# 聊聊gbk与utf8互转的乱码问题

作为一个程序员，乱码问题，应该我们都有遇到，但对于这个问题，很难用一句话概括乱码是怎么一回事，具体的问题还需要具体分析。

我们知道在计算机内存中，存储的是二进制数据，在网络传输中，也是二进制数据，但最终呈现给用户的是字符串，二进制与字符串的转化就需要编码、解码的参与，如果世界上只有一种字符编码方式，就不会有乱码这一说了，但事实是，编码的方式太多了，utf-8、utf-32、utf-16、gbk、gb2312、iso-8859-1、big5、unicode等等。由于每个编码的规则不一样，一般都不能用一种进行编码，用另一种进行解码。如**utf-8中，一个字母用一个字节表示，一个汉字用三个字节表示，特殊的汉字用四个字节表示，而gbk中，一个字母用一个字节表示，一个汉字用两个字节表示。**

有一个说法，内存中存储的二进制是unicode码，如果内存中的数据需要存储或传输时，才会进行一次转化，将unicode码转化成其它的编码二进制（有待考证）。个人觉得这种方式很合理，毕竟unicode码中每个字符都有独一无二的二进制与之对应。

排查乱码问题，难度在于是在哪个环节出了问题，但乱码的本质都是一样的，**读取二进制的编码和最初将字符串转化成二进制的编码方式不一致。**

此处说明一个概念，**编码指将字符串转化成二进制，解码指将二进制转化成字符串**。

## UTF-8编码，GBK解码

在这我们讨论一下，gbk和utf-8互转的乱码问题，直接上代码。

package com.anjz.test;

import java.io.UnsupportedEncodingException;

public class CodingTest {

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

String str = "你好，世界";

System.out.println("字符串长度:"+str.length());

byte[] utfBytes = str.getBytes("utf-8");

System.out.println("utf-8需要"+utfBytes.length+"字节存储");

byte[] gbkBytes = str.getBytes("gbk");

System.out.println("gbk需要"+gbkBytes.length+"字节存储");

}

}

以上代码运行打印出一下内容:

字符串长度:5

utf-8需要15字节存储

gbk需要10字节存储

可以看出，utf-8存储一个汉字，需要3个字节，gbk存储一个汉字，需要2个字节。

现用单个字符测试。

package com.anjz.test;

import java.io.UnsupportedEncodingException;

public class CodingTest {

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

String str = "你";

byte[] utfBytes = str.getBytes("utf-8");

for(byte utfByte:utfBytes){

//字节对应的十进制是负数，因二进制高位为1，如下进行补码操作，最后转化成16进制

System.out.print(Integer.toHexString((utfByte & 0xFF)) +",");

}

System.out.println();

String utf2gbkStr = new String(str.getBytes("utf-8"),"gbk");

System.out.println("utf-8转化成gbk:"+utf2gbkStr);

byte[] gbkBytes = utf2gbkStr.getBytes("gbk");

for(byte gbkByte:gbkBytes){

System.out.print(Integer.toHexString((gbkByte & 0xFF))+",");

}

System.out.println();

String gbk2utfStr = new String(utf2gbkStr.getBytes("gbk"),"utf-8");

System.out.println("gbk转化成utf-8:"+gbk2utfStr);

}

}

运行上面代码，得出的结果：

e4,bd,a0,

utf-8转化成gbk:浣�

e4,bd,3f,

gbk转化成utf-8:�?

用两个字符测试，将上述代码String str = “你”改成String str = “你好”。运行代码，得出的结果：

e4,bd,a0,e5,a5,bd,

utf-8转化成gbk:浣犲ソ

e4,bd,a0,e5,a5,bd,

gbk转化成utf-8:你好

上述实验中，utf-8转化成gbk出现乱码，这个很好理解，但是再还原回去，gbk转化成utf-8，单个中文字符依然是乱码，两个字符却能正常显示，这个到底是怎么回事呢？

经过一番研究，想把这个事说明白，还需要从它们的编码规则着手。

**ISO-8859-1**

单字节编码，向下兼容ASCII，其编码范围是0x00-0xFF，0x00-0x7F之间完全和ASCII一致，0x80-0x9F之间是控制字符，0xA0-0xFF之间是文字符号。

**GBK**

采用单双字节变长编码，英文使用单字节编码，完全兼容ASCII字符编码，中文部分采用双字节编码。双字节其编码范围从8140至FEFE（剔除xx7F）。

单字节：00000000 - 01111111

双字节：10000001 01000000 - 11111110 11111110 (剔除 \*\*\*\*\*\*\*\* 01111111)

单字节、双字节的区分通过高字节高位区分，单字节高位为0，双字节的高字节高位为1。

**UTF-8**

可变长字符编码，是unicode码的具体实现，UTF-8用1到6个字节编码Unicode字符。

**UTF-8编码规则：如果只有一个字节则其最高二进制位为0；如果是多字节，其第一个字节从最高位开始，连续的二进制位值为1的个数决定了其编码的字节数，其余各字节均以10开头。**

1字节 0xxxxxxx

2字节 110xxxxx 10xxxxxx

3字节 1110xxxx 10xxxxxx 10xxxxxx

4字节 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

5字节 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

6字节 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

明白上述GBK和UTF-8的编码规则，我们再分析一下，单个中文字符是乱码，两个字符却能正常显示的问题。

“你”

UTF-8编码对应的二进制：11100100 10111101 10100000

将上述二进制通过GBK进行解码，根据GBK规则，第一个字节高位为1，使用双字节编码，

“11100100 10111101”解码成“浣”,“10100000”对于GBK来说是非法的，就解码成了一种特殊字符“�”。

看看能不能将“浣�”还原回“你”呢？

GBK编码对应的二进制：11100100 10111101 00111111

看到上述的二进制，根本不符合UTF-8编码规则，故用UTF-8进行解码，是解码成了一些特殊字符“�?”。

对于上述情况可以看出，**一个二进制，如果不符合当前的编码规则，会被解码成特殊字符，但此特殊字符再进行编码，是回不到最初的二进制的**。

用同样的方式，分析“你好”为什么最终可以正常显示。

UTF-8编码对应的二进制：11100100 10111101 10100000 11100101 10100101 10111101

将上述二进制通过GBK进行编码，根据GBK规则，使用双字节编码，“1100100 10111101”解码成“浣”，“10100000 11100101”解码成“犲”，“10100101 10111101”解码成“ソ”。

看看能不能将“浣犲ソ”还原成“你好”呢？

GBK 编码对应的二进制：11100100 10111101 10100000 11100101 10100101 10111101

可以看出二进制是可以被还原的，将此二进制通过UTF-8解码，肯定能变成“你好”。

可以看出，一个字符串，通过UTF-8进行编码，再通过GBK进行解码，再将得到的字符串进行GBK编码，最后将得到的二进制通过UTF-8解码，能否还原到最初的字符串，在于UTF-8编码后得到的二进制，是否符合GBK的编码规则，如果符合，最终就可以还原，如果不符合，就不可还原。

## GBK编码，UTF-8解码

package com.anjz.test;

import java.io.UnsupportedEncodingException;

public class CodingTest {

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

String str = "你好";

byte[] gbkBytes = str.getBytes("gbk");

for(byte gbkByte:gbkBytes){

//字节对应的十进制是负数，因二进制高位为1，如下进行补码操作，最后转化成16进制

System.out.print(Integer.toHexString((gbkByte & 0xFF)) +",");

}

System.out.println();

String gbk2utfStr = new String(str.getBytes("gbk"),"utf-8");

System.out.println("gbk转化成utf-8:"+gbk2utfStr);

byte[] utfBytes = gbk2utfStr.getBytes("utf-8");

for(byte utfByte:utfBytes){

System.out.print(Integer.toHexString((utfByte & 0xFF))+",");

}

System.out.println();

String utf2gbkStr = new String(gbk2utfStr.getBytes("utf-8"),"gbk");

System.out.println("utf-8转化成gbk:"+utf2gbkStr);

}

}

运行上述代码，结果为：

c4,e3,ba,c3,

gbk转化成utf-8:���

ef,bf,bd,ef,bf,bd,ef,bf,bd,

utf-8转化成gbk:锟斤拷锟�

上述结果应该都在意料之中，我们通过上述的方法分析一下。

“你好”GBK编码的二进制：11000100 11100011 10111010 11000011

GBK编码的二进制数据，完全匹配不了UTF-8的编码规则，最终UTF-8只能按如下方式匹配，查看第一个字节，开头“110”，理论上匹配两个字节，但看下一个字节，开头却不是“10”，最终“11000100”解码成“�”,看第二个字节开头是“1110”，理论匹配三个字节，看下个字节符合，以“10”开头，但下下个字节开头是“110”，不符合匹配，最终“ 11100011 10111010”解码成“�”，同理“11000011”也解码成“�”，这个符号都是为找不到对应规则随意匹配的一个特殊字符。

“���”UTF-8编码的二进制为：11101111 10111111 10111101 11101111 10111111 10111101 11101111 10111111 10111101

这个二进制和原先的二进制不相同，根本转化不到最初的字符串，按照GBK的编码规则，“11101111 10111111”编码成“锟”,“10111101 11101111” 编码成“斤”，“10111111 10111101”编码成“拷”，“11101111 10111111”编码成“锟”，“10111101”不符合GBK规则，编码成特殊字符“�”。

理论上说，用GBK编码，UTF-8解码的字符串是不能还原到最初的字符串的，因UTF-8编码规则的特殊性，GBK编出的二进制，是很难匹配上的。

## 总结

理论上说，系统出现乱码，将乱码还原到最初的样子，上述UTF-8编码，GBK解码，这个有时是可以还原的，有时是还原不了的，要看UTF-8编码的二进制是否都能符合GBK的编码规则，但GBK编码，UTF-8解码，这个基本是条不归路。

**但实际中，有一种情况，是100%可以将乱码还原成最初的字符串。就是任意编码格式编码，ISO-8859-1解码，这个主要因为ISO-8859-1是单字节编码，而且匹配所有单字节情况，乱码字符串总是可以还原到最初的二进制。**

拓展一个小知识点：

关于进制的表示有两种方式，一种是前缀表示法，一种是后缀表示法。

**前缀表示法**

十六进制：0x

十进制：无前缀

八进制：0

二进制：没有表示符号

**后缀表示法**

B ：二进制数。

Q ：八进制数。

D ：十进制数。

H ：十六进制数。

对于十进制数通常不加后缀，也即十进制数后的字母 D 可省略。

参考文章

<http://blog.csdn.net/u010234516/article/details/52853214>

<http://www.cnblogs.com/yelongsan/p/6290206.html>

工具：

在线进制转换 http://tool.oschina.net/hexconvert/