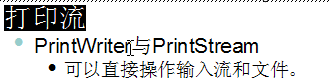
# IO包中的其他类

## 打印流



/\*

打印流：

该流提供了打印方法，可以将各种数据类型的数据都原样打印。

字节打印流：

PrintStream

构造函数可以接收的参数类型：

1，file对象。File

2，字符串路径。String

3，字节输出流。OutputStream

字符打印流：

PrintWriter

构造函数可以接收的参数类型：

1，file对象。File

2，字符串路径。String

3，字节输出流。OutputStream

4，字符输出流，Writer。

\*/

import java.io.\*;

class PrintStreamDemo

{

public static void main(String[] args) throws IOException

{

BufferedReader bufr =

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

PrintWriter out = new PrintWriter(new FileWriter("a.txt"),true); // true，则println、printf 或 format 方法将刷新输出缓冲区，这样我们调用了这些方法之后就不需要手动地去调用flush()方法了

String line = null;

while((line=bufr.readLine())!=null)

{

if("over".equals(line))

break;

out.println(line.toUpperCase());

//out.flush();

}

out.close();

bufr.close();

}

}

## 序列流



SequenceInputStream 它是InputStream的子类，这个类没有对应的outputSream类。它的功能是把多个输入流合并成一个，示例如下：

把3个txt文件合并成一个：

class SequenceDemo {

public static void main(String[] args) throws IOException{

Vector<FileInputStream> v = new Vector<FileInputStream>();

v.add(new FileInputStream("c:\\1.txt"));

v.add(new FileInputStream("c:\\2.txt"));

v.add(new FileInputStream("c:\\3.txt"));

Enumeration<FileInputStream> en = v.elements();

SequenceInputStream sis = new SequenceInputStream(en);

FileOutputStream fos = new FileOutputStream("c:\\4.txt");

byte[] buf = new byte[1024];

int len =0;

while((len=sis.read(buf))!=-1) {

fos.write(buf,0,len);

}

fos.close();

sis.close();

}

}

**切割文件，然后再合并：**

class SplitFile {

public static void main(String[] args) throws IOException{

//splitFile();

merge();

}

public static void merge()throws IOException{

ArrayList<FileInputStream> al = new ArrayList<FileInputStream>();

for(int x=1; x<=3; x++){

al.add(new FileInputStream("c:\\splitfiles\\"+x+".part"));

}

final Iterator<FileInputStream> it = al.iterator();

// 因为Vector效果低，所以这里使用List的方式来存储，但是List没有返回枚举的方法，所以这里实现了一个枚举类，这样这个枚举类就可以访问List列表。

Enumeration<FileInputStream> en = new Enumeration<FileInputStream>() {

public boolean hasMoreElements(){

return it.hasNext();

}

public FileInputStream nextElement() {

return it.next();

}

};

SequenceInputStream sis = new SequenceInputStream(en);

FileOutputStream fos = new FileOutputStream("c:\\splitfiles\\0.bmp");

byte[] buf = new byte[1024];

int len = 0;

while((len=sis.read(buf))!=-1) {

fos.write(buf,0,len);

}

fos.close();

sis.close();

}

public static void splitFile()throws IOException{

FileInputStream fis = new FileInputStream("c:\\1.bmp");

FileOutputStream fos = null;

byte[] buf = new byte[1024\*1024];

int len = 0;

int count = 1;

while((len=fis.read(buf))!=-1) {

fos = new FileOutputStream("c:\\splitfiles\\"+(count++)+".part");

fos.write(buf,0,len);

fos.close();

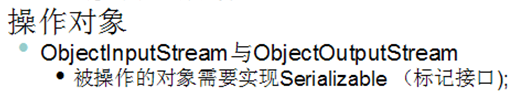
}

fis.close();

}

}

## 操作对象的的流



ObjectInputStream、ObjectOutputStream

这两个对象除了可以读写对象，还可以写基本数据类型(int, long, boolean等)

操作的对象必须实现Serializable接口，这个接口没有成员变量也没有方法，像这样的接口称为标记接口。

静态变量不能序列化

加transient修改符的非静态变量也不会被序列化

写出去的对象的文件名一般不要用.txt，因为这样可以双击打开，打开后又是乱码没什么用，还容易被别人修改，可以起一些无法打开的，如\*.obj

对象可以写多个，读的时候就按写入的序去调用读取方法读就可以了。

**class** Person **implements** Serializable{

String name;

**static** String *country* = "cn"; // 静态修饰的成员不会被序列化，因为静态的变量属于静态区，序列化只能是在堆内存中的

**transient** **int** age; // 如果不是静态的，但是也不想被序列化，则可加transient修饰

Person(String name,**int** age,String country) {

**this**.name = name;

**this**.age = age;

**this**.*country* = country;

}

**public** String toString() {

**return** name+":"+age+":"+*country*;

}

}

**class** ObjectStreamDemo {

**public** **static** **void** main(String[] args) **throws** Exception {

// writeObj();

*readObj*();

}

**public** **static** **void** readObj() **throws** Exception {

ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("obj.txt"));

Person p = (Person) ois.readObject();// 如果有写多个对象，则可以多次调用这个方法可顺序读出多个对象

System.*out*.println(p);

ois.close();

}

**public** **static** **void** writeObj() **throws** IOException {

ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("obj.txt"));

oos.writeObject(**new** Person("lisi0", 399, "kr"));

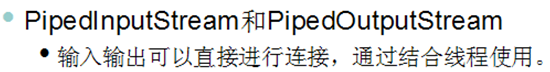
oos.close();

}

}

注：在序列化的时候Java虚拟机会根据Person类的成员变量来生成一个名叫serialVersionUID的值，如果Person类进行了修改，如改变name属性为private，则在读取对象将会失败，因为这个修改过后的Person它的serialVersionUID计算的时候将和以前保存的时候不同了，所以认为不是同一个对象，所以不能读取，如果想要修改后仍能读取，则可以在Person类中写死serialVersionUID的值，如：public static final long serialVersionUID = 42L;

## 管道流



以前使用的IO流中的输入流和输出流是没有直接联系的，是通过输入流把内容读取到一个byte数组或字符串或其它中转变量中保存，写的的时候再从这些变量中取出来写入文件。而管道流就可以把输入和输出流进行连接，从输出流中写数据，则从输入流中即可读取到数据。

PipedInputStream

管道输入流应该连接到管道输出流；管道输入流提供要写入管道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，因为这样可能死锁线程。管道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。如果向连接管道输出流提供数据字节的线程不再存在，则认为该管道已损坏。

因为输入流的read方法是阻塞式的，所以，如果用单线程的话，如果先执行了read方法，则read就一直等待，因为还没有写入任何东西，所以管道流是关系到多线程的IO流，一个线程读，一个线程写，这样不论是先执行读或是先执行写都没有问题。

示例：

**class** Read **implements** Runnable {

**private** PipedInputStream in;

Read(PipedInputStream in) {

**this**.in = in;

}

**public** **void** run() {

**try** {

**byte**[] buf = **new** **byte**[1024];

System.*out*.println("读取前。。没有数据阻塞");

**int** len = in.read(buf);

System.*out*.println("读到数据。。阻塞结束");

String s = **new** String(buf, 0, len);

System.*out*.println(s);

in.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException("管道读取流失败");

}

}

}

**class** Write **implements** Runnable {

**private** PipedOutputStream out;

Write(PipedOutputStream out) {

**this**.out = out;

}

**public** **void** run() {

**try** {

System.*out*.println("开始写入数据，等待6秒后。");

Thread.*sleep*(6000);

out.write("piped lai la".getBytes());

out.close();

} **catch** (Exception e) {

**throw** **new** RuntimeException("管道输出流失败");

}

}

}

**class** PipedStreamDemo {

**public** **static** **void** main(String[] args) **throws** IOException {

PipedInputStream in = **new** PipedInputStream();

PipedOutputStream out = **new** PipedOutputStream();

in.connect(out);

Read r = **new** Read(in);

Write w = **new** Write(out);

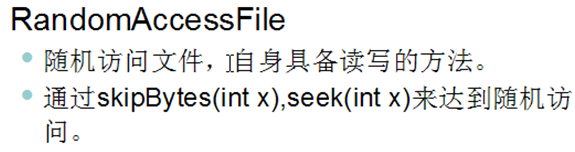
**new** Thread(r).start();

**new** Thread(w).start();

}

}

## RandomAccessFile



通过RandomAccessFile的write方法写的是一个字节，如果需要写的是一个int，则使用writeInt方法，它是将int的4个字节都写入，如writeInt(97)，打开记事本会发现有个字母a，97对应了字母a，a的前面有3个空格，加起来就是4个字节了，因为记事本软件读取这4个字节时去查GBK码表了，所以显示了3个空格和一个字母a。

**public** **static** **void** writeFile() **throws** IOException {

RandomAccessFile raf = **new** RandomAccessFile("ran.txt", "rw");

raf.write("李四".getBytes());

raf.writeInt(97);

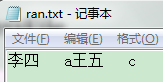
raf.write("王五".getBytes());

raf.writeInt(99);

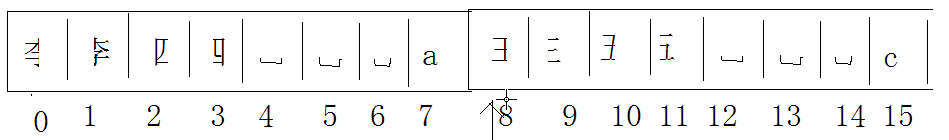
raf.close();

}

用此方法写入了李四和王五的年龄信息，用记本打开ran.txt文件如下：



在GBK编码中，一个汉字占两个字节，下面用字节数组的方式表示写入的情况：



把李四的信息读取出来：

**public** **static** **void** readFile() **throws** IOException {

RandomAccessFile raf = **new** RandomAccessFile("ran.txt", "r");

**byte**[] buf = **new** **byte**[4]; // 前面4个字节是李四

raf.read(buf);

String name = **new** String(buf);

**int** age = raf.readInt();

System.*out*.println("name=" + name);

System.*out*.println("age=" + age);

raf.close();

}

如果想跳过李四直接读取王五的信息，则只需要在读取前先把指针定位到索引为8（raf.seek(8)），然后开始读就是王五的数据了。假如有4个人，要取第4个人的，则可以：raf.seek(8 \* 3);

skipBytes()方法也能调整指针的位置，但是只能前进不能后退，而seek()方法既能前进也能后退。读的时候可以跳跃的读，写的时候也一样，如可以直接定位到王五后面写入一个张三的信息。

像这种写入信息，如果有些人的名字两个字，有些3个字，则在跳跃读取的时候就不好计算了，所以可以把名字统一使用16个字节来存储，即不够16位的用空格填充再写入文件，读的时候再把空格删除即可。对于年龄，用一个int足够了，用byte还是有点危险，因为byte只能表示-128 ~ 127，如果有人能活到128岁就装不下了。

RandomAccessFile有一个应该就是多线程下载，把一个文件分成多段，用多个线程去下载并写到相应的位置。

## 操作基本数据类型



可以用于操作基本数据类型的数据的流对象。

前面学的ObjectInputStream、ObjectOutputStream和RandomAccessFile也可以写基本数据类型，但是如果只是写数据类型的话就使用DataInputStream和DataOutputStream，它还有一个特殊方法:writeUTF()、readUTF。

**class** DataStreamDemo {

**public** **static** **void** main(String[] args) **throws** IOException {

// writeData();

// readData();

// writeUTFDemo(); // 这个方法写的数据用转换流无法正确读取，即使给转换流设置编码为UTF-8，查看writeUTF方法文档可知：以与机器无关方式使用 [UTF-8 修改版](mk:@MSITStore:G:\1JavaWeb资料\jdk6.ZH_cn.chm::/j2se6/api/java/io/DataInput.html#modified-utf-8)编码将一个字符串写入基础输出流。点击文档中的“[UTF-8 修改版](mk:@MSITStore:G:\1JavaWeb资料\jdk6.ZH_cn.chm::/j2se6/api/java/io/DataInput.html#modified-utf-8)”可查看它与UTF-8的区别。writeUTF方法写入“你好”是8个字节，用转换流写的话是6个字节。

// 用转换流来写入中文的方式

// OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("utf-8.txt"),"UTF-8");

// osw.write("你好");

// osw.close();

// readUTFDemo();

}

**public** **static** **void** readUTFDemo() **throws** IOException {

DataInputStream dis = **new** DataInputStream(**new** FileInputStream("utf.txt"));

String s = dis.readUTF();

System.*out*.println(s);

dis.close();

}

**public** **static** **void** writeUTFDemo() **throws** IOException {

DataOutputStream dos = **new** DataOutputStream(**new** FileOutputStream("utfdate.txt"));

dos.writeUTF("你好");

dos.close();

}

**public** **static** **void** readData() **throws** IOException {

DataInputStream dis = **new** DataInputStream(**new** FileInputStream("data.txt"));

**int** num = dis.readInt();

**boolean** b = dis.readBoolean();

**double** d = dis.readDouble();

System.*out*.println("num=" + num);

System.*out*.println("b=" + b);

System.*out*.println("d=" + d);

dis.close();

}

**public** **static** **void** writeData() **throws** IOException {

DataOutputStream dos = **new** DataOutputStream(**new** FileOutputStream("data.txt"));

dos.writeInt(234);

dos.writeBoolean(**true**);

dos.writeDouble(9887.543);

dos.close();

}

}

## 操作字节数组



这两个流可以称为“内存流”，因为它们操作的只是内存，不操作文件等其它IO设置。

原来学的字节流内部不就是封装了字节数组吗？为什么还要有这两个类呢？new ByteArrayInputStream的时候就需要把byte数组传给构造函数了，说明这个类跟底层IO资源没有联系，也就是不需要调close操作。查看ByteArrayInputStream的文档说明如下：

ByteArrayInputStream 包含一个内部缓冲区，该缓冲区包含从流中读取的字节。内部计数器跟踪 read 方法要提供的下一个字节。

关闭 ByteArrayInputStream 无效。此类中的方法在关闭此流后仍可被调用，而不会产生任何 IOException。

/\*

用于操作字节数组的流对象。

ByteArrayInputStream ：在构造的时候，需要接收数据源，。而且数据源是一个字节数组。

ByteArrayOutputStream： 在构造的时候，不用定义数据目的，因为该对象中已经内部封装了可变长度的字节数组。

因为这两个流对象都操作的数组，并没有使用系统资源。所以，不用进行close关闭。

在流操作规律讲解时：

源设备，

键盘 System.in，硬盘 FileStream，内存 ArrayStream。

目的设备：

控制台 System.out，硬盘FileStream，内存 ArrayStream。

用流的读写思想来操作数组。

\*/

**import** java.io.\*;

**class** ByteArrayStream {

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

// 数据源。

ByteArrayInputStream bis = **new** ByteArrayInputStream("ABCDEFD".getBytes());

// 数据目的

ByteArrayOutputStream bos = **new** ByteArrayOutputStream();

**int** by = 0;

**while** ((by = bis.read()) != -1) {

bos.write(by);

}

System.*out*.println(bos.size());

System.*out*.println(bos.toString());

// bos.writeTo(new FileOutputStream("a.txt"));

}

}

## 操作字符数组



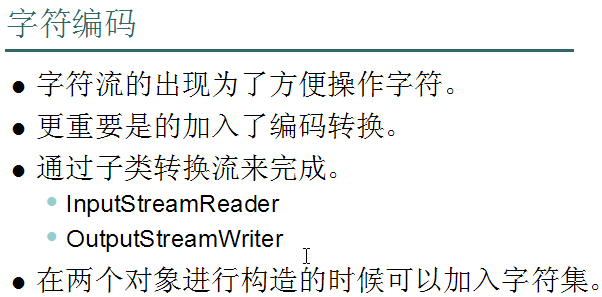
跟ByteArrayInputStream和ByteArrayOutputStream原理相同

## 操作字符串



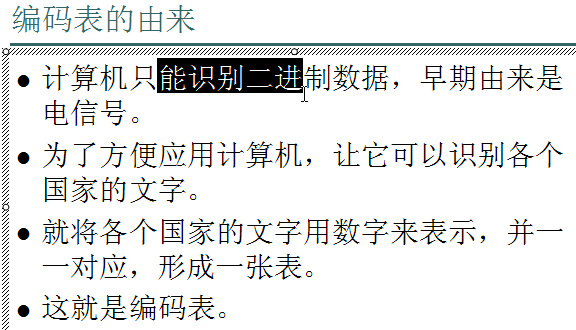
跟ByteArrayInputStream和ByteArrayOutputStream原理相同

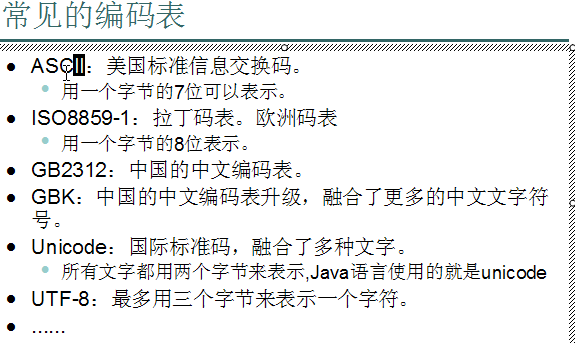
# 字符编码



字符流的出现是为了方便操作字符，因为字符流的内部加入了编码表，InputStreamReader和OutputStreamWriter内部封装了编码表，还有另外两个流内部也封装了编码表：PrintStream和PrintWriter，但是这两个只能打印，不能读取。所以玩编码表还是得用转换流。

字符与字节的转换就是通过InputStreamReader和OutputStreamWriter来完成的，InputStreamReader就是读取字节，把字节转换为字符，用于显示，而OutputStreamWriter则是把字符转换为字节保存到硬盘或其他存储设备上。不论是字节转成字符，还是字符转成字节，它们都需要去查编码表来成。 在做Web项目时，如数据传输，如果传输的数据是字符数据，则就要想这些传过来的字符数据肯定是字节，这些字节我应该用什么码表去查找出对应的字符来，这样才不会有乱码，如服务器是把字符用UTF-8编码转换为字节传给了客户端，如果客户端拿到了这些字节并且使用GBK码表来查找这些字节对应的字符，很显示会是乱码。所以，别人给你传了一段字节数据，你需要把这些字节显示成正确的字符，那你就要问给你传字节数据的人，它是用什么码表把字符转换为字节传过来的。





转换流原理：

OutputStreamWriter out = new OutputStreamWriter(new FileOutputStream(“abc.txt”), “UTF-8”);

String data = “你好”;

out.write(data, 0, data.length());

原理是：

Java中的字符是用Unicode编码的，也就是说char是使用Unicode编码的

上面转换流使用UTF-8的方式去把字符写入文件，其实就是去“UTF-8”码表中查找“你好”对应的编码，把这个编码写入文件。为什么记事本打开这个文件看到的不是编码而是文字呢？是因为记事本软件就是用来处理字符的，它知道这些编码是UTF-8码表的，就去UTF-8码表中查找到对应的字符显示出来。

假如用GBK编码保存了“你好”，则是保存了4个字节，因为在GBK编码中用两个字节来表示1个中文字符，所以保存的时候“你好”会有4个字节。如果此时用UTF-8来读取，则UTF-8是3个字节一个中文，所以会拿前面3个字节去UTF-8查找对应的字符，发现没找到有对应的字符，也就是说这个编码在UTF-8编码表中是没有使用的，换句话说UTF-8编码表中有一些编码是没有按排对应的字符的，这些没使用的编码就用“？”来与之对应，这个问号就是所谓的点占位符，很多的码表都一样，如果一个编码在这个码表中没有被使用，则使用？来占位。所以“你好”的前3个字节去UTF-8码表查找是一个未使用的编码，查出来的是一个占位符“？”，然后拿第4个字节去查找也是查出了一个占位符，所以显示结果为两个问号。假如用UTF-8编码来保存“你好”，则将会有6个字节，如果此时用GBK码表来读取这6个字节，则会显示3个字符，每两个字节拿去GBK编码表查找对应的字符。

## 改变Java默认字符编码

默认字符集是Java环境中的file.encoding属性，查看System.*getProperties*()可得到这个属性，虚拟机运行时读的就是这个属性。

Class<Charset> clazz = Charset.**class**;

Field field = clazz.getDeclaredField("defaultCharset");

field.setAccessible(**true**);

field.set(Charset.**class**, Charset.*forName*("GBK"));

String csn = Charset.*defaultCharset*().name();

System.*out*.println("defaultCharset = " + csn);

## 字符编码、解码

编码：字符串变成字节数组。

解码：字节数组变成字符串。

String-->byte[]; str.getBytes()用本地编码，str.getBytes(String charsetName); 用指定的字符编码表

byte[] -->String: new String(byte[]);用本地编码，new String(byte[],String charsetName); 用指定的字符编码表

## 如何查看字符的编码：

String s = "哈哈";

byte[] b1 = s.getBytes("GBK");

System.out.println(Arrays.toString(b1));

运行结果如下：



String s = "哈哈";

byte[] b1 = s.getBytes("GBK"); // 编码，保存到文件中的就是这些byte字节

System.out.println(Arrays.toString(b1));

String s1 = new String(b1,"GBK"); // 解码，把字节变成字符串

System.out.println("s1="+s1);

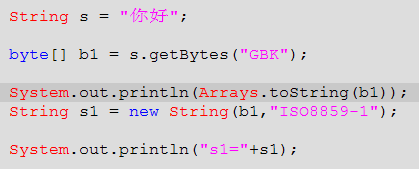
像中文的编码不是用GBK就是用UTF-8，所以如果发现乱码则分别使用GBK和UTF-8都试一下就知道用哪个码表来解码了。但是假如别人在编码的时候使用的是其它的编码表来编码的（如ISO8859-1），这又该如何解码呢？如下代码：

String s = "哈哈";

byte[] b1 = s.getBytes("ISO8859-1");

像这种编码已经编错了，没法再解码了，因为ISO8859-1这个码表根本就不能识别中文，它会去找与之比较像的字符的编码返回，所以这个过程是不可逆的，无法解码。

如果编码时的指定的码表是正确的，只是解码时指定的码表错误，这是可解的，如下：

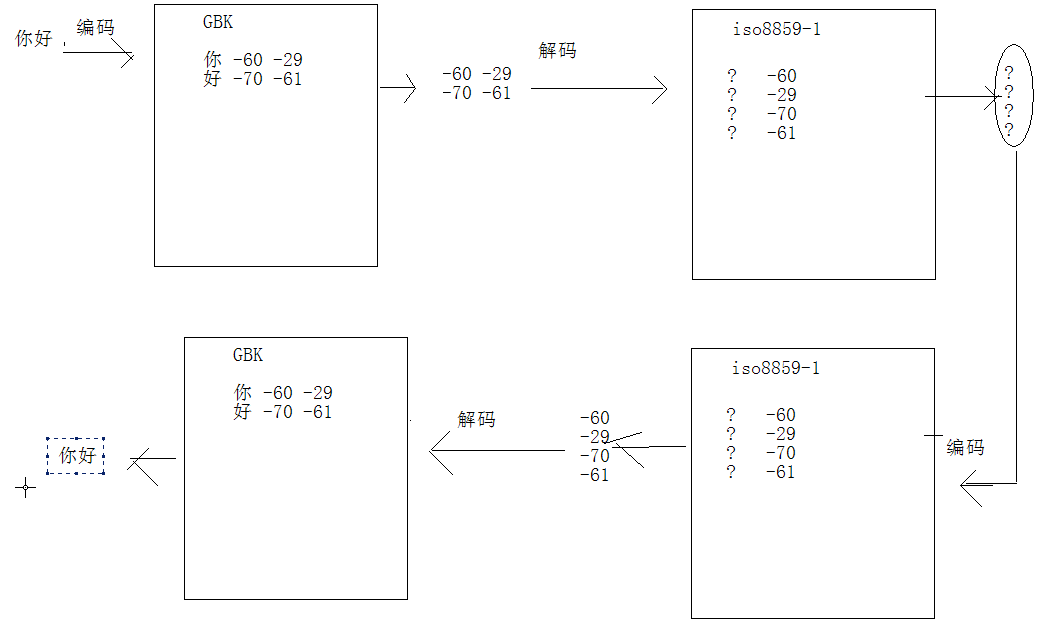


运行结果为：

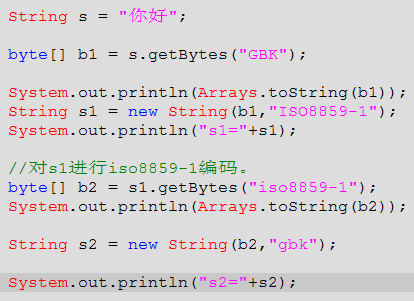


解码时显示了4个问号，这是因为它把GBK的4个字节分别拿去ISO8859-1的表中进行字符查找，而这4个字节在ISO8859-1中都是没有对应字符的，也就是说这4个字节在ISO8859-1码表中是没有使用的，对于没有使用的编码都是用一个占位符表示，即用？号来填充未使用的编码，所以看到结果显示了4个问号。

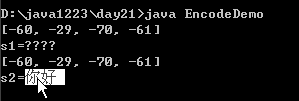
上面这种情况是可以解码的，查出来的4个问号还是可以转换回原来的编码，是可逆的，原理如下：



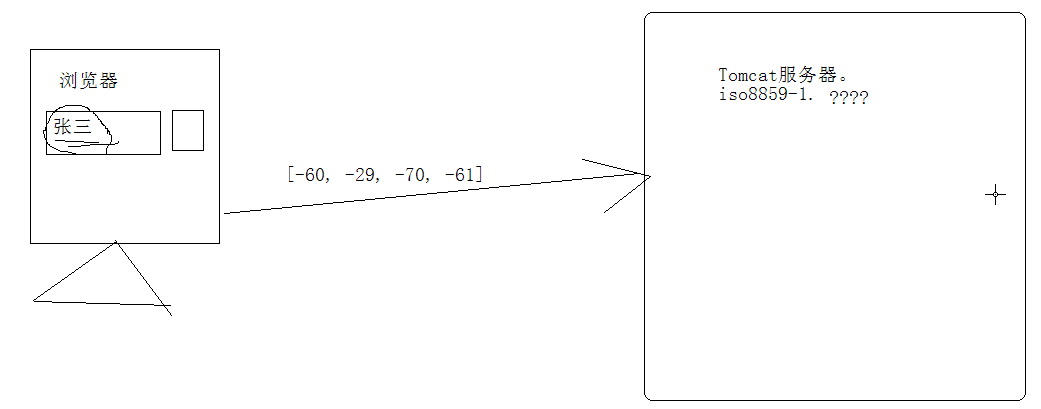
也就是说看起来一模一样的4个问号还是可以还原为原来GBK的编码，用后再用GBK编码表来读取就是正确的中文了。解码的代码如下：



运行结果如下：

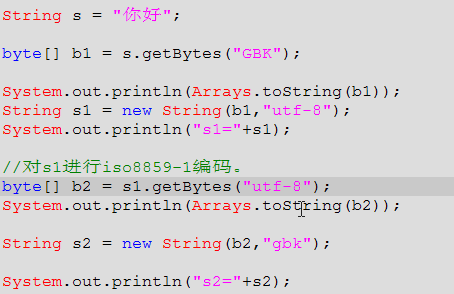


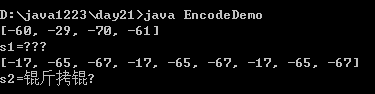
现实例子：



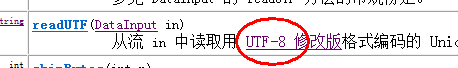
浏览器把中文的GBK编码发给了服务器，而Tomcat服务器是外国人写的，默认使用iso8859-1来解码，所以服务器接收到的时候是4个问号，解决的办法就是拿着这4个问题用iso8859-1编码成字节，这些字节就是GBK编码的字节了，这时再用GBK编码了解码即可。如果用的是GET请求，则必须要这样来解码。如果用的是POST请求，则可以在服务器通过设置一个对象的setCharactorEndcoding方法来指定使用GBK来解码。

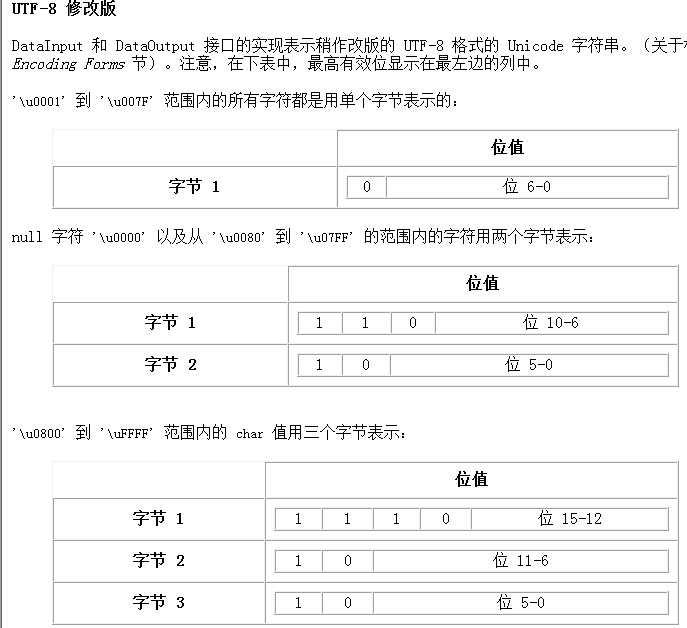
假如编码的时候使用UTF-8来编码，那还能像ISO8859-1那样来解码吗？代码如下：





从运行结果可知是不行的，因为UTF-8会把一个问号返回3个编码，这样就得不到原来的原始编码了。这里UTF-8去解码4个字节为什么返回了3个问号？因为UTF-8对于不同的字符使用不同的长度的字节来表示，如字母用一个字节即可，那使用UTF-8解码的时候怎么知道要读几个字节来表示一个字符呢？查看DataInputStream的如下方法：



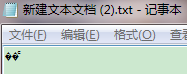


可看到字节1的前面的位就表达了要连续读几个字节来查码表，如果第1位是0，则读一个字节，如果第1、2位是1，则读两个字节，如果第1、2、3位是1，则读3个字节。如下面的一连串字节，会被识别为3个整体来查码表：

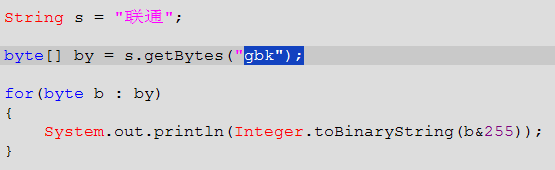


## 联通问题

新建一个文本文件，打开，输入“联通”，关闭，再打开，发觉看到的是乱码，如下：



这是为什么呢？思想：输入“联通”，保存的是候用的是GBK默认编码，编码没有问题，问题在于重新打开的解码出了问题，把联通的GBK编码转换成二进制看一下：



&255是为了只保留后8位

打印结果如下：



这个二进制组合刚好是UTF-8编码的格式，所以解码的时候是按UTF-8编码去读，所以看到了乱码。

解决方式：在记事本输入“你联通”，关闭再打开，发现没有乱码，这是因为记事本读取第一个字符的时候不符合UTF-8编码，所以用GBK去读了。