标题：深入浅出字符编码

[深入浅出字符编码 3](#_Toc269155890)

[1 编码常识 3](#_Toc269155891)

[1.1 字符集与字符编码 3](#_Toc269155892)

[1.2 字节序（Byte Order） 3](#_Toc269155893)

[2 字符的输入与输出 3](#_Toc269155894)

[2.1 输入输出原理 3](#_Toc269155895)

[2.2 输入输出过程 3](#_Toc269155896)

[2.2.1 字符的编码与解码 3](#_Toc269155897)

[2.2.2 字符的输入 3](#_Toc269155898)

[2.2.3 从外码到内码的转化 3](#_Toc269155899)

[2.2.4 字符的显示 3](#_Toc269155900)

[3 常见字符集及其编码方式的介绍 3](#_Toc269155901)

[3.1 ASCII字符集及其编码 3](#_Toc269155902)

[3.2 ISO 8859系列 3](#_Toc269155903)

[3.2.1 ISO 8859字符集与编码 3](#_Toc269155904)

[3.2.2 ISO 8859字符集介绍 3](#_Toc269155905)

[3.2.2.1 ISO 8859-1字符集 3](#_Toc269155906)

[3.2.2.2 ISO 8859字符集概括 3](#_Toc269155907)

[3.3 中文字符集及其编码 3](#_Toc269155908)

[3.3.1 GB2312-80 3](#_Toc269155909)

[3.3.2 GBK 3](#_Toc269155910)

[3.3.3 GB18030 3](#_Toc269155911)

[3.3.4 BIG5 3](#_Toc269155912)

[3.4 UCS和Unicode 3](#_Toc269155913)

[3.4.1 UCS介绍 3](#_Toc269155914)

[3.4.1.1 概要 3](#_Toc269155915)

[3.4.1.2 UCS使用划分 3](#_Toc269155916)

[3.4.1.3 UCS组合字符 3](#_Toc269155917)

[3.4.1.4 UCS的实现级别 3](#_Toc269155918)

[3.4.1.5 UCS编码方式 3](#_Toc269155919)

[3.4.2 Unicode介绍 3](#_Toc269155920)

[3.4.2.1 概要 3](#_Toc269155921)

[3.4.2.2 Unicode发布的版本 3](#_Toc269155922)

[3.4.2.3 编码方式 3](#_Toc269155923)

[3.4.2.4 辅助平面 3](#_Toc269155924)

[3.4.3 UCS与Unicode的异同 3](#_Toc269155925)

[3.4.4 UTF（Unicode/UCS Translation Format） 3](#_Toc269155926)

[3.4.4.1 UTF简介 3](#_Toc269155927)

[3.4.4.2 UTF-8 3](#_Toc269155928)

[3.4.4.3 UTF-16 3](#_Toc269155929)

[3.4.4.4 UTF-32 3](#_Toc269155930)

[4 程序如何处理字符编码 3](#_Toc269155931)

[4.1 程序中为什么会出现乱码 3](#_Toc269155932)

[4.2 HTML文件编码 3](#_Toc269155933)

[4.2.1 “charset“参数的意义 3](#_Toc269155934)

[4.2.2 如何选择一种编码 3](#_Toc269155935)

[4.2.3 如何指定HTML文件的编码 3](#_Toc269155936)

[4.2.4 字符引用 3](#_Toc269155937)

[4.2.5 浏览器如何处理不可显示的字符 3](#_Toc269155938)

[4.2.6 几个问题 3](#_Toc269155939)

[4.3 XML文件编码如何识别 3](#_Toc269155940)

[4.4 字符与编码在程序中的实现 3](#_Toc269155941)

[4.4.1 程序中的字符与字节 3](#_Toc269155942)

[4.4.1.1 C++ 中相关实现方法 3](#_Toc269155943)

[4.4.1.2 Java中相关实现方法 3](#_Toc269155944)

[4.4.2 几种乱码产生的原因和解决办法 3](#_Toc269155945)

[4.4.2.1 容易产生的误解 3](#_Toc269155946)

[4.4.2.2 非UNICODE程序在不同语言环境间移植时的乱码 3](#_Toc269155947)

[4.4.2.3 深入理解URL编码及乱码问题 3](#_Toc269155948)

[4.4.2.4 从数据库读取字符串 3](#_Toc269155949)

[4.4.2.5 电子邮件中的字符串 3](#_Toc269155950)

[4.4.3 几个错误理解的纠正 3](#_Toc269155951)

[4.4.3.1 误解1：“ISO-8859-1 是国际编码？” 3](#_Toc269155952)

[4.4.3.2 误解2：“Java中，怎样知道某个字符串的内码？” 3](#_Toc269155953)

[5 一个通用的编码识别算法简介 3](#_Toc269155954)

[6 参考资料 3](#_Toc269155955)

深入浅出字符编码

# 编码常识

## 字符集与字符编码

**字符**(Character)是文字与符号的总称，包括文字、图形符号、数学符号等。

一组抽象字符的集合就是**字符集**(Charset)。

字符集常常和一种具体的语言文字对应起来，该文字中的所有字符或者大部分常用字符就构成了该文字的字符集，比如英文字符集。一组有共同特征的字符也可以组成字符集，比如繁体汉字字符集、日文汉字字符集。字符集的子集也是字符集。

计算机要处理各种字符，就需要将字符和二进制内码对应起来，这种对应关系就是**字符编码**(Encoding)。制定编码首先要确定字符集，并将字符集内的字符排序，然后和二进制数字对应起来。根据字符集内字符的多少，来确定用几个字节来编码。

每种编码都限定了一个明确的字符集合，叫做**被编码过的字符集**(Coded Character Set)，这是字符集的另外一个含义。通常所说的字符集大多是这个含义。

## 字节序（Byte Order）

谈到字节序的问题，必然牵涉到两大CPU派系。那就是Motorola的PowerPC系列CPU和Intel的x86系列CPU。PowerPC系列采用**Big Endian**方式存储数据，而x86系列则采用**Little Endian**方式存储数据。那么究竟什么是Big Endian，什么又是Little Endian呢？

其实big endian是指低地址存放最高有效字节（MSB），而little endian则是低地址存放最低有效字节（LSB），即常说的低位在先，高位在后。

用文字说明可能比较抽象，下面用图像加以说明。比如数字0x12345678在两种不同字节序CPU中的存储顺序如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **Big Endian** | | **Little Endian** | |
| 低地址 | 高地址 | 低地址 | 高地址 |
| | 12 | 34 | 56 | 78 | | | | 78 | 56 | 34 | 12 | |

从上面两图可以看出，采用Big Endian方式存储数据是符合我们人类的思维习惯的。而Little Endian，不太好理解哦@\_@~

* **为什么要注意字节序**

你可能这么问。当然，如果你写的程序只在单机环境下面运行，并且不和别人的程序打交道，那么你完全可以忽略字节序的存在。但是，如果你的程序要跟别人的程序产生交互呢？尤其是当你把你在微机上运算的结果运用到计算机群上去的话。

C/C++语言编写的程序里数据存储顺序是跟编译平台所在的CPU相关的，而JAVA编写的程序则唯一采用Big Endian方式来存储数据。试想，如果你用C/C++语言在x86平台下编写的程序跟别人的JAVA程序互通时会产生什么结果？就拿上面的0x12345678来说，你的程序传递给别人的一个数据，将指向0x12345678的指针传给了JAVA程序，由于JAVA采取Big Endian方式存储数据，很自然的它会将你的数据翻译为0x78563412。什么？竟然变成另外一个数字了？是的，就是这种后果。因此，在你的C程序传给JAVA程序之前有必要进行字节序的转换工作。

无独有偶，所有网络协议也都是采用Big Endian的方式来传输数据的。所以有时我们也会把Big Endian方式称之为**网络字节序**。当两台采用不同字节序的主机通信时，在发送数据之前都必须经过字节序的转换成为网络字节序后再进行传输。ANSI C中提供了四个转换字节序的宏。

* **各种字符编码的字节序**

UNICODE16-LE

UNICODE16-BE

UTF-8（BE）

GBK(BE)

# 字符的输入与输出

这一节主要解决3个问题：

1. 字符如何输入（输入码怎么表示）
2. 字符在计算机内如何存储
3. 字符如何在屏幕或打印机上显示

## 输入输出原理

**各种输入码（外码）**

**交换码（国标码）**

内码

字形码

显示汉字

打印汉字

## 输入输出过程

### 字符的编码与解码

**不认图只认数**

我们知道计算机所认识的数字只有两个，即0和1，不同0、1组合组成二进制数字。我们在显示器上看到的文字对计算机而言其实是一种不认识的图形，计算机是不能直接处理图形的，所以我们需要将字符的图形保存在电脑的存储器中，然后操作系统中制定一个对应的规则让字符图形和数字逐一对应。其中，字符图形的集合就是我们的**字体库**，而对应的规则就是**字符编码**了。

**字符的编码和解码**

当我们通过输入法输入文字后，系统根据字符编码规则将字符信息转换成为计算机可以处理的二进制代码，这就是**字符编码**。字符编码就是将字符编码成为数字序列，以便计算机识别。当计算机处理完字符信息后，系统将通过根据一定的文字编码规则到字体库中查找二进制数字代码所对应的字符图形，然后显示在屏幕上，这样就能够在屏幕上看到该文字，这就是**字符解码**。

也就是说，在字符输入到显示在屏幕需要经过两个过程——**编码和解码**，在编码和解码的过程中都需要通过**字符编码规则**，从计算机发展至今都有哪些字符编码规则？

在中国，我们比较熟悉的编码规则有GB2312，GBK等等，在美国用ASCII，欧州有ISO-8859，还有全国大统一编码Unicode，等等。

### 字符的输入

这里以中文操作系统为例来说明计算机如何处理字符输入的。

汉字的输入码也称为**外码**，是指从键盘上输入汉字时使用的编码，它与汉字的内码相对应。主要有三类：数字编码，拼音编码和字型编码。

数字编码是用一串数字代表一个汉字。最常用的是区位码输入法。它是把国标码的每一个字节减去00100000(20H)得到的，被减去的是控制字符。高字节为区码，低字节为位码。无重码，难记忆。

拼音编码是以汉字读音为基础的输入方法。简单，重码率高，输入速度较慢。常见的有微软、搜狗、谷歌、紫光、腾讯拼音输入法，智能ABC等。

字形编码是根据汉字形状确定的编码。因为构成汉字的部件是一定的，所以对这些部件进行编码，按书写顺序依次输入，就能表示一个汉字。常用的就是五笔输入法，如操作系统自带的86五笔输入法、搜狗五笔输入法、极点五笔输入法、陈桥五笔输入法等。

|  |  |  |
| --- | --- | --- |
|  | **优点** | **缺点** |
| **拼音码** | 只需会拼音，容易使用 | 同音字太多，输入重码率高 |
| **字型编码** | 重码率低，输入效率高 | 需要记字根 |
| **数字编码** | 无重码 | 代码难以记忆 |

### 从外码到内码的转化

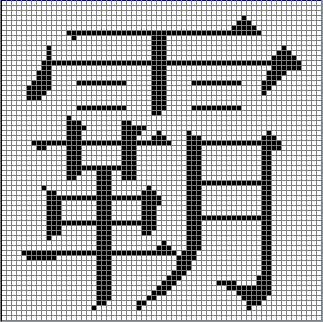
各种输入法自身都有一套从外码到内码的转化规则。如果操作系统的默认编码，相应的输入法会根据外码转化为相应的内码。并不是说所以的输入法都能顺利输入如GBK字符集中的一个字符，取决于该输入法的转化规则是否完备。很有可能碰到某种输入法不能输入某个汉字的情况。

### 字符的显示

字符在计算机中一般有**字符码**（内码）和**字形码**(可以看做字体)，字符码是字形码的索引，计算机通过字符码找到字形码，然后通过字形码显示出来。

字形码有**点阵字形**与**矢量字形**两种表示方法。

**点阵字形**是用点阵表示的汉字字形代码，它是汉字的输出形式。根据汉字输出的要求不同，点阵的多少也不同。字模点阵的信息量是很大的，所占存储空间也很大。以16×16点阵为例，每个汉字要占用32个字节。简易汉字为16×16点阵，提高型汉字为24×24点阵、32×32点阵，甚至更高。



例如：

“霸”的一种字形码：

64×64点阵

共4096个点

占512字节存储空间

注：可以用windows的造字程序来设计自己的字形码。

**矢量字形**，就是用矢量来表示字形。

# 常见字符集及其编码方式的介绍

编码发展的三个阶段：

* **ASCII**以及**EASCII**

美国人、欧州人的天下，谁叫人家发达呢！

* **ANSI**

计算机发展了一定的阶段，特别是亚州等国家崛起了，但是人家老外这一套拿来不好用。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。

当然对于ANSI编码而言，0x00~0x7F之间的字符，依旧是1个字节代表1个字符。这一点是ASNI编码与Unicode编码之间最大也最明显的区别。比如“A君是第131号”，在ANSI编码中，占用12个字节，而在Unicode编码中，占用16个字节。因为A和1、3、1这4个字符，在ANSI编码中只各占1位，而在Unicode编码中，是需要各占2位的。

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GB 2312编码，在日文操作系统下，ANSI编码代表JIS编码。我们也习惯称这些编码方式为**DBCS**（**Double Byte Character Set**），而ASCII与扩展ASCII称为**SBCS(Single Byte Character Set**)。

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。

* **关于ansi编码的BUG**

很多细心的人会发现，当新建文本文档只输入“联通”2字保存再打开时将是乱码。

当txt文档中一切字符都在C0（11000000）≤AA≤DF（11011111） 80（1000000）≤BB≤BF（10111111）这个范围时【这个正好是UTF-8两字节的表示模式，详见3.4.4.2】，notepad都无法确认文档地格式【到底是UTF-8呢还是ANSI，因为所有有字符都落在这个区间，所以没有更多的信息推断出编码方式来】，没有自动依照UTF-8格式来显示。而“联通”就是C1 AA CD A8，刚好在上面地范围内，所以不能正常显现。

记事本默认是以ANSI编码保存文本文档的，而正是这种编码存在的bug招致了上述怪现象。。假如保存时选择Unicode、Unicode(big endian)、UTF-8编码就正常了。此外，假如以ANSI编码保存含有某些特别符号的文本文档，再次打开后符号也会变成英文问号。写字板要强大一些，修正了这个Bug

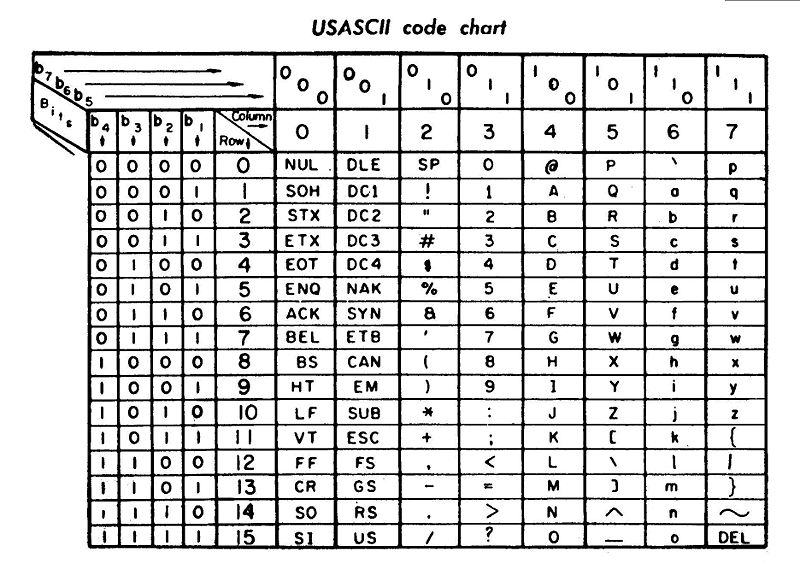
* UNICODE一统天下

末来UTF-8是全世界通通的标准！

## ASCII字符集及其编码

**ASCII**（American Standard Code for Information Interchange，**美国信息互换标准代码**）是基于[拉丁字母](http://zh.wikipedia.org/zh-cn/%E6%8B%89%E4%B8%81%E5%AD%97%E6%AF%8D)的一套[电脑](http://zh.wikipedia.org/zh-cn/%E7%94%B5%E8%84%91)[编码](http://zh.wikipedia.org/zh-cn/%E7%BC%96%E7%A0%81)系统。它主要用于显示[现代英语](http://zh.wikipedia.org/zh-cn/%E7%8F%BE%E4%BB%A3%E8%8B%B1%E8%AA%9E)，而其扩展版本[**EASCII**](http://zh.wikipedia.org/zh-cn/EASCII)则可以勉强显示其他[西欧](http://zh.wikipedia.org/zh-cn/%E8%A5%BF%E6%AC%A7)[语言](http://zh.wikipedia.org/zh-cn/%E8%AF%AD%E8%A8%80)。它是现今最通用的单[字节](http://zh.wikipedia.org/zh-cn/%E5%AD%97%E8%8A%82)编码系统（但是有被UniCode追上的迹象），并等同于国际标准[ISO/IEC 646](http://zh.wikipedia.org/zh-cn/ISO/IEC_646)。

ASCII第一次以规范标准发表是在[1967年](http://zh.wikipedia.org/zh-cn/1967%E5%B9%B4)，最后一次更新则是在[1986年](http://zh.wikipedia.org/zh-cn/1986%E5%B9%B4)，至今为止共定义了128个字符；其中**33个字符无法显示**（这是以现今[操作系统](http://zh.wikipedia.org/zh-cn/%E4%BD%9C%E6%A5%AD%E7%B3%BB%E7%B5%B1)为依归，但在[DOS](http://zh.wikipedia.org/zh-cn/DOS)模式下可显示出一些诸如[笑脸](http://zh.wikipedia.org/w/index.php?title=%E7%AC%91%E8%87%89&action=edit&redlink=1)、[扑克牌花式](http://zh.wikipedia.org/w/index.php?title=%E6%92%B2%E5%85%8B%E7%89%8C%E8%8A%B1%E5%BC%8F&action=edit&redlink=1)等8-bit符号），且这33个字符多数都已是废旧的[控制字符](http://zh.wikipedia.org/zh-cn/%E6%8E%A7%E5%88%B6%E5%AD%97%E5%85%83)。控制字符的用途主要是用来操控已经处理过的文字。在33个字符之外的是**95个可显示的字符**，包含用[键盘](http://zh.wikipedia.org/zh-cn/%E9%8D%B5%E7%9B%A4)敲下空白键所产生的空白字符也算1个可显示字符（显示为空白）。

[](http://upload.wikimedia.org/wikipedia/commons/8/85/ASCII_Code_Chart-Quick_ref_card.jpg)

## ISO 8859系列

ASCII收录了空格及94个“可印刷字符”，足以给英语使用。但是，其他使用拉丁字母的语言(主要是欧洲国家的语言)，都有一定数量的变音字母，故可以使用ASCII及控制字符以外的区域来储存及表示。除了使用拉丁字母的语言外，使用西里尔字母的东欧语言、希腊语、泰语、现代阿拉伯语、希伯来语等，都可以使用这个形式来储存及表示。为此，国际标准化组织又制定了**ISO2022**标准，它规定了在保持与ISO646兼容的前提下将ASCII字符集扩充为8位代码的统一方法。ISO陆续制定了一批适用于不同地区的扩充ASCII字符集，每种扩充ASCII字符集分别可以扩充128个字符，这些扩充字符的编码均为高位为1的8位代码（即十进制数128~255），称为**扩展ASCII码（**[**EASCII**](http://zh.wikipedia.org/zh-cn/EASCII)**）**。

目前使用最广泛的西文字符集及其编码是ISO 8859，全称ISO/IEC 8859，是国际标准化组织（ISO）及国际电工委员会(IEC)联合制定的**一系列8位字符集的标准**，现时定义了15个字符集。

### ISO 8859字符集与编码

ISO 8859不是一个标准，而是一系列的标准，这套字符集与编码系统的共同特色是，**以同样的码位（0x00–0xFF）对应不同字符集**。其基本精神是：

1.与 ASCII 相容，所以所有的低位皆不使用。

2.高位中的前32个码位(0x80-0x9F或128-159)，保留给扩充定义的32个控制码，称为C1控制码(0-31称为C0控制码)。

3.高位中第33个码位(0xA0或160)，也就是对应ASCII中SP(空格)的码位，总是代表Non-breakable space，也就是不准许折行的空格。

4.每个字符集定义至多95个字符，其码位都在0xA1-0xFF或161-255。

5.每个字符集收录欧洲某地区的共同常用字符。

### ISO 8859字符集介绍

#### ISO 8859-1字符集

**ISO 8859-1**，正式编号为**ISO/IEC 8859-1:1998**，又称Latin-1或“西欧语言”，是[国际标准化组织](http://zh.wikipedia.org/zh-cn/%E5%9C%8B%E9%9A%9B%E6%A8%99%E6%BA%96%E5%8C%96%E7%B5%84%E7%B9%94)内[ISO/IEC 8859](http://zh.wikipedia.org/zh-cn/ISO/IEC_8859)的第一个8位字符集。它以[ASCII](http://zh.wikipedia.org/zh-cn/ASCII)为基础，在空置的0xA0-0xFF的范围内，加入96个字母及符号，藉以供使用[附加符号](http://zh.wikipedia.org/zh-cn/%E9%99%84%E5%8A%A0%E7%AC%A6%E5%8F%B7)的[拉丁字母](http://zh.wikipedia.org/zh-cn/%E6%8B%89%E4%B8%81%E5%AD%97%E6%AF%8D)语言使用。曾推出过ISO 8859-1:1987版。

此字符集支援部分于[欧洲](http://zh.wikipedia.org/zh-cn/%E6%AC%A7%E6%B4%B2)使用的语言，包括[阿尔巴尼亚语](http://zh.wikipedia.org/zh-cn/%E9%98%BF%E5%B0%94%E5%B7%B4%E5%B0%BC%E4%BA%9A%E8%AF%AD)、[巴斯克语](http://zh.wikipedia.org/zh-cn/%E5%B7%B4%E6%96%AF%E5%85%8B%E8%AF%AD)、[布列塔尼语](http://zh.wikipedia.org/zh-cn/%E5%B8%83%E5%88%97%E5%A1%94%E5%B0%BC%E8%AF%AD)、[加泰罗尼亚语](http://zh.wikipedia.org/zh-cn/%E5%8A%A0%E6%B3%B0%E7%BD%97%E5%B0%BC%E4%BA%9A%E8%AF%AD)、[丹麦语](http://zh.wikipedia.org/zh-cn/%E4%B8%B9%E9%BA%A6%E8%AF%AD)、[荷兰语](http://zh.wikipedia.org/zh-cn/%E8%8D%B7%E5%85%B0%E8%AF%AD)、[法罗语](http://zh.wikipedia.org/zh-cn/%E6%B3%95%E7%BD%97%E8%AF%AD)、[弗里西语](http://zh.wikipedia.org/zh-cn/%E5%BC%97%E9%87%8C%E8%A5%BF%E8%AF%AD)、[加利西亚语](http://zh.wikipedia.org/zh-cn/%E5%8A%A0%E5%88%A9%E8%A5%BF%E4%BA%9A%E8%AF%AD)、[德语](http://zh.wikipedia.org/zh-cn/%E5%BE%B7%E8%AF%AD)、[格陵兰语](http://zh.wikipedia.org/zh-cn/%E6%A0%BC%E9%99%B5%E5%85%B0%E8%AF%AD)、[冰岛语](http://zh.wikipedia.org/zh-cn/%E5%86%B0%E5%B2%9B%E8%AF%AD)、[爱尔兰盖尔语](http://zh.wikipedia.org/zh-cn/%E7%88%B1%E5%B0%94%E5%85%B0%E7%9B%96%E5%B0%94%E8%AF%AD)、[意大利语](http://zh.wikipedia.org/zh-cn/%E6%84%8F%E5%A4%A7%E5%88%A9%E8%AF%AD)、[拉丁语](http://zh.wikipedia.org/zh-cn/%E6%8B%89%E4%B8%81%E8%AF%AD)、[卢森堡语](http://zh.wikipedia.org/zh-cn/%E5%8D%A2%E6%A3%AE%E5%A0%A1%E8%AF%AD)、[挪威语](http://zh.wikipedia.org/zh-cn/%E6%8C%AA%E5%A8%81%E8%AF%AD)、[葡萄牙语](http://zh.wikipedia.org/zh-cn/%E8%91%A1%E8%90%84%E7%89%99%E8%AF%AD)、[里托罗曼斯语](http://zh.wikipedia.org/zh-cn/%E9%87%8C%E6%89%98%E7%BD%97%E6%9B%BC%E6%96%AF%E8%AF%AD)、[苏格兰盖尔语](http://zh.wikipedia.org/zh-cn/%E8%8B%8F%E6%A0%BC%E5%85%B0%E7%9B%96%E5%B0%94%E8%AF%AD)、[西班牙语](http://zh.wikipedia.org/zh-cn/%E8%A5%BF%E7%8F%AD%E7%89%99%E8%AF%AD)及[瑞典语](http://zh.wikipedia.org/zh-cn/%E7%91%9E%E5%85%B8%E8%AF%AD)。

[英语](http://zh.wikipedia.org/zh-cn/%E8%8B%B1%E8%AF%AD)虽然没有重音字母，但仍会标明为ISO/IEC 8859-1编码。除此之外，欧洲以外的部分语言，如[南非荷兰语](http://zh.wikipedia.org/zh-cn/%E5%8D%97%E9%9D%9E%E8%8D%B7%E5%85%B0%E8%AF%AD)、[斯瓦希里语](http://zh.wikipedia.org/zh-cn/%E6%96%AF%E7%93%A6%E5%B8%8C%E9%87%8C%E8%AF%AD)、[印尼语](http://zh.wikipedia.org/zh-cn/%E5%8D%B0%E5%B0%BC%E8%AF%AD)及[马来语](http://zh.wikipedia.org/zh-cn/%E9%A9%AC%E6%9D%A5%E8%AF%AD)、菲律宾[他加洛语](http://zh.wikipedia.org/zh-cn/%E4%BB%96%E5%8A%A0%E6%B4%9B%E8%AF%AD)等也可使用ISO/IEC 8859-1编码。

[法语](http://zh.wikipedia.org/zh-cn/%E6%B3%95%E8%AF%AD)及芬兰语本来也使用ISO/IEC 8859-1来表示。但因它没有法语使用的œ、Œ、Ÿ三个字母及芬兰语使用的Š、š、Ž、ž，故于1998年被[ISO/IEC 8859-15](http://zh.wikipedia.org/zh-cn/ISO/IEC_8859-15)所取代。（ISO 8859-15同时加入了[欧元](http://zh.wikipedia.org/zh-cn/%E6%AC%A7%E5%85%83)符号）。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ISO/IEC 8859-1** | | | | | | | | | | | | | | | | | |
|  | **x0** | **x1** | **x2** | **x3** | **x4** | **x5** | **x6** | **x7** | **x8** | **x9** | **xA** | **xB** | | **xC** | **xD** | **xE** | **xF** |
| **0x** |  | | | | | | | | | | | | | | | | |
| **1x** |  | | | | | | | | | | | | | | | | |
| **2x** | *SP* | ! | " | # | $ | % | & | ' | ( | ) | \* | + | , | - | | . | / |
| **3x** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | | > | ? |
| **4x** | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | | N | O |
| **5x** | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | | ^ | \_ |
| **6x** | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | | n | o |
| **7x** | p | q | r | s | t | u | v | w | x | y | z | { | | | } | | ~ |  |
| **8x** |  | | | | | | | | | | | | | | | | |
| **9x** |  | | | | | | | | | | | | | | | | |
| **Ax** | *NBSP* | ¡ | ¢ | £ | ¤ | ¥ | ¦ | § | ¨ | © | ª | « | ¬ | *SHY* | | ® | ¯ |
| **Bx** |  | ± | ² | ³ | ´ | µ | ¶ | · | ¸ | ¹ | º | » | ¼ | ½ | | ¾ | ¿ |
| **Cx** | [À](http://zh.wikipedia.org/zh-cn/%C3%80) | [Á](http://zh.wikipedia.org/zh-cn/%C3%81) | [Â](http://zh.wikipedia.org/zh-cn/%C3%82) | [Ã](http://zh.wikipedia.org/zh-cn/%C3%83) | [Ä](http://zh.wikipedia.org/zh-cn/%C3%84) | [Å](http://zh.wikipedia.org/zh-cn/%C3%85) | [Æ](http://zh.wikipedia.org/zh-cn/%C3%86) | [Ç](http://zh.wikipedia.org/zh-cn/%C3%87) | [È](http://zh.wikipedia.org/zh-cn/%C3%88) | [É](http://zh.wikipedia.org/zh-cn/%C3%89) | [Ê](http://zh.wikipedia.org/zh-cn/%C3%8A) | [Ë](http://zh.wikipedia.org/zh-cn/%C3%8B) | [Ì](http://zh.wikipedia.org/zh-cn/%C3%8C) | [Í](http://zh.wikipedia.org/zh-cn/%C3%8D) | | [Î](http://zh.wikipedia.org/zh-cn/%C3%8E) | [Ï](http://zh.wikipedia.org/zh-cn/%C3%8F) |
| **Dx** | [Ð](http://zh.wikipedia.org/zh-cn/%C3%90) | [Ñ](http://zh.wikipedia.org/zh-cn/%C3%91) | [Ò](http://zh.wikipedia.org/zh-cn/%C3%92) | [Ó](http://zh.wikipedia.org/zh-cn/%C3%93) | [Ô](http://zh.wikipedia.org/zh-cn/%C3%94) | [Õ](http://zh.wikipedia.org/zh-cn/%C3%95) | [Ö](http://zh.wikipedia.org/zh-cn/%C3%96) | [×](http://zh.wikipedia.org/zh-cn/%C3%97) | [Ø](http://zh.wikipedia.org/zh-cn/%C3%98) | [Ù](http://zh.wikipedia.org/zh-cn/%C3%99) | [Ú](http://zh.wikipedia.org/zh-cn/%C3%9A) | [Û](http://zh.wikipedia.org/zh-cn/%C3%9B) | [Ü](http://zh.wikipedia.org/zh-cn/%C3%9C) | [Ý](http://zh.wikipedia.org/zh-cn/%C3%9D) | | [Þ](http://zh.wikipedia.org/zh-cn/%C3%9E) | [ß](http://zh.wikipedia.org/zh-cn/%C3%9F) |
| **Ex** | [à](http://zh.wikipedia.org/zh-cn/%C3%80) | [á](http://zh.wikipedia.org/zh-cn/%C3%81) | [â](http://zh.wikipedia.org/zh-cn/%C3%82) | [ã](http://zh.wikipedia.org/zh-cn/%C3%83) | [ä](http://zh.wikipedia.org/zh-cn/%C3%84) | [å](http://zh.wikipedia.org/zh-cn/%C3%85) | [æ](http://zh.wikipedia.org/zh-cn/%C3%86) | [ç](http://zh.wikipedia.org/zh-cn/%C3%87) | [è](http://zh.wikipedia.org/zh-cn/%C3%88) | [é](http://zh.wikipedia.org/zh-cn/%C3%89) | [ê](http://zh.wikipedia.org/zh-cn/%C3%8A) | [ë](http://zh.wikipedia.org/zh-cn/%C3%8B) | [ì](http://zh.wikipedia.org/zh-cn/%C3%8C) | [í](http://zh.wikipedia.org/zh-cn/%C3%8D) | | [î](http://zh.wikipedia.org/zh-cn/%C3%8E) | [ï](http://zh.wikipedia.org/zh-cn/%C3%8F) |
| **Fx** | [ð](http://zh.wikipedia.org/zh-cn/%C3%90) | [ñ](http://zh.wikipedia.org/zh-cn/%C3%91) | [ò](http://zh.wikipedia.org/zh-cn/%C3%92) | [ó](http://zh.wikipedia.org/zh-cn/%C3%93) | [ô](http://zh.wikipedia.org/zh-cn/%C3%94) | [õ](http://zh.wikipedia.org/zh-cn/%C3%95) | [ö](http://zh.wikipedia.org/zh-cn/%C3%96) | ÷ | [ø](http://zh.wikipedia.org/zh-cn/%C3%98) | [ù](http://zh.wikipedia.org/zh-cn/%C3%99) | [ú](http://zh.wikipedia.org/zh-cn/%C3%9A) | [û](http://zh.wikipedia.org/zh-cn/%C3%9B) | [ü](http://zh.wikipedia.org/zh-cn/%C3%9C) | [ý](http://zh.wikipedia.org/zh-cn/%C3%9D) | | [þ](http://zh.wikipedia.org/zh-cn/%C3%9E) | [ÿ](http://zh.wikipedia.org/zh-cn/%C5%B8) |

* 在上表中，0x20是[空格](http://zh.wikipedia.org/zh-cn/%E7%A9%BA%E6%A0%BC)、0xA0是不换行空格、0xAD是选择性[连接号](http://zh.wikipedia.org/zh-cn/%E8%BF%9E%E6%8E%A5%E5%8F%B7)。
* 0x00-0x1F、0x7F、0x80-0x9F在此字符集中未有定义。
* Windows中的某些字体也定义了从0x80到0x9F的某些字符，但这些不是ANSI/ISO 8859-1标准的一部分。

#### ISO 8859字符集概括

**ISO 8859-1** (Latin-1) - 西欧语言

**ISO 8859-2** (Latin-2) - 中欧语言

**ISO 8859-3** (Latin-3) - 南欧语言。世界语也可用此字符集显示。

**ISO 8859-4** (Latin-4) - 北欧语言

**ISO 8859-5** (Cyrillic) - 斯拉夫语言

**ISO 8859-6** (Arabic) - 阿拉伯语

**ISO 8859-7** (Greek) - 希腊语

**ISO 8859-8** (Hebrew) - 希伯来语(视觉顺序)

**ISO 8859-8-I** - 希伯来语(逻辑顺序)

**ISO 8859-9** (Latin-5 或 Turkish) - 它把Latin-1的冰岛语字母换走，加入土耳其语字母。

**ISO 8859-10** (Latin-6 或 Nordic) - 北日耳曼语支，用来代替Latin-4。

**ISO 8859-11** (Thai) - 泰语，从泰国的 TIS620 标准字集演化而来。

**ISO 8859-13** (Latin-7 或 Baltic Rim) - 波罗的语族

**ISO 8859-14** (Latin-8 或 Celtic) - 凯尔特语族

**ISO 8859-15** (Latin-9) - 西欧语言，加入Latin-1欠缺的法语及芬兰语重音字母，以及欧元符号。

**ISO 8859-16** (Latin-10) - 东南欧语言。主要供罗马尼亚语使用，并加入欧元符号。

## 中文字符集及其编码

**GB2312**字符集是简体字集，全称为GB2312(80)字集，共包括国标简体汉字6763个。**BIG5**字符集是台湾繁体字集，共包括国标繁体汉字13053个。

**HKSCS**是中国香港使用的编码标准，字体也是繁体，但跟Big5有所不同。

**GBK**字符集是简繁字集，包括了GB字集、BIG5字集和一些符号，共包括21003个字符。

**GB18030**字符集是国家制定的一个强制性大字集标准，全称为GB18030-2000，它的推出使汉字集有了一个“大统一”的标准。

### GB2312-80

**GB 2312**或**GB 2312-80**是[中国国家标准](http://zh.wikipedia.org/zh-cn/%E4%B8%AD%E5%8D%8E%E4%BA%BA%E6%B0%91%E5%85%B1%E5%92%8C%E5%9B%BD%E5%9B%BD%E5%AE%B6%E6%A0%87%E5%87%86)[简体中文](http://zh.wikipedia.org/zh-cn/%E7%AE%80%E4%BD%93%E4%B8%AD%E6%96%87)[字符集](http://zh.wikipedia.org/zh-cn/%E5%AD%97%E7%AC%A6%E9%9B%86)，全称《**信息交换用汉字编码字符集·基本集**》，又称[**GB0**](http://zh.wikipedia.org/zh-cn/%E5%9B%BD%E5%AE%B6%E6%A0%87%E5%87%86%E4%BB%A3%E7%A0%81)，由[中国国家标准总局](http://zh.wikipedia.org/w/index.php?title=%E4%B8%AD%E5%9B%BD%E5%9B%BD%E5%AE%B6%E6%A0%87%E5%87%86%E6%80%BB%E5%B1%80&action=edit&redlink=1)发布，[1981年](http://zh.wikipedia.org/zh-cn/1981%E5%B9%B4)[5月1日](http://zh.wikipedia.org/zh-cn/5%E6%9C%881%E6%97%A5)实施。GB2312编码通行于中国大陆；[新加坡](http://zh.wikipedia.org/zh-cn/%E6%96%B0%E5%8A%A0%E5%9D%A1)等地也采用此编码。中国大陆几乎所有的中文系统和国际化的软件都支持GB 2312。

GB 2312标准共收录6763个[汉字](http://zh.wikipedia.org/zh-cn/%E6%B1%89%E5%AD%97)，其中[一级汉字](http://zh.wikipedia.org/zh-cn/%E5%B8%B8%E7%94%A8%E5%AD%97)3755个，[二级汉字](http://zh.wikipedia.org/zh-cn/%E6%AC%A1%E5%B8%B8%E7%94%A8%E5%AD%97)3008个；同时收录了包括[拉丁字母](http://zh.wikipedia.org/zh-cn/%E6%8B%89%E4%B8%81%E5%AD%97%E6%AF%8D)、[希腊字母](http://zh.wikipedia.org/zh-cn/%E5%B8%8C%E8%85%8A%E5%AD%97%E6%AF%8D)、[日文](http://zh.wikipedia.org/zh-cn/%E6%97%A5%E8%AF%AD)[平假名](http://zh.wikipedia.org/zh-cn/%E5%B9%B3%E5%81%87%E5%90%8D)及[片假名](http://zh.wikipedia.org/zh-cn/%E7%89%87%E5%81%87%E5%90%8D)字母、[俄语](http://zh.wikipedia.org/zh-cn/%E4%BF%84%E8%AF%AD)[西里尔字母](http://zh.wikipedia.org/zh-cn/%E6%96%AF%E6%8B%89%E5%A4%AB%E5%AD%97%E6%AF%8D)在内的682个字符，还包括ASCII里本来就有的数字、标点、字母，这些都重新编了两个字节长的编码，这就是常说的“**全角**”字符，而原来在127号以下的那些就叫“**半角**”字符了。

GB 2312的出现，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。对于[人名](http://zh.wikipedia.org/zh-cn/%E4%BA%BA%E5%90%8D)、[古汉语](http://zh.wikipedia.org/zh-cn/%E5%8F%A4%E6%B1%89%E8%AF%AD)等方面出现的[罕用字](http://zh.wikipedia.org/zh-cn/%E7%BD%95%E7%94%A8%E5%AD%97)，GB2312不能处理，这导致了后来[GBK](http://zh.wikipedia.org/zh-cn/GBK)及[GB18030](http://zh.wikipedia.org/zh-cn/GB_18030)汉字字符集的出现。

* **区位码**

将GB2312-80的全部字符集组成一个94×94的方阵，每一行称为一个“**区**”，编号为0l～94；每一列称为一个“**位**”，编号为0l～94。用区位图的位置来表示的汉字编码，称为**区位码**。

* 各区的分配情况如下：

**01-09区**为符号、数字区；

**16-87区**为汉字区；

**10-15区、88-94区**是有待进一步标准化的空白区。

* GB2312将收录的汉字分成两级：
* 第一级是常用汉字计3755个，置于16-55区，按汉语拼音字母/笔形顺序排列；
* 第二级汉字是次常用汉字计3008个，置于56-87区，按部首/笔画顺序排列。

如“保”字在二维代码表中处于17区第3位，区位码即为“1703”。（可用区位码输法法验证，4位数字不能通过小键盘输入）

* **国标码**

国标码是汉字信息交换的标准编码，它与区位码的关系是：先将十进制区码和位码转换为十六进制的区码和位码，这样就得了一个与国标码有一个相对位置差的代码，再将这个代码的第一个字节和第二个字节分别加上20H，就得到国标码。

**区/位码**

**国标码**

**+(20)H或+(32)D**

如：“保”字的国标码为3123Ｈ，它是经过下面的转换得到的：

1703D－>1103H->+20H－>3123H

* **机内码**

国标码是汉字信息交换的标准编码，但因其前后字节的最高位为0，与ASCII码发生冲突，如“保”字，国标码为31H和23H，而西文字符“1”和“#”的ASCII也为31H和23H，现假如内存中有两个字节为31H和23H，这到底是一个汉字，还是两个西文字符“1”，和“#”？于是就出现了二义性，显然，国标码是不可能在计算机内部直接采用的，于是，汉字的机内码采用变形国标码，其变换方法为：将国标码的每个字节都加上128，即将两个字节的最高位由0改1，其余7位不变。

如：由上面我们知道，“保”字的国标码为3123H，前字节为00110001B，后字节为00100011B，高位改1为10110001B和10100011B即为B1A3H，因此，“保”字的机内码就是B1A3H。（可用GBK内码输入法验证）

* **区位码、国标码、内码之间的关系**

**区位码**

**国标码**

**机内码**

**＋(20)Ｈ**

**＋(80)Ｈ**

**＋(A0)Ｈ**

* **GB2313字符集的编码**

GB2312规定“对任意一个图形字符都采用两个字节表示，每个字节均采用七位编码表示”，习惯上称第一个字节为“**高位字节**”，第二个字节为“**低位字节**”。为了保证于ASCII码的兼容，GB2312采用单字节来保存ASCII码表中的字符。所于GB2312采用的是**单双字节变长编码**，单字节码位兼容ASCII。

“高位字节”使用了0xA1-0xF7（把01-87区的区号加上0xA0），“低位字节”使用了0xA1-0xFE（把01-94加上0xA0）。由于一级汉字从16区起始，汉字区的“高位字节”的范围是0xB0-0xF7，“低位字节”的范围是0xA1-0xFE，占用的码位是72\*94=6768。其中有5个空位是D7FA-D7FE。所以，GB2312-80最多能表示6763个汉字。

### GBK

GB2312-80仅收汉字6763个，这大大少于现有汉字，随着时间推移及汉字文化的不断延伸推广，有些原来很少用的字，现在变成了常用字，例如：朱镕基的“镕”字，未收入GB2312-80，现在大陆的报业出刊只得使用（金+容）、（金容）、（左金右容）等来表示，形式不一而同，这使得表示、存储、输入、处理都非常不方便，对于搜索引擎等软件的构造来说也不是好消息，而且这种表示没有统一标准。从我们对人民日报98年数据的处理过程中，得出这样的经验：回填外字最困难的就是如何得到这种表示方法的集合。

为了解决这些问题，以及配合UNICODE的实施，全国信息技术化技术委员会于1995年12月1日《汉字内码扩展规范》。GBK向下与GB2312完全兼容，向上支持ISO10646国际标准，在前者向后者过渡过程中起到的承上启下的作用。

GBK共收入21886个汉字和图形符号，包括：1）GB2312中的全部汉字、非汉字符号；2）BIG5中的全部汉字；3）与ISO10646相应的国家标准GB13000中的其它CJK汉字，以上合计20902个汉字；4）其它汉字、部首、符号，共计984个。

微软公司自Windows 95简体中文版开始支持GBK代码，但目前的多数搜索引擎都不能很好地支持GBK汉字。

* GBK编码分区

**汉字区**

GBK/2：OXBOA1-F7FE, 收录GB2312汉字6763个，按原序排列；

GBK/3：OX8140-AOFE，收录CJK汉字6080个（CJK就是中日韩的意思。Unicode为了节省码位，将中日韩三国语言中的文字统一编码）；

GBK/4：OXAA40-FEAO，收录CJK汉字和增补的汉字8160个。

**图形符号区**

GBK/1：OXA1A1-A9FE，除GB2312的符号外，还增补了其它符号

GBK/5：OXA840-A9AO，扩除非汉字区。

**用户自定义区**

即GBK区域中的空白区，用户可以自己定义字符。

* GBK编码

GBK采用**单双字节变长编码**方式编码。字符有一字节和双字节编码，00–7F范围内是一位，和[ASCII](http://zh.wikipedia.org/zh-cn/ASCII)保持一致，此范围内严格上说有96个文字和32个控制符号。之后的双字节中，前一字节是双字节的第一位。总体上说第一字节的范围是81–FE（也就是不含80和FF），第二字节的一部分领域在40–FE，其他领域在80–FE。具体来说，定义的是下列字节：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| GBK的编码范围 | | | | |
| **范围** | **第1字节** | **第2字节** | **编码数** | **字数** |
| GBK/1 | A1–A9 | A1–FE | 846 | 717 |
| GBK/2 | B0–F7 | A1–FE | 6,768 | 6,763 |
| GBK/3 | 81–A0 | 40–FE (7F除外) | 6,080 | 6,080 |
| GBK/4 | AA–FE | 40–A0 (7F除外) | 8,160 | 8,160 |
| GBK/5 | A8–A9 | 40–A0 (7F除外) | 192 | 166 |
| 用户定义 | AA–AF | A1–FE | 564 | 0 |
| 用户定义 | F8–FE | A1–FE | 658 | 0 |
| 用户定义 | A1–A7 | 40–A0 (7F除外) | 672 | 0 |
| **合计:** |  |  | **23,940** | **21,886** |

### GB18030

GB18030是最新的汉字编码字符集国家标准，向下兼容GBK和GB2312标准。GB18030编码是**一二四字节变长编码**。

一字节部分从0x0~0x7F与ASCII编码兼容。

二字节部分，首字节从0x81~0xFE，尾字节从0x40~0x7E以及0x80~0xFE与GBK标准基本兼容。

四字节部分，第一字节从0x81~0xFE，第二字节从0x30~0x39，第三和第四字节的范围和前两个字节分别相同。四字节部分覆盖了从0x0080开始，除去二字节部分已经覆盖的所有Unicode 3.1码位。也就是说，GB18030编码在码位空间上做到了与Unicode标准一一对应，这一点与UTF-8编码类似。

目前最新的glibc2.2.x系列已经全面支持了GB18030 Locale和GB18030与UCS-4之间的编码转换，也就是说在系统层上Linux已经可以支持GB18030标准了。下面问题的关键就是怎样让XFree86窗口系统也支持 GB18030标准。

### BIG5

BIG5是通行于台湾、香港地区的一个繁体字编码方案。虽然存在一些瑕疵，但广泛应用于电脑行业，尤其是互联网中，从而成为一种事实上的行业标准。

1983年10月，台湾国家科学委员会、教育部国语推行委员会、中央标准局、行政院共同制定了《通用汉字标准交换码》，后经修订于1992年5月公布，更名为《中文标准交换码》，BIG5是台湾资讯工业策进会根据以上标准制定的编码方案。

BIG5码采用**单双字节变长编码**方案，其中第一个字节的值在OXAO-OXFE之间，第二个字节在OX40-OX7E和OXA1-OXFE之间。

**BIG5收录13461个汉字和符号**，包括：

**符号**408个，编码位置A140-A3BE

**常用字**5401个，编码位置A440-C67E，包括台湾教育部颁布的《常用国字标准字体表》的全部汉字4808个，台湾教科书常用字587个，异体字6个。

**次常用字**7652个，编码位置C940-F9D5，包括台湾教育部颁布的《次常用国字标准字体表》的全部汉字6341个，《罕用国字标准字体表》中使用频率较高的字1311个。

## UCS和Unicode

历史上，有两个独立的创立单一字符集的尝试：一个是国际标准化组织(ISO)的**ISO 10646项目**，另一个是由多语言软件制造商(一开始大多是美国的)组成的协会组织的**Unicode项目**。ISO 10646制定的字符集称为**UCS**（Universal Character Set），Unicode项目制定的字符集叫**Unicode**(**统一码、万国码、单一码、标准万国码**)。

[1991年](http://zh.wikipedia.org/zh-cn/1991%E5%B9%B4)前后，两个项目的参与者都认识到，世界不需要两个不兼容的字符集。于是，它们开始合并双方的工作成果，并为创立一个单一编码表而协同工作。从Unicode 2.0开始，Unicode采用了与ISO 10646-1相同的字库和字码；**ISO也承诺，ISO 10646将不会替超出U+10FFFF的UCS-4编码赋值，以使得两者保持一致。**

两个项目仍都存在，并独立地公布各自的标准。但统一码联盟Unicode和ISO/IEC JTC1/SC2都同意保持两者标准的码表兼容，并紧密地共同调整任何未来的扩展。在发布的时候，Unicode一般都会采用有关字码最常见的字型，但ISO 10646一般都尽可能采用[Century字型](http://zh.wikipedia.org/w/index.php?title=Century%E5%AD%97%E5%9E%8B&action=edit&redlink=1)。

### UCS介绍

#### 概要

**通用字符集**（Universal Character Set，**UCS**）是由[ISO](http://zh.wikipedia.org/zh-cn/%E5%9C%8B%E9%9A%9B%E6%A8%99%E6%BA%96%E5%8C%96%E7%B5%84%E7%B9%94)制定的ISO 10646（或称ISO/IEC 10646）标准所定义的字符编码方式，采用4[字节](http://zh.wikipedia.org/zh-cn/%E5%AD%97%E8%8A%82)编码。

通用字符集又称Universal Multiple-Octet Coded Character Set，中国大陆译为**通用多八位编码字符集**，台湾译为**广用多八位元编码字元集**。

通用字符集是所有包括了其他字符集。它保证了与其他字符集的双向兼容，即，如果你将任何文本字符串翻译到UCS格式，然后再翻译回原编码，你不会丢失任何信息。

UCS包含了已知语言的所有字符。除了拉丁语、希腊语、斯拉夫语、希伯来语、阿拉伯语、亚美尼亚语、格鲁吉亚语，还包括中文、日文、韩文这样的象形文字，UCS还包括大量的图形、印刷、数学、科学符号。

UCS不仅给每个字符分配一个代码，而且赋予了一个正式的名字。表示一个UCS或Unicode值的十六进制数，通常在前面加上"U+"，就像U+0041代表字符"拉丁大写字母A"，**UCS字符U+0000到U+007F与US-ASCII(ISO-646)是一致的，U+0000到U+00FF与ISO-8859-1(Latin-1)也是一致的**，从U+E000到U+F8FF，以及BMP以外的大范围的编码是为私用保留的。

#### UCS使用划分

**ISO 10646-1标准**

第一次发表于1993年，分配了前65534（2^16）个码位（0x0000到0xFFFD）。这个UCS的16位子集称为**基本多语言面**（(Basic Multilingual Plane，**BMP**）。ISO-10646-1定义了字符集与BMP内容的架构。新的字符仍源源不断地加入到BMP中，但已经存在的字符是稳定且不会再改变的。

**ISO 10646-2标准**

正在准备中，定义BMP外的字符编码的第二部分。作为被编码在16位BMP以外的字符都属于非常特殊的字符(比如象形文字)，且只有专家在历史和科学领域里才会用到它们。按当前的计划，将来也许再也不会有字符被分配到从0x000000到0x10FFFF这个覆盖了超过100万个潜在的未来字符的21位的编码空间以外去了，覆盖了超过100万个潜在的末来字符。

#### UCS组合字符

UCS里有些编码点分配给了组合字符，它们类似于打字机上的无间隔重音键。单个的组合字符不是一个完整的字符。它是一个类似于重音符或其他指示标记，加在前一个字符后面。因而，重音符可以加在任何字符后面。那些最重要的被加重的字符，就象普通语言的正字法(orthographies of common languages)里用到的那种，在UCS里都有自己的位置，以确保同老的字符集的向后兼容性。既有自己的编码位置，又可以表示为一个普通字符跟随一个组合字符的被加重字符，被称为**预作字符**(precomposed characters)。UCS里的预作字符是为了同没有预作字符的旧编码，比如ISO 8859，保持向后兼容性而设的。组合字符机制允许在任何字符后加上重音符或其他指示标记，这在科学符号中特别有用，比如数学方程式和国际音标字母，可能会需要在一个基本字符后组合上一个或多个指示标记。

组合字符跟随着被修饰的字符。比如，德语中的元音变音字符(“拉丁大写字母A加上分音符”)，既可以表示为UCS码U+00C4的预作字符，也可以表示成一个普通“拉丁大写字母A”跟着一个“组合分音符”：U+0041 U+0308这样的组合。当需要堆叠多个重音符，或在一个基本字符的上面和下面都要加上组合标记时，可以使用多个组合字符。比如在泰国文中，一个基本字符最多可加上两个组合字符。

#### UCS的实现级别

并不是所有的系统都需要支持像[组合字符](http://zh.wikipedia.org/w/index.php?title=%E7%BB%84%E5%90%88%E5%AD%97%E7%AC%A6&action=edit&redlink=1)这样的先进机制。因此ISO 10646指定了如下三种[实现级别](http://zh.wikipedia.org/w/index.php?title=%E5%AE%9E%E7%8E%B0%E7%BA%A7%E5%88%AB&action=edit&redlink=1)：

* 级别1：不支持组合字符和韩语Hangul Jamo字符(一种特别的，更加复杂的韩国文的编码, 使用两个或三个子字符来编码一个韩文音节)；
* 级别2：类似于级别1，但在某些文字中，允许一列固定的组合字符(例如, 希伯来文，阿拉伯文，Devangari，孟加拉语，果鲁穆奇语，Gujarati，Oriya，泰米尔语，Telugo，印.埃纳德，Malayalam，泰国语和老挝语)。如果没有这最起码的几个组合字符，UCS就不能完整地表达这些语言；
* 级别3：支持所有的UCS字符，例如数学家可以在任意一个字符上加上一个tilde(颚化符号，西班牙语字母上面的～)或一个箭头(或两者都加)。

#### UCS编码方式

UCS字符集有两种编码形式：UCS-2（双字节）和UCS4（4字节）。

**UCS-2编码：**

1）定义：完全和UNICODE字符集一致的双字节编码，UCS-2有2^16个码位

2）支持的平台：WindowsAPI，UNIX接口，ANSI标准接口自己开发

**UCS-4编码：**

1）定义：完全支持ISO-10646规范字符数，UCS-4就是用4个字符（实际上只用了31位，最高位必须为0）编码，UCS-4有2^31个码位。

2）根据最高位为0的最高字节分成2^7=128个group，每个group根据次高字节分成256个plane，每个plane根据第3个字节分为256个row，每个row根据第4个字节分为256个cell。

3）支持的平台：目前还没有支持的操作系统，第三方软件产口接口自己开发

4）与BMP的关系：group0、plane0被称为BMP，或者说UCS-4中，高两个字节为0的码位被称作BMP

5）和UCS-2的关系：将UCS-4的BMP去掉前面两个零字节就得到了UCS-2。在UCS-2的两个字节前加上两个零字节就得到了UCS-4的BMP。

### Unicode介绍

#### 概要

**Unicode**（统一码、万国码、单一码、标准万国码）是一种在计算机上使用的字符编码。它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。1990年开始研发，1994年正式公布。随着计算机工作能力的增强，Unicode也在面世以来的十多年里得到普及。

Unicode是基于通用[字符集](http://baike.baidu.com/view/51987.htm)（Universal Character Set）的标准来发展，并且同时也以书本的形式（The Unicode Standard，目前第五版由Addison-Wesley Professional出版，ISBN-10:0321480910）对外发表。

2006年7[月](http://baike.baidu.com/view/75273.htm)的最新版本的Unicode是5.0版本。2005年[3月31日](http://baike.baidu.com/view/477558.htm)推出的Unicode 4.1.0。另外，5.0Beta于2005年12月12日推出，**5.2.0**版本（unicode standard）于2009年10月1日正式推出，是最新的版本。

#### Unicode发布的版本

Unicode截至目前为止历次的版次与发布时间如下：

* Unicode 1.0：1991年10月
* Unicode 1.0.1：1992年6月
* Unicode 1.1：1993年6月
* Unicode 2.0：1997年7月
* Unicode 2.1：1998年5月
* Unicode 2.1.2：1998年5月
* **Unicode 3.0**：1999年9月；涵盖了来自ISO 10646-1的十六位通用字符集（UCS）基本多文种平面（Basic Multilingual Plane）
* **Unicode 3.1**：2001年3月；新增从ISO 10646-2定义的辅助平面（Supplementary Planes）
* Unicode 3.2：2002年3月
* Unicode 4.0：2003年4月
* Unicode 4.0.1：2004年3月
* Unicode 4.1：2005年3月
* Unicode 5.0：2006年7月
* Unicode 5.1：2008年4月
* Unicode 5.2：2009年10月

#### 编码方式

Unicode的编码方式与[ISO 10646](http://zh.wikipedia.org/wiki/ISO_10646)的[通用字符集](http://zh.wikipedia.org/wiki/%E9%80%9A%E7%94%A8%E5%AD%97%E7%AC%A6%E9%9B%86)（Universal Character Set，UCS）概念相对应，目前实际应用的Unicode版本对应于[UCS-2](http://zh.wikipedia.org/wiki/UCS-2)，使用16[位](http://zh.wikipedia.org/wiki/%E4%BD%8D)的编码空间。也就是每个字符占用2个[字节](http://zh.wikipedia.org/wiki/%E5%AD%97%E8%8A%82)。这样理论上一共最多可以表示2^16即65536个字符。基本满足各种语言的使用。实际上当前版本的Unicode尚未填充满这16位编码，保留了大量空间作为特殊使用或将来扩展。

上述16位Unicode字符构成[**基本多文种平面**](http://zh.wikipedia.org/wiki/%E5%9F%BA%E6%9C%AC%E5%A4%9A%E6%96%87%E7%A8%AE%E5%B9%B3%E9%9D%A2)（Basic Multilingual Plane，简称**BMP**）。最新（但未实际广泛使用）的Unicode版本定义了**16个**[**辅助平面**](http://zh.wikipedia.org/wiki/%E8%BE%85%E5%8A%A9%E5%B9%B3%E9%9D%A2)，两者合起来至少需要占据21位的编码空间，比3字节略少。但事实上辅助平面字符仍然占用4字节编码空间，与[UCS-4](http://zh.wikipedia.org/wiki/UCS-4)保持一致。未来版本会扩充到ISO 10646-1实现级别3，即涵盖UCS-4的所有字符。UCS-4是一个更大的尚未填充完全的31位字符集，加上恒为0的首位，共需占据32位，即4字节。理论上最多能表示231个字符，完全可以涵盖一切语言所用的符号。

**BMP字符的Unicode编码表示为U+hhhh**，其中每个h代表一个[十六进制](http://zh.wikipedia.org/wiki/%E5%8D%81%E5%85%AD%E8%BF%9B%E5%88%B6)数位。与UCS-2编码完全相同。对应的4字节UCS-4编码后两个字节一致，前两个字节的所有位均为0。

在表达一个Unicode的字元时，通常会用[U+]然后紧接一组十六进制的数字来表示这一字元。在基本多语言平面（BMP）里的所以字元，保要使用四位十六进制数（例如U+4AE0，共支持六万多个字符）来表示，但在BMP以外的字元则需要使用五位或六位十六进制制数了。如在**Unicode3.0**里使用「**U-**」后接8位数，而「**U+**」则必须后接4位数。

#### 辅助平面

原有的Unicode空间称为**基本平面**或[**基本多文种平面**](http://zh.wikipedia.org/zh-cn/%E5%9F%BA%E6%9C%AC%E5%A4%9A%E6%96%87%E7%A8%AE%E5%B9%B3%E9%9D%A2)（Basic Multilingual Plane，缩写**BMP**），俗称**第0平面**（**Plane 0**）。为鉴于[Unicode](http://zh.wikipedia.org/zh-cn/Unicode)原有的16[位](http://zh.wikipedia.org/zh-cn/%E4%BD%8D%E5%85%83)空间不足以应用，于是从Unicode 3.1版本开始，设立了16个扩展字码空间，称为辅助平面，使Unicode的可使用空间由6万多字增至约100万字。辅助平面字符要用上4[字节](http://zh.wikipedia.org/zh-cn/%E5%AD%97%E8%8A%82)来存储。

第一辅助平面又称**多文种补充平面**（Supplementary Multilingual Plane，缩写**SMP**，或简称**Plane 1**），摆放拼音文字（主要为现时已不再使用的文字）和[音符](http://zh.wikipedia.org/zh-cn/%E9%9F%B3%E7%AC%A6)等符号。范围在U+10000～U+1FFFD。

第二辅助平面又称为**表意文字补充平面**（Supplementary Ideographic Plane，缩写**SIP**，或简称**Plane 2**），整个范围在U+20000～U+2FFFD。整个平面配置的都是一些罕用的[汉字](http://zh.wikipedia.org/zh-cn/%E6%B1%89%E5%AD%97)或地区的方言用字，如[粤语](http://zh.wikipedia.org/zh-cn/%E7%B2%B5%E8%AA%9E)用字及[越南语](http://zh.wikipedia.org/zh-cn/%E8%B6%8A%E5%8D%97%E8%AA%9E)的[字喃](http://zh.wikipedia.org/zh-cn/%E5%AD%97%E5%96%83)。现时摆放了“[中日韩统一表意文字](http://zh.wikipedia.org/zh-cn/%E4%B8%AD%E6%97%A5%E9%9F%93%E7%B5%B1%E4%B8%80%E8%A1%A8%E6%84%8F%E6%96%87%E5%AD%97)扩展B区”（4万3253个[汉字](http://zh.wikipedia.org/zh-cn/%E6%B1%89%E5%AD%97)）、“中日韩统一表意文字扩展C区”（4149个[汉字](http://zh.wikipedia.org/zh-cn/%E6%B1%89%E5%AD%97)）以及中日韩兼容表意文字增补（CJK Compatibility Ideographs Supplement）。预计“中日韩统一表意文字扩展D区”和“中日韩统一表意文字扩展E区”也会使用此平面。

已分配和计划分配的编码区段为：

U+20000-U+2A6DF：中日韩统一表意文字扩展B区

U+2A700-U+2B73F：中日韩统一表意文字扩展C区

U+2B740-U+2B81F：中日韩统一表意文字扩展D区，还在草拟阶段。

U+2B820-U+2????：中日韩统一表意文字扩展E区，还在草拟阶段。

U+2F800-U+2FA1F：中日韩兼容表意文字增补

第三至十三辅助平面尚未使用，但打算用来摆放[甲骨文](http://zh.wikipedia.org/zh-cn/%E7%94%B2%E9%AA%A8%E6%96%87)、[金文](http://zh.wikipedia.org/zh-cn/%E9%87%91%E6%96%87)、[小篆](http://zh.wikipedia.org/zh-cn/%E5%B0%8F%E7%AF%86)、[中国战国时期](http://zh.wikipedia.org/zh-cn/%E6%88%B0%E5%9C%8B)文字等。 计划分配的编码区段为：

U+30000-U+317FF：甲骨文

U+32000-U+32FFF：金文

U+34000-U+368FF：小篆

第四至第十三辅助平面并未计划使用。

第十四辅助平面又称**特别用途补充平面**（Supplementary Special-purpose Plane，简称**SSP**），摆放“语言编码标签”和“字形变换选取器”，它们都是[控制字符](http://zh.wikipedia.org/zh-cn/%E6%8E%A7%E5%88%B6%E5%AD%97%E7%AC%A6)。范围在U+E0000～U+E01FF。编码表详见[E0000-E0FFF](http://zh.wikibooks.org/wiki/Unicode/E0000-E0FFF)。

第十五至十六辅助平面都是私人使用区。它们的范围是U+F0000～U+FFFFD 及U+100000～U+10FFFD。

### UCS与Unicode的异同

统一码联盟公布的Unicode标准包含了ISO/IEC 10646-1实现级别3的[**基本多文种平面**](http://zh.wikipedia.org/zh-cn/%E5%9F%BA%E6%9C%AC%E5%A4%9A%E6%96%87%E7%A8%AE%E5%B9%B3%E9%9D%A2)。在两个标准里，所有的字符都在相同的位置并且有相同的名字。

ISO/IEC 10646标准，就像[ISO/IEC 8859](http://zh.wikipedia.org/zh-cn/ISO/IEC_8859)标准一样，只不过是一个简单的字符集表。它定义了一些编码的别名，指定了一些与标准有关的术语，并包括了规范说明，指定了怎样使用UCS连接其他ISO标准的实现，比如[ISO/IEC 6429](http://zh.wikipedia.org/w/index.php?title=ISO/IEC_6429&action=edit&redlink=1)和[ISO/IEC 2022](http://zh.wikipedia.org/zh-cn/ISO/IEC_2022)。还有一些与ISO紧密相关的，比如[ISO/IEC 14651](http://zh.wikipedia.org/w/index.php?title=ISO/IEC_14651&action=edit&redlink=1)是关于UCS字符串排序的。

Unicode标准，额外定义了许多与字符有关的语义符号学。Unicode详细说明了绘制某些语言（如阿拉伯语）表达形式的算法，处理双向文字（比如拉丁文和希伯来文的混合文字）的算法，排序与字符串比较所需的算法，等等。

**由于Unicode这一名字比较好记，因而它使用更为广泛。**不过一般认为，用于打印ISO/IEC 10646-1标准的字体在某些方面的质量，要高于Unicode 2.0。

两者部分样例字形有显著的区别。ISO/IEC 10646-1标准同样使用四种不同的风格变体来显示表意文字如中文、日文、韩文（即CJK），但Unicode 2.0的表里只有中文的变体。甚至存在“Unicode对日本用户来说不可接受”的不实传说。

在Unicode和ISO/IEC 10646两个标准中，虽然两者部分样例字形有显著的区别，所有的字符都在相同的位置并且有相同的名字。ISO/IEC 10646-1标准使用四种不同的风格来显示中(繁、简)、日、韩文字，但Unicode的表中，只用简体中文风格来显示，像“Unicode对日本用户来说不可接受”的误解，便由此产生。其实，上文已说明，ISO/IEC 10646或Unicode都只是编码标准，并不处理字形风格标准之问题。字形的风格可以透过改变显示字体来解决。

### UTF（Unicode/UCS Translation Format）

#### UTF简介

Unicode的实现方式不同于编码方式。一个字符的Unicode编码是确定的。但是在实际传输过程中，由于不同[系统平台](http://zh.wikipedia.org/wiki/%E7%B3%BB%E7%BB%9F%E5%B9%B3%E5%8F%B0)的设计不一定一致，以及出于节省空间的目的，对 Unicode 编码的实现方式有所不同。Unicode 的实现方式称为**Unicode转换格式**（Unicode Translation Format，简称为**UTF**）。

例如，如果一个仅包含基本7位[ASCII](http://zh.wikipedia.org/wiki/ASCII)字符的Unicode文件，如果每个字符都使用2字节的原Unicode编码传输，其第一字节的8位始终为0。这就造成了比较大的浪费。对于这种情况，可以使用[**UTF-8**](http://zh.wikipedia.org/wiki/UTF-8)编码，这是一种变长编码，它将基本7位ASCII字符仍用7位编码表示，占用一个字节（首位补0）。而遇到与其他Unicode字符混合的情况，将按一定算法转换，每个字符使用1-3个字节编码，并利用首位为0或1进行识别。这样对以7位ASCII字符为主的西文文档就大大节省了编码长度（具体方案参见[UTF-8](http://zh.wikipedia.org/wiki/UTF-8)）。类似的，对未来会出现的需要4个字节的辅助平面字符和其他UCS-4扩充字符，2字节编码的[**UTF-16**](http://zh.wikipedia.org/wiki/UTF-16)也需要通过一定的算法进行转换。

再如，如果直接使用与Unicode编码一致（仅限于BMP字符）的[**UTF-16**](http://zh.wikipedia.org/wiki/UTF-16)编码，由于每个字符占用了两个字节，在[Macintosh](http://zh.wikipedia.org/wiki/Macintosh)机和[PC](http://zh.wikipedia.org/wiki/PC)机上，对字节顺序的理解是不一致的。这时同一字节流可能会被解释为不同内容，如编码为U+594E的字符“奎”同编码为U+4E59的“乙”就可能发生混淆。于是在UTF-16编码实现方式中使用了[大尾序](http://zh.wikipedia.org/wiki/Big-endian)（**big-endian**）、[小尾序](http://zh.wikipedia.org/wiki/Little-endian)（**little-endian**）的概念，以及[**BOM**](http://zh.wikipedia.org/wiki/BOM)（Byte Order Mark）解决方案。（具体方案参见[UTF-16](http://zh.wikipedia.org/wiki/UTF-16)）

#### UTF-8

**UTF-8**（8[位](http://zh.wikipedia.org/zh-cn/%E4%BD%8D%E5%85%83)[Universal Character Set](http://zh.wikipedia.org/zh-cn/UCS)／Unicode Transformation Format）是一种针对[Unicode](http://zh.wikipedia.org/zh-cn/Unicode)的**可变长度**[**字符编码**](http://zh.wikipedia.org/zh-cn/%E5%AD%97%E7%AC%A6%E7%BC%96%E7%A0%81)，也是一种[**前缀码**](http://zh.wikipedia.org/w/index.php?title=%E5%89%8D%E7%BC%80%E7%A0%81&action=edit&redlink=1)。它可以用来表示[Unicode](http://zh.wikipedia.org/zh-cn/Unicode)标准中的任何字符，且其编码中的第一个[字节](http://zh.wikipedia.org/zh-cn/%E5%AD%97%E8%8A%82)仍与[ASCII](http://zh.wikipedia.org/zh-cn/ASCII)兼容，这使得原来处理ASCII字符的[软件](http://zh.wikipedia.org/zh-cn/%E8%BB%9F%E9%AB%94)无须或只须做少部份修改，即可继续使用。因此，它逐渐成为[电子邮件](http://zh.wikipedia.org/zh-cn/%E9%9B%BB%E5%AD%90%E9%83%B5%E4%BB%B6)、[网页](http://zh.wikipedia.org/zh-cn/%E7%B6%B2%E9%A0%81)及其他[存储](http://zh.wikipedia.org/zh-cn/%E5%84%B2%E5%AD%98%E8%A3%9D%E7%BD%AE)或传送文字的应用中，优先采用的编码。

* UTF-8的编码范围

**UTF-8使用一至四个**[**字节**](http://zh.wikipedia.org/zh-cn/%E5%AD%97%E8%8A%82)**为每个字符编码：**

128个**US-ASCII**字符只需一个字节编码（Unicode范围由U+0000至U+007F）。

带有[附加符号](http://zh.wikipedia.org/zh-cn/%E9%99%84%E5%8A%A0%E7%AC%A6%E5%8F%B7)的[拉丁文](http://zh.wikipedia.org/zh-cn/%E6%8B%89%E4%B8%81%E6%96%87)、[希腊文](http://zh.wikipedia.org/zh-cn/%E5%B8%8C%E8%87%98%E6%96%87)、[西里尔字母](http://zh.wikipedia.org/zh-cn/%E8%A5%BF%E9%87%8C%E7%88%BE%E5%AD%97%E6%AF%8D)、[亚美尼亚语](http://zh.wikipedia.org/zh-cn/%E4%BA%9E%E7%BE%8E%E5%B0%BC%E4%BA%9E%E8%AA%9E)、[希伯来文](http://zh.wikipedia.org/zh-cn/%E5%B8%8C%E4%BC%AF%E4%BE%86%E6%96%87)、[阿拉伯文](http://zh.wikipedia.org/zh-cn/%E9%98%BF%E6%8B%89%E4%BC%AF%E6%96%87)、[叙利亚文](http://zh.wikipedia.org/w/index.php?title=%E6%95%98%E5%88%A9%E4%BA%9E%E6%96%87&action=edit&redlink=1)及[它拿字母](http://zh.wikipedia.org/zh-cn/%E5%AE%83%E6%8B%BF%E5%AD%97%E6%AF%8D)则需要二个字节编码（Unicode范围由U+0080至U+07FF）。

**其他**[**基本多文种平面**](http://zh.wikipedia.org/zh-cn/%E5%9F%BA%E6%9C%AC%E5%A4%9A%E6%96%87%E7%A8%AE%E5%B9%B3%E9%9D%A2)（BMP）中的字符（这包含了大部分常用字）使用三个字节编码。

其他极少使用的Unicode[**辅助平面**](http://zh.wikipedia.org/zh-cn/%E8%BC%94%E5%8A%A9%E5%B9%B3%E9%9D%A2)的字符使用四字节编码。

[**互联网工程工作小组**](http://zh.wikipedia.org/zh-cn/%E7%B6%B2%E9%9A%9B%E7%B6%B2%E8%B7%AF%E5%B7%A5%E7%A8%8B%E5%B7%A5%E4%BD%9C%E5%B0%8F%E7%B5%84)（IETF）要求所有[互联网](http://zh.wikipedia.org/zh-cn/%E7%B6%B2%E9%9A%9B%E7%B6%B2%E8%B7%AF)[协议](http://zh.wikipedia.org/zh-cn/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE)都必须支持UTF-8编码。

[互联网邮件联盟](http://zh.wikipedia.org/w/index.php?title=%E4%BA%92%E8%81%AF%E7%B6%B2%E9%83%B5%E4%BB%B6%E8%81%AF%E7%9B%9F&action=edit&redlink=1)（IMC）建议所有电子邮件软件都支持UTF-8编码。

* **UTF-8的编码方式**

UTF-8就是以8位为单元对UCS进行编码，而UTF-8不使用大尾序和小尾序的形式，每个使用UTF-8存储的字符，除了第一个字节外，其余字节的头两个位都是以"10"开始，使文字处理器能够较快地找出每个字符的开始位置。

但为了与以前的ASCII码兼容（ASCII为一个字节），因此UTF-8选择了使用可变长度[字节](http://zh.wikipedia.org/zh-cn/%E5%AD%97%E8%8A%82)来存储Unicode：

|  |  |
| --- | --- |
| **Unicode和UTF-8之间的转换关系表** | |
| **UCS-4编码** | **UTF-8字节流** |
| U+00000000 – U+0000007F | 0xxxxxxx |
| U+00000080 – U+000007FF | 110xxxxx 10xxxxxx |
| U+00000800 – U+0000FFFF | 1110xxxx 10xxxxxx 10xxxxxx |
| U+00010000 – U+001FFFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx |
| U+00200000 – U+03FFFFFF | 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx |
| U+04000000 – U+7FFFFFFF | 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx |

* **UTF-8的特性**

|  |  |
| --- | --- |
| UTF-8图表说明 | |
| Smallest code point | 0000 |
| Largest code point | 10FFFF |
| Code unit size | 8 bits |
| Byte order | N/A |
| Minimal bytes/character | 1 |
| Maximal bytes/character | 4 |

* UCS字符U+0000到U+007F (ASCII)被编码为字节0x00到0x7F（ASCII兼容），这也意味着只包含7位ASCII字符的文件在ASCII和UTF-8两种编码方式下是一样的；
* 所有大于U+007F的UCS字符被编码为一个多个字节的串，每个字节都有标记位集。因此，ASCII字节(0x00-0x7F)不可能作为任何其他字符的一部分；
* 表示非ASCII字符的多字节串的第一个字节总是在0xC0到0xFD的范围里，并指出这个字符包含多少个字节。多字节串的其余字节都在0x80到0xBF范围里，这使得重新同步非常容易，并使编码无国界，且很少受丢失字节的影响；
* 可以编入所有可能的2^31个UCS代码；
* UTF-8编码字符理论上可以最多到6个字节长，然而16位BMP字符最多只用到3字节长；
* Big Endian UCS-4字节串的排列顺序是预定的；
* 字节0xFE和0xFF在UTF-8编码中从未用到，同时，UTF-8以字节为编码单元，它的字节顺序在所有系统中都是一样的，没有字节序的问题，也因此它实际上并不需要[**BOM**](http://zh.wikipedia.org/zh-cn/%E4%BD%8D%E5%85%83%E7%B5%84%E9%A0%86%E5%BA%8F%E8%A8%98%E8%99%9F)；在UTF-8文件的开始，很多时都放置一个U+FEFF字符（UTF-8以EF,BB,BF代表），以显示这个文字文件是以UTF-8编码；
* 与UTF-16或其他Unicode编码相比，对于不支持Unicode和XML的系统，UTF-8更不容易造成问题。

#### UTF-16

在基本多语言平面内定义的符号（Basic Multilingual Plane, BMP），或称第零平面（Plane 0）），使用2个字节表示，在此之外的字符（其他平面内的字符），则使用4个字节表示。由于第零平面内，从0XD8到0XDF之间的区段是没有使用的，因此可以利用0XD8-0XDF之间的值来对辅助平面的字符进行编码。

* **编码方法**
  1. 如果字符编码U小于0x10000，也就是十进制的0到65535之内，则直接使用两字节表示；
  2. 如果字符编码U大于0x10000，由于UNICODE编码范围最大为0x10FFFF，从0x10000到0x10FFFF之间共有0x100000（0-FFFFF）个编码，也就是需要20个bit就可以标示这些编码。用U'表示从0-0xFFFFF之间的值，将其前10bit作为低位和16 bit的数值0xD800进行逻辑or操作，将后10bit作为低位和0xDC00做逻辑or操作，这样组成的4个byte就构成了U的编码。
* **范例: UTF-16 编码程序**

假设要将U+64321(16进位)转成UTF-16编码。因为它超过U+FFFF，所以他必须编译成32位元(4个byte)的格式，如下所示：

**第一步：获取相对编码**

V = 0x64321（16进制的编码）

Vx = V - 0x10000（获得编码的相对位置）

= 0x54321

= 0101 0100 0011 0010 0001

**第二步：计算高位部分与低位部分**

Vh = 01 0101 0000 // Vx的高位部份的10bits

Vl = 11 0010 0001 // Vx 的低位部份的10bits

**第三步：计算高位内码**

w1 = 0xD800 // 结果的前16位元初始值

w2 = 0xDC00 // 结果的后16位元初始值

w1 = w1 | Vh

= 1101 1000 0000 0000

01 0101 0000

= 1101 1001 0101 0000

= 0xD950（注：该值的高位取值范围是D8-DB，而00XD8到0XDF未分配）

**第四步：计算低位内码**

w2 = w2 | Vl

= 1101 1100 0000 0000

| 11 0010 0001

= 1101 1111 0010 0001

= 0xDF21(注：该值的高位取值范围是DC-DF，而00XD8到0XDF未分配)

所以这个字 U+64321 最后正确的 UTF-16 编码应该是：**0xD950 0xDF21**（注：通过第1个2字节的高位D9可以判断这是一个Unicode码大于FFFF的字符）

因为这个字超过 U+FFFF 所以无法用 UCS-2 的格式编码

|  |  |  |  |
| --- | --- | --- | --- |
| **16进制编码范围** | **UTF-16表示方法（二进制）** | **10进制码范围** | **字节数量** |
| 0000 0000---0000 FFFF | xxxxxxxx xxxxxxxx | 0-65535 | 2 |
| 0001 0000---0010 FFFF | 110110yyyyyyyyyy 110111xxxxxxxxxx | 65536-1114111 | 4 |

UTF-16比起[UTF-8](http://zh.wikipedia.org/zh-cn/UTF-8)，好处在于大部分字符都以固定长度的字节（2字节）储存，但UTF-16却无法相容于[ASCII](http://zh.wikipedia.org/zh-cn/ASCII)编码。

* **UTF-8与UTF-16字节序问题**

UTF-8以字节为编码单元，没有字节序的问题。UTF-16以两个字节为编码单元，在解释一个UTF-16文本前，首先要弄清楚每个编码单元的字节序。例如收到一个“奎”的Unicode编码是594E，“乙”的Unicode编码是4E59。如果我们收到UTF-16字节流“594E”，那么这是“奎”还是“乙”？

Unicode规范中推荐的标记字节顺序的方法是BOM。BOM不是“Bill Of Material”的BOM表，而是Byte Order Mark。BOM是一个有点小聪明的想法：

在UCS编码中有一个叫做"**ZERO WIDTH NO-BREAK SPACE**"的字符，它的编码是FEFF。而FFFE在UCS中是不存在的字符，所以不应该出现在实际传输中。UCS规范建议我们在传输字节流前，先传输字符“ZERO WIDTH NO-BREAK SPACE”。

这样如果接收者收到FEFF，就表明这个字节流是Big-Endian的；如果收到FFFE，就表明这个字节流是Little-Endian的。因此字符“ZERO WIDTH NO-BREAK SPACE”又被称作**BOM**。

UTF-8不需要BOM来表明字节顺序，但可以用BOM来表明编码方式。字符"ZERO WIDTH NO-BREAK SPACE"的UTF-8编码是EF BB BF（读者可以用我们前面介绍的编码方法验证一下）。所以如果接收者收到以EF BB BF开头的字节流，就知道这是UTF-8编码了。

Windows就是使用BOM来标记文本文件的编码方式的。

#### UTF-32

**UTF-32**(或**UCS-4**)是一种将[Unicode](http://zh.wikipedia.org/zh-cn/Unicode)字符编码的协定，对每一个Unicode码位使用恰好32[位元](http://zh.wikipedia.org/zh-cn/%E4%BD%8D%E5%85%83)。

因为UTF-32对每个字符都使用4[字节](http://zh.wikipedia.org/zh-cn/%E5%AD%97%E8%8A%82)，就空间而言，是非常没有效率的。特别地，非[基本多文种平面](http://zh.wikipedia.org/zh-cn/%E5%9F%BA%E6%9C%AC%E5%A4%9A%E6%96%87%E7%A8%AE%E5%B9%B3%E9%9D%A2)的字符在大部分文件中通常很罕见，以致于它们通常被认为不存在占用空间大小的讨论，使得UTF-32通常会是其它编码的2到4倍。

虽然每一个码位使用固定长定的字节看似方便，它并不如其它Unicode编码使用得广泛。而且，现阶段还没有操作系统直接支持UTF-32。

* **历史**

原本[ISO 10646](http://zh.wikipedia.org/zh-cn/ISO_10646)标准定义了一个32位元的编码形式，称作UCS-4，使用[通用字符集](http://zh.wikipedia.org/zh-cn/%E9%80%9A%E7%94%A8%E5%AD%97%E7%AC%A6%E9%9B%86)（UCS）的每一个字符，会在0到[十六进制](http://zh.wikipedia.org/zh-cn/%E5%8D%81%E5%85%AD%E9%80%B2%E5%88%B6)的7FFFFFFF这样的字码空间中，被表示成一个的32位元的码值。

UCS-4足以用来表示所有的Unicode的字码空间，其最大的码位为十六进制的10FFFF，所以其空间约有百万个码位。有些人认为保留如此大的字码空间却只为了对应这很小的码集是浪费的所以一个新的编码UTF-32被提出来了。UTF-32是一个UCS-4的子集，使用32-位元的码值，只在0到10FFFF的字码空间。

UTF-32原本是UCS-4的子集，但JTC1/SC2/WG2声明，所有未来对字符的指定都将会限制在BMP及其14个补充平面，并移除先前在E0到FF平面的60到7F群的私用空间。

于是就现状而言，除了UTF-32标准包含额外的Unicode意涵，UCS-4和UTF-32大体是相同的。

# 程序如何处理字符编码

## 程序中为什么会出现乱码

乱码是个老问题，从上面我们知道，字符在保存时的编码格式如果和要显示的编码格式不一样的话，就会出现乱码问题。

我们的Web系统，从底层数据库编码、Web应用程序编码到HTML页面编码，如果有一项不一致的话，就会出现乱码。

所以，解决乱码问题说难也难说简单也简单，关键是让交互系统之间编码一致。

## HTML文件编码

### “charset“参数的意义

“charset”在这里指定了一种字符的编码方式，是如何把一串字节（二进制）转化为字符（可视符号）的方法。这种转化通常指的是Web的活动：服务器以字节流的形式发送HTML文档给用户代理（一般是浏览器），用户代理把这字节流解析为一系列的字母的过程。

### 如何选择一种编码

编排工具(如：文本编辑器)可能会以该工具选择的编码方式来为HTML文档编码，并且这种选择很大程序上取决于系统软件的约定（操作系统的内码）。这些工具通常会指定一个能覆盖绝大部分字符的字符集，而对于不在该字符集范围内的字符，可以通过字符引用的方式来表示。

服务器或者代理可能会转化编码来满足用户代理的需求（如HTTP请求头中指定了“Accept-Charset”参数）。但是，服务器或代理并没有这种义务支持全部类型的字符集。如国内出的一些浏览器通常对中文字符集支持较好，外文的就不行了。强大的用户代理（一般指浏览器）必须能够表示ISO10646包含的全部字符（至少在它们碰到的时候做出反映）。

如果一个HTML文件被编码为UTF-16，那么就会有字节序的问题，这时候强烈建议传一个**BOM**(0xFEFF)。

### 如何指定HTML文件的编码

* **服务器如何知道一个HTML文件的编码**

有一些服务器通过检查一个文档的前面几个字节或者通过查看一个包含了一些知名文件与编码方式的数据库来确定文档的编码方式。当今的Web服务器比以往的服务器在字符编码的配置上更多的控制（也就是能支持更多的字符编码）。服务器需要尽可能地发送一个“charset”参数给客户端，但必须注意不能发送一个错误的字符编码。

* **客户端怎么知道一个服务端传来的HTML使用了哪个编码**

当然，服务器得告诉客户端它发送过去的字节流使用了哪一个编码方式。一般，可以通过下面三种方式来指定，它们的发现优先级从高到低排列：

1. 使用HTTP协议的头域“**Content-Type**”来指定“charset”参数，如：Content-Type: text/html; charset=EUC-JP

HTTP协议规定，如何Content-Type中没有指定charset，那么默认使用ISO-8859-1编码方式。在实际过程中，这一推荐被证明是没什么作用的，因为很多服务器不允许发送“charset”这样一个参数，并且其它的一些服务器也许没有被配置为允许发送这一参数。因此，用户代理一定不能为“charset”参数假定任何默认值（注：人家都可能没传你假定有啥意义啊！）

1. 使用**meta声明**，如下面的示例声明了当前的编码方式为ISO-8859-1：

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

3）从一些带有charset属性的元素上推断出来。

上述三种方法给了浏览器很多发现编码方式的可能，如果还找不到字符的编码，那么浏览器可以通过探测机制或者让用户指定（很多浏览器都可以让用户选择html文件的编码方法）。

### 字符引用

当然，不可能每一个字符都能通过键盘或其它设备输入出来，那么HTML提供了两种方式来保存这些字符：

1）通过数字来引用（可以是十进制或十六进制）

可以表示为：&#D，这里D是一个十进制的数，数值对应于ISO 10646中的一个码点。或者表示为：&#xH，这里H是一个十六进制的数，数值对应于ISO 10646该十六进制数所在的码点。

2）字母实体引用

HTML预定义了几个引用实体来表示一些字符，如：

* + - * “&lt;”表示大于号<
      * “&gt;”表示小于号>
      * “&amp;”表示引用号&
      * “&quot;”表示引号”

### 浏览器如何处理不可显示的字符

浏览器可显示的字符跟所在机器的内码、字体都有关系，如在英文的操作系统上如果没有中文的字符的号那怎么都不可能显示出中文的。这时候，一般有两种做法：

1）提示用户所无法显示某些字符；

2）以上面讲到的16进制数值表示法来显示这些字符。

### 几个问题

问题1：如何读取到charset这一标记所在的位置？

答：一般都有某种算法确定HTML文件的编码，然后读取charset这一值。

问题2：HTML文件保存时设置的编码与charset选择的编码不一致时会出现什么情况？

答：一种情况，连charset这个字符都找不到了，那么情况就不清楚了；另一种情况，也许服务器或浏览器能正确解析出charset来，但是charset与实际编码不一致，服务器或浏览器还是按照charset指定下的编码来解析文件的话就会出现乱码。

## XML文件编码如何识别

XML由于它的结构化、良定义等特性越来越成为通用性的工具。无论是作了脚本语言、标记语言、还是作为数据传输的工具，越来越被广泛接受。

这里主要讨论如何识别XML文件的编码方式。由于XML的良定义，也使得XML文件的编码探测机制更加简单。接下来将介绍一种XML文件的编码自动探测机制。

* **一些XML规范的规定**

1. 所有的XML分析程序必须支持UTF-8和UTF-16编码；
2. 对于一个XML文件，如果没有外部信息说明它的编码方式或者该文件没有保存为UTF-8和UTF16，那么则必须使用xml的编码指示声明（像这样<?xml version=’1.0’ encoding=’gbk’>）；
3. 一个XML文件在出现“<?”串之前，不能出现任何字符（实际上现在的XML处理器允许<之前出现一些空白符），并且“<?xml”中的字符必须是连接出现。

* **一种编码自动探测算法**

1、对于没有encoding指示的，默认按照UTF-8或者UTF-16处理；

2、对于有encoding指示的，那么首先必须正确读取指示的内容，在正确读取到“encoding”之前，必须先推断出XML文件的编码。本节主要就是介绍这一推断的算法。

我们知道，在UCS-4中，’<’的码点是“#x0000003C”，’?’的码点是“#x0000003F”，BOM的UTF-16表示为#xFEFF，“<?xm”的ASCII表示为0x3C 0x3F 0x78 0x6D，根据这些知识以及XML语法的约束，该算法可以推断出XML文件的编码。该推断分两种情况处理：

* + **带BOM信息的文件**

|  |  |
| --- | --- |
| **bytes readed** | **encoding infered** |
| **00 00 FE FF** | **UCS-4，big-endian machine (1234 order)** |
| **FF FE 00 00** | **UCS-4，little-endian machine (4321 order)** |
| **00 00 FF FE** | **UCS-4，unusual octet order (2143)** |
| **FE FF 00 00** | **UCS-4，unusual octet order (3412)** |
| **FE FF ## ##** | **UTF-16，big-endian** |
| **FF FE ## ##** | **UTF-16，little-endian** |
| **EF BB BF** | **UTF-8** |

* + **不带BOM信息的文件**

|  |  |
| --- | --- |
| **bytes readed** | **encoding infered** |
| **00 00 00 3C** | 各种UCS-4编码方式或者其它以4字节为单位编码的编码方式，必须读取到encoding指示来确定具体是哪一种编码 |
| **3C 00 00 00** |
| **00 00 3C 00** |
| **00 3C 00 00** |
| **00 3C 00 3F** | UTF-16BE或者big-endian ISO-10646-UCS-2 或者其它以16个字节为单位编码的编码方式，必须读取到指示来真正确定是哪一种编码 |
| **3C 00 3F 00** | UTF-16LE 或者little-endian ISO-10646-UCS-2 或者其它以16个字节为单位编码的编码方式，必须读取到指示来真正确定是哪一种编码 |
| **3C 3F 78 6D** | 可能是UTF-8，ISO 646，ASCII，ISO 8859簇，Shift-JIS，EUC，或者其它任何7bit、8bit及混合编码的字符集，这些字符集对于串“<?xm”具有相同的码位，必须读取到指示来真正确定是哪一种编码 |
| **4C 6F A7 94** | EBCDIC (Extended Binary Coded Decimal Interchange Code) 扩增二进式十进交换码，为[IBM](http://zh.wikipedia.org/zh-cn/IBM)于[1963年](http://zh.wikipedia.org/zh-cn/1963%E5%B9%B4)－[1964年](http://zh.wikipedia.org/zh-cn/1964%E5%B9%B4)间推出的字符编码表 |
| **其它** | 不带encoding声明的UTF-8编码，或者数据流被错误标记（缺少一个必要的encoding声明），或者数据破坏了等等 |

## 字符与编码在程序中的实现

### 程序中的字符与字节

在C++和Java中，用来代表“字符”和“字节”的数据类型，以及进行编码的方法：

|  |  |  |
| --- | --- | --- |
| **类型或操作** | **C++** | **Java** |
| **字符** | wchar\_t | char |
| **字节** | char | byte |
| **ANSI字符串** | char[] | byte[] |
| **UNICODE 字符串** | wchar\_t[] | String |
| **字节串→字符串** | mbstowcs(), MultiByteToWideChar() | string = new String(bytes, "encoding") |
| **字符串→字节串** | wcstombs(), WideCharToMultiByte() | bytes = string.getBytes("encoding") |

以上需要注意几点：

1、Java中的char代表一个“UNICODE字符（宽字节字符）”，而C++中的char代表一个字节。

2、MultiByteToWideChar()和WideCharToMultiByte()是Windows API函数。

#### C++ 中相关实现方法

声明一段字符串常量：

|  |
| --- |
| // ANSI 字符串，内容长度 7 字节  char sz[20] = "中文123";  // UNICODE 字符串，内容长度5个wchar\_t（10字节）  wchar\_t wsz[20] = L"\x4E2D\x6587\x0031\x0032\x0033"; |

UNICODE字符串的I/O操作，字符与字节的转换操作：

|  |
| --- |
| // 运行时设定当前ANSI编码，VC格式  setlocale(LC\_ALL, ".936");  // GCC 中格式 setlocale(LC\_ALL, "zh\_CN.GBK");  // Visual C++中使用小写%s，按照setlocale指定编码输出到文件  // GCC 中使用大写%S  fwprintf(fp, L"%s\n", wsz);  // 把 UNICODE 字符串按照setlocale 指定的编码转换成字节  wcstombs(sz, wsz, 20);  // 把字节串按照 setlocale 指定的编码转换成UNICODE字符串  mbstowcs(wsz, sz, 20); |

在Visual C++中，UNICODE字符串常量有更简单的表示方法。如果源程序的编码与当前默认ANSI编码不符，则需要使用#pragma setlocale，告诉编译器源程序使用的编码：

|  |
| --- |
| // 如果源程序的编码与当前默认ANSI编码不一致，  // 则需要此行，编译时用来指明当前源程序使用的编码  #pragma setlocale(".936")  // UNICODE 字符串常量，内容长度 10 字节  wchar\_t wsz[20] = L"中文123"; |

以上需要注意#pragma setlocale与setlocale(LC\_ALL, "")的作用是不同的，#pragma setlocale在编译时起作用，setlocale()在运行时起作用。

#### Java中相关实现方法

字符串类String中的内容是UNICODE字符串：

|  |
| --- |
| // Java 代码，直接写中文  String string = "中文123";  // 得到长度为 5，因为是5个字符  System.out.println(string.length()); |

字符串I/O操作，字符与字节转换操作。在Java包java.io.\*中，以“**Stream**”结尾的类一般是用来操作“**字节串**”的类，以“**Reader**”，“**Writer**”结尾的类一般是用来操作“**字符串**”的类。

|  |
| --- |
| // 字符串与字节串间相互转化  // 按照GB2312得到字节（得到多字节字符串）  byte [] bytes = string.getBytes("GB2312");  // 从字节按照GB2312得到 UNICODE 字符串  string = new String(bytes, "GB2312");  // 要将String按照某种编码写入文本文件，有两种方法：  // 第一种办法：用Stream类写入已经按照指定编码转化好的字节串 OutputStream os = new FileOutputStream("1.txt");  os.write(bytes);  os.close();  // 第二种办法：构造指定编码的Writer来写入字符串  Writer ow = new OutputStreamWriter(new FileOutputStream("2.txt"), "GB2312");  ow.write(string);  ow.close();  /\* 最后得到的 1.txt 和 2.txt 都是 7 个字节 \*/ |

如果java的源程序编码与当前默认ANSI编码不符，则在编译的时候，需要指明一下源程序的编码。比如：

|  |
| --- |
| E:\>javac -encoding BIG5 Hello.java |

以上需要注意区分源程序的编码与I/O操作的编码，前者是在编译时起作用，后者是在运行时起作用。

### 几种乱码产生的原因和解决办法

#### 容易产生的误解

**误解一：**

在将“字节串”转化成“UNICODE字符串”时，比如在读取文本文件时，或者通过网络传输文本时，容易将“字节串”简单地作为**单字节字符串**，采用每“一个字节”就是“一个字符”的方法进行转化。

而实际上，在非英文的环境中，应该将“字节串”作为ANSI字符串，采用适当的编码来得到UNICODE字符串，有可能“多个字节”才能得到“一个字符”。

在DOS，Windows 98等非UNICODE环境下，字符串都是以ANSI编码的字节形式存在的。这种以字节形式存在的字符串，必须知道是哪种编码才能被正确地使用。这使我们形成了一个惯性思维：“字符串的编码”。

通常，一直在英文环境下做开发的程序员们，容易有这种误解。

**误解二：**

当UNICODE被支持后，Java中的String是以字符的“序列”来存储的，不是以“某种编码的字节”来存储的，因此已经不存在“字符串的编码”这个概念了。只有在“字符串”与“字节串”转化时，或者，将一个“字节串”当成一个ANSI字符串时，才有编码的概念。

不少的人都有这个误解。

第一种误解，往往是导致乱码产生的原因。

第二种误解，往往导致本来容易纠正的乱码问题变得更复杂。

在这里，我们可以看到，其中所讲的“误解一”，即采用每“一个字节‘就是’一个字符”的转化方法，实际上也就等同于采用iso-8859-1进行转化。因此，我们常常使用bytes = String.getBytes("iso-8859-1")来进行逆向操作，得到原始的“字节串”。然后再使用正确的ANSI编码，比如String = new String(bytes, "GB2312")，来得到正确的“UNICODE字符串”。

#### 非UNICODE程序在不同语言环境间移植时的乱码

非UNICODE程序中的字符串，都是以某种ANSI编码形式存在的。如果程序运行时的语言环境与开发时的语言环境不同，将会导致ANSI字符串的显示失败。

比如，在日文环境下开发的非UNICODE的日文程序界面，拿到中文环境下运行时，界面上将显示乱码。如果这个日文程序界面改为采用UNICODE来记录字符串，那么当在中文环境下运行时，界面上将可以显示正常的日文。

由于客观原因，有时候我们必须在中文操作系统下运行非UNICODE的日文软件，这时我们可以采用一些工具，比如，南极星，AppLocale等，暂时的模拟不同的语言环境。

#### 深入理解URL编码及乱码问题

* **URL编码简介**

几条规则：

* 字母数字字符“a”到“z”、“A”到“Z”和“0”到“9”保持不变。
* 特殊字符“.”、“-”、“\*”和“\_”保持不变。
* 空格字符“ ”转换为一个加号“+”。
* 所有其他字符都是不安全的，因此首先使用一些编码机制将它们转换为一个或多个字节。然后每个字节用一个包含3个字符的字符串“%xy”表示，其中xy为该字节的两位十六进制表示形式。**推荐的编码机制是UTF-8**。但是，出于兼容性考虑，如果未指定一种编码，则使用相应平台的默认编码。

例如，使用UTF-8编码机制，字符串"The string @foo-bar"将转换为"The+string+%C3%BC%40foo-bar"，因为在UTF-8中，字符ü编码为两个字节，C3（十六进制）和BC（十六进制），字符@编码为一个字节40（十六进制）。

* **一个乱码现象例子(Tomcat服务器下)**

当页面中的表单提交字符串时，首先把字符串转按照当前页面的编码对应的字节串。然后再将每个字节转化成“%XX”的格式提交到Web服务器。比如，一个编码为GB2312的页面，提交“中”这个字符串时，提交给服务器的内容为“%D6%D0”。

在服务器端，Web服务器把收到的“%D6%D0”转化成[0xD6, 0xD0]两个字节，然后再根据GB2312编码规则得到“中”字。

在Tomcat服务器中，request.getParameter()得到乱码时，常常是因为前面提到的“误解一”造成的。默认情况下，当提交“%D6%D0”给Tomcat服务器时，request.getParameter()将返回[0x00D6, 0x00D0]两个的UNICODE字符，而不是返回一个“中”字符。因此，我们需要使用bytes = string.getBytes("iso-8859-1")得到原始的字节串，再用String = new String(bytes, "GB2312")重新得到正确的字符串“中”。

**1、**在URL中中文字符通常出现在以下两个地方：

1）query string中的参数值，比如<http://search.china.alibaba.com/search/offer_search.htm?keywords>=中国

2）servlet path，比如：

<http://search.china.alibaba.com/selloffer/>中国.html

**2、**出现乱码问题的原因主要是以下几方面：

1）浏览器：我们的客户端（浏览器）本身并没有遵循URI编码的规范（<http://www.w3.org/International/O-URL-code.html>）；

2）Servlet服务器：Servlet服务器的没有正确配置；

3）开发人员并不了解Servlet的规范和API的含义。

* **基础知识回顾**

**1、一个http请求经过的几个环节：**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **第一步** | **第二步** | **第三步** |
| 目标 | 浏览器(ie、firefox)【get/post】 | Servlet服务器 | 浏览器显示 |
| **动作** | 编码 | 解码成Unicode，将显示的内容编码 | 解码 |

1）浏览器把URL(以及post提交的内容)经过编码后发送给服务器。

2）这里的Servlet服务器实际上指的是由Servlet服务器提供的servlet实现ServletRequestWrapper，不同应用服务器的servlet实现不同，这些servlet的实现把这些内容解码转换为Unicode，处理完毕后，然后再把结果(即网页)编码返回给浏览器。

3）浏览器按照指定的编码显示该网页。

当对字符串进行编码和解码的时候都涉及到字符集，通常使用的字符集为ISO8859-1、GBK、UTF-8、UNICODE。

**2、URL的组成**

域名:端口/contextPath/servletPath/pathInfo?queryString

说明：

1、ContextPath是在Servlet服务器的配置文件中指定的。

对于Tomcat：

contextPath是在server.xml中配置。

<Context path="/" docBase="D:/server/blog.war" debug="5" reloadable="true" crossContext="true"/>

2、ServletPath是在应用的web.xml中配置。

<servlet-mapping>

<servlet-name>Example</servlet-name>

<url-pattern>/example/\*</url-pattern>

</servlet-mapping>

**3、Servlet API**

我们使用以下servlet API获得URL的值及参数。

request.getParameter("name"); // 获得queryString的参数值(来自于get和post)，其值经过Servlet服务器URL Decode过的

request.getPathInfo(); // 注意：pathinfo返回的字符串是经过Servlet服务器URL Decode过的。

requestURI = request.getRequestURI(); // 内容为：contextPath/servletPath/pathinfo浏览器提交过来的原始数据，未被Servlet服务器URL Decode过。

**3、开发人员必须清楚的servlet规范：**

1）HttpServletRequest.setCharacterEncoding()方法仅仅只适用于设置post提交的request body的编码而不是设置get方法提交的queryString的编码。该方法告诉应用服务器应该采用什么编码解析post传过来的内容。很多文章并没有说明这一点。

2）HttpServletRequest.getPathInfo()返回的结果是由Servlet服务器解码(decode)过的。

3）HttpServletRequest.getRequestURI()返回的字符串没有被Servlet服务器解码（decode）过的。

4）POST提交的数据是作为request body的一部分。

5）网页的Http头中ContentType("text/html; charset=GBK")的作用：

* + 1. 告诉浏览器网页中数据是什么编码；
    2. 表单提交时，通常浏览器会根据ContentType（HTTP协议头中）指定的charset对表单中的数据编码，然后发送给服务器的；
    3. 这里需要注意的是：这里所说的ContentType是指http头的ContentType，而不是在网页中meta中的ContentType（这个编码用于浏览器解析从服务端传送过来的html数据流，即html文件）。
* **详细过程举例**

下面我们分别从浏览器和应用服务器（Tomcat）来举例说明：

URL：<http://localhost:8080/example/>中国?name=中国

汉字编码二进制表示

“中国”UTF-8表示为： 0xe4 0xb8 0xad 0xe5 0x9b 0xbd

“中国”的GBK表示为： 0xd6 0xd0 0xb9 0xfa

“中国”的ISO8859-1表示为：0x3f 0x3f信息失去

**(一)、浏览器**

**1、**GET方式提交，浏览器会对URL进行URL encode，然后发送给服务器

1）对于中文IE，如果在高级选项中选中总以UTF-8发送(默认方式)，则PathInfo是URL Encode是按照UTF-8编码，QueryString是按照GBK编码。

<http://localhost:8080/example/>中国?name=中国

实际上提交是：

GET /example/%E4%B8%AD%E5%9B%BD?name=%D6%D0%B9%FA

问题：servlet服务器咋知道%E4%B8%AD%E5%9B%BD表示“中国呢”，还不是三位串“涓浗”呢？难道根据UTF的三位模式1110xxxx 10xxxxxx 10xxxxxx？

2）对于中文IE，如果在高级选项中取消总以UTF-8发送，则PathInfo和QueryString是URL encode按照GBK编码。

实际上提交是：

GET /example/%D6%D0%B9%FA?name=%D6%D0%B9%FA

3）对于中文firefox，则pathInfo和queryString都是URL encode按照GBK编码。实际上提交是：

GET /example/%D6%D0%B9%FA?name=%D6%D0%B9%FA

很显然，不同的浏览器以及同一浏览器的不同设置，会影响最终URL中PathInfo的编码。对于中文的IE和FIREFOX都是采用GBK编码QueryString。

**小结：**解决方案：

1、URL中如果含有中文等非ASCII字符，则浏览器会对它们进行URLEncode。为了避免浏览器采用了我们不希望的编码，所以最好不要在URL中直接使用非ASCII字符，而采用URL Encode编码过的字符串%XX。比如：

URL：<http://localhost:8080/example/>中国?name=中国

建议：URL：<http://localhost:8080/example/%D6%D0%B9%FA?name=%D6%D0%B9%FA>

2、我们建议URL中PathInfo和QueryString采用相同的编码，这样对服务器端处理的时候会更加简单。

java.net.URLEncoder包中就有如何将一个字符串编码为一个URL的方法：

Static String encode(String s, String enc)；

**2、POST提交**

对于POST方式，表单中的参数值对是通过request body发送给服务器，此时浏览器会根据网页的HTTP协议头中ContentType(如："text/html; charset=GBK")中指定的编码进行对表单中的数据进行编码，然后发给服务器。

在服务器端的程序中我们可以通过Request.setCharacterEncoding()设置编码，然后通过request.getParameter获得正确的数据。

**解决方案：**

从最简单，所需代价最小来看，我们对URL以及网页中的编码使用统一的编码对我们来说是比较合适的。如果不使用统一编码的话，我们就需要在程序中做一些编码转换的事情。这也是我们为什么看到有网络上大量的资料介绍如何对乱码进行处理，其中很多解决方案都只是一时的权宜之计，没有从根本上解决问题。

**(二)、Servlet服务器**

Servlet服务器实现的Servlet遇到URL和POST提交的数据中含有%的字符串，它会按照指定的字符集解码。下面两个Servlet方法返回的结果都是经过解码的：

request.getParameter("name");

request.getPathInfo();

这里所说的“指定的字符集”是在应用服务器的配置文件中配置。

Tomcat服务器关于字符编码的配置文件是：server.xml

<Connector port="8080" protocol="HTTP/1.1"

maxThreads="150" connectionTimeout="20000"

redirectPort="8443" URIEncoding="GBK"/>

URIEncoding告诉服务器servlet解码URL时采用的编码。

<Connector port="8080" useBodyEncodingForURI="true" />

useBodyEncodingForURI告诉服务器解码URL时候需要采用request body指定的编码。

**(三)浏览器显示**

浏览器根据http头中的ContentType("text/html; charset=GBK")，指定的字符集来解码服务器发送过来的字节流。我们可以调用HttpServletResponse.setContentType()设置http头的ContentType。

总结：

1、URL中的PathInfo和QueryString字符串的编码和解码是由浏览器和应用服务器的配置决定的，我们的程序不能设置，不要期望用request.setCharacterEncoding()方法能设置URL中参数值解码时的字符集。

所以我们建议URL中不要使用中文等非ASCII字符，如果含有非ASCII字符的话要使用URLEncode编码一下，比如：

<http://localhost:8080/example1/example/>中国

正确的写法：

<http://localhost:8080/example1/example/%E4%B8%AD%E5%9B%BD>

并且我们建议URL中不要在PathInfo和QueryString同时使用非ASCII字符，比如：

<http://localhost:8080/example1/example/>中国?name=中国

原因很简单：不同浏览器对URL中PathInfo和QueryString编码时采用的字符集不同，但应用服务器对URL通常会采用相同的字符集来解码。

2、我们建议URL中的URL Encode编码的字符集和网页的contentType的字符集采用相同的字符集，这样程序的实现就很简单，不用做复杂的编码转换。

#### 从数据库读取字符串

通过数据库驱动（比如ODBC或JDBC）从数据库服务器中读取字符串时，驱动需要从数据库服务器获知所使用的ANSI编码。当数据库服务器发送字节流给驱动时，驱动负责将字节流按照正确的编码转化成UNICODE字符串（这里指JDBC的处理情况）。

如果从数据库读取字符串时得到乱码，而数据库中存放的数据又是正确的，那么往往还是因为前面提到的“误解一”造成的。解决的办法还是通过String = new String(string.getBytes("iso-8859-1"), "GB2312")的方法，重新得到原始的字节串，再重新使用正确的编码转化成字符串。

#### 电子邮件中的字符串

当一段Text或者HTML通过电子邮件传送时，发送的内容首先通过一种指定的**字符编码**转化成“字节串”，然后再把“字节串”通过一种指定的**传输编码**（Content-Transfer-Encoding）进行转化得到另一串“字节串”。比如，打开一封电子邮件源代码，可以看到类似的内容：

|  |
| --- |
| Content-Type: text/plain;  charset="gb2312"  Content-Transfer-Encoding: base64  sbG+qcrQuqO17cf4yee74bGjz9W7+b3wudzA7dbQ0MQNCg0KvPKzxqO6uqO17cnnsaPW0NDEDQoNCg== |

最常用的Content-Transfer-Encoding有Base64和Quoted-Printable两种。在对二进制文件或者中文文本进行转化时，Base64得到的“字节串”比Quoted-Printable更短。在对英文文本进行转化时，Quoted-Printable得到的“字节串”比Base64更短。

邮件的标题，用了一种更简短的格式来标注“字符编码”和“传输编码”。比如，标题内容为“中”，则在邮件源代码中表示为：

|  |
| --- |
| // 正确的标题格式（标题内容“中”）  Subject: =?GB2312?B?1tA=?= |

其中，

* 第一个“=?”与“?”中间的部分指定了字符编码，在这个例子中指定的是GB2312。
* “?”与“?”中间的“B”代表Base64。如果是“Q”则代表Quoted-Printable。
* 最后“?”与“?=”之间的部分，就是经过GB2312转化成字节串，再经过Base64转化后的标题内容。

如果“传输编码”改为Quoted-Printable，同样，如果标题内容为“中”：

|  |
| --- |
| // 正确的标题格式  Subject: =?GB2312?Q?=D6=D0?= |

如果阅读邮件时出现乱码，一般是因为“字符编码”或“传输编码”指定有误，或者是没有指定。比如，有的发邮件组件在发送邮件时，标题“中”：

|  |
| --- |
| // 错误的标题格式  Subject: =?ISO-8859-1?Q?=D6=D0?= |

这样的表示，实际上是明确指明了标题为[0x00D6, 0x00D0]，即“ÖÐ”，而不是“中”。

### 几个错误理解的纠正

#### 误解1：“ISO-8859-1 是国际编码？”

非也！iso-8859-1只是单字节字符集中最简单的一种，也就是“字节编号”与“UNICODE字符编号”一致的那种编码规则。当我们要把一个“字节串”转化成“字符串”，而又不知道它是哪一种ANSI编码时，先暂时地把“每一个字节”作为“一个字符”进行转化，不会造成信息丢失。然后再使用bytes = string.getBytes("iso-8859-1")的方法可恢复到原始的字节串。

#### 误解2：“Java中，怎样知道某个字符串的内码？”

Java中，字符串类java.lang.String处理的是UNICODE字符串，不是ANSI字符串。我们只需要把字符串作为“抽象的符号的串”来看待。因此不存在字符串的内码的问题。

# 一个通用的编码识别算法简介

<http://www.mozilla.org/projects/intl/UniversalCharsetDetection.html>



一个python实现：

<http://chardet.feedparser.org/>

# 参考资料

1、[http://zh.wikipedia.org](http://zh.wikipedia.org/)

2、