

第2章 Unicode

随着Microsoft公司的Windows操作系统在全世界日益广泛的流行,对于软件开发人员来说,将目标瞄准国际上的各个不同市场,已经成为一个越来越重要的问题。美国的软件版本比国际版本提前6个月推向市场,这曾经是个司空见惯的现象。但是,由于各国对 Windows操作系统提供了越来越多的支持,因此就更加容易为国际市场生产各种应用软件,从而缩短了软件的美版与国版推出的时间间隔。

Windows操作系统始终不渝地提供各种支持,以帮助软件开发人员进行应用程序的本地化工作。应用软件可以从各种不同的函数中获得特定国家的信息,并可观察控制面板的设置,以确定用户的首选项。Windows甚至支持不同的字体,以适应应用的需要。

之所以将这一章放在本书的开头,是因为考虑到 Unicode是开发任何应用程序时要采用的基本步骤。本书的每一章中几乎都要讲到关于 Unicode的问题,而且书中给出的所有示例应用程序都是“用Unicode实现的”。如果你为Microsoft Windows 2000或Microsoft Windows CE开发应用程序,你应该使用Unicode进行开发。如果你为Microsoft Windows 98开发应用程序,你必须对某些问题作出决定。本章也要讲述Windows 98的有关问题。

2.1 字符集

软件的本地化要解决的真正问题,实际上就是如何处理不同的字符集。多年来,许多人一直将文本串作为一系列单字节字符来进行编码,并在结尾处放上一个零。对于我们来说,这已经成了习惯。当调用strlen函数时,它在以0结尾的单字节字符数组中返回字符的数目。

问题是,有些文字和书写规则(比如日文中的汉字就是个典型的例子)的字符集中的符号太多了,因此单字节(它提供的符号最多不能超过256个)是根本不敷使用的。为此出现了双字节字符集(DBCS),以支持这些文字和书写规则。

2.1.1 单字节与双字节字符集

在双字节字符集中,字符串中的每个字符可以包含一个字节或包含两个字节。例如,日文中的汉字,如果第一个字符在0x81与0x9F之间,或者在0xE0与0xFC之间,那么就必须观察下一个字节,才能确定字符串中的这个完整的字符。使用双字节字符集,对于程序员来说简直是个很大的难题,因为有些字符只有一个字节宽,而有些字符则是两个字节宽。

如果只是调用strlen函数,那么你无法真正了解字符串中究竟有多少字符,它只能告诉你到达结尾的0之前有多少个字节。ANSI的C运行期库中没有配备相应的函数,使你能够对双字节字符集进行操作。但是,Microsoft Visual C++的运行期库却包含许多函数,如_mbslen,它可以用来操作多字节(既包括单字节也包括双字节)字符串。

为了帮助你对DBCS字符串进行操作,Windows提供了下面的一组帮助函数(见表2-1)。

前两个函数CharNext和CharPrev允许前向或逆向遍历DBCS字符串,方法是每次一个字节。第三个函数IsDBCSLeadByte,在字节返回到一个两字节字符的第一个字节时将返回TRUE。

表2-1 对DBCS字符串进行操作的帮助函数

函 数	描 述
PTSTR CharNext (PCTSTR pszCurrentChar) ;	返回字符串中的下一个字符的地址
PTSTR CharPrev (PCTSTR pszStart,PCTSTR pszCurrentChar) ;	返回字符串中的上一个字符的地址
BOOL IsDBCSLeadByteTRUE(BYTE bTestChar) ;	如果该字节是DBCS字符的第一个字节, 则返回

尽管这些函数使得我们对 DBCS的操作更容易, 但还需要, 一个更好的方法让我们来看看Unicode。

2.1.2 Unicode : 宽字节字符集

Unicode是Apple和Xerox公司于1988年建立的一个技术标准。1991年, 成立了一个集团机构负责Unicode的开发和推广应用。该集团由 Apple、Compaq、HP、IBM、Microsoft、Oracle、Silicon Graphics, Inc.、Sybase、Unisys和Xerox等公司组成(若要了解该集团的全部成员, 请通过网址 www.Unicode.org 查找)。该集团负责维护Unicode标准。Unicode的完整描述可以参阅AddisonWesley出版的《Unicode Standard》一书(该书可以通过网址 www.Unicode.org 订购)。

Unicode提供了一种简单而又一致的表示字符串的方法。Unicode字符串中的所有字符都是16位的(两个字节)。它没有专门的字节来指明下一个字节是属于同一个字符的组成部分, 还是一个新字符。这意味着你只需要对指针进行递增或递减, 就可以遍历字符串中的各个字符, 不再需要调用CharNext、CharPrev和IsDBCSLeadByte之类的函数。

由于Unicode用一个16位的值来表示每个字符, 因此总共可以得到 65 000个字符, 这样, 它就能够对世界各国的书面文字中的所有字符进行编码, 远远超过了单字节字符集的 256个字符的数目。

目前, 已经为阿拉伯文、中文拼音、西里尔字母(俄文)、希腊文、西伯莱文、日文、韩文和拉丁文(英文)字母定义了 Unicode代码点[⊖]。这些字符集中还包含了大量的标点符号、数学符号、技术符号、箭头、装饰标志、区分标志和其他许多字符。如果将所有这些字母和符号加在一起, 总计约达 35000个不同的代码点, 这样, 总计 65 000多个代码点中, 大约还有一半可供将来扩充时使用。

这65 536个字符可以分成不同的区域。表 2-2 显示了这样的区域的一部分以及分配给这些区域的字符。

表2-2 区域字符

16位代码	字 符	16 位 代 码	字 符
0000-007F	ASCII	0300-036F	通用区分标志
0080-00FF	拉丁文1字符	0400-04FF	西里尔字母
0100-017F	欧洲拉丁文	0530-058F	亚美尼亚文
0180-01FF	扩充拉丁文	0590-05FF	西伯莱文
0250-02AF	标准拼音	0600-06FF	阿拉伯文
02B0-02FF	修改型字母	0900-097F	梵文

目前尚未分配的代码点大约还有 29 000个, 不过它们是保留供将来使用的。另外, 大约有 6000个代码点是保留供个人使用的。

⊖ 代码点是字符集中符号的位置。

2.2 为什么使用Unicode

当开发应用程序时，当然应该考虑利用 Unicode的优点。即使现在你不打算对应用程序进行本地化，开发时将Unicode放在心上，肯定可以简化将来的代码转换工作。此外，Unicode还具备下列功能：

- 可以很容易地在不同语言之间进行数据交换。
- 使你能够分配支持所有语言的单个二进制 .exe 文件或 DLL 文件。
- 提高应用程序的运行效率（本章后面还要详细介绍）。

2.3 Windows 2000与Unicode

Windows 2000是使用Unicode从头进行开发的，用于创建窗口、显示文本、进行字符串操作等的核心函数都需要 Unicode字符串。如果调用任何一个 Windows函数并给它传递一个 ANSI字符串，那么系统首先要将字符串转换成 Unicode，然后将Unicode字符串传递给操作系统。如果希望函数返回 ANSI字符串，系统就会首先将 Unicode字符串转换成ANSI字符串，然后将结果返回给你的应用程序。所有这些转换操作都是在你看不见的情况下发生的。当然，进行这些字符串的转换需要占用系统的时间和内存。

例如，如果调用CreateWindowEx函数，并传递类名字和窗口标题文本的非Unicode字符串，那么CreateWindowEx必须分配内存块（在你的进程的默认堆中），将非Unicode字符串转换成Unicode字符串，并将结果存储在分配到的内存块中，然后调用Unicode版本的CreateWindowEx函数。

对于用字符串填入缓存的函数来说，系统必须首先将Unicode字符串转换成非Unicode字符串，然后你的应用程序才能处理该字符串。由于系统必须执行所有这些转换操作，因此你的应用程序需要更多的内存，并且运行的速度比较慢。通过从头开始用Unicode来开发应用程序，就能够使你的应用程序更加有效地运行。

2.4 Windows 98与Unicode

Windows 98不是一种全新的操作系统。它继承了16位Windows操作系统的特性，它不是用来处理Unicode的。如果要增加对Unicode的支持，其工作量非常大，因此在该产品的特性列表中没有包括这个支持项目。由于这个原因，Windows 98像它的前任产品一样，几乎都是使用ANSI字符串来进行所有的内部操作的。

仍然可以编写用于处理Unicode字符和字符串的Windows应用程序，不过，使用Windows函数要难得多。例如，如果想要调用CreateWindowEx函数并将ANSI字符串传递给它，这个调用的速度非常快，不需要从你进程的默认堆栈中分配缓存，也不需要进行字符串转换。但是，如果想要调用CreateWindowEx函数并将Unicode字符串传递给它，就必须明确分配缓存，并调用函数，以便执行从Unicode到ANSI字符串的转换操作。然后可以调用CreateWindowEx，传递ANSI字符串。当CreateWindowEx函数返回时，就能释放临时缓存。这比使用Windows 2000上的Unicode要麻烦得多。本章的后面要介绍如何在Windows 98下进行这些转换。

虽然大多数Unicode函数在Windows 98中不起任何作用，但是仍有少数Unicode函数确实非常有用。这些函数是：

EnumResourceLanguagesW

GetTextExtentPoint32W

EnumResourceNamesW

GetTextExtentPointW

EnumResourceTypesW	LstrlenW
ExtTextOutW	MessageBoxExW
FindResourceW	MessageBoxW
FindResourceExW	TextOutW
GetCharWidthW	WideCharToMultiByte
GetCommandLineW	MultiByteToWideChar

可惜的是，这些函数中有许多函数在 Windows 98 中会出现各种各样的错误。有些函数无法使用某些字体，有些函数会破坏内存堆栈，有些函数会使打印机驱动程序崩溃，等等。如果要使用这些函数，必须对它们进行大量的测试。即使这样，可能仍然无法解决问题。因此必须向用户说明这些情况。

2.5 Windows CE与Unicode

Windows CE操作系统是为小型设备开发的，这些设备的内存很小，并且不带磁盘存储器。你可能认为，由于 Microsoft 公司的主要目标是建立一种尽可能小的操作系统，因此它会使用 ANSI 作为自己的字符集。但是 Microsoft 公司并非鼠目寸光，他们懂得，Windows CE 的设备要在世界各地销售，他们希望降低软件开发成本，这样就能更加容易地开发应用程序。为此，Windows CE 本身就是使用 Unicode 的一种操作系统。

但是，为了使 Windows CE 尽量做得小一些，Microsoft 公司决定完全不支持 ANSI Windows 函数。因此，如果要为 Windows CE 开发应用程序，必须懂得 Unicode，并且在整个应用程序中使用 Unicode。

2.6 需要注意的问题

下面让我们进一步明确一下“Microsoft 公司对 Unicode 支持的情况”：

- Windows 2000 既支持 Unicode，也支持 ANSI，因此可以为任意一种开发应用程序。
- Windows 98 只支持 ANSI，只能为 ANSI 开发应用程序。
- Windows CE 只支持 Unicode，只能为 Unicode 开发应用程序。

虽然 Microsoft 公司试图让软件开发人员能够非常容易地开发在这 3 种平台上运行的软件，但是 Unicode 与 ANSI 之间的差异使得事情变得困难起来，并且这种差异通常是我遇到的最大的问题之一。请不要误解，Microsoft 公司坚定地支持 Unicode，并且我也坚决鼓励你使用它。不过你应该懂得，你可能遇到一些问题，需要一定的时间来解决这些问题。建议你尽可能使用 Unicode。如果运行 Windows 98，那么只有在必要时才需转换到 ANSI。

不过，还有另一个小问题你应该了解，那就是 COM。

2.7 对COM的简单说明

当 Microsoft 公司将 COM 从 16 位 Windows 转换成 Win32 时，公司作出了一个决定，即需要字符串的所有 COM 接口方法都只能接受 Unicode 字符串。这是个了不起的决定，因为 COM 通常用于使不同的组件能够互相进行通信，而 Unicode 则是传递字符串的最佳手段。

如果你为 Windows 2000 或 Windows CE 开发应用程序，并且也使用 COM，那么你将会如虎添翼。在你的整个源代码中使用 Unicode，将使与操作系统进行通信和与 COM 对象进行通信的操作变成一件轻而易举的事情。

如果你为 Windows 98 开发应用程序，并且也使用 COM，那么将会遇到一些问题。COM 要

求使用Unicode字符串，而操作系统的大多数函数要求使用ANSI字符串。那是多么难办的事情啊！我曾经从事过若干个项目的开发，在这些项目中，我编写了许多代码，仅仅是为了来回进行字符串的转换。

2.8 如何编写Unicode源代码

Microsoft公司为Unicode设计了Windows API，这样，可以尽量减少对你的代码的影响。实际上，你可以编写单个源代码文件，以便使用或者不使用Unicode来对它进行编译。只需要定义两个宏（UNICODE和_UNICODE），就可以修改然后重新编译该源文件。

2.8.1 C运行期库对Unicode的支持

为了利用Unicode字符串，定义了一些数据类型。标准的C头文件String.h已经作了修改，以便定义一个名字为wchar_t的数据类型，它是一个Unicode字符的数据类型：

```
typedef unsigned short wchar_t;
```

例如，如果想要创建一个缓存，用于存放最多为99个字符的Unicode字符串和一个结尾为零的字符，可以使用下面这个语句：

```
wchar_t szBuffer[100];
```

该语句创建了一个由100个16位值组成的数组。当然，标准的C运行期字符串函数，如strcpy、strchr和strcat等，只能对ANSI字符串进行操作，不能正确地处理Unicode字符串。因此，ANSI C也拥有一组补充函数。清单2-1显示了一些标准的ANSI C字符串函数，后面是它们的等价Unicode函数。

```
char * strcat(char *, const char *);  
wchar_t * wcscat(wchar_t *, const wchar_t *);
```

清单2-1 标准的ANSI C字符串函数和它们的等价Unicode函数

```
char * strchr(const char *, int);  
wchar_t * wcschr(const wchar_t *, wchar_t);  
  
int strcmp(const char *, const char *);  
int wcscmp(const wchar_t *, const wchar_t *);  
  
char * strcpy(char *, const char *);  
wchar_t * wcscpy(wchar_t *, const wchar_t *);  
  
size_t strlen(const char *);  
size_t wcslen(const wchar_t *);
```

请注意，所有的Unicode函数均以wcs开头，wcs是宽字符串的英文缩写。若要调用Unicode函数，只需用前缀wcs来取代ANSI字符串函数的前缀str即可。

注意 大多数软件开发人员可能已经不记得这样一个非常重要的问题了，那就是Microsoft公司提供的C运行期库与ANSI的标准C运行期库是一致的。ANSI C规定，C运行期库支持Unicode字符和字符串。这意味着始终都可以调用C运行期函数，以便对Unicode字符和字符串进行操作，即使是在Windows 98上运行，也可以调用这些函数。换句话说，wcscat、wcslen和wcstok等函数都能够在Windows 98上很好地运行，这些都是必须关心的操作系统函数。

对于包含了对 `str` 函数或 `wcs` 函数进行显式调用的代码来说，无法非常容易地同时为 ANSI 和 Unicode 对这些代码进行编译。本章前面说过，可以创建同时为 ANSI 和 Unicode 进行编译的单个源代码文件。若要建立双重功能，必须包含 `TChar.h` 文件，而不是包含 `String.h` 文件。

`TChar.h` 文件的唯一作用是帮助创建 ANSI/Unicode 通用源代码文件。它包含你应该用在源代码中的一组宏，而不应该直接调用 `str` 函数或者 `wcs` 函数。如果在编译源代码文件时定义了 `_UNICODE`，这些宏就会引用 `wcs` 这组函数。如果没有定义 `_UNICODE`，那么这些宏将引用 `str` 这组宏。

例如，在 `TChar.h` 中有一个宏称为 `_tcsncpy`。如果在包含该头文件时没有定义 `_UNICODE`，那么 `_tcsncpy` 就会扩展为 ANSI 的 `strcpy` 函数。但是如果定义了 `_UNICODE`，`_tcsncpy` 将扩展为 Unicode 的 `wcsncpy` 函数。拥有字符串参数的所有 C 运行期函数都在 `TChar.h` 文件中定义了一个通用宏。如果使用通用宏，而不是 ANSI/Unicode 的特定函数名，就能够顺利地创建可以为 ANSI 或 Unicode 进行编译的源代码。

但是，除了使用这些宏之外，还有一些操作是必须进行的。`TChar.h` 文件包含了另外一些宏。

若要定义一个 ANSI/Unicode 通用的字符串数组，请使用下面的 `TCHAR` 数据类型。如果定义了 `_UNICODE`，`TCHAR` 将声明为下面的形式：

```
typedef wchar_t TCHAR;
```

如果没有定义 `_UNICODE`，则 `TCHAR` 将声明为下面的形式：

```
typedef char TCHAR;
```

使用该数据类型，可以像下面这样分配一个字符串：

```
TCHAR szString[100];
```

也可以创建对字符串的指针：

```
TCHAR *szError = "Error";
```

不过上面这行代码存在一个问题。按照默认设置，Microsoft 公司的 C++ 编译器能够编译所有的字符串，就像它们是 ANSI 字符串，而不是 Unicode 字符串。因此，如果没有定义 `_UNICODE`，该编译器将能正确地编译这一行代码。但是，如果定义了 `_UNICODE`，就会产生一个错误。若要生成一个 Unicode 字符串而不是 ANSI 字符串，必须将该代码行改写为下面的样子：

```
TCHAR *szError = L"Error";
```

字符串（literal string）前面的大写字母 `L`，用于告诉编译器该字符串应该作为 Unicode 字符串来编译。当编译器将字符串置于程序的数据部分中时，它在每个字符之间分散插入零字节。这种变更带来的问题是，现在只有当定义了 `_UNICODE` 时，程序才能成功地进行编译。我们需要另一个宏，以便有选择地在字符串的前面加上大写字母 `L`。这项工作由 `_TEXT` 宏来完成，`_TEXT` 宏也在 `TChar.h` 文件中做了定义。如果定义了 `_UNICODE`，那么 `_TEXT` 定义为下面的形式：

```
#define _TEXT(x) L ## x
```

如果没有定义 `_UNICODE`，`_TEXT` 将定义为

```
#define _TEXT(x) x
```

使用该宏，可以改写上面这行代码，这样，无论是否定义了 `_UNICODE` 宏，它都能够正确地进行编译。如下所示：

```
TCHAR *szError = _TEXT("Error");
```

_TEXT宏也可以用于字符串。例如，若要检查一个字符串的第一个字符是否是大写字母 J，只需编写下面的代码即可：

```
if (szError[0] == _TEXT('J')) {  
    // First character is a 'J'  
    :  
}  
else {  
    // First character is not a 'J'  
    :  
}
```

2.8.2 Windows定义的Unicode数据类型

Windows头文件定义了表2-3列出的数据类型。

表2-3 Unicode 数据类型

数据类型	说明
WCHAR	Unicode字符
PWSTR	指向Unicode字符串的指针
PCWSTR	指向一个恒定的Unicode字符串的指针

这些数据类型是指Unicode字符和字符串。Windows头文件也定义了ANSI/Unicode通用数据类型PTSTR和PCTSTR。这些数据类型既可以指ANSI字符串，也可以指Unicode字符串，这取决于当编译程序模块时是否定义了UNICODE宏。

请注意，这里的UNICODE宏没有前置的下划线。_UNICODE宏用于C运行期头文件，而UNICODE宏则用于Windows头文件。当编译源代码模块时，通常必须同时定义这两个宏。

2.8.3 Windows中的Unicode函数和ANSI函数

前面已经讲过，有两个函数称为CreateWindowEx，一个CreateWindowEx接受Unicode字符串，另一个CreateWindowEx接受ANSI字符串。情况确实如此，不过，这两个函数的原型实际上是下面的样子：

```
HWND WINAPI CreateWindowEx(  
    DWORD dwExStyle,  
    PCWSTR pClassName,  
    PCWSTR pWindowName,  
    DWORD dwStyle,  
    int X,  
    int Y,  
    int nWidth,  
    int nHeight,  
    HWND hWndParent,  
    HMENU hMenu,  
    HINSTANCE hInstance,  
    PVOID pParam);
```

```
HWND WINAPI CreateWindowExA(  
    DWORD dwExStyle,  
    PCSTR pClassName,
```

```
PCSTR pWindowName,  
DWORD dwStyle,  
int X,  
int Y,  
int nWidth,  
int nHeight,  
HWND hWndParent,  
HMENU hMenu,  
HINSTANCE hInstance,  
PVOID pParam);
```

CreateWindowExW是接受Unicode字符串的函数版本。函数名结尾处的大写字母W是英文wide(宽)的缩写。每个Unicode字符的长度是16位，因此，它们常常称为宽字符。CreateWindowExA的结尾处的大写字母A表示该函数可以接受ANSI字符串。

但是，在我们的代码中，通常只包含了对 CreateWindowEx的调用，而不是直接调用 CreateWindowExW或者CreateWindowExA。在WinUser.h文件中，CreateWindowEx实际上是定义为下面这种形式的一个宏：

```
#ifndef UNICODE  
#define CreateWindowEx CreateWindowExW  
#else  
#define CreateWindowEx CreateWindowExA  
#endif // !UNICODE
```

当编译源代码模块时，UNICODE是否已经作了定义，将决定你调用的是哪个CreateWindowEx版本。当转用一个16位的Windows应用程序时，你在编译期间可能没有定义UNICODE。对CreateWindowEx函数的任何调用都会将该宏扩展为对CreateWindowExA的调用，即对CreateWindowEx的ANSI版本的调用。由于16位Windows只提供了CreateWindowsEx的ANSI版本，因此可以比较容易地转用它的应用程序。

在Windows 2000下，Microsoft的CreateWindowExA源代码只不过是一个形实替换程序层或翻译层，用于分配内存，以便将ANSI字符串转换成Unicode字符串。该代码然后调用CreateWindowExW，并传递转换后的字符串。当CreateWindowExW返回时，CreateWindowExA便释放它的内存缓存，并将窗口句柄返回给你。

如果要创建其他软件开发人员将要使用的动态链接库（DLL），请考虑使用下面的方法。在DLL中提供两个输出函数。一个是ANSI版本，另一个是Unicode版本。在ANSI版本中，只需要分配内存，执行必要的字符串转换，并调用该函数的Unicode版本（本章后面部分介绍这个进程）。

在Windows 98下，Microsoft的CreateWindowExA源代码是执行操作的函数。Windows 98提供了接受Unicode参数的所有Windows函数的进入点，但是这些函数并不将Unicode字符串转换成ANSI字符串，它们只返回运行失败的消息。调用 GetLastError将返回ERROR_CALL_NOT_IMPLEMENTED。这些函数中只有ANSI版本的函数才能正确地运行。如果编译的代码调用了任何宽字符函数，应用程序将无法在Windows 98下运行。

Windows API中的某些函数，比如WinExec和OpenFile等，只是为了实现与16位Windows程序的向后兼容而存在，因此，应该避免使用。应该使用对CreateProcess和CreateFile函数的调用来取代对WinExec和OpenFile函数的调用。从系统内部来讲，老的函数完全可以调用新的函数。老的函数存在的一个大问题是，它们不接受Unicode字符串。当调用这些函数时，必须传递ANSI字符串。另一方面，所有新的和未过时的函数在Windows 2000中都同时拥有ANSI和

Unicode两个版本。

2.8.4 Windows字符串函数

Windows还提供了一组范围很广的字符串操作函数。这些函数与C运行期字符串函数（如strcpy和wcscpy）很相似。但是该操作系统函数是操作系统的一个组成部分，操作系统的许多组件都使用这些函数，而不使用C运行期库。建议最好使用操作系统函数，而不要使用C运行期字符串函数。这将有助于稍稍提高你的应用程序的运行性能，因为操作系统字符串函数常常被大型应用程序比如操作系统的外壳进程 Explorer.exe 所使用。由于这些函数使用得很多，因此，在你的应用程序运行时，它们可能已经被装入RAM。

若要使用这些函数，系统必须运行 Windows 2000 或 Windows 98。如果安装了 Internet Explorer 4.0 或更新的版本，也可以在较早的 Windows 版本中获得这些函数。

在经典的操作系统函数样式中，操作系统字符串函数名既包含大写字母，也包含小写字母，它的形式类似这个样子：StrCat、StrChr、StrCmp和StrCpy等。若要使用这些函数，必须加上ShlwApi.h头文件。另外，如前所述，这些字符串函数既有ANSI版本，也有Unicode版本，例如StrCatA和StrCatW。由于这些函数属于操作系统函数，因此，当创建应用程序时，如果定义了UNICODE（不带前置下划线），那么它们的符号将扩展为宽字符版本。

2.9 成为符合ANSI和Unicode的应用程序

即使你不打算立即使用Unicode，最好也应该着手将你的应用程序转换成符合Unicode的应用程序。下面是应该遵循的一些基本原则：

- 将文本串视为字符数组，而不是chars数组或字节数组。
- 将通用数据类型（如TCHAR和PTSTR）用于文本字符和字符串。
- 将显式数据类型（如BYTE和PBYTE）用于字节、字节指针和数据缓存。
- 将TEXT宏用于原义字符和字符串。
- 执行全局性替换（例如用PTSTR替换PSTR）。
- 修改字符串运算问题。例如函数通常希望你传递一个缓存的大小，而不是字节。

这意味着你不应该传递 sizeof(szBuffer)，而应该传递 (sizeof(szBuffer)/sizeof(TCHAR))。另外，如果需要为字符串分配一个内存块，并且拥有该字符串中的字符数目，那么请记住要按字节来分配内存。这就是说，应该调用 malloc(nCharacters * sizeof(TCHAR))，而不是调用 malloc(nCharacters)。在上面所说的所有原则中，这是最难记住的一条原则，如果操作错误，编译器将不发出任何警告。

当我为本书的第一版编写示例程序时，我编写的原始程序只能编译为ANSI程序。后来，当我开始撰写本章的内容时，我想我应该鼓励使用Unicode，并且打算创建一些示例程序，以便展示你可以非常容易地编写既可以用Unicode也可以用ANSI来编译的程序。这时我发现最好的办法是将本书的所有示例程序进行转换，使它们都能够用Unicode和ANSI进行编译。

我用了大约4个小时将所有程序进行了转换。考虑到我以前从来没有这方面的转换经验，这个速度是相当不错了。

2.9.1 Windows字符串函数

Windows也提供了一组用于对Unicode字符串进行操作的函数，表2-4对它们进行了描述。

表2-4 对Unicode字符串进行操作的函数

函 数	描 述
lstrcat	将一个字符串置于另一个字符串的结尾处
lstrcmp	对两个字符串进行区分大小写的比较
lstrcmpi	对两个字符串进行不区分大小写的比较
lstrcpy	将一个字符串拷贝到内存中的另一个位置
lstrlen	返回字符串的长度（按字符数来计量）

这些函数是作为宏来实现的，这些宏既可以调用函数的 Unicode版本，也可以调用函数的 ANSI版本，这要根据编译源代码模块时是否已经定义了 UNICODE而定。例如，如果没有定义 UNICODE，lstrcat函数将扩展为lstrcatA。如果定义了UNICODE，lstrcat将扩展为lstrcatW。

有两个字符串函数，即lstrcmp和lstrcmpi，它们的行为特性与等价的C运行期函数是不同的。C运行期函数strcmp、strcmpi、wcscmp和wcscmpi只是对字符串中的代码点的值进行比较，这就是说，这些函数将忽略实际字符的含义，只是将第一个字符串中的每个字符的数值与第二个字符串中的字符的数值进行比较。而 Windows函数lstrcmp和lstrcmpi是作为对 Windows函数CompareString的调用来实现的。

```
int CompareString(
    LCID lcid,
    DWORD fdwStyle,
    PCWSTR pString1,
    int cch1,
    PCTSTR pString2,
    int cch2);
```

该函数对两个 Unicode字符串进行比较。CompareString的第一个参数用于设定语言 ID (LCID)，这是个32位值，用于标识一种特定的语言。CompareString使用这个LCID来比较这两个字符串，方法是对照一种特定的语言来查看它们的字符的含义。这种操作方法比 C运行期函数简单地进行数值比较更有意义。

当lstrcmp函数系列中的任何一个函数调用 CompareString时，该函数便将调用 Windows的GetThreadString函数的结果作为第一个参数来传递：

```
LCID GetThreadLocale();
```

每次创建一个线程时，它就被赋予一种语言。函数将返回该线程的当前语言设置。

CompareString的第二个参数用于标识一些标志，这些标志用来修改该函数比较两个字符串时所用的方法。表2-5显示了可以使用的标志。

表2-5 Compare String 的标志及含义

标 志	含 义
NORM_IGNORECASE	忽略字母的大小写
NORM_IGNOREKANATYPE	不区分平假名与片假名字符
NORM_IGNORENONSPACE	忽略无间隔字符
NORM_IGNORESYMBOLS	忽略符号
NORM_IGNOREWIDTH	不区分单字节字符与作为双字节字符的同一个字符
SORT_STRINGSORT	将标点符号作为普通符号来处理

当lstrcmp调用 CompareString时，它传递 0作为fdwStyle的参数。但是，当 lstrcmpi调用 CompareString时，它就传递 NORM_IGNORECASE。CompareString的其余4个参数用于设定两

个字符串和它们各自的长度。如果为 cch1 参数传递 -1，那么该函数将认为 pString1 字符串是以 0 结尾，并计算该字符串的长度。对于 pString2 字符串来说，参数 cch2 的作用也是一样。

其他 C 运行期函数没有为 Unicode 字符串的操作提供很好的支持。例如，tolower 和 toupper 函数无法正确地转换带有重音符号的字符。为了弥补 C 运行期库中的这些不足，必须调用下面这些 Windows 函数，以便转换 Unicode 字符串的大小写字母。这些函数也可以正确地用于 ANSI 字符串。

头两个函数：

```
PTSTR CharLower(PTSTR pszString);
```

和

```
PTSTR CharUpper(PTSTR pszString);
```

既可以转换单个字符，也可以转换以 0 结尾的整个字符串。若要转换整个字符串，只需要传递字符串的地址即可。若要转换单个字符，必须像下面这样传递各个字符：

```
TCHAR cLowerCaseChar = CharLower((PTSTR) szString[0]);
```

将单个字符转换成一个 PTSTR，便可调用该函数，将一个值传递给它，在这个值中，较低的 16 位包含了该字符，较高的 16 位包含 0。当该函数看到较高位是 0 时，该函数就知道你想要转换单个字符，而不是整个字符串。返回的值是个 32 位值，较低的 16 位中是已经转换的字符。

下面两个函数与前面两个函数很相似，差别在于它们用于转换缓存中包含的字符（该缓存不必以 0 结尾）：

```
DWORD CharLowerBuff(  
    PTSTR pszString,  
    DWORD cchString);  
DWORD CharUpperBuff(  
    PTSTR pszString,  
    DWORD cchString);
```

其他的 C 运行期函数，如 isalpha、islower 和 isupper，返回一个值，指明某个字符是字母字符、小写字母还是大写字母。Windows API 提供了一些函数，也能返回这些信息，但是 Windows 函数也要考虑用户在控制面板中指定的语言：

```
BOOL IsCharAlpha(TCHAR ch);  
BOOL IsCharAlphaNumeric(TCHAR ch);  
BOOL IsCharLower(TCHAR ch);  
BOOL IsCharUpper(TCHAR ch);
```

printf 函数家族是要介绍的最后一组 C 运行期函数。如果在定义了 _UNICODE 的情况下编译你的源代码模块，那么 printf 函数家族便希望所有字符和字符串参数代表 Unicode 字符和字符串。但是，如果在没有定义 _UNICODE 的情况下编译你的源代码模块，printf 函数家族便希望传递给它的所有字符和字符串都是 ANSI 字符和字符串。

Microsoft 公司已经给 C 运行期的 printf 函数家族增加了一些特殊的域类型。其中有些域类型尚未被 ANSI C 采用。新类型使你能够很容易地对 ANSI 和 Unicode 字符和字符串进行混合和匹配。操作系统的 wsprintf 函数也得到了增强。下面是一些例子（请注意大写 S 和小写 s 的使用）：

```
char  szA[100];    // An ANSI string buffer  
WCHAR szW[100];    // A Unicode string buffer  
  
// Normal sprintf: all strings are ANSI  
sprintf(szA, "%s", "ANSI Str");
```

```
// Converts Unicode string to ANSI
sprintf(sza, "%S", L"Unicode Str");

// Normal sprintf: all strings are Unicode
sprintf(szw, L"%s", L"Unicode Str");

// Converts ANSI string to Unicode
sprintf(szw, L"%S", "ANSI Str");
```

2.9.2 资源

当资源编译器对你的所有资源进行编译时，输出文件是资源的二进制文件。资源（字符串表、对话框模板和菜单等）中的字符串值总是写作 Unicode 字符串。在 Windows 98 和 Windows 2000 下，如果应用程序没有定义 UNICODE 宏，那么系统就会进行内部转换。

例如，如果在编译源代码模块时没有定义 UNICODE，调用 LoadString 实际上就是调用 LoadStringA 函数。这时 LoadStringA 就从你的资源中读取字符串，并将该字符串转换成 ANSI 字符串。ANSI 形式的字符串将从该函数返回给你的应用程序。

2.9.3 确定文本是 ANSI 文本还是 Unicode 文本

到现在为止，Unicode 文本文件仍然非常少。实际上，Microsoft 公司自己的大多数产品并没有配备任何 Unicode 文本文件。但是预计将来这种情况是会改变的（尽管这需要一个很长的过程）。当然，Windows 2000 的 Notepad (记事本) 应用程序允许你既能打开 Unicode 文件，也能打开 ANSI 文件，并且可以创建这些文件。图 2-1 显示了 Notepad 的 Save As (文件另存为) 对话框。请注意可以用不同的方法来保存文本文件。

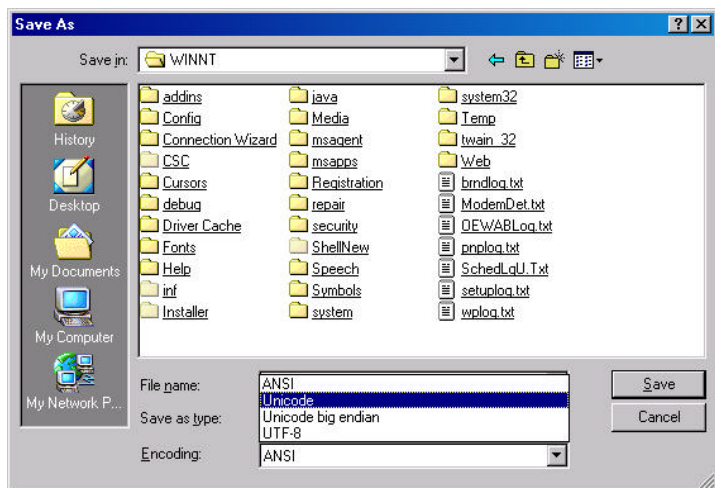


图2-1 Windows 2000 Notepad的File Save As对话框

对于许多用来打开文本文件和处理这些文件的应用程序（如编译器）来说，打开一个文件后，应用程序就能方便地确定该文本文件是包含 ANSI 字符还是 Unicode 字符。IsTextUnicode 函数能够帮助进行这种区分：

```
DWORD IsTextUnicode(CONST PVOID pBuffer, int cb, PINT pResult);
```

文本文件存在的问题是，它们的内容没有严格和明确的规则，因此很难确定该文件是包含

ANSI字符还是Unicode字符。IsTextUnicode使用一系列统计方法和定性方法，以便猜测缓存的内容。由于这不是一种确切的科学方法，因此IsTextUnicode有可能返回不正确的结果。

第一个参数pvBuffer用于标识要测试的缓存的地址。该数据是个无效指针，因为你不知道你拥有的是ANSI字符数组还是Unicode字符数组。

第二个参数cb用于设定pvBuffer指向的字节数。同样，由于你不知道缓存中放的是什么，因此cb是个字节数，而不是字符数。请注意，不必设定缓存的整个长度。当然，IsTextUnicode能够测试的字节越多，得到的结果越准确。

第三个参数pResult是个整数的地址，必须在调用IsTextUnicode之前对它进行初始化。对该整数进行初始化后，就可以指明你要IsTextUnicode执行哪些测试。也可以为该参数传递NULL，在这种情况下，IsTextUnicode将执行它能够进行的所有测试（详细说明请参见Platform SDK文档）。

如果IsTextUnicode认为缓存包含Unicode文本，便返回TRUE，否则返回FALSE。确实是这样，尽管Microsoft将该函数的原型规定为返回DWORD，但是它实际上返回一个布尔值。如果在pResult参数指向的整数中必须进行特定的测试，该函数就会在返回之前设定整数中的信息位，以反映每个测试的结果。

Windows98 在Windows 98下，IsTextUnicode函数没有有用的实现代码，它只是返回FALSE。调用GetLastError函数将返回ERROR_CALL_NOT_IMPLEMENTED。

第17章中的Flie Rev示例应用程序演示了IsTextUnicode函数的使用。

2.9.4 在Unicode与ANSI之间转换字符串

Windows函数MultiByteToWideChar用于将多字节字符串转换成宽字符串。下面显示了MultiByteToWideChar函数。

```
int MultiByteToWideChar(  
    UINT uCodePage,  
    DWORD dwFlags,  
    PCSTR pMultiByteStr,  
    int cchMultiByte,  
    PWSTR pWideCharStr,  
    int cchWideChar);
```

uCodePage参数用于标识一个与多字节字符串相关的代码页号。dwFlags参数用于设定另一个控件，它可以用重音符号之类的区分标记来影响字符。这些标志通常并不使用，在dwFlags参数中传递0。pMultiByteStr参数用于设定要转换的字符串，cchMultiByte参数用于指明该字符串的长度（按字节计算）。如果为cchMultiByte参数传递-1，那么该函数用于确定源字符串的长度。

转换后产生的Unicode版本字符串将被写入内存中的缓存，其地址由pWideCharStr参数指定。必须在cchWideChar参数中设定该缓存的最大值（以字符为计量单位）。如果调用MultiByteToWideChar，给cchWideChar参数传递0，那么该参数将不执行字符串的转换，而是返回为使转换取得成功所需要的缓存的值。一般来说，可以通过下列步骤将多字节字符串转换成Unicode等价字符串：

- 1) 调用MultiByteToWideChar函数，为pWideCharStr参数传递NULL，为cchWideChar参数传递0。

- 2) 分配足够的内存块，用于存放转换后的Unicode字符串。该内存块的大小由前面对

MultiByteToWideChar的调用返回。

3) 再次调用MultiByteToWideChar，这次将缓存的地址作为pWideCharStr参数来传递，并传递第一次调用MultiByteToWideChar时返回的缓存大小，作为cchWidechar参数。

4. 使用转换后的字符串。

5) 释放Unicode字符串占用的内存块。

函数WideCharToMultiByte将宽字符串转换成等价的多字节字符串，如下所示：

```
int WideCharToMultiByte(
    UINT uCodePage,
    DWORD dwFlags,
    PCWSTR pWideCharStr,
    int cchWideChar,
    PSTR pMultiByteStr,
    int cchMultiByte,
    PCSTR pDefaultChar,
    PBOOL pfUsedDefaultChar);
```

该函数与MultiBiteToWideChar函数相似。同样，uCodePage参数用于标识与新转换的字符串相关的代码页。dwFlags则设定用于转换的其他控件。这些标志能够作用于带有区分符号的字符和系统不能转换的字符。通常不需要为字符串的转换而拥有这种程度的控制手段，你将为dwFlags参数传递0。

pWideCharStr参数用于设定要转换的字符串的内存地址，cchWideChar参数用于指明该字符串的长度（用字符数来计量）。如果你为cchWideChar参数传递-1，那么该函数用于确定源字符串的长度。

转换产生的多字节版本的字符串被写入由pMultiByteStr参数指明的缓存。必须在cchMultiByte参数中设定该缓存的最大值（用字节来计量）。如果传递0作为WideCharToMultiByte函数的cchMultiByte参数，那么该函数将返回目标缓存需要的大小值。通常可以使用将多字节字符串转换成宽字节字符串时介绍的一系列类似的事件，将宽字节字符串转换成多字节字符串。

你会发现，WideCharToMultiByte函数接受的参数比MultiByteToWideChar函数要多2个，即pDefaultChar和pfUsedDefaultChar。只有当WideCharToMultiByte函数遇到一个宽字节字符，而该字符在uCodePage参数标识的代码页中并没有它的表示法时，WideCharToMultiByte函数才使用这两个参数。如果宽字节字符不能被转换，该函数便使用pDefaultChar参数指向的字符。如果该参数是NULL（这是大多数情况下的参数值），那么该函数使用系统的默认字符。该默认字符通常是个问号。这对于文件名来说是危险的，因为问号是个通配符。

pfUsedDefaultChar参数指向一个布尔变量，如果宽字符串中至少有一个字符不能转换成等价多字节字符，那么函数就将该变量置为TRUE。如果所有字符均被成功地转换，那么该函数就将该变量置为FALSE。当函数返回以便检查宽字节字符串是否被成功地转换后，可以测试该变量。同样，通常为该测试传递NULL。

关于如何使用这些函数的详细说明，请参见Platform SDK文档。

如果使用这两个函数，就可以很容易创建这些函数的Unicode版本和ANSI版本。例如，你可能有一个动态链接库，它包含一个函数，能够转换字符串中的所有字符。可以像下面这样编写该函数的Unicode版本：

```
BOOL StringReverseW(PWSTR pWideCharStr) {
    // Get a pointer to the last character in the string.
    PWSTR pEndOfStr = pWideCharStr + wcslen(pWideCharStr) - 1;
```

```

wchar_t cCharT;
// Repeat until we reach the center character in the string.
while (pWideCharStr < pEndOfStr) {
    // Save a character in a temporary variable.
    cCharT = *pWideCharStr;

    // Put the last character in the first character.
    *pWideCharStr = *pEndOfStr;

    // Put the temporary character in the last character.
    *pEndOfStr = cCharT;

    // Move in one character from the left.
    pWideCharStr++;

    // Move in one character from the right.
    pEndOfStr--;
}

// The string is reversed; return success.
return(TRUE);
}

```

你可以编写该函数的ANSI版本以便该函数根本不执行转换字符串的实际操作。你也可以编写该函数的ANSI版本，以便该函数它将ANSI字符串转换成Unicode字符串，将Unicode字符串传递给StringReverseW函数，然后将转换后的字符串重新转换成ANSI字符串。该函数类似下面的样子：

```

BOOL StringReverseA(PSTR pMultiByteStr) {
    PWSTR pWideCharStr;
    int nLenOfWideCharStr;
    BOOL fOk = FALSE;

    // Calculate the number of characters needed to hold
    // the wide-character version of the string.
    nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,
        pMultiByteStr, -1, NULL, 0);

    // Allocate memory from the process's default heap to
    // accommodate the size of the wide-character string.
    // Don't forget that MultiByteToWideChar returns the
    // number of characters, not the number of bytes, so
    // you must multiply by the size of a wide character.
    pWideCharStr = HeapAlloc(GetProcessHeap(), 0,
        nLenOfWideCharStr * sizeof(WCHAR));

    if (pWideCharStr == NULL)
        return(fOk);

    // Convert the multibyte string to a wide-character string.
    MultiByteToWideChar(CP_ACP, 0, pMultiByteStr, -1,
        pWideCharStr, nLenOfWideCharStr);

    // Call the wide-character version of this
    // function to do the actual work.
    fOk = StringReverseW(pWideCharStr);
}

```

```
if (fOk) {
    // Convert the wide-character string back
    // to a multibyte string.
    WideCharToMultiByte(CP_ACP, 0, pWideCharStr, -1,
        pMultiByteStr, strlen(pMultiByteStr), NULL, NULL);
}

// Free the memory containing the wide-character string.
HeapFree(GetProcessHeap(), 0, pWideCharStr);

return(fOk);
}
```

最后，在用动态链接库分配的头文件中，可以像下面这样建立这两个函数的原型：

```
BOOL StringReverseW(PWSTR pWideCharStr);
BOOL StringReverseA(PSTR pMultiByteStr);
```

```
#ifdef UNICODE
#define StringReverse StringReverseW
#else
#define StringReverse StringReverseA
#endif // !UNICODE
```