C++11与Unicode及使用标准库进行UTF-8、UTF-16、UCS2、UCS4/UTF-32编码转换

By 破晓

发表于 2014-02-28

文章目录

Unicode

Unicode是计算机领域的一项行业标准，它对世界上绝大部分的文字的进行整理和统一编码，Unicode的编码空间可以划分为17个平面（plane），每个平面包含2的16次方（65536）个码位。17个平面的码位可表示为从U+0000到U+10FFFF，共计1114112个码位，第一个平面称为基本多语言平面（Basic Multilingual Plane, BMP），或称第零平面（Plane 0）。其他平面称为辅助平面（Supplementary Planes）。基本多语言平面内，从U+D800到U+DFFF之间的码位区段是永久保留不映射到Unicode字符，所以有效码位为1112064个。最新的版本是Unicode 6.3发布于2013年9月30日。

Unicode的编码方式

对于被Unicode收录的字符其编码是唯一且确定的。但是Unicode的实现方式(出于传输、存储、处理或向后兼容的考虑)却有不同的几种，其中最流行的是UTF-8、UTF-16、

UCS2、UCS4/UTF-32等，细分的话还有大小端的区别。

UTF-8(8-bit Unicode Transformation Format)

UTF-8是一种变长编码，对于一个Unicode的字符被编码成1至4个字节。Unicode编码与UTF-8的编码的对应关系如下表

Unicode编码	UTF-8编码(二进制)
U+0000 – U+007F	0xxxxxxx
U+0080 – U+07FF	110xxxxx 10xxxxxx
U+0800 – U+FFFF	1110xxxx 10xxxxxx 10xxxxxx
U+10000 – U+10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

其中绝大部分的中文用三个字节编码，部分中文用四个字节编码，举例如下:

Unicode	字符	UTF-8编码
U+0041	A	0x41
U+7834	破	0xE7 0xA0 0xB4
U+6653	晓	0xE6 0x99 0x93
U+2A6A5	龘(四个龍)	0xF0 0xAA 0x9A 0xA5

优点:

1. 向后兼容ASCII编码；
2. 没有字节序(大小端)的问题适合网络传输；
3. 存储英文和拉丁文等字符非常节省存储空间。

缺点:

1. 变长编码不利于文本处理；
2. 对于CJK等文字比较浪费存储空间。

UTF-16(16-bit Unicode Transformation Format)

UTF-16也是一种变长编码，对于一个Unicode字符被编码成1至2个码元，每个码元为16位。

基本多语言平面（码位范围U+0000-U+FFFF）

在基本多语言平面内的码位UTF-16编码使用1个码元且其值与Unicode是相等的（不需要转换）。举例如下

Unicode	字符	UTF-16（码元）	UTF-16 LE（字节）	UTF-16 BE（字节）
U+0041	A	0x0041	0x41 0x00	0x00 0x41
U+7834	破	0x7834	0x34 0x78	0x78 0x34
U+6653	晓	0x6653	0x53 0x66	0x66 0x53

辅助平面（码位范围U+10000-U+10FFFF）

在辅助平面内的码位在UTF-16中被编码为一对16bit的码元（即32bit,4字节），称作代理对(surrogate pair)。组成代理对的两个码元前一个称为前导代理(lead surrogates)范围为0xD800-0xDBFF，后一个称为后尾代理(trail surrogates)范围为0xDC00-0xDFFF。UTF-16辅助平面代理对与Unicode的对应关系如下表

Lead \ Trail	0xDC00	0xDC01	…	0xDFFF
0xD800	U+10000	U+10001	…	U+103FF
0xD801	U+10400	U+10401	…	U+107FF
⋮	⋮	⋮	⋱	⋮
0xDBFF	U+10FC00	U+10FC01	…	U+10FFFF

举例如下

Unicode	字符	UTF-16（码元）	UTF-16 LE（字节）	UTF-16 BE（字节）
U+2A6A5	龍龍	0xD869 0xDEA5	0x69 0xD8 0xA5 0xDE	0xD8 0x69 0xDE 0xA5

优点:

1. 绝大部分的文字都可以用两个字节编码，对于CJK文字是比较节省空间的；
2. 文本处理比UTF-8方便得多。

缺点:

1. 存储和传输需要考虑字节序的问题；
2. 不兼容ASCII。

UCS2(2-byte Universal Character Set)

UCS2是一种定长编码，编码范围为0x0000-0xFFFF。在基本多语言平面内与UTF-16是等价。UCS2没有类似于UTF-16中代理对的概念，所以对于0xD869 0xDEA5会识别成两个字符。因为是定长编码，所以文本处理很方便。缺点是不能表示全部的Unicode字符。

UCS4(4-byte Universal Character Set)/UTF-32(32-bit Unicode Transformation Format)

UCS4/UTF-32是一种定长编码，使用1个32bit的码元，其值与Unicode编码值相等。举例如下:

Unicode	字符	UTF-32（码元）	UTF-32 LE（字节）	UTF-32 BE（字节）
U+0041	A	0x00000041	0x41 0x00 0x00 0x00	0x00 0x00 0x00 0x41
U+7834	破	0x00007834	0x34 0x78 0x00 0x00	0x00 0x00 0x78 0x34
U+6653	晓	0x00006653	0x53 0x66 0x00 0x00	0x00 0x00 0x66 0x53
U+2A6A5	龘	0x0002A6A5	0xA5 0xA6 0x02 0x00	0x00 0x02 0xA6 0xA5

优点是编码定长容易进行文本处理，缺点是浪费存储空间及存在字节序的问题。

C++11对Unicode的支持

C++11对Unicode提供了语言级别和库级别的支持。

USL(Unicode String Literals)

USL是C++11对Unicode提供的语言级别的支持。在C++11之前C++中有个wchar_t的类型用于存储宽字符（Wide-Character）。你可以写下面这样的代码

```
wchar_t wc = L'中';
wchar_t wc_array[] = L"破晓的博客";
std::wstring wstr = L"破晓的博客";
```

以L开头的字符(或字符串)字面量称为WCSL(Wide-Character String Literals)。本意大概也是用来提供Unicode支持的，可惜标准没有规定这个的实现，wchar_t及其字面量是实现相关的。比如

1. 在windows平台下sizeof(wchar_t)为2，而在linux平台下sizeof(wchar_t)为4；
2. 在windows平台下宽字符(或字符串)字面量使用UTF-16编码，linux平台下使用UTF-32编码。

这导致了下面这段代码在windows下编译时会报错，而在linux下可以编译通过。

```
wchar_t wc = L'龘'; // U+2A6A5
```

C++11新增了char16_t(至少16位)和char32_t(至少32位)以及USL允许下面这样的代码

```
// utf-8
char u8_array[] = u8"破晓的博客";
std::string u8_str = u8"破晓的博客";
// utf-16
char16_t u16_c = u'中';
char16_t u16_array[] = u"破晓的博客";
std::u16string u16_str = u"破晓的博客";
// ucs4
```

```
char32_t u32_c = U'龘';
char32_t u32_array[] = U"破晓的博客";
std::u32string u32_str = U"破晓的博客";
```

上面在字符(或字符串)字面量前面的u8、u及U前缀分别表示这是UTF-8、UTF-16和UCS4编码的字符(或字符串)字面量，用法与L前缀类似。下面是一段测试代码，print_code_uint_sequence函数模板用于输出字符串的码元序列。

```
#include <string>
#include <iostream>
#include <iomanip>
#include <type_traits>

template<typename tStringType, typename tTraits = typename tStringType::traits_ty
void print_code_unit_sequence(tStringType str)
{
  using char_type = typename tTraits::char_type;
  static_assert(std::is_same<char_type, char>::value || std::is_same<char_type, c
  using unsigned_char_type = typename std::make_unsigned<char_type>::type;
  using unsigned_int_type = typename std::make_unsigned<typename tTraits::int_typ
  int w = std::is_same<char, char_type>::value ? 2 : std::is_same<char16_t, char_
  for(auto c : str) {
    auto value = static_cast<unsigned_int_type>(static_cast<unsigned_char_type>(c
    std::cout << "0x" << std::hex << std::uppercase << std::setw(w) << std::setfi
  }
}

int main()
{
  std::string u8_str = u8"龘"; // utf-8
  std::u16string u16_str = u"龘"; // utf-16
  std::u32string u32_str = U"龘"; // ucs4
  std::cout << "utf-8: ";
  print_code_unit_sequence(u8_str);
  std::cout << std::endl;
  std::cout << "utf-16: ";
  print_code_unit_sequence(u16_str);
  std::cout << std::endl;
  std::cout << "ucs4: ";
  print_code_unit_sequence(u32_str);
  std::cout << std::endl;
}
```

输出

```
utf-8: 0xF0 0xAA 0x9A 0xA5
utf-16: 0xD869 0xDEA5
ucs4: 0x0002A6A5
```

使用C++11标准库进行编码转换

C++11标准库在 `<codecvt>` 头文件中定义了3个Unicode编码转换的Facet

Facet	说明

std::codecvt_utf8	封装了UTF-8与UCS2及UTF-8与UCS4的编码转换
std::codecvt_utf16	封装了UTF-16与UCS2及UTF-16与UCS4的编码转换
std::codecvt_utf8_utf16	封装了UTF-8与UTF-16的编码转换

当要转换的字符串为std::basic_string使用`<locale>`头文件中定义的std::wstring_convert类模板会带来极大的方便。

```
template<class Codecvt,
  class Elem = wchar_t,
  class Wide_alloc = std::allocator<Elem>,
  class Byte_alloc = std::allocator<char>>
class wstring_convert;
```

UTF-8与UTF-16编码转换

UTF-8与UTF-16的编码转换使用std::codecvt_utf8_utf16类模板

```
template<class Elem,
  unsigned long Maxcode = 0x10ffff,
  std::codecvt_mode Mode = (std::codecvt_mode)0>
class codecvt_utf8_utf16 : public std::codecvt<Elem, char, std::mbstate_t>;
```

其中Elem用于存储UTF-16码元，可以是char16_t、char32_t或wchar_t（这些类型都至少能够存储16bit）。下面的代码演示了UTF-8到UTF-16和UTF-16到UTF-8的编码转换

```
std::string u8_source_str = u8"破晓的博客"; // utf-8
std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> cvt;
std::u16string u16_cvt_str = cvt.from_bytes(u8_source_str); // utf-8 to utf-16
std::string u8_cvt_str = cvt.to_bytes(u16_cvt_str); // utf-16 to utf-8
std::cout << "u8_source_str = ";
print_code_unit_sequence(u8_source_str);
std::cout << std::endl;
std::cout << "u16_cvt_str = ";
print_code_unit_sequence(u16_cvt_str);
std::cout << std::endl;
std::cout << "u8_cvt_str = ";
print_code_unit_sequence(u8_cvt_str);
std::cout << std::endl;
```

输出

```
u8_source_str = 0xE7 0xA0 0xB4 0xE6 0x99 0x93 0xE7 0x9A 0x84 0xE5 0x8D 0x9A 0xE5 0
u16_cvt_str = 0x7834 0x6653 0x7684 0x535A 0x5BA2
u8_cvt_str = 0xE7 0xA0 0xB4 0xE6 0x99 0x93 0xE7 0x9A 0x84 0xE5 0x8D 0x9A 0xE5 0xAE
```

UTF-8与UCS2编码转换及UTF-8与UCS4编码转换

UTF-8与UCS2或UCS4编码转换使用std::codecvt_utf8类模板

```
template<class Elem,
    unsigned long Maxcode = 0x10ffff,
```

```
    std::codecvt_mode Mode = (std::codecvt_mode)0>
class codecvt_utf8 : public std::codecvt<Elem, char, std::mbstate_t>;
```

当Elem为char16_t时转换为UTF-8与UCS2，当Elem为char32_t时转换为UTF-16与UCS4，当Elem为wchar_t时转换取决于实现。演示如下

```
std::string u8_source_str = u8"破晓的博客"; // utf-8
std::wstring_convert<std::codecvt_utf8<char16_t>, char16_t> utf8_ucs2_cvt;
std::u16string ucs2_cvt_str = utf8_ucs2_cvt.from_bytes(u8_source_str); // utf-8 t
std::string u8_str_from_ucs2 = utf8_ucs2_cvt.to_bytes(ucs2_cvt_str); // ucs2 to u
std::wstring_convert<std::codecvt_utf8<char32_t>, char32_t> utf8_ucs4_cvt;
std::u32string ucs4_cvt_str = utf8_ucs4_cvt.from_bytes(u8_source_str); // utf-8 t
std::string u8_str_from_ucs4 = utf8_ucs4_cvt.to_bytes(ucs4_cvt_str); // ucs4 to u
std::cout << "u8_source_str: ";
print_code_unit_sequence(u8_source_str);
std::cout << std::endl;
std::cout << "ucs2_cvt_str: ";
print_code_unit_sequence(ucs2_cvt_str);
std::cout << std::endl;
std::cout << "u8_str_from_ucs2: ";
print_code_unit_sequence(u8_str_from_ucs2);
std::cout << std::endl;
std::cout << "ucs4_cvt_str: ";
print_code_unit_sequence(ucs4_cvt_str);
std::cout << std::endl;
std::cout << "u8_str_from_ucs4: ";
print_code_unit_sequence(u8_str_from_ucs4);
std::cout << std::endl;
```

输出

```
u8_source_str: 0xE7 0xA0 0xB4 0xE6 0x99 0x93 0xE7 0x9A 0x84 0xE5 0x8D 0x9A 0xE5 0x
ucs2_cvt_str: 0x7834 0x6653 0x7684 0x535A 0x5BA2
u8_str_from_ucs2: 0xE7 0xA0 0xB4 0xE6 0x99 0x93 0xE7 0x9A 0x84 0xE5 0x8D 0x9A 0xE5
ucs4_cvt_str: 0x00007834 0x00006653 0x00007684 0x0000535A 0x00005BA2
u8_str_from_ucs4: 0xE7 0xA0 0xB4 0xE6 0x99 0x93 0xE7 0x9A 0x84 0xE5 0x8D 0x9A 0xE5
```

与UCS2进行转换时需要注意的是，由于UCS2不能表示全部Unicode，所以当向UCS2转换时UCS2无法表示时会抛出std::range_error异常。

UTF-16与UCS2编码转换及UTF-16与UCS4编码转换

UTF-16转UCS2或UCS4使用std::codecvt_utf16类模板

```
template<class Elem,
  unsigned long Maxcode = 0x10ffff,
  std::codecvt_mode Mode = (std::codecvt_mode)0>
class codecvt_utf16 : public std::codecvt<Elem, char, std::mbstate_t>;
```

这里以UTF-16与UCS4的转换为例Elem为char32_t，UTF-16与UCS2的转换类似只是Elem需为char16_t。
由于std::wstring_convert是用于在byte string和wide string之间转换，使用

std::codecvt_utf16时UTF-16字符串作为byte string，因此使用这个转换时就需要考虑字节序的问题。std::codecvt_utf16类模板的第3个模板参数Mode类型为std::codecvt_mode

```
enum codecvt_mode {
  consume_header = 4,
  generate_header = 2,
  little_endian = 1
};
```

这三个枚举值的含义如下表

枚举值	描述
consume_header	告诉codecvt需要处理输入的byte string中的BOM(Byte Order Mark)
generate_header	告诉codecvt在输出的byte string中添加BOM
little_endian	告诉codecvt将byte string的字节序视为小端(Little Endian)，默认为大端(Big Endian)

下面的代码演示了UTF-16 BE和UTF-16 LE编码到UCS4编码的转换

```
std::string u16le_byte_str = "\x34\x78\x53\x66"; // utf-16 Little Endian
std::string u16be_byte_str = "\x78\x34\x66\x53"; // utf-16 Big Endian
std::wstring_convert<std::codecvt_utf16<char32_t, 0x10ffff, std::little_endian>,
std::wstring_convert<std::codecvt_utf16<char32_t>, char32_t> utf16be_cvt; // defa
std::u32string u32_str_from_le = utf16le_cvt.from_bytes(u16le_byte_str); // utf-1
std::u32string u32_str_from_be = utf16be_cvt.from_bytes(u16be_byte_str); // utf-1
std::cout << "u16le_byte_str: ";
print_code_unit_sequence(u16le_byte_str);
std::cout << std::endl;
std::cout << "u16be_byte_str: ";
print_code_unit_sequence(u16be_byte_str);
std::cout << std::endl;
std::cout << "u32_str_from_le: ";
print_code_unit_sequence(u32_str_from_le);
std::cout << std::endl;
std::cout << "u32_str_from_be: ";
print_code_unit_sequence(u32_str_from_be);
std::cout << std::endl;
```

输出如下

```
u16le_byte_str: 0x34 0x78 0x53 0x66
u16be_byte_str: 0x78 0x34 0x66 0x53
u32_str_from_le: 0x00007834 0x00006653
u32_str_from_be: 0x00007834 0x00006653
```

通过设置Mode成功将不同字节序UTF-16编码的字符串进行了正确的转换。

BOM(Byte Order Mark)

字节序标记是插入到以UTF-8、UTF-16或UTF-32编码Unicode文件开头的特殊标记，用于标识文本编码及字节序。

编码	BOM
UTF-8	0xEF 0xBB 0xBF
UTF-16 BE	0xFE 0xFF
UTF-16 LE	0xFF 0xFE
UTF-32 BE	0x00 0x00 0xFE 0xFF
UTF-32 LE	0xFF 0xFE 0x00 0x00

下面的代码演示了通过BOM指示UTF-16编码字符串的字节序，codecvt的第3个参数需设置为std::consume_header

```
std::string u16le_byte_str = "\xff\xfe\x34\x78\x53\x66"; // utf-16 little endian
std::string u16be_byte_str = "\xfe\xff\x78\x34\x66\x53"; // utf-16 big endian wit
std::wstring_convert<std::codecvt_utf16<char32_t, 0x10ffff, std::consume_header>,
std::wstring_convert<std::codecvt_utf16<char32_t, 0x10ffff, std::consume_header>,
std::u32string u32_str_from_le = utf16le_cvt.from_bytes(u16le_byte_str); // utf-1
std::u32string u32_str_from_be = utf16be_cvt.from_bytes(u16be_byte_str); // utf-1
std::cout << "u16le_byte_str: ";
print_code_unit_sequence(u16le_byte_str);
std::cout << std::endl;
std::cout << "u16be_byte_str: ";
print_code_unit_sequence(u16be_byte_str);
std::cout << std::endl;
std::cout << "u32_str_from_le: ";
print_code_unit_sequence(u32_str_from_le);
std::cout << std::endl;
std::cout << "u32_str_from_be: ";
print_code_unit_sequence(u32_str_from_be);
std::cout << std::endl;
```

输出

```
u16le_byte_str: 0xFF 0xFE 0x34 0x78 0x53 0x66
u16be_byte_str: 0xFE 0xFF 0x78 0x34 0x66 0x53
u32_str_from_le: 0x00003478 0x00005366
u32_str_from_be: 0x00007834 0x00006653
```

当UCS4转UTF-16时输出为byte string，可以通过设置Mode为std::generate_header来使输出带BOM，下面的代码通过UCS4转UTF-16 LE和UTF-16 BE演示了std::generate_header的使用

```
std::u32string u32_str = U"破晓"; // ucs4
std::wstring_convert<std::codecvt_utf16<char32_t, 0x10ffff, static_cast<std::code
std::wstring_convert<std::codecvt_utf16<char32_t, 0x10ffff, static_cast<std::code
std::string u16le_byte_str = utf16le_cvt.to_bytes(u32_str); // ucs4 to utf-16 lit
std::string u16be_byte_str = utf16be_cvt.to_bytes(u32_str); // ucs4 to utf-16 big
std::cout << "u32_str: ";
print_code_unit_sequence(u32_str);
std::cout << std::endl;
std::cout << "u16le_byte_str: ";
print_code_unit_sequence(u16le_byte_str);
```