着外层流的关闭而一起被关闭。

我们结合上面对过滤流的使用，给出下面的代码：

```

import java.io.*;
public class TestDataStream{
    public static void main(String args[]) throws Exception{
        //创建节点流
        FileOutputStream fout = new FileOutputStream("pi.dat");
        //封装过滤流
        DataOutputStream dout = new DataOutputStream(fout);
        //写数据
        dout.writeDouble(3.14);
        //关闭外层流
        dout.close();

        //创建节点流
        FileInputStream fin= new FileInputStream ("pi.dat");
        //封装过滤流
        DataInputStream din = new DataInputStream(fin);
        //读数据
        double pi = din.readDouble();
        //关闭外层流
        din.close();

        System.out.println(pi);
    }
}

```

有兴趣的读者可以尝试着处理一下上述代码的异常。

需要注意的是，由于 **Data** 流保存八种基本类型的方式采用的是拆分字节的方式，而不是采用文本的方式，因此保存的 **pi.dat** 这个文件无法用文本编辑器直接进行编辑。

2.2.2 Buffered Stream

下面我们要介绍的是一对流：**BufferedInputStream** 和 **BufferedOutputStream**。这两个流增强了缓冲区的功能。

什么叫缓冲区呢？在之前的代码中，我们每调用一次 **read** 或者 **write** 方法，都会触发一次 **I/O** 操作。而由于 **I/O** 操作要跨越 **JVM** 的边界，因此进行 **I/O** 操作的时候，事实上效率会非常低，这非常不利于程序的高效。为了让程序的效率得到提升，我们引入了缓冲机制。

我们会在内存中开辟一块空间，当调用 **read** 或者 **write** 方法时，并不真正进行 **I/O** 操作，而是对内存中的这块空间进行操作。我们以 **write** 操作为例，使用了缓冲机制之后，我们调用 **write** 方法时，并不真正把数据写入到文件中，而是先把数据放到缓冲区里。等到缓冲区满了之后，再一次性把数据写入文件中。

为什么这样就能提高效率了呢？考虑下面这个生活中的例子。在大学里面，我们都用手洗衣服。手洗衣物有一个很大的问题，往往衣服洗完以后很难拧干，晾在阳台上之后会滴水。如果任由衣物在阳台上滴水的话，有可能会让阳台变得很脏，如果阳台上放了一些其他的杂物的话，更有可能因为滴水而损坏那些物品。因此，我们需要使用一个方式，把衣服上滴的

水运输到洗手间，倒掉。

那怎么运输呢？一般来说，我们不会在阳台傻等，等一滴水下来以后马上就用手接着然后跑到洗手间倒掉。我们往往会用一个盆先接水，当这个盆满了以后，我们才会真正把这个盆中的水倒掉，也就是真正完成 I/O 操作。这个盆，就好比是我们的缓冲区。通过这个盆，我们减少了 I/O 的次数，从而提高了 I/O 的效率。

Buffered 流几乎没有为流增加新的方法，我们给出一个输出流的例子代码：

```
import java.io.*;

public class TestBufferedStream{
    public static void main(String args[]) throws Exception{
        String data = "Hello World";
        byte[] bs = data.getBytes();
        //创建节点流
        FileOutputStream fout
            = new FileOutputStream("test.txt");
        //封装过滤流
        BufferedOutputStream bout
            = new BufferedOutputStream(fout);
        //写数据
        bout.write(bs);
        //关闭外层流
        bout.close();
    }
}
```

需要注意的是，如果把 `bout.close()` 方法去掉，此时在看 `test.txt` 文件，会发现文件的内容为空。这是因为，我们在调用 `write` 方法的时候，其实并没有真正把数据写入到文件中，而只是把数据写入到缓冲区中。那什么时候缓冲区中的数据会真正写入到文件中呢？有三种情况：第一种情况是缓冲区已满，第二种情况是调用 `close` 方法。

除了这两种情况之外，假设程序员希望在缓冲区没有满并且不关闭流的情况下，把缓冲区内的东西真正写入流中，应当调用一个方法：`flush()`。这个方法用来清空缓冲区，往往用在输出流上面。当一个带缓冲的输出流调用 `flush()` 之后，就能保证之前在缓冲区中的内容真正进行了 I/O 操作，而不是仅仅停留在缓冲区。

2.2.3 PrintStream

PrintStream 是一个比较特殊的过滤流，我们简单介绍一下，读者作为一般性的了解即可。

PrintStream 作为过滤流，增强的功能有以下几个：

- 1、缓冲区的功能
- 2、写八种基本类型和字符串
- 3、写对象

需要注意的是，这个流写基本类型和写对象的时候，是按照字符串的方式写的。也就是

说，这个流写八种基本类型的时候，会把基本类型转换成字符串以后再写，而写对象的时候，会写入对象的 `toString()` 方法返回值。

这个类具体如何使用我们不多介绍了。需要介绍的是这个类的一个对象：我们所熟知的向屏幕输出数据的对象：`System.out` 对象，这就是一个 `PrintStream` 类型的对象。

2.3 对象序列化

2.3.1 序列化的概念

下面我们为大家介绍另外一对过滤流：`ObjectInputStream` 和 `ObjectOutputStream`。这两个也是过滤流，增强的功能如下：

- 1、增强了缓冲区功能
- 2、增强了读写八种基本类型和字符串的功能。读写基本类型和字符串的方式，与 `Data` 流完全一样。
- 3、增强了读写对象的功能。这是这两个流最主要的作用。在 `ObjectInputStream` 类有一个 `readObject` 方法，这个方法能够从流中读取一个对象；而 `ObjectOutputStream` 类中有一个 `writeObject` 方法，这个方法能够向流中写入一个对象。

如上所述，`ObjectInputStream` 和 `ObjectOutputStream` 能够完成对对象的读写。这种把对象放到流上进行传输的过程，称之为“对象序列化”。一个对象如果能够放到流上进行传输，则我们称这个对象是“可序列化”的。

2.3.2 `Serializable` 接口和 `transient` 关键字

需要注意的是，并不是所有对象都是“可序列化”的。举个例子说，搬家就是一个传输对象的过程。然而，搬家的时候并不是所有对象都能够搬走的。例如，家具、电器，这些对象往往在搬家的时候是能够搬走的，但是，门、窗户、地板，这些对象无法搬走。那我们可以说家具、电器是可序列化的对象，而窗户、地板是不可序列化的对象。

那怎么让对象能够在流上进行传输呢？如果要让一个类成为可序列化的，只要让这个类实现一个接口：`java.io.Serializable` 接口即可。

要实现这个接口，就要实现这个接口中的所有方法。好了，现在请去查一下 `Serializable` 接口，看看这个接口中定义了哪些方法？

让你惊讶吧，这个接口中没有任何的方法。也就是说，如果来实现这个 `Serializable` 接口，只需要写上 `implements Serializable` 就可以了。

我们可以尝试一下。定义一个 `Student` 类，创建两个对象并保存到文件中，然后再利用另一个流读取文件。代码如下：

```
import java.io.*;

class Student implements Serializable{
    String name;
    int age;
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```



```

}

public class TestSerializable {
    public static void main(String[] args) throws Exception {
        Student stu1 = new Student("tom", 18);
        Student stu2 = new Student("jerry", 18);

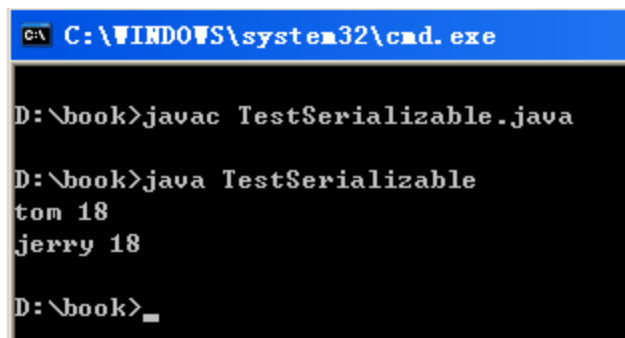
        FileOutputStream fout = new FileOutputStream("stu.dat");
        ObjectOutputStream oout = new ObjectOutputStream(fout);
        oout.writeObject(stu1);
        oout.writeObject(stu2);
        oout.close();

        FileInputStream fin = new FileInputStream("stu.dat");
        ObjectInputStream oin = new ObjectInputStream(fin);
        Student s1 = (Student) oin.readObject();
        Student s2 = (Student) oin.readObject();
        oin.close();
        System.out.println(s1.name + " " + s1.age);
        System.out.println(s2.name + " " + s2.age);

    }
}

```

需要注意的是,由于 `readObject` 方法返回值为 `Object` 类型,因此需要对返回值进行强转。运行结果如下:



```

C:\WINDOWS\system32\cmd.exe

D:\book>javac TestSerializable.java

D:\book>java TestSerializable
tom 18
jerry 18

D:\book>_

```

同时,也产生了一个 `stu.dat` 文件。这个文件中保存的都是一些二进制数据,这些二进制数据就是保存对象时保存的数据。

下面,我们介绍一个新的关键字: `transient`。这个关键字是一个修饰符,这个修饰符可以用来修饰属性,用 `transient` 修饰的属性表示:这个属性不参与序列化。

我们可以修改原有的代码,把 `Student` 类的 `age` 属性修改为 `transient` 的。这样之后产生的改变是: 1、`stu.dat` 文件的大小变小了。这很容易理解,因为由于参与序列化的属性变少了,因此序列化之后保存的数据也变少了,从而导致文件也变小了。2、输出的 `age` 属性都为 0。这是因为由于在 `writeObject` 的时候没有保存 `age` 属性,而读取时从文件中也读取不到

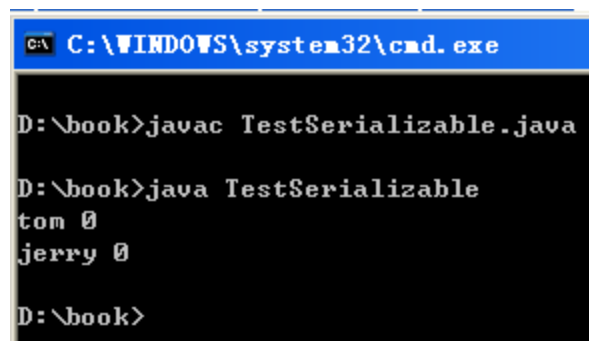
age 属性，从而导致这个属性的值只能是默认值：0。

修改后的代码以及运行结果如下：

（其他代码与上一个例子相同）

```
class Student implements Serializable{
    String name;
    transient int age;
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

运行结果：



另外，在使用对象序列化的时候，注意这样两个问题：

1、不要使用追加的方式写对象。也就是说，如果我们创建一个文件输出流采用 `FileOutputStream(file, true)` 的方式创建节点流，然后再在外面封装 `ObjectOutputStream`，这样将无法完成我们设想的结果。如果对一个文件多次写入的话，读取对象的时候只能读取第一次写入的对象，而后面用追加的方式写入的对象将无法被读取。这是对象序列化底层机制所决定的。

2、如果一个对象的属性又是一个对象，则要求这个属性对象也实现了 `Serializable` 接口，如果一个对象的属性是一个集合，则要求集合中所有对象都实现 `Serializable` 接口。除非这个对象的属性被标记为 `transient`，不参与序列化。

这就是最基本的对象序列化的内容。关于对象序列化的更多内容，可以参考 Sun 公司的网站以及相关文档。

3 字符流

研究完字节流之后，我们开始介绍字符流。要理解字符流，首先要理解字符编码的含义。

3.1 字符编码

计算机中显示文字的时候，本质上是在屏幕上绘制一些图像用来显示文字。从这个意义上说，文字就是一种特殊的图片。

然而，在计算机中保存文字的时候，并不是按照图片的方式保存。当保存文件的时候，

计算机底层会把文字转换成数字，然后再进行保存。计算机把字符转换为数字的过程，称之为“编码”。而读取文件的时候，则过程相反，计算机会把数字转化为文字，并绘制到屏幕上。计算机把数字转换为字符的过程，称之为“解码”。

很显然，不同的字符必须对应不同的数字，不然，在解码时会遇到问题。

那么，什么字符对应于什么数字呢？有些标准化组织，会规定字符和数字之间的对应关系，这种对应关系就是所谓的编码规范。常见的编码规范如下：

ASCII：最早的编码方式，规定了英文字母和英文标点对应的编码

ISO-8859-1：这种编码方式包括了所有的西欧字符以及西欧标点。

GB2312/ GBK：大陆广泛使用的简体中文编码。其中，GB2312 是 GBK 的一个子集，也就是说，在 GB2312 中有的汉字，在 GBK 中也有，且同一个汉字在 GB2312 和 GBK 中的编码相同。GBK 主要是在 GB2312 的基础上扩展了很多新的字符。

GBK 是简体中文 Windows 的默认编码方式。

Big5：台湾地区广泛使用的繁体中文编码。

UTF-8：一种国际通用编码，包括简体和繁体中文。与 GB2312/GBK 不兼容，也就是说，同一个汉字，在 GBK 和 UTF-8 的编码是不同的。大部分简体中文 Linux 使用的是 UTF-8 编码。

由于有了多种编码规范，因此就会有乱码的问题。乱码问题是怎么产生的呢。例如，我们在保存文件的时候，使用了 GBK 编码，这个时候，假设一个字符“程”，被编码成了数字 31243，于是这个文件在底层保存的数据就是 31243。之后，这个文件被传送到台湾，台湾地区的工程师打开文件的时候，使用的软件对这个文件采用 Big5 解码。此时，就会把 31243 这个数字给解码成字符“最”。这样，就会产生理解上的误会，从而产生乱码。

换言之，产生乱码的根源在于：编解码方式不一致。

我们可以用程序来演示编解码。之前我们说过，String 类有一个方法 `getBytes()`，这个方法能够把字符串转换成一个 byte 类型的数组，实际上就是把字符转化为数字的过程，本质上，就是在进行编码。在调用 `getBytes` 的时候，也可以指定编码方式。

那得到 byte 数组之后，如何解码呢？String 类有一个构造方法，能够接受 byte 数组作为参数，这就是能够把数字转化为字符串，本质上是解码的过程。在构造的时候，也可以指定解码的方式。

示例代码如下：

```
public class TestEncoder {
    public static void main(String[] args) throws Exception {
        String str = "欢迎学习 Java";
        //编码，指定编码方式为 GBK
        byte[] bs = str.getBytes("GBK");
        //解码，指定解码方式为 GBK
        String str2 = new String(bs, "GBK");
        System.out.println(str2);
    }
}
```

上面的程序中，编码和解码的方式都为 GBK，输出结果为：

```
C:\WINDOWS\system32\cmd.exe

D:\book>javac TestEncoder.java

D:\book>java TestEncoder
欢迎学习Java

D:\book>
```

可以看到，没有乱码。

如果修改一下，编码用 GBK，解码用 Big5，则输出结果如下：

```
D:\book>javac TestEncoder.java

D:\book>java TestEncoder
辣荦悝煨Java

D:\book>
```

可以看到，产生了乱码。同样的，如果编码用 GBK，而解码用 UTF-8，则输出结果如下：

```
D:\book>javac TestEncoder.java

D:\book>java TestEncoder
?????Java

D:\book>
```

同样产生了乱码。

需要注意的是，英文字母（例如上面字符串中的“Java”），无论采用什么方式编码和解码，都不会产生乱码。世界上任何一种编码方式，都与 ASCII 编码兼容，也就是说，任何一种编码方式下面，A 都对应 65，a 都对应 97，没有例外。

3.2 获得字符流与桥转换

由于编码方式的不一致，导致了传输文本的时候会有一些比较棘手的问题。为了让传输文本文件更加方便，我们使用字符流。

首先是字符流的父类。所有输入字符流的父类是 `Reader`，所有输出字符流的父类是 `Writer`。与 `InputStream` 和 `OutputStream` 类似，这两个类也是抽象类。

此外，与 `FileInputStream` 以及 `FileOutputStream` 类似，有两个类 `FileReader` 和 `FileWriter`，这两个类分别表示文件输入字符流和文件输出字符流。这两个流的使用与 `FileInputStream` 以及 `FileOutputStream` 也非常雷同，在此不多介绍，需要注意的是，使用这两个流的时候，无法指定编解码方式。

通过 `FileReader` 和 `FileWriter` 可以直接获得文件字符流。

下面，我们介绍两个流：`InputStreamReader` 和 `OutputStreamWriter`。

`InputStreamReader` 这个类本身是 `Reader` 类的子类，因此这个类的对象是一个字符流。而这个流的构造方法如下：

Constructor Summary

<code>InputStreamReader(InputStream in)</code>	Creates an <code>InputStreamReader</code> that uses the default charset.
<code>InputStreamReader(InputStream in, Charset cs)</code>	Creates an <code>InputStreamReader</code> that uses the given charset.
<code>InputStreamReader(InputStream in, CharsetDecoder dec)</code>	Creates an <code>InputStreamReader</code> that uses the given charset decoder.
<code>InputStreamReader(InputStream in, String charsetName)</code>	Creates an <code>InputStreamReader</code> that uses the named charset.

可以看到，这个流所有构造方法，都可以接受一个 `InputStream` 类型的参数。也就是说，通过这个流，可以接受一个字节流作为参数，创建一个字符流。这个对象就起到了字节流向字符流转换的功能，我们往往称之为：桥转换。

类似的，`OutputStreamWriter` 类能够把一个输出字节流转换为一个输出字符流。

在桥转换的过程中，我们还可以指定编解码方式。如果不指定的话，则编码方式采用系统默认的编码方式。

因此，通过桥转换获得字符流，也是一个获得字符流的方式。这种方式有两个用法：

- 1、如果需要指定编码方式，则应当使用桥转换。
- 2、在无法直接获得字符流的情况下，可以先获得字节流，再通过桥转换获得字符流。

利用桥转换进行编程，需要以下五个步骤：

- 1、创建节点流
- 2、桥转换为字符流
- 3、在字符流的基础上封装过滤流
- 4、读/写数据
- 5、关闭外层流

3.3 字符过滤流

介绍完如何获得字符流之后，下面单刀直入，介绍一些字符流的过滤流。对于字符流来说，常用的过滤流只有两个：读入使用 `BufferedReader`，写出使用 `PrintWriter`。下面我们分别进行介绍。

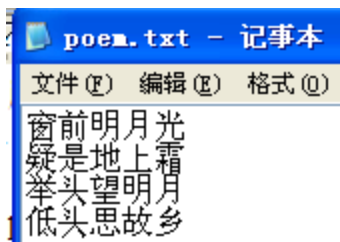
3.3.1 BufferedReader

顾名思义，`BufferedReader` 提供了缓冲区功能。但是更重要的是，`BufferedReader` 中有一个 `readLine()` 方法，签名如下：

```
public String readLine()
```

这个方法也很容易理解：每次读入一行文本，并把读入的这一行文本当做返回值返回。当读到流末尾时，返回一个 `null` 值。

下面给出一个示例代码，介绍 `BufferedReader` 的使用。首先在当前目录下准备一个文本文件，文本文件中保存一段文字。对于 Windows 来说，默认的编码为 GBK。如下：



然后有如下代码：

```
import java.io.*;
public class TestPoem {
    public static void main(String[] args) throws Exception {
        //创建节点流
        FileInputStream fin = new FileInputStream("poem.txt");
        //桥转换
        Reader r = new InputStreamReader(fin, "GBK");
        //封装过滤流
        BufferedReader br = new BufferedReader(r);
        //读/写数据
        String line = null;
        while( (line=br.readLine()) !=null){
            System.out.println(line);
        }
        //关闭外层流
        br.close();
    }
}
```

上面的代码演示了如何使用 **BufferedReader** 读取文件。

3.3.2 PrintWriter

PrintWriter 是一个很特殊的类。

首先，**PrintWriter** 可以作为一个过滤流。这个流可以接受一个 **Writer** 作为参数。增强了如下一些功能：

- 1、缓冲区的功能。因此使用 **PrintWriter** 应当及时关闭或刷新
- 2、写八种基本类型和字符串的功能。
- 3、写对象的功能。

在 **PrintWriter** 类中，有一系列 **print** 方法，这些方法能够接受八种基本类型、字符串和对象。同样的，还有一系列 **println** 方法，这些方法在写入数据之后，会在数据后面写入一个换行符。

要注意的是，**PrintWriter** 写基本类型的方式，是把基本类型转换为字符串再写入流中，与 **Data** 流不同。举例来说，对于 3.14 这个 **double** 类型的数，**Data** 流会把这个数拆分成 8 个字节写入文件，而 **PrintWriter** 会把这个数字转化为字符串 “3.14”，写入文件中。

此外, `PrintWriter` 写对象的时候, 写入的是对象的 `toString()` 方法返回值, 与对象序列化有本质区别。

`PrintWriter` 除了可以作为过滤流之外, 还可以作为节点流。`PrintWriter` 类的构造方法中, 可以直接接受一个文件名或 `File` 对象作为参数, 直接获得一个输出到文件的 `PrintWriter`。当然, 编码方式采用的是系统默认的编码方式。

最后, `PrintWriter` 的构造方法可以接受一个 `InputStream`, 也就是说, 可以使用 `PrintWriter` 进行桥转换。只不过使用 `PrintWriter` 进行桥转换的时候, 无法指定编码方式, 采用的是系统默认的编码方式。

下面, 我们把 `PrintWriter` 当做过滤流, 给出一段代码的例子:

```
import java.io.*;

public class TestPrintWriter {
    public static void main(String[] args) throws Exception {
        FileOutputStream fout
            = new FileOutputStream("poem2.txt");
        Writer w = new OutputStreamWriter(fout, "GBK");
        PrintWriter pw = new PrintWriter(w);
        pw.println("一个人在清华园");
        pw.println("我写的 Java 程序");
        pw.println("是全天下");
        pw.println("最面向对象的");
        pw.close();
    }
}
```

上面的程序在当前目录下写入了一首现代诗。读者可以思考一下, 如果把 `PrintWriter` 当做节点流、以及把 `PrintWriter` 用来做桥转换, 分别应当怎么改写上面的代码。