**1. 基础**  
  
**1.1 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 0000000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.  
  
**1.2 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 (?), 在俄语编码中又会代表另一个符号.  
  
NOTE:  
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. // MMMMM  
  
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.  
  
**2. Unicode**  
  
**2.1 Unicode的定义**  
  
正如上一节所说, 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也是一种字符编码方法, 不过它是由国际组织设计, 可以容纳全世界所有语言文  
字的编码方案. Unicode的学名是"**Universal Multiple-Octet Coded Character Set**",  
简称为UCS. UCS可以看作是"**Unicode Character Set**"的缩写.  
  
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, 或者专门的汉字对应表.  
  
**2.2 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  
  
  
**3. 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和UTF-32, 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1~6个字节表示一个符  
号, 根据不同的符号而变化字节长度.  
  
**3.1 UTF-8的编码规则**  
  
UTF-8的编码规则很简单, 只有两条:  
  
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.

// #txt---

| Unicode符号范围 | UTF-8编码方式

n | (十六进制) | (二进制)

---+-----------------------+------------------------------------------------------

1 | 0000 0000 - 0000 007F | 0xxxxxxx

2 | 0000 0080 - 0000 07FF | 110xxxxx 10xxxxxx

3 | 0000 0800 - 0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx

4 | 0001 0000 - 0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

5 | 0020 0000 - 03FF FFFF | 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

6 | 0400 0000 - 7FFF FFFF | 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

表 1. UTF-8的编码规则

// #txt---end

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(1001110 00100101), 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.  
  
  
**4. Little endian和Big endian**  
  
上一节已经提到, 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.  
// Big Endian(4E25)    Little Endian(254E)  
  
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).  
  
**4.1 计算机怎么知道某一个文件到底采用哪一种方式编码?(零宽度非换行空格(FEFF))**  
  
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.  
// Big Endian(FEFF)    Little Endian(FFFE)  
  
NOTE:  
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.  
  
**5. Unicode与UTF-8之间的转换**  
  
从表1我们很明显可以得知Unicode与UTF-8的关系, 下面以C语言实现两者之间的转换.

1) 将一个字符的Unicode(UCS-2和UCS-4)编码转换成UTF-8编码.

// #c---

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 将一个字符的Unicode(UCS-2和UCS-4)编码转换成UTF-8编码.

\*

\* 参数:

\* unic 字符的Unicode编码值

\* pOutput 指向输出的用于存储UTF8编码值的缓冲区的指针

\* outsize pOutput缓冲的大小

\*

\* 返回值:

\* 返回转换后的字符的UTF8编码所占的字节数, 如果出错则返回 0 .

\*

\* 注意:

\* 1. UTF8没有字节序问题, 但是Unicode有字节序要求;

\* 字节序分为大端(Big Endian)和小端(Little Endian)两种;

\* 在Intel处理器中采用小端法表示, 在此采用小端法表示. (低地址存低位)

\* 2. 请保证 pOutput 缓冲区有最少有 6 字节的空间大小!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int enc\_unicode\_to\_utf8\_one(unsigned long unic, unsigned char \*pOutput,

int outSize)

{

assert(pOutput != NULL);

assert(outSize >= 6);

if ( unic <= 0x0000007F )

{

// \* U-00000000 - U-0000007F: 0xxxxxxx

\*pOutput = (unic & 0x7F);

return 1;

}

e[ls](http://www.linuxso.com/command/ls.html)e if ( unic >= 0x00000080 && unic <= 0x000007FF )

{

// \* U-00000080 - U-000007FF: 110xxxxx 10xxxxxx

\*(pOutput+1) = (unic & 0x3F) | 0x80;

\*pOutput = ((unic >> 6) & 0x1F) | 0xC0;

return 2;

}

else if ( unic >= 0x00000800 && unic <= 0x0000FFFF )

{

// \* U-00000800 - U-0000FFFF: 1110xxxx 10xxxxxx 10xxxxxx

\*(pOutput+2) = (unic & 0x3F) | 0x80;

\*(pOutput+1) = ((unic >> 6) & 0x3F) | 0x80;

\*pOutput = ((unic >> 12) & 0x0F) | 0xE0;

return 3;

}

else if ( unic >= 0x00010000 && unic <= 0x001FFFFF )

{

// \* U-00010000 - U-001FFFFF: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

\*(pOutput+3) = (unic & 0x3F) | 0x80;

\*(pOutput+2) = ((unic >> 6) & 0x3F) | 0x80;

\*(pOutput+1) = ((unic >> 12) & 0x3F) | 0x80;

\*pOutput = ((unic >> 18) & 0x07) | 0xF0;

return 4;

}

else if ( unic >= 0x00200000 && unic <= 0x03FFFFFF )

{

// \* U-00200000 - U-03FFFFFF: 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

\*(pOutput+4) = (unic & 0x3F) | 0x80;

\*(pOutput+3) = ((unic >> 6) & 0x3F) | 0x80;

\*(pOutput+2) = ((unic >> 12) & 0x3F) | 0x80;

\*(pOutput+1) = ((unic >> 18) & 0x3F) | 0x80;

\*pOutput = ((unic >> 24) & 0x03) | 0xF8;

return 5;

}

else if ( unic >= 0x04000000 && unic <= 0x7FFFFFFF )

{

// \* U-04000000 - U-7FFFFFFF: 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

\*(pOutput+5) = (unic & 0x3F) | 0x80;

\*(pOutput+4) = ((unic >> 6) & 0x3F) | 0x80;

\*(pOutput+3) = ((unic >> 12) & 0x3F) | 0x80;

\*(pOutput+2) = ((unic >> 18) & 0x3F) | 0x80;

\*(pOutput+1) = ((unic >> 24) & 0x3F) | 0x80;

\*pOutput = ((unic >> 30) & 0x01) | 0xFC;

return 6;

}

return 0;

}

// #c---end

2) 将一个字符的UTF8编码转换成Unicode(UCS-2和UCS-4)编码.

// #c---

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 将一个字符的UTF8编码转换成Unicode(UCS-2和UCS-4)编码.

\*

\* 参数:

\* pInput 指向输入缓冲区, 以UTF-8编码

\* Unic 指向输出缓冲区, 其保存的数据即是Unicode编码值,

\* 类型为unsigned long .

\*

\* 返回值:

\* 成功则返回该字符的UTF8编码所占用的字节数; 失败则返回0.

\*

\* 注意:

\* 1. UTF8没有字节序问题, 但是Unicode有字节序要求;

\* 字节序分为大端(Big Endian)和小端(Little Endian)两种;

\* 在Intel处理器中采用小端法表示, 在此采用小端法表示. (低地址存低位)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int enc\_utf8\_to\_unicode\_one(const unsigned char\* pInput, unsigned long \*Unic)

{

assert(pInput != NULL && Unic != NULL);

// b1 表示UTF-8编码的pInput中的高字节, b2 表示次高字节, ...

char b1, b2, b3, b4, b5, b6;

\*Unic = 0x0; // 把 \*Unic 初始化为全零

int utfbytes = enc\_get\_utf8\_size(\*pInput);

unsigned char \*pOutput = (unsigned char \*) Unic;

switch ( utfbytes )

{

case 0:

\*pOutput = \*pInput;

utfbytes += 1;

break;

case 2:

b1 = \*pInput;

b2 = \*(pInput + 1);

if ( (b2 & 0xE0) != 0x80 )

return 0;

\*pOutput = (b1 << 6) + (b2 & 0x3F);

\*(pOutput+1) = (b1 >> 2) & 0x07;

break;

case 3:

b1 = \*pInput;

b2 = \*(pInput + 1);

b3 = \*(pInput + 2);

if ( ((b2 & 0xC0) != 0x80) || ((b3 & 0xC0) != 0x80) )

return 0;

\*pOutput = (b2 << 6) + (b3 & 0x3F);

\*(pOutput+1) = (b1 << 4) + ((b2 >> 2) & 0x0F);

break;

case 4:

b1 = \*pInput;

b2 = \*(pInput + 1);

b3 = \*(pInput + 2);

b4 = \*(pInput + 3);

if ( ((b2 & 0xC0) != 0x80) || ((b3 & 0xC0) != 0x80)

|| ((b4 & 0xC0) != 0x80) )

return 0;

\*pOutput = (b3 << 6) + (b4 & 0x3F);

\*(pOutput+1) = (b2 << 4) + ((b3 >> 2) & 0x0F);

\*(pOutput+2) = ((b1 << 2) & 0x1C) + ((b2 >> 4) & 0x03);

break;

case 5:

b1 = \*pInput;

b2 = \*(pInput + 1);

b3 = \*(pInput + 2);

b4 = \*(pInput + 3);

b5 = \*(pInput + 4);

if ( ((b2 & 0xC0) != 0x80) || ((b3 & 0xC0) != 0x80)

|| ((b4 & 0xC0) != 0x80) || ((b5 & 0xC0) != 0x80) )

return 0;

\*pOutput = (b4 << 6) + (b5 & 0x3F);

\*(pOutput+1) = (b3 << 4) + ((b4 >> 2) & 0x0F);

\*(pOutput+2) = (b2 << 2) + ((b3 >> 4) & 0x03);

\*(pOutput+3) = (b1 << 6);

break;

case 6:

b1 = \*pInput;

b2 = \*(pInput + 1);

b3 = \*(pInput + 2);

b4 = \*(pInput + 3);

b5 = \*(pInput + 4);

b6 = \*(pInput + 5);

if ( ((b2 & 0xC0) != 0x80) || ((b3 & 0xC0) != 0x80)

|| ((b4 & 0xC0) != 0x80) || ((b5 & 0xC0) != 0x80)

|| ((b6 & 0xC0) != 0x80) )

return 0;

\*pOutput = (b5 << 6) + (b6 & 0x3F);

\*(pOutput+1) = (b5 << 4) + ((b6 >> 2) & 0x0F);

\*(pOutput+2) = (b3 << 2) + ((b4 >> 4) & 0x03);

\*(pOutput+3) = ((b1 << 6) & 0x40) + (b2 & 0x3F);

break;

default:

return 0;

break;

}

return utfbytes;

}

// #c---end