**编码和解码**

## 一、字符与编码的发展

|  |  |  |
| --- | --- | --- |
|  | **系统内码** | **说明** |
| 阶段一 | **ASCII** | 计算机刚开始只支持英语，其它语言不能够在计算机上存储和显示。ASCII码一共规定了128个字符的编码，比如空格“SPACE”是32（二进制00100000），大写的字母A是65（二进制01000001）。这128个符号（包括32个不能打印出来的控制符号），只占用了一个字节的后面7位，最前面的1位统一规定为0。 |
| 阶段二 | **ANSI编码（本地化）** | 为使计算机支持更多语言，通常使用 0x80~0xFF 范围的 2 个字节来表示 1 个字符。比如：汉字 '中' 在中文操作系统中，使用 [0xD6,0xD0] 这两个字节存储。不同的国家和地区制定了不同的标准，由此产生了 GB2312, BIG5, JIS 等各自的编码标准。这些使用 2 个字节来代表一个字符的各种汉字延伸编码方式，称为 ANSI 编码。在简体中文系统下，ANSI 编码代表 GB2312 编码，在日文操作系统下，ANSI 编码代表 JIS 编码。不同 ANSI 编码之间互不兼容，当信息在国际间交流时，无法将属于两种语言的文字，存储在同一段 ANSI 编码的文本中。 |
| 阶段三 | **UNICODE（国际化）** | 为了使国际间信息交流更加方便，国际组织制定了 UNICODE 字符集，为各种语言中的每一个字符设定了统一并且唯一的数字编号，以满足跨语言、跨平台进行文本转换、处理的要求。Unicode当然是一个很大的集合，现在的规模可以容纳100多万个符号。每个符号的编码都不一样，比如，U+0639表示阿拉伯字母Ain，U+0041表示英语的大写字母A，U+4E25表示汉字“严”。需要注意的是，Unicode只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。 |

**1 在接下去讨论之前，首先要明确几个概念**

（1）字符是我们可见和可以理解的符号，字节是计算机存储字符的形式。

（2）计算机在把字符存储到字节的时候需要使用某种编码方式，以便在重现字符的时候使用这种已知的编码方式解码。因此保存和打开都存在选择编码方式的问题。由上表可以看出，类似GB2312这样的编码属于ANSI规范中的一种。在windows的记事本中我们可以选择ANSI编码方式保存文本（默认用这种编码方式），而在不同语言版本的windows系统中ANSI编码方式是不同的；在简体中文系统中，记事本所指的ANSI就是GB2312。因此如果在英文系统下，使用默认的保存方式（ANSI）保存含有中文字符的文本，记事本将会给出提示，如果此时不予理会的话，中文信息将会丢失。当使用记事本打开一个文件时，记事本将自动检测当前文件的编码方式，并使用对应的编码方式解码，以重现文字符，当然默认使用当前系统语言环境下的ANSI编码。

**2 Unicode的实现---UTF-8**

互联网的普及，强烈要求出现一种统一的编码方式。UTF-8就是在互联网上使用最广的一种unicode的实现方式。其他实现方式还包括UTF-16和UTF-32，不过在互联网上基本不用。重复一遍，这里的关系是，UTF-8是Unicode的实现方式之一。

UTF-8最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度。

**3 UTF-8的编码规则很简单，只有二条**：

1）对于单字节的符号，字节的第一位设为0，后面7位为这个符号的unicode码。因此对于英语字母，UTF-8编码和ASCII码是相同的。

2）对于n字节的符号（n>1），第一个字节的前n位都设为1，第n+1位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的unicode码。（这里有人可能有疑问，n不是最多2吗。其实Unicode字符集码不一定是上面举例的两个字节，可能多于2的。）

下表总结了编码规则，字母x表示可用编码的位。

**Unicode符号范围 | UTF-8编码方式**   
**(十六进制) | （二进制）   
--------------------+---------------------------------------------**   
0000 0000-0000 007F | 0xxxxxxx   
0000 0080-0000 07FF | 110xxxxx 10xxxxxx   
0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx   
0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

下面，还是以汉字“严”为例，演示如何实现UTF-8编码。

已知“严”的unicode是4E25（100111000100101），根据上表，可以发现4E25处在第三行的范围内（0000 0800-0000 FFFF），因此“严”的UTF-8编码需要三个字节，即格式是“1110xxxx 10xxxxxx 10xxxxxx”。然后，从“严”的最后一个二进制位开始，依次从后向前填入格式中的x，多出的位补0。这样就得到了，“严”的UTF-8编码是“11100100 10111000 10100101”，转换成十六进制就是E4B8A5。

**4 windows记事本程序可选的编码方式有**

**ANSI，Unicode，Unicode big endian 和 UTF-8。**

1）ANSI是默认的编码方式。对于英文文件是ASCII编码，对于简体中文文件是GB2312编码（只针对Windows简体中文版，如果是繁体中文版会采用Big5码）。

2）Unicode编码指的是UCS-2编码方式，即直接用两个字节存入字符的Unicode码（对于大于2个字节的字符无法存储。UTF-16扩充了Unicode，包括了一些稀有字符，想我们国家的满文，藏文等等，两者基本上等价）。这个选项用的little endian格式。

3）Unicode big endian编码与上一个选项相对应。下一节会解释little endian和big endian的涵义。

4）UTF-8编码，也就是上面谈到的编码方法。

**5 Little endian和Big endian**

上一节已经提到，Unicode码可以采用UCS-2格式直接存储。以汉字”严“为例，Unicode码是4E25，需要用两个字节存储，一个字节是4E，另一个字节是25。存储的时候，4E在前，25在后，就是Big endian方式；25在前，4E在后，就是Little endian方式。

**6 自动检测编码方式---BOM**

Unicode规范中定义，每一个文件的最前面分别加入一个表示编码顺序的字符，这个字符的名字叫做”零宽度非换行空格“（ZERO WIDTH NO-BREAK SPACE），用FE FF表示。这正好是两个字节，而且FF比FE大1。如果一个文本文件的头两个字节是FE FF，就表示该文件采用大头方式；如果头两个字节是FF FE，就表示该文件采用小头方式。在windows中Unicode编码中表示字节排列顺序的这个个文件头，也叫做BOM（byte-order mark），FFFE和FEFF就是不同的BOM。例如：

1）Unicode：FF FE 表明是小头方式存储。

2）Unicode big endian：FE FF 表明是大头方式存储。

3）UTF-8：EF BB BF 表示这是UTF-8编码。

**7 一些思考和体会**

（1）UTF-8之所以被广泛接受，可能是因为它的统一性和卓越的“压缩”能力。首先，它是Unicode的一种实现，即可以实现国际化；其次，对于英文的存储它只用一个字节即可，而UTF-32不管什么都用4个字节、Unicode(UTF-16)都用两个字节等之辈都不如它省空间。而对于中文比较多的文档使用UTF-8未必有多好的效果，很多中文常用字都用3个字节存储。

（2）可以利用Unicode作为中间编码方式，实现各种编码之间的转化。详见下面所述的问题2。

**8 .NET下与字符编码相关的编程**

Encoding类用于处理编码的相关问题。

EncodingInfo[] Encoding.GetEncodings()

这是个静态方法。返回当前系统包含的所有编码方式，每种编码方式有名字、显示名字、对应的CodePage值。

Encoding.GetEncoding(string name)

这是个静态方法。这个方法还有多个重载，根据编码的名﻿称返回编码对象。

﻿Encoding.Default/Encoding.Unicode...

静态属性。提供一种便捷的方式返回一些常用的编码对象，其中﻿Encoding.Default返回当前系统设置的本地化编码方式，简体中文的window系统下应该等同于GB2312。

byte[] Encoding.GetBytes(string s)

这是个实例方法。假如实例对象是对应UTF-8编码的对象，那么这个方法表示将字符串s，以UTF-8的编码方式编码成字节数组。（由于window内核使用了Unicode字符集，因此string在内存中是一Unicode方式存储的，这个方法实际上做的事情便是将Unicode转化成UTF-8）

string Encoding.GetString(byte[] bytes)

这是个实例方法。假如实例对象是对应UTF-8编码的对象，那么这个方式表示将字符数组bytes，以UTF-8的编码方式解码成字符串。（也就是UTF-8转化成Unicode）这个方法经常用于将流按照某种编码转化成字符串。

**问题1：判断字符是否是中文**

看到有同仁是这样做的：  
static public bool IsChina(char chr)  
{

　　if (Convert.ToInt32(chr) < Convert.ToInt32(Convert.ToChar(128)))

　　　　return false;

　　else

　　　　return true;  
}

这里显然是错的，字符不是只有中文、英文和数字。我查阅了Unicode编码表，得出的结论是在Unicode字符集中中文的范围应该在0x4E00-0x9FC3之间，所以光判断大于128显然不正确。还有强调一下，之所以查阅Unicode而不是GB2312或是UTF-8编码表，因为char和string在windows管理的内存中都是Unicode方式存储的。

**问题2：如何将http请求返回的包含网页代码的流正确解码**

这个问题主要源于编码方式不统一。在互联网不发达的时候，本地化的编码层出不穷，导致当互联网发展起来后，出现了编码不统一的问题，很多网站只考虑本地化的编码，国内包括百度在内的网页的charset都是gb2312。一些国际化的网站一般使用UTF-8，比如微软的，谷歌的。（如果http请求[www.google.com](http://www.google.com/" \t "https://blog.csdn.net/bbjjqq/article/details/_blank)会发现charset是big5的，但是用IE打开是charset变成UTF-8，不解）。因此，如果我们编程请求一个网页，那么返回的是流，如何才能用合适编码来解码这个流呢？思路其实很简单：对于一般的网页代码都有meta、都有charset，而英文字符在各种编码方式中是兼容的，我们可以用任何一种编码方式先把流解码出来编程string，然后用正则表达式匹配charset，得到网页的编码方式，然后再将string用之前的编码方式编码回字节流，再用匹配出来的编码方式解码这个字节流就可以了。

这种方法对于没有charset和网页无能为力！关键代码：

...

responseStream = webRequest.GetResponse().GetResponseStream();  
string ResultStringContent;  
string PreStringContent;  
using (StreamReader sr = new StreamReader(responseStream, encoding))//encoding可以是任何一种编码方式，一般可以是Encoding.Default  
{  
    PreStringContent = sr.ReadToEnd();  
    byte[] byteContent = encoding.GetBytes(PreStringContent);//用这种编码编码这个string，得到原始byte[]  
    ResultStringContent = TryToGetEncoding(PreStringContent, encoding).GetString(byteContent);//TryToGetEncoding见下面  
}

private static Encoding TryToGetEncoding(string content, Encoding tryEncoding)  
{  
    /\*try to parse html\*/  
    MatchCollection mc = Regex.Matches(content, "<meta [^>]\*>", RegexOptions.IgnoreCase);  
    foreach (Match m in mc)  
    {  
        if (m != null && m.Value != string.Empty)  
        {  
            Match mm = Regex.Match(m.Value, @"charset[ /t]\*=[ /t]\*[/w-\_]+");  
            if (mm != null && mm.Value != string.Empty)  
            {  
                string s = mm.Value;  
                int start = s.IndexOf('=') + 1;   
                Encoding retEncoding = null;  
                try  
                {  
                    retEncoding = Encoding.GetEncoding(s.Substring(start, s.Length - start).Trim());  
                }  
                catch  
                {  
                }  
                if (retEncoding != null)  
                    return retEncoding;  
            }  
        }  
    }  
    return tryEncoding;  
}

**问题3：StreamReader读取流或文件乱码**

StreamReader实际上是一个带有编解码功能的流读取类，所以你可以看到它有ReadToEnd()方法可以直接返回string、它的Read方法返回的不是Byte[]，而是char[]。

StreamReader读取乱码用一句话归纳那肯定是：编码问题！但这个答案太概括了。其实本质是编码问题，但实际上是我们没有注意到使用StreamReader时的细节。

在对StreamReader做了一些测试实验后，我总结如下：

StreamReader的构造函数

注意到StreamReader的构造函数有多达10个重载。其中有4个关键的构造参数，还有一个参数跟编码无关这里不讨论：

1）Stream stream：传递一个流对象给StreamReader

2）string path：传递一个文件路径给StreamReader

3) bool detectEncodingFromByteOrderMarks：指定是否自动检测BOM，只针对Unicode(L/B)、UTF-8、UTF-16(L/B)、UTF-32(L/B)编码

4）Encoding encoding：指定编码方式

这四个参数相互的组合，构造出来多达10个的重载，这些组合使得StreamReader的工作方式令人疑惑不解，这也是出现乱码的根本原因，现在我们来一一解读：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **原型** | **文件的编码** | | |
| **UTF-8** | **Default(GB2312)** | **Unicode(L)** |
| 1 | public StreamReader(Stream stream) 默认将encoding设置为UTF-8，自动检测BOM | 正常 | 乱码（因为没有BOM，用UTF-8解码） | 正常（有BOM） |
| 2 | public StreamReader(string path) 默认将encoding设置为UTF-8（这里MSDN说The default character encoding is used，实验证明默认是UTF-8），自动检测BOM | 正常 | 乱码（因为没有BOM，用UTF-8解码） | 正常（有BOM） |
| 3 | public StreamReader(  Stream stream,  bool detectEncodingFromByteOrderMarks ) 默认将encoding设置为UTF-8，可设置检测BOM，默认检测 | 正常 | 乱码（因为没有BOM，用UTF-8解码） | 如果设置不检测会是乱码，如果检测则正常 |
| 4 | public StreamReader(  Stream stream,  Encoding encoding ) 先检测BOM，如果有BOM则忽略encoding参数，如果没有则应用encoding | 正常 | 在encoding设置为Encoding.Default时正常 | 正常 |
| 5 | public StreamReader(  string path,  bool detectEncodingFromByteOrderMarks ) 默认将encoding设置为UTF-8，可设置检测BOM，默认检测 | 正常 | 乱码（因为没有BOM，用UTF-8解码） | 如果设置不检测会是乱码，如果检测则正常 |
| 6 | public StreamReader(  string path,  Encoding encoding ) 先检测BOM，如果有BOM则忽略encoding参数，如果没有则应用encoding | 正常 | 在encoding设置为Encoding.Default时正常 | 正常 |
| 7 | public StreamReader(  Stream stream,  Encoding encoding,  bool detectEncodingFromByteOrderMarks ) 先看detectEncodingFromByteOrderMarks是否为true，如果true那么行为同4，否则直接用encoding解码 | 略 | 略 | 略 |
| 8 | public StreamReader(  string path,  Encoding encoding,  bool detectEncodingFromByteOrderMarks ) 先看detectEncodingFromByteOrderMarks是否为true，如果true那么行为同6，否则直接用encoding解码 | 略 | 略 | 略 |

由上表的总结可以看出，自动检测BOM的优先级最高，如果取消检测或者没有BOM则应用Encoding，如果参数提供了Encoding那么应用参数的Encoding，否则就用UTF-8。

所以最后的一个终极解决方案就是StreamReader sr = new StreamReader(fileStream,Encoding.Default)。另外，对于上表的第二条，我的实验结果跟MSDN的描述有出入，有待考证。

## 二、深入解析Java中的编码转换以及编码和解码

**1 Java编码转换过程**  
 我们总是用一个java类文件和用户进行最直接的交互（输入、输出），这些交互内容包含的文字可能会包含中文。无论这些java类是与数据库交互，还是与前端页面交互，他们的生命周期总是这样的：  
 (1)、程序员在操作系统上通过编辑器编写程序代码并且以.java的格式保存操作系统中，这些文件我们称之为源文件。  
 (2)、通过JDK中的javac.exe编译这些源文件形成.class类。  
 (3)、直接运行这些类或者部署在WEB容器中运行，得到输出结果。

**2 这些过程是从宏观上面来观察的，了解这个肯定是不行的，我们需要真正来了解java是如何来编码和被解码的：**

**第一步：当我们用编辑器编写java源文件，**程序文件在保存时会采用操作系统默认的编码格式（一般我们中文的操作系统采用的是GBK编码格式）形成一个.java文件。java源文件是采用操作系统默认支持的file.encoding编码格式保存的。下面代码可以查看系统的file.encoding参数值。

[?](https://www.jb51.net/article/79921.htm)

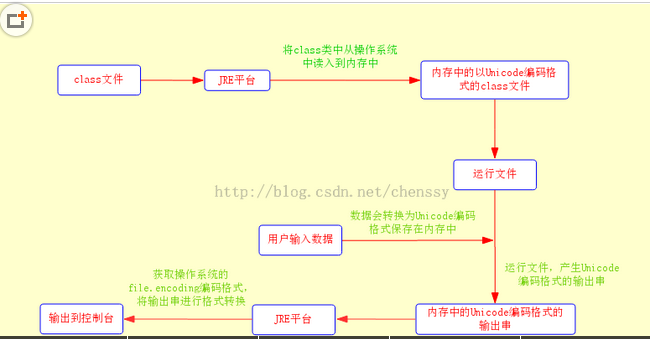
|  |  |
| --- | --- |
| 1 | System.out.println(System.getProperty("file.encoding")); |

**第二步：当我们使用javac.exe编译我们的java文件时**，JDK首先会确认它的编译参数encoding来确定源代码字符集，如果我们不指定该编译参数，JDK首先会获取操作系统默认的file.encoding参数，然后JDK就会把我们编写的java源程序从file.encoding编码格式转化为JAVA内部默认的UNICODE格式放入内存中。

**第三步：JDK将上面编译好的且保存在内存中信息写入class文件中**，形成.class文件。此时.class文件是Unicode编码的，也就是说我们常见的.class文件中的内容无论是中文字符还是英文字符，他们都已经转换为Unicode编码格式了。  
 在这一步中对对JSP源文件的处理方式有点儿不同：WEB容器调用JSP编译器，JSP编译器首先会查看JSP文件是否设置了文件编码格式，如果没有设置则JSP编译器会调用调用JDK采用默认的编码方式将JSP文件转化为临时的servlet类，然后再编译为.class文件并保持到临时文件夹中。

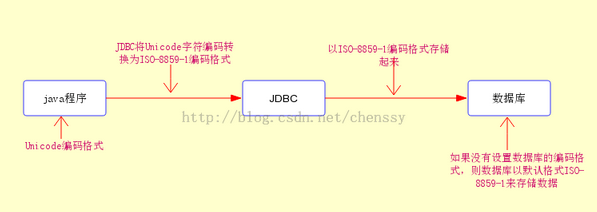
**第四步：运行编译的类：在这里会存在一下几种情况**  
 (1)、直接在console上运行。  
 (2)、JSP/Servlet类。  
 (3)、java类与数据库之间。  
 这三种情况每种情况的方式都会不同

1.Console上运行的类  
 这种情况下，JVM首先会把保存在操作系统中的class文件读入到内存中，这个时候内存中class文件编码格式为Unicode，然后JVM运行它。如果需要用户输入信息，则会采用file.encoding编码格式对用户输入的信息进行编码同时转换为Unicode编码格式保存到内存中。程序运行后，将产生的结果再转化为file.encoding格式返回给操作系统并输出到界面去。整个流程如下：

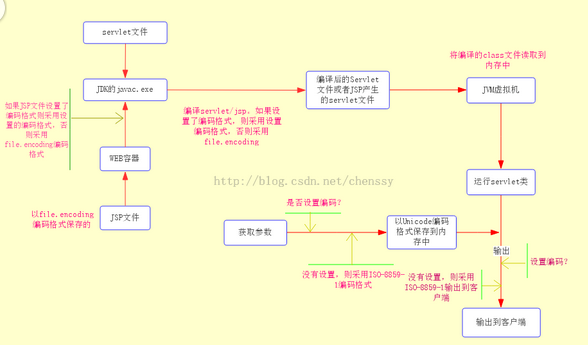


在上面整个流程中，凡是涉及的编码转换都不能出现错误，否则将会产生乱码。

2.Servlet类  
 由于JSP文件最终也会转换为servlet文件（只不过存储的位置不同而已），所以这里我们也将JSP文件纳入其中。当用户请求Servlet时，WEB容器会调用它的JVM来运行Servlet。首先JVM会把servlet的class加载到内存中去，内存中的servlet代码是Unicode编码格式的。然后JVM在内存中运行该Servlet，在运行过程中如果需要接受从客户端传递过来的数据（如表单和URL传递的数据），则WEB容器会接受传入的数据，在接收过程中如果程序设定了传入参数的的编码则采用设定的编码格式，如果没有设置则采用默认的ISO-8859-1编码格式，接收的数据后JVM会将这些数据进行编码格式转换为Unicode并且存入到内存中。运行Servlet后产生输出结果，同时这些输出结果的编码格式仍然为Unicode。紧接着WEB容器会将产生的Unicode编码格式的字符串直接发送置客户端，如果程序指定了输出时的编码格式，则按照指定的编码格式输出到浏览器，否则采用默认的ISO-8859-1编码格式。整个过程流程图如下：

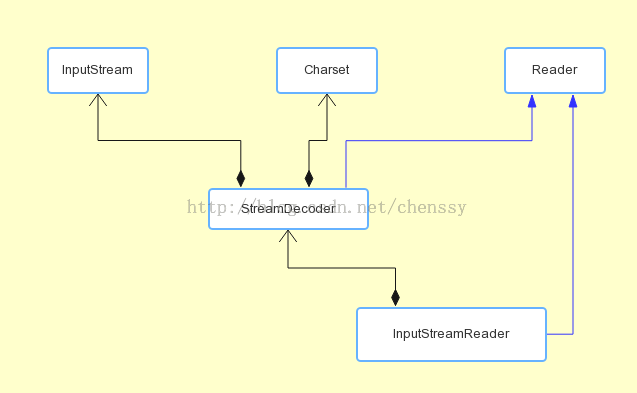


3.数据库部分  
 我们知道java程序与数据库的连接都是通过JDBC驱动程序来连接的，而JDBC驱动程序默认的是ISO-8859-1编码格式的，也就是说我们通过java程序向数据库传递数据时，JDBC首先会将Unicode编码格式的数据转换为ISO-8859-1的编码格式，然后在存储在数据库中，即在数据库保存数据时，默认格式为ISO-8859-1。



**3 编码&解码**  
下面将结束java在那些场合需要进行编码和解码操作，并详序中间的过程，进一步掌握java的编码和解码过程。在java中主要有四个场景需要进行编码解码操作：  
 (1)：I/O操作  
 (2)：内存  
 (3)：数据库  
 (4)：javaWeb  
 下面主要介绍前面两种场景，数据库部分只要设置正确编码格式就不会有什么问题，javaWeb场景过多需要了解URL、get、POST的编码，servlet的解码，所以javaWeb场景下节LZ介绍。

**（1）.I/O操作**  
 在前面LZ就提过乱码问题无非就是转码过程中编码格式的不统一产生的，比如编码时采用UTF-8，解码采用GBK，但最根本的原因是字符到字节或者字节到字符的转换出问题了，而这中情况的转换最主要的场景就是I/O操作的时候。当然I/O操作主要包括网络I/O（也就是javaWeb）和磁盘I/O。网络I/O下节介绍。首先我们先看I/O的编码操作。



InputStream为字节输入流的所有类的超类，Reader为读取字符流的抽象类。java读取文件的方式分为按字节流读取和按字符流读取，其中InputStream、Reader是这两种读取方式的超类。

**----按字节-----**  
我们一般都是使用InputStream.read()方法在数据流中读取字节（read()每次都只读取一个字节，效率非常慢，我们一般都是使用read(byte[])），然后保存在一个byte[]数组中，最后转换为String。在我们读取文件时，读取字节的编码取决于文件所使用的编码格式，而在转换为String过程中也会涉及到编码的问题，如果两者之间的编码格式不同可能会出现问题。例如存在一个问题test.txt编码格式为UTF-8，那么通过字节流读取文件时所获得的数据流编码格式就是UTF-8，而我们在转化成String过程中如果不指定编码格式，则默认使用系统编码格式（GBK）来解码操作，由于两者编码格式不一致，那么在构造String过程肯定会产生乱码，如下：

[?](https://www.jb51.net/article/79921.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | File file = new File("C:\\test.txt");  InputStream input = new FileInputStream(file);  StringBuffer buffer = new StringBuffer();  byte[] bytes = new byte[1024];  for(int n ; (n = input.read(bytes))!=-1 ; ){   buffer.append(new String(bytes,0,n));  }  System.out.println(buffer); |

输出结果为乱码....  
test.txt中的内容为：我是 cm。  
 要想不出现乱码，在构造String过程中指定编码格式，使得编码解码时两者编码格式保持一致即可：

|  |  |
| --- | --- |
| 1 | buffer.append(new String(bytes,0,n,"UTF-8")); |

**----按字符----**  
 其实字符流可以看做是一种包装流，它的底层还是采用字节流来读取字节，然后它使用指定的编码方式将读取字节解码为字符。在java中Reader是读取字符流的超类。所以从底层上来看按字节读取文件和按字符读取没什么区别。在读取的时候字符读取每次是读取留个字节，字节流每次读取一个字节。

**----字节&字符转换----**  
 字节转换为字符一定少不了InputStreamReader。API解释如下：InputStreamReader 是字节流通向字符流的桥梁：它使用指定的 charset 读取字节并将其解码为字符。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。 每次调用 InputStreamReader 中的一个 read() 方法都会导致从底层输入流读取一个或多个字节。要启用从字节到字符的有效转换，可以提前从底层流读取更多的字节，使其超过满足当前读取操作所需的字节。API解释非常清楚，InputStreamReader在底层读取文件时仍然采用字节读取，读取字节后它需要根据一个指定的编码格式来解析为字符，如果没有指定编码格式则采用系统默认编码格式。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | String file = "C:\\test.txt";     String charset = "UTF-8";     // 写字符换转成字节流     FileOutputStream outputStream = new FileOutputStream(file);     OutputStreamWriter writer = new OutputStreamWriter(outputStream, charset);     try {     writer.write("我是 cm");     } finally {     writer.close();     }       // 读取字节转换成字符     FileInputStream inputStream = new FileInputStream(file);     InputStreamReader reader = new InputStreamReader(     inputStream, charset);     StringBuffer buffer = new StringBuffer();     char[] buf = new char[64];     int count = 0;     try {     while ((count = reader.read(buf)) != -1) {      buffer.append(buf, 0, count);     }     } finally {     reader.close();     }     System.out.println(buffer); |

**2.内存**  
 首先我们看下面这段简单的代码

|  |  |
| --- | --- |
| 1  2  3  4 | String s = "我是 cm";  byte[] bytes = s.getBytes();  String s1 = new String(bytes,"GBK");  String s2 = new String(bytes); |

 在这段代码中我们看到了三处编码转换过程（一次编码，两次解码）。先看String.getTytes():

|  |  |
| --- | --- |
| 1  2  3 | public byte[] getBytes() {    return StringCoding.encode(value, 0, value.length);   } |

 内部调用StringCoding.encode()方法操作：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | static byte[] encode(char[] ca, int off, int len) {    String csn = Charset.defaultCharset().name();    try {     // use charset name encode() variant which provides caching.     return encode(csn, ca, off, len);    } catch (UnsupportedEncodingException x) {     warnUnsupportedCharset(csn);    }    try {     return encode("ISO-8859-1", ca, off, len);    } catch (UnsupportedEncodingException x) {     // If this code is hit during VM initialization, MessageUtils is     // the only way we will be able to get any kind of error message.     MessageUtils.err("ISO-8859-1 charset not available: "          + x.toString());     // If we can not find ISO-8859-1 (a required encoding) then things     // are seriously wrong with the installation.     System.exit(1);     return null;    }   } |

 encode(char[] paramArrayOfChar, int paramInt1, int paramInt2)方法首先调用系统的默认编码格式，如果没有指定编码格式则默认使用ISO-8859-1编码格式进行编码操作，进一步深入如下：

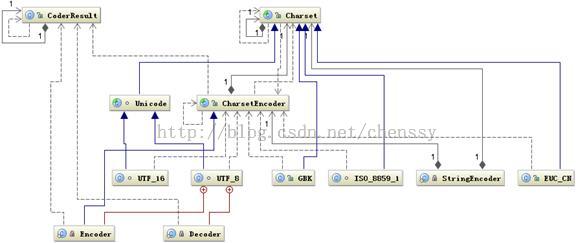
|  |  |
| --- | --- |
| 1 | String csn = (charsetName == null) ? "ISO-8859-1" : charsetName; |

 同样的方法可以看到new String 的构造函数内部是调用StringCoding.decode()方法：

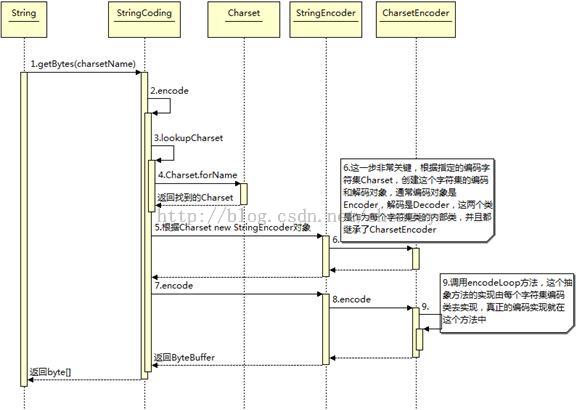
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public String(byte bytes[], int offset, int length, Charset charset) {    if (charset == null)     throw new NullPointerException("charset");    checkBounds(bytes, offset, length);    this.value = StringCoding.decode(charset, bytes, offset, length);   } |

 decode方法和encode对编码格式的处理是一样的。  
 对于以上两种情况我们只需要设置统一的编码格式一般都不会产生乱码问题。

**3 编码&编码格式**  
 首先先看看java编码类图



首先根据指定的chart设置ChartSet类，然后根据ChartSet创建ChartSetEncoder对象，最后再调用 CharsetEncoder.encode 对字符串进行编码，不同的编码类型都会对应到一个类中，实际的编码过程是在这些类中完成的。下面时序图展示详细的编码过程：



通过这编码的类图和时序图可以了解编码的详细过程。下面将通过一段简单的代码对ISO-8859-1、GBK、UTF-8编码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | public class Test02 {   public static void main(String[] args) throws UnsupportedEncodingException {    String string = "我是 cm";    Test02.printChart(string.toCharArray());    Test02.printChart(string.getBytes("ISO-8859-1"));    Test02.printChart(string.getBytes("GBK"));    Test02.printChart(string.getBytes("UTF-8"));   }     /\*\*    \* char转换为16进制    \*/   public static void printChart(char[] chars){    for(int i = 0 ; i < chars.length ; i++){     System.out.print(Integer.toHexString(chars[i]) + " ");    }    System.out.println("");   }     /\*\*    \* byte转换为16进制    \*/   public static void printChart(byte[] bytes){    for(int i = 0 ; i < bytes.length ; i++){     String hex = Integer.toHexString(bytes[i] & 0xFF);      if (hex.length() == 1) {      hex = '0' + hex;      }      System.out.print(hex.toUpperCase() + " ");    }    System.out.println("");   }  } |

输出：

|  |  |
| --- | --- |
| 1  2  3  4 | 6211 662f 20 63 6d  3F 3F 20 63 6D  CE D2 CA C7 20 63 6D  E6 88 91 E6 98 AF 20 63 6D |

通过程序我们可以看到“我是 cm”的结果为：

|  |  |
| --- | --- |
| 1  2  3  4 | char[]：6211 662f 20 63 6d  ISO-8859-1：3F 3F 20 63 6D  GBK：CE D2 CA C7 20 63 6D  UTF-8：E6 88 91 E6 98 AF 20 63 6D |

图如下

