* [博客园](http://www.cnblogs.com/)
* [闪存](http://home.cnblogs.com/ing/)
* [首页](http://www.cnblogs.com/kingstarspe/)
* [新随笔](http://i.cnblogs.com/EditPosts.aspx?opt=1)
* [联系](http://space.cnblogs.com/msg/send/Robin+Huang)
* [管理](http://i.cnblogs.com/)
* [订阅](http://www.cnblogs.com/kingstarspe/rss) [订阅](http://www.cnblogs.com/kingstarspe/rss)

随笔- 14  文章- 0  评论- 21

[**字符编码中ASCII、Unicode和UTF-8的区别**](http://www.cnblogs.com/kingstarspe/p/ASCII.html)

**1. ASCII码**

我们知道，在计算机内部，所有的信息最终都表示为一个二进制的字符串。每一个二进制位（bit）有0和1两种状态，因此八个二进制位就可以组合出 256种状态，这被称为一个字节（byte）。也就是说，一个字节一共可以用来表示256种不同的状态，每一个状态对应一个符号，就是256个符号，从 00000000到11111111。

上个世纪60年代，美国制定了一套字符编码，对英语字符与二进制位之间的关系，做了统一规定。这被称为ASCII码，一直沿用至今。

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。

**2、非ASCII编码**

英语用128个符号编码就够了，但是用来表示其他语言，128个符号是不够的。比如，在法语中，字母上方有注音符号，它就无法用ASCII码表示。 于是，一些欧洲国家就决定，利用字节中闲置的最高位编入新的符号。比如，法语中的é的编码为130（二进制10000010）。这样一来，这些欧洲国家使 用的编码体系，可以表示最多256个符号。

但是，这里又出现了新的问题。不同的国家有不同的字母，因此，哪怕它们都使用256个符号的编码方式，代表的字母却不一样。比如，130在法语编码 中代表了é，在希伯来语编码中却代表了字母Gimel (ג)，在俄语编码中又会代表另一个符号。但是不管怎样，所有这些编码方式中，0--127表示的符号是一样的，不一样的只是128--255的这一段。

至于亚洲国家的文字，使用的符号就更多了，汉字就多达10万左右。一个字节只能表示256种符号，肯定是不够的，就必须使用多个字节表达一个符号。 比如，简体中文常见的编码方式是GB2312，使用两个字节表示一个汉字，所以理论上最多可以表示256x256=65536个符号。

中文编码的问题需要专文讨论，这篇笔记不涉及。这里只指出，虽然都是用多个字节表示一个符号，但是GB类的汉字编码与后文的Unicode和UTF-8是毫无关系的。

**3.Unicode**

世界上存在着多种编码方式，同一个二进制数字可以被解释成不同的符号。因此，要想打开一个文本文件，就必须知道它的编码方式，否则用错误的编码方式解读，就会出现乱码。为什么电子邮件常常出现乱码？就是因为发信人和收信人使用的编码方式不一样。

可以想象，如果有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。这就是Unicode，就像它的名字都表示的，这是一种所有符号的编码。

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"。具体的符号对应表，可以查询[unicode.org](http://www.unicode.org/)，或者专门的[汉字对应表](http://www.chi2ko.com/tool/CJK.htm)。

**4. Unicode的问题**

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。

比如，汉字"严"的unicode是十六进制数4E25，转换成二进制数足足有15位（100111000100101），也就是说这个符号的表示至少需要2个字节。表示其他更大的符号，可能需要3个字节或者4个字节，甚至更多。

这里就有两个严重的问题，第一个问题是，如何才能区别Unicode和ASCII？计算机怎么知道三个字节表示一个符号，而不是分别表示三个符号 呢？第二个问题是，我们已经知道，英文字母只用一个字节表示就够了，如果Unicode统一规定，每个符号用三个或四个字节表示，那么每个英文字母前都必 然有二到三个字节是0，这对于存储来说是极大的浪费，文本文件的大小会因此大出二三倍，这是无法接受的。

它们造成的结果是：1）出现了Unicode的多种存储方式，也就是说有许多种不同的二进制格式，可以用来表示Unicode。2）Unicode在很长一段时间内无法推广，直到互联网的出现。

**5.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（字符用两个字节或四个字节表示）和UTF-32（字符用四个字节表示），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。

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。

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

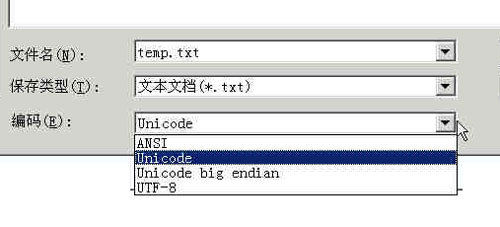
下面，还是以汉字"严"为例，演示如何实现UTF-8编码。

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。

**6. Unicode与UTF-8之间的转换**

"严"的Unicode码是4E25，UTF-8编码是E4B8A5，两者是不一样的。它们之间的转换可以通过程序实现。

在Windows平台下，有一个最简单的转化方法，就是使用内置的记事本小程序Notepad.exe。打开文件后，点击"文件"菜单中的"另存为"命令，会跳出一个对话框，在最底部有一个"编码"的下拉条。

[](http://www.ruanyifeng.com/blog/2007/10/bg2007102801.jpg)

里面有四个选项：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。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。我在下一节会解释little endian和big endian的涵义。

4）UTF-8编码，也就是上一节谈到的编码方法。

选择完"编码方式"后，点击"保存"按钮，文件的编码方式就立刻转换好了。

**7. Little endian和Big endian**

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。

这两个古怪的名称来自英国作家斯威夫特的《格列佛游记》。在该书中，小人国里爆发了内战，战争起因是人们争论，吃鸡蛋时究竟是从大头(Big- Endian)敲开还是从小头(Little-Endian)敲开。为了这件事情，前后爆发了六次战争，一个皇帝送了命，另一个皇帝丢了王位。

因此，第一个字节在前，就是"大头方式"（Big endian），第二个字节在前就是"小头方式"（Little endian）。

那么很自然的，就会出现一个问题：计算机怎么知道某一个文件到底采用哪一种方式编码？

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FEFF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。

**8. 实例**

下面，举一个实例。

打开"记事本"程序Notepad.exe，新建一个文本文件，内容就是一个"严"字，依次采用ANSI，Unicode，Unicode big endian 和 UTF-8编码方式保存。

然后，用文本编辑软件[UltraEdit中](http://www.google.cn/search?aq=t&oq=UltraEdit&complete=1&hl=zh-CN&newwindow=1&rlz=1B3GGGL_zh-CNCN216CN216&q=ultraedit+%E4%B8%8B%E8%BD%BD&btnG=Google+%E6%90%9C%E7%B4%A2&meta=)的"十六进制功能"，观察该文件的内部编码方式。

1）ANSI：文件的编码就是两个字节"D1 CF"，这正是"严"的GB2312编码，这也暗示GB2312是采用大头方式存储的。

2）Unicode：编码是四个字节"FF FE 25 4E"，其中"FF FE"表明是小头方式存储，真正的编码是4E25。

3）Unicode big endian：编码是四个字节"FE FF 4E 25"，其中"FE FF"表明是大头方式存储。

4）UTF-8：编码是六个字节"EF BB BF E4 B8 A5"，前三个字节"EF BB BF"表示这是UTF-8编码，后三个"E4B8A5"就是"严"的具体编码，它的存储顺序与编码顺序是一致的

# [Python字符编码详解](http://www.cnblogs.com/huxi/archive/2010/12/05/1897271.html)

本文简单介绍了各种常用的字符编码的特点，并介绍了在python2.x中如何与编码问题作战 ：）   
请注意本文关于Python的内容仅适用于2.x，3.x中str和unicode有翻天覆地的变化，请查阅其他相关文档。   
尊重作者的劳动，转载请注明作者及原文地址 >.<

## 1. 字符编码简介

### 1.1. ASCII

ASCII(American Standard Code for Information Interchange)，是一种单字节的编码。计算机世界里一开始只有英文，而单字节可以表示256个不同的字符，可以表示所有的英文字符和许多的控制 符号。不过ASCII只用到了其中的一半（\x80以下），这也是MBCS得以实现的基础。

### 1.2. MBCS

然而计算机世界里很快就有了其他语言，单字节的ASCII已无法满足需求。后来每个语言就制定了一套自己的编码，由于单字节能表示的字符太少，而且同时也需要与ASCII编码保持兼容，所以这些编码纷纷使用了多字节来表示字符，如GBxxx、BIGxxx等等，他们的规则是，如果第一个字节是\x80以下，则仍然表示ASCII字符；而如果是\x80以上，则跟下一个字节一起（共两个字节）表示一个字符，然后跳过下一个字节，继续往下判断。

这里，IBM发明了一个叫Code Page的概念，将这些编码都收入囊中并分配页码，GBK是第936页，也就是CP936。所以，也可以使用CP936表示GBK。

MBCS(Multi-Byte Character Set)是这些编码的统称。目前为止大家都是用了双字节，所以有时候也叫做DBCS(Double-Byte Character Set)。必须明确的是，MBCS并不是某一种特定的编码，Windows里根据你设定的区域不同，MBCS指代不同的编码，而Linux里无法使用 MBCS作为编码。在Windows中你看不到MBCS这几个字符，因为微软为了更加洋气，使用了ANSI来吓唬人，记事本的另存为对话框里编码ANSI就是MBCS。同时，在简体中文Windows默认的区域设定里，指代GBK。

### 1.3. Unicode

后来，有人开始觉得太多编码导致世界变得过于复杂了，让人脑袋疼，于是大家坐在一起拍脑袋想出来一个方法：所有语言的字符都用同一种字符集来表示，这就是Unicode。

最初的Unicode标准UCS-2使用两个字节表示一个字符，所以你常常可以听到Unicode使用两个字节表示一个字符的说法。但过了不久有人觉得256\*256太少了，还是不够用，于是出现了UCS-4标准，它使用4个字节表示一个字符，不过我们用的最多的仍然是UCS-2。

UCS(Unicode Character Set)还仅仅是字符对应码位的一张表而已，比如"汉"这个字的码位是6C49。字符具体如何传输和储存则是由UTF(UCS Transformation Format)来负责。

一开始这事很简单，直接使用UCS的码位来保存，这就是UTF-16，比如，"汉"直接使用\x6C\x49保存(UTF-16-BE)，或是倒过来使用\x49\x6C保存(UTF-16-LE)。但用着用着美国人觉得自己吃了大亏，以前英文字母只需要一个字节就能保存了，现在大锅饭一吃变成了两个字节，空间消耗大了一倍……于是UTF-8横空出世。

UTF-8是一种很别扭的编码，具体表现在他是变长的，并且兼容ASCII，ASCII字符使用1字节表示。然而这里省了的必定是从别的地方抠出来 的，你肯定也听说过UTF-8里中文字符使用3个字节来保存吧？4个字节保存的字符更是在泪奔……（具体UCS-2是怎么变成UTF-8的请自行搜索）

另外值得一提的是BOM(Byte Order Mark)。我们在储存文件时，文件使用的编码并没有保存，打开时则需要我们记住原先保存时使用的编码并使用这个编码打开，这样一来就产生了许多麻烦。 （你可能想说记事本打开文件时并没有让选编码？不妨先打开记事本再使用文件 -> 打开看看）而UTF则引入了BOM来表示自身编码，如果一开始读入的几个字节是其中之一，则代表接下来要读取的文字使用的编码是相应的编码：

BOM\_UTF8 '\xef\xbb\xbf'   
BOM\_UTF16\_LE '\xff\xfe'   
BOM\_UTF16\_BE '\xfe\xff'

并不是所有的编辑器都会写入BOM，但即使没有BOM，Unicode还是可以读取的，只是像MBCS的编码一样，需要另行指定具体的编码，否则解码将会失败。

你可能听说过UTF-8不需要BOM，这种说法是不对的，只是绝大多数编辑器在没有BOM时都是以UTF-8作为默认编码读取。即使是保存时默认使 用ANSI(MBCS)的记事本，在读取文件时也是先使用UTF-8测试编码，如果可以成功解码，则使用UTF-8解码。记事本这个别扭的做法造成了一个 BUG：如果你新建文本文件并输入"姹塧"然后使用ANSI(MBCS)保存，再打开就会变成"汉a"，你不妨试试 ：）

## 2. Python2.x中的编码问题

### 2.1. str和unicode

str和unicode都是basestring的子类。严格意义上说，str其实是字节串，它是unicode经过编码后的字节组成的序列。对 UTF-8编码的str'汉'使用len()函数时，结果是3，因为实际上，UTF-8编码的'汉' == '\xE6\xB1\x89'。

unicode才是真正意义上的字符串，对字节串str使用正确的字符编码进行解码后获得，并且len(u'汉') == 1。

再来看看encode()和decode()两个basestring的实例方法，理解了str和unicode的区别后，这两个方法就不会再混淆了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # coding: UTF-8    u = u'汉'  print repr(u) # u'\u6c49'  s = u.encode('UTF-8')  print repr(s) # '\xe6\xb1\x89'  u2 = s.decode('UTF-8')  print repr(u2) # u'\u6c49'    # 对unicode进行解码是错误的  # s2 = u.decode('UTF-8')  # 同样，对str进行编码也是错误的  # u2 = s.encode('UTF-8') |

需要注意的是，虽然对str调用encode()方法是错误的，但实际上Python不会抛出异常，而是返回另外一个相同内容但不同id的str； 对unicode调用decode()方法也是这样。很不理解为什么不把encode()和decode()分别放在unicode和str中而是都放在 basestring中，但既然已经这样了，我们就小心避免犯错吧。

### 2.2. 字符编码声明

源代码文件中，如果有用到非ASCII字符，则需要在文件头部进行字符编码的声明，如下：

|  |  |
| --- | --- |
| 1 | #-\*- coding: UTF-8 -\*- |

实际上Python只检查#、coding和编码字符串，其他的字符都是为了美观加上的。另外，Python中可用的字符编码有很多，并且还有许多别名，还不区分大小写，比如UTF-8可以写成u8。参见<http://docs.python.org/library/codecs.html#standard-encodings>。

另外需要注意的是声明的编码必须与文件实际保存时用的编码一致，否则很大几率会出现代码解析异常。现在的IDE一般会自动处理这种情况，改变声明后同时换成声明的编码保存，但文本编辑器控们需要小心 ：）

### 2.3. 读写文件

内置的open()方法打开文件时，read()读取的是str，读取后需要使用正确的编码格式进行decode()。write()写入时，如果 参数是unicode，则需要使用你希望写入的编码进行encode()，如果是其他编码格式的str，则需要先用该str的编码进行decode()， 转成unicode后再使用写入的编码进行encode()。如果直接将unicode作为参数传入write()方法，Python将先使用源代码文件 声明的字符编码进行编码然后写入。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # coding: UTF-8    f = open('test.txt')  s = f.read()  f.close()  print type(s) # <type 'str'>  # 已知是GBK编码，解码成unicode  u = s.decode('GBK')    f = open('test.txt', 'w')  # 编码成UTF-8编码的str  s = u.encode('UTF-8')  f.write(s)  f.close() |

另外，模块codecs提供了一个open()方法，可以指定一个编码打开文件，使用这个方法打开的文件读取返回的将是unicode。写入时，如 果参数是unicode，则使用open()时指定的编码进行编码后写入；如果是str，则先根据源代码文件声明的字符编码，解码成unicode后再进 行前述操作。相对内置的open()来说，这个方法比较不容易在编码上出现问题。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | # coding: GBK    import codecs    f = codecs.open('test.txt', encoding='UTF-8')  u = f.read()  f.close()  print type(u) # <type 'unicode'>    f = codecs.open('test.txt', 'a', encoding='UTF-8')  # 写入unicode  f.write(u)    # 写入str，自动进行解码编码操作  # GBK编码的str  s = '汉'  print repr(s) # '\xba\xba'  # 这里会先将GBK编码的str解码为unicode再编码为UTF-8写入  f.write(s)  f.close() |

### 2.4. 与编码相关的方法

sys/locale模块中提供了一些获取当前环境下的默认编码的方法。

|  |  |
| --- | --- |
| 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 | # coding:gbk    import sys  import locale    def p(f):      print '%s.%s(): %s' % (f.\_\_module\_\_, f.\_\_name\_\_, f())    # 返回当前系统所使用的默认字符编码  p(sys.getdefaultencoding)    # 返回用于转换Unicode文件名至系统文件名所使用的编码  p(sys.getfilesystemencoding)    # 获取默认的区域设置并返回元祖(语言, 编码)  p(locale.getdefaultlocale)    # 返回用户设定的文本数据编码  # 文档提到this function only returns a guess  p(locale.getpreferredencoding)    # \xba\xba是'汉'的GBK编码  # mbcs是不推荐使用的编码，这里仅作测试表明为什么不应该用  print r"'\xba\xba'.decode('mbcs'):", repr('\xba\xba'.decode('mbcs'))    #在笔者的Windows上的结果(区域设置为中文(简体, 中国))  #sys.getdefaultencoding(): gbk  #sys.getfilesystemencoding(): mbcs  #locale.getdefaultlocale(): ('zh\_CN', 'cp936')  #locale.getpreferredencoding(): cp936  #'\xba\xba'.decode('mbcs'): u'\u6c49' |

## 3.一些建议

### 3.1. 使用字符编码声明，并且同一工程中的所有源代码文件使用相同的字符编码声明。

这点是一定要做到的。

### 3.2. 抛弃str，全部使用unicode。

按引号前先按一下u最初做起来确实很不习惯而且经常会忘记再跑回去补，但如果这么做可以减少90%的编码问题。如果编码困扰不严重，可以不参考此条。

### 3.3. 使用codecs.open()替代内置的open()。

如果编码困扰不严重，可以不参考此条。

### 3.4. 绝对需要避免使用的字符编码：MBCS/DBCS和UTF-16。

这里说的MBCS不是指GBK什么的都不能用，而是不要使用Python里名为'MBCS'的编码，除非程序完全不移植。

Python中编码'MBCS'与'DBCS'是同义词，指当前Windows环境中MBCS指代的编码。Linux的Python实现中没有这种 编码，所以一旦移植到Linux一定会出现异常！另外，只要设定的Windows系统区域不同，MBCS指代的编码也是不一样的。分别设定不同的区域运行 2.4小节中的代码的结果：

|  |  |
| --- | --- |
| 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 | #中文(简体, 中国)  #sys.getdefaultencoding(): gbk  #sys.getfilesystemencoding(): mbcs  #locale.getdefaultlocale(): ('zh\_CN', 'cp936')  #locale.getpreferredencoding(): cp936  #'\xba\xba'.decode('mbcs'): u'\u6c49'    #英语(美国)  #sys.getdefaultencoding(): UTF-8  #sys.getfilesystemencoding(): mbcs  #locale.getdefaultlocale(): ('zh\_CN', 'cp1252')  #locale.getpreferredencoding(): cp1252  #'\xba\xba'.decode('mbcs'): u'\xba\xba'    #德语(德国)  #sys.getdefaultencoding(): gbk  #sys.getfilesystemencoding(): mbcs  #locale.getdefaultlocale(): ('zh\_CN', 'cp1252')  #locale.getpreferredencoding(): cp1252  #'\xba\xba'.decode('mbcs'): u'\xba\xba'    #日语(日本)  #sys.getdefaultencoding(): gbk  #sys.getfilesystemencoding(): mbcs  #locale.getdefaultlocale(): ('zh\_CN', 'cp932')  #locale.getpreferredencoding(): cp932  #'\xba\xba'.decode('mbcs'): u'\uff7a\uff7a' |

可见，更改区域后，使用mbcs解码得到了不正确的结果，所以，当我们需要使用'GBK'时，应该直接写'GBK'，不要写成'MBCS'。

UTF-16同理，虽然绝大多数操作系统中'UTF-16'是'UTF-16-LE'的同义词，但直接写'UTF-16-LE'只是多写3个字符而 已，而万一某个操作系统中'UTF-16'变成了'UTF-16-BE'的同义词，就会有错误的结果。实际上，UTF-16用的相当少，但用到的时候还是 需要注意。

--END--