

**UNICODE使用经验总结**

**软件一部：徐少辉**

**2019.04.26**

**目录**

[一 常用字符集简介 3](#_Toc7171152)

[1.1 ASCII码 3](#_Toc7171153)

[1.2 ISO-8859-1 3](#_Toc7171154)

[1.3 GBK 3](#_Toc7171155)

[1.4 Unicode 4](#_Toc7171156)

[1.4.1 Unicode实现方案 4](#_Toc7171157)

[1.4.2 UTF-16 5](#_Toc7171158)

[1.4.2.1 举例 5](#_Toc7171159)

[1.4.3 UTF-8 6](#_Toc7171160)

[1.4.3.1 举例 6](#_Toc7171161)

[1.4.3.2 UTF-8解码 6](#_Toc7171162)

[1.4.4 UCS-2 7](#_Toc7171163)

[1.4.5 UCS-4/UTF-32 7](#_Toc7171164)

[二 大端/小端与BOM 8](#_Toc7171165)

[三 编程实践 9](#_Toc7171166)

[3.1 Windows 9](#_Toc7171167)

[3.1.1 编码转换函数 10](#_Toc7171168)

[3.1.1.1 UTF-8转换为UTF-16 10](#_Toc7171169)

[3.1.1.2 UTF-16转换为UTF-8 10](#_Toc7171170)

[3.1.1.3 UTF-16转ANSI 10](#_Toc7171171)

[3.1.1.4 ANSI转UTF-16 11](#_Toc7171172)

[3.2 Linux 11](#_Toc7171173)

[3.2.1 iconv-维基百科 11](#_Toc7171174)

[3.2.2 编码转换函数 11](#_Toc7171175)

[3.2.2.1 UTF-8转UTF-16LE 12](#_Toc7171176)

[3.2.2.2 UTF-8转GB2312 12](#_Toc7171177)

[四 HMITerm字符编码优化分享 13](#_Toc7171178)

[4.1 优化准则 13](#_Toc7171179)

[4.2 性能提升 13](#_Toc7171180)

# 常用字符集简介

## ASCII码

要谈Unicode的发展历史，得先从ASCII讲起，相信学过计算机的同学都听过ASCII。ASCII 是 “美国信息交换标准代码”（American Standard Code for Information Interchange）的缩写，念作 “阿斯奇”（ass-kee）。

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，最前面的一位统一规定为0。

最终的代码有26个小写字母、26个大写字母、10个数字、32个符号、33个控制代码和一个空格。

ASCII现在的标准是ANSI X3.4-1986。

## ISO-8859-1

当计算机开始在欧洲开始使用的时候，人们发现ASCII不够用了，比如法语、德语中的一些字母，于是开始利用ASCII中最前面的那一位，这样可以多出128个编码。

此字符集支持部分欧洲使用的语言，于1987年被标准化。

ISO-8859-1的别名有：iso-ir-100, csISOLatin1, latin1, l1, IBM819，Oracle数据库称之为WE8ISO8859P1。

## GBK

等到中国人们开始使用计算机后，已经没有可以扩展的bit位了，况且又6000多个常用汉字需要编码。于是经过长期的摸索，得出了以下的方案来编码汉字：

* 把127之后的奇异符号直接取消掉。
* 一个小于127的字符的意义与ASCII相同。
* 两个大于127的字符连在一起，表示一个汉字。
* 前面的一个字节从0xA1用到0xF7，后面一个字节从0xA1到0xFE。

这样就可以组合出7000多个简体汉字，甚至把ASCII里本来就有的数字、标点、字母都重新编了两个字节的编码，这就是常说的“全角”字符，而原来ASCII中的那些就叫“半角”字符。

并且最终这个汉字编码方案被叫做“GB2312”。

中国的汉字实在太多了，很快就发现这个方案不够用了，比如“镕”。于是干脆不再要求低字节一定是大于127的，只要高字节大于127就表示这是一个汉字的开始。这个编码方案被称为GBK，比GB2312多了近20000个汉字和符号。

GBK 只为“技术规范指导性文件”，不属于国家标准。国家质量技术监督局于2000年3月17日推出了GB 18030-2000标准，以取代GBK。GB 18030-2000除保留全部GBK编码汉字，在第二字节把能使用范围再度进行扩展，增加了大约一百个汉字及四位元组编码空间，但是将GBK作为子集全部保留。

## Unicode

在这个时期，各个国家或地区都像中国这样搞出一套自己的编码标准，结果互相之间谁也不懂谁的编码，谁也不支持别人的编码，连大陆和台湾这样只相隔了150海里，使用着同一种语言，也是这样，台湾用的是BIG5编码方案。

ISO决定着手解决这个问题，他们采用的方法很简单：废除了所有的地区性编码方案，重新搞一个包括了地球上所有文化、所有字母和符号的编码!他们打算叫它"Universal Multiple-Octet Coded Character Set"，简称 UCS, 俗称 "UNICODE"。

Unicode开始制订时，计算机的存储器容量极大地发展了，空间再也不成为问题了。于是 ISO 就直接规定必须用两个字节，也就是16位来统一表示所有的字符，对于ascii里的那些"半角"字符，UNICODE 保持其原编码不变，只是将其长度由原来的8位扩展为16位，而其他文化和语言的字符则全部重新统一编码。

如上所述，UNICODE 是用两个字节来表示一个字符，他总共可以组合出65535不同的字符，这大概已经可以覆盖世界上所有文化的符号。如果还不够也没有关系，ISO已经准备了UCS-4方案，说简单了就是四个字节来表示一个字符，这样我们就可以组合出21亿个不同的字符出来了。

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（中日韩汉字Unicode编码表：<http://www.chi2ko.com/tool/CJK.htm>），转换成二进制数有15位（100111000100101），也就是说，这个符号的表示至少需要2个字节。表示其他的符号，可能需要3个字节或者4个字节，甚至更多。

### Unicode实现方案

在Unicode字符集中的某个字符对应的代码值，称作代码点（**Code Point**），用16进制书写，并加上U+前缀。比如，‘田’的代码点是U+7530；‘A’的代码点是U+0041。

Unicode定义的字符集已经超过16位所能表达的范围，把所有这些Code Point分成17个代码平面（**Code Plane**）：

* U+0000 ~ U+FFFF划入基本多语言平面（Basic MultilingualPlane，简记为**BMP**）；
* 其余划入16个辅助平面（Supplementary Plane），代码点范围U+10000 ~ U+10FFFF。

虽然这样划分，但并不是每个Plane中的Code point都对应有字符，这里面有保留的，还有特殊用途的。

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，但是在实际存储和传输过程中，由于不同系统平台的设计不一定一致，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 Transformation Format，简称为UTF）。

对Unicode编码的主要有UTF-16BE、UTF-16LE、UTF-8、UTF-7以及UTF-32等实现方式，目前常用的实现方式是UTF-16LE、UTF-16BE和UTF-8。

### UTF-16

UTF-16是用16bit编码来表达Unicode，这样表达范围是216（即65536），也就是UTF-16的代码单元（Code Unit）为16bits。如果表达BMP内的字符，用一个UTF-16的Code Unit就可表达，对于辅助平面内的字符，UTF-16有巧妙的设计。

* 落在BMP内，从U+D800到U+DFFF之间的Code Point区段是永久保留不映射到字符， UTF-16利用这保留下来的0xD800-0xDFFF区段的CodePoint来对辅助平面内的字符的Code Point进行编码。
* 辅助平面(Supplementary Planes)中的CodePoint，在UTF-16中被编码为一对16bit长的Code Unit（即32bit，4Bytes），称作代理对(**Surrogate Pair**)。

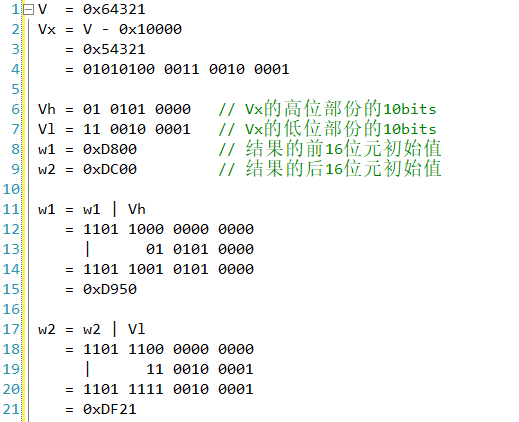
具体方法是：

1. Code Point减去0x10000， 得到的值是长度为20bit（0..0xFFFFF）；
2. 步骤1得到数值的高位的10比特的值（值范围为0..0x3FF）被加上0xD800得到第一个Code Unit或称作高位代理（high surrogate）或前导代理（lead surrogate）。取值范围是0xD800..0xDBFF。
3. 步骤1得到数值的低位的10比特的值（值范围为0..0x3FF）被加上0xDC00得到第二个Code Unit或称作低位代理（low surrogate）或后尾代理（trail surrogate）。取值范围是0xDC00..0xDFFF。

这样，这个范围内的字符就被编码成了一个代理对[lead surrogate,trail surrogate]：两个16bits的Code Unit，取值范围分别是0xD800..0xDBFF和0xDC00..0xDFFF。而BMP中得到的Code Unit的范围是0x0000..0xFFFF（0xD800..0xDFFF是保留的，不包含其中），所以这三个区段是相互不重叠的，在解码时很容易实现。

#### 举例

对U+64321进行UTF-16编码：



所以，U+64321最终的UTF-16编码是：0xD950 0xDF21

UTF-16的Code Unit是16bits，两个字节。存储一个Code Unit的时候，还有存取的先后顺序问题，也就是大小端问题，这在后面再说。

### UTF-8

UTF-8（8-bit Unicode Transformation Format）是一种针对Unicode的可变长度字符编码，使用一至四个字节为每个字符编码：

1. Unicode范围为U+0000..U+007F 的128个ASCII字符只需一个字节编码；
2. Unicode范围为U+0080..U+07FF的字符需要二个字节编码；
3. Unicode范围为U+0800..U+FFFF的其他BMP中的字符（这包含了大部分常用字）使用三个字节编码；
4. Unicode 辅助平面的字符（其他极少使用的字符）使用四字节编码。

对CodePoint各个范围内的字符进行UTF-8编码的规则如下：



#### 举例

对U+7530进行UTF-8编码：

1. U+7530落在U+0800..U+FFFF区间，采用三字节编码；
2. 0x7530转换为二进制为111 010100 110000；
3. 代入表中，得到11100111 10010100 10110000；

这样，最终的UTF-8编码为：0xE7 94 B0

#### UTF-8解码

知道UTF-8的编码规则，我们可以对于UTF-8编码中的任意字节BYTE，进行下面解码：

1. 如果B的第一位为0，则B为ASCII码，并且B独立的表示一个字符；如果B的第一位为1，第二位为0，则B为一个非ASCII字符（该字符由多个字节表示）中的一个字节，并且不为字符的第一个字节编码（字符的第一个字节之外的后编码）；
2. 如果B的前两位为1，第三位为0，则B为一个非ASCII字符（该字符由多个字节表示）中的第一个字节，并且该字符由两个字节表示；
3. 如果B的前三位为1，第四位为0，则B为一个非ASCII字符（该字符由多个字节表示）中的第一个字节，并且该字符由三个字节表示；
4. 如果B的前四位为1，第五位为0，则B为一个非ASCII字符（该字符由多个字节表示）中的第一个字节，并且该字符由四个字节表示。

### UCS-2

UCS-2每个字符占用2个字节。UCS-2是UTF-16的子集。在没有辅助平面前，UTF-16与UCS-2所指的是同一的意思。但当引入辅助平面字符后，UTF-16加入了对辅助平面内的字符的支持。现在若有软件声称自己支持UCS-2编码，那其实是暗指它不支持UTF-16中超过2bytes的字集。亦即，对于小于0x10000的UCS码，UTF-16编码就等于UCS码。Java早期版本对Unicode的支持，就只是UCS-2的支持，现在加入了对UTF-16的完整支持。

### UCS-4/UTF-32

UCS-4与UTF-32的意义一致，对每个字符都使用4字节（31位字符集，加上恒为0的首位，共需占据32位）。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。虽然每一个Code Point使用固定长定的字节看似方便，对于普通只需要2个字节存储的常用字占绝大对数的字符集来说，却极大的浪费了空间，并没怎么得到应用。

# 大端/小端与BOM

在讲UTF-16编码方式时说到，UTF-16编码的Code Unit是2个字节，这两个字节在传输和存储过程中，高/低位位置不同，是不同的字符。比如，“田”的UTF-16编码是0x7530，但是如果存成0x3075，就变成了“ふ”，成了另外的字符。

所以，为了识别一个编码过的字符的存储顺序，必须用特殊字符来指示。Unicode字符中U+FEFF被用来指示这种存储顺序，被称作Byte Order Mark（BOM）。

**大端Big-Endian**：最低位地址存放高位字节，可称高位优先，内存从最低地址开始按顺序存放（高数位数字先写）。最高位字节放最前面。

**小端Little Endian**：最低位地址存放低位字节，可称低位优先，内存从最低地址开始按顺序存放（低数位数字先写）。最低位字节放最前面。

BOM在Big-Endian系统上存储为**FE FF**；而在Big-Endian系统上存储则为**FF FE**。所以在以Big-Endian存储的UTF-16（UTF-16BE）的文件的开头，用FEFF指示；以Little-Endian存储的UTF-16（UTF-16LE）的文件的开头，用FFFE指示。

BOM的UTF-8编码为11101111 1011101110111111 (**EF BB BF**)，所以一般EF BB BF被放在文本的开头，用来指示其编码为UTF-8。

# 编程实践

## Windows

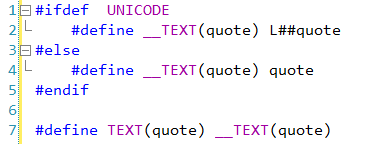
设想我们在写代码时，定义了一个变量：

char \*str = "china";

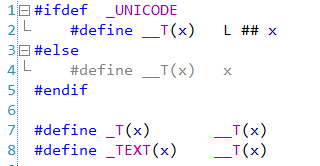
如果后来该字符串需要换成中文，那就得换成：

wchar\_t \*str = L"中国"; //L表示紧跟的字符串按UNICODE处理

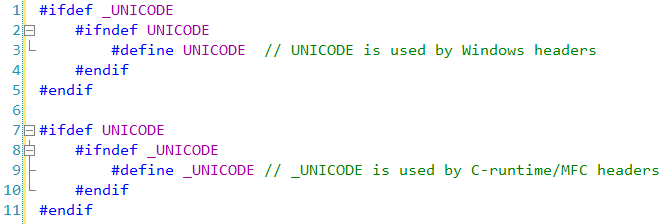
如果有很多字符串的话，这样改起来就很麻烦了。所以VisualStudio团队的人想出了一个解决办法，那就是\_\_TEXT()，\_\_T()等一系列宏。



同样的，Windows团队的人也做了类似的事情，定义了一些相似的宏。

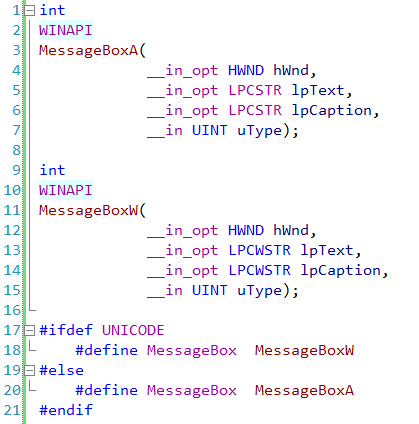


那么，疑问来了，为什么要定义\_UNICODE和UNICODE两个宏？微软这么做的理由是什么呢？要是一个宏定义了，另一个宏没定义引起冲突怎么办？于是再来查找一番，在atldef.h和afxv\_w32.h两个头文件中:



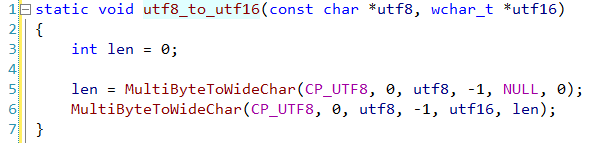
这段代码的目的很清晰，就是要保证UNICODE和\_UNICODE这两个宏要么都定义了，要么都未定义，不能只定义一个。

还有，大部分的Win32 API都提供了兼容ANSI和UNICODE的版本：

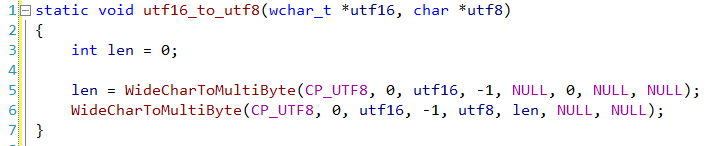


### 编码转换函数

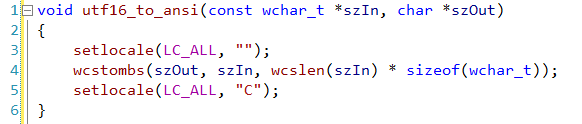
#### UTF-8转换为UTF-16



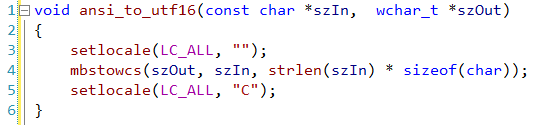
#### UTF-16转换为UTF-8



#### UTF-16转ANSI



#### ANSI转UTF-16



## Linux

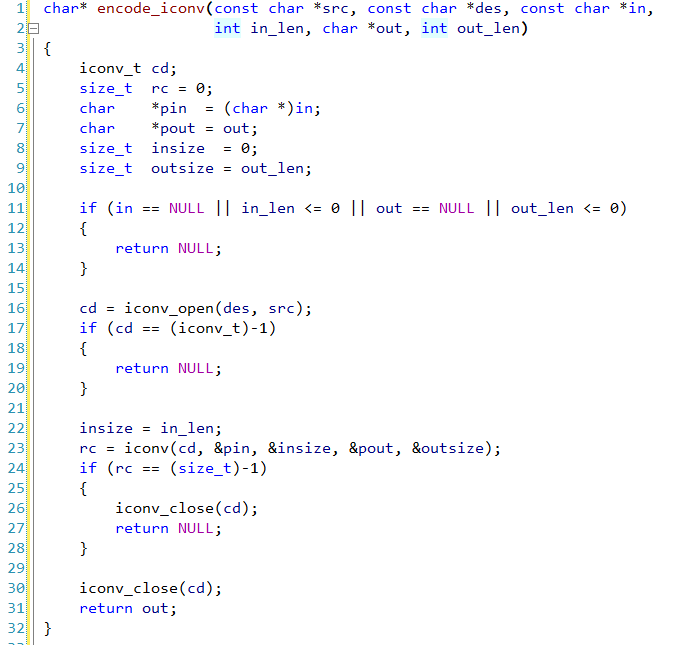
提示：Linux系统下没有Windows的代码页的概念。Linux用的是locale system，Linux使用iconv库来完成各种编码的转换。

### iconv-维基百科

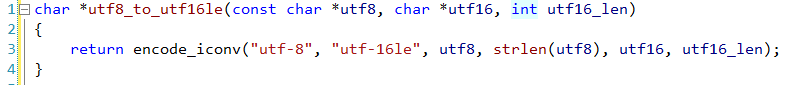
iconv是一个计算机程序以及一套应用程序编程接口的名称。iconv基于GPL公开源代码，是GNU项目的一部分。它的作用是在多种国际编码格式之间进行文本内码的转换。支持的内码包括：Unicode相关编码，如UTF-8、UTF-16等等，各国采用的ANSI编码，其中包括GB2312、BIG5等中文编码方式。

当前，libiconv已经包含在C运行时刻库libc.so中。因此，Linux平台上使用iconv库函数的程序，需要包含<iconv.h>，但链接时不需要引入libiconv库了。

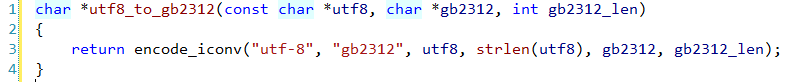
### 编码转换函数



#### UTF-8转UTF-16LE



#### UTF-8转GB2312



# HMITerm字符编码优化分享

2019年3月份，软件一部集中力量组织了一次封闭式开发，着手解决HMITerm中字符编码的问题，主要的问题集中在两点:

1. Linux下的wchar\_t被定义为4个字节，映射到UNICODE的UTF-32。这与Windows下的UTF-16不同，在HMITerm代码中需要在多处进行特殊处理两个平台的差异。
2. HMITerm中大部分函数，字符串，字符数组都是宽字符版本，而xml画面文件是UTF-8编码，在解析xml和存储部件信息时，采用宽字符版本，浪费内存。

## 优化准则

基于上面两点，此次封闭式开发，事先拟定了一些准则：

* cpp源代码文件格式，统一用utf-8带签名，在windows和linux上都能正常识别，vs需要安装VS90SP1-KB980263-x86补丁。
* 路径操作
* windows: fopen函数打不开utf8编码的文件名，需要转换为utf16，然后调用\_wfopen来打开文件。
* linux：本身的locale就是utf-8，fopen可以直接打开。
* xml解析库改成utf8进，utf8出，不做任何转码。
* sqlite操作，去掉所有的宽窄字符的转换，保持utf8进，utf8出。
* SCREENINFO和PARTLIST两个结构不使用TCHAR来处理字符串，而是char，统一为utf8。
* hmiui也是使用的utf16开发，建议改成utf8，少执行一次互转，并且如果ui和term合并的话，建议统一基于utf8编码开发。
* 所有的工程都不定义UNICODE或\_UNICODE。

## 性能提升

下面是经过优化后，HMITerm在内存使用上的提升数据：

