# IO

在IO流中导入的包都是**java.io**包。

### ⑴流的概念及作用

流是一组有顺序的，有起点和终点的字节集合，是对数据传输的总称或抽象。数据在两设备间的传输称为流，**流的本质是数据传输，根据数据传输特性将流抽象为各种类，方便直观的进行数据操作。**

**流的作用：为数据源和目的地建立一个输送通道。**

### ⑵流的特性：

**①先进先出：**最先写入输出流的数据最先被输入流读取到。

**②顺序存取：**可以一个接一个地往流中写入一串字节，读出时也将按写入顺序读取一串字节，不能随机访问中间的数据。**(RandomAccessFile类可以从文件的任意位置进行存取（输入输出）操作)。**

**③只读或只写：**每个流只能是输入流或输出流的一种，不能同时具备两个功能，输入流只能进行读操作，对输出流只能进行写操作。

## 一、File类

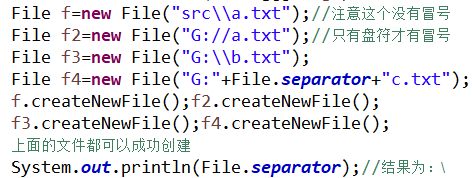


**File类只能修改或创建文件、文件夹的名，不能修改文件内部的内容。即File类只能针对文件或目录**。

在创建目录时，目录的中间分隔符可以是双斜杠，也可以是双反斜杠，单斜杠代表转义字符，也可以使用File.separator设置分隔符。

**File.separator**：代表系统分隔符**\**。

**File.pathSeparator**：表示分隔“**；**”。

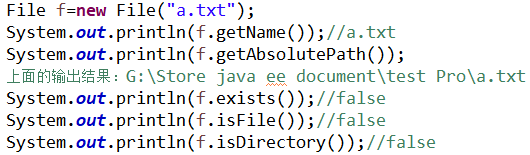




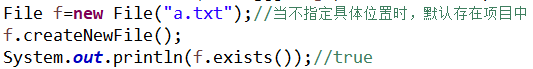
### 1.File类常用的构造器

**⑴File file=new File(String pathname)**

只有调用了File类的createNewFile()方法或者mkdir()方法才能创建出实际的文件或目录。括号中的参数若没有指定具体位置，则创建的文件或文件夹默认存放在项目中。若在项目中或者盘中实际存在一个文件如在项目src目录下存在一个Library.xml文件， File file=new File(“src\\Library.xml”);这个file就代表Library.xml文件。



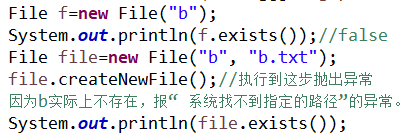
**例子：**

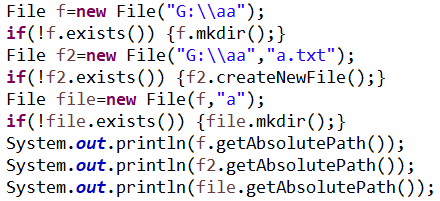


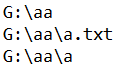
**⑵File file=new File(String parent,String child)**

**⑶File file=new File(File parent,String child)**

这两个构造器的用法类似，若第一个参数对应的文件或目录实际上不存在，则直接创建这个file文件或目录时，会抛异常。







### 2.File类中常用的方法

File**.exists()**：判断文件或文件夹(目录)是否存在，返回值为boolean类型。

File**.createNewFile()**：创建文件，注意**只能创建文件**，不能创建文件夹。返回值为boolean类型。如果文件名不存在并且成功创建了，返回值则为true。当返回值为false时，说明该文件名已经存在或者说创建文件失败。

File**.mkdir()**：创建文件夹，注意**只能创建文件夹**。该方法只能创建一个文件夹，若需要创建多级目录则要使用File**.mkdirs()**。创建成功返回值为true，否则为false。

File**.delete()**：删除文件或文件夹。若成功删除，则返回true，否则返回false。

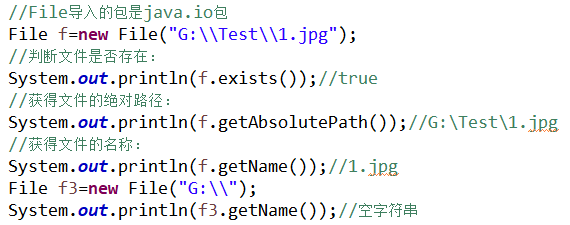
File**.getAbsolutePath()**：获得文件或目录的绝对路径，返回值为String类型。

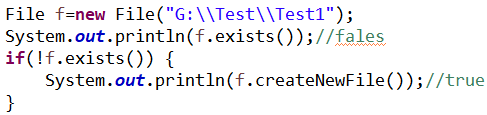
File**.getName()**：返回值为String类型，该方法是返回路径名中的最后一个名称，也就是返回文件夹名或文件名。当该路径名只有盘符名时，返回的是空串。

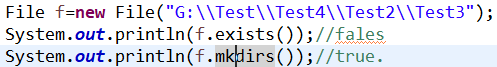
File**.getParent()**：返回值为String类型，返回的是这个路径中上一层的路径名称，如果这个路径名没有指定的上一层目录的名称，则返回null。

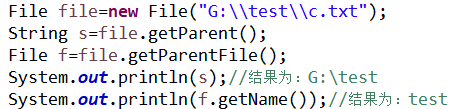
File**.getParentFile()**：返回值为File类型，返回的是这个路径中上一层的路径所对应的目录文件，若这个路径名没有指定的上一层目录，则返回null。

File**.isHiddern()**：返回值为boolean类型，用于判断文件或目录是否隐藏。









File**.list()**：以String[] 的形式返回当前路径下的子文件和子文件夹，但不包含子文件夹中的内容。

File**.listFiles()**：以File[]数组的形式，返回当前路径下的子文件或子文件夹。**list()和listFiles()的区别在于list()返回的是String类型，而listFiles()返回的是File类型，也就是说listFiles返回的可以使用File的各类方法，如getName()方法等**。

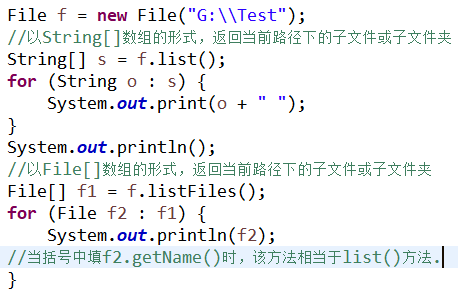
File**.isFile()**：返回值为boolean类型。**当返回true时表示，在当前路径下的该文件存在并且是一个文件(注意不是文件夹)**。返回false时，说明该路径下该文件不存在或者是一个文件夹，而不是文件。

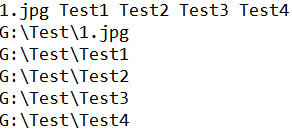
File**.isDirectory()**：判断是否是一个目录，若是目录则返回true，若不是目录或者目录不存在则返回false。

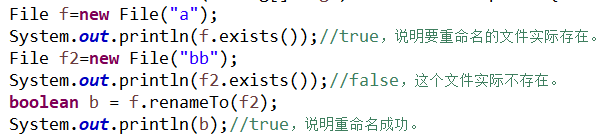
File**.length()**：返回值为long类型的整型。返回的是这个文件内容的字节大小。

File**.lastModified()**：返回值为long类型。返回的是文件最后一次修改之后距离1970年0时0分0秒之间的毫秒数。

File**.renameTo(File dest)**：返回值为boolean类型，该方法是用来重命名文件，括号内的参数为一个抽象的文件，实际上要不存在。若成功的重命名了文件，则返回true，否则返回false。







### 3.文件过滤和遍历

#### ⑴过滤

**①**File**.listFiles(FileFilter filter)**：括号内的参数为过滤器的名称。该方法用于过滤得到所需要的文件或者文件夹。java.io中的**文件过滤器FileFilter是一个接口，不能创建对象，因此采用创建对象的方式将ff填到该方法中是行不通的**。。但是可通过匿名内部类的方式在括号中填入过滤器。

**要特别注意这里的FileFilter是java.io.FileFilter包中的FileFilter接口，而不是javax.swing.fileChooser包中的FileFilter抽象类**。**这个导包一定要注意别导错了**。如下所示：

**java.io.FileFilter包中的FileFilter接口的源码如下**：

**package** java.io;

@FunctionalInterface

**public** **interface** FileFilter {...}

**boolean** accept(File pathname)：这个方法用于测试具体的抽象路径名**是否应该**包含在路径列表中。参数为要测试的File类型的路径名，若路径名应该要包含在路径列表中则返回true，否则返回false。

这是一个功能接口，要想实现该接口的功能，必须重写该接口的accept()方法。

**javax.swing.fileChooser包中FileFilter抽象类的源码如下：**

**package** javax.swing.filechooser;

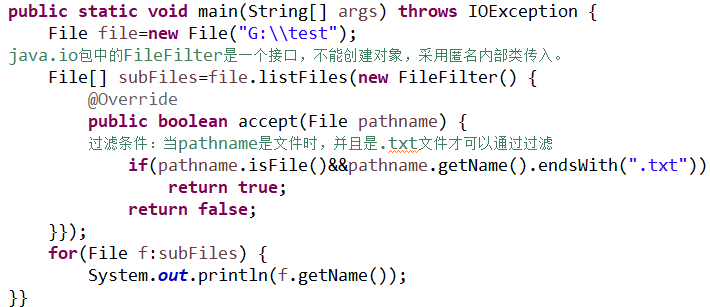
**public** **abstract** **class** FileFilter {...}

**public** **abstract** **boolean** accept(File f);

**public** **abstract** String getDescription();

若File.listFile(FileFilter filter)中导入的是javax.swing.fileChooser包中的FileFilter则在编译时会报错。

**例子：**





**②**File**.listFiles(FilenameFilter filter)**：参数为一个FilenameFilter类型的过滤器。

**FilenameFilter是一个功能接口，其源码如下**：

**package** java.io;

@FunctionalInterface

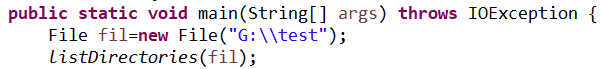
**public** **interface** FilenameFilter {....}

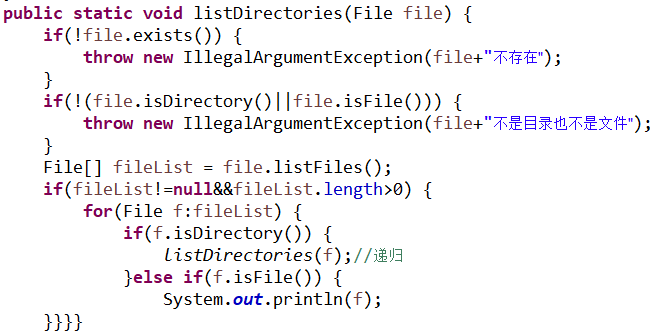
**boolean** accept(File dir, String name)：这个方法用来测试具体的文件**是否应该**包含在一个文件列表中。dir参数是一个目录，这个目录中包含了这个文件，name参数为这个文件的String类型的名称。只有当这个文件应该要包含在这个目录中时，才返回true，否则返回false。

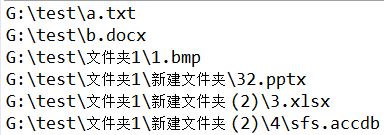
例子：

#### ⑵遍历

**查询文件夹下所有的文件**，例子：

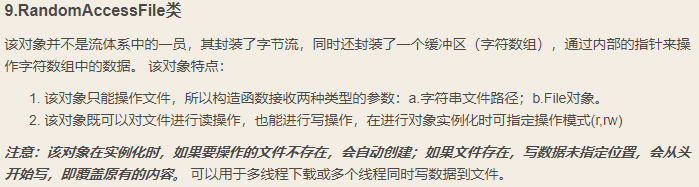






## 二、RandomAccessFile类

有空再学习



## 三、IO流分类

### 1.分类

按**方式或者数据流向**可分为**：输出流(写)、输入流(读)。输入流就是从数据源中读取数据到程序中，输出流就是从程序中将数据输出到外界**。

按**方向或者处理数据类型**可分为：**字节流、字符流**。其中，**字节流是一次读一个字节**，即8位，**一般读取视频、音频、图片等较快**，因为这些存储的都是二进制类型。而**字符流则一般读取文本文档较快**。

字节流一般以**InputStream、OutputStream**结尾。而字符流一般以**Reader、Writer**结尾。**InputStream和Reader是所有输入流的父类，OutputStream和Writer则是所有输出流的父类**。

### 2.java IO中常用的类



**java中常用的输入输出流分类表**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **分类** | **字节输入流** | **字节输出流** | **字符输入流** | **字符输出流** |
| **基类** | ***InputStream*** | ***OutputStream*** | ***Reader*** | ***Writer*** |
| **文件流** | FileInput  Stream | FileOutput  Stream | FileReader | FileWriter |
| **缓冲流** | BufferedInput  Stream | BufferedOutput  Stream | BufferedReader | BufferedWriter |
| **数据流** | DataInput  Stream | DataOutput  Stream |  |  |
| **对象流** | ObjectInput  Stream | ObjectOutput  Stream |  |  |
| **打印流** |  | PrintStream |  | PrintWriter |
| **转换流** |  |  | InputStream  Reader | OutputStream  Writer |
| **数组** | ByteArrayInput  Stream | ByteArrayOutpt  Stream | CharArray  Reader | CharArray  Writer |
| **管道** | PipedInput  Stream | PipedOutput  Stream | PipedReader | PipedWriter |
| **字符串** |  |  | StringReader | StringWriter |
|  | FilterInput  Stream | FilterOutput  Stream | ***FilterReader*** | ***FilterWriter*** |
|  | PushbackInput  Stream |  | Pushback  Reader |  |

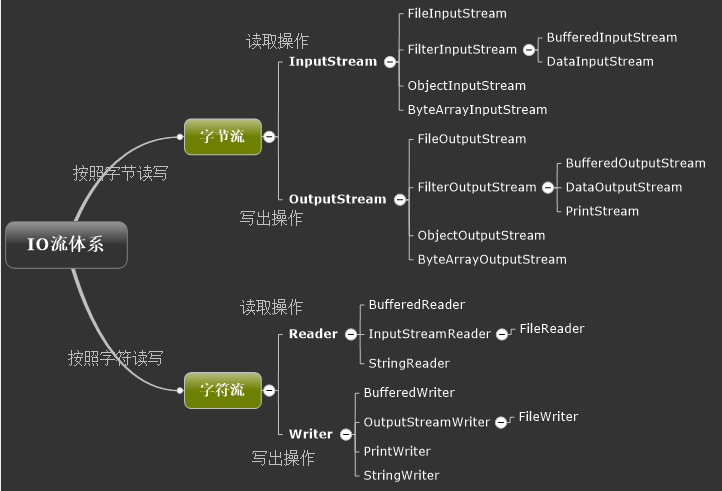
注解：红色斜体加粗的都是抽象类。

其中PipeInputStream、PipeOutputStream、PipeReader、PipeWriter是向与其它线程共用的管道中读取和写入数据。

其中上面的文件流、访问数组的流、访问字符串的流都是**介质流**，都是直接对应物理介质，如对文件、对数组、对字符串进行读写等。

上面的对象流、缓冲流等则属于**装饰流**，要配合其它流一起使用才行，如BufferedInputStream创建对象时必须传入一个InputStream类型的流作为参数。

**java中常用的流的继承关系如下：**



### 3.字符流和字节流的区别

**字符流本质上就是基于字节流读取时去查了指定的码表**。

**①读写单位不同**：字节流以字节(8bit)为单位；字符流以字符为单位，根据码表映射字符，一次可能读多个字节。java中的字符是Unicode编码，一个字符占2个字节。

**②处理对象不同：**字节流能处理所有类型的数据（如图片、视频等），而字符流只能处理字符类型的数据。

**只要是处理纯文本类型的数据，就优先考虑使用字符流。除此之外都使用字节流**。

**字节流没有缓冲区，是直接输出的，而字符流是输出到缓冲区的**。因此在输出时，字节流不调用close()方法时，信息已经输出了，而字符流只有在调用close()方法关闭缓冲区时，信息才输出。要想字符流在未关闭时输出信息，则需要手动调用flush()方法。实际上字节流在操作中是对文件本身进行直接操作的。而字符流在操作中是通过缓冲区再进行操作的。

## 四、字节流

### 1.字节流中常用的方法

### ⑴InputStream的读方法

**int read()**：从输入流中读取一个字节的数据，这个字节数据的值以int的形式返回，这个int对应的范围是在0到255之间。

**int read(byte[] b)**：从输入流中读取多个字节数据并存放在缓冲数组byte[] b中，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。相当于read(b,0,b.length)。

**int read(byte[] b,int off,int len)**：从输入流中读取到指定长度(len)的字节数据并存放在缓冲数组b中，同时返回实际读取的字节数量。如果返回-1，表示读到了输入流的末尾。第一个读取的字节数据存放在字节数组b[off]的位置，下一个读取的字节数据则存放在b[off+1]的位置，依次类推。读取的字节总数量<=len。

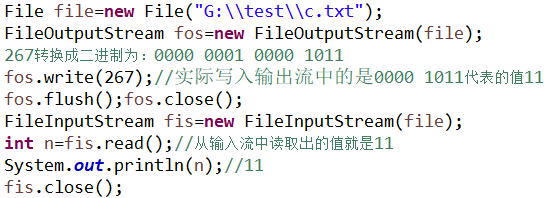
**流中的数据读取结束判定**：**read()方法的返回值为-1，readLine()方法的返回值为null**。

### ⑵OutputStream的写方法

**void write(int b)**：往输出流中写入一个字节。这个字节是参数b转换成二进制的低8位代表的字节。

**void write(byte[] b)**：将字节数组b中所有的字节写入输出流中。这个方法的效果和write(b,0,b.length)的效果是一样的。

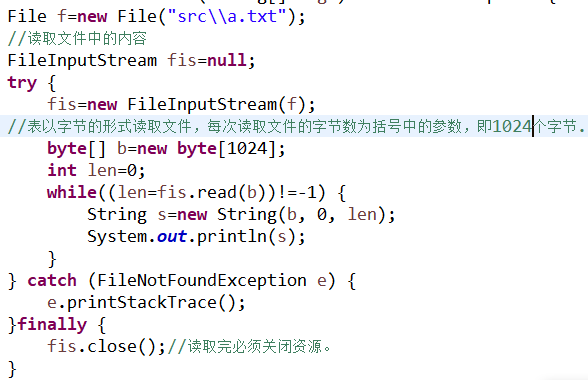
**void write(byte[] b,int off,int len)**： 从字节数组b中的off位置开始，将len个字节写入输出流中。b[off]是第一个被写入输出流中的字节，b[off+len-1]是最后一个被写入输出流中的字节。

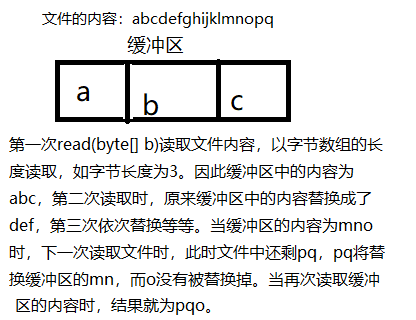


### 2.文件流(FileInputStream、FileOutputStream)

文件流是对文件进行读写操作。

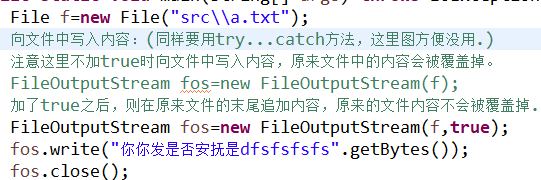
FileInputStream**.read(byte[] b)**：括号内的参数为字节数组，该方法的返回值为int类型。该方法返回的是在缓冲区中读取的总字节数。当返回值为-1时表示已经读取到文件末尾。



在文件的读取过程中，若把上面的代码的int len=0去掉，以每3个字节读取文件，则得到的结果就会多一个o。

中文乱码解决：若全是中文，则一般byte b[] =new byte[2]括号内填2的倍数。若是中英文，则会出现乱码。当文本内容比较少时，可以一次读取1024个字节，则不会出现乱码。**数组的最大长度为(231-1)-8**。

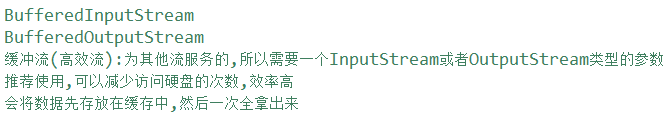
FileOutputStream**.write(byte[] b)**：将括号内的字节数组中的内容整个写入文件中。无返回值。



FileNotFoundException是IOException的子类

### 3.缓冲流(高效流)：BufferedInput(Output)Stream

缓冲流是为其他流服务的，因此需要一个InputStream或OutputStream类型的参数。



注意在使用BufferedOutputStream时，只有写了bos.flush()和bos.close()，文本中的写入的内容才会显示出来，否则内容还一直存在缓冲区中。



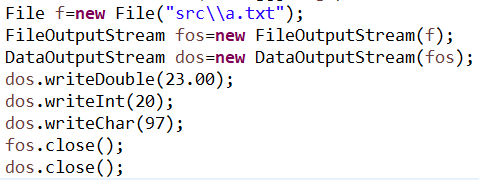
**注意，在缓冲流中，如果向文本中写入整型数据，如将97写入文本中，bos.write(97)得到的是字母a，将数字换成20等其他数字，则会出现乱码(bos.write()方法中不能填小数，否则编译会报错)。这表示缓冲流中括号内写入了的数据为整型时，会将整型先转换成整型对应的ASCII码值，然后再写入文本中，若没找到对应的ASCII值，则出现乱码。**

### 4.数据流(DataInputStream、DataOutputStream)

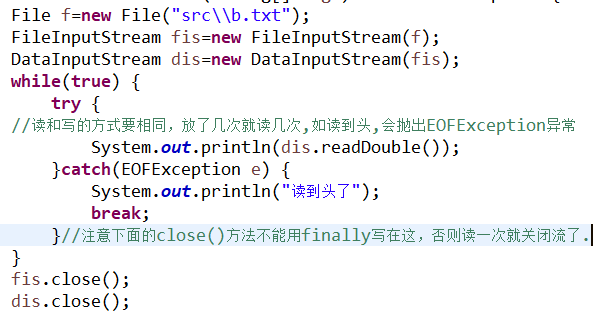
数据流可将基本数据类型写入到流中。**写入的流是以二进制文件的形式存在的，所以打开文件时会出现乱码。但是读取出来不会出现乱码。**DataInputStream、DataOutputStream括号内也相应的填入InputStream、OutputStream的类型。

DataOutputStream中的write()方法有多种，如writeInt(int v)、writeDouble(double v)括号内填对应的数据类型的值。但是writeChar(int v)括号内填的就是int类型的值，即26个字母大小写对应的ASCII码值。

**例子**：



**例子**：





直接导包，之后new 后面的内容会自动出来

### 5.对象流(ObjectInput/OutputStream)

对象流：将对象通过流的形式传输到文本中，此时需要将对象所属的类实现**Serializable**接口。若不想让对象的某个属性序列化，加上**transient**关键字，阻止属性序列化。注意transient关键字只能修饰属性。

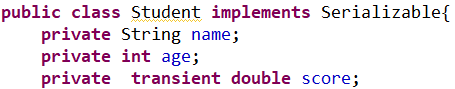
序列化：将对象转换成字节。

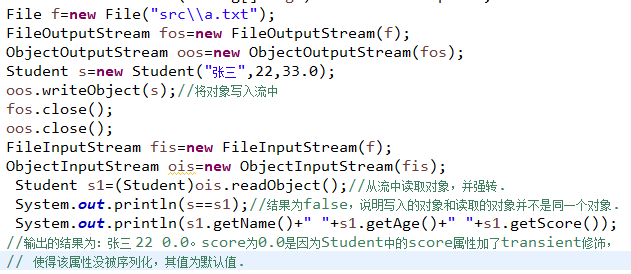
反序列化：将字节转换成对象。

ObjectOutputStream**.writeObject(Object o)**：括号内的参数为写入的对象。将对象写入流中。无返回值

ObjectOutputStream**.readObject(Object o)**：从流中读取对象。返回值为一个Object对象。

ObjectStream中write()写入的对象、和read()中读取的对象是不同的对象，即read()之后是创建了一个新对象。但是该新对象会读取原对象中的内容。若对象的属性被transient修饰了，则该属性不会序列化，最后得到的该属性的值为系统的默认值。

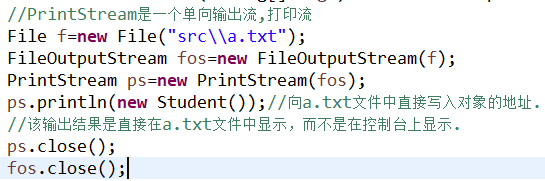




### 6.打印流(PrintStream)

PrintStream是一个单项输出流。

PrintStream**.println(Object x)**：向文件中写入括号中的内容。



## 五、字符流

字符流的括号中不能放字节流，转换流除外，同时字节流的括号中也不能放字符流。乱码原因：编码和解码采用不同的方式。

### 1.字符流中常用的方法

### ⑴Reader的读方法

**int read()**：从输入流中读取单个字符，返回值为该字符对应的int类型的整数，范围为0到65535。如果读到流的末尾则返回-1。

**int read(char[] cbuf)**：从输入流中读取多个字符并存放在缓冲字符数组cbuf中，返回值读取的字符总数量。若读到流的末尾则返回-1。

**int read(char[] cbuf,int off,int len)**：从输入流中读取指定长度len的字符数量并存放在缓冲字符数组cbuf中。存放的起始位置为cbuf[off]。返回的是实际读取的字符总数量。若读到流的末尾则返回-1。

### ⑵Writer的写方法

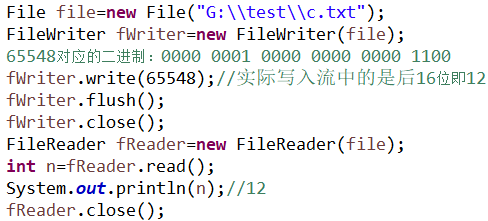
**void write(int c)**：往输出流中写入一个字符，这个字符是参数c转换成二进制数的低16位代表的字符。

**void write(char[] cbuf)**：将字符数组cbuf中所有的字符写入输出流中。这个方法的效果和write(cbuf,0,cbuf.length)的效果是一样的。

**void write(char[] cbuf,int off,int len)**：将字符数组cbuf中的从索引为off的位置处开始的len个字符写入输出流 。

**void write(String str)**：将字符串str中的字符写入输出流中。

**void write(String str,int off,int len)**：将字符串str 中从索引off开始处的len个字符写入输出流 。



### 2.字符流中的文件流(FileReader、FileWriter)



**FileReader fr=new FileReader(File file)**：括号内的参数为要读的文件。

Reader**.read(char[] cbuf)**：返回值为int类型，返回值为要读的字符数，当返回-1时，表示已经读到文件末尾了。该方法和字节流中的reader()方法一样。括号内的参数为char类型的数组。

**FileWriter fw=new FileWriter(File file，boolean append)**：括号内的参数为要写入的文件名和boolean类型的追加方式。

**FileReader和Filewriter有个缺点**：就是无法指定读取文件或者写文件的编码方式，容易出现乱码。对于与项目的编码方式不同的文本文件采用FileReader读取是会出现乱码。



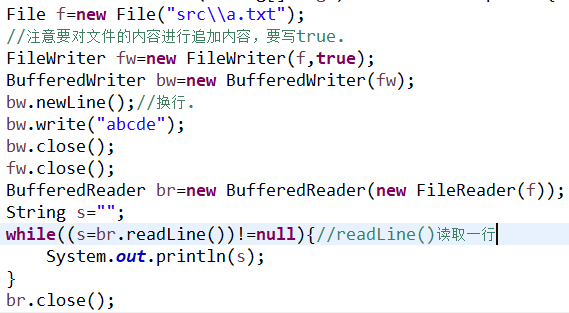
### 3.字符流中的缓冲流(BufferedWriter、BufferedReader)

BufferedWriter**.newLine()**：换行

BufferedWriter**.write(String s)**：向文本中写入内容。

BufferedReader**.readLine()**：读取一行，并以String类型返回，当读到末尾一行无内容时，返回null。

**注意readLine()方法一次读一行，不识别换行符和不会自动换行**。**BufferedWriter和PrintWriter的write()方法也不会识别换行符和不会自动换行，在文件中换行可以采用newLine()实现**。



**BufferedWriter bw=new BufferedWriter(Writer in)**：括号内参数填Writer类型。

**BufferedReader br=new BufferedReader(Reader in)**：括号内参数填Reader类型。

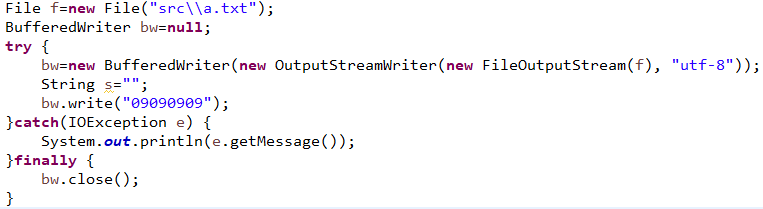
### 4.转换流(InputStreamReader、OutputStreamWriter)

将字节流转换成字符流。

**InputStreamReader isr=new InputStreamReader(InputStream in，String charsetName)**：括号中第一个参数为输入流，第二个参数为字符编码，如utf-8、GBK等，表以什么编码格式读取文件内容。

**OutputStreamWriter osr=new OutputStreamWriter (OutputStream out，String charsetName)**：括号中第一个参数为输出流，第二个参数为字符编码，如utf-8、GBK等，表以什么编码格式写入文件内容。





## 六、System类对IO的支持

在java.lang包中的System类被final修饰，该类不能被继承，同时它的构造器被private修饰，因此不能在外面创建对象。

**public** **final** **class** System {...}

**private** System() {}

System类提供了如下三个常量：

**public** **final** **static** InputStream ***in*=null** ; 对应标准输入

**public** **final** **static** PrintStream ***out*=null**; 对应标准输出

**public** **final** **static** PrintStream ***err*=null**; 对应标准错误输出

**注意常量out和err都是PrintStream类型的，常量in是InputStream类型的**。

java语言系统中自带的标准数据流，分别是：

**①System.in**：标准输入流，通常代表键盘输入。

**②System.out**：标准输出流，通常是输出到显示器。

**③System.err**：标准错误输出流，通常是输出到显示器。

每当执行main方法时，就会自动生成上面三个

### 1.标准输出流System.out

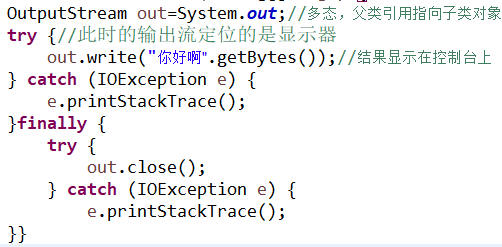
System.out向标准输出设备中输出数据，通常将数据输出到显示器中。对应的数据类型是PrintStream，因此可以调用PrintStream中的各种方法。常用的方法：

void print(参数)；

void println(参数)；

如System.out.println(“输出内容”)；

FileOutputStream这个文件输出流定位的是文件，也就是说写入这个输出流中的数据实际上存在文件中。PrintStream类型的System.out流定位的是显示器，也就是说写入这个流中的数据会显示在屏幕中。PrintStream类是OutputStream类的子类。



### 2.标准输入流System.in

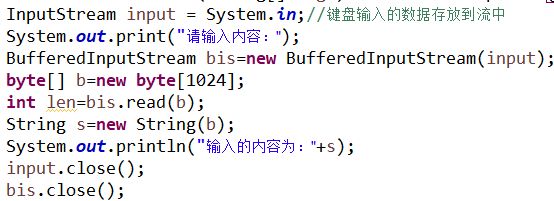
System.in是从标准输入设备中读取数据，如键盘。对应的数据类型为InputStream，因此可以调用InputStream中的各种方法。常用方法如下：

int read();

int read(byte[] b);

int read(byte[] b,int off,int len);

InputStream类型的System.in流定位的是键盘，也就是从键盘中读取数据。

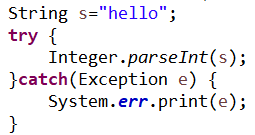




### 3.标准错误输出流System.err

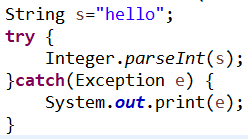
System.err是向标准输出设备中输出错误，通常输出错误信息到显示器中。对应的数据类型为PrintStream。可以直接使用PrintStream中的方法。

采用System.err输出错误结果如下：





采用System.out输出错误结果如下：





#### ⑴System.out和System.err的区别

System.out和System.err都是PrintStream类型，两者都可以输出错误信息。通常，System.out是将信息显示给用户看，是正常的信息显示，而System.err输出的内容是不希望被用户看到，会直接在后台打印，是专门显示错误的。一般，输出错误信息最好用System.err而不是System.out。

### 4.标准输入输出流的重定向

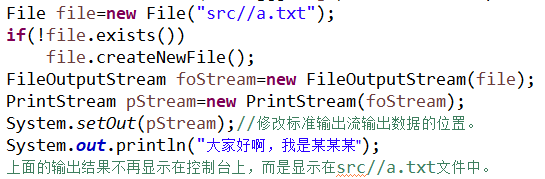
System.in标准输入流通常的数据来源是键盘，System.out和System.err流通常数据输出到显示器上。但通过System类的方法，可以改变System.in的输入来源以及System.out/err的输出位置。System类中的三个方法的源码如下：

**public** **static** **void** setIn(InputStream in) {}：重定向标准输入流。

**public** **static** **void** setOut(PrintStream out) {}：重定向标准输出流。

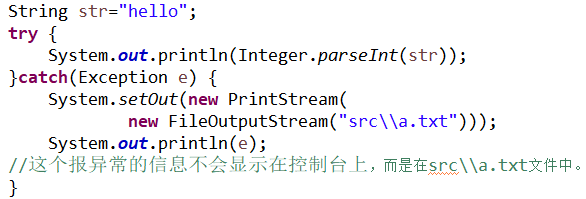
**public** **static** **void** setErr(PrintStream err) {}：重定向标准错误输出流。

#### ⑴System.out输出重定向



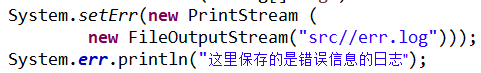


修改标准输出流的输出位置，可以让System.out标准输出流输出的信息不显示在控制台上或者显示器中。而是专门保存在一个文件中。如将输出的异常信息存放在指定的文件中。





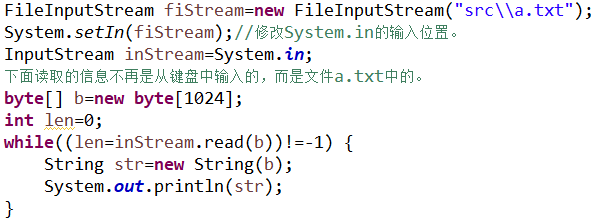
#### ⑵System.err输出错误的重定向





#### ⑶System.in输入重定向

默认情况下System.in是从键盘中输入信息。通过System.setIn()方法可以改变输入的方式，如输入的信息是从文件中读取而不再是从键盘中读取。



### 5.小结

System.out输出的信息是希望用户可以看到。在IDE(如Eclipse)中，若输出的错误信息，字体是黑色的。

System.err输出的信息是不希望用户看到，在IDE中以红色字体的形式输出错误信息

System.in是从键盘输入信息。System.in读取数据的时候可能出现乱码的情况。乱码的原因可能是编码和解码的方式不一致，也可能是在读取的过程中，因为是按字节的方式读取中文，而一个中文占2个或3个字节，从而导致一个中文被读了一半就转换成字符了。解决读取中文乱码问题，可以采用BufferedReader字符缓冲流或者其它缓冲流的方式解决。

上面三个常量都可以重定向，但一般只建议修改System.setOut()。

## 七、IO中常见的异常

**IOException**异常：当I/O操作失败或者发生中断时抛出该异常。

下面三种异常都是IOException异常的子类。

**EOFException**异常：在输入过程中意外的到达了文件的末尾或者流的末尾时抛出该异常。

**FileNotFoundException**异常：当试图打开指定的文件失败或者找不到该文件时抛出该异常。

**InterruptedIOException**异常：当I/O操作被中断时抛出该异常。

# 其它

## 一、字符串与字节数组相互转换

**在utf-8编码中，中文占3个字节，英文占1个字节。在gbk编码中，中文占2个字节，英文占1个字节**。

**java采用双字节编码(utf-16be编码)，就是java中一个字符占2个字节，中文占2个字节，英文也占2个字节**。

**不管什么编码格式，一个中文都是占一个字符**。

String**.getBytes()**：返回值为byte[]字节数组。该方法是将String类型转换成Byte类型的数组，并且使用的是工作台或者项目默认的编码格式。

String**.getBytes(Charset charset)**：返回值也是byte[]字节数组，该方法是将String类型的字符串按照给定的编码格式转换成字节数组。

**String s=new String(byte[] b,int offset,int length)**：这是将字节数组中的内容转换成字符串。new String alt+/会出现括号中的参数括号的参数为。

**String s=new String(byte[] b,Charset charset)**：这是将字节数组以指定的编码格式转换成字符串。

**String s=new String(byte[] b)**：这是将字节数组以默认的编码格式转换成字符串。

注意：当字节数组指定的某种编码格式时，当转换成字符串时也要指定该种编码格式，否则会出现乱码。

下面这段代码有一部分没明白。

0xff代表的是16进制的数，在16进制中，f代表15，则0xff代表10进制的15\*16+15=255，表示为二进制为11111111。&符号是进行按位与计算。

