**P221 IO流（概述）**

字节流和字符流的区别：

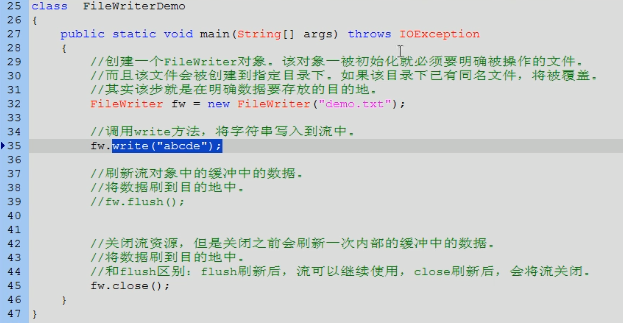
数据在计算机中流动，最终都是转换成二进制字节。而数据有很多种类，比如mp4，mp3，文本等等。媒体数据本身就是二进制的，字节流能够搞定。

而为了能够方便处理文本数据，将字符流单独分离出来。为了建立字符与二进制之间的映射关系。英文字母：ascii；汉字gbk；全球unicode—>utf-8。这样的结果就是，同一个二进制数，在不同码表下，对应的字符是不一样的。同样的，一个汉字，在不同的识别中文的码表中，转换成的二进制数也是不同的。基于此问题，就在字节流的基础上产生了字符流。字符流能够在内部融合编码表和字节流。

字节流的两个基类：InputStream；OutputStream

字符流的两个基类：Reader；Writer

**P222 IO流（FileWriter）**

通过write方法，将输入的字符串等存入到内存中。需要再调用flush方法将内存中的数据存储到指定的文件中。

这些方法都会抛出IO异常，比如write，他不能保证自己不会出错（写的内容超过了内存）。

**P223 IO流（IO异常处理方式）**

既然IO的方法很多都会抛出异常，那么怎么处理这些异常呢？

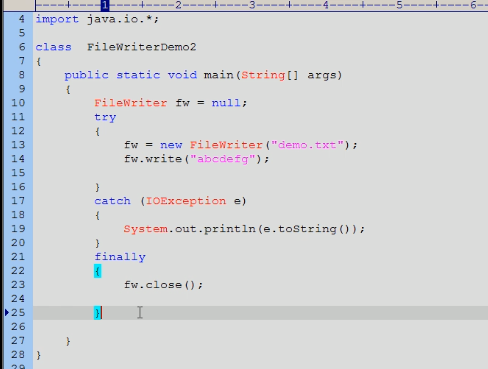


为什么会报错呢？

注意，finally代码块中用到了fw，而fw是定义在try代码块中的局部变量

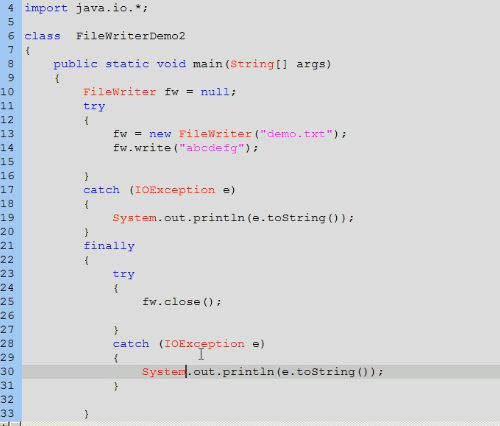
那把第12行那句移动到第10行行吗？不行。因为12行那句本身也会报错，你又要去捕捉，一捕捉又成了try代码块中的局部变量。

解决方式：



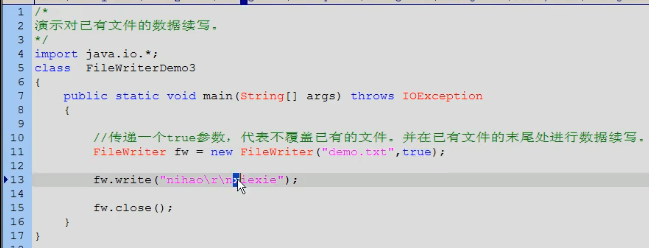
再外部先定义fw这个引用变量，但是不进行初始化。

但是这样写还是会报错。因为finally语句块中的close方法也会有IO异常。



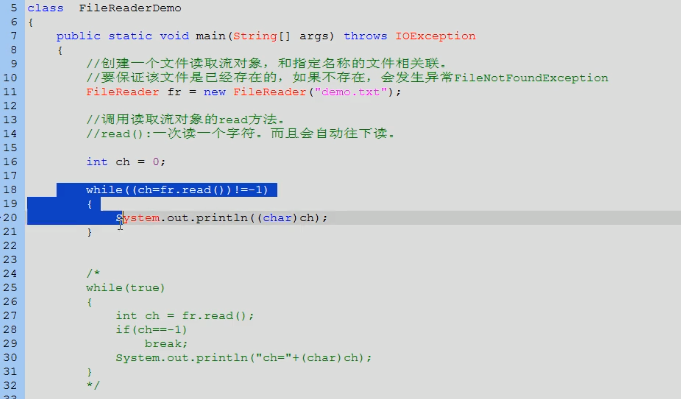
**P224 IO流（文件的续写）**

像在P223中new一个对象：如果目录下没有该文件，就会新建一个；有该文件，就会覆盖这个文件。那么怎么对原有文件进行续写呢？



\n是linux的回车符，一到windows的记事本中，就会变为黑方块，windows的回车符是\r\n

**P225 IO流（文本文件读取方式一）**

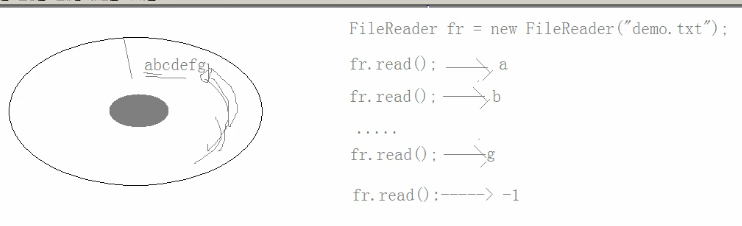


16-21的方法等同于25-31

读取的对象会以-1结束。

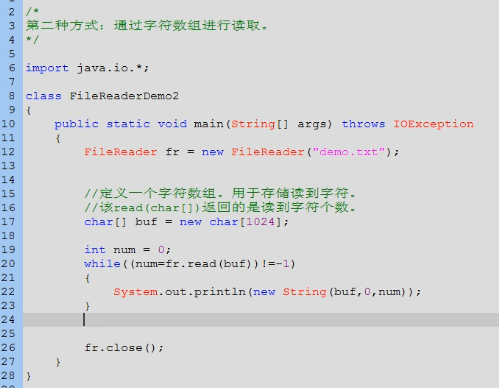
原理解释：

当new了一个FileReader类型的对象以后，就与硬盘中的该文件相关联，相当于操作计算机硬件磁头移动到该文件存储硬盘位置的开头。调用read方法，就相当于操作计算机硬件磁头在读取磁盘上的数据。磁盘上的数据会用一个标识来分隔，表示一个文件结束。当java读到这个标识时，就会返回一个-1.



**P226 IO流（文本文件读取方式二）**

查看read方法的api文档，知道当直接调用read方法时，返回一个字符。当向read方法传入一个字符数组时，就会返回一个数组。



关键就是20行num=fr.read(buf)。

计算机硬件的实现跟read()方法一样，从目标文件中逐个读取字符，然后逐个存放在字符数组中。直到读到文件的结束标识或者是数组装满。返回的是当前数组的指针位置。当数组装满时，就是数组的长度；当数组没装满时，返回的就是当前装了的个数。

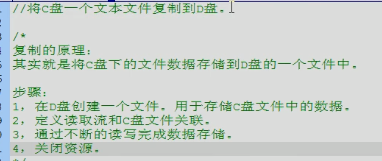
举个例子，我们的数组长度为1024。文件长度为3000。那么先将1-1024装入数组，返回1024，并打印该数组。然后将1025-2048装入数组，覆盖掉1-1024的字符，返回1024，并打印该数组。最后将2049-3000装入数组，返回952，并打印该数组。那么对于这个数组而言，0-951是新装入的字符，952-1023是未被覆盖的老字符。接下来再对文件进行读取，一上来就碰到结束标识，跟read()一样，返回-1，当然数组也保持原状。

跟直接调用read()的区别就是read()是读一个输出一个，二read(buf)是一直一个一个的读入，直到数组满了才输出。

**P227 IO流（文本文件读取练习）**

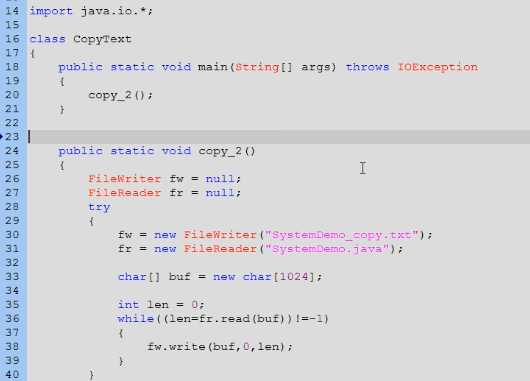
//

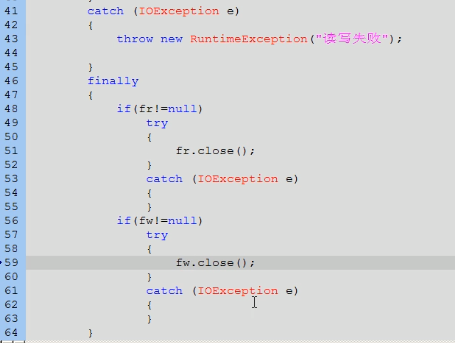
**P228 IO流（拷贝文本文件）**



-copy\_1方法可行，但是效率低，只为了一个字符，让磁头在目标文件和存放目的文件之间反复切换。

看似复杂，其实就是将异常就地解决了，没有向外抛出。18行就可以不抛



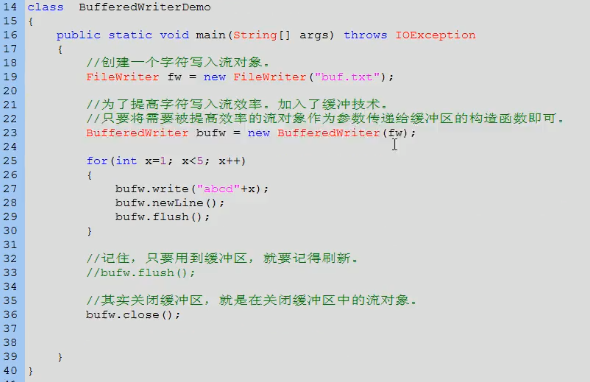


**P229 IO流（拷贝文本文件图例）**



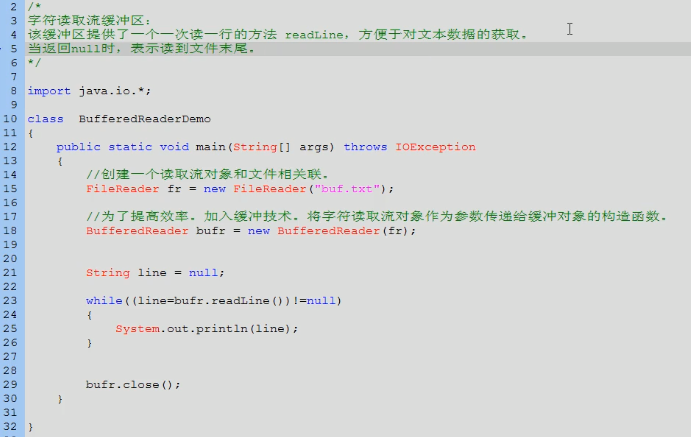
FileWriter这个流可以凭空产生，FileReader这个流必须要有个文件与之关联。两个流通过字符数组传递数据。

**P230 IO流（BufferedWriter）**



但是以上操作，仅靠FileWriter也能实现。但是更加高效？

**P231 IO流（BufferedReader）**

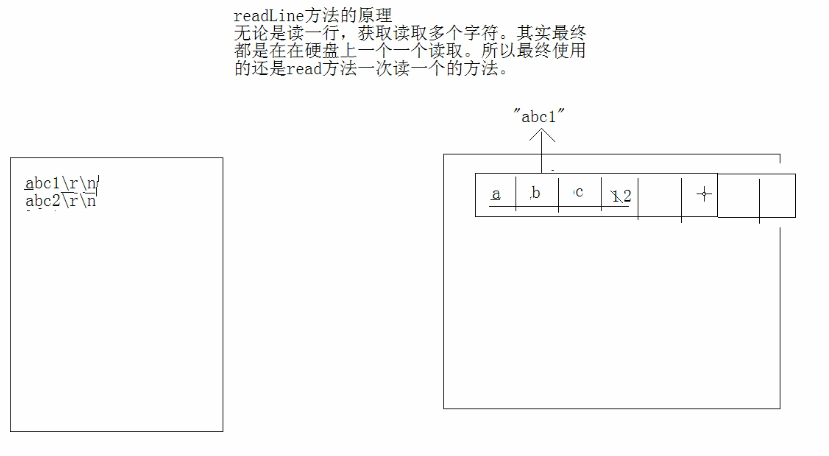


**P232 IO流（通过缓冲区复制文本文件）**

//

**P233 IO流（readLine的原理图例）**

其实就是建立了一个足够长的字符数组，然后从文件中一个字符一个字符的读取，当读到回车符（\r\n）时，将当前的字符数组转换成String返回。



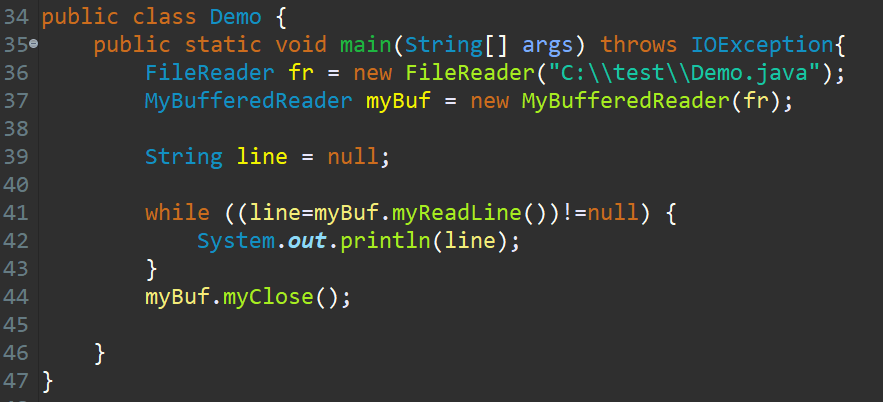
**P234 IO流（MyBufferedReader）**

既然知道了BufferedReader类中最重要的readLine方法的原理，那么我们就可以复现。

注意第14行：这个循环会一直进行，直到整个文件的结束。

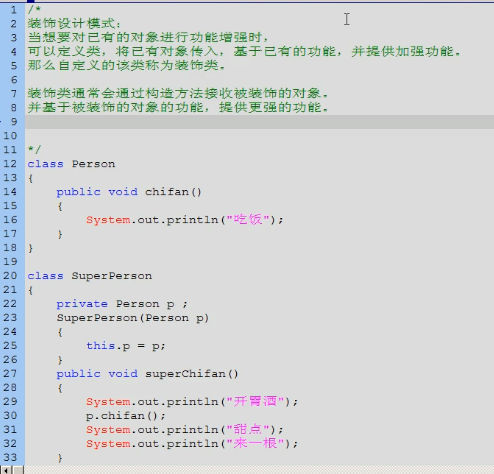
同时第23-24行，保证了当文件是以字符结束而不是回车符结束时，最后一行字符数组的稳定输出





**P235 IO流（装饰修饰模式）**

通过P234的例子，我们发现readLine方法是对read方法的功能增强，并且底层实现还是调用read方法。增强的实现是将FileReader对象传给BufferedReader对象。这就是装饰设计模式。



**P236 IO流（装饰和继承的区别）**

装饰类是可以通过继承来实现，即上一P的SuperPerson可以继承Person来实现，并把30行改为super.chifan();既可。

那么为什么还需要装饰类呢？

其实，装饰类是继承方法在某种条件下的简化。为什么？考虑下面这种情况。

一个父类直接产生了很多子类。而这些子类又需要增强同一功能，于是每个子类自身又需要产生一个新的子类，整体看上去很臃肿。（虽然有时可以将新功能设计成一个接口，但是要实现接口功能，子子类还是需要定义出来。而且有时候要基于父类方法来增强）

所以不如直接再定义一个增强类，谁用，就把谁传递进来。



这样，就不需要定义子子类。增强类的构造函数还可以通过多态进一步优化，并且继承父类的方法。优化后的体系就变成了这样。



装饰模式比继承要灵活，避免了继承体系臃肿。

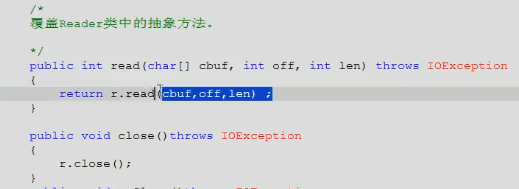
而且降低了类与类之间的关系。

装饰类因为增强已有对象，具备的功能和已有的是相同的，只不过提供了更强功能。所以装饰类和被装饰类通常都是属于一个体系中的。

**P237 IO流（自定义装饰类）**

那么，我们可以对P234中的MyBufferedReader类进行优化。我们可以在构造函数中接收Reader类，同时继承Reader类。

但是在继承Reader类时我们发现它是个抽象类，我们还要去实现抽象方法。但是有些方法涉及到计算机的底层调用。那么我们直接让传入的Reader子类去实现



**P238 IO流（LineNumberReader）**

LineNumberReader是BufferedReader的子类

setLineNumber方法设置行号，getLineNumber方法获取行号。

**P239 IO流（MyLineNumberReader）**

//

------------------------------------------------------------------------------------------------------------------------------------------

**P240 IO流（字节流File读写操作）**

InputStream读文件

OutpurStream写文件

**P241 IO流（拷贝图片）**

//

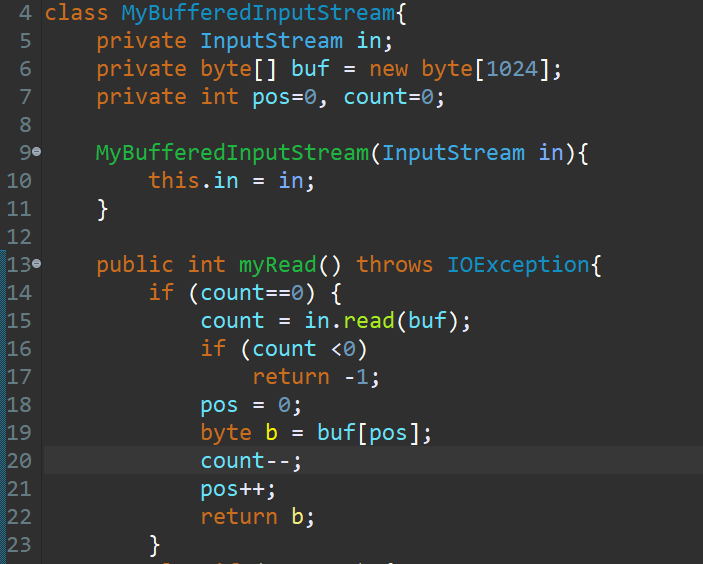
**P242 IO流（字节流的缓冲区）**

//

**P243 IO流（自定义字节流的缓冲区-read和write的特点）**

字节流缓冲的底层实现：调用FileInputStream的read方法，并向read方法传入一个byte数组，返回该数组和数组长度（媒体文件都是以二进制存储在硬盘中，每次read都会得到byte长度的二进制数据存放到数组中）。然后通过指针和计数器依次读取数组中的元素。读完整个数组后再次调用传入数组的read方法读取数据，直到整个文件读取完毕。

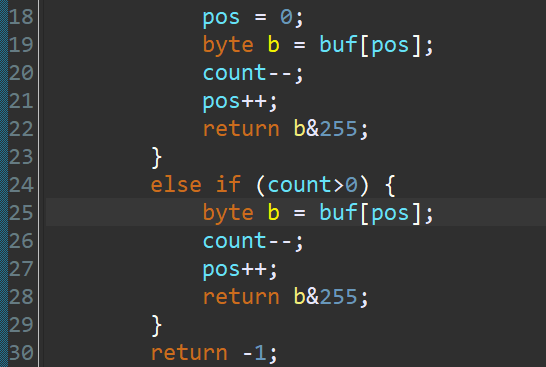
那么我们去复现BufferedInputStream类





不幸的是，复制失败。

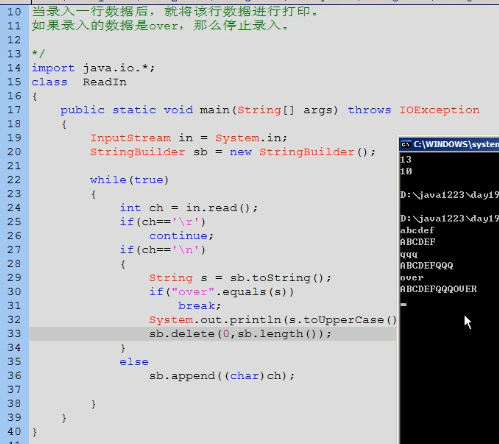
失败原因：文件以二进制存储，它返回的b本来是byte类型（也就是4个二进制位），强制提升为了int类型（也就是16个二进制位）。根据44行的判断条件：很有可能b本来是1111，然后被强制提升为1111-1111-1111-1111，也就是int类型的-1，使得循环结束。那么我们应该使得b返回的int结果应该是0000-0000-0000-1111，这里我们与上一个int255（也就是0000-0000-0000-1111），就ok了。



只需对22和28两行略作修改。

但是我们也发现一个问题：myRead方法读取的数组元素是byte类型，返回的int类型，为什么写入的文件和原文件一样，而不是原文件的4倍？其实write方法本身，对于16位的二进制数，只选择了最后的4位进行写入。

**P244 IO流（读取键盘录入）**



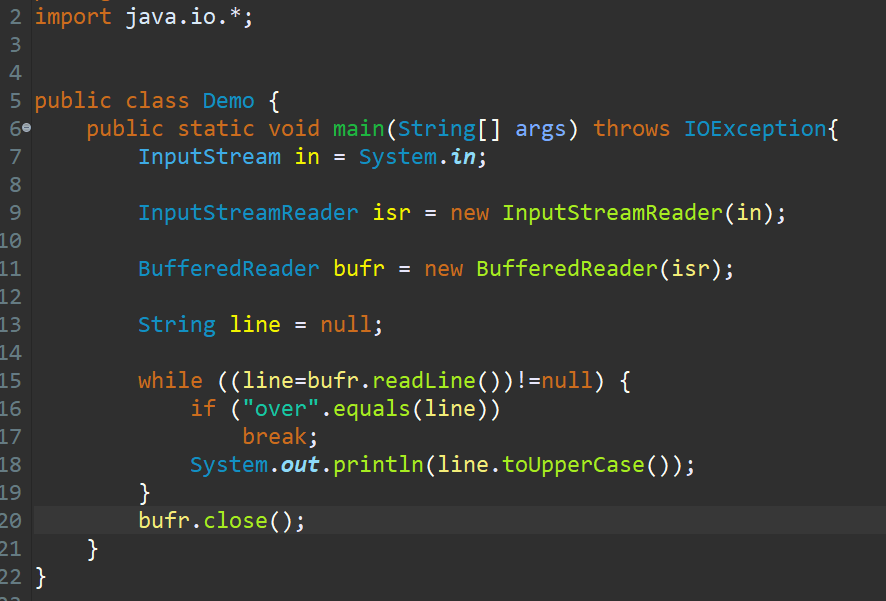
**P245 IO流（读取转换流）**

我们发现P244有点像readLine方法。于是产生了一个需求：

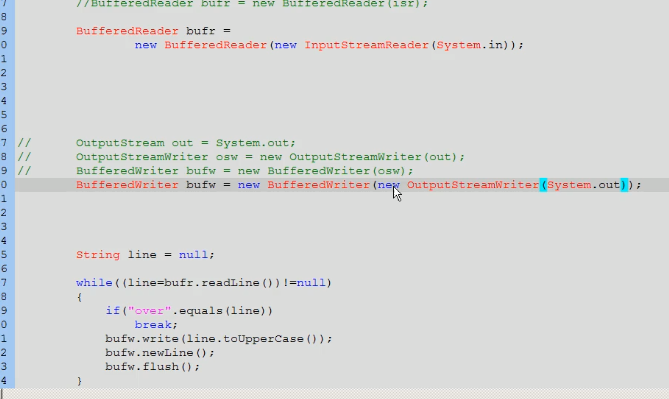
readLine是字符流BufferedReader的方法

而键盘录入的read方法是字节流InputStream的方法

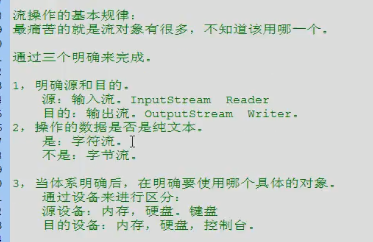
那么能不能将字节流转成字符流再使用字符流缓冲区的readLine方法呢？



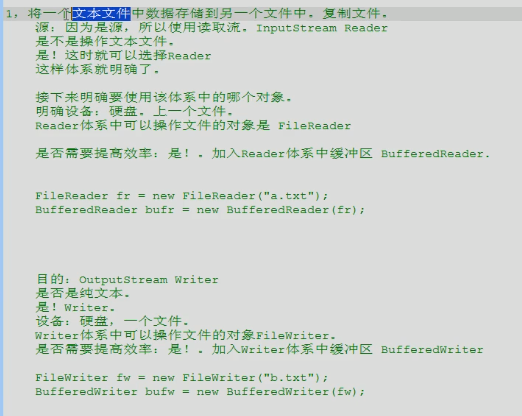
**P246 IO流（写入转换流）**



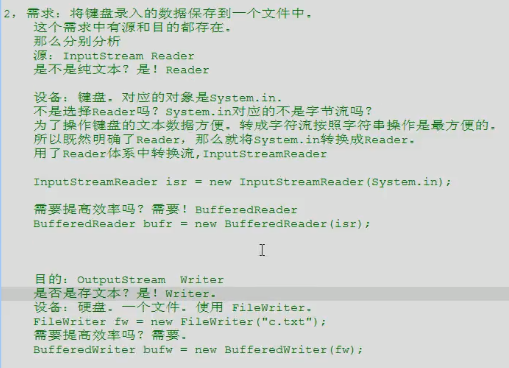
**P247 IO流（流操作规律-1）**

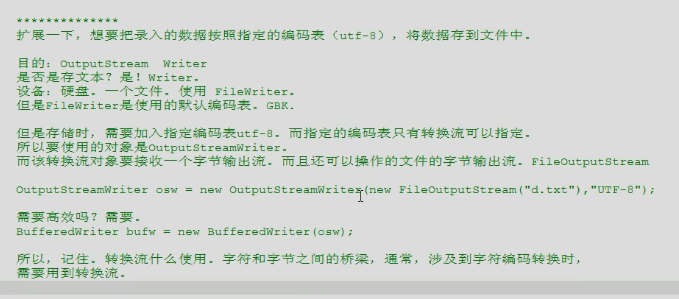


举个例子



**P248 IO流（流操作规律-2）**





**P249 IO流（改变标准输入输出设备）**

//

**P250 IO流（异常的日志信息）**

//

**P251 IO流（系统信息）**

//

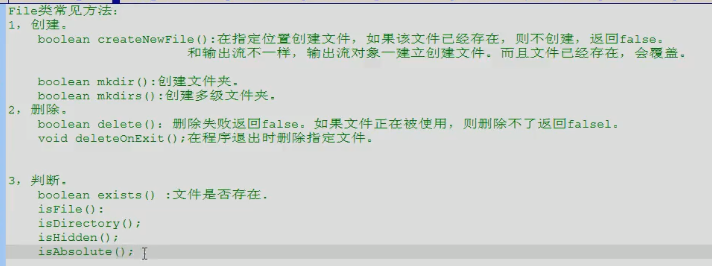
**P252 IO流（File概述）**

流只能操作数据，而想要操作由数据封装成的文件的信息，必须要用File对象。

**P253 IO流（File对象功能-创建和删除）**

//

**P254 IO流（File对象功能-判断）**



**P255 IO流（File对象功能-获取）**



**P256 IO流（File对象功能-文件列表）**

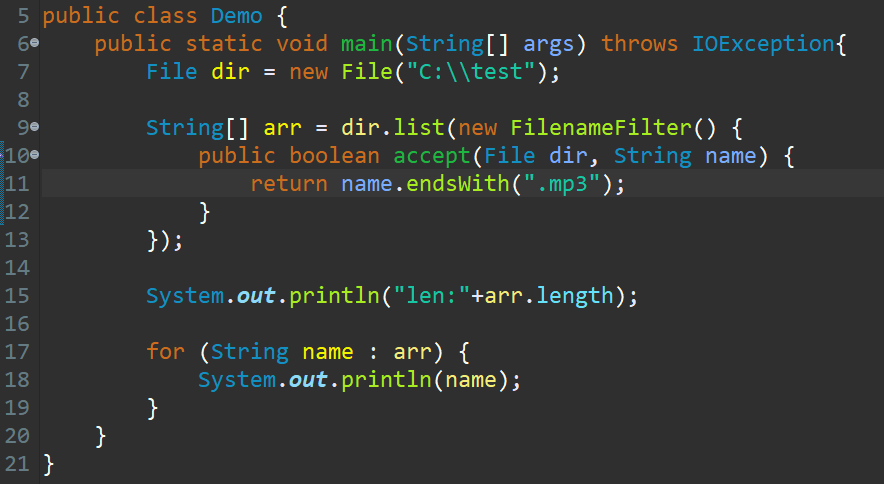
//

**P257 IO流（File对象功能-文件列表2）**

需求：返回文件夹下符合指定名称的文件和文件夹

传入一个匿名类并重写accept方法：true全部，false全不。

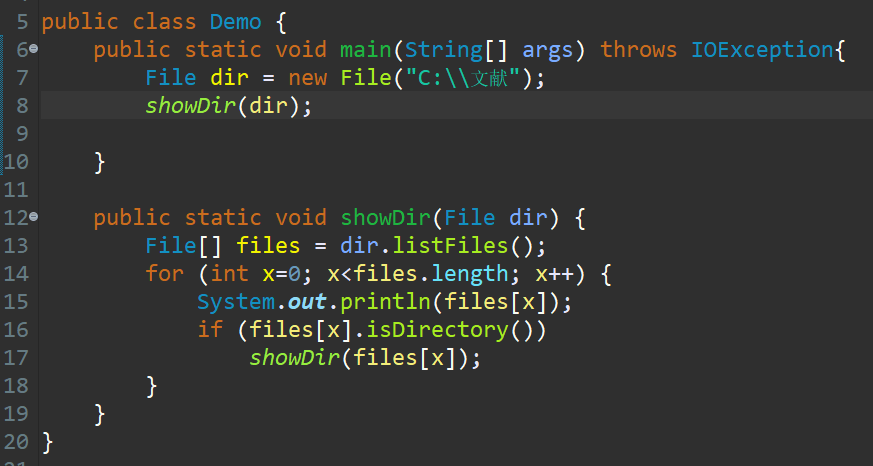
所以让符合条件的返回true，不符合的返回false



-list方法和listFiles方法的区别：list返回的是String[];listFiles返回的是File[]

**P258 IO流（列出目录下所有内容-递归）**

需求：上述方法只能返回当前目录下的所有文件和文件夹，但是无法获得文件夹中的文件和文件夹。我想要获取当前目录下所有的文件（文件夹扒到文件为止）。递归实现

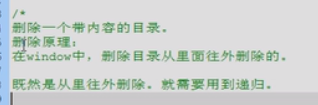


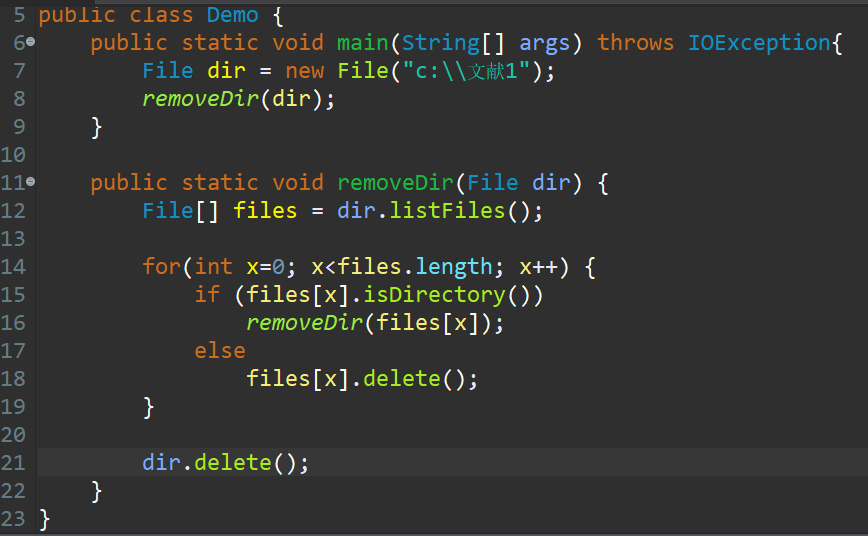
递归在调用过程中会不断的在栈内存中开辟空间。

**P259 IO流（列出目录下所有内容-带层次）**

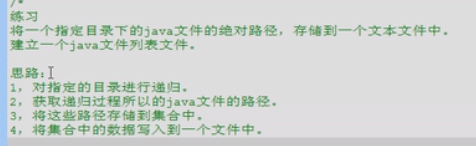
//

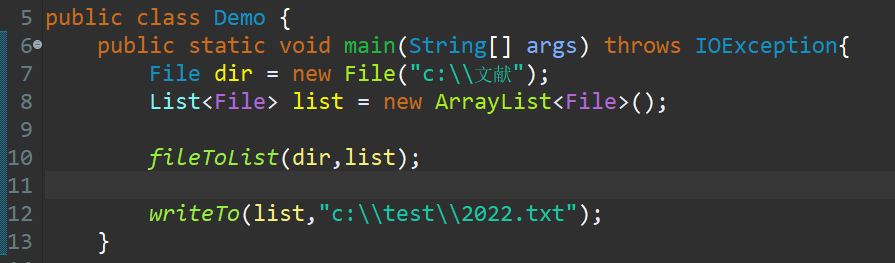
**P260 IO流（删除带内容的目录）**

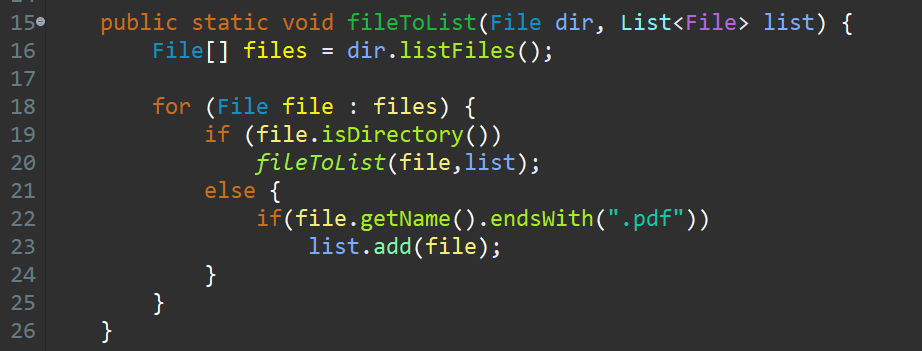


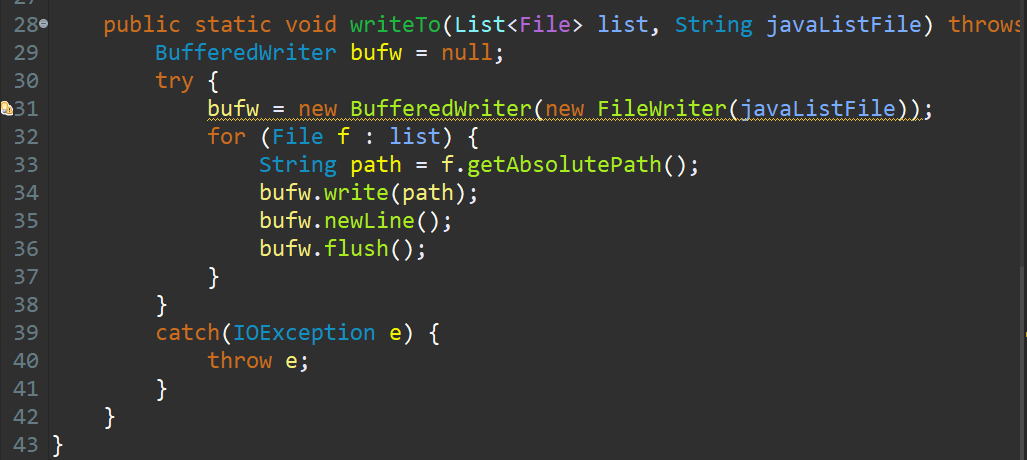


**P261 IO流（创建java文件列表）**

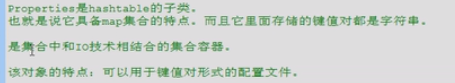








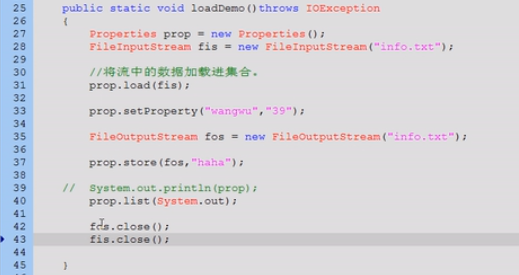
**P262 IO流（Properties简述）**



**P263 IO流（Properties存取）**

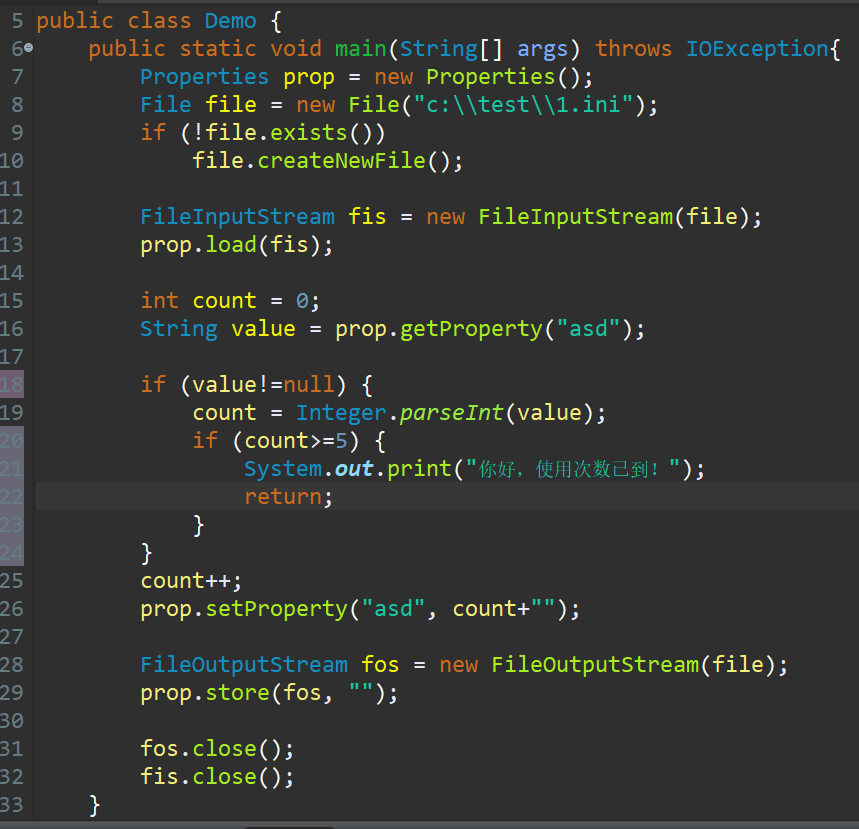
//

**P264 IO流（Properties存取配置文件）**



注意33行，只是在内存中对键值对进行了修改，所以如果要把硬盘上文件的键值对进行修改，新建一个输出流store一下就行。

**P265 IO流（Properties练习）**

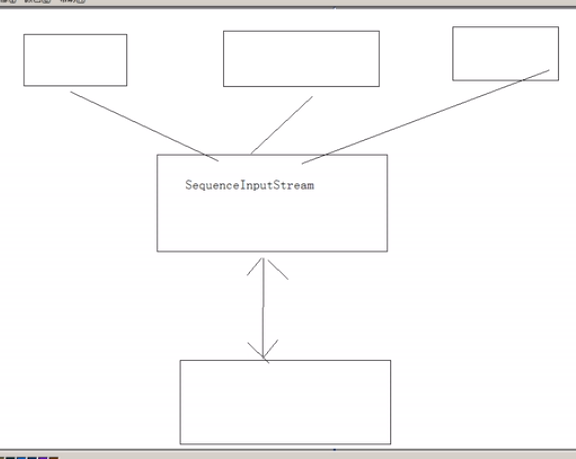


**P266 IO流（PrintWriter）**

//

**P267 IO流（合并流）**

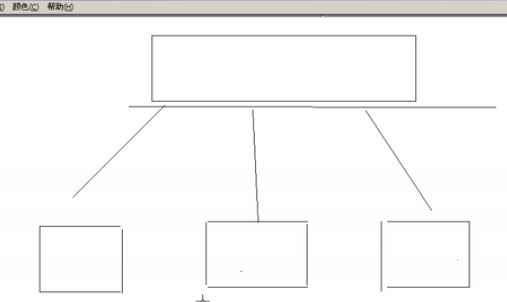
作用：将多个流对象合并，读完一个流，就接着读下一个流



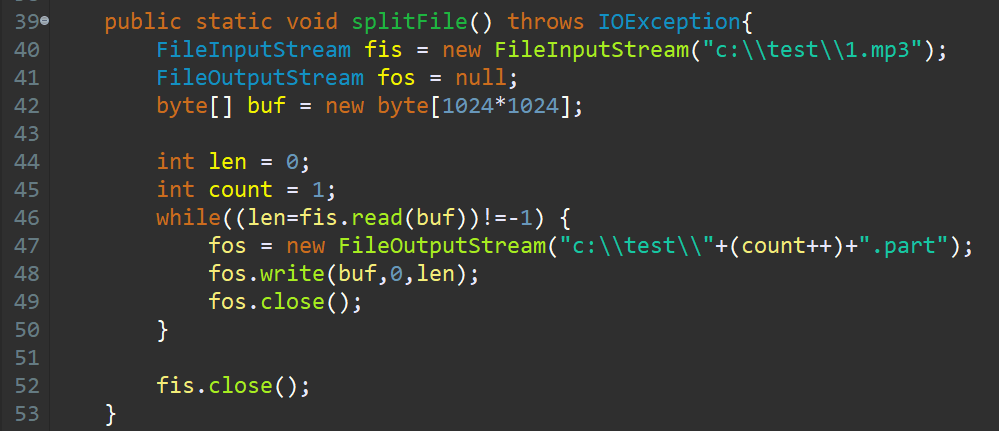


**P268 IO流（切割文件）**

将一个读取流切割成多个



将一个mp3文件切割，每个大小为1mb



将切割后得到的碎片文件重新拼回成mp3



**P269 IO流（对象的序列化）**

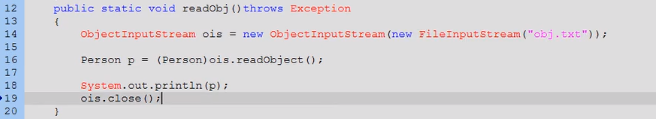
操作对象的流：ObjectInputStream和ObjectOutputStream

对象存在于堆内存当中，可以将封装有特有数据的对象保存到硬盘中，以便将来的读取（这个就像是网游的游戏账号）。这就是对象的序列化（也叫持久化）。



例子是将Person对象封装进了硬盘（只是保存其中的成员变量和成员函数，也就是说无法序列化静态的变量和函数，因为它们在方法区而不是在堆内存中）。但是，对象要实现java.io.Serializable接口来启用其序列化功能。

将序列化的文件读取回来



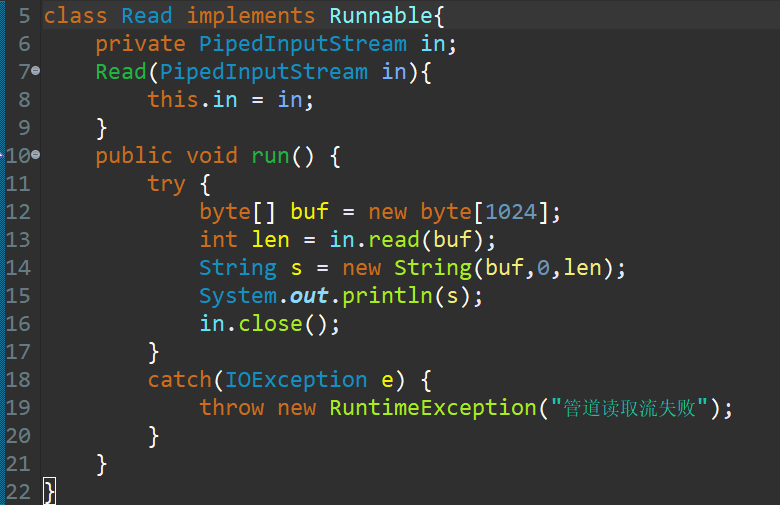
同时又出现一个需求：静态变量无法被序列化。但是我有一个非静态变量，也不想被序列化，怎么办？

加上transient关键字：int age; -🡪 transient int age;

**P270 IO流（管道流）**

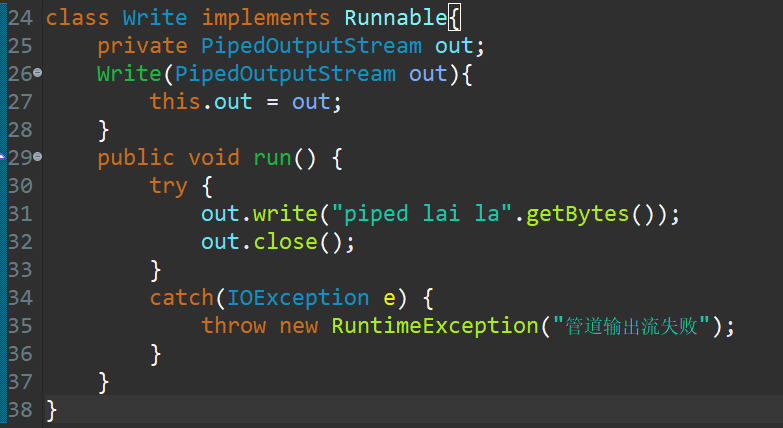
用于多线程

管道读取流

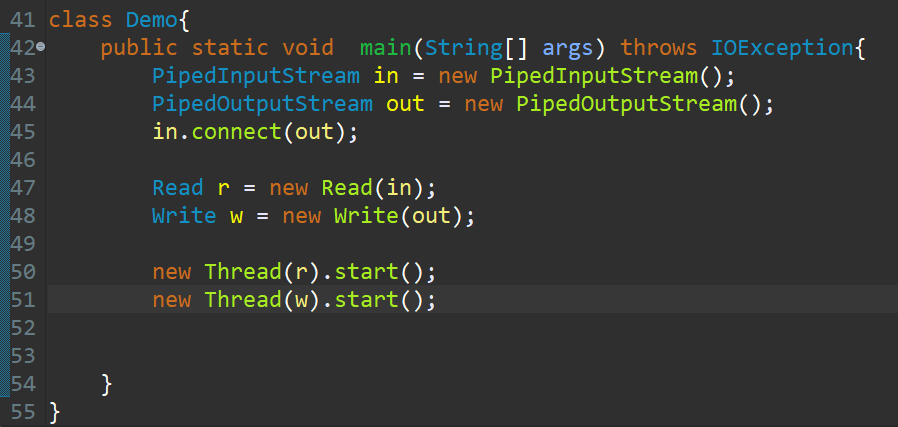


特点在13行的read方法，如果多线程中读取流先执行，因为没有东西可读，代码运行到13行时会阻塞，直到管道中有数据。

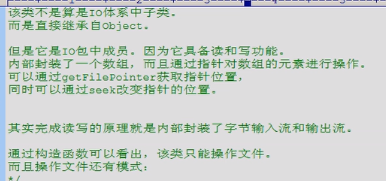
管道输出流

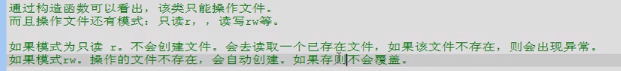


主程序



**P271 IO流（RandomAccessFile）**





RandomAccess即随机访问，体现在哪里？

一般读取文件，需要从头到尾一行一行的读取。但是存储在RandomAccess中的数据是以字节为单位依次存储，所以可以通过seek方法控制指针，从任意一个位置开始读取（前提是数据有规律：比如这个文件存储了1000个人的信息，每个人的信息所占的字节应该是相等的）。写入当然和读取的道理一样

用途：随机访问有什么好处呢？

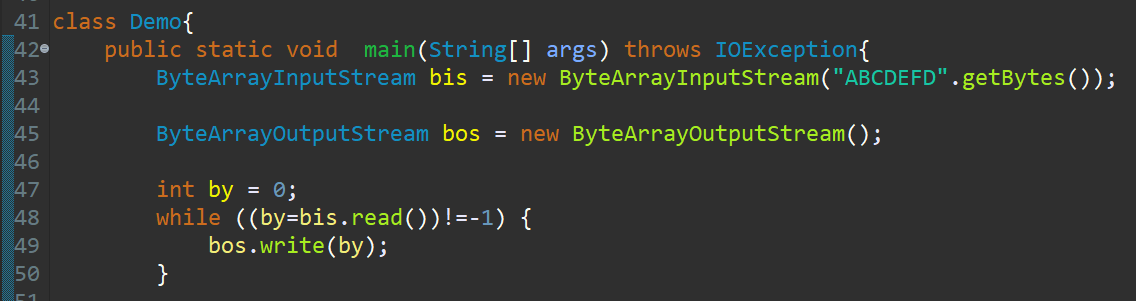
你写入的位置是已知的，那么你可以将你要写入的文件分割成多份，用多个线程实现同时写入。（你在网上下载软件之类的就是这样）

**P272 IO流（操作基本数据类型的流对象DataStream）**

能直接读写int，double，boolean等类型的数据

**P273 IO流（ByteArrayStream）**

操作字节数组



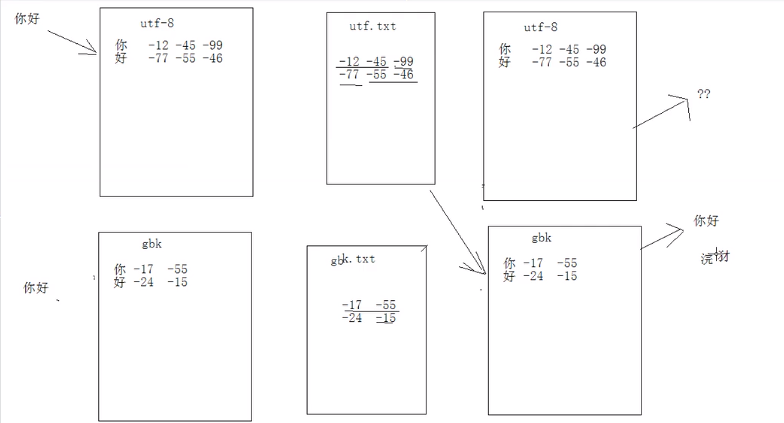
对内存的数据进行读写。当然例子中从内存中读取数据，然后写到内存中并没有啥意义。但是，从文件中读到内存中，或将内存写到文件中，就很有意义了。

**P274 IO流（转换流的字符编码）**

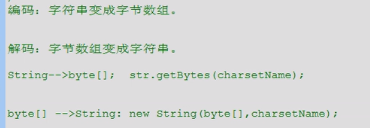
一个字节（Byte）有8个二进制位

举个例子，你写入了两个中文汉字：“你好”，如果用utf-8编码，生成的txt是6个字节（就是6个8位的二进制数），用gbk编码生成的txt是4个字节。

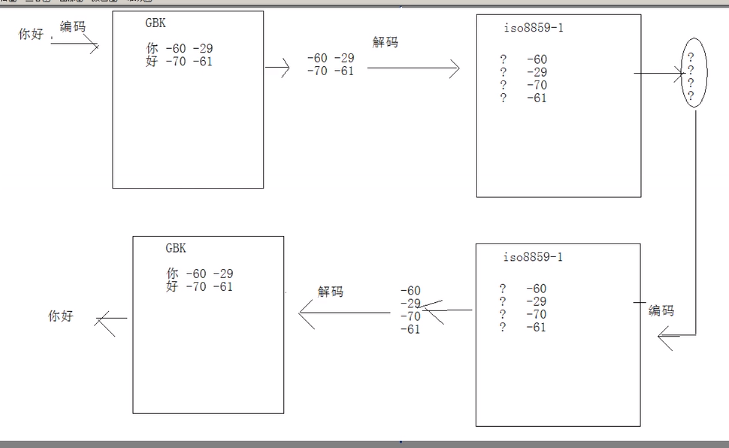
对“你好”分别用utf-8和gbk编码成utf.txt和gbk.txt。用对应的码表都能读出来，但是用utf-8的码表读gbk，gbk的4字节就会变成3字节+1字节去查表；同样用gbk的码表去读utf，utf的6字节就会变成2字节+2字节+2字节去查表



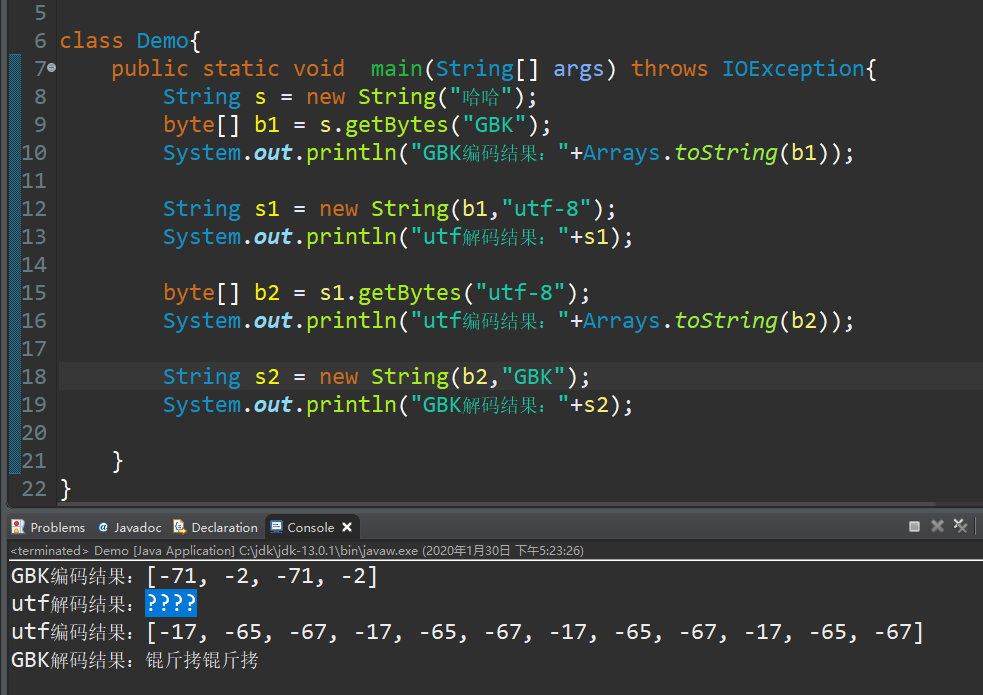
**P275 字符编码**



在中国常用的只有两个码表：GBK和UTF-8.通常编码没问题。因为想你要将中文汉字转换成字节数组，只能用GBK和UTF-8，用其他的码表会因为不识别中文而报错。但是解码的时候，因为对象是字节数组，任何一张码表都能够对数组进行解码。

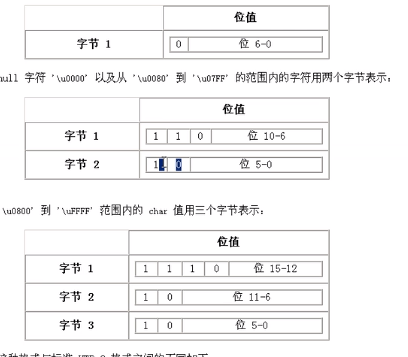


如果使用的编码表是iso8859-1，因为不识别中文，你可以再编一次，解一次得到结果。但是如果你使用的是gbk编码，对方是utf-8，就无法再编一次解一次。



**P276 字符编码-联通**

UTF-8编码，最多3个字节。那么它是如何判断读取1个字节去查表，还是读取2个字节去查表，还是读取3个字节去查表？



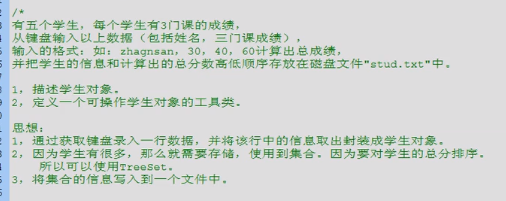
0打头，读一个字节就去查表

110打头，读两个字节才去查表

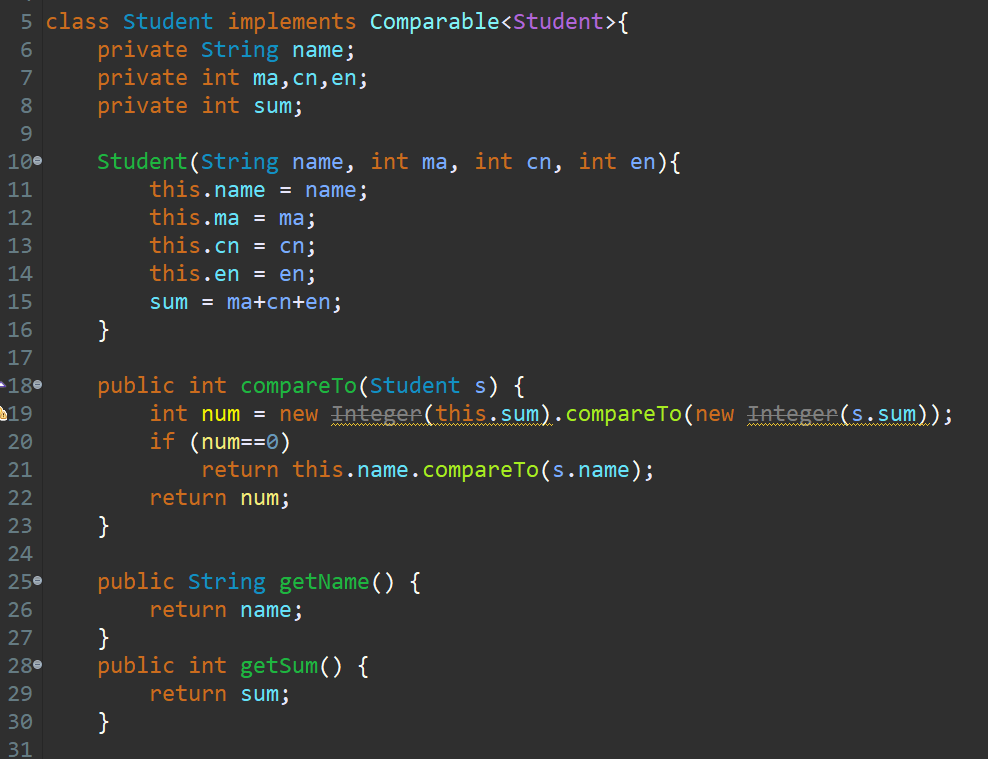
1110打头，读三个字节才去查表

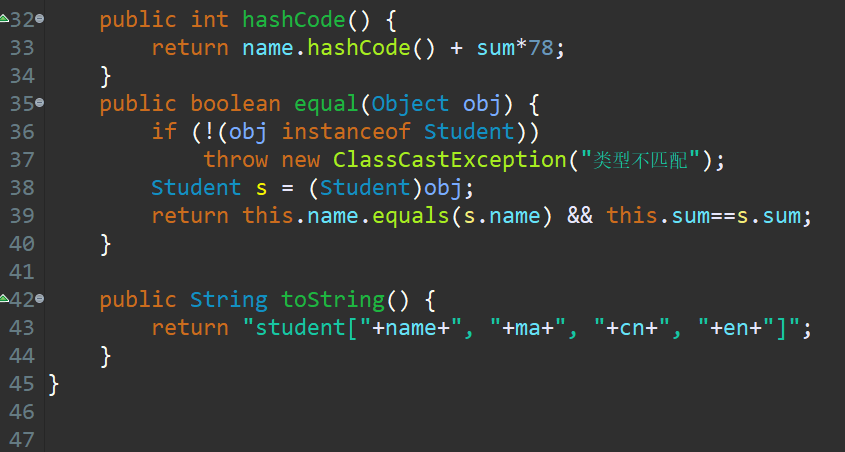
**P277 字符编码-练习**

需求及分析



定义一个可以根据总分比较的学生类





然后定义操作学生的工具类，定义两个方法，getStudents用于获取键盘录入，并把录入的学生对象放入有序集合。Write2File将传入的集合写入磁盘。



主程序

