彻底解密C++宽字符

作者：龙飞
2010/6/26

1、从char到wchar_t

“这个问题比你想象中复杂”
（我也学下BS的风格，虽然这句话是我自己临时想说的。^^）

从字符到整数

char是一种整数类型，这句话的含义是，char所能表示的字符在C/C++中都是整数类型。好，接下来，很多文章就会举出一个典型例子，比如，'a'的数值就是0x61。这种说法对吗？如果你细心的读过K&R和BS对于C和C++描述的原著，你就会马上反驳道，0x61只是'a'的ASCII值，并没有任何规定C/C++的char值必须对应ASCII。C/C++甚至没有规定char占几位，只是规定了sizeof(char)等于1。
当然，目前大部分情况下，char是8位的，并且，在ASCII范围内的值，与ASCII对应。

本地化策略集（locale）

“将'a'翻译成0x61的整数值”，“将ASCII范围内的编码与char的整数值对应起来”，类似这样的规定，是特定系统和特定编译器制定的，C/C++中有个特定的名词来描述这种规定的集合：本地化策略集（locale。也有翻译成“现场”）。而翻译——也就是代码转换（codecvt）只是这个集合中的一个，C++中定义为策略（facet。也有翻译为“刻面”）

C/C++的编译策略

“本地化策略集”是个很好的概念，可惜在字符和字符串这个层面上，C/C++并不使用（C++的locale通常只是影响流（stream）），C/C++使用更直接简单的策略：硬编码。
简单的说，字符（串）在程序文件（可执行文件，非源文件）中的表示，与在程序执行中在内存中的表示一致。考虑两种情况：
A、char c = 0x61;
B、char c = 'a';
情况A下，编译器可以直接认识作为整数的c，但是在情况B下，编译器必须将'a'翻译成整数。编译器的策略也很简单，就是直接读取字符（串）在源文件中的编码数值。比如：
const char* s = "中文abc";
这段字符串在GB2312（Windows 936），也就是我们的windows默认中文系统源文件中的编码为：
0xD60xD00xCE0xC4 0x61 0x62 0x63
在UTF-8，也就是Linux默认系统源文件中的编码为：
0xE4 0xB8 0xAD 0xE6 0x96 0x87 0x61 0x62 0x63
一般情况下，编译器会忠实于源文件的编码为s赋值，例外的情况比如VC会自作聪明的把

大部分其他类型编码的字符串转换成 GB2312（除了像 UTF-8 without signature 这样的幸存者）。
程序在执行的时候，s 也就保持是这样的编码，不会再做其他的转换。

宽字符 wchar_t

正如 char 没有规定大小，wchar_t 同样没有标准限定，标准只是要求一个 wchar_t 可以表示任何系统所能认识的字符，在 win32 中，wchar_t 为 16 位；Linux 中是 32 位。wchar_t 同样没有规定编码，因为 Unicode 的概念我们后面才解释，所以这里只是提一下，在 win32 中，wchar_t 的编码是 UCS-2BE；而 Linux 中是 UTF-32BE（等价于 UCS-4BE），不过简单的说，在 16 位以内，一个字符的这 3 种编码值是一样的。因此：

```
const wchar_t* ws = L"中文 abc";
```

的编码分别为：

```
0x4E2D  0x6587  0x0061  0x0062  0x0063                    //win32，16 位
0x00004E2D  0x00006587  0x00000061  0x00000062  0x00000063    //Linux，32 位
```

大写的 L 是告诉编译器：这是宽字符串。所以，这时候是需要编译器根据 locale 来进行翻译的。
比如，在 Windows 环境中，编译器的翻译策略是 GB2312 到 UCS-2BE；Linux 环境中的策略是 UTF-8 到 UTF-32BE。
这时候就要求源文件的编码与编译器的本地化策略集中代码翻译的策略一致，例如 VC 只能读取 GB2312 的源代码（这里还是例外，VC 太自作聪明了 ，会将很多其他代码在编译时自动转换成 GB2312），而 gcc 只能读取 UTF-8 的源代码（这里就有个尴尬，MinGW 运行 win32 下，所以只有 GB2312 系统才认；而 MinGW 却用 gcc 编写，所以自己只认 UTF-8，所以结果就是，MinGW 的宽字符被废掉了）。
宽字符（串）由编译器翻译，还是被硬编码进程序文件中。

2、Unicode 和 UTF

Unicode 和 UCS

Unicode 和 UCS 是两个独立的组织分别制定的一套编码标准，但是因为历史的原因，这两套标准是完全一样的。Unicode 这个词用得比较多的原因可能是因为比较容易记住，如果没有特别的声明，在本文所提及的 Unicode 和 UCS 就是一个意思。Unicode 的目标是建立一套可以包含人类所有语言文字符号你想得到想不到的各种东西的编码，其编码容量甚至预留了火星语以及银河系以外语言的空间——开个玩笑，反正简单的说，Unicode 编码集足够的大，如果用计算机单位来表示，其数量比 3 个字节大一些，不到 4 个字节。

Unicode 和 UTF

因为 Unicode 包含的内容太多，其编码在计算机中的表示方法就成为了一个有必要研究的问题。传统编码，比如标准的 7 位 ASCII，在计算机中的表示方法就是占一个字节的后 7 位，这似乎是不需要解释就符合大家习惯的表示方法。但是当今 Unicode 的总数达到 32 位（计算机的最小单位是字节，所以大于 3 字节，就只能至少用 4 字节表示），对于大部分常用字符，比如 Unicode 编码只占一个字节大小的英语字母，占两个字节大小汉字，都用 4 个字节来储存太奢侈了。另外，如果都用 4 字节直接表示，就不可避免的出现为 0 的字节。而

我们知道，在C语言中，0x00的字节就是'\0'，表示的是一个字符串（char字符串，非wchar_t串）的结束，换句话说，C风格的char字符串无法表示Unicode。
因为类似的种种问题，为Unicode在计算机中的编码方法出现了，这就是UTF；所对应的，为UCS编码实现的方式也有自己的说法。一般来说，UTF-x，x表示这套编码一个单位至少占用x位，因为Unicode最长达到32位，所以UTF-x通常是变长的——除了UTF-32；而UCS-y表示一个单位就占用y个字节，所以能表示当今Unicode的UCS-y只有UCS-4，但是因为历史的原因，当Unicode还没那么庞大的时候，2个字节足够表示，所以有UCS-2，现在看来，UCS-2所能表示的Unicode部分只是当今Unicode的一个子集。
也就是说，如果某种编码，能根据一定的规则算法，得到Unicode编码，那么这种编码方式就可以称之为UTF。

UTF-8和Windows GB2312

UTF-8是一套“聪明”的编码，可能用1，2，3，4个字节表示。通过UTF-8的算法，每一个字节表示的信息都很明确：这是不是某个Unicode编码的第一个字节；如果是第一个字节，这是一个几位Unicode编码。这种“聪明”被称为UTF-8的自我同步，也是UTF-8成为网络传输标准编码的原因。
另外，UTF-8也不会出现0字节，所以可以表示为char字符串，所以可以成为系统的编码。Linux系统默认使用UTF-8编码。
Windows GB2312一般自称为GB2312，其实真正的名字应该是Windows Codepage 936，这也是一种变长的编码：1个字节表示传统的ASCII部分；汉字部分是两个字节的GBK（国标扩（展），拼音声母）。Codepage 936也可以表示为char字符串，是简体中文Windows系统的默认编码。
我们在第1节中看到的
```
const char* s = "中文abc";
```
在Windows中的编码就是Codepage 936；在Linux中的编码就是UTF-8。
需要注意的是，Codepage 936不像UTF，跟Unicode没有换算的关系，所以只能通过“代码页”技术查表对应。

UTF-16和UCS-2

UTF-16用2个字节或者4个字节表示。在2个字节大小的时候，跟UCS-2是一样的。UTF-16不像UTF-8，没有自我同步机制，所以，编码大位在前还是小位在前，就成了见仁见智的问题。我们在第1节中，“中”的UCS-2BE（因为是两个字节，所以也等价于UTF-16BE）编码是0x4E2D，这里的BE就是大位在后的意思（也就是小位在前了），对应的，如果是UCS-2LE，编码就成了0x2D4E。
Windows中的wchar_t就是采用UCS-2BE编码。需要指出的是，C++标准中对wchar_t的要求是要能表示所有系统能识别的字符。Windows自称支持Unicode，但是其wchar_t却不能表示所有的Unicode，由此违背了C++标准。

UTF-32和UCS-4

UTF-32在目前阶段等价于UCS-4，都用定长的4个字节表示。UTF-32同样存在BE和LE的问题。Linux的wchar_t编码就是UTF-32BE。在16位以内的时候，UTF-32BE的后两位（前两位

是 0x00 0x00）等价于 UTF-16BE 也就等价于 UCS-2BE

BOM

为了说明一个文件采用的是什么编码，在文件最开始的部分，可以有 BOM，比如 0xFE 0xFF 表示 UTF-16BE，0xFF 0xFE 0x00 0x00 表示 UTF-32LE。UTF-8 原本是不需要 BOM 的，因为其自我同步的特性，但是为了明确说明这是 UTF-8（而不是让文本编辑器去猜），也可以加上 UTF-8 的 BOM：0xEF 0xBB 0xBF 。

以上内容都讲述得很概略，详细信息请查阅维基百科相关内容。

3、利用 C 运行时库函数转换

std::locale

通过前面两节的知识，我们知道了在 C/C++中，字符（串）和宽字符（串）之间的转换不是简单的，固定的数学关系，宽窄转换依赖于本地化策略集（locale）。换句话说，一个程序在运行之前并不知道系统的本地化策略集是什么，程序只有在运行之后才通过 locale 获得当时的本地化策略集。
C 有自己的 locale 函数，我们这里直接介绍 C++的 locale 类。
先讨论 locale 的构造函数：

```
locale() throw();
```

这个构造函数是获得当前程序的 locale，用法如下：

```
std::locale app_loc = std::locale(); 或者 std::locale app_loc;
```

（这是构造对象的两种表示方式，后同）
另外一个构造函数是：

```
explicit locale(const char* name);
```

这个构造函数以 name 的名字创建新的 locale。重要的 locale 对象有：

```
std::locale sys_loc("");
//获得当前系统环境的 locale
std::locale C_loc("C"); 或者 std::locale C_loc = std::locale::classic();
//获得 C 定义 locale
std::locale old_loc = std::locale::global(new_loc);
//将 new_loc 设置为当前全局 locale，并将原来的 locale 返回给 old_loc
```

除了这些，其它的 name 具体名字依赖于 C++编译器和操作系统，比如 Linux 下 gcc 中文系统的 locale 名字为"zh_CN.UTF-8"，中文 Windows 可以用"chs"（更加完整的名字可以用 name()函数查看）。

mbstowcs()和 wcstombs()

这两个 C 运行时库函数依赖于全局 locale 进行转换，所以，使用前必须先设置全局 locale。
std::locale 已经包含在<iostream>中了，再加上我们需要用到的 C++字符串，所以包含 <string>。

我们先看窄到宽的转换函数：

```
const std::wstring s2ws(const std::string& s)
{
    std::locale old_loc =
        std::locale::global(std::locale(""));

    const char* src_str = s.c_str();
    const size_t buffer_size = s.size() + 1;
    wchar_t* dst_wstr = new wchar_t[buffer_size];
    wmemset(dst_wstr, 0, buffer_size);
    mbstowcs(dst_wstr, src_str, buffer_size);
    std::wstring result = dst_wstr;
    delete []dst_wstr;
    std::locale::global(old_loc);
    return result;
}
```

我们将全局 locale 设置为系统 locale，并保存原来的全局 locale 在 old_loc 中。
在制定转换空间缓存大小的时候，考虑如下：char 是用 1 个或多个对象，也就是 1 个或者多个字节来表示各种符号：比如，GB2312 用 1 个字节表示数字和字母，2 个字节表示汉字；UTF-8 用一个字节表示数字和字母，3 个字节表示汉字，4 个字节表示一些很少用到的符号，比如音乐中 G 大调符号等。wchar_t 是用 1 个对象（2 字节或者 4 字节）来表示各种符号。因此，表示同样的字符串，宽字符串的大小（也就是 wchar_t 对象的数量）总是小于或者等于窄字符串大小（char 对象数量）的。+1 是为了在最后预留一个值为 0 的对象，以便让 C 风格的 char 或者 wchar_t 字符串自动截断——这当然是宽串大小等于窄串大小的时候才会用上的，大部分时候，字符串早在前面某个转换完毕的位置就被 0 值对象所截断了。
最后我们将全局 locale 设置回原来的 old_loc。
窄串到宽串的转换函数：

```
const std::string ws2s(const std::wstring& ws)
{
    std::locale old_loc =
        std::locale::global(std::locale(""));

    const wchar_t* src_wstr = ws.c_str();
    size_t buffer_size = ws.size() * 4 + 1;
    char* dst_str = new char[buffer_size];
    memset(dst_str, 0, buffer_size);
    wcstombs(dst_str ,src_wstr, buffer_size);
    std::string result = dst_str;
    delete []dst_str;
    std::locale::global(old_loc);
    return result;
}
```

4、利用 codecvt 和 use_facet 转换

locale 和 facet

C++的 locale 框架比 C 更完备。C++除了一个笼统本地策略集 locale，还可以为 locale 指定具体的策略 facet，甚至可以用自己定义的 facet 去改造一个现有的 locale 产生一个新的 locale。如果有一个 facet 类 NewFacet 需要添加到某个 old_loc 中形成新 new_loc，需要另外一个构造函数，通常的做法是：

```
std::locale new_loc(old_loc, new NewFacet);
```

标准库里的标准 facet 都具有自己特有的功能，访问一个 locale 对象中特定的 facet 需要使用模板函数 use_facet：

```
template <class Facet> const Facet& use_factet(const locale&);
```

换一种说法，use_facet 把一个 facet 类实例化成了对象，由此就可以使用这个 facet 对象的成员函数。

codecvt

codecvt 就是一个标准 facet。在 C++的设计框架里，这是一个通用的代码转换模板——也就是说，并不是仅仅为宽窄转换制定的。

```
templat <class I, class E, class State> class std::codecvt: public locale, public codecvt_base{...};
```

I 表示内部编码，E 表示外部编码，State 是不同转换方式的标识，如果定义如下类型：

```
typedef std::codecvt<wchar_t, char, mbstate_t> CodecvtFacet;
```

那么 CodecvtFacet 就是一个标准的宽窄转换 facet，其中 mbstate_t 是标准宽窄转换的 State。

内部编码和外部编码

我们考虑第 1 节中提到的 C++编译器读取源文件时候的情形，当读到 L"中文 abc"的时候，外部编码，也就是源文件的编码，是 GB2312 或者 UTF-8 的 char，而编译器必须将其翻译为 UCS-2BE 或者 UTF-32BE 的 wchar_t，这也就是程序的内部编码。如果不是宽字符串，内外编码都是 char，也就不需要转换了。类似的，当 C++读写文件的时候 ，就会可能需要到内外编码转换。事实上，codecvt 就正是被文件流缓存 basic_filebuf 所使用的。理解这一点很重要，原因会在下一小节看到。

CodecvtFacet 的 in()和 out()

因为在 CodecvtFacet 中，内部编码设置为 wchar_t，外部编码设置为 char，转换模式是标准宽窄转换 mbstate_t，所以，类方法 in()就是从 char 标准转换到 wchar_t，out()就是从 wchar_t 标准转换到 char。这就成了我们正需要的内外转换函数。

```
result in(State& s,
        const E* from, const E* from_end, const E*& from_next,
        I* to,  I* to_end, I*& to_next) const;
```

```
result out(State& s,
        const I* from, const I* from_end, const I*& from_next,
        E* to, E* to_end, E*& to_next) const;
```

其中，s 是非 const 引用，保存着转换位移状态信息。这里需要重点强调的是，因为转换的实际工作交给了运行时库，也就是说，转换可能不是在程序的主进程中完成的，而转换工作依赖于查询 s 的值，因此，如果 s 在转换结束前析构，就可能抛出运行时异常。所以，最安全的办法是，将 s 设置为全局变量！

const 的 3 个指针分别是待转换字符串的起点，终点，和出现错误时候的停点（的下一个位置）；另外 3 个指针是转换目标字符串的起点，终点以及出现错误时候的停点（的下一个位置）。

代码如下：

头文件

```
//Filename string_wstring_cppcvt.hpp

#ifndef STRING_WSTRING_CPPCVT_HPP
#define STRING_WSTRING_CPPCVT_HPP

#include <iostream>
#include <string>

const std::wstring s2ws(const std::string& s);
const std::string ws2s(const std::wstring& s);

#endif
```

```
#include "string_wstring_cppcvt.hpp"

mbstate_t in_cvt_state;
mbstate_t out_cvt_state;

const std::wstring s2ws(const std::string& s)
{
    std::locale sys_loc("");

    const char* src_str = s.c_str();
    const size_t BUFFER_SIZE = s.size() + 1;

    wchar_t* intern_buffer = new wchar_t[BUFFER_SIZE];
    wmemset(intern_buffer, 0, BUFFER_SIZE);

    const char* extern_from = src_str;
    const char* extern_from_end = extern_from + s.size();
    const char* extern_from_next = 0;
    wchar_t* intern_to = intern_buffer;
    wchar_t* intern_to_end = intern_to + BUFFER_SIZE;
    wchar_t* intern_to_next = 0;
```

```
    typedef std::codecvt<wchar_t, char, mbstate_t> CodecvtFacet;

    CodecvtFacet::result cvt_rst =
        std::use_facet<CodecvtFacet>(sys_loc).in(
            in_cvt_state,
            extern_from, extern_from_end, extern_from_next,
            intern_to, intern_to_end, intern_to_next);
    if (cvt_rst != CodecvtFacet::ok) {
        switch(cvt_rst) {
            case CodecvtFacet::partial:
                std::cerr << "partial";
                break;
            case CodecvtFacet::error:
                std::cerr << "error";
                break;
            case CodecvtFacet::noconv:
                std::cerr << "noconv";
                break;
            default:
                std::cerr << "unknown";
        }
        std::cerr<< ", please check in_cvt_state."
                    << std::endl;
    }
    std::wstring result = intern_buffer;

    delete []intern_buffer;

    return result;
}

const std::string ws2s(const std::wstring& ws)
{
    std::locale sys_loc("");

    const wchar_t* src_wstr = ws.c_str();
    const size_t MAX_UNICODE_BYTES = 4;
    const size_t BUFFER_SIZE =
                ws.size() * MAX_UNICODE_BYTES + 1;

    char* extern_buffer = new char[BUFFER_SIZE];
    memset(extern_buffer, 0, BUFFER_SIZE);
```

```
    const wchar_t* intern_from = src_wstr;
    const wchar_t* intern_from_end = intern_from + ws.size();
    const wchar_t* intern_from_next = 0;
    char* extern_to = extern_buffer;
    char* extern_to_end = extern_to + BUFFER_SIZE;
    char* extern_to_next = 0;

    typedef std::codecvt<wchar_t, char, mbstate_t> CodecvtFacet;

    CodecvtFacet::result cvt_rst =
        std::use_facet<CodecvtFacet>(sys_loc).out(
            out_cvt_state,
            intern_from, intern_from_end, intern_from_next,
            extern_to, extern_to_end, extern_to_next);
    if (cvt_rst != CodecvtFacet::ok) {
        switch(cvt_rst) {
            case CodecvtFacet::partial:
                std::cerr << "partial";
                break;
            case CodecvtFacet::error:
                std::cerr << "error";
                break;
            case CodecvtFacet::noconv:
                std::cerr << "noconv";
                break;
            default:
                std::cerr << "unknown";
        }
        std::cerr<< ", please check out_cvt_state."
                    << std::endl;
    }
    std::string result = extern_buffer;

    delete []extern_buffer;

    return result;
}
```

最后补充说明一下：

std::use_facet<CodecvtFacet>(sys_loc).in() 和 std::use_facet<CodecvtFacet>(sys_loc).out()。sys_loc 是系统的 locale，这个 locale 中就包含着特定的 codecvt facet，我们已经 typedef 为了 CodecvtFacet。用 use_facet 对 CodecvtFacet 进行了实例化，所以可以使用这个 facet 的方法 in()和 out()。

5、利用 fstream 转换

C++的流和本地化策略集

BS 在设计 C++流的时候希望其具备智能化，并且是可扩展的智能化，也就是说，C++的流可以“读懂”一些内容。比如：

std::cout << 123 << "ok" << std::endl;

这句代码中，std::cout 是能判断出 123 是 int 而"ok"是 const char[3]。利用流的智能，甚

至可以做一些基础类型的转换，比如从 int 到 string，string 到 int：

```
std::string str("123");
std::stringstream sstr(str);
int i;
sstr >> i;int i = 123;
std::stringstream sstr;
sstr << i;
std::string str = sstr.str();
```

尽管如此，C++并不满足，C++甚至希望流能“明白”时间，货币的表示法。而时间和货币的表示方法在世界范围内是不同的，所以，每一个流都有自己的 locale 在影响其行为，C++中叫做激活（imbue，也有翻译成浸染）。而我们知道，每一个 locale 都有多个 facet，这些 facet 并非总是被 use_facet 使用的。决定使用哪些 facet 的，是流的缓存 basic_streambuf 及其派生类 basic_stringbuf 和 basic_filebuf。我们要用到的 facet 是 codecvt，这个 facet 只被 basic_filebuf 使用——这就是为什么只能用 fstream 来实现宽窄转换，而无法使用 sstream 来实现的原因。

头文件：

```
//filename string_wstring_fstream.hpp
#ifndef STRING_WSTRING_FSTREAM_HPP
#define STRING_WSTRING_FSTREAM_HPP

#include <string>

const std::wstring s2ws(const std::string& s);
const std::string ws2s(const std::wstring& s);

#endif
```

实现：

```
#include <string>
#include <fstream>
#include "string_wstring_fstream.hpp"

const std::wstring s2ws(const std::string& s)
{
    std::locale sys_loc("");
```

```
    std::ofstream ofs("cvt_buf");
    ofs << s;
    ofs.close();

    std::wifstream wifs("cvt_buf");
    wifs.imbue(sys_loc);
    std::wstring wstr;
    wifs >> wstr;
    wifs.close();

    return wstr;
}

const std::string ws2s(const std::wstring& s)
{
    std::locale sys_loc("");

    std::wofstream wofs("cvt_buf");
    wofs.imbue(sys_loc);
    wofs << s;
    wofs.close();

    std::ifstream ifs("cvt_buf");
    std::string str;
    ifs >> str;
    ifs.close();

    return str;
}
```

在窄到宽的转化中，我们先使用默认的本地化策略集（locale）将 s 通过窄文件流 ofs 传入文件，这是 char 到 char 的传递，没有任何转换；然后我们打开宽文件流 wifs，并用系统的本地化策略集（locale）去激活（imbue）之，流在读回宽串 wstr 的时候，就是 char 到 wchar_t 的转换，并且因为激活了 sys_loc，所以实现标准窄到宽的转换。
在宽到窄的转化中，我们先打开的是宽文件流 wofs，并且用系统的本地化策略集 sys_loc 激活（imbue）之，这时候，因为要写的文件 cvt_buf 是一个外部编码，所以执行了从 wchar_t 到 char 的标准转换。读回来的文件流从 char 到 char，不做任何转换。

6、国际化策略

硬编码的硬伤

我们现在知道，C/C++的宽窄转换是依赖系统的 locale 的，并且在运行时完成。考虑这样一种情况，我们在简体中文 Windows 下编译如下语句：
`const char* s = "中文 abc";`
根据我们之前的讨论，编译器将按照 Windows Codepage936（GB2312）对这个字符串进行编码。如果我们在程序中运行宽窄转换函数，将 s 转换为宽字符串 ws，如果这个程序运行在简体中文环境下是没问题的，将执行从 GB2312 到 UCS-2BE 的转换；但是，如果在其他语言环境下，比如是繁体中文 BIG5，程序将根据系统的 locale 执行从 BIG5 到 UCS-2BE 的转换，这显然就出现了错误。

补救

有没有补救这个问题的办法呢？一个解决方案就是执行不依赖 locale 的宽窄转换。实际上，这就已经不是宽窄转换之间的问题了，而是编码之间转换的问题了。我们可以用 GNU 的 libiconv 实现任意编码间的转换，对于以上的具体情况，指明是从 GB2312 到 UCS-2BE 就不会出错。（**附 A**：libiconv 简介），但这显然是一个笨拙的策略：我们在简体中文 Windows 下必须使用 GB2312 到 UCS-2BE 版本的宽窄转换函数；到了 BIG5 环境下，就必须重新写从 BIG5 到 UCS-2BE 的宽窄转换函数。

Windows 的策略

Windows 的策略是淘汰了窄字符串，干脆只用宽字符串。所有的硬编码全部加上特定宏，比如 TEXT()，如果程序是所谓 Unicode 编译，在编译时就翻译为 UCS2-BE——Windows 自称为 Unicode 编程，其本质是使用了 UCS-2BE 的 16 位宽字符串。

Linux 的策略

Linux 下根本就不存在这个问题！因为各种语言的 Linux 都使用 UTF-8 的编码，所以，无论系统 locale 如何变化，窄到宽转换的规则一直是 UTF-8 到 UTF32-BE 。

跨平台策略

因为在 16 位的范围内，UTF32-BE 的前 16 位为 0，后 16 位与 UCS2-BE 是一样的，所以，即使 wchar_t 的 sizeof()不一样，在一般情况下，跨平台使用宽字符（串）也应该是兼容的。但是依然存在潜在的问题，就是那些 4 字节的 UTF32 编码。

gettext 策略

以上都是将 ASCII 及以外的编码硬编码在程序中的办法。GNU 的 gettext 提供了另外一种选择：在程序中只硬编码 ASCII，多语言支持由 gettext 函数库在运行时加载。（对 gettext 的介绍请参考**附 B**：GetText 简介）。gettext 的多语言翻译文件不在程序中，而是单独的提出来放在特定的位置。gettext 明确的知道这些翻译文件的编码，所以可以准确的告诉给系统翻译的正确信息，而系统将这些信息以当前的系统 locale 编码成窄字符串反馈给程序。例如，在简体中文 Windows 中，gettext 的 po 文件也可以以 UTF-8 储存，gettext 将 po 文件翻译成 mo 文件，确保 mo 文件在任何系统和语言环境下都能够正确翻译。在运行是传给 win32 程序的窄串符合当前 locale，是 GB2312。gettext 让国际化的翻译更加的方便，缺点是目前我没找到支持宽字符串的版本（据说是有 ugettext()支持宽字符串），所以要使用 gettext 只能使用窄字符串。但是 gettext 可以转换到宽字符串，而且不会出现宽窄转换的问题，因为 gettext 是运行时根据 locale 翻译的。例如：

```
const char* s = gettext("Chinese a b c");
```

其中"Chinese a b c"在 po 中的翻译是"中文 abc"

使用依赖 locale 的运行时宽窄转换函数：

```
const std::wstring wstr = s2ws(s);
```

运行时调用该 po 文件对应的 mo 文件，在简体中文环境下就以 GB2312 传给程序，在繁体中文中就以 BIG5 传给程序，这样 s2ws()总能够正常换算编码。

更多

在本文的最后，我想回到 C++的 stream 问题上。用 fstream 转换如此的简单，sstream 却不支持。改造一个支持 codecvt 的 string stream 需要改造 basic_stringbuf。basic_stringbuf 和 basic_filebuf 都派生自 basic_streambuf，所不同的是 basic_filebuf 在构造和 open()的时候调用了 codecvt，只需要在 basic_stringbuf 中添加这个功能就可以了。说起来容易，实际上是需要重新改造一个 STL 模板，尽管这些模板源代码都是在标准库头文件中现成的，但是我还是水平有限，没有去深究了。另外一个思路是构建一个基于内存映射的虚拟文件，这个框架在 boost 的 iostreams 库中，有兴趣的朋友可以深入的研究。
（完）

附 A：libiconv 简介

GNU 的 libiconv 项目
http://www.gnu.org/software/libiconv/
libiconv 支持许多字符集，包括我们将用到的 GB2312，UTF-8 和 UCS-2（Unicode）。具体的，在项目主页上有详细的说明。我们需要新学习直接用的类容并不繁多，同样的，如果你没什么兴趣自己编译源代码，可以直接用在 win32 下编译好的头文件，库和动态链接库（DLL）。win32 下的项目主页是：
http://gettext.sourceforge.net/
有趣的是，作者把它作为了我们前面提到的 gettext 的一部分。在下载页面上，我们直接选择 libiconv-win32，同样的，我直接给出所需要的三部分文件的相关信息：
iconv.h：头文件，请在 C++代码中#include 进来；
iconv.lib：库文件，在编译时候使用；
iconv.dll：动态链接库，请放到 exe 文件能找到的路径下（通常与 exe 在同一文件夹下面）

下面的程序演示在 win32 下，GB2312 到 UCS-2BE 的转化。

头文件：

```
//Filename iconv_class.hpp
#ifndef ICONV_CLASS_HPP
#define ICONV_CLASS_HPP

#include <iostream>
#include <string>
#include <vector>

#include <iconv.h>

class GB2312toUCS2{
```

```
private:
    std::string GB2312String;
    std::vector<char> UCS2CharVector;
public:
    GB2312toUCS2(const std::string& gb2312_string);
    ~GB2312toUCS2();
    const std::vector<char>& getUCS2() const;
};

#endif
```

实现：

```
//Filename iconv_class.cpp
#include "iconv_class.hpp"

GB2312toUCS2::GB2312toUCS2(const std::string& gb2312_string):
GB2312String(gb2312_string)
{
    const size_t destBufferSize = GB2312String.size() * 2;
    char* destBuffer = new char[destBufferSize];
    memset(destBuffer, 0, destBufferSize);

    const char* src = GB2312String.c_str();
    size_t src_len_left = strlen(src);
    size_t dst_len_left = destBufferSize;

    const char* p_in = src;
    char* p_out = destBuffer;

    iconv_t cd = iconv_open("UCS-2BE", "GBK");
    if (cd == (iconv_t)(-1)) {
        std::cerr << "iconv_open() failed." << std::endl;
        exit(1);
    }
    size_t non_reversible_char_size = iconv(cd,
                                        &p_in, &src_len_left,
                                        &p_out, &dst_len_left);
    if (non_reversible_char_size == (size_t)(-1)) {
        std::cerr << "iconv() failed." << std::endl;
        exit(1);
    }
    iconv_close(cd);
```

```
    const size_t usc2_vector_size =
        destBufferSize - dst_len_left;
    for (size_t i = 0; i < usc2_vector_size; i++) {
        UCS2CharVector.push_back(*(destBuffer + i));
    }

    delete []destBuffer;
}

GB2312toUCS2::~GB2312toUCS2()
{}

const std::vector<char>& GB2312toUCS2::getUCS2() const
{
    return UCS2CharVector;
}
```

演示程序：

```
#include <iomanip>
#include "iconv_class.hpp"

int main(int argc, char* argv[])
{
    std::string test_str = "中文abc";

    GB2312toUCS2* my_test = new GB2312toUCS2(test_str);
    std::vector<char> ucs_charv = my_test->getUCS2();
    delete my_test;

    std::cout << std::hex;
    for (std::vector<char>::const_iterator c_iter = ucs_charv.begin();
        c_iter != ucs_charv.end(); c_iter++) {
        std::cout << (int)(unsigned char)(*c_iter) << "\t";
    }
    std::cout << std::dec << std::endl;

    const wchar_t* w_str = L"中文abc";
    std::cout << wcslen(w_str) << std::endl;
    std::cout << std::hex;
    for (size_t i = 0; i < wcslen(w_str); i++) {
        std::cout << (unsigned short)(*(w_str+i)) << "\t";
    }
    std::cout << std::dec << std::endl;
```

```
    const char* m_str = "中文abc";
    std::cout << strlen(m_str) << std::endl;
    std::cout << std::hex;
    for (size_t i = 0; i < strlen(m_str); i++) {
        std::cout << (int)(unsigned char)(*(m_str+i)) << "\t";
    }
    std::cout << std::dec << std::endl;

    return 0;
}
```

附 B：GetText 简介

po，mo 与 gettext

线索从 Wesnoth 的发布游戏与源代码中开始，我们知道，在 Wesnoth 游戏中，有个名为 po 的文件夹，多国语言翻译都放在了这个文件夹下面。游戏程序中多为*.mo 文件，源代码中多为*.po 文件。通过搜索，po 与 mo 的背景浮出水面——它们来自 GNU 项目 gettext。
gettext 项目是专门为多语言设计的。我们不需要修改源代码和程序的情况下，可以让程序支持多国语言。程序将根据系统所在的国家和区域选择相应的语言，当然，也可以在执行过程中让玩家自由的选择。既然是开放源代码的，自然也很容易的被移植到 win32 下。win32 下的这个项目主页如下：
http://gnuwin32.sourceforge.net/packages/gettext.htm
为了方便的使用，我还是建议你下载完整的安装包（Complete package）。然后，你可以看英文说明，也可以凭着直觉去试验，找到哪些库和哪些 DLL 文件是编译和运行时必须的——当然，我也可以直接告诉你答案。
设置编译环境的问题就不再多说了，不清楚的请看前面的章节。反正都三部分：*.h 文件，*.lib 文件和*.dll 文件，放到相应的文件夹下面并在编译时候指明就可以了。
我们下面将用到的文件有：
libintl.h：请在写源代码的时候#include 进来；
libintl.lib：这是编译时候需要的库文件；
libintl3.dll 和 libiconv2.dll：这是程序运行时候需要的文件，放到*.exe 文件可以找到的地方。

演示程序以及说明

```
#include <iostream>
#include <string>
#include <clocale>
#include "GNU/libintl.h"

int main(int argc, char* argv[])
```

```
{
    setlocale(LC_ALL, "");
    bindtextdomain("myText",
         "E:/My Documents/Visual Studio 2008/po");
    textdomain("myText");
    std::string test = gettext("Hello, World!");
    std::cout << test << std::endl;
    return 0;
}
```

我们先说#include 进来的<clocale>，我用“<>”表示它是标准 C++的一部分。它包含了函数 setlocale()。这个函数在这里的两个参数——常量 LC_ALL 与空字符串""的意思是，在这个程序中的所有语言与区域，都设置为系统默认的语言与区域。

libintl.h 是我们刚才加入的 GNU 的一部分，这意味着在 Linux 系统下，这个头文件是系统本身自带的。它包含了后面三个函数：bindtextdomain()将一个文件夹目录绑定到一个域名上，这个域名也是将来*.mo 文件的文件名；textdomain()表明我们将使用的域名；gettext()中的字符串将是被多语言翻译替换的部分。

将这个程序编译，在没有多语言包的时候，程序也能正常的运行，显示“Hello, World!”。

为源程序制作 po 文件和 mo 文件

如果你已经安装了完整的安装包，找到相关文件夹的 bin 目录，这里有很多工具软件。你可以通过 cmd 的方式一步步的转换，也可以，偷点儿懒，因为有更加现成的工具可以用。但是，第一步，从源代码提取 gettext()的文本，还得靠命令：xgettext。就跟用 g++命令一样，假设我们的源文件名是 main.cpp，我们把它先转换成一个模板文件 a.pot：

```
xgettext -o a.pot main.cpp
```

你可以用 vim 之类的文本编辑器看看*.pot 文件的内容，你会发现，一些说明，以及提取文本的详细信息被纪录了下来。

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2008-03-30 00:24+0800\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
```

```
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: main.cpp:11
msgid "Hello, World!"
msgstr ""
```

下面，我们使用一个简单的小工具 poedit。又一个跨平台的软件，主页在：
http://www.poedit.net/
安装运行后，选择“从 POT 文件更新类目”，然后打开我们刚才的 a.pot，什么都不用修改（当然，你也可以把自己信息都写上去），确保“字符集”是 UTF-8 就可以了。然后，在英语下面也上替换的文字吧，保存的时候，相应的 mo 文件也就建立起来了。

```
msgid ""
msgstr ""
"Project-Id-Version: \n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2008-03-30 00:24+0800\n"
"PO-Revision-Date: 2008-03-30 00:25+0800\n"
"Last-Translator: lf426 <zbln426@163.com>\n"
"Language-Team: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: main.cpp:11
msgid "Hello, World!"
msgstr "浜茬垢鐨勫笘鐣岋紝鎴戞潵浜嗭紒"
```

这是 po 文件。怎么是乱码？那是因为 windows 不是用 UTF-8 保存的文本文件（默认一般是 GB2312）。用 poedit 打开时候是正确显示的。我的文本内容是：“亲爱的世界，我来了！”。如果你用的是 vim，可以通过设置环境变量解决显示乱码的问题，在_vimrc 文件中添加这一句：

```
set fileencodings=gb2312,ucs-bom,utf-8,chinese
```

设置 mo 文件的目录

下面的工作可能就有些教条了。还记得我们绑定域名的路径吧，我用的是
E:\My Documents\Visual Studio 2008\po
（请注意在 C++程序里面把斜杠反过来！）
*.mo 文件并不是直接放到这个路径下，而是这个路径下的./LL/LC_MESSAGES 或者./LL_CC/LC_MESSAGES。其中 LL 表示语种，CC 表示国家或区域。具体的请参考 Wesnoth。就我们的中文来说，这个例子放 mo 文件的路径是：
E:\My Documents\Visual Studio 2008\po\zh_CN\LC_MESSAGES

现在运行程序就可以看到文本已经被替换了。如果我们删除 mo 文件或修改 mo 文件名（与绑定域名不一致），程序会继续显示原来的英文。如果我们改变系统环境，只要不是中国中文，程序都还是显示英文。如果我们要更新替换内容，直接用 poedit 更新 po 和 mo 文件就可以了。

构建 StringData 类

我们希望字符串的数据单独的保存在一个文件里，这样既方便被 gettext 提取，也方便修改。而且，在程序里面，我们尽量把 gettext 涉及到的一些特殊的设置隐藏了。所以，我们构建 StringDada 类，在程序中需要用到的地方，直接调用它的对象就可以了。

```
//FileName: string_data.h
#ifndef STRING_DATA_H
#define STRING_DATA_H

#include <clocale>
#include <string>
#include <vector>
#include "GNU/libintl.h"

class StringData
{
private:
    std::vector<std::string> data;
public:
    StringData();
    std::string operator [](const unsigned int& n) const;
};

#endif
```

我重载了[]，这样在调用数据的时候更加直观。我们将数据都写在 StringData 的构造函数中，将来 gettext 也只需要提取 StringData 的实现文件就可以了。

```
#include "string_data.h"

StringData::StringData()
{
    setlocale(LC_ALL, "");
    bindtextdomain("StringData", "./po");
    textdomain("StringData");

    //0
    data.push_back(gettext("Up was pressed."));
```

```
    //1
    data.push_back(gettext("Down was pressed."));
    //2
    data.push_back(gettext("Left was pressed."));
    //3
    data.push_back(gettext("Right was pressed."));
    //4
    data.push_back(gettext("Other key was pressed."));
}

std::string StringData::operator [](const unsigned int& n) const
{
    if ( n >= data.size() )
        return 0;
    return data[n];
}
```

做个 gettext 的批处理

如果你按照我全面介绍的，安装了 Poedit，也安装了 GnuWin32，那么，我们做个批处理文件让从 string_data.cpp 到 StringData.mo 的转换更加简单吧。（如果安装路径不一样请做相应的修改）。

```
@set path=C:\Program Files\GnuWin32\bin;%PATH%;
xgettext --force-po -o string_data.pot string_data.cpp
msginit -l zh_CN -o StringData.po -i string_data.pot
@set path=C:\Program Files\Poedit\bin;%PATH%;
poedit StringData.po
del string_data.pot
del StringData.po
```

Poedit 打开 StringData.po 的时候会报错，那是因为文件指明的编码不可用，请在“字符集”中选择 UTF-8，另外，在“工程名称以及版本”中写点信息，不要使用默认值就可以了。然后翻译并保存，StringData.mo 文件就生成了。

本文源自本人博客：
http://www.cppblog.com/1f426/
彻底解密 C++宽字符系列文章：
http://www.cppblog.com/1f426/archive/2010/06/25/118707.html
转载请注明作者和原文地址。
E-mail： zb1n426@163.com