# Java流与文件

在Java程序设计语言中，一个可以读取字节序列的对象被称为输入流（input stream）。一个可以写入字节序列的对象被称为输出流（output stream）。在抽象类InputStream和OutputStream中对它们进行了说明。而对于读写基于双字节的Unicode代码单元，由抽象类Reader和Writer进行处理。

**读写字节：**

**InputStream**类有一个抽象方法：abstract int read()。该方法将读取一个字节，并将其返回。如果已经读到了输入源的末尾，将返回-1。具体的输入流类的设计者应该覆盖这个方法以提供具体的功能实现。例如，在FileInputStream类中，这个方法将从一个文件中读取一个字节。System.in是一个预定义的InputStream的子类的对象，使用它可以从键盘读取信息。

**OutputStream**类定义了抽象方法：abstract void write(int b)。将一个字节写到指定的输出位置。

不论是read方法还是write方法都能阻塞（block）一个线程直到字节被真正地读取或写入。这意味着如果流不能立即被读取或者写入（通常是因为网络连接繁忙），Java就会挂起这个调用的线程。这样就能够给其他线程提供机会去做有用的工作，同时这个方法将一直等待，直到流再次可用为止。

available方法能够检查目前可以读取的字节数。也就是说，下面这样的代码片段不可能永远被阻塞：

int bytesAvailable = in.available();

if(bytesAvailable > 0) {

byte[] data = new byte[bytesAvailable];

in.read(data);

}

当完成了对一个流的读取或者写入后，就应该调用close方法将它关闭，这样可以释放流所占用的操作系统资源。关闭一个输出流可以刷新（flush）输出流占用的缓存区，即临时存储在缓冲区中等待形成较大的数据包后再发送的那些字符，此时将它们发送出去。特别要注意的是，如果不关闭文件，最后一个字节包可能永远也不会被发送。当然也可以用flush方法手工刷新输出。

Java给你提供了许多由基本的InputStream和OutputStream类派生出来的流类。Java有一系列流类型，其数量超过60种。库类设计者声称：“有足够的理由为用户提供丰富的流类型的选择：这样做可以减少程序的错误。”

   

前面已经说过，InputStream和OutputStream类仅仅读取和写入单个的字节和字节数组；它们没有读取和写入字符串和数值的方法。需要具有功能更加强大的子类来实现这些目标。例如，DataInputStream和DataOutputStream能够读取所有的基本的Java类型。

另一方面，对于Unicode文本来说，前面已经讲过，可以使用从Reader和Writer中派生出来的类。Reader和Writer类的基本方法和InputStream以及OutputStream类非常类似。

abstract int read();

abstract int writer(int b);

区别在于read方法返回的是一个Unicode字符单元（0到65535中的整数），或是-1（当已经到达文件的末尾的时候）。

**文件路径中的注意事项**：因为反斜线符在Java字符串中作为转意字符使用，所以要在Windows格式的路径名中使用\\作为路径的分隔符（”C:\\Windows\\win.ini”）。在Windows系统中，也可以使用单独的正斜线符号（”C:/Windows/win.ini”），这是因为大多数Wiondows文件处理系统将正斜线符号做为文件分隔符。但是我们不推荐这么做——Windows的系统功能可能会改变，并且在其他的操作系统上，文件分隔符可能并不相同。为了程序的可移植性，我们应该使用File.separator。

**所有在java.io中的类都是将相对路径名解释为起始于用户的当前工作目录，所以应该清楚当前的目录。可以通过通过调用System.getProperty(“user.dir”);来获得。**

FileInputStream和FileOutputStream能够把输入和输出流与磁盘文件关联起来：

FileInputStream fin = new FileInputStream(new File(“employee.dat”));

与抽象的类InputStream和OutputStream类似，这些类只在字节层次上支持读写。这意味着只能从fin对象中读取字节和字节数组。马上就可以看到，如果正好有一个DataInputStream，就可以读取数值类型：

DataInputStream din = new DataInputStream(new FileInputStream(new File(“employee.dat”)));

double s = din.readDouble();

Java使用了一种很聪明的策略来划分这两种职责。一些流（例如FileInputStream以及URL类中利用opeStream方法来返回输入流）可以从文件以及其他地方接收字节。另一些流（例如DataInputStream和PrintWriter）可以将字节组合成更加有用的数据类型。Java程序员采用将一个已经存在的流传递给另一个流的构造器方法，将这两种流结合起来，结合后的流被称为**过滤流**（filtered stream）。例如，为了能够从文件中读取数值，首先创建一个FileInputStream，然后将它传递给一个DataInputStream的构造器。

在默认情况下，流是不能进行缓冲处理的。换句话说，每次对流进行read读取都会要求操作系统发送一个新字节。如果想要对当前目录下的名为employee.dat文件进行缓冲和数据输入操作，就需要使用下面这种有着奇异顺序的构造器：

DataInputStream din = new DataInputStream(new BufferedInputStream(new FileInputStream(“employee.dat”)));

注意，将DataInputStream放置在这个构造器链的最后，其原因是要利用DataInputStream方法，并且希望它们使用有缓冲的read方法。尽管上述代码的格式有些别扭，但是，有必要这样做：**必须将流构造器连续分层，直到能够使用所需要的功能为止。**

当读取输入的时候，经常需要查看下一字节是否是想要的值。Java为这种需要提供了PushbackInputStream。

PushbackInputStream pbin = new PushbackInputStream(new BufferedInputStream(new FileInputStream(“employee.dat”)));

现在，可以冒险的读下一字节：

int b = pbin.read();

如果不是自己想要的，将它扔回去。

if(b != ‘<’) pbin.unread(b);

如果既想预查看，又想读入数值，就需要同时使用一个PushbackInputStream输入流和数据输入流引用：

DataInputStream dataInputStream = new DataInputStream(new PushbackInputStream(new BufferedInputStream(new FileInputStream(“employee.dat”))));

求助流分层过滤器，这的确有些麻烦。但是，因为能够将过滤器进行混合与匹配，从而构造出真正实用的流序列，所以这也带来了很大的灵活性。例如：利用下面的流序列可以从一个压缩的ZIP文件中读取数值：

ZinInputStream zin = ZinInputStream(new FileInputStream(“employee.zip”));

DataInputStream din = new DataInputStream(zin);

RandomAccessFile流类能够在文件的任何位置查找或者写入数据。它同时实现了DataInput和DataOutput接口。磁盘文件可以随机存取，但是来自网络的数据流就不行。打开一个随机存取文件，要么进行只读操作，要么进行读写操作。可以通过构造器的第二个参数带入“r”（读取）或“rw”（读写）指定相应的操作：

RandomAccessFile in = new RandomAccessFile(“employee.dat”, “r”);

RandomAccessFile inOut = new RandomAccessFile(“employee.dat”, “rw”);

**注意，当打开一个现有文件作为RandomAccessFile时，原来的文件不会被删除。**

随机存取文件同时提供了一个**文件指针**(file pointer)。文件指针总是指向下一条要进行读写操作记录的位置。Seek方法将文件指针设定在文件内部的任意字节位置。Seek使用的参数是一个在0到文件长度（以字节计）之间的long整数。

要想从一个随机存储文件中读取数据，可以使用与DataInputs对象相同的方法，例如readChar和readChar。因为这些方法实际上是在DataInput接口中定义的，而DataInputStream和RandomAccessFile都实现了这个接口。同样，要写入一个随机存取文件，可以使用DataOutputStream类中的writeInt和writeChar方法。这些方法都是在DataOutput接口中定义的，而上面提到的两个类都实现了这个接口。

在JDK1.4中引入了java.nio包通过引入Charset来同意字符集的转换，字符集给出了双字节Unicode码序列与在本地字符编码中采用的字节序列间的映射。最流行的字符编码之一是ISO-8859-1，在这种编码中，前面的256个Unicode字符是单字节编码。

Charset类使用的是IANA字符集注册的标准字符集名。每一个字符集都有一系列的别名，字符集的名称不区分大小写。可以使用官方名称或者任何一个别名调用静态方法forName来获得Charset：

Charset cset = Charset.*forName*("ISO-8859-1");

aliases方法返回别名的Set对象：

**for**(String alias : cset.aliases()) {

System.*out*.println(alias);

}

Java的国际版支持许多编码形式。还为附加字符集的提供者提供了相应的机制。为了找出特定的实现中可用的字符集，可以使用如下代码：

SortedMap<String, Charset> charsets = Charset.*availableCharsets*();

**for**(String name : charsets.keySet()) {

System.*out*.println(name);

}

本地的编码方案无法表示所有的Unicode字符，如果一个字符无法表示，则将变换为“?”。

一旦有了字符集，就可以用它在Unicode字符串和字节序列编码之间进行转换，下面代码对Unicode字符串进行编码：

Charset cset = Charset.*forName*("ISO-8859-1");

String str = "abcd12345";

ByteBuffer bbuf = cset.encode(str);

byte[] bytes = buffer.array();

反过来，对一个字节序列编码，需要一个字节缓冲区。使用ByteBuffer数组的静态wrap方法把一个字节数组转化为一个字节缓冲区。decode方法的结果是一个CharBuffer。调用它的toString方法可以得到一个字符串。

我们知道，当以二进制格式写入数据时，应当使用DataOutputStream。而当以文本格式写入数据时，应当使用PrintWriter。PrintWriter可以以文本格式打印字符串和数值，与DataOutput类似，尽管提供了一些很有用的输出方法，但却没有定义目的地，一个PrintWriter必须与一个目标Writer相结合：

PrintWriter out = new PrintWriter(new FileWriter("employee.txt"));

也可以将一个PrintWriter与目的（输出）流合并在一起：

PrintWriter out = new PrintWriter(new FileOutputStream("employee.txt"));

PrintWriter(OutputStream)构造器自动增加一个OutputStreamWriter来将Unicode字符转换为流内的字节。

**在文件中，如果要写入换行符，由于不同操作系统文件换行符不一样（在Windows中是\r\n，在Unix上是\n，在Macs上是\r），可以使用System.getProperty("line.separator")来获取;**

如果PrintWriter被设为自动刷新模式，则任何时候调用pringln，缓冲区内所有字符都会被送到目的地(PrintWriter总是带缓冲的，默认不开启自动刷新)。可以使用PrintWriter(Writer, boolean)构造器来开启或关闭自动刷新：

PrintWriter out = new PrintWriter(new FileWriter("employee.txt"), true);

在JDK5.0之前，唯一能用来处理文本输入的是BufferedReader，在这个类中，有一个readLine方法，它能以行的方式读取文本，并需要将一个缓冲的读取器同一个输入源结合起来：

BufferedReader in = new BufferedReader(new FileReader("employee.txt"));

如果readLine方法返回null。

FileReader类已经把字节转化为Unicode字符。对于其他输入源，需要使用InputStreamReader，与PrintWriter不同，InputStreamReader没有包含自动弥补字节和Unicode字符之间差别的方法。

BufferedReader in2 = new BufferedReader(new InputStreamReader(System.in));

BufferedReader in2 = new BufferedReader(new InputStreamReader(url.openStream()));

Java的StringReader和StringWriter类可以像对待一个数据流一样对待字符串。这对于想要使用相同的代码从流中分析字符串和数据来说十分有用。

处理ZIP文件的类java.util.zip中而不在java.io中。

当读取一行输入时，会得到一个很长的字符串，有时候希望将它分隔为独立的字符串。这就意味着要找出其中的（例如”|”）分隔符并且将其分离成单独片段，即下一个分隔符之间的字符序列（这种分隔符通常称为记号token）。java.util的StringTokenizer类就是为这个目的设计的：

StringTokenizer tokenizer = new StringTokenizer(line, “|”);

也可以在一个字符串里指定多个分隔符：

StringTokenizer tokenizer = new StringTokenizer(line, “|,;”);

如果不指定分隔符集合，默认的就是”\t\n\r”,即所有的空白字符（空格、tab，新行、回车），一旦构造好了一个字符串记号处理器，就可以从字符串中快速找出记号：

while(tokenizer.hasMoreTokens()) {

String token = tokenizer.nextToken();

……

}

成功跳过序列化部分。。。。。。

在**File**类中封装了对用户机器的文件系统进行操作的功能。File类仅仅有最少的公共操作方法。比如在Windows中，额可以查询（或设置）文件的只读标识，但尽管可以查询一个文件是否隐藏，却不能将它设置为隐藏，除非使用本地方法。

File对象的一个最简单的构造器需要一个（完整的）文件名。如果不提供一个路径名，Java会使用当前目录。“**当前目录**”就是目前执行虚拟机的处理程序的当前目录。如果从命令行运行虚拟机，这个目录就是开始java运行的目录。

如果文件名不存在，则对这个构造器的调用不会创建使用这个名字的新文件。实际上，用一个File对象创建文件，需要使用一个流泪的构造器或者使用由File类提供的**createNewFile**方法。只有当前不存在指定名字的文件时，createNewFile方法才会创建文件，而且返回的boolean值用于表示是否成功。另一方面，一旦有了File对象，File类中的**exists**方法就会告诉我们是否存在指定名字的文件。但即使这个文件不存在，我们仍然可以调用**getAbsolutePath**方法来返回它的绝对路径，**getCanonicalPath**方法用来返回规范的路径名。

File还有另外两个构造器，File(String path, String name)它创建了一个由path参数指定目录下的给定名称的File对象。（如果path为null，这个构造器将会创建一个使用当前目录的File对象。）File(File dir, String name)构造器能在已经存在File目录对象下创建一个指定名称的File对象。（同理，如果dir为null，构造器则会在当前目录下创建一个File对象。）

File即可表示一个文件，也可以表示一个目录，当目录不存在时，可以通过**mkdir**和**mkdirs**来创建它。可以通过**isDirectory**和**isFile**来判断文件对象表示的是目录还是文件。如果一个File对象表示目录，可以通过list方法来获得该目录下的所有文件名的数组。

当编写可移植程序时，指定带有子目录的文件名是一个很大的挑战，在Windows中可以用一个正斜杠（UNIX系统分隔符）作为目录的分隔符。然而其他操作系统可能不允许这样，所以不推荐使用正斜杠。可以用File.separator来解决这种问题，它会在不同的操作系统中提供正确的文件分隔符。如：

File foo = new File(“Documents” + File.separator + “data.txt”);

注意，如果在Windows中构造File对象时，使用正斜杠作为目录分隔符，那么getAbsolutePath方法将返回一个包括正斜杠的文件名,Windows用户会感觉有点奇怪。实际上，应该使用getCanonicalPath方法。它将使用反斜杠代替正斜杠。