<http://hippo-jessy.com/2017/01/17/Charset-and-Encoding/>

最近重新整理一遍 ****Java**** 基础，在写到注释相关的内容时发现自己对各种字符编码问题一直似懂非懂。相信很多人都听说过ASCII, Unicode, GB2312, UTF-8之类的编码，但是对于它们的印象可能仅仅停留在知道它们是不同的字符编码，使用不当可能会导致乱码的层面上。

查了很多资料，脑子有点乱，下面这篇笔记主要用来梳理思路。文章可能涉及到ASCII, ISO8859, GB2312, GBK, Unicode, UTF-8, UTF-16等多个字符集和编码方式进行探讨，篇幅有限，泛泛而谈，希望对有相同困惑的朋友有所帮助~~

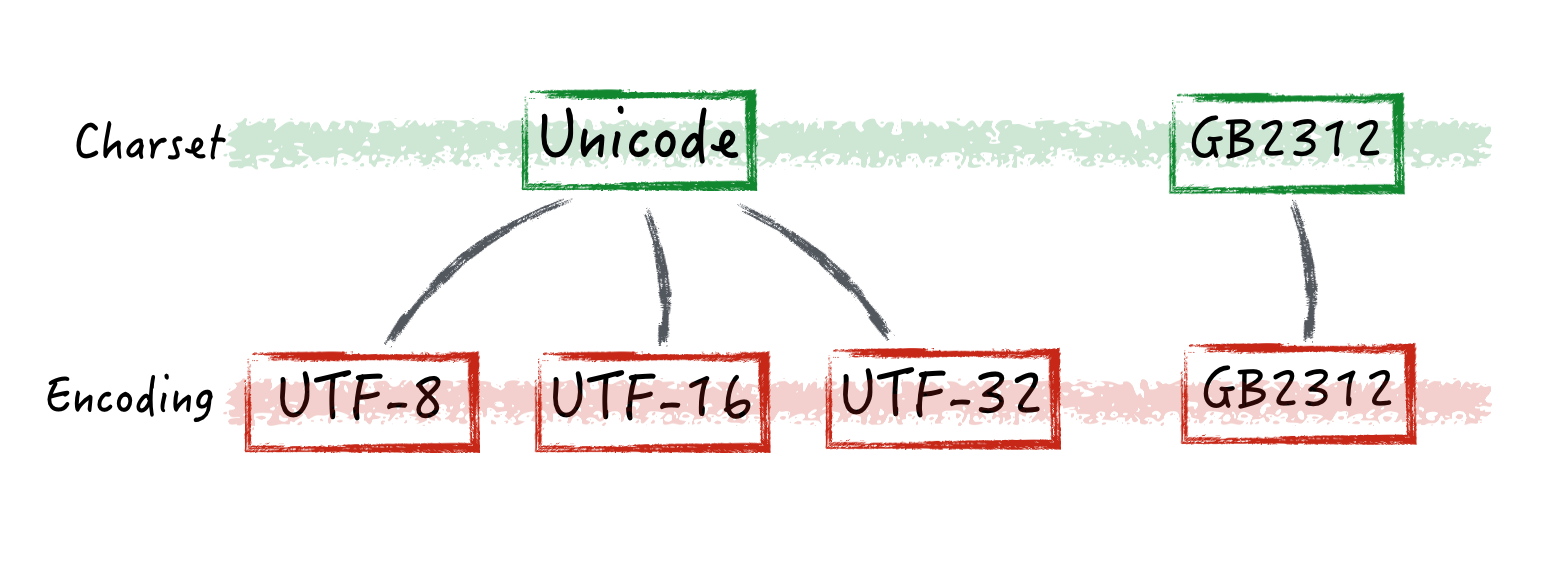
## **Charset and Encoding**

强调，这一部分****非常重要****，很多人在弄清楚Unicode, GB2312等多种编码原理之后可能都不一定弄清楚了字符集和字符编码的区别。在具体讲解各种编码规则之前，先来明确一下字符集(Charset)和字符编码(Encoding)的区别。

Charset (Character set) 字符集： 见名知义，字符集就是字符的集合。比如ASCII字符集收录了英文字符和标点以及控制字符，GB2312字符集收录了常用的汉字字符、标点等。

Encoding (Charset Encoding) 字符编码：实现字符与字节（二进制）映射的编码规则。可以通俗的看做是实现语言字符与计算机机器码相互翻译的规则，比如UTF-8编码方式等。

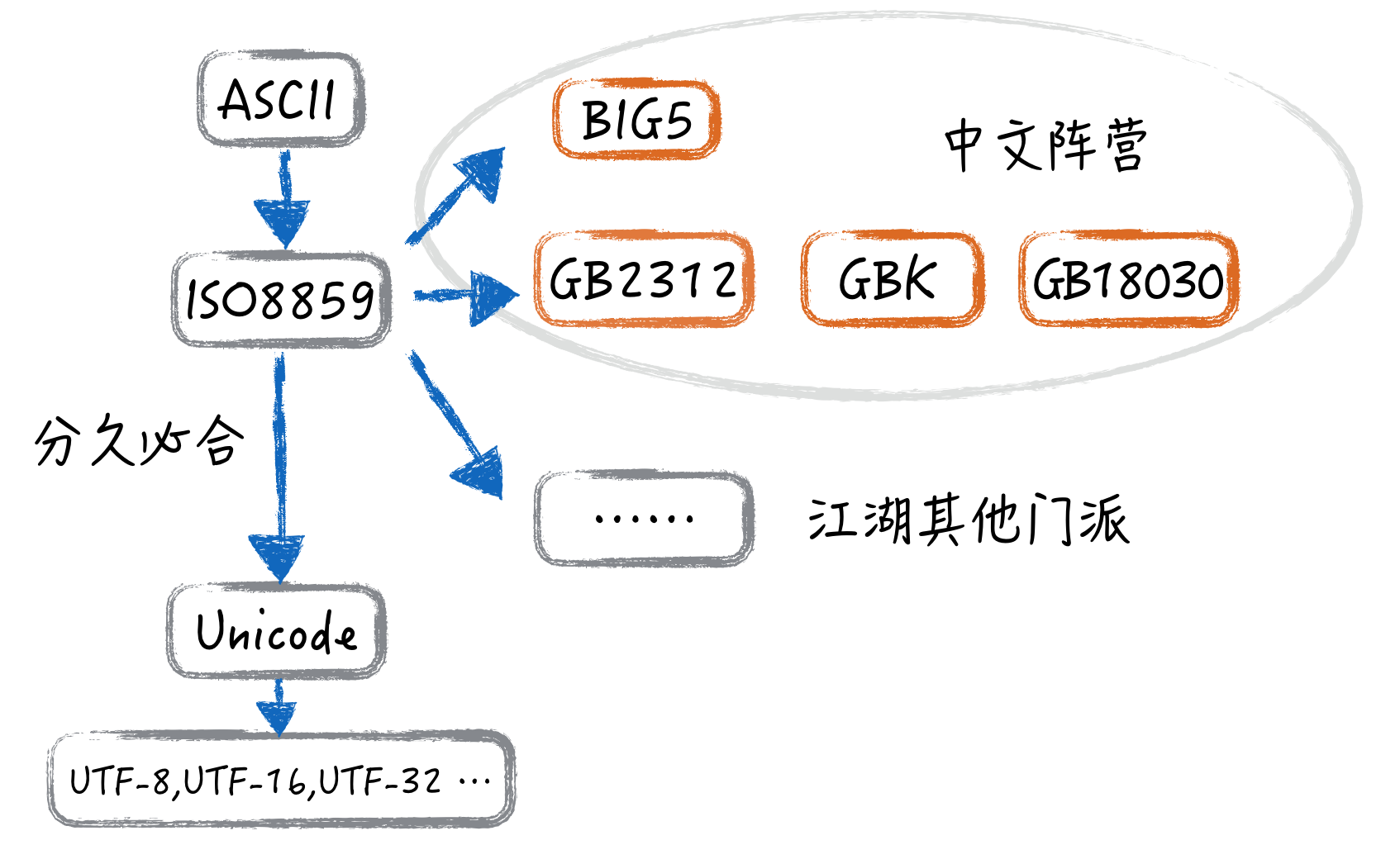
为什么经常容易混淆Charset与Encoding？主要由于很多Charset只对应一种Encoding的方式，并且有相同的名字，比如ASCII字符集只对应ASCII编码规则，GB2312字符集只能由GB2312编码规则来进行翻译。然而下面这个例子很好地诠释了Charset与Encoding的区别

[](http://ojnnon64z.bkt.clouddn.com/Charset and Encoding.png)

Unicode字符集对应了多种Encoding方式（UTF-8等），而GB2312字符集只有GB2312一种编码方式。因此实际上我们经常看到的 charset = "uft-8" 是不准确的说法，UTF-8实际上只是一种编码方式而不是一个字符集，它所对应的字符集是Unicode字符集。额外提一句，在windows平台上(比如记事本)有时候会出现Unicode编码这种不规范的说法，其实它指的的UTF-16编码。

## **从历史说起**

首先上图，这张图按照历史发展顺序对多种字符编码集进行了梳理。

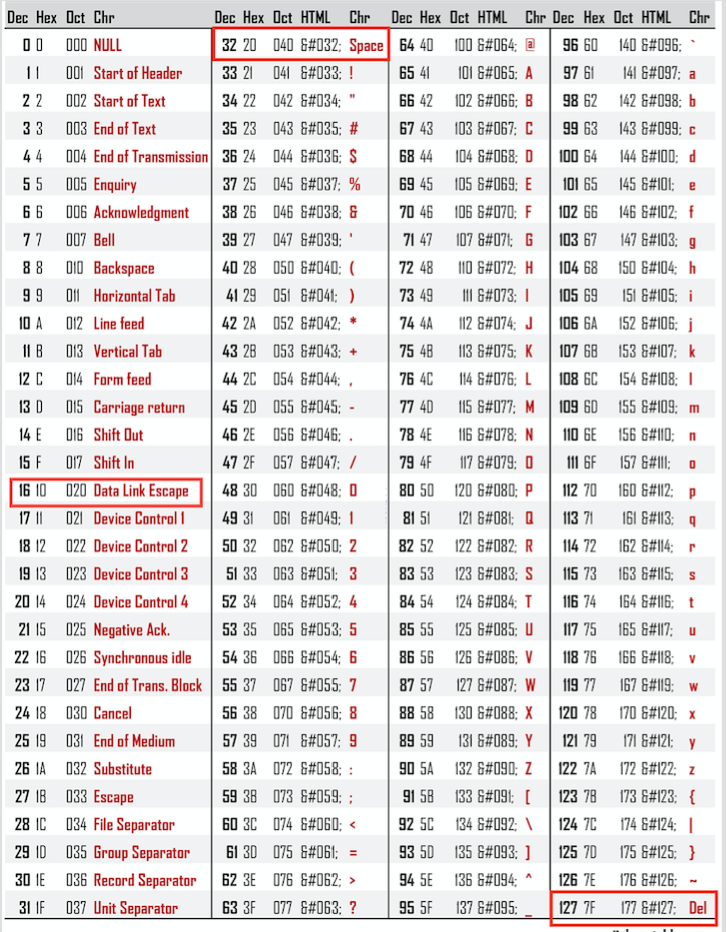
[](http://ojnnon64z.bkt.clouddn.com/Charset and Encoding_1.png)

简单的来说，最开始为了能实现二进制与人类语言字符的映射转换，人们想出了ASCII码来实现英文字符的转换。后来西欧各国也需要表示自己的语言，于是对ASCII进行扩展产生ISO8859。再然后我天朝崛起，博大精深的中文通过GB2312等字符集进入计算机，当然世界上还有各种语言，于是各种字符集编码层出不穷。。。为了促进世界的和谐，Unicode出现了希望能够包罗万象，以一种字符编码集收录所有的语言。再后来随着网络世界的崛起，为了提高网络传输效率，UTF-8等编码方式出现以期望提高Unicode的编码效率。

下面将大致根据上图的脉络，依次对各个字符编码展开讲解。

## **ASCII**

​ 计算机使用二进制保存指令和数据，字节作为计算接处理数据的基本单位，有256(28)种可能状态。这些状态被人们利用来标记指令和文字。最初前32(0x20)种状态被用来表示终端、打印机等设备的某些特殊动作，比如终端遇到字节0x10则换行，类似的还有回车(CR 0x0D)，震铃(BELL 0x07)等。这前32种状态又被称为控制码。除了这些特殊用途的状态外，还有256-32=224种状态没有被利用，资源浪费怎么能忍，何况除了控制指令外，咱们的文字还没有被表示。于是这些多出来的状态被用来表示文字(English Character)和符号标点等，如此这般，计算机就可以显示和记录文字了，这便是****ASCII****(American Standard Code for Information Interchange)最初的由来。下面附张ASCII的图加深印象。

[](http://ojnnon64z.bkt.clouddn.com/Charset and Encoding_2.png)

从图片中可以看到，所有英文字符和标点等可显示字符即使加上32种控制字符，也只需要128种状态(7 bits)就可以表示完，因此ASCII码实际上每个字节的第一位都是0。

## **ISO8859**

​ 然而，世界上不止只有英文这一种语言，西欧各个国家蠢蠢欲动。由于西欧的多种语言和英语有着共同的起源，都是拉丁语系，因此除了ASCII码原有的128个字符，借助一个字节剩下的256-128=128种状态也足以表示他们的语言字符不同的地方（比如130即0x82在法语编码中代表了é）。由此，引出了****ISO8859****这种编码方式。ISO8859实际上是对ASCII的8位扩展（ASCII实际上只用到了一个字节的低7位），是由****ISO****(the International Organization for Standardization )提出的。

## **GB2312**

​ 终于等到我大天朝了，区区256种状态怎够表示我朝上下五千年的灿烂文明！！！仅仅常用的汉字就多达6000多个，更别提那些生僻字和繁体字了。勤劳智慧的中国人民需要更多位来表示汉字，于是****GB2312****应运而生。GB2312采用两个字节来表示一个中文字符（即一个汉字），仍然使用一个字节表示英文字符标点等（保留ASCII的方式），一共收录了6763个常用汉字，其他一些特殊符号，以及ASCII表示的字符。这样一来每个字符对应的二进制位数可变（8位或16位），计算机要如何识别一个字节对应一个字符还是两个字节对应一个字符呢？

​ 我们知道ASCII实际只用到了字节的低七位，因此字节的第一位始终为0。GB2312使用第一位为1的两个连续字节表示汉字，使用第一位为0的单个字节表示原来ASCII编码的字符（由此实现了对ASCII的兼容）。这样，当计算机读到首位为1的字节时便可知需要将连续两个字节显示成一个字符，读到首位为0的字节时便直接将该字节翻译成一个字符。下面详述汉字表示的部分。

​ GB2312使用首位为1的两个字节来表示一个汉字字符(一个汉字或者一个中文标点等)，第一个字节称为高位字节，范围是0xA1-0xF7（87种状态），用来表示区码，分别对应区号01-87；第二个字节为低位字节，范围是0xA1-0xFE（94种状态），用来表示位码，分别对应位号01-94。这种表示方式也被称为“区位码”，其实说白了就是类似于坐标定位的原理，你可以想象这些汉字字符都被放在了一个二维平面上，区位码类似于横纵坐标，依此实现编码与汉字字符的一一对应。

​ 对于区码对应的字符有如下特征：

01-09区为特殊符号

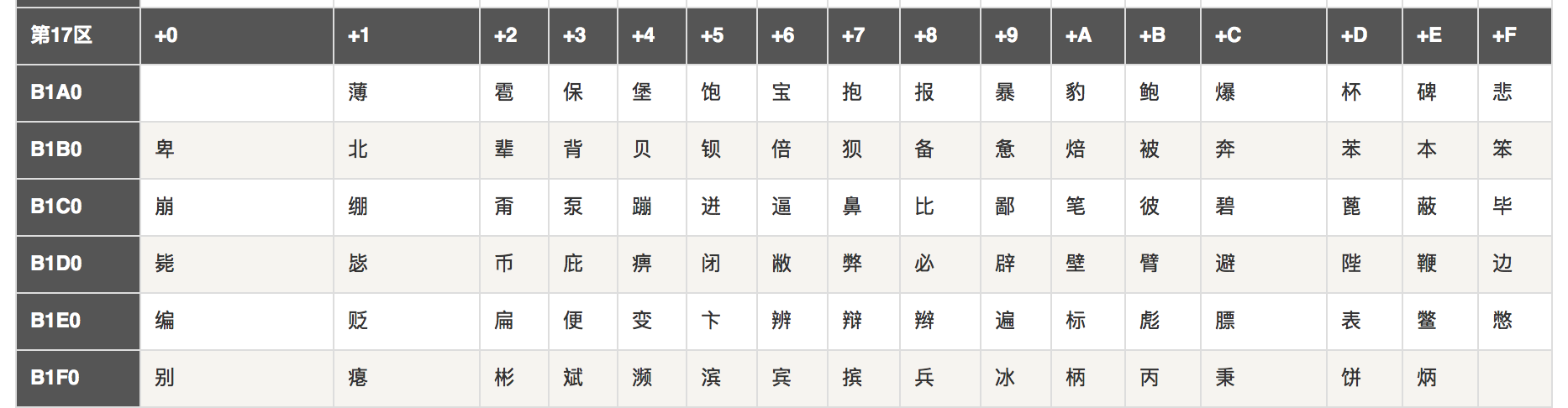
10-15区没有编码

16-55区为一级汉字，按拼音排序，共3755个

56-87区为二级汉字，按部首／笔画排序，共3008个

88-94区没有编码

由于编码表过大([地址一](http://tools.jb51.net/table/gb2312" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank) [地址二](http://www.knowsky.com/resource/gb2312tbl.htm" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank))，这里只截取第17区部分示例讲解。

[](http://ojnnon64z.bkt.clouddn.com/Charset and Encoding_3.png)

第17区对应于区码0xA1+17=0xB1，因此如上图所示该区所有汉字字符的高位字节均为0xB1。而且也可以看到，随着位码的增长，其对应的汉字字符也是按拼音顺序排序的。另外提一句，由于两个字节可以表示的状态很多而实际上GB2312只收录了六千多个常用的汉字。于是仗着空间大，尽管已经收录了ASCII中的字符，GB2312又做了一套类似ASCII中的一些标点符号收录进来，也就出现了我们经常遇到的全角半角标符的区别。

## **GBK & GB18030 & BIG5**

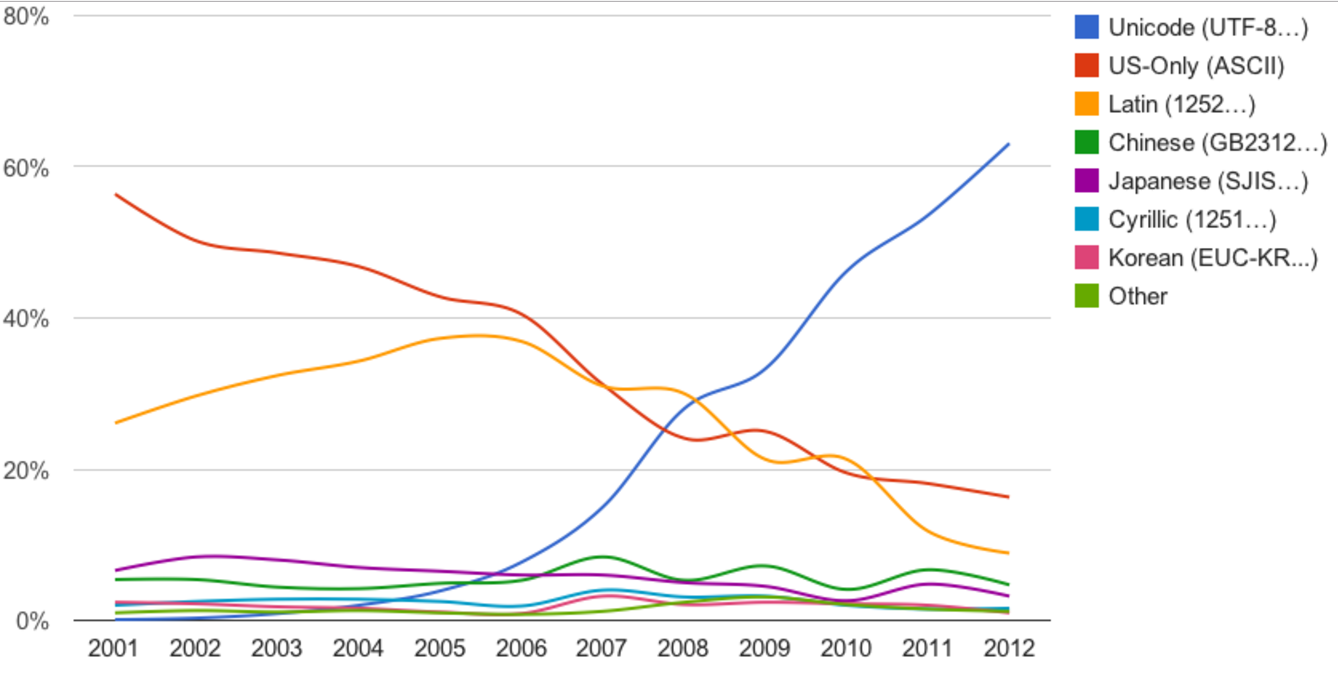
中国汉字总数将近十万个，GB2312也仅仅只能表示常用的汉字，但是对于生僻字以及繁体字，则需要****GBK****对GB2312进行进一步的扩展。还有，别忘了，五十六个民族五十六朵花。咱们少数民族同胞的各种语言瑰宝怎能蒙尘，于是你懂得，****GB18030****继续扩展（写到这里，突然好奇去翻了个[藏文网站](http://www.amdotibet.cn/" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank)，虽然如同天书一般，但是也莫名感觉到了一股浓郁的政府芬芳）。

​ BIG5又称****大五码****，台湾同胞的繁体编码集。。。实在没精力一一介绍了，感兴趣的童鞋自行搜索吧>\_<。

## **Unicode**

仅仅为了解决汉字编码问题就已经陆续出现了GB2312, BIG5等多种中文派系的编码方式，世界上有多种语言，针对各种语言门派的编码方式层出不穷，且互不兼容。为了解决这种诸侯纷争的局面，****ISO****决定一统江湖，推出Unicode这种****字符集****，希望这部武林秘籍能够保罗所有语言字符。

​ 下面 **[这张图](https://googleblog.blogspot.com/2012/02/unicode-over-60-percent-of-web.html" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank)** 足以说明Unicode的发展趋势和普及程度。

[](http://ojnnon64z.bkt.clouddn.com/Charset and Encoding_4.png)

​Unicode不仅仅是一个字符集，它还为每个字符配上了一个号码。值得注意的是，这个号码并不是字符在计算机的字节中存储的号码。可以将号码看做是对整个字符集进行了一个编号（1号字符，2号字符，etc），就好比是学生都有学号，方便管理。在Unicode中将这种编号命名为****码点(code point)****。这使得Unicode字符集称为一种标准，一旦公布了Unicode字符集以及对应码点，大家就可以自行设计编码方式，将字符码点按照自定义的编码规则映射为相应编码储存到字节中。这也侧面验证了前文谈到的字符集与字符编码的区别，Unicode字符集就对应了UTF-8, UTF-16等多种编码方式。Unicode的码点范围在U+0000~U+10FFFF(十六进制表示数字)，大致可以算出范围在111万左右。为了方便分类管理，这些码点又被分为多个平面（plane）来管理，我们只需要知道****BMP****(Basic Multilingual Plane)这个平面即可。BMP包含了U+0000~U+FFFF这个范围内的码点，世界人民常用的字符都落在这个平面内。BMP所包含的所有字符参见传送门☞[BMP](http://unifoundry.com/pub/unifont-7.0.03/unifont-7.0.03.bmp" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank)。

​ 这里再多说几句，以免混淆****码点****和真正存储在计算机字节中的****编码****。我们知道编码规则的作用是将字符翻译成机器中的二进制编码。其实也就是实现字符与编码一一映射的规则。码点只是Unicode字符集额外提供的一个工具，方便编码规则来实现这种映射。试想如果编码规则实现了学号与编码的一一映射，那么其实也就实现了学生与编码的翻译过程。对于码点也是同样的，只要实现了码点与编码的映射（这个映射由编码规则实现，如UTF-8），也就实现了字符与编码的翻译。那么码点和编码的数值大小是一样的吗？这个就得由编码规则来定了。

​ Unicode的最大码点0x10FFFF，至少需要三个字节来保存。如果要设计一种编码方式来保存码点，你会怎么做？最简单的做法就是直接将码点当做编码来用，用三个字节来保存一个字符编码。当然这是一种可行的方案，实际上UTF-32就是采用的类似思想，不过它是用四个字节来保存一个字符。然而这种方式有明显的缺点，对于较小的码点(比如U+0001,U+0002等)，根本不需要那么多字节来保存。而且随着网络的发展，很多文件需要通过网络传输，这种低效的编码方式使得文件大小过大，不利于网络传输。

​ 于是一种新的编码思想产生的——****变长编码****。我们之前谈到的编码方式大多采用定长编码，即使用固定的字节数目来表示每一个字符。实际上我们可以将定长与变长编码当做朗读课文时的一种断句方式。由于文件是以二进制保存的，计算机如何知道多少位表示一个字符？对于定长编码很简单，计算机只用按照固定的字节翻译成一个字符（类比朗读时两个字当做一个词来断句，“我们/大家/努力/学习/编码/知识”）；对于变长编码，有时一个字节翻译成一个字符，有时两三个字节翻译成一个字符（类比两个字或四个字当做一个词断句，“我们/大家/努力学习/编码知识”）。实际上之前提到的GB2312就是一种变长编码，它采用单字节存储ASCII字符集，采用双字节存储中文字符集部分。定长编码有其优势——规则简单且方便计算机“断句”，但是对于规模较大的字符集（比如百万级别的Unicode）,这种编码方式极容易造成空间浪费。UTF-8以及UTF-16这些变长编码规则的出现较好地解决了这个问题，它们采用较少的字节来保存数值较小的码点，使用较多字节来保存数值较大的码点。

## **UTF系列（UTF-8, UTF-16, etc）**

UTF(Unicode Transformation Format), Unicode字符集的编码方式，主要有UTF-8，UTF-16，UTF-32等。由于UTF-32采用定长四字节编码，这里不多说，下面主要介绍UTF-8，附带介绍UTF-16。

### **UTF-8**

UTF-8是变长编码，使用1~4个字节来表示一个字符，下图大致呈现了UTF-8对Unicode码点的变长转换原理。

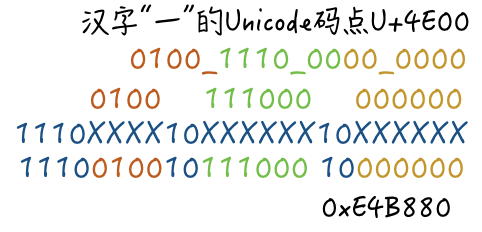
[](http://ojnnon64z.bkt.clouddn.com/Charset and Encoding_5.png)

单字节表示对应Unicode码点0x0000-0x007F的字符，且此时首位为0，有效位7位，也就是说可以表示27=128种字符，这部分其实刚好兼容了ASCII编码规则，而Unicode0x0000-0x007F的字符也就是ASCII字符。

双字节表示对应Unicode码点0x0080-0x07FF的字符，且此时第一个字节以110开头，后面的字节以10开头，有效 位数为11位，可以表示2048种字符。其实由上图可以看到UTF-8变长编码的规律，为了让计算机知道以几个字节来断句（即翻译为一个字符），在首字节的前几位就有几个1，例如如果是双字节断句，则首字节开头以110，如果是三字节断句，则首字节以1110开头。然而单字节断句为什么用0开头而不用1或者10开头呢？其实原因前文已经涉及了，就是为了兼容ASCII编码。这里还有几个问题大家可以思考一下：当多字节表示一个字符的情况时，既然首字节已经标记了如何断句，为什么后面的字节也要指定10开头？如果首字节后面的字节不指定用10开头岂不是可以增加两个有效位，增加编码空间进而提高效率？这些问题涉及到变长编码的一种设计思想，将在后文****定长编码与变长编码****部分详细解答。

三字节表示对应Unicode码点0x0800-0xFFFF的字符，有效位为16位，可以容纳65536种字符，我们常用的中文字符也都落在三字节部分。因此UTF-8是以三个字节来表示常用汉字字符的，当然实际上汉字有将近十万个，65536无法容纳如此庞大的汉字体系，因此有些冷门汉字只能用四字节表示了。我们知道Unicode将人们常用的字符都放在BMP平面（U+0000-U+FFFF），而至此，UTF-8已经用1-3个字节表示了BMP平面的全部内容。

以汉字“一”为例讲解UTF-8如何对Unicode码点进行编码。汉字“一”的Unicode码点是U+4E00，在范围0x0800-0xFFFF中，因此采用1110XXX\_10XXXXXX\_10XXXXXX模板，有效位为16位，0x4E00换做二进制表示为0b0100\_1110\_0000\_0000，将这十六位二进制数依次填入模板中缺失的X即可。于是我们得到汉字“一”的UTF-8编码为0b11100100\_10111000\_10000000，即0xE4B880。想要验证更多可以移步 ☞ **[传送门](http://www.endmemo.com/unicode/unicodeconverter.php" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank)**

[](http://ojnnon64z.bkt.clouddn.com/Charset and Encoding_6.png)

如果看到一串的16进制码有如下的形式：****E****X XX XX ****E****X XX XX…每个三字节组前面都是E打头，那么它很可能就是一串汉字的UTF-8编码了。

### **UTF-16**

UTF-16是以二字节或四字节表示一个字符的变长编码。Unicode在BMP平面的字符（U+0000~U+FFFF）用二字节来编码，其余字符用四字节来编码。对于二字节编码部分直接用码点值来编码，四字节编码部分的转换规则这里不再详述。我们只需记住常用的中英文字符在UTF-16中使用二字节表示即可。具体编码规则以后有空再加进来吧。

简单总结下：

UTF-8中，英文字符是一个字节表示，常用中文字符使用三个字节表示，某些中文字符可能采用四字节表示。

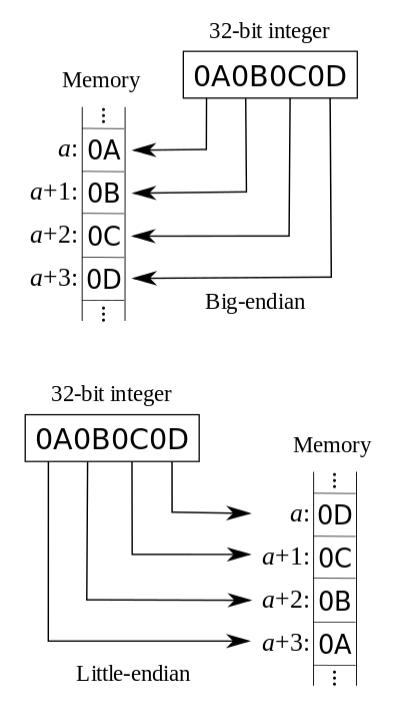
UTF-16中，英文字符是二字节表示，常用中文字符使用二字节表示，某些中文字符可能采用四字节表示。

大致了解了UTF系列的编码规则之后，可能还会有一个疑问：UTF-8，UTF-16，UTF-32这些名字中数字的由来。其实8，16，32指的是UTF用来表示一个字符最少需要用的位数，这个位数又被称为是****代码单元(Code Unit)****。UTF-8使用1~4个字节来表示字符，最少需要用8位来表示一个字符，代码单元为一个字节；UTF-16使用2或者4个字节来表示字符，最少需要16位来表示一个字符，代码单元为两个字节；UTF-32使用4个字节来表示字符，由于是定长编码，一定需要32位来表示一个字符，代码单元为四个字节。

### **BOM**

讲到BOM之前，需要大端Big Endian和小端Little Endian的相关知识, 这里只是简单提一下大小端的问题，具体细节请先自行google。

只有涉及到多于一个字节的数据结构保存和传输时才会产生大小端的问题。Big EndIan普遍用于网络传输，又被称为Network Byte Order；Little Endian通常用于微处理器，部分受到Intel处理器设计的影响。二者的区别简单来说就是，Big EndIan优先保存高位字节，最终表现为低址存储高位字节，高址存储低位字节； Little Endian优先保存低位字节， 最终表现为低址存储低位字节，高址存储高位字节。如下图所示（图片来自Wikipedia）：

[](http://ojnnon64z.bkt.clouddn.com/Charset and Encoding_7.png)

需要注意的是，不管大端还是小端，单个字节内部的存储顺序是一致的，因为字节是计算机数据处理的最小可寻址单位是字节，因此字节内部也就不存在高址低址的概念，故而更不会有大小端的概念了。

BOM, Byte Order Mark, 字节序标志。 刚才也提到过，只有对于多于一个字节的数据结构才存在字节序（Byte Order）这个概念，因此我们可以向字符编码中类推： UTF-8采用单个字节作为代码单元，因此没有并不会涉及字节序，而UTF-16使用两个字节作为一个代码单元，因此在代码单元的保存上用到了BOM（同理，UTF - 32也会用到BOM）。

UTF - 16 Big EndIan BOM： FE FF

UTF - 16 Little EndIan BOM： FF FE

实际上UTF-8早期也是有BOM的，其值为EF BB BF，但是通过之前的分析也可以知道这个BOM是没有意义的。更多的时候它并没有起到字节序标志的作用，而是用作标志此文件是UTF - 8编码，类似于魔数（Magic Number）的作用。

Java中默认使用Big EndIan，下面这段代码简单演示了“拜拜bye”字符串分别使用UTF-16大小端编码时的数据：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Demo {  public static void main(String[] args) {  String sample = "拜拜bye";  try {  //Java UTF-16默认使用大端  System.out.println(printHexBinary(sample.getBytes("UTF-16")));  System.out.println(printHexBinary(sample.getBytes("UTF-16LE")));  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  }  }  } |

输出的结果如下：

|  |  |
| --- | --- |
| 1  2 | FEFF62DC62DC006200790065  DC62DC62620079006500 |

在这里我们不仅看到了大端时的BOM是FEFF，并且还可以回忆验证一下UTF-16的编码规则：所有英文和大部分汉字使用两个字节来表示一个字符，因此这里采用十个字节来编码sample字符串。每个代码单元两个字节，字节顺序大小端刚好颠倒。

## **定长编码与变长编码**

前文****UTF-8****部分提出了的几个问题：当多字节表示一个字符的情况时，既然首字节已经标记了如何断句，为什么后面的字节也要指定10开头？如果首字节后面的字节不指定用10开头岂不是可以增加两个有效位，增加编码空间使得相同的字节长度可以表示更多的字符？

​ 首字节中110，1110，11110这些标记确实已经告诉计算机该如何“断句”，但是除了断句以外，计算机还需要完成我们的另一项需求：字符搜索。字符搜索实际上就是进行比特串匹配。假如UTF-8首字节后面的字节不指定用10开头很可能出现下面这种情况：

​ 我们想要搜索"a"，其UTF-8编码为0x61(0b01100001)。当我们搜索“a”时，计算机会自动搜索匹配01100001字符串，当然“a”会被搜索匹配出来，但是110XXXXX\_01100001，1110XXXX\_01100001\_XXXXXXXX这些二三字节表示的字符也会被错误匹配为“a”。因此在变长编码设计之初就需要考虑到搜索匹配的问题，多字节模式不可以与单字节模式或者较少字节模式的比特串发生字节重叠。对于UTF-8中首字节之后的字节的开始两位“10”就是起这个作用（详见下图）。细心的童鞋可能留意到GB2312在用二字节表示中文字符时也规定两个字节的首位都必须为1，其中第二个字节首位必须为1的规定也是为了避免和ASCII编码比特串（GB2312用一字节表示英文字符的方式和ASCII兼容）重叠。

[](http://ojnnon64z.bkt.clouddn.com/Charset and Encoding_8.png)

## **乱码原因**

编码方式不兼容会导致乱码。具体是怎么一回事呢？当你想打开一个保存有文字的文件时，首先这个文件肯定是事先经过字符输入的，那么在输入字符的过程中必须采用一种编码方式A来将字符转化为字节保存在存贮载体上。然而你通过某种渠道（网络传输，拷贝等）获得了这份文件，计算机打开这个文件时必须选择一种编码方式B来将字节翻译成字符以供阅读。如果编码方式B和编码方式A是同一种编码方式，或者编码方式B可以兼容编码方式A，我们就可以顺利地查看文件中保存的文字，否则就显示乱码。

你可能碰到过这样的事，把一个文本文件从Windows平台上传到Linux平台，并在Linux平台下打开时发现乱码了，但这不意味着文件内容有了什么变化，通常的原因是你的文件是用GBK编码的，但Linux平台下打开时它缺省可能用的是UTF-8编码去读取，因此，你只要调整成正确的编码去读取即可。

对于Web系统的情况，如果想要html中正常显示文字字符，就必须保证数据库编码，web应用程序编码以及html页面的编码都相同或者兼容。

## **Java采用的编码集**

Java内部字符串采用Unicode字符集，~~本想在这篇笔记中一起介绍Java涉及的编码方式以及一些相关实验验证，但是考虑到篇幅有限，还是以后有空另开一篇博客专门介绍。~~

更新于2017/03/09

避免立下太多flag，决定不再单独开博客介绍Java中涉及的编码方式，记住“字符传输存储用UTF-8，字符内存操作用UTF-16”这句话基本可以应对大多数场景需求了。可是，为什么选择UTF-8用作传输存储，而UTF-16用作内存操作？

其实也很好理解，如今网络带宽还比较有限，因此尽量选择效率高，占用空间小的编码方式来编码需要网络传输的数据。而对于内存操作而言，变长编码可能会导致算法复杂度以及操作复杂度的增加，虽然空间也很重要（即内存资源），但是对于处理逻辑（即CPU资源以及算法设计复杂度）而言，显得没那么重要。UTF-8采用1-4字节变长编码，大部分常用字符用一个字节就可以表示，相比UTF-16最少采用两个字节表示字符而言，UTF-8在节省空间上表现出了明显的优势，因此被广泛用于网络传输。而UTF-16采用两个或四个字节表示一个字符，虽然也是变长编码，但是除去生僻的字符外，大多数常用字符都只需要两个字节表示，因此比较方便设计字符的存储逻辑以及操作逻辑（比如Java中的基础数据类型char采用两个字节存储，而String底层也是简单采用char数组实现，都是利用了UTF-16大多数字符采用两个字节存储的特性）。

再就是，我们应该弄明白Java中String类的构造器new String(byte[] data) 和String类的实例方法public byte[] getBytes()是如何实现字符编码的转换的。

String类的内部实际采用private final char[] value这个字符数组来保存字符串，而new String(byte[] data)构造器内部会调用****默认的编码方式X****来解析字节数组data，然后将其转换为UTF-16编码存储到String内部的字符数组value中。类似的，public byte[] getBytes() 方法会用UTF-16解析String内部的value字符数组，然后转用****默认的编码方式X****进行编码后保存数据到方法的返回值byte数组中。当然如果你调用的是上述方法的重载方法：

public String(byte bytes[], Charset charset)

public byte[] getBytes(Charset charset)

那么，恭喜你，你可以自由选择****编码方式X****。

可是这里仍然有个疑问，Java默认采用UTF-16来进行字符内存操作，那这是不是意味着上一段提到的****默认的编码方式X**** 就是指UTF-16？

[https://www.zhihu.com/question/27562173](https://www.zhihu.com/question/27562173" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank)

然而事情并不是那么简单，这个疑问实际上是由于不明白“字符内存操作”的定义导致的。首先应该明确的是，对于Java而言，什么是字符内存操作？简而言之，字符内存操作即所有和char这种基本数据类型打交道的操作，就是字符内存操作。而Java默认采用UTF-16来进行字符内存操作的含义在于，Java必须保证char中保存的数据是采用UTF-16编码的。

感兴趣的童鞋可以移步下面这几篇专门讨论Java编码原理的博客：

[文本在内存中的编码(1)](https://my.oschina.net/goldenshaw/blog/470946" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank)

[文本在内存中的编码(2)](https://my.oschina.net/goldenshaw/blog/470948" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank)

[文本在内存中的编码(3)](https://my.oschina.net/goldenshaw/blog/471370" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank)

如果对Java中的字符操作比较感兴趣也可以看看下面这些问题的讨论：

[why does InputStream#read() return an int and not a byte](http://stackoverflow.com/questions/4659659/why-does-inputstreamread-return-an-int-and-not-a-byte" \t "file://D:\\WorkSpace\\IntelliJ\\Document\\Computer_Science\\charset_encoding\\_blank)

磨磨唧唧，终于写完了，估计你们早就看累了。还是那句老话，文章如有纰误，欢迎大家在评论区批评斧正，提前谢过~~

<http://hippo-jessy.com/2017/01/17/Charset-and-Encoding/>