北京圣思园科技有限公司
http://www.shengsiyuan.com

主讲人：张龙

Java I/O系统

- 课程目标
  - 理解Java I/O系统
  - 熟练使用java.io包中的相关类与接口进行I/O编程
  - **掌握Java I/O的设计原则与使用的设计模式**

Java I/O系统

- 对程序语言设计者来说，设计一个令人满意的I/O（输入输出）系统，是件极艰巨的任务
  - 摘自《Thinking in Java》

流 类

- 流的概念
  - Java程序通过流来完成输入/输出。流是生产或消费信息的抽象。流通过Java的输入/输出系统与物理设备链接。尽管与它们链接的物理设备不尽相同，所有流的行为具有同样的方式。这样，相同的输入/输出类和方法适用于所有类型的外部设备。这意味着一个输入流能够抽象多种不同类型的输入：从磁盘文件，从键盘或从网络套接字。同样，一个输出流可以输出到控制台，磁盘文件或相连的网络。流是处理输入/输出的一个洁净的方法，例如它不需要代码理解键盘和网络的不同。Java中流的实现是在java.io包定义的类层次结构内部的。

输入/输出流概念

- 输入/输出时，数据在通信通道中流动。所谓“数据流(stream)”指的是所有数据通信通道之中，数据的起点和终点。信息的通道就是一个数据流。只要是数据从一个地方“流”到另外一个地方，这种数据流动的通道都可以称为数据流。
- 输入/输出是相对于程序来说的。程序在使用数据时所扮演的角色有两个：一个是源，一个是目的。若程序是数据流的源，即数据的提供者，这个数据流对程序来说就是一个“输出数据流”(数据从程序流出)。若程序是数据流的终点，这个数据流对程序而言就是一个“输入数据流”（数据从程序外流向程序)

输入/输出类

- 在java.io包中提供了60多个类（流）。
- 从功能上分为两大类：输入流和输出流。
- 从流结构上可分为字节流（以字节为处理单位或称面向字节）和字符流（以字符为处理单位或称面向字符）。
- 字节流的输入流和输出流基础是InputStream和OutputStream这两个抽象类，字节流的输入输出操作由这两个类的子类实现。字符流是Java 1.1版后新增加的以字符为单位进行输入输出处理的流，字符流输入输出的基础是抽象类Reader和Writer

字节流和字符流

- Java 2 定义了两种类型的流：字节流和字符流。字节流（byte stream）为处理字节的输入和输出提供了方便的方法。例如使用字节流读取或写入二进制数据。字符流（character stream）为字符的输入和输出处理提供了方便。它们采用了统一的编码标准，因而可以国际化。当然，在某些场合，字符流比字节流更有效

字节流和字符流

- Java的原始版本（Java 1.0）不包括字符流，因此所有的输入和输出都是以字节为单位的。Java 1.1中加入了字符流的功能
- 需要声明：在最底层，所有的输入/输出都是字节形式的。基于字符的流只为处理字符提供方便有效的方法

字节流和字符流

- **字节流类**（Byte Streams） 字节流类用于向字节流读写8位二进制的字节。一般地，字节流类主要用于读写诸如图象或声音等的二进制数据。
- **字符流类**（Character Streams） 字符流类用于向字符流读写16位二进制字符。

字节流和字符流

Byte Streams	Character Streams
InputStream	Reader
OutputStream	Writer

流的分类

- 两种基本的流是：输入流(Input Stream)和输出流(Output Stream)。可从中读出一系列字节的对象称为输入流。而能向其中写入一系列字节的对象称为输出流。

输入流

- 读数据的逻辑为：

  open a stream

  while more information

  read information

  close the stream

输出流

写数据的逻辑为：

open a stream

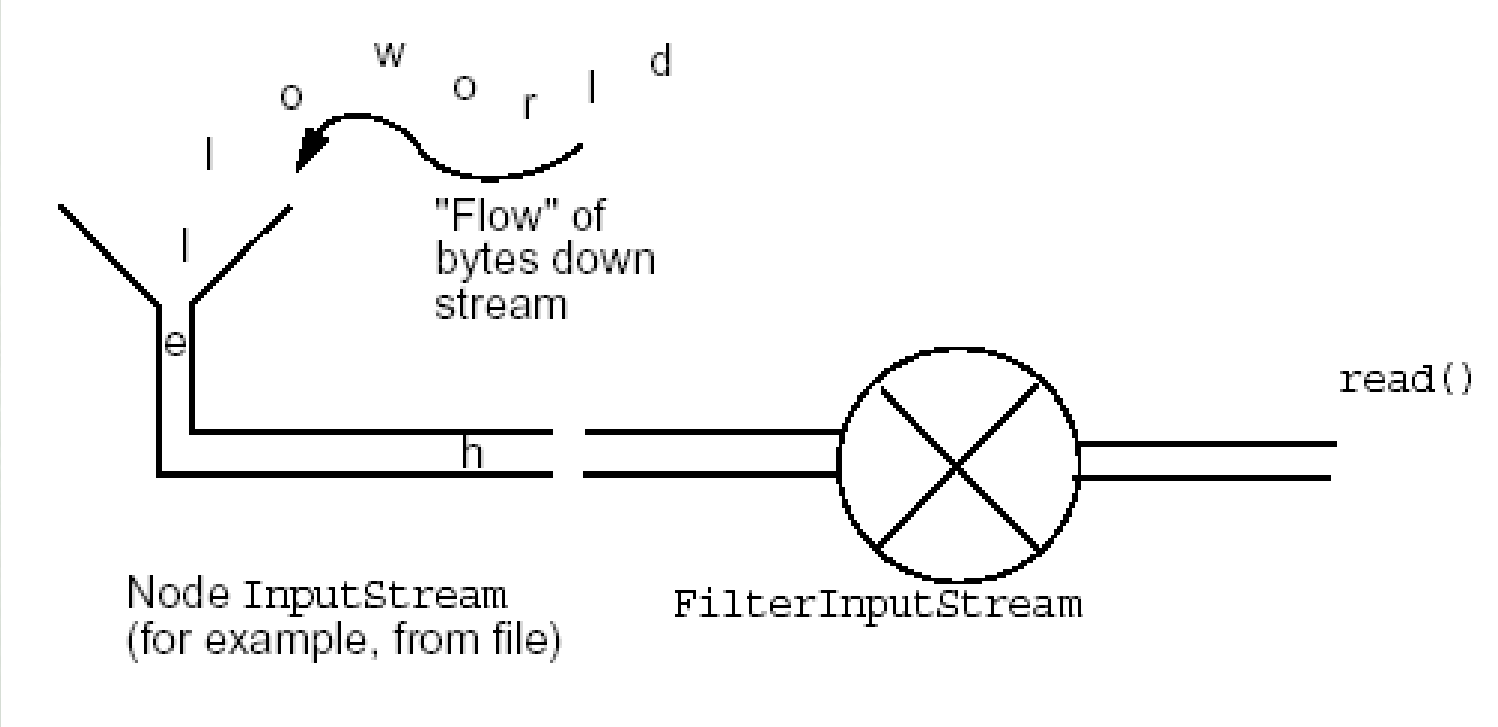
while more information

write information

close the stream

流的分类

- 节点流：从特定的地方读写的流类，例如：磁盘或一块内存区域。
- 过滤流：使用节点流作为输入或输出。过滤流是使用一个已经存在的输入流或输出流连接创建的。



字节流

- 字节流类为处理字节式输入/输出提供了丰富的环境。一个字节流可以和其他任何类型的对象并用，包括二进制数据。这样的多功能性使得字节流对很多类型的程序都很重要。
- 字节流类以InputStream 和OutputStream为顶层类,他们都是抽象类(abstract)

字节流

- 抽象类InputStream 和 OutputStream定义了实现其他流类的关键方法。最重要的两种方法是read()和write()，它们分别对数据的字节进行读写。两种方法都在InputStream 和OutputStream中被定义为抽象方法。它们被派生的流类重写。
- 每个抽象类都有多个具体的子类，这些子类对不同的外设进行处理，例如磁盘文件，网络连接，甚至是内存缓冲区。
- 要使用流类，必须导入java.io包

InputStream

- 三个基本的读方法

  abstract int **read**() ：读取一个字节数据，并返回读到的数据，如果返回-1，表示读到了输入流的末尾。

  int **read**(byte[] b) ：将数据读入一个字节数组，同时返回实际读取的字节数。如果返回-1，表示读到了输入流的末尾。

  int read(byte[] b, int off, int len) ：将数据读入一个字节数组，同时返回实际读取的字节数。如果返回-1，表示读到了输入流的末尾。off指定在数组b中存放数据的起始偏移位置；len指定读取的最大字节数。

InputStream

- 思考：为什么只有第一个read方法是抽象的，而其余两个read方法都是具体的？

InputStream

- 因为第二个read方法依靠第三个read方法来实现，而第三个read方法又依靠第一个read方法来实现，所以说只有第一个read方法是与具体的I/O设备相关的，它需要InputStream的子类来实现

InputStream

- 其它方法

  long **skip**(long n)：在输入流中跳过n个字节，并返回实际跳过的字节数。

  int **available**()：返回在不发生阻塞的情况下，可读取的字节数。

  void **close**()：关闭输入流，释放和这个流相关的系统资源。

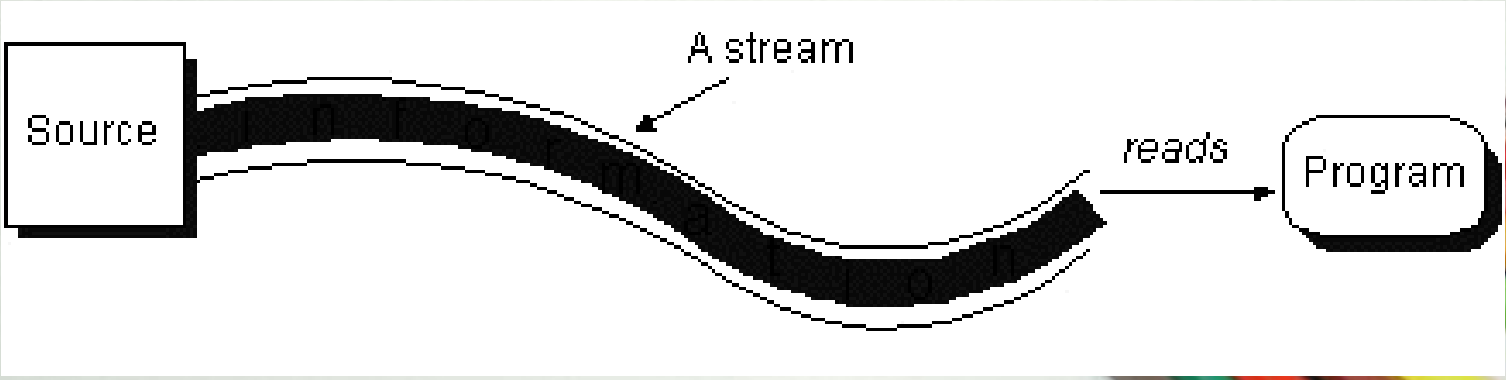
  void **mark**(int readlimit)：在输入流的当前位置放置一个标记，如果读取的字节数多于readlimit设置的值，则流忽略这个标记。

  void **reset**()：返回到上一个标记。

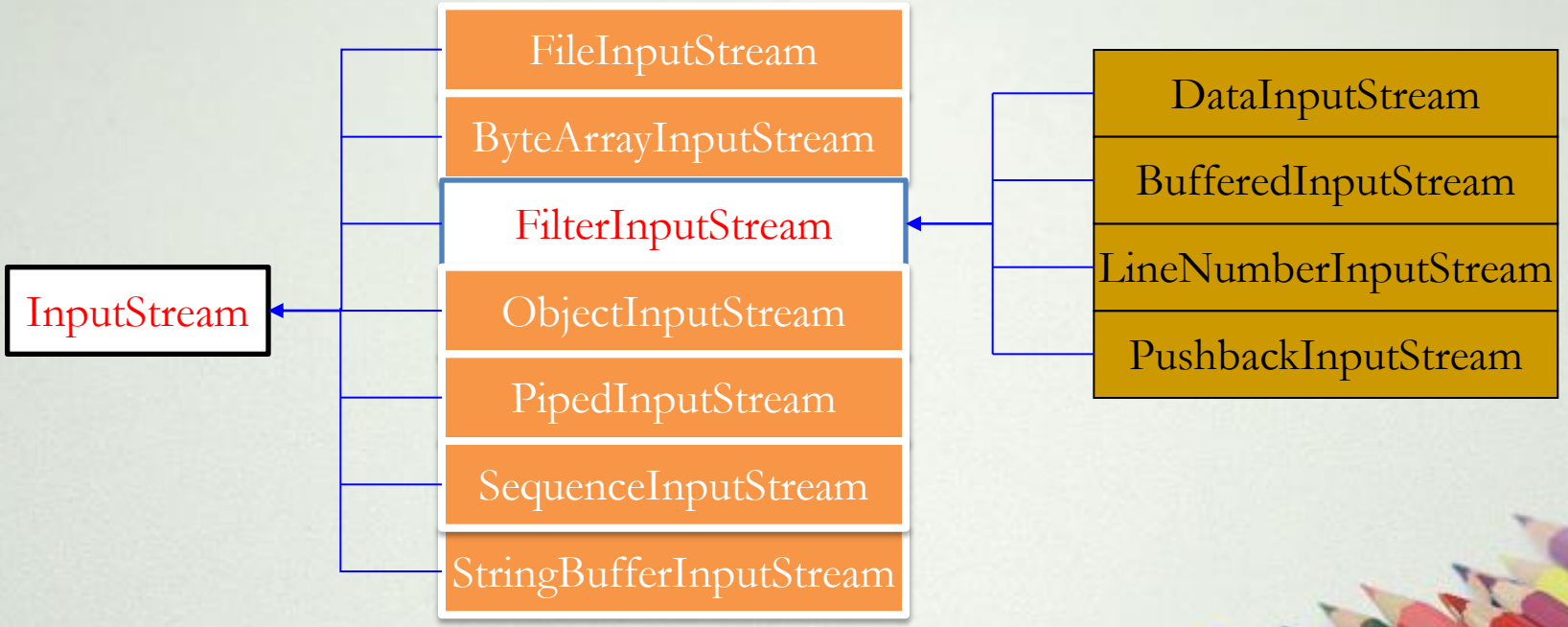
  boolean **markSupported**()：测试当前流是否支持mark和reset方法。如果支持，返回true，否则返回false。

InputStream

- 该类的所有方法在出错条件下引发一个 IOException 异常
- 通过打开一个到数据源（文件、内存或网络端口上的数据）的输入流，程序可以从数据源上顺序读取数据。



java.io包中 InputStream的类层次



InputStream

- InputStream中包含一套字节输入流需要的方法，可以完成最基本的从输入流读入数据的功能。当Java程序需要外设的数据时，可根据数据的不同形式，创建一个适当的InputStream子类类型的对象来完成与该外设的连接，然后再调用执行这个流类对象的特定输入方法来实现对相应外设的输入操作。
- InputStream类子类对象自然也继承了InputStream类的方法。常用的方法有：读数据的方法read()，获取输入流字节数的方法available()，定位输入位置指针的方法skip()、reset()、mark()等。

OutputStream

- 三个基本的写方法

  abstract void **write**(int b) ：往输出流中写入一个字节。

  void **write**(byte[] b) ：往输出流中写入数组b中的所有字节。

  void **write**(byte[] b, int off, int len) ：往输出流中写入数组b中从偏移量off开始的len个字节的数据

OutputStream

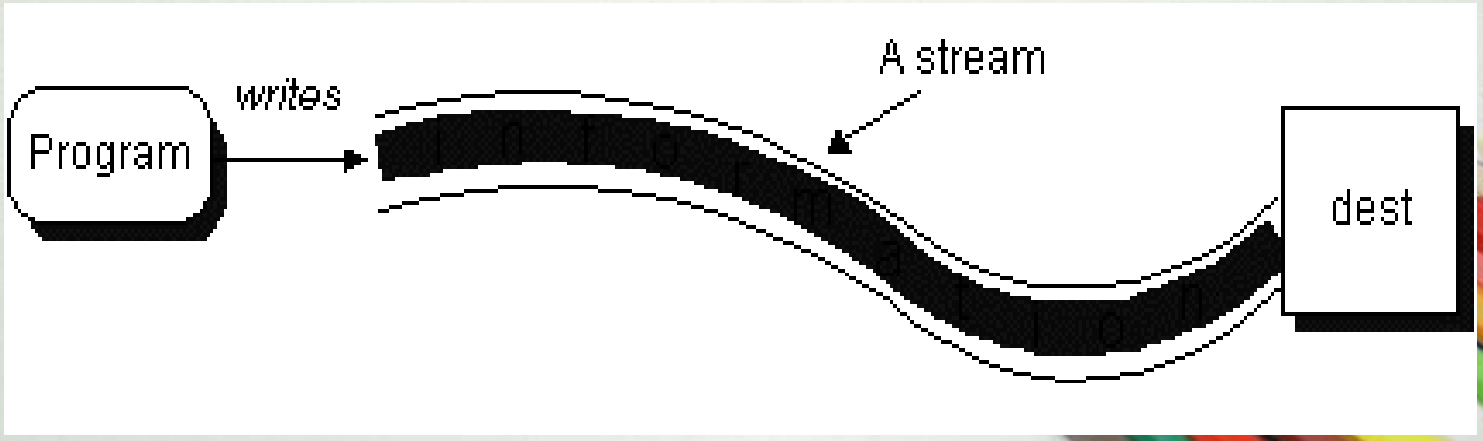
- 其它方法

  void **flush**() ： 刷新输出流，强制缓冲区中的输出字节被写出。

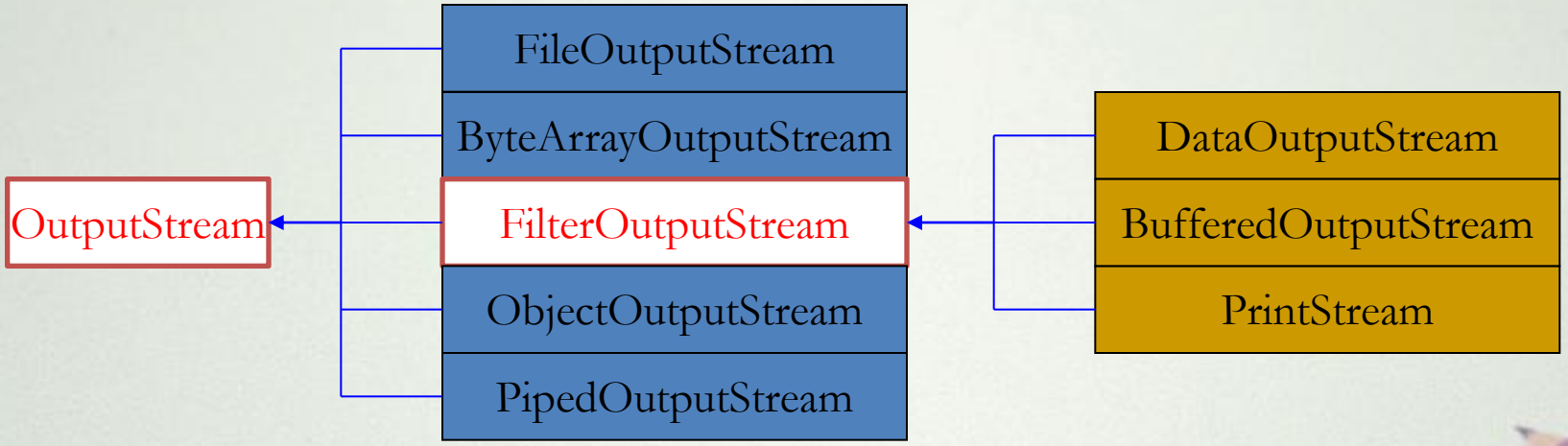
  void **close**() ： 关闭输出流，释放和这个流相关的系统资源。

OutputStream

- OutputStream是定义了流式字节输出模式的抽象类。该类的所有方法返回一个void 值并且在出错情况下引发一个IOException异常。
- 通过打开一个到目标的输出流，程序可以向外部目标顺序写数据



java.io包中 OutputStream的类层次



OutputStream

- OutputStream中包含一套字节输出流需要的方法，可以完成最基本的输出数据到输出流的功能。当Java程序需要将数据输出到外设时，可根据数据的不同形式，创建一个适当的OutputStream子类类型的对象来完成与该外设的连接，然后再调用执行这个流类对象的特定输出方法来实现对相应外设的输出操作。
- OutputStream类子类对象也继承了OutputStream类的方法。常用的方法有：写数据的方法write()，关闭流方法close()等。

过滤流

- 在InputStream类和OutputStream类子类中，FilterInputStream 和 FilterOutputStream 过滤流抽象类又派生出DataInputStream和DataOutputStream数据输入输出流类等子类。

过滤流

- 过滤流的主要特点是在输入输出数据的同时能对所传输的数据做指定类型或格式的转换，即可实现对二进制字节数据的理解和编码转换。
- 数据输入流DataInputStream中定义了多个针对不同类型数据的读方法，如readByte()、readBoolean()、readShort()、readChar()、readInt()、readLong()、readFloat()、readDouble()、readLine()等。
- 数据输出流DataOutputStream中定义了多个针对不同类型数据的写方法，如writeByte()、writeBoolean()、writeShort()、writeChar()、writeInt()、writeLong()、writeFloat()、writeDouble()、writeChars()等。

过滤流

- 过滤流在读/写数据的同时可以对数据进行处理，它提供了同步机制，使得某一时刻只有一个线程可以访问一个I/O流，以防止多个线程同时对一个I/O流进行操作所带来的意想不到的结果。
- 类FilterInputStream和FilterOutputStream分别作为所有过滤输入流和输出流的父类。

基本的流类

- FileInputStream和FileOutputStream

  节点流，用于从文件中读取或往文件中写入字节流。如果在构造FileOutputStream时，文件已经存在，则覆盖这个文件。

- BufferedInputStream和BufferedOutputStream

  过滤流，需要使用已经存在的节点流来构造，提供带缓冲的读写，提高了读写的效率。

- DataInputStream和DataOutputStream

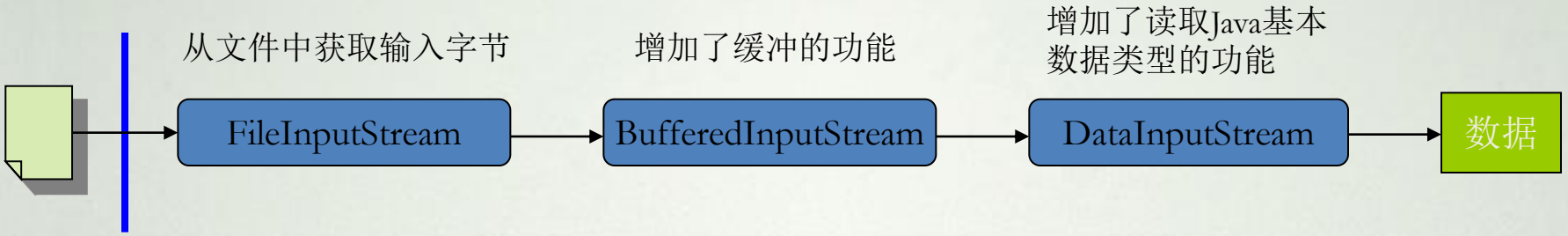
  过滤流，需要使用已经存在的节点流来构造，提供了读写Java中的基本数据类型的功能。

- PipedInputStream和PipedOutputStream

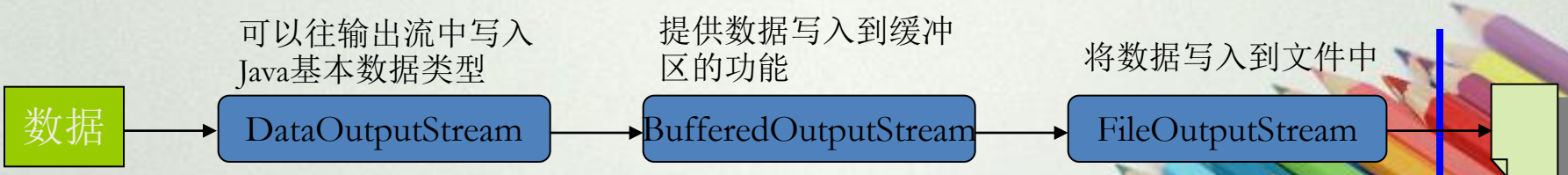
  管道流，用于线程间的通信。一个线程的PipedInputStream对象从另一个线程的PipedOutputStream对象读取输入。要使管道流有用，必须同时构造管道输入流和管道输出流。

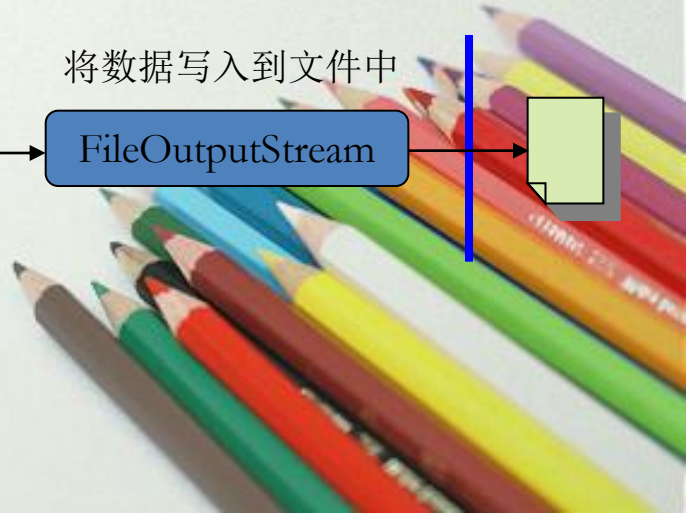
I/O流的链接

Input Stream Chain



Output Stream Chain





Java I/O库的设计原则

- Java的I/O库提供了一个称做链接的机制，可以将一个流与另一个流首尾相接，形成一个流管道的链接。这种机制实际上是一种被称为Decorator(装饰)设计模式的应用。
- 通过流的链接，可以动态的增加流的功能，而这种功能的增加是通过组合一些流的基本功能而动态获取的。
- 我们要获取一个I/O对象，往往需要产生多个I/O对象，这也是Java I/O库不太容易掌握的原因，但在I/O库中Decorator模式的运用，给我们提供了实现上的灵活性。

预定义流

- 为方便使用计算机常用的输入输出设备，各种高级语言与操作系统对应，都规定了可用的标准设备（文件）。所谓标准设备（文件），也称为预定义设备（文件），在程序中使用这些设备（文件）时，可以不用专门的打开操作就能简单的应用。一般地，标准输入设备是键盘，标准输出设备是终端显示器，标准错误输出设备也是显示器

预定义流

- 所有的Java程序自动导入java.lang包。该包定义了一个名为System的类，该类封装了运行时环境的多方面信息。例如，使用它的某些方法，你能获得当前时间和与系统有关的不同属性。System 同时包含三个预定义的流变量，in，out和err。这些成员在System中是被定义成public 和static型的，这意味着它们可以不引用特定的System对象而被用于程序的其他部分。

预定义流

- System.out是标准的输出流。默认情况下，它是一个控制台。System.in是标准输入，默认情况下，它指的是键盘。System.err指的是标准错误流，它默认是控制台。然而，这些流可以重定向到任何兼容的输入/输出设备
- System.in 是InputStream的对象；System.out和System.err是PrintStream的对象。它们都是字节流，尽管它们用来读写外设的字符。如果愿意，你可以用基于字符的流来包装它们

标准输入

- System类的类变量in表示标准输入流，其定义为:
  - public static final InputStream in
- 标准输入流已打开，作好提供输入数据的准备。一般这个流对应键盘输入，可以使用InputStream 类的read()和skip(long n)等方法来从输入流获得数据。read()从输入流中读一个字节，skip(long n)在输入流中跳过n个字节。

标准输出

- System类的类变量out表示标准输出流，其定义为：
  - public static final PrintStream out
- 标准输出流也已打开，作好接收数据的准备。一般这个流对应显示器输出。

- 可以使用PrintStream 类的print()和println()等方法来输出数据，这两个方法支持Java的任意基本类型作为参数。

标准错误输出

- System类的类变量err表示标准错误输出流，其定义为：
  - public static final PrintStream err
- 标准错误输出流已打开，作好接收数据的准备。一般这个流也对应显示器输出，与System.out一样，可以访问PrintStream类的方法。

预定义流

- 参见程序SystemTest.java
- 该程序将键盘输入的字符输出到屏幕，遇到字母q时程序终止

文件的读写

- Java提供了一系列的读写文件的类和方法。在Java中，所有的文件都是字节形式的。Java提供从文件读写字节的方法。而且，Java允许在字符形式的对象中使用字节文件流。
- 两个最常用的流类是FileInputStream和FileOutputStream，它们生成与文件链接的字节流。为打开文件，你只需创建这些类中某一个类的一个对象，在构造方法中以参数形式指定文件的名称
- 当你对文件的操作结束后，需要调用close( )来关闭文件。
  - void close( ) throws IOException

FileInputStream（文件输入流）

- FileInputStream 类创建一个能从文件读取字节的InputStream 类，它的两个常用的构造方法如下
  - FileInputStream(String filepath)
  - FileInputStream(File fileObj)
  - 它们都能引发FileNotFoundException异常。这里，filepath 是文件的全称路径，fileObj是描述该文件的File对象。

FileInputStream（文件输入流）

- 下面的例子创建了两个使用同样磁盘文件且各含一个上述构造方法的FileInputStream类：
  - FileInputStream f0 = new FileInputStream("c:\\file\\name.txt")
  - File f = new File(" c:\\file\\name.txt ");
  - FileInputStream f1 = new FileInputStream(f);

FileInputStream（文件输入流）

- 为读文件，可以使用在FileInputStream中定义的read( )方法。
  - int read( ) throws IOException
- 该方法每次被调用，它仅从文件中读取一个字节并将该字节以整数形式返回。当读到文件尾时，read( )返回−1。该方法可以引发IOException异常

FileInputStream（文件输入流）

- 参见程序 FileInputStream1.java
- 该程序用read()来输入和显示文本文件的内容，该文件名以命令行形式指定。注意try/catch块处理程序运行时可能发生的两个错误——未找到指定的文件或用户忘记包括文件名了。

FileInputStream（文件输入流）

- 参见程序FileInputStream2.java
- 该程序说明了怎样读取单个字节、字节数组以及字节数组的子集。它同样阐述了怎样运用available( )判定剩余的字节个数及怎样用skip( )方法跳过不必要的字节。该程序读取它自己的源文件，该源文件必定在当前目录中
- 这个有些刻意创作的例子说明了怎样读取数据的三种方法，怎样跳过输入以及怎样检查流中可以获得数据的数目。

FileOutputStream（文件输出流）

- FileOutputStream 创建了一个可以向文件写入字节的类OutputStream，它常用的构造方法如下
  - FileOutputStream(String filePath)
    FileOutputStream(File fileObj)
    FileOutputStream(String filePath, boolean append)
- 它们可以引发IOException或SecurityException异常。这里filePath是文件的全称路径，fileObj是描述该文件的File对象。如果append为true，文件以追加模式打开，即新写入的字节将附加在文件的末尾，而不是从头开始

FileOutputStream（文件输出流）

- FileOutputStream的创建不依赖于文件是否存在。在创建对象时FileOutputStream在打开输出文件之前创建它。这种情况下你试图打开一个只读文件，会引发一个IOException异常

FileOutputStream（文件输出流）

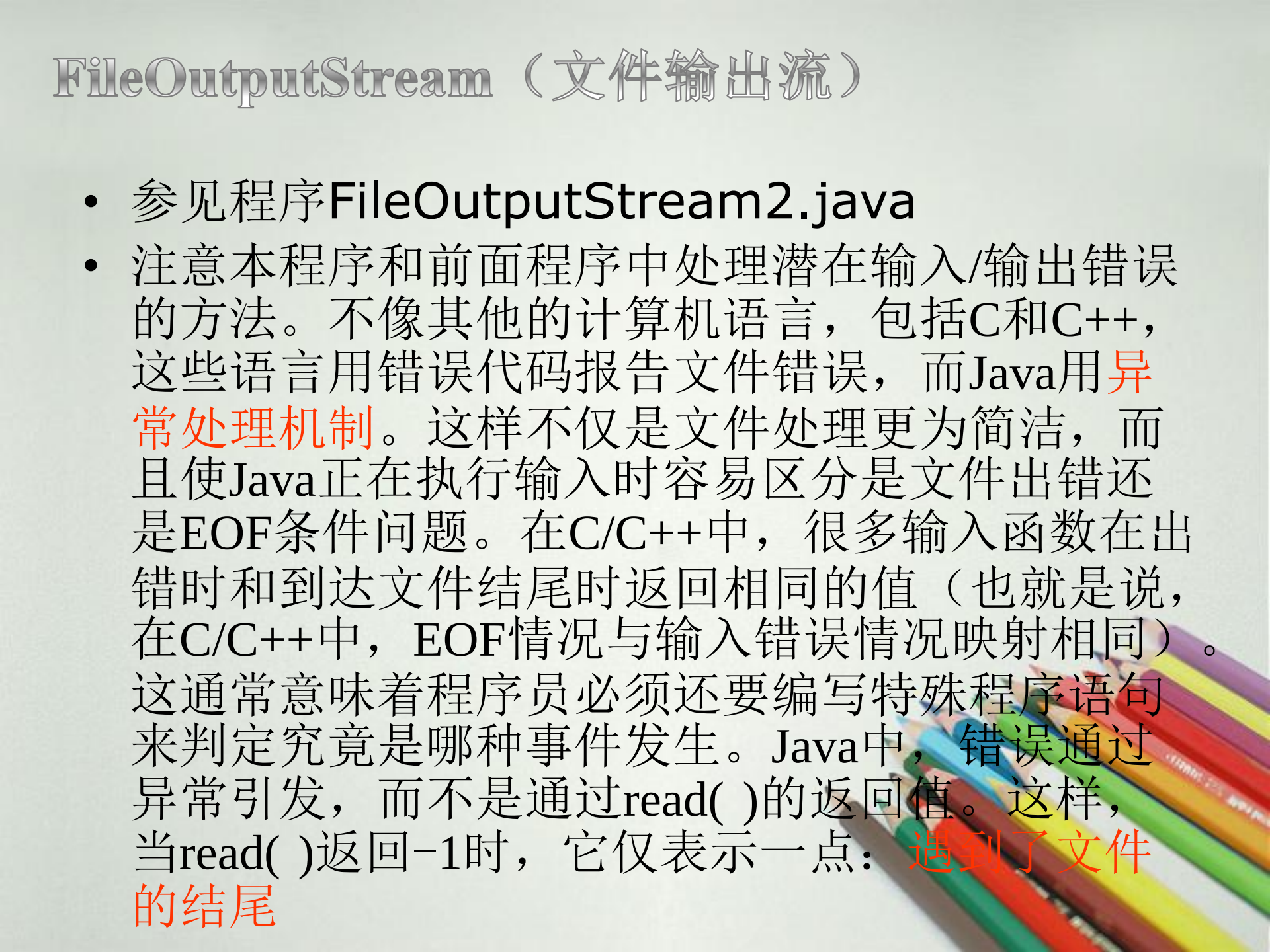
- 向文件中写数据，需用FileOutputStream定义的write()方法。它的最简单形式如下
  - `void write(int byteval) throws IOException`
  - 该方法按照byteval指定的数向文件写入字节。尽管byteval作为整数声明，但仅低8位字节可以写入文件。如果在写的过程中出现问题，一个IOException被引发

FileOutputStream（文件输出流）

- 参见程序 FileOutputStream1.java
- 该程序先生成一个String对象，接着用getBytes( ) 方法提取字节数组对等体。然后创建了三个文件。第一个file1.txt将包括样本中的偶数字节。第二个文件是file2.txt，它包括所有字节。第三个也是最后一个文件file3.txt，仅包含最后的四分之一。不像FileInputStream类的方法，所有FileOutputStream类的方法都返回一个void类型值。在出错情况下，这些方法将引发IOException异常
- 将FileOutputStream的构造方法改为追加模式，即第二个参数为true,重新运行该程序

FileOutputStream（文件输出流）

- 参见程序FileOutputStream2.java
- 注意本程序和前面程序中处理潜在输入/输出错误的方法。不像其他的计算机语言，包括C和C++，这些语言用错误代码报告文件错误，而Java用异常处理机制。这样不仅是文件处理更为简洁，而且使Java正在执行输入时容易区分是文件出错还是EOF条件问题。在C/C++中，很多输入函数在出错时和到达文件结尾时返回相同的值（也就是说，在C/C++中，EOF情况与输入错误情况映射相同）。这通常意味着程序员必须还要编写特殊程序语句来判定究竟是哪种事件发生。Java中，错误通过异常引发，而不是通过read( )的返回值。这样，当read( )返回-1时，它仅表示一点：遇到了文件的结尾



ByteArrayInputStream（字节数组输入流）

- ByteArrayInputStream是把字节数组当成源的输入流。该类有两个构造方法，每个构造方法需要一个字节数组提供数据源
  - ByteArrayInputStream(byte array[ ])
  - ByteArrayInputStream(byte array[ ], int start, int numBytes)
  - 这里，array是输入源。第二个构造方法创建了一个InputStream类，该类从字节数组的子集生成，以start指定索引的字符为起点，长度由numBytes决定

ByteArrayInputStream（字节数组输入流）

- 下面的例子创建了两个ByteArrayInputStream，用字母表的字节表示初始化它们

```
import java.io.*;

class ByteArrayInputStreamDemo {
public static void main(String args[]) throws IOException {
String tmp = "abcdefghijklmnopqrstuvwxyz";
byte b[] = tmp.getBytes();
ByteArrayInputStream input1 = new
   ByteArrayInputStream(b);
ByteArrayInputStream input2 = new
   ByteArrayInputStream(b, 0, 3);
}
}
```

input1对象包含整个字母表中小写字母，input2仅包含开始的三个字母。

ByteArrayInputStream（字节数组输入流）

- 参见程序ByteArrayInputStream1.java
- ByteArrayInputStream实现mark( )和reset( )方法。然而，如果 mark( )不被调用，reset( )在流的开始设置流指针——该指针是传递给构造方法的字节数组的首地址
- 该例先从流中读取每个字符，然后以小写字母形式打印。然后重新设置流并从头读起，这次在打印之前先将字母转换成大写字母

ByteArrayOutputStream（字节数组输出流）

- ByteArrayOutputStream是一个把字节数组当作输出流的实现。ByteArrayOutputStream 有两个构造方法
  - ByteArrayOutputStream( )
  - ByteArrayOutputStream(int numBytes)
  - 在第一种形式里，一个32位字节的缓冲区被生成。第二个构造方法生成一个numBytes大小的缓冲区。缓冲区保存在ByteArrayOutputStream的受保护的buf 成员里。缓冲区的大小在需要的情况下会自动增加。缓冲区保存的字节数是由ByteArrayOutputStream的受保护的count域保存的

ByteArrayOutputStream（字节数组输出流）

- 参见程序 ByteArrayOutputStream1.java
- 该例用 writeTo( )这一便捷的方法将f 的内容写入test.txt
  - **writeTo:**将此字节数组输出流的全部内容写入到指定的输出流参数中
  - **reset:**将此字节数组输出流的 count 字段重置为零，从而丢弃输出流中目前已累积的所有输出。通过重新使用已分配的缓冲区空间，可以再次使用该输出流

缓冲字节流

- 对于字节流，缓冲流（buffered stream），通过把内存缓冲区连到输入/输出流扩展一个过滤流类。该缓冲区允许Java对多个字节同时进行输入/输出操作，提高了程序性能。因为缓冲区可用，所以可以跳过、标记和重新设置流。缓冲字节流类是BufferedInputStream 和BufferedOutputStream。PushbackInputStream 也可实现缓冲流

缓冲字节流

- 若处理的数据量较多，为避免每个字节的读写都对流进行，可以使用过滤流类的子类缓冲流。缓冲流建立一个内部缓冲区，输入输出数据先读写到缓冲区中进行操作，这样可以提高文件流的操作效率。

BufferedInputStream（缓冲输入流）

- 缓冲输入/输出是一个非常普通的性能优化。Java的BufferedInputStream 类允许把任何InputStream类“包装”成缓冲流并使它的性能提高
- BufferedInputStream 有两个构造方法
  - BufferedInputStream(InputStream inputStream)
  - BufferedInputStream(InputStream inputStream, int bufSize)
  - 第一种形式**创建BufferedInputStream流对象并为以后的使用保存InputStream参数in，并创建一个内部缓冲区数组来保存输入数据。**
  - 第二种形式**用指定的缓冲区大小size创建BufferedInputStream流对象，并为以后的使用保存InputStream参数in。**

BufferedInputStream（缓冲输入流）

- 缓冲一个输入流同样提供了在可用缓冲区的流内支持向后移动的必备基础。除了在任何InputStream类中执行的read()和skip()方法外，BufferedInputStream 同样支持mark() 和reset()方法。
  BufferedInputStream.markSupported()返回true是这一支持的体现。
- 当创建缓冲输入流BufferedInputStream时，一个输入缓冲区数组被创建，来自流的数据填入缓冲区，一次可填入许多字节

BufferedInputStream（缓冲输入流）

- 参见程序 BufferedInputStream1.java
  - public void **mark**(int readlimit) ：在此输入流中标记当前的位置。对 reset 方法的后续调用会在最后标记的位置重新定位此流，以便后续读取重新读取相同的字节
  - public void **reset**() ：将此流重新定位到对此输入流最后调用 mark 方法时的位置

BufferedOutputStream（缓冲输出流）

- 缓冲输出流BufferedOutputStream类提供和FileOutputStream类同样的写操作方法，但所有输出全部写入缓冲区中。当写满缓冲区或关闭输出流时，它再一次性输出到流，或者用flush()方法主动将缓冲区输出到流。
- BufferedOutputStream与任何一个OutputStream相同，除了用一个另外的flush( ) 方法来保证数据缓冲区被写入到实际的输出设备。BufferedOutputStream通过减小系统写数据的时间而提高性能

BufferedOutputStream（缓冲输出流）

- 不像缓冲输入，缓冲输出不提供额外的功能，Java中输出缓冲区是为了提高性能的。下面是两个可用的构造方法
  - BufferedOutputStream(OutputStream outputStream)
  - BufferedOutputStream(OutputStream outputStream, int bufSize)
  - 第一种形式创建了一个使用512字节缓冲区的缓冲流。
  - 第二种形式，缓冲区的大小由bufSize参数传入。

BufferedOutputStream（缓冲输出流）

- 用flush()方法更新流
- 要想在程序结束之前将缓冲区里的数据写入磁盘，除了填满缓冲区或关闭输出流外，还可以显式调用flush()方法。flush()方法的声明为：
  - public void flush() throws IOException

BufferedOutputStream（缓冲输出流）

- 参见程序BufferedOutputStream1.java
  - 在调用完BufferedOutputStream后一定要flush或者将其close掉，否则缓冲区中的字节不会输出来

PushbackInputStream（推回输入流）

- 缓冲的一个新颖的用法是实现推回（pushback）。Pushback用于输入流允许字节被读取然后返回（即“推回”）到流。PushbackInputStream类实现了这个想法。它提供了一种机制来“窥视”在没有受到破坏的情况下输入流生成了什么。

PushbackInputStream（推回输入流）

- PushbackInputStream有两个构造方法
  - PushbackInputStream(InputStream inputStream)
  - PushbackInputStream(InputStream inputStream, int numBytes)
  - 第一种形式创建了一个允许一个字节推回到输入流的流对象。
  - 第二种形式创建了一个具有numBytes长度缓冲区的推回缓冲流。它允许多个字节推回到输入流

PushbackInputStream（推回输入流）

- 除了具有与InputStream相同的方法，PushbackInputStream提供了unread( )方法，表示如下
  - void unread(int ch)
  - void unread(byte buffer[ ])
  - void unread(byte buffer, int offset, int numChars)
  - 第一种形式推回ch的低位字节，它将是随后调用read( )方法所返回的下一个字节。
  - 第二种形式推回buffer缓冲器中的字节。
  - 第三种形式推回buffer中从offset处开始的numChars个字节。

PushbackInputStream（推回输入流）

- 参见程序 PushbackInputStream1.java
  - 该例子演示一个编程语言解析器怎样用PushbackInputStream和unread( )来处理==操作符和=操作符之间的不同的

SequenceInputStream（顺序输入流 ）

- SequenceInputStream类允许连接多个InputStream流。SequenceInputStream 表示其他输入流的逻辑串联。它从输入流的有序集合开始，并从第一个输入流开始读取，直到到达文件末尾，接着从第二个输入流读取，依次类推，直到到达包含的最后一个输入流的文件末尾为止

SequenceInputStream（顺序输入流 ）

- SequenceInputStream的构造不同于任何其他的InputStream。SequenceInputStream构造方法要么使用一对InputStream，要么用InputStream的一个Enumeration，显示如下：
  - SequenceInputStream(InputStream first, InputStream second)
  - SequenceInputStream(Enumeration streamEnum)

SequenceInputStream（顺序输入流 ）

- 操作上来说，该类满足读取完第一个InputStream后转去读取第二个流的读取要求。使用Enumeration的情况下，它将继续读取所有InputStream流直到最后一个被读完

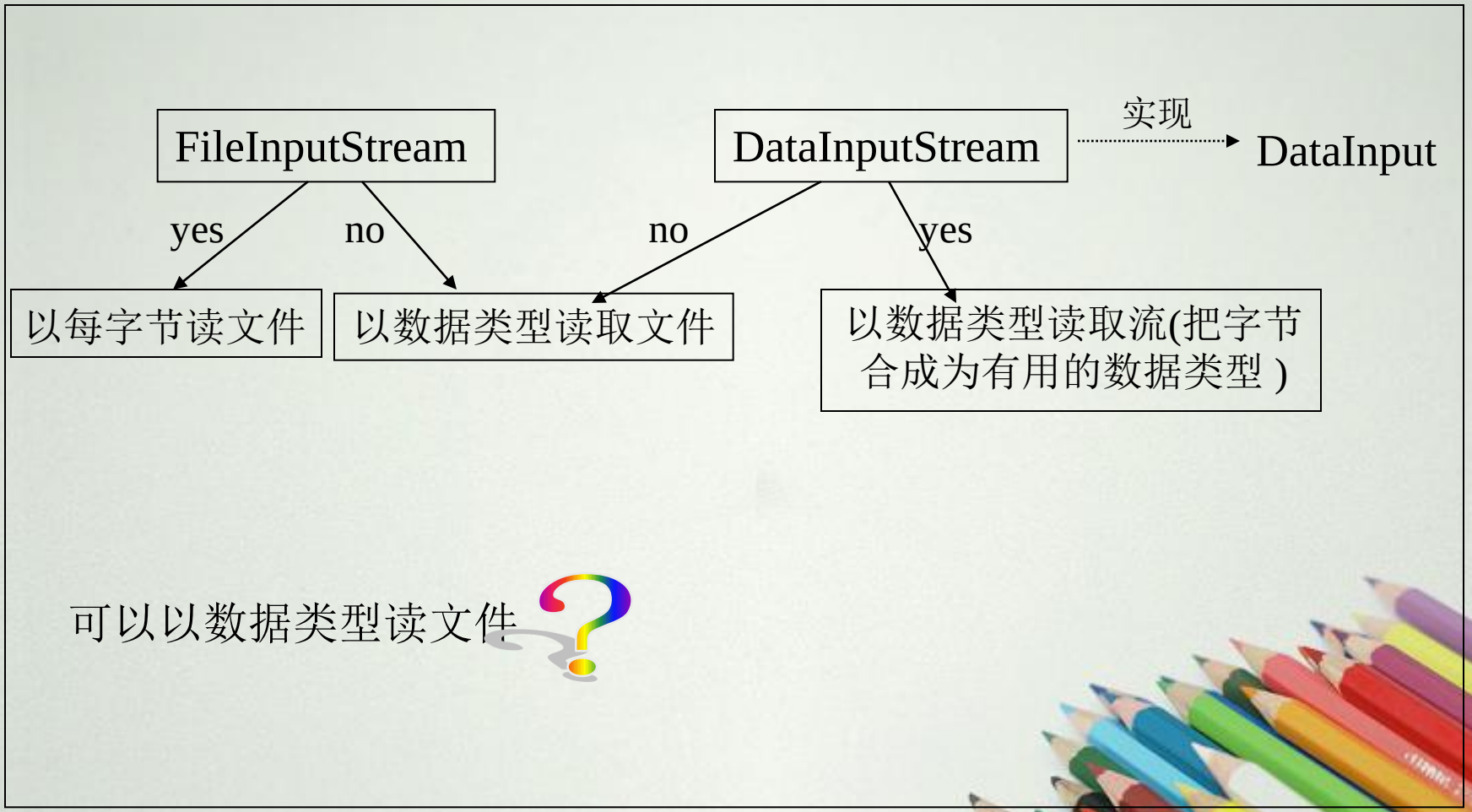
SequenceInputStream（顺序输入流）

- 参见程序 SequenceInputStream1.java
  - 该例创建了一个Vector向量并向它添加了两个文件名。它把名字向量传给InputStreamEnumerator类，设计该类是为了提供向量包装器，向量返回的元素不是文件名，而是用这些名称打开FileInputStream流。SequenceInputStream依次打开每个文件，该程序打印了两个文件的内容

DataInputStream与DataOutputStream

- 提供了允许从流读写任意对象与基本数据类型功能的方法。
- 字节文件流FileInputStream 和FileOutputStream只能提供纯字节或字节数组的输入/输出，如果要进行基本数据类型如整数和浮点数的输入/输出。则要用到过滤流类的子类二进制数据文件流DataInputStream 和DataOutputStream类。这两个类的对象必须和一个输入类或输出类联系起来，而不能直接用文件名或文件对象建立

流类分层


FileInputStream
DataInputStream
实现
DataInput
yes
no
no
yes
以每字节读文件
以数据类型读取文件
以数据类型读取流(把字节合成为有用的数据类型)
可以以数据类型读文件

流类分层

- 流类分层：把两种类结合在一起从而构成过滤器流，其方法是使用一个已经存在的流来构造另一个流。（Pattern Of Decorator）
- 比如：FileInputStream fin = new FileInputStream(“employee.dat”);
- DataInputStream din = new DataInputStream(fin);
- double s = din.readDouble();

流类分层

- 继承自FilterInputStream和FilterOutputStream的类，比如DataInputStream和BufferedInputStream，可以把它们结合进一个新的过滤流(也即继承自FilterInputStream和FilterOutputStream的类)中以构造你需要的流。如：DataInputStream din = new DataInputStream(new BufferedInputStream (new FileInputStream("employee.dat")));

流类分层

- 上面的例子说明了我们需要使用DataInputStream的方法，并且这些方法还要使用缓冲的read方法。
- 就如同上面的例子，可以分层构造流直到得到需要的访问功能为止。
- **实际上这就是装饰模式的最佳实践**

DataInputStream与DataOutputStream

- 使用数据文件流的一般步骤
  - (1)建立字节文件流对象;
  - (2)基于字节文件流对象建立数据文件流对象;
  - (3)用流对象的方法对基本类型的数据进行输入/输出。

DataInputStream与DataOutputStream

- DataInputStream类的构造方法如下
  - DataInputStream(InputStream in)创建过滤流FilterInputStream对象并为以后的使用保存InputStream参数in。
- DataOutputStream类的构造方法如下
  - DataOutputStream(OutputStream out)**创建输出数据流对象,写数据到指定的**OutputStream

DataInputStream与DataOutputStream

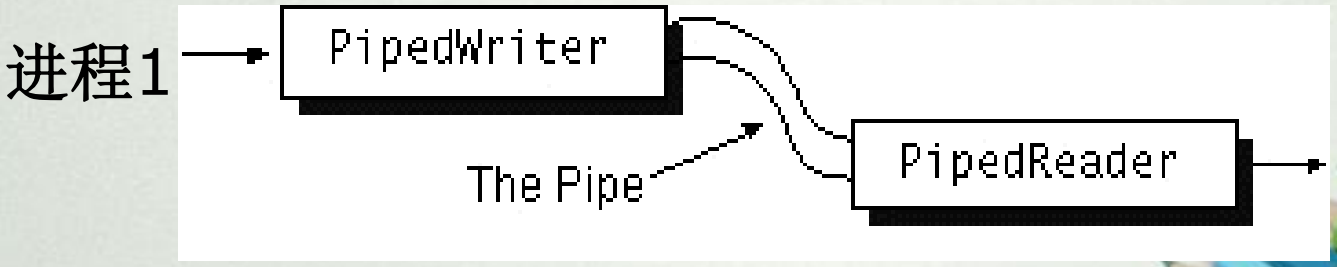
- 参见程序DataStream1.java
  - 由于DataOutputStream写入的为二进制信息，所以我们无法使用记事本查看内容
- 参见程序DataStream2.java
  - 该程序使用DataInputStream将刚才写入文件的二进制信息读取出来显示在命令行中
  - 注意：读取的顺序一定要与写入的顺序完全一致，否则会发生错误

DataInputStream与DataOutputStream

- 参见程序DataStream3.java
  - 在当前目录下建立文件fibonacci.dat，存储Fibonacci数列的前20个数。Fibonacci数列的前两个数是1，从第三个数开始，是其前两个数之和。即1,1,2,3,5,8,13,21,.…。
- 参见程序DataStream4.java

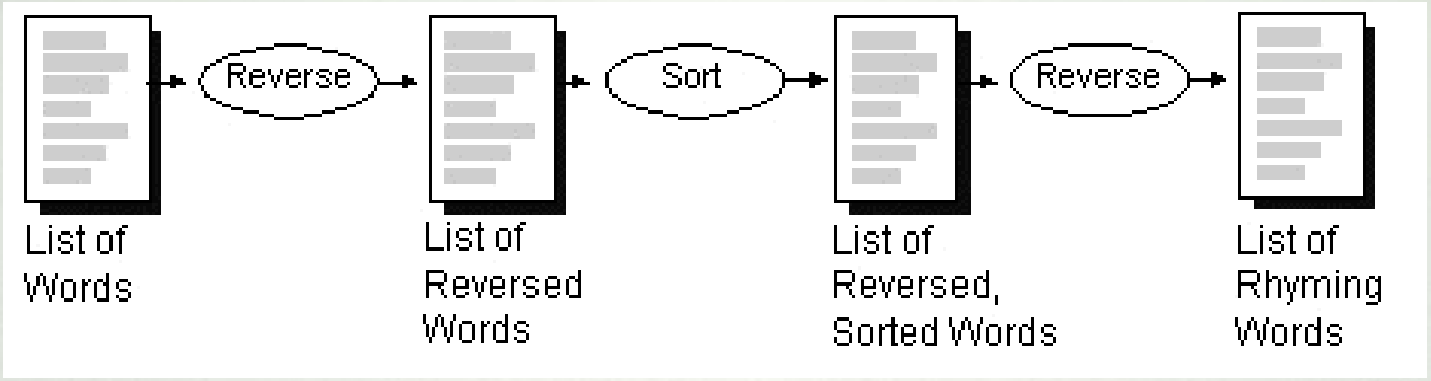
管道流

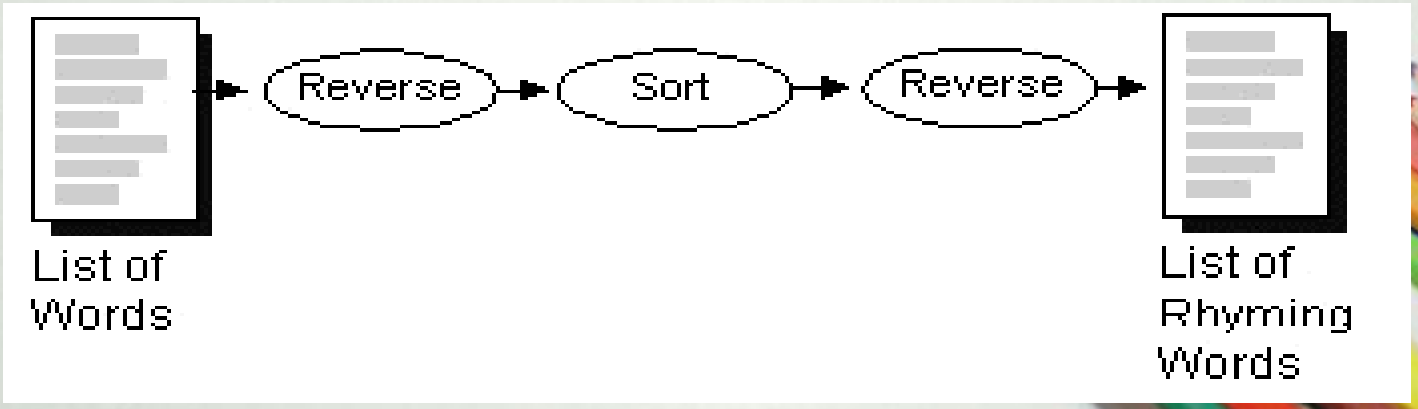
- 信息管道本身是虚拟的，但为线程或程序提供了通过流进行通信的功能。



管道流

- 使用管道的好处：


Reverse
Sort
Reverse
List of Words
List of Reversed Words
List of Reversed, Sorted Words
List of Rhyming Words


Reverse
Sort
Reverse
List of Words
List of Rhyming Words

PipedInputStream 与PipedOutputStream

管道流，用于线程间的通信。一个线程的PipedInputStream对象从另一个线程的PipedOutputStream对象读取输入。要使管道流有用，必须同时构造管道输入流和管道输出流。

PipedInputStream 与PipedOutputStream总是成对出现的

管道流继承自InputStream与OutputStream，所以他们并不是过滤流

PipedInputStream 与PipedOutputStream

- PipedInputStream
  - 传送输入流应该连接到传送输出流；传送输入流会提供要写入传送输出流的所有数据字节。通常，数据由某个线程从 PipedInputStream 对象读取，并由其他线程将其写入到相应的 PipedOutputStream。不建议对这两个对象尝试使用单个线程，因为这样可能会死锁该线程。传送输入流包含一个缓冲区，可在缓冲区限定的范围内将读操作和写操作分离开

PipedInputStream 与PipedOutputStream

- PipedOutputStream
  - 传送输出流可以连接到传送输入流，以创建通信管道。传送输出流是管道的发送端。通常，数据由某个线程写入 PipedOutputStream 对象，并由其他线程从连接的 PipedInputStream 读取。不建议对这两个对象尝试使用单个线程，因为这样可能会死锁该线程

PipedInputStream 与PipedOutputStream

- **PipedInputStream** 构造方法
  - PipedInputStream()
    - 创建尚未连接的 PipedInputStream
  - PipedInputStream(PipedOutputStream src)
    - 创建 PipedInputStream，以使其连接到传送输出流 src
- public void **connect**(PipedOutputStream src)
  - 使此传送输入流连接到传送输出流 src。如果此对象已经连接到其他某个传送输出流，则抛出 IOException

PipedInputStream 与PipedOutputStream

- 如果 src 为未连接的传送输出流，snk 为未连接的传送输入流，则可以通过以下任一调用使其连接：
- snk.connect(src)
  - 或：
  - src.connect(snk) 这两个调用的效果相同。

PipedInputStream 与PipedOutputStream

- **PipedOutputStream** 构造方法
  - PipedOutputStream()
    - 创建尚未连接到传送输入流的传送输出流
  - PipedOutputStream(PipedInputStream snk)
    - 创建连接到指定传送输入流的传送输出流。
- public void **connect**(PipedInputStream snk)
  - 将此传送输出流连接到接收者。如果此对象已经连接到其他某个传送输入流，则抛出 IOException。

PipedInputStream 与PipedOutputStream

- 如果 snk 为未连接的传送输入流，而 src 为未连接的传送输出流，则可以通过以下任一调用使其连接：
- src.connect(snk)
- 或：
- snk.connect(src)
- 这两个调用的效果相同。

PipedInputStream 与PipedOutputStream

- 参见程序 PipedStream1.java
  - 该程序中我们可以在构造方法中就将管道输入输出流连接起来
- 参见程序 PipedStream2.java
- 参见程序 PipedStream3.java
  - public final void writeUTF(String str) throws IOException
  - 解释

PipedInputStream 与PipedOutputStream

- 以与机器无关方式使用 UTF-8 修改版编码将一个字符串写入基础输出流
  - 首先，通过 writeShort 之类的方法将两个字节写入输出流，表示后跟的字节数。该值是实际写出的字节数，不是字符串的长度。根据此长度，使用字符的 UTF-8 修改版编码按顺序输出字符串的每个字符。如果没有抛出异常，则计数器 written(protected int written) 增加写入输出流的字节总数。该值至少是 2 加 str 的长度，最多是 2 加 str 的三倍长度

PipedInputStream 与PipedOutputStream

- 本程序实现了将两个线程的管道输入流和管道输出流串连起来的目的，从而实现了两个线程之间的通信

实现我们自己的I/O流

- 观察InputStream抽象类，它所定义的唯一抽象方法便是
  - public abstract int read() throws IOException;
- 而其它相关用以读取的方法还有（它们不是抽象的， 也就是说事实上InputStream已提供了实现
  - public int read(byte b[]) throws IOException;
    public int read(byte b[], int off, int len) throws IOException;

实现我们自己的I/O流

- 这两个同名异式的版本（方法重载），前者用來从此InputStream中读取最多b.length bytes的 数据储存于b中，并且返回所读取的数目；而后者则用來从此InputStream中读取最多len bytes的数据，並且从b这个byte数组索引值为off之处开始儲存起， 同樣返回所讀取的數目
- 很显然的，InputStream把如何提取数据給读取者的实现部份加以抽离,不同的InputStream实现可能都有不同的实现方式。

实现我们自己的I/O流

- 先让我提一个问题。那为什么InputStream的设计者不將所有的read()方法皆 定义為抽象，而单单只將int read()這一個版本定义為抽象呢？

实现我们自己的I/O流

- 很显然的，這三個版本的read()方法其实是可以互相利用 彼此來实现的。比方說，int read(byte b[])的实现其实就可以写成
  - return read(b, 0, b.length);

实现我们自己的I/O流

- 事实上，在Sun所提供的原始程序中，我们也是看到同样的結果。使用第 三個版本read(byte b[], int off, int len)來加以实现；而第三個版本还可以使用第 一個版本的read()來实现自己。我们也不是不可以用第二個版本或是第三個版 本來实现第一個版本，例如
  - byte b[] = new byte[1];
  - if(read(b) < 0) return –1;
  - return b[0];

实现我们自己的I/O流

- 但是，显然InputStream的設計者，选择让int read()成為最後被抽離的對象。而 讓第二版本的read()依赖第三個版本的read()，而讓第三個版本的read()建构在 第一個版本的read()之上。所以，第一個版本的read()就成了此抽象類別唯一的 抽象方法。在某種程度上來說，它是唯一和具体实现會有相关的部份。

动手写一个InputStream

- 我們知道有個名为java.io.ByteArrayInputStream的類別，它允許我們以流 （stream）的方式依序地讀取一個byte array中的內容。假設，我們現在要实现 自己的ByteArrayInputStream，那麼因為它勢必继承Inputstream類別，所以免 不了要实现那唯一的抽象方法：int read()。

动手写一个InputStream

- 先來看看构造方法（constructor），假設我們可以在构造它時傳入一個byte array（java.io.ByteArrayInputStream提供的是更強的功能，在此处我們先只要实现这樣子就好了）：
  - public MyByteArrayInputStream(byte b[])

动手写一个InputStream

- 那麼我們使用一個名为data的成員变量來儲存傳入的byte array。 並且因為串流的性質，我們必須記錄目前已讀取到的位置，所以用int ptr這個 成員变量來加以表示。因此，到目前為止，除了int read()之外，我們已有的实现大概會像是這樣：

动手写一个InputStream

- public class MyByteArrayInputStream extends java.io.InputStream
- { protected byte data[];
- protected int ptr = 0;
- public MyByteArrayInputStream(byte b[])
- {
- data = b;
- }
- }
- 那int read()又該如何实现呢？

动手写一个InputStream

- public int read()
- {
- return (ptr < data.length) ? (data[ptr++]) : -1;
- }

- 参见程序 MyOwnStream1.java

动手写一个InputStream

- 可以看到，不管是直接通过我們所实现的int read()來加以读取，或者是通过InputStream本 身对第二個版本之read()的实现（第二個版本调用第三個版本，而第三個版本 再调用第一個版本），我們都順利地读取到所传入byte array的值

动手写一个InputStream

- 这样子來实现，的确使得MyByteArrayInputStream看起來运作的很顺畅。但是 似乎就是有一些事情不太对劲。比方說，我們在程序中先以byte b[]來构造MyByteArrayInputStream的对象，並由mbais指向該对象。然後依序地以一個 循环读取其中的內容。因為串流的性質，當我們讀取完所有的內容時，我們若 是再调用read()將會得到-1，因為实现中的ptr值已经大于等于data.length的值 了。所以在上述的程序中，我們重新以byte b[]构造了一個新的 MyByteArrayInputStream的对象，並且还是以mbais指向該对象，以供我們下一次的讀取。

动手写一个InputStream

- 而事实上，InputStream中提供了reset()這個方法，使调用可以重設此 InputStream中的讀取位置。當然，前提是，這個InputStream要能夠被重設讀取位置。而在上述的实现中，我們完全沒有覆写（override）掉所继承自InputStream 中的reset()方法，因而，它所表現出來的行为便是InputSteram之reset()預設的 行为，那么也就是一什么都不做。事实上，我們所欠缺的還不只是reset()方法。 以下列出在InputStream的定义中，我們還需要再加以实现的方法
  - 除了抛出 IOException 之外，类 InputStream 的方法 reset 不执行任何操作。

动手写一个InputStream

- public int available() throws IOException;
- public void close() throws IOException;
- public synchronized void mark(int readlimit);
- public synchronized void reset() throws IOException
- public boolean markSupported();
- 所以，我们应该为我们的MyByteArrayInputStream补足这些部份，才可以让 它成为一個完备的InputStream子类

动手写一个InputStream

- 参见程序MyOwnStream2.java