# 为什么要有字符集

我们在计算机屏幕上看到的是**实体化的文字**，而在计算机存储介质中存放的实际是**二进制的比特流**。那 么在这两者之间的**转换规则**就需要一个统一的标准，否则把我们的U盘插到老板的电脑上，文档就乱码了；小伙伴QQ上传过来的文件，在我们本地打开又乱码了。

<http://os.51cto.com/art/201503/467929.htm>

# 什么是字符集

字符集就规定了某个文字对应的二进制数字存放方式（编码）和某串二进制数值代表了哪个文字（解 码）的转换关系。

# 为什么会有那么多字符集标准呢？

这个问题实际非常容易回答。问问自己为什么我们的插头拿到英国就不能用了呢？为什么显示器同时有 DVI，VGA，HDMI，DP这么多接口呢？很多规范和标准在最初制定时并不会意识到这将会是以后全球普适的准则，或者处于组织本身利益就想从本质上区 别于现有标准。于是，就产生了那么多具有相同效果但又不相互兼容的标准了。



# 什么是字符编码

字符集只是一个规则集合的名字，对应到真实生活中，字符集就是对某种语言的称呼。例如：英语，汉语，日语。对于一个字符集来说要正确编码转码一个字符需要三个关键元素：字库表（character repertoire）、编码字符集（coded character set）、字符编码（character encoding form）。

**字库表**: 是一个相当于所有可读或者可显示字符的数据库，字库表决定了整个字符集能够展现表示的所有字符的范围。

**编码字符集**，即用一个编码值 code point来表示一个字符在字库中的位置。

**字符编码**，将编码字符集和实际存储数值之间的转换关系。

一般来说都会直接将code point的值作为编码后的值直接存储。例如在ASCII中A在表中排第65位，而编码后A的数值是0100 0001也即十进制的65的二进制转换结果。

看到这里，可能很多读者都会有和我当初一样的疑问：字库表和编码字符集看来是必不可少的，那既然字库表中的每一个字符都有一个自己的序号，直接把序号作为存储内容就好了。为什么还要多此一举通过字符编码把序号转换成另外一种存储格式呢？

其实原因也比较容易理解：统一字库表的目的是为了能够涵盖世界上所有的字符，但实际使用过程中会发现真正用的上的字符相对整个字库表来说比例非常 低。例如中文地区的程序几乎不会需要日语字符，而一些英语国家甚至简单的ASCII字库表就能满足基本需求。而如果把每个字符都用字库表中的序号来存储的 话，每个字符就需要3个字节（这里以Unicode字库为例），这样对于原本用仅占一个字符的ASCII编码的英语地区国家显然是一个额外成本（存储体积 是原来的三倍）。算的直接一些，同样一块硬盘，用ASCII可以存1500篇文章，而用3字节Unicode序号存储只能存500篇。于是就出现了 UTF-8这样的变长编码。在UTF-8编码中原本只需要一个字节的ASCII字符，仍然只占一个字节。而像中文及日语这样的复杂字符就需要2个到3个字 节来存储。

# UTF-8和Unicode的关系

看完上面两个概念解释，那么解释UTF-8和Unicode的关系就比较简单了。Unicode就是上文中提到的编码字符集，而UTF-8就是字符 编码，即Unicode规则字库的一种实现形式。随着互联网的发展，对同一字库集的要求越来越迫切，Unicode标准也就自然而然的出现。它几乎涵盖了 各个国家语言可能出现的符号和文字，并将为他们编号。详见：[Unicode on Wikipedia](http://en.wikipedia.org/wiki/Unicode)。

Unicode的编号从0000开始一直到10FFFF共分为16个Plane，每个Plane中有65536(2^16)个字符。而UTF-8则只实现了第一 个Plane，可见UTF-8虽然是一个当今接受度最广的字符集编码，但是它并没有涵盖整个Unicode的字库，这也造成了它在某些场景下对于特殊字符 的处理困难（下文会有提到）。

# UTF-8编码简介

为了更好的理解后面的实际应用，我们这里简单的介绍下UTF-8的编码实现方法。即UTF-8的物理存储和Unicode序号的转换关系。

UTF-8编码为变长编码。最小编码单位（code unit）为一个字节。一个字节的前1-3个bit为描述性部分，后面为实际序号部分。

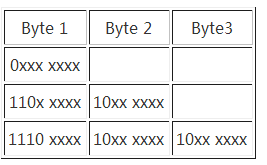
如果一个字节的第一位为0，那么代表当前字符为单字节字符，占用一个字节的空间。0之后的所有部分（7个bit）代表在Unicode中的序号。

如果一个字节以110开头，那么代表当前字符为双字节字符，占用2个字节的空间。110之后的所有部分（7个bit）代表在Unicode中的序号。且第二个字节以10开头

如果一个字节以1110开头，那么代表当前字符为三字节字符，占用2个字节的空间。110之后的所有部分（7个bit）代表在Unicode中的序号。且第二、第三个字节以10开头

如果一个字节以10开头，那么代表当前字节为多字节字符的第二个字节。10之后的所有部分（6个bit）代表在Unicode中的序号。

具体每个字节的特征可见下表，其中x代表序号部分，把各个字节中的所有x部分拼接在一起就组成了在Unicode字库中的序号



# 为什么会出现乱码

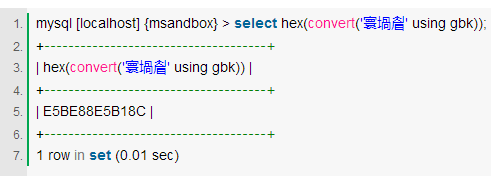
简单的说乱码的出现是因为：编码和解码时用了不同或者不兼容的字符集。对应到真实生活中，就好比是一个英国人为了表示祝福在纸上写了bless（编 码过程）。而一个法国人拿到了这张纸，由于在法语中bless表示受伤的意思，所以认为他想表达的是受伤（解码过程）。这个就是一个现实生活中的乱码情况。在计算机科学中一样，一个用UTF-8编码后的字符，用GBK去解码。由于两个字符集的字库表不一样，同一个汉字在两个字符表的位置也不同，最终就会 出现乱码。

# 如何识别乱码的本来想要表达的文字

要从乱码字符中反解出原来的正确文字需要对各个字符集编码规则有较为深刻的掌握。但是原理很简单，这里用最常见的UTF-8被错误用GBK展示时的乱码为例，来说明具体反解和识别过程.

## 第1步 编码

先通过GBK把乱码编码成二进制表达式。当然查表编码效率很低，我们也可以用以下SQL语句直接通过MySQL客户端做编码工作：



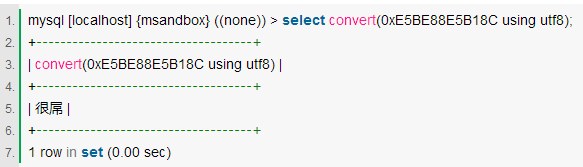
## 第2步 识别

现在我们得到了**解码后的二进制字符串E5BE88E5B18C**。然后我们将它按字节拆开。  
Byte 1Byte 2Byte 3Byte 4Byte 5Byte 6  
E5 BE 88 E5 B1 8C

然后套用之前UTF-8编码介绍章节中总结出的规律，就不难发现这6个字节的数据符合UTF-8编码规则。如果整个数据流都符合这个规则的话，我们就能大胆假设乱码之前的编码字符集是UTF-8

## 第3步 解码

然后我们就能拿着E5BE88E5B18C用UTF-8解码，查看乱码前的文字了。当然我们可以不查表直接通过SQL获得结果：



# [GBK与 UTF-8编码](http://www.cnblogs.com/tmscnz/archive/2012/12/12/2815339.html)区别

## **GBK**

GBK包含全部中文字符，是国家编码，通用性比UTF8差，不过UTF8占用的数据库比GBK大。

中国人为了能够正常使用计算机这一伟大方明，做出了多方面的努力。GB2312就是这一努力的成果， 该标准于1980年发布，1981年5月1日开始实施。它标志着我国在使用电子计算机方面迈出了重要的一步。GB2312 编码共收录了6763个汉字，同时还兼容 ASCII。这一字符编码基本满足了汉字的计算机处理需要，它所收录的汉字已经覆盖中国大陆99.75%的使用频率，对一些古汉语和繁体字 GB2312 没法处理。后来就在GB2312的基础上创建了一种叫 GBK 的编码，于1995年正式发布。GBK 不仅收录了GB 2312 中的全部汉字、非汉字符号，同时还收录了日韩语中出现的汉字，如韩国著名围棋手李世乭中的乭 GBK编码是0x8168(0x表示16进制)。[这里](http://www.qqxiuzi.cn/zh/hanzi-gbk-bianma.php)可以查询汉字对应的GBK编码。

**GBK编码一般用两个字节表示一个字符，如果是英文字母，则使用一个字符，与ASCII编码相同，因此，GBK 也是兼容 ASCII 编码的，**但并不与任何扩展的ASCII编码兼容。这可以从它的编码序列看出来。

GBK 采用双字节表示，总体编码范围为 0x8140-0xFEFE（1000000101000000-1111111011111110），首字节在 0x81-0xFE 之间，尾字节在 0x40-0xFE之间。可以看出首字节最高位都为1，这样一来，**如果尾字节后的字节最高位为0，那么就可以解析为一个ASCII编码字符，**否则就是一个连续的二字节字符

### 例如：

a = "我的English学的不好" #

print type(a), len(a), a

b = **unicode**(a, "gbk")

print type(b), len(b), b

**显示：**

<type '**str**'> 19 ▒ҵ▒Englishѧ▒Ĳ▒▒▒

<type '**unicode**'> 13 我的English学的不好

**备注：len()计算的是字符个数； 对应ascii编码遇到汉字会被解析成2个字符来处理。**

**不清楚的地方：**unicode为何是数据类型？

## ****UTF-8****

Unicode TransformationFormat-8bit，允许含BOM，但通常不含BOM。是用以解决国际上字符的一种多字节编码，**它对英文使用8位（即一个字节）**，**中文使用24为（三个字节）来编码**。UTF-8包含全世界所有国家需要用到的字符，是国际编码，通用性强。UTF-8编码的文字可以在各国支持UTF8字符集的浏览器上显示。如果是UTF8编码，则在外国人的英文IE上也能显示中文，他们无需下载IE的中文语言支持包。

UTF-8版本虽然具有良好的国际兼容性，但中文需要比GBK/BIG5版本多占用50%的数据库存储空间，因此并非推荐使用，仅供对国际兼容性有特殊要求的用户使用。简单地说：对于中文较多的网站，适宜用GBK编码节省数据库空间。对于英文较多的网站，适宜用UTF-8节省数据库空间。

# [文件转换为utf-8编码（python小脚本）](http://blog.csdn.net/csqazwsxedc/article/details/59186769)

## 手动方式

1. 右键文件；
2. 以notepad++打开；
3. 更改文件编码；
4. 保存。

# 中英文混合字串的统一编码

<http://blog.csdn.net/qinbaby/article/details/23201883>

# ASCII编码

众所周知，计算机中的所有数据，不论是文字、图片、视频、还是音频文件，本质上最终都是按照类似 01010101 的二进制形式存储的。然而，计算机中的字符，并不能完全以这种方式来表示。由于计算机最初是由美国人发明的，因而最初的计算机编码使用的也是美国人的标准，即**ASCII( American Standard Code for Information Interchange，美国信息交换标准代码)**。ASCII码一共规定了**128个字符的编码**，比如大写的字母A是65（二进制01000001），符号@的编码是64（二进制01000000）。这128个符号中， 0～31及127(共33个)是控制字符或通信字符，32–126 分配给了能在键盘上找到并且能打印出来的字符。所有**ASCII编码表示的内容，只占用了一个字节的后面7位，最高位统一规定为0。**

****

后来为了能够表示欧洲地区除了英文字母以外的其它字母，出现了扩展的ASCII编码。 扩展的**ASCII**包含**原有的**128个字符，又增加了128个字符，总共是256个。编码时最高位为1，这样就可以与ASCII码完全兼容。可以表示诸如音标æ（编码145，二进制10010001）以及法语中的字母é（编码为130，二进制10000010）等字符。

这个编码能表示音标和欧洲大多数非英语系字母，但是它并不是国际标准，在不同的国家， 128 到 255对应的字符并不完全相同，这就产生了各种不同的扩展ASCII编码。比如 ISO8859-1 字符集，也就是 Latin-1，加入了西欧常用字符，包括德法两国的字母。ISO8859-2 字符集，也称为 Latin-2，收集了东欧字符。 ISO8859-3 字符集，也称为 Latin-3，收集了南欧字符，等等。

这样的编码方式够吗？显然不够，比如汉字，就无法用ASCII表示。扩展的ASCII 也远远不够。

## 备注：

### 1.python实例

python2.7默认使用的是ascii，而现在python3.x默认使用的是unicode。

所以python2.7在运行py后缀文件时也是**默认以ascii编码读取文件**。如果文件中没有中文不会出现问题。但是如果有中文的话，由于中文编码超出了ascii编码范围，所以python2.7将会报错。

**Python2.7提供了 Codecs 模块**，解决各种编码类型的字符串解码与编码问题。

例如： ann\_file = codecs.open(DICTIONARY, 'r', 'utf-8')

**例如：**

a = "我的English学的不好" #

print type(a),len (a), a

**显示**：

**<type 'str'> 19 ▒ҵ▒Englishѧ▒Ĳ▒▒▒**

**ASCII编码，英文字符占一个字节。**汉字，就无法用ASCII表示。扩展的ASCII 也远远不够。所以显示才会出来乱码。

### 2. 查看默认编码方法：

import sys

print sys.getdefaultencoding() # ascii

# Unicode

## 字符与字节

基本事实是，若想正确的处理文本，就必须了解字符的抽象概念。不严谨的定义一下，**字符表示的是文本中的单个符号**。更重要的是，一个字符不是一个字节。我再强调一遍！一个字符不是一个字节！！！而且，**一个字符有许多表示方法，不同的表示方法会使用不同的字节数**。就像前面我说的那样，**字符就是文本中最小的单元**。

Unicode以大家都认可的方式定义了一系列的字符。可以将**Unicode理解成一个字符数据库，每个字符都与唯一的数字关联，称为code point。**这样，英文大写字母A的codepoint是U+0041。而欧元符号的codepoint是U+20A0，其他类似。一个文本字符串就是这样一系列的codepoint，表示字符串中每个字符元素。

由于同一个字符的字节表现形式不止一种。这意味着当遇到了一串字节，如果不知道使用的是什么编码，即使知道这些字节表示的是文本，也不知道是什么意思。所能做的就是猜使用的编码。简而言之，**字节不是文本**。即使忘了文中介绍的所有内容，也要记住这句话。

Python, unicode编码每一个字符占两个字节

## python里，string object与unicode object类型的区别

**string object**是由characters（字符）组成的sequence，而**unicode object**是**Unicode code units**组成的sequence。

### String类型

**string里的character是有多种编码方式的**，比如单字节的ASCII，双字节的GB2312等等，再比如UTF-8。很明显要想解读string，必需知道string里的character是用哪种编码方式，然后才能进行。

### Unicode类型

Unicode code unit又是什么东西呢？**一个Unicode code unit是一个16-bit或者32-bit的数值，每个数值代表一个unicode符号**。**在python里，16-bit的unicode，对应的是ucs2编码。**32-bit对应的是ucs4编码。是不是感觉string里character的编码没什么区别？反正我现在脑子里就是这样一个印象：在Python里，ucs2或者ucs4编码的，我们叫做unicode object，其他编码的我们就叫做string。

至于python里的unicode到底是ucs2还是ucs4的，可以在编译时指定。例如Linux下，要用ucs2做unicode的编码，可以这样  
# **./configure --enable-unicode=ucs2**  
# make  
# make install

下载的Windows预编译版本，一般都是ucs2的。要想知道某个python运行环境是ucs2还是ucs4，可以查看**sys.maxunicde，65535就是ucs2的，另一个很大的数值就是ucs4。**

### Str 与 unicodes转换

s = "人生苦短"

print type(s) #str

s1 = unicode(s, "gbk") #“utf-8”会报错

print type(s1) #unicode

s1 = s1.encode("utf-8")

print type(s1) #str

#ss = unicode(s, "gbk")

print s1 #正确显示

### 备注：

1、string直接用引号来表示，unicode在引号前加一个u （如：b = u'你好'）

2、直接输入的string常量会用系统缺省编码方式来编码，例如在GBK环境下，'你好'会编码成'/xc4/xe3/xba/xc3'，而在UTF-8环境下就成了'/xe4/xbd/xa0/xe5/xa5/xbd'。

3、len(string)返回string的字节数，len(unicode)返回的是字符数  
4、很重要的一点，print unicode不会乱码。

### Python是如何处理Unicode

**字符编码（encoding）**是在理想的**字符**和实际的**字节**表示方法之间的映射.

**sys.setdefaultencoding()** 设置默认编码，每次使用都需要设置；

在Python的类型层次中，有3种不同的字符串类型：“unicode”，表示Unicode字符串（文本字符串）、“str”，表示字节字符串（二进制数据）；“basestring”, 表示前两种字符串类型的父类。

正确的解决方法是修改代码，以正确的方式处理文本。下面是一些应该做到的指导性意见：

* 所有**文本字符串**都应该是unicode类型，而不是str类型。如果处理的是文本，而变量类型是str，这就是bug了！
* 若要将字节串解码成字符串，需要使用正确的解码，即var.decode(encoding)（如，var.decode('utf-8')）。将文本字符串编码成字节，使用var.encode(encoding)。
* 永远不要对unicode字符串使用str()，也不要在不指定编码的情况下就对字节串使用unicode()。
* 当应用**从外部读取数据**时，应将其视为**字节串**，即str类型的，接着调用.decode()将其解释成文本。同样，在将文本发送到外部时，总是对文本调用.encode()。
* 如果代码中使用字符串字面值来表示文本，总是应该含有’u’前缀。但实际上，永远不要在代码中定义原始的字符串字面值。不管怎样，我自己是很讨厌这一条，也许其他人也和我一样吧。

<http://python.jobbole.com/80939/>

# 字典树

<http://stevehanov.ca/blog/index.php?id=114> 主要代码来源

编辑距离（Edit Distance）

又称Levenshtein距离，是指两个字串之间，由一个转成另一个所需的最少编辑操作次数。

许可的编辑操作包括将一个字符替换成另一个字符，插入一个字符，删除一个字符。

例如将kitten一字转成sitting：编辑距离=3

sitten （k→s）

sittin （e→i）

sitting （→g）

俄罗斯科学家Vladimir Levenshtein在1965年提出这个概念。