

第四部分 动态链接库

第19章 DLL基础

自从Microsoft公司推出第一个版本的Windows操作系统以来，动态链接库（DLL）一直是这个操作系统的基础。Windows API中的所有函数都包含在DLL中。3个最重要的DLL是Kernel32.dll，它包含用于管理内存、进程和线程的各个函数；User32.dll，它包含用于执行用户界面任务（如窗口的创建和消息的传送）的各个函数；GDI32.dll，它包含用于画图和显示文本的各个函数。

Windows还配有若干别的DLL，它们提供了用于执行一些特殊任务的函数。例如，AdvAPI32.dll包含用于实现对象安全性、注册表操作和事件记录的函数；ComDlg32.dll包含常用对话框（如File Open和File Save）；ComCtl32.DLL则支持所有的常用窗口控件。

本章将要介绍如何为应用程序创建DLL。下面是为什么要使用DLL的一些原因：

- 它们扩展了应用程序的特性。由于DLL能够动态地装入进程的地址空间，因此应用程序能够在运行时确定需要执行什么操作，然后装入相应的代码，以便根据需要执行这些操作。例如，当一家公司开发了一种产品，想要让其他公司改进或增强该产品的功能时，那么就可以使用DLL。
- 它们可以用许多种编程语言来编写。可以选择手头拥有的最好的语言来编写DLL。也许你的应用程序的用户界面使用Microsoft Visual Basic编写得最好，但是用C++来处理它的商用逻辑更好。系统允许Visual Basic程序加载C++ DLL、Cobol DLL和Fortran DLL等。
- 它们简化了软件项目的管理。如果在软件开发过程中不同的工作小组在不同的模块上工作，那么这个项目管理起来比较容易。但是，应用程序在销售时附带的文件应该尽量少一些。我知道有一家公司销售的产品附带了100个DLL——每个程序员最多有5个DLL。这样，应用程序的初始化时间将会长得吓人，因为系统必须打开100个磁盘文件之后，程序才能执行它的操作。
- 它们有助于节省内存。如果两个或多个应用程序使用同一个DLL，那么该DLL的页面只要放入RAM一次，所有的应用程序都可以共享它的各个页面。C/C++运行期库就是个极好的例子。许多应用程序都使用这个库。如果所有的应用程序都链接到这个静态库，那么sprintf、strcpy和malloc等函数的代码就要多次存在于内存中。但是，如果所有这些应用程序链接到DLL C/C++运行期库，那么这些函数的代码就只需要放入内存一次，这意味着内存的使用将更加有效。
- 它们有助于资源的共享。DLL可以包含对话框模板、字符串、图标和位图等资源。多个应用程序能够使用DLL来共享这些资源。
- 它们有助于应用程序的本地化。应用程序常常使用DLL对自己进行本地化。例如，只包含代码而不包含用户界面组件的应用程序可以加载包含本地化用户界面组件的DLL。
- 它们有助于解决平台差异。不同版本的Windows配有不同的函数。开发人员常常想要调用新的函数（如果它们存在于主机的Windows版本上的话）。但是，如果你的源代码包含

了对一个新函数的调用，而你的应用程序将要在不能提供该函数的 Windows 版本上运行，那么操作系统的加载程序将拒绝运行你的进程。即使你实际上从不调用该函数，情况也是这样。如果将这些新函数保存在 DLL 中，那么应用程序就能够将它们加载到 Windows 的老版本上。当然，你仍然可以成功地调用该函数。

- 它们可以用于一些特殊的目的。Windows 使得某些特性只能为 DLL 所用。例如，只有当 DLL 中包含某个挂钩通知函数的时候，才能安装某些挂钩（使用 SetWindowsHookEx 和 SetWinEventHook 来进行安装）。可以通过创建必须在 DLL 中生存的 COM 对象来扩展 Windows Explorer 的外壳程序。对于可以由 Web 浏览器加载的、用于创建内容丰富的 Web 页的 ActiveX 控件来说，情况也是一样。

19.1 DLL 与进程的地址空间

创建 DLL 常常比创建应用程序更容易，因为 DLL 往往包含一组应用程序可以使用的自主函数。在 DLL 中通常没有用来处理消息循环或创建窗口的支持代码。DLL 只是一组源代码模块，每个模块包含了应用程序（可执行文件）或另一个 DLL 将要调用的一组函数。当所有源代码文件编译后，它们就像应用程序的可执行文件那样被链接程序所链接。但是，对于一个 DLL 来说，你必须设定该链接程序的 /DLL 开关。这个开关使得链接程序能够向产生的 DLL 文件映像发出稍有不同的信息，这样，操作系统加载程序就能将该文件映像视为一个 DLL 而不是应用程序。

在应用程序（或另一个 DLL）能够调用 DLL 中的函数之前，DLL 文件映像必须被映射到调用进程的地址空间中。若要进行这项操作，可以使用两种方法中的一种，即加载时的隐含链接或运行期的显式链接。隐含链接将在本章的后面部分介绍，显式链接将在第 20 章中介绍。

一旦 DLL 的文件映像被映射到调用进程的地址空间中，DLL 的函数就可以供进程中运行的所有线程使用。实际上，DLL 几乎将失去它作为 DLL 的全部特征。对于进程中的线程来说，DLL 的代码和数据看上去就像恰巧是在进程的地址空间中的额外代码和数据一样。当一个线程调用 DLL 函数时，该 DLL 函数要查看线程的堆栈，以便检索它传递的参数，并将线程的堆栈用于它需要的任何局部变量。此外，DLL 中函数的代码创建的任何对象均由调用线程所拥有，而 DLL 本身从来都不拥有任何东西。

例如，如果 VirtualAlloc 函数被 DLL 中的一个函数调用，那么将从调用线程的进程地址空间中保留一个地址空间的区域，该地址空间区域将始终处于保留状态，因为系统并不跟踪 DLL 中的函数保留该区域的情况。保留区域由进程所拥有，只有在线程调用 VirtualFree 函数或者进程终止运行时才被释放。

如你所知，可执行文件的全局变量和静态变量不能被同一个可执行文件的多个运行实例共享。Windows 98 能够确保这一点，方法是在可执行文件被映射到进程的地址空间时为可执行文件的全局变量和静态变量分配相应的存储器。Windows 2000 确保这一点的方法是使用第 13 章介绍的写入时拷贝（copy-on-write）机制。DLL 中的全局变量和静态变量的处理方法是完全相同的。当一个进程将 DLL 的映像文件映射到它的地址空间中去时，系统将同时创建全局数据变量和静态数据变量的实例。

注意 必须注意的是，单个地址空间是由一个可执行模块和若干个 DLL 模块组成的。这些模块中，有些可以链接到静态版本的 C/C++ 运行期库，有些可以链接到一个 DLL 版本的 C/C++ 运行期库，而有些模块（如果不是用 C/C++ 编写的话）则根本不需要 C/C++ 运行期库。许多开发人员经常会犯一个常见的错误，因为他们忘记了若干个 C/C++ 运行期库可以存在于单个地址空间中。请看下面的代码：

```
VOID EXEFunc() {
    PVOID pv = DLLFunc();
    // Access the storage pointed to by pv...
    // Assumes that pv is in EXE's C/C++ run-time heap
    free(pv);
}

PVOID DLLFunc() {
    // Allocate block from DLL's C/C++ run-time heap
    return(malloc(100));
}
```

那么你是怎么看待这个问题的呢？上面这个代码能够正确运行吗？DLL函数分配的内存块是由EXE的函数释放的吗？答案是可能的。上面显示的代码并没有为你提供足够的信息。如果EXE和DLL都链接到DLL的C/C++运行期库，那么上面的代码将能够很好地运行。但是，如果两个模块中的一个或者两个都链接到静态C/C++运行期库，那么对free函数的调用就会失败。我经常看到编程人员编写这样的代码，结果都失败了。

有一个很方便的方法可以解决这个问题。当一个模块提供一个用于分配内存块的函数时，该模块也必须提供释放内存的函数。让我们将上面的代码改写成下面的样子：

```
VOID EXEFunc() {
    PVOID pv = DLLFunc();
    // Access the storage pointed to by pv...
    // Makes no assumptions about C/C++ run-time heap
    DLLFreeFunc(pv);
}

PVOID DLLFunc() {
    // Allocate block from DLL's C/C++ run-time heap
    PVOID pv = malloc(100);
    return(pv);
}

BOOL DLLFreeFunc(PVOID pv) {
    // Free block from DLL's C/C++ run-time heap
    return(free(pv));
}
```

这个代码是正确的，它始终都能正确地运行。当你编写一个模块时，不要忘记其他模块中的函数也许没有使用C/C++来编写，因此可能无法使用malloc和free函数进行内存的分配。应该注意不要在代码中使用这些假设条件。另外，在内部调用 malloc和free函数时，这个原则对于C++的new和delete操作符也是适用的。

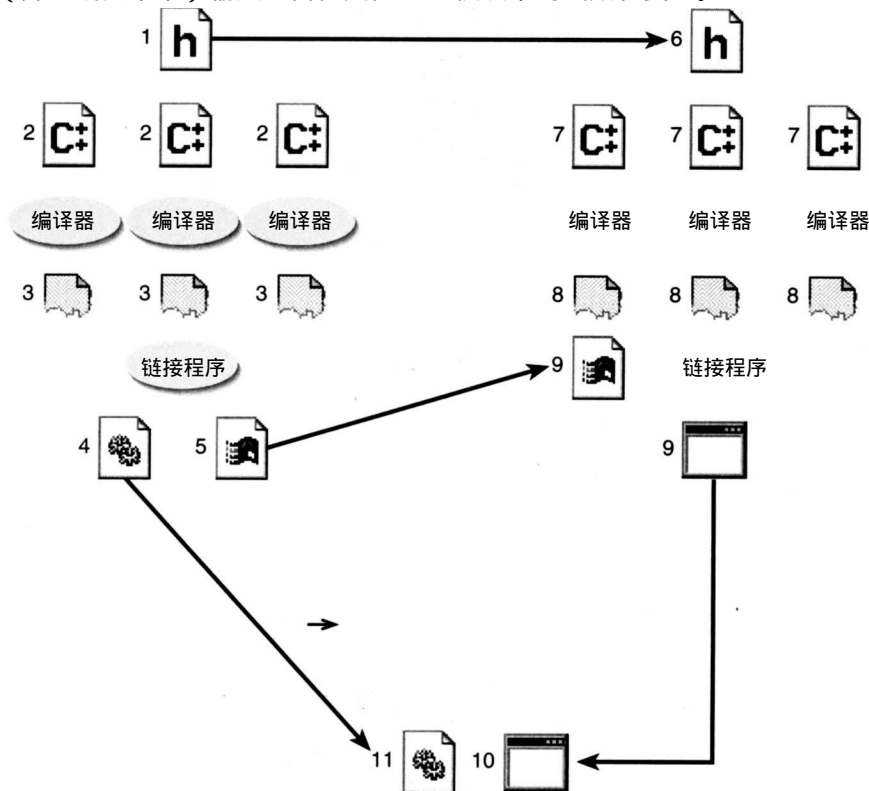
19.2 DLL的总体运行情况

为了全面理解DLL是如何运行的以及你和系统如何使用DLL，让我们首先观察一下DLL的整个运行情况。图19-1综合说明了它的所有组件一道配合运行的情况。

现在要重点介绍可执行模块和DL模块之间是如何隐含地互相链接的。隐含链接是最常用的链接类型。Windows也支持显式链接（第20章介绍这个问题）。

在图19-1中你可以看到，当一个模块（比如一个可执行文件）使用DLL中的函数或变量时，将有若干个文件和组件参与发挥作用。为了简单起见，我将“可执行模块”称为来自DLL的输

入函数和变量，将“DLL模块”称为用于可执行模块的输出函数和变量。但是要记住，DLL模块能够（并且确实常常）输入包含在其他DLL模块中的函数和变量。



创建DLL：

- 1) 建立带有输出原型/结构/符号的头文件。
- 2) 建立实现输出函数/变量的C/C++源文件。
- 3) 编译器为每个C/C++源文件生成 .obj模块。
- 4) 链接程序将生成DLL的 .obj模块链接起来。
- 5) 如果至少输出一个函数/变量，那么链接程序也生成lib 文件。

创建EXE：

- 6) 建立带有输入原型/结构/符号的头文件。
- 7) 建立引用输入函数/变量的C/C++源文件。
- 8) 编译器为每个C/C++源文件生成 .obj源文件。
- 9) 链接程序将各个 .obj模块链接起来，产生一个 .exe文件(它包含了所需要DLL模块的名字和输入符号的列表)。

运行应用程序：

- 10) 加载程序为 .exe 创建地址空间。
- 11) 加载程序将需要的DLL加载到地址空间中进程的主线程开始执行；应用程序启动运行。

图19-1 应用程序如何创建和隐含链接DLL的示意图

若要创建一个从DLL模块输入函数和变量的可执行模块，必须首先创建一个DLL模块。然后就可以创建可执行模块。

若要创建DLL模块，必须执行下列操作步骤：

1) 首先必须创建一个头文件，它包含你想要从DLL输出的函数原型、结构和符号。DLL的所有源代码模块均包含该头文件，以帮助创建DLL。后面将会看到，当创建需要使用DLL中包含的函数和变量的可执行模块（或多个模块）时，也需要这个头文件。

2) 要创建一个C/C++源代码模块（或多个模块），用于实现你想要在DLL模块中实现的函数和变量。由于这些源代码模块在创建可执行模块时是不必要的，因此创建DLL的公司能够保

护公司的秘密。

3) 创建DLL模块, 将使编译器对每个源代码模块进行处理, 产生一个 .obj模块 (每个源代码模块有一个 .obj模块)。

4) 当所有的 .obj模块创建完成后, 链接程序将所有 .obj模块的内容组合在一起, 产生一个DLL映像文件。该映像文件 (即模块) 包含了用于DLL的所有二进制代码和全局/静态数据变量。为了执行这个可执行模块, 该文件是必不可少的。

5) 如果链接程序发现DLL的源代码模块至少输出了一个函数或变量, 那么链接程序也生成一个 .lib文件。这个 .lib文件很小, 因为它不包含任何函数或变量。它只是列出所有已输出函数和变量的符号名。为了创建可执行模块, 该文件是必不可少的。

一旦创建了DLL模块, 就可以创建可执行模块。其创建步骤是:

6) 在引用函数、变量、数据、结构或符号的所有源代码模块中, 必须包含DLL开发人员创建的头文件。

7) 要创建一个C/C++源代码模块 (或多个模块), 用于实现你想要在可执行模块中实现的函数和变量。当然该代码可以引用DLL头文件中定义的函数和变量。

8) 创建可执行模块, 将使编译器对每个源代码模块进行处理, 生成一个 .obj模块 (每个源代码模块有一个 .obj模块)。

9) 当所有 .obj模块创建完成后, 链接程序便将所有的 .obj模块的内容组合起来, 生成一个可执行的映像文件。该映像文件 (或模块) 包含了可执行文件的所有二进制代码和全局/静态变量。该可执行模块还包含一个输入节, 列出可执行文件需要的所有DLL模块名 (关于各个节的详细说明, 参见第17章)。此外, 对于列出的每个DLL名字, 该节指明了可执行模块的二进制代码引用了哪些函数和变量符号。下面你会看到操作系统的加载程序将对该输入节进行分析。

一旦DLL和可执行模块创建完成, 一个进程就可以执行。当试图运行可执行模块时, 操作系统的加载程序将执行下面的操作步骤:

10) 加载程序为新进程创建一个虚拟地址空间。可执行模块被映射到新进程的地址空间。加载程序对可执行模块的输入节进行分析。对于该节中列出的每个DLL名字, 加载程序要找出用户系统上的DLL模块, 再将该DLL映射到进程的地址空间。注意, 由于DLL模块可以从另一个DLL模块输入函数和变量, 因此DLL模块可以拥有它自己的输入节。若要对进程进行全面的初始化, 加载程序要分析每个模块的输入节, 并将所有需要的DLL模块映射到进程的地址空间。如你所见, 对进程进行初始化是很费时间的。

一旦可执行模块和所有DLL模块被映射到进程的地址空间中, 进程的主线程就可以启动运行, 同时应用程序也可以启动运行。下面各节将更加详细地介绍这个进程的运行情况。

19.3 创建DLL模块

当创建DLL时, 要创建一组可执行模块 (或其他DLL) 可以调用的函数。DLL可以将变量、函数或C/C++类输出到其他模块。在实际工作环境中, 应该避免输出变量, 因为这会删除你的代码中的一个抽象层, 使它更加难以维护你的DLL代码。此外, 只有当使用同一个供应商提供的编译器对输入C++类的模块进行编译时, 才能输出C++类。由于这个原因, 也应该避免输出C++类, 除非知道可执行模块的开发人员使用的工具与DLL模块开发人员使用的工具相同。

当创建DLL模块时, 首先应该建立一个头文件, 该文件包含了你想要输出的变量 (类型和名字) 和函数 (原型和名字)。头文件还必须定义用于输出函数和变量的任何符号和数据结构。你的DLL的所有源代码模块都应该包含这个头文件。另外, 必须分配该头文件, 以便它能够包

含在可能输入这些函数或变量的任何源代码中。拥有单个头文件，供 DLL 创建程序和可执行模块的创建程序使用，就可以大大简化维护工作。

下面的代码说明了应该如何对单个头文件进行编码，以便同时包含可执行文件和 DLL 的源代码文件：

```

/*****
Module:  MyLib.h
*****/

#ifdef MYLIBAPI

// MYLIBAPI should be defined in all of the DLL's source
// code modules before this header file is included.

// All functions/variables are being exported.

#else

// This header file is included by an EXE source code module.
// Indicate that all functions/variables are being imported.
#define MYLIBAPI extern "C" __declspec(dllimport)

#endif

////////////////////////////////////
// Define any data structures and symbols here.

////////////////////////////////////

// Define exported variables here. (NOTE: Avoid exporting variables.)
MYLIBAPI int g_nResult;

////////////////////////////////////

// Define exported function prototypes here.
MYLIBAPI int Add(int nLeft, int nRight);

//////////////////////////////////// End of File //////////////////////////////////

```

在你的每个 DLL 源代码文件中，应该包含下面的头文件：

```

/*****
Module:  MyLibFile1.cpp
*****/

// Include the standard Windows and C-Runtime header files here.
#include <windows.h>

// This DLL source code file exports functions and variables.
#define MYLIBAPI extern "C" __declspec(dllexport)

// Include the exported data structures, symbols, functions, and variables.
#include "MyLib.h"

```

```
////////////////////////////////////
```

```
// Place the code for this DLL source code file here.  
int g_nResult;
```

```
int Add(int nLeft, int nRight) {  
    g_nResult = nLeft + nRight;  
    return(g_nResult);  
}
```

```
//////////////////////////////////// End of File //////////////////////////////////
```

当上面的DLL源代码文件被编译时,在MyLib.h头文件的前面使用__declspec(dllexport)对MYLIBAPI进行定义。当编译器看到负责修改变量、函数或C++类的__declspec(dllexport)时,它就知道该变量、函数或C++类是从产生的DLL模块输出的。注意,MYLIBAPI标志被置于头文件中要输出的变量的定义之前和要输出的函数之前。

另外,在源代码文件(MyLibFile1.cpp0)中,MYLIBAPI标志并不出现在输出的变量和函数之前。MYLIBAPI标志在这里是不必要的,因为编译器在分析头文件时能够记住要输出哪些变量或函数。

你会发现,MYLIBAPI标志包含了extern“C”修改符。只有当你编写C++代码而不是直接编写C代码时,才能使用这个修改符。通常来说,C++编译器可能会改变函数和变量的名字,从而导致严重的链接程序问题。例如,假设你用C++编写一个DLL,并直接用C编写一个可执行模块,当你创建DLL时,函数名被改变,但是,当你创建可执行模块时,函数名没有改变。当链接程序试图链接可执行模块时,它就会抱怨说,可执行模块引用的符号不存在。如果使用extern“C”,就可以告诉编译器不要改变变量名或函数名,这样,变量和函数就可以供使用C、C++或任何其他编程语言编写的可执行模块来访问。

现在你已经知道DLL源代码文件是如何使用这个头文件的。但是,可执行模块的源代码文件情况又是如何呢?可执行模块的源代码文件不应该在这个头文件的前面定义MYLIBAPI。由于MYLIBAPI没有定义,因此头文件将MYLIBAPI定义为__declspec(dllimport)。编译器看到可执行模块的源代码文件从DLL模块输入变量和函数。

如果观察Microsoft的标准Windows头文件,如WinBase.h,你将会发现Microsoft使用的方法基本上与上面介绍的方法相同。

19.3.1 输出的真正含义是什么

上一节介绍的一个真正有意思的东西是__declspec(dllexport)修改符。当Microsoft的C/C++编译器看到变量、函数原型或C++类之前的这个修改符的时候,它就将某些附加信息嵌入产生的.obj文件中。当链接DLL的所有.obj文件时,链接程序将对这些信息进行分析。

当DLL被链接时,链接程序要查找关于输出变量、函数或C++类的信息,并自动生成一个.lib文件。该.lib文件包含一个DLL输出的符号列表。当然,如果要链接引用该DLL的输出符号的任何可执行模块,该.lib文件是必不可少的。除了创建.lib文件外,链接程序还要将一个输出符号表嵌入产生的DLL文件。这个输出节包含一个输出变量、函数和类符号的列表(按字母顺序排列)。该链接程序还将能够指明在何处找到每个符号的相对虚拟地址(RVA)放入DLL模块。

使用Microsoft的Visual Studio的DumpBin.exe实用程序(带有-exports开关),你能够看到

DLL的输出节是个什么样子。下面是 Kernel32.dll的输出节的一个代码段（我已经删除了DUMPBIN的某些输出，这样就不会占用本书的太多篇幅）。

```
C:\WINNT\SYSTEM32>DUMPBIN -exports Kernel32.DLL
```

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

```
Dump of file kernel32.dll
```

```
File Type: DLL
```

```
Section contains the following exports for KERNEL32.dll
```

```
    0 characteristics
36DB3213 time date stamp Mon Mar 01 16:34:27 1999
    0.00 version
    1 ordinal base
    829 number of functions
    829 number of names
```

ordinal	hint	RVA	name
1	0	0001A3C6	AddAtomA
2	1	0001A367	AddAtomW
3	2	0003F7C4	AddConsoleAliasA
4	3	0003F78D	AddConsoleAliasW
5	4	0004085C	AllocConsole
6	5	0002C91D	AllocateUserPhysicalPages
7	6	00005953	AreFileApisANSI
8	7	0003F1A0	AssignProcessToJobObject
9	8	00021372	BackupRead
10	9	000215CE	BackupSeek
11	A	00021F21	BackupWrite
:			
828	33B	00003200	lstrlenA
829	33C	000040D5	lstrlenW

```
Summary
```

```
3000 .data
4000 .reloc
4D000 .rsrc
59000 .text
```

如你所见，这些符号是按字母顺序排列的，RVA这一列下面的数字用于指明在DLL文件映像中的什么位置能够找到输出符号的位移量。序号列可以与16位Windows源代码向后兼容，并且它不应该用于现在的应用程序中。hint（提示码）列可供系统用来改进代码的运行性能，在此并不重要。

注意 许多开发人员常常通过为函数赋予一个序号值来输出DLL函数。对于那些来自16位Windows环境的函数来说，情况尤其是如此。但是，Microsoft并没有公布系统DLL的序号值。当你的可执行模块或DLL模块链接到任何一个Windows函数时，

Microsoft要求你使用符号的名字进行链接。如果你按照序号进行链接，那么你的应用程序有可能无法在其他Windows平台或将来的Windows平台上运行。

实际上，我就遇到过这样的情况。我曾经发布了一个示例应用程序，它使用Microsoft System Journal中的序号。我的应用程序在Windows NT 3.1上运行得很好，但是当Windows NT 3.5推出时，我的应用程序就无法正确地运行。为了解决这个问题，我不得不用函数名代替序号。现在该应用程序既能够在Windows NT 3.1上运行，而且能够在所有更新的版本上运行。

我问过Microsoft公司，为什么它不使用序号，我得到的回答是：“我们认为可移植的可执行文件格式不仅具有序号的优点（查找迅速），而且提供了按名字输入的灵活性。我们可以随时增加函数。在带有多个实现代码的大型程序项目中，序号很难管理。”

你可以将序号用于你创建的任何DLL，并且按照序号将你的可执行文件链接到这些DLL。Microsoft保证，即使在将来的操作系统版本中，这个方法也是可行的。但是我在我的工作中总是避免使用序号，并且从现在起只按名字进行链接。

19.3.2 创建用于非Visual C++工具的DLL

如果使用Microsoft Visual C++来创建DLL和将要链接到该DLL的可执行模块，可以跳过本节内容的学习。但是，如果使用Visual C++创建DLL，而这个DLL要链接到使用任何供应商的工具创建的可执行模块，那么必须做一些额外的工作。

前面讲过当进行C和C++混合编程时使用extern “C”修改符的问题。也讲过C++类的问题以及为什么因为名字改变的缘故你必须使用同一个编译器供应商的工具的问题。当你直接将C语言编程用于多个工具供应商时将会出现另一个问题。这个问题是，即使你根本不使用C++，Microsoft的C编译器也会损害C函数。当你的函数使用__stdcall(WINAPI)调用规则时会出现这种问题。这种调用规则是最流行的一种类型。当使用__stdcall将C函数输出时，Microsoft的编译器就会改变函数的名字，设置一个前导下划线，再加上一个@符号的前缀，后随一个数字，表示作为参数传递给函数的字节数。例如，下面的函数是作为DLL的输出节中的_MyFunc@8输出的：

```
__declspec(dllexport) LONG __stdcall MyFunc(int a, int b);
```

如果用另一个供应商的工具创建了一个可执行模块，它将设法链接到一个名叫MyFunc的函数，该函数在Microsoft编译器已有的DLL中并不存在，因此链接将失败。

若要使用与其他编译器供应商的工具链接的Microsoft的工具创建一个可执行模块，必须告诉Microsoft的编译器输出没有经过改变的函数名。可以用两种方法来进行这项操作。第一种方法是为编程项目建立一个.def文件，并在该.def文件中加上类似下面的EXPORTS节：

```
EXPORTS
MyFunc
```

当Microsoft的链接程序分析这个.def文件时，它发现_MyFunc@8和MyFunc均被输出。由于这两个函数名是互相匹配的（除了截断的尾部外），因此链接程序使用MyFunc的.def文件名来输出该函数，而根本不使用_MyFunc@8的名字来输出函数。

现在你可能认为，如果使用Microsoft的工具创建一个可执行模块，并且设法将它链接到包含未截断名字的DLL，那么链接程序的运行将会失败，因为它将试图链接到称为_MyFunc@8的函数。当然，你会高兴地了解到Microsoft的链接程序进行了正确的操作，将可执行模块链接到名字为MyFunc的函数。

如果想避免使用 .def 文件，可以使用第二种方法输出未截断的函数版本。在 DLL 的源代码模块中，可以添加下面这行代码：

```
#pragma comment(linker, "/export:MyFunc=_MyFunc@8")
```

这行代码使得编译器发出一个链接程序指令，告诉链接程序，一个名叫 MyFunc 的函数将被输出，其进入点与称为 _MyFunc@8 的函数的进入点相同。第二种方法没有第一种方法容易，因为你必须自己截断函数名，以便创建该代码行。另外，当使用第二种方法时，DLL 实际上输出用于标识单个函数的两个符号，即 MyFunc 和 _MyFunc@8，而第一种方法只输出符号 MyFunc。第二种方法并没有给你带来更多的好处，它只是使你可以避免使用 .def 的文件而已。

19.4 创建可执行模块

下面的代码段显示了一个可执行的源代码文件，它输入了 DLL 的输出符号，并且在代码中引用了这些符号。

```

/*****
Module:  MyExeFile1.cpp
*****/

// Include the standard Windows and C-Runtime header files here.
#include <windows.h>

// Include the exported data structures, symbols, functions, and variables.
#include "MyLib\MyLib.h"

////////////////////////////////////

int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

    int nLeft = 10, nRight = 25;

    TCHAR sz[100];
    wsprintf(sz, TEXT("%d + %d = %d"), nLeft, nRight, Add(nLeft, nRight));
    MessageBox(NULL, sz, TEXT("Calculation"), MB_OK);

    wsprintf(sz, TEXT("The result from the last Add is: %d"), g_nResult);
    MessageBox(NULL, sz, TEXT("Last Result"), MB_OK);
    return(0);
}

//////////////////////////////////// End of File //////////////////////////////////////

```

当创建可执行源代码文件时，必须加上 DLL 的头文件。如果没有头文件，输入的符号将不会被定义，而且编译器将会发出许多警告和错误消息。

可执行源代码文件不应该定义 DLL 的头文件前面的 MYLIBAPI。当上面显示的这个可执行源代码文件被编译时，MYLIBAPI 由 MyLib.h 头文件使用 __declspec(dllimport) 进行定义。当编译器看到修改变量、函数或 C++ 类的 __declspec(dllimport) 时，它知道这个符号是从某个 DLL 模块输入的。它不知道是从哪个 DLL 模块输入的，并且它也不关心这个问题。编译器只想确保你用正确的方法访问这些输入的符号。现在你在源代码中可以引用输入的符号，一切都将能够正常工作。

接着，链接程序必须将所有 .obj 模块组合起来，创建产生的可执行模块。该链接程序必须确定哪些 DLL 包含代码引用的所有输入符号的 DLL。因此你必须将 DLL 的 .lib 文件传递给链接程序。如前所述，.lib 文件只包含 DLL 模块输出的符号列表。链接程序只想知道是否存在引用的符号和哪个 DLL 模块包含该符号。如果连接程序转换了所有外部符号的引用，那么可执行模块就因此而产生了。

输入的真正含义是什么

上一节介绍了修改符 `--declspec(dllimport)`。当输入一个符号时，不必使用关键字 `--declspec(dllimport)`，只要使用标准的 C 关键字 `extern` 即可。但是，如果编译器预先知道你引用的符号将从一个 DLL 的 .lib 文件输入，那么编译器就能够生成运行效率稍高的代码。因此建议你尽量将 `--declspec(dllimport)` 关键字用于输入函数和数据符号。当你调用标准 Windows 函数中的任何一个时，Microsoft 将为你进行这项设置。

当链接程序进行输入符号的转换时，它就将一个称为输入节的特殊的节嵌入产生的可执行模块。输入节列出了该模块需要的 DLL 模块以及由每个 DLL 模块引用的符号。

使用 Visual Studio 的 DumpBin.exe 实用程序（带有 `-imports` 开关），能够看到模块的输入节的样子。下面是 Calc.exe 文件的输入节的一个代码段（同样，我删除了 DUMPBIN 的某些输出，这样它就不会占用太多的篇幅）。

```
C:\WINNT\SYSTEM32>DUMPBIN -imports Calc.EXE
```

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

```
Dump of file calc.exe
```

```
File Type: EXECUTABLE IMAGE
```

```
Section contains the following imports:
```

```
SHELL32.dll
```

```
10010F4 Import Address Table
1012820 Import Name Table
FFFFFFFF time date stamp
FFFFFFFF Index of first forwarder reference
```

```
77C42983 7A ShellAboutW
```

```
MSVCRT.dll
```

```
1001094 Import Address Table
10127C0 Import Name Table
FFFFFFFF time date stamp
FFFFFFFF Index of first forwarder reference
```

```
78010040 295 memmove
78018124 42 _EH_prolog
78014C34 2D1 toupper
78010F6E 2DD wcschr
78010668 2E3 wcslen
```

```
:
```

```

ADVAPI32.dll
    1001000 Import Address Table
    101272C Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference

779858F4  19A  RegQueryValueExA
77985196  190  RegOpenKeyExA
77984BA1  178  RegCloseKey

KERNEL32.dll
    100101C Import Address Table
    1012748 Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference

77ED4134  336  lstrcpyW
77ED33E8  1E5  LocalAlloc
77EDEF36   DB  GetCommandLineW
77ED1610  15E  GetProfileIntW
77ED4BA4  1EC  LocalReAlloc

```

```

:
```

Header contains the following bound import information:

```

Bound to SHELL32.dll [36E449E0] Mon Mar 08 14:06:24 1999
Bound to MSVCRT.dll [36BB8379] Fri Feb 05 15:49:13 1999
Bound to ADVAPI32.dll [36E449E1] Mon Mar 08 14:06:25 1999
Bound to KERNEL32.dll [36DDAD55] Wed Mar 03 13:44:53 1999
Bound to GDI32.dll [36E449E0] Mon Mar 08 14:06:24 1999
Bound to USER32.dll [36E449E0] Mon Mar 08 14:06:24 1999

```

Summary

```

2000 .data
3000 .rsrc
13000 .text

```

如你所见，这一节为 Calc.exe 需要的每个 DLL 设置了一个项目，这些 DLL 是 Shell32.dll、MSVCRT.dll、AdvAPI32.dll、Kernel32.dll、GDI32.dll 和 User32.dll。在每个 DLL 的模块名下面，有一个 Calc.exe 从该特定模块输入的符号列表。例如，Calc 模块调用包含在 Kernel32.dll 中的下列函数：lstrcpyW、LocalAlloc、GetCommandLineW 和 GetProfileIntW 等。

紧靠符号名左边的数字是符号的提示（hint）值，它与讨论无关。每个符号行最左边的数字用于指明该符号在进程的地址空间中所在的内存地址。该内存地址只有在可执行模块相链接时才出现。在 DumpBin 的输出结尾处，可以看到更多的链接信息。

19.5 运行可执行模块

当一个可执行文件被启动时，操作系统加载程序将为该进程创建虚拟地址空间。然后，加载程序将可执行模块映射到进程的地址空间中。加载程序查看可执行模块的输入节，并设法找出任何需要的 DLL，并将它们映射到进程的地址空间中。

- 1) 包含可执行映像文件的目录。
- 2) 进程的当前目录。
- 3) Windows系统目录。
- 4) Windows目录。
- 5) PATH环境变量中列出的各个目录。

如果加载程序无法找到需要的 DLL 模块，用户会看到图 19-2、图 19-3 所示的消息框中的一个：如果是 Windows 2000，那么将出现图 19-2 所示的消息框，如果是 Windows 98，则出现图 19-3 所示的消息框。

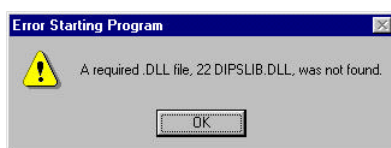


图19-3 Windows 98下加载程序
搜索DLL时的消息框




图19-5 Windows 98下加载程序
查看DLL时出现的消息框

如果这个符号不存在，那么加载程序将要检索该符号的 RVA，并添加DLL模块被加载到的虚拟地址空间（符号在进程的地址空间中的位置）。然后它将该虚拟地址保存在可执行模块的输入节中。这时，当代码引用一个输入符号时，它将查看调用模块的输入节，并且捕获输入符号的地址，这样它就能够成功地访问输入变量、函数或 C++ 类的成员函数。好了，动态链接完成，进程的主线程开始执行，应用程序终于也开始运行了！

当然，这需要加载程序花费相当多的时间来加载这些 DLL 模块，并用所有使用输入符号的正确地址来调整每个模块的输入节。由于所有这些工作都是在进程初始化的时候进行的，因此应用程序运行期的性能不会降低。不过，对于许多应用程序来说，初始化的速度太慢是不行的。为了缩短应用程序的加载时间，应该调整你的可执行模块和 DLL 模块的位置并且将它们连接起来。真可惜很少有开发人员知道如何进行这项操作，因为这些技术是非常重要的。如果每个公司都能够使用这些技术，系统将能运行的更好。实际上，我认为操作系统销售时应该配有一个能够自动执行这些操作的实用程序。下一章将要介绍对模块调整位置和进行连接的方法。