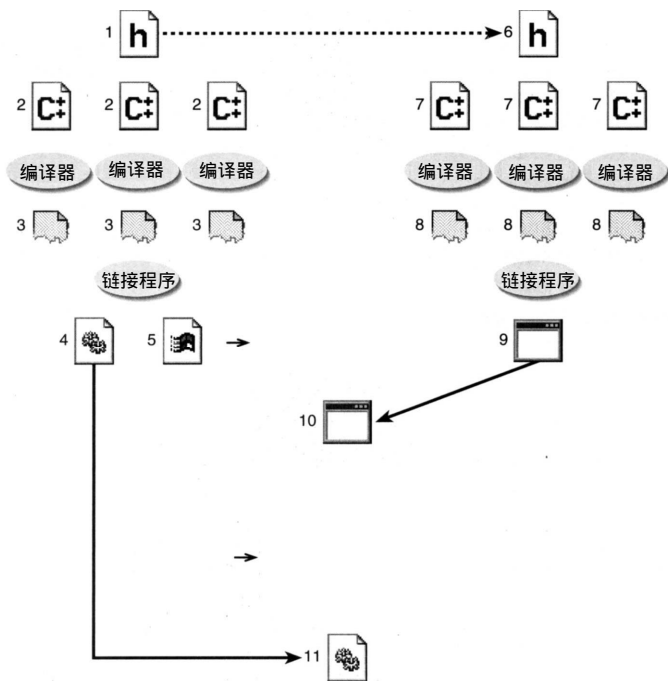


第20章 DLL的高级操作技术

上一章介绍了DLL链接的基本方法，并且重点说明了隐含链接的技术，这是 DLL链接的最常用的形式。虽然对于大多数应用程序来说，只要了解上一章介绍的知识就足够了，但是还可以使用DLL进行更多的工作。本章将要介绍与DLL相关的各种操作方法。大多数应用程序不一定需要这些方法，但是它们是非常有用的，所以应该对它们有所了解。

20.1 DLL模块的显式加载和符号链接

如果线程需要调用DLL模块中的函数，那么DLL的文件映像必须映射到调用线程的进程地



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(视情况而定)。
 - 7) 建立不引用输入函数/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，生成 .exe文件。
- 注：DLL的lib文件是不需要的，因为并不直接引用输出符号。 .exe 文件不包含输入表。

运行应用程序：

- 10) 加载程序为 .exe 创建模块地址空进程的主线程开始执行；应用程序启动运行。

显式加载DLL：

- 11) 一个线程调用LoadLibrary (Ex)函数，将DLL加载到进程的地址空间这时线程可以调用GetProcAddress以便间接引用DLL的输出符号。

图20-1 应用程序创建和显式链接DLL的示意图

址空间中。可以用两种方法进行这项操作。第一种方法是让应用程序的源代码只引用 DLL 中包含的符号。这样,当应用程序启动运行时,加载程序就能够隐含加载(和链接)需要的 DLL。

第二种方法是在应用程序运行时让应用程序显式加载需要的 DLL 并且显式链接到需要的输出符号。换句话说,当应用程序运行时,它里面的线程能够决定它是否要调用 DLL 中的函数。该线程可以将 DLL 显式加载到进程的地址空间,获得 DLL 中包含的函数的虚拟内存地址,然后使用该内存地址调用该函数。这种方法的优点是一切操作都是在应用程序运行时进行的。

图20-1显示了一个应用程序是如何显式地加载 DLL 并且链接到它里面的符号的。

20.1.1 显式加载 DLL 模块

无论何时,进程中的线程都可以决定将一个 DLL 映射到进程的地址空间,方法是调用下面两个函数中的一个:

```
HINSTANCE LoadLibrary(PCTSTR pszDLLPathName);
```

```
HINSTANCE LoadLibraryEx(  
    PCTSTR pszDLLPathName,  
    HANDLE hFile,  
    DWORD dwFlags);
```

这两个函数均用于找出用户系统上的文件映像(使用上一章中介绍的搜索算法),并设法将 DLL 的文件映像映射到调用进程的地址空间中。两个函数返回的 HINSTANCE 值用于标识文件映像映射到的虚拟内存地址。如果 DLL 不能被映射到进程的地址空间,则返回 NULL。若要了解关于错误的详细信息,可以调用 GetLastError。

你会注意到,LoadLibraryEx 函数配有两个辅助参数,即 hFile 和 dwFlags。参数 hFile 保留供将来使用,现在必须是 NULL。对于参数 dwFlags,必须将它设置为 0,或者设置为 DONT_RESOLVE_DLL_REFERENCES、LOAD_LIBRARY_AS_DATAFILE 和 LOAD_WITH_ALTERED_SEARCH_PATH 等标志的一个组合。

1. DONT_RESOLVE_DLL_REFERENCES

DONT_RESOLVE_DLL_REFERENCES 标志用于告诉系统将 DLL 映射到调用进程的地址空间中。通常情况下,当 DLL 被映射到进程的地址空间中时,系统要调用 DLL 中的一个特殊函数,即DllMain(本章后面介绍)。该函数用于对 DLL 进行初始化。DONT_RESOLVE_DLL_REFERENCES 标志使系统不必调用 DllMain 函数就能映射文件映像。

此外,DLL 能够输入另一个 DLL 中包含的函数。当系统将一个 DLL 映射到进程的地址空间中时,它也要查看该 DLL 是否需要其他的 DLL,并且自动加载这些 DLL。当 DONT_RESOLVE_DLL_REFERENCES 标志被设定时,系统并不自动将其他的 DLL 加载到进程的地址空间中。

2. LOAD_LIBRARY_AS_DATAFILE

LOAD_LIBRARY_AS_DATAFILE 标志与 DONT_RESOLVE_DLL_REFERENCES 标志相类似,因为系统只是将 DLL 映射到进程的地址空间中,就像它是数据文件一样。系统并不花费额外的时间来准备执行文件中的任何代码。例如,当一个 DLL 被映射到进程的地址空间中时,系统要查看 DLL 中的某些信息,以确定应该将哪些页面保护属性赋予文件的不同的节。如果设定了 LOAD_LIBRARY_AS_DATAFILE 标志,系统将以它要执行文件中的代码时的同样方式来设置页面保护属性。

由于下面几个原因,该标志是非常有用的。首先,如果有一个 DLL(它只包含资源,但不

包含函数),那么可以设定这个标志,使DLL的文件映像能够映射到进程的地址空间中。然后可以在调用加载资源的函数时,使用LoadLibraryEx函数返回的HINSTANCE值。通常情况下,加载一个.exe文件,就能够启动一个新进程,但是也可以使用LoadLibraryEx函数将.exe文件的映像映射到进程的地址空间中。借助映射的.exe文件的HINSTANCE值,就能够访问文件中的资源。由于.exe文件没有DllMain函数,因此,当调用LoadLibraryEx来加载一个.exe文件时,必须设定LOAD_LIBRARY_AS_DATAFILE标志。

3. LOAD_WITH_ALTERED_SEARCH_PATH

LOAD_WITH_ALTERED_SEARCH_PATH标志用于改变LoadLibraryEx用来查找特定的DLL文件时使用的搜索算法。通常情况下,LoadLibraryEx按照第19章讲述的顺序进行文件的搜索。但是,如果设定了LOAD_WITH_ALTERED_SEARCH_PATH标志,那么LoadLibraryEx函数就按照下面的顺序来搜索文件:

- 1) pszDLLPathName参数中设定的目录。
- 2) 进程的当前目录。
- 3) Windows的系统目录。
- 4) Windows目录。
- 5) PATH环境变量中列出的目录。

20.1.2 显式卸载DLL模块

当进程中的线程不再需要DLL中的引用符号时,可以从进程的地址空间中显式卸载DLL,方法是调用下面的函数:

```
BOOL FreeLibrary(HINSTANCE hinstDll);
```

必须传递HINSTANCE值,以便标识要卸载的DLL。该值是较早的时候调用LoadLibrary(Ex)而返回的值。

也可以通过调用下面的函数从进程的地址空间中卸载DLL:

```
VOID FreeLibraryAndExitThread(  
    HINSTANCE hinstDll,  
    DWORD dwExitCode);
```

该函数是在Kernel32.dll中实现的,如下所示:

```
VOID FreeLibraryAndExitThread(HINSTANCE hinstDll, DWORD dwExitCode) {  
    FreeLibrary(hinstDll);  
    ExitThread(dwExitCode);  
}
```

初看起来,这并不是个非常高明的代码,你可能不明白,为什么Microsoft要创建FreeLibraryAndExitThread这个函数。其原因与下面的情况有关:假定你要编写一个DLL,当它被初次映射到进程的地址空间中时,该DLL就创建一个线程。当该线程完成它的操作时,它通过调用FreeLibrary函数,从进程的地址空间中卸载该DLL,并且终止运行,然后立即调用ExitThread。

但是,如果线程分开调用FreeLibrary和ExitThread,就会出现一个严重的问题。这个问题是调用FreeLibrary会立即从进程的地址空间中卸载DLL。当调用的FreeLibrary返回时,包含对ExitThread调用的代码就不再可以使用,因此线程将无法执行任何代码。这将导致访问违规,同时整个进程终止运行。

但是,如果线程调用FreeLibraryAndExitThread,该函数调用FreeLibrary,使DLL立即被卸

载。下一个执行的指令是在 Kernel32.dll 中，而不是在刚刚被卸载的 DLL 中。这意味着该线程能够继续执行，并且可以调用 ExitThread。ExitThread 使该线程终止运行并且不返回。

一般来说，并没有很大的必要去调用 FreeLibraryAndExitThread 函数。我曾经使用过一次，因为我执行了一个非常特殊的任务。另外，我为 Microsoft Windows 3.1 编写了一个代码，它并没有提供这个函数。因此我高兴地看到 Microsoft 将这个函数增加到了较新的 Windows 版本中。

在实际环境中，LoadLibrary 和 LoadLibraryEx 这两个函数用于对与特定的库相关的进程使用计数进行递增，FreeLibrary 和 FreeLibraryAndExitThread 这两个函数则用于对库的每个进程的使用计数进行递减。例如，当第一次调用 LoadLibrary 函数来加载 DLL 时，系统将 DLL 的文件映像映射到调用进程的地址空间中，并将 DLL 的使用计数设置为 1。如果同一个进程中的线程后来调用 LoadLibrary 来加载同一个 DLL 文件映像，系统并不第二次将 DLL 映像文件映射到进程的地址空间中，它只是将与该进程的 DLL 相关的使用计数递增 1。

为了从进程的地址空间中卸载 DLL 文件映像，进程中的线程必须两次调用 FreeLibrary 函数。第一次调用只是将 DLL 的使用计数递减为 1，第二次调用则将 DLL 的使用计数递减为 0。当系统发现 DLL 的使用计数递减为 0 时，它就从进程的地址空间中卸载 DLL 的文件映像。试图调用 DLL 中的函数的任何线程都会产生访问违规，因为特定地址上的代码不再被映射到进程的地址空间中。

系统为每个进程维护了一个 DLL 的使用计数，也就是说，如果进程 A 中的一个线程调用下面的函数，然后进程 B 中的一个线程调用相同的函数，那么 MyLib.dll 将被映射到两个进程的地址空间中，这样，进程 A 和进程 B 的 DLL 使用计数都将是 1。

```
HINSTANCE hinstDll = LoadLibrary("MyLib.dll");
```

如果进程 B 中的线程后来调用下面的函数，那么进程 B 的 DLL 使用计数将变成 0，并且该 DLL 将从进程 B 的地址空间中卸载。但是，进程 A 的地址空间中的 DLL 映射不会受到影响，进程 A 的 DLL 使用计数仍然是 1。

```
FreeLibrary(hinstDll);
```

如果调用 GetModuleHandle 函数，线程就能够确定 DLL 是否已经被映射到进程的地址空间中：

```
HINSTANCE GetModuleHandle(PCTSTR pszModuleName);
```

例如，只有当 MyLib.dll 尚未被映射到进程的地址空间中时，下面这个代码才能加载该文件：

```
HINSTANCE hinstDll = GetModuleHandle("MyLib"); // DLL extension assumed
if (hinstDll == NULL) {
    hinstDll = LoadLibrary("MyLib"); // DLL extension assumed
}
```

如果只有 DLL 的 HINSTANCE 值，那么可以调用 GetModuleFileName 函数，确定 DLL（或 .exe）的全路径名：

```
DWORD GetModuleFileName(
    HINSTANCE hinstModule,
    PTSTR pszPathName,
    DWORD cchPath);
```

第一个参数是 DLL（或 .exe）的 HINSTANCE。第二个参数 pszPathName 是该函数将文件映像的全路径名放入的缓存的地址。第三参数 cchPath 用于设定缓存的大小（以字符为计量单位）。

20.1.3 显式链接到一个输出符号

一旦 DLL 模块被显式加载，线程就必须获取它要引用的符号的地址，方法是调用下面的函数：

```
FARPROC GetProcAddress(  
    HINSTANCE hinstDll,  
    PCSTR pszSymbolName);
```

参数hinstDll是调用LoadLibrary(Ex)或GetModuleHandle函数而返回的，它用于设定包含符号的DLL的句柄。参数pszSymbolName可以采用两种形式。第一种形式是以0结尾的字符串的地址，它包含了你想要其地址的符号的名字：

```
FARPROC pfn = GetProcAddress(hinstDll, "SomeFuncInDll");
```

注意，参数pszSymbolName的原型是PCSTR，而不是PCTSTR。这意味着GetProcAddress函数只接受ANSI字符串，决不能将Unicode字符串传递给该函数，因为编译器/链接程序总是将符号名作为ANSI字符串存储在DLL的输出节中。

参数pszSymbolName的第二种形式用于指明你想要其地址的符号的序号：

```
FARPROC pfn = GetProcAddress(hinstDll, MAKEINTRESOURCE(2));
```

这种用法假设你知道你需要的符号名被DLL创建程序赋予了序号值2。同样，我要再次强调，Microsoft非常反对使用序号，因此你不会经常看到GetProcAddress的这个用法。

这两种方法都能够提供包含在DLL中的必要符号的地址。如果DLL模块的输出节中不存在你需要的符号，GetProcAddress就返回NULL，表示运行失败。

应该知道，调用GetProcAddress的第一种方法比第二种方法要慢，因为系统必须进行字符串的比较，并且要搜索传递的符号名字符串。对于第二种方法来说，如果传递的序号尚未被分配给任何输出的函数，那么GetProcAddress就会返回一个非NULL值。这个返回值将会使你的应用程序错误地认为你已经拥有一个有效的地址，而实际上你并不拥有这样的地址。如果试图调用该地址，肯定会导致线程引发一个访问违规。我在早期从事Windows编程时，并不完全理解这个行为特性，因此多次出现这样的错误。所以一定要小心（这个行为特性是应该避免使用序号而使用符号名的另一个原因）。

20.2 DLL的进入点函数

一个DLL可以拥有单个进入点函数。系统在不同的时间调用这个进入点函数，这个问题将在下面加以介绍。这些调用可以用来提供一些信息，通常用于供DLL进行每个进程或线程的初始化和清除操作。如果你的DLL不需要这些通知信息，就不必在DLL源代码中实现这个函数。例如，如果你创建一个只包含资源的DLL，就不必实现该函数。如果确实需要在DLL中接受通知信息，可以实现类似下面的进入点函数：

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {  
  
    switch (fdwReason) {  
        case DLL_PROCESS_ATTACH:  
            // The DLL is being mapped into the process's address space.  
            break;  
  
        case DLL_THREAD_ATTACH:  
            // A thread is being created.  
            break;  
  
        case DLL_THREAD_DETACH:  
            // A thread is exiting cleanly.  
            break;
```



```
case DLL_PROCESS_DETACH:
    // The DLL is being unmapped from the process's address space.
    break;
}
return(TRUE); // Used only for DLL_PROCESS_ATTACH
}
```

注意 函数名DllMain是区分大小写的。许多编程人员有时调用的函数是 DLLMain。这是一个非常容易犯的错误，因为 DLL这个词常常使用大写来表示。如果调用的进入点函数不是 DllMain，而是别的函数，你的代码将能够编译和链接，但是你的进入点函数永远不会被调用，你的DLL永远不会被初始化。

参数hinstDll包含了DLL的实例句柄。与(w)WinMain函数的hinstExe参数一样，这个值用于标识DLL的文件映像被映射到进程的地址空间中的虚拟内存地址。通常应将这个参数保存在一个全局变量中，这样就可以在调用加载资源的函数（如 DialogBox和LoadString）时使用它。最后一个参数是fImpLoad，如果DLL是隐含加载的，那么该参数将是个非0值，如果DLL是显式加载的，那么它的值是0。

参数fdwReason用于指明系统为什么调用该函数。该参数可以使用4个值中的一个。这4个值是：DLL_PROCESS_ATTACH、DLL_PROCESS_DETACH、DLL_THREAD_ATTACH或DLL_THREAD_DETACH。这些值将在下面介绍。

注意 必须记住，DLL使用DllMain函数来对它们进行初始化。当你的DllMain函数执行时，同一个地址空间中的其他DLL可能尚未执行它们的DllMain函数。这意味着它们尚未初始化，因此你应该避免调用从其他DLL中输入的函数。此外，你应该避免从DllMain内部调用LoadLibrary(Ex)和FreeLibrary函数，因为这些函数会形式一个依赖性循环。

Platform SDK文档说，你的DllMain函数只应该进行一些简单的初始化，比如设置本地存储器（第21章介绍），创建内核对象和打开文件等。你还必须避免调用 User、Shell、ODBC、COM、RPC和套接字函数（即调用这些函数的函数），因为它们的DLL也许尚未初始化，或者这些函数可能在内部调用LoadLibrary(Ex)函数，这同样会形成一个依赖性循环。

另外，如果创建全局性的或静态的C++对象，那么应该注意可能存在同样的问题，因为在你调用DllMain函数的同时，这些对象的构造函数和析构函数也会被调用。

20.2.1 DLL_PROCESS_ATTACH通知

当DLL被初次映射到进程的地址空间中时，系统将调用该DLL的DllMain函数，给它传递参数fdwReason的值DLL_PROCESS_ATTACH。只有当DLL的文件映像初次被映射时，才会出现这种情况。如果线程在后来为已经映射到进程的地址空间中的DLL调用LoadLibrary(Ex)函数，那么操作系统只是递增DLL的使用计数，它并不再次用DLL_PROCESS_ATTACH的值来调用DLL的DllMain函数。

当处理DLL_PROCESS_ATTACH时，DLL应该执行DLL中的函数要求的任何与进程相关的初始化。例如，DLL可能包含需要使用它们自己的堆栈（在进程的地址空间中创建）的函数。通过处理DLL_PROCESS_ATTACH通知时调用HeapCreate函数，该DLL的DllMain函数就能够创建这个堆栈。已经创建的堆栈的句柄可以保存在DLL函数有权访问的一个全局变量中。

当DllMain处理一个DLL_PROCESS_ATTACH通知时，DllMain的返回值能够指明DLL的初

始化是否已经取得成功。如果对 HeapCreate 的调用取得了成功, DllMain 应该返回 TRUE。如果堆栈不能创建, 它应该返回 FALSE。如果 fdwReason 使用的是其他的值, 即 DLL_PROCESS_DETACH、DLL_THREAD_ATTACH 和 DLL_THREAD_DETACH, 那么系统将忽略 DllMain 返回的值。

当然, 系统中的有些线程必须负责执行 DllMain 函数中的代码。当一个新线程创建时, 系统将分配进程的地址空间, 然后将 .exe 文件映像和所有需要的 DLL 文件映像映射到进程的地址空间中。然后它创建进程的主线程, 并使用该线程调用每个 DLL 的带有 DLL_PROCESS_ATTACH 值的 DllMain 函数。当已经映射的所有 DLL 都对通知信息作出响应后, 系统将使进程的主线程开始执行可执行模块的 C/C++ 运行期启动代码, 然后执行可执行模块的进入点函数 (main、wmain、WinMain 或 wWinMain)。如果 DLL 的任何一个 DllMain 函数返回 FALSE, 指明初始化没有取得成功, 系统便终止整个进程的运行, 从它的地址空间中删除所有文件映像, 给用户显示一个消息框, 说明进程无法启动运行。Windows 2000 的这个消息框如图 20-2 所示, 再下面是 Windows 98 的消息框 (见图 20-3)。

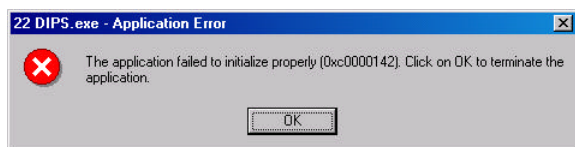


图20-2 Windows 2000下显示的消息框

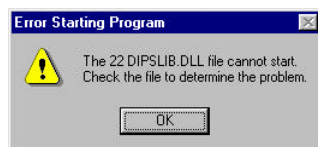


图20-3 Windows 98下显示的消息框

下面让我们来看一看 DLL 被显式加载时的情况。当进程中的一个线程调用 LoadLibrary(Ex) 时, 系统会找出特定的 DLL, 并将它映射到进程的地址空间中。然后, 系统使用调用 LoadLibrary(Ex) 的线程, 调用 DLL 的带有 DLL_PROCESS_ATTACH 值的 DllMain 函数。当 DLL 的 DllMain 函数处理了通知消息后, 系统便允许调用的 LoadLibrary(Ex) 函数返回, 同时该线程像平常一样继续进行处理。如果 DllMain 函数返回 FALSE, 指明初始化没有取得成功, 那么系统就自动从进程的地址空间中卸载 DLL 的文件映像, 而对 LoadLibrary(Ex) 的调用则返回 NULL。

20.2.2 DLL_PROCESS_DETACH 通知

DLL 从进程的地址空间中被卸载时, 系统将调用 DLL 的 DllMain 函数, 给它传递 fdwReason 的值 DLL_PROCESS_DETACH。当 DLL 处理这个值时, 它应该执行任何与进程相关的清除操作。例如, DLL 可以调用 HeapDestroy 函数来撤消它在 DLL_PROCESS_DETACH 通知期间创建的堆栈。注意, 如果 DllMain 函数接收到 DLL_PROCESS_DETACH 通知时返回 FALSE, 那么 DllMain 就不是用 DLL_PROCESS_DETACH 通知调用的。如果因为进程终止运行而使 DLL 被卸载, 那么调用 ExitProcess 函数的线程将负责执行 DllMain 函数的代码。在正常情况下, 这是应用程序的主线程。当你的进入点函数返回到 C/C++ 运行期库的启动代码时, 该启动代码将显式调用 ExitProcess 函数, 终止进程的运行。

如果因为进程中的线程调用 FreeLibrary 或 FreeLibraryAndExitThread 函数而将 DLL 卸载, 那么调用函数的线程将负责执行 DllMain 函数的代码。如果使用 FreeLibrary, 那么要等到 DllMain 函数完成对 DLL_PROCESS_DETACH 通知的执行后, 该线程才从对 FreeLibrary 函数的调用中返回。

注意, DLL 能够阻止进程终止运行。例如, 当 DllMain 接收到 DLL_PROCESS_DETACH 通知时, 它就会进入一个无限循环。只有当每个 DLL 都已完成对 DLL_PROCESS_DETACH 通知

的处理时，操作系统才会终止该进程的运行。

注意 如果因为系统中的某个线程调用了 TerminateProcess 而使进程终止运行，那么系统将不调用带有 DLL_PROCESS_DETACH 值的 DLL 的 DllMain 函数。这意味着映射到进程的地址空间中的任何 DLL 都没有机会在进程终止运行之前执行任何清除操作。这可能导致数据的丢失。只有在迫不得已的情况下，才能使用 TerminateProcess 函数。

图20-4显示了线程调用 LoadLibrary 时执行的操作步骤。图20-5显示了线程调用 FreeLibrary 函数时执行的操作步骤。

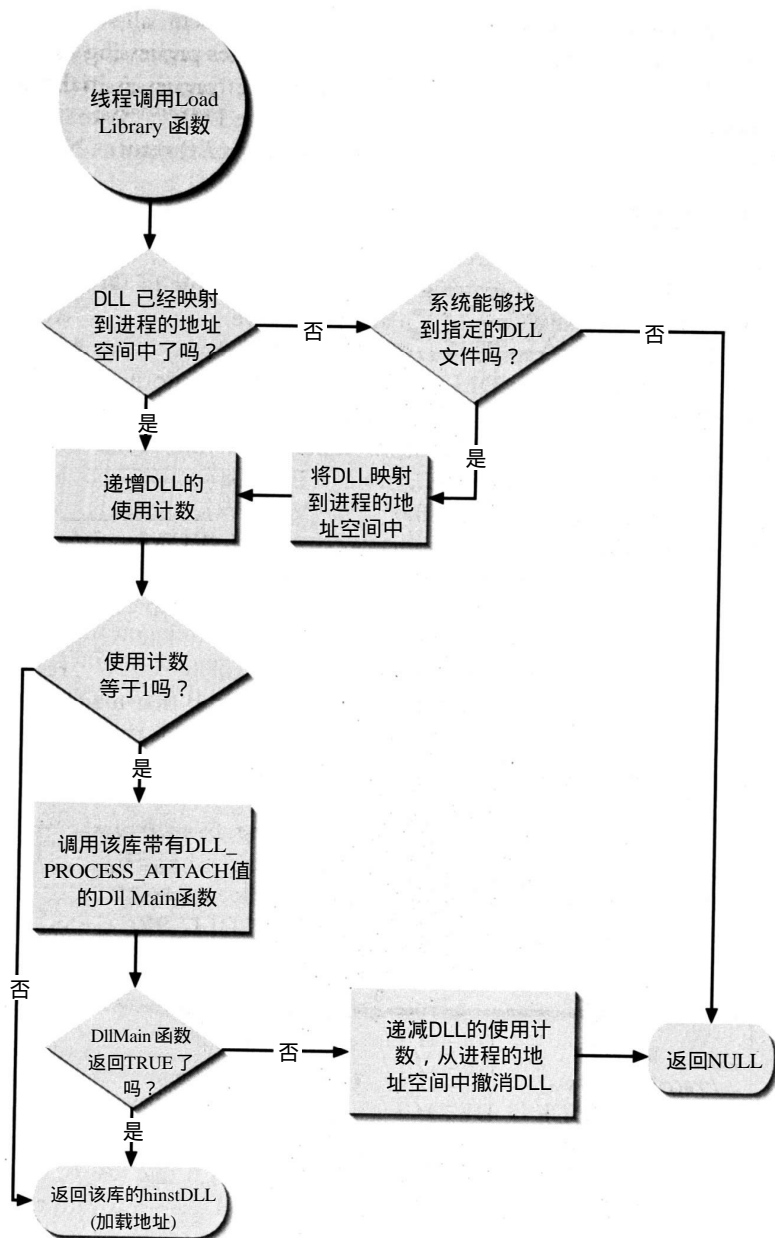


图20-4 线程调用LoadLibrary时系统执行的操作步骤

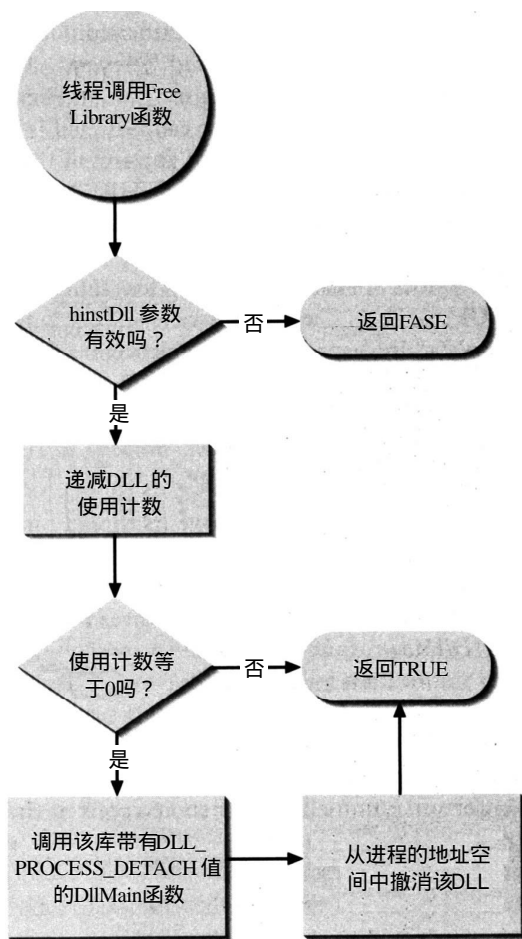


图20-5 线程调用FreeLibrary时系统执行的操作步骤

20.2.3 DLL_THREAD_ATTACH通知

当在一个进程中创建线程时，系统要查看当前映射到该进程的地址空间中的所有 DLL 文件映像，并调用每个文件映像的带有 DLL_THREAD_ATTACH 值的 DllMain 函数。这可以告诉所有的 DLL 执行每个线程的初始化操作。新创建的线程负责执行 DLL 的所有 DllMain 函数中的代码。只有当所有的 DLL 都有机会处理该通知时，系统才允许新线程开始执行它的线程函数。

当一个新 DLL 被映射到进程的地址空间中时，如果该进程内已经有若干个线程正在运行，那么系统将不为现有的线程调用带有 DLL_THREAD_ATTACH 值的 DLL 的 DllMain 函数。只有当新线程创建时 DLL 被映射到进程的地址空间中，它才调用带有 DLL_THREAD_ATTACH 值的 DLL 的 DllMain 函数。

另外要注意，系统并不为进程的主线程调用带有 DLL_THREAD_ATTACH 值的任何 DllMain 函数。进程初次启动时映射到进程的地址空间中的任何 DLL 均接收 DLL_PROCESS_ATTACH 通知，而不是 DLL_THREAD_ATTACH 通知。

20.2.4 DLL_THREAD_DETACH通知

让线程终止运行的首选方法是使它的线程函数返回。这使得系统可以调用 ExitThread 来撤

消该线程。ExitThread函数告诉系统，该线程想要终止运行，但是系统并不立即将它撤消。相反，它要取出这个即将被撤消的线程，并让它调用已经映射的DLL的所有带有DLL_THREAD_DETACH值的DllMain函数。这个通知告诉所有的DLL执行每个线程的清除操作。例如，DLL版本的C/C++运行期库能够释放它用于管理多线程应用程序的数据块。

注意，DLL能够防止线程终止运行。例如，当DllMain函数接收到DLL_THREAD_DETACH通知时，它就能够进入一个无限循环。只有当每个DLL已经完成对DLL_THREAD_DETACH通知的处理时，操作系统才会终止线程的运行。

注意 如果因为系统中的线程调用TerminateThread函数而使该线程终止运行，那么系统将不调用带有DLL_THREAD_DETACH值的DLL的所有DllMain函数。这意味着映射到进程的地址空间中的任何一个DLL都没有机会在线程终止运行之前执行任何清除操作。这可能导致数据的丢失。与TerminateProcess一样，只有在迫不得已的时候，才可以使用TerminateThread函数。

如果当DLL被撤消时仍然有线程在运行，那么就不为任何线程调用带有DLL_THREAD_DETACH值的DllMain。可以在进行DLL_THREAD_DETACH的处理时查看这个情况，这样就能够执行必要的清除操作。

上述规则可能导致发生下面这种情况。当进程中的一个线程调用LoadLibrary来加载DLL时，系统就会调用带有DLL_PROCESS_ATTACH值的DLL的DllMain函数（注意，没有为该线程发送DLL_THREAD_ATTACH通知）。接着，负责加载DLL的线程退出，从而导致DLL的DllMain函数被再次调用，这次调用时带有DLL_THREAD_DETACH值。注意，DLL得到通知说，该线程将被撤消，尽管它从未收到DLL_THREAD_ATTACH的这个通知，这个通知告诉该库说线程已经附加。由于这个原因，当执行任何特定的线程清除操作时，必须非常小心。不过大多数程序在编写时就规定调用LoadLibrary的线程与调用FreeLibrary的线程是同一个线程。

20.2.5 顺序调用DllMain

系统是顺序调用DLL的DllMain函数的。为了理解这样做的意义，可以考虑下面这样一个环境。假设一个进程有两个线程，线程A和线程B。该进程还有一个DLL，称为SomeDLL.dll，它被映射到了它的地址空间中。两个线程都准备调用CreateThread函数，以便再创建两个线程，即线程C和线程D。

当线程A调用CreateThread来创建线程C时，系统调用带有DLL_THREAD_ATTACH值的SomeDLL.dll的DllMain函数。当线程C执行DllMain函数中的代码时，线程B调用CreateThread函数来创建线程D。这时系统必须再次调用带有DLL_THREAD_ATTACH值的DllMain函数，这次是让线程D执行代码。但是，系统是顺序调用DllMain函数的，因此系统会暂停线程D的运行，直到线程C完成对DllMain函数中的代码的处理并且返回为止。

当线程C完成DllMain的处理后，它就开始执行它的线程函数。这时系统唤醒线程D，让它处理DllMain中的代码。当它返回时，线程D开始处理它的线程函数。

通常情况下，根本不会考虑到DllMain的这个顺序操作特性。我曾经遇到过一个人，他的代码中有一个DllMain顺序操作带来的错误。他创建的代码类似下面的样子：

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {  
  
    HANDLE hThread;  
    DWORD dwThreadId;
```

```

switch (fdwReason) {
case DLL_PROCESS_ATTACH:
    // The DLL is being mapped into the process's address space.

    // Create a thread to do some stuff.
    hThread = CreateThread(NULL, 0, SomeFunction, NULL,
        0, &dwThreadId);

    // Suspend our thread until the new thread terminates.
    WaitForSingleObject(hThread, INFINITE);

    // We no longer need access to the new thread.
    CloseHandle(hThread);
    break;

case DLL_THREAD_ATTACH:
    // A thread is being created.
    break;

case DLL_THREAD_DETACH:
    // A thread is exiting cleanly.
    break;

case DLL_PROCESS_DETACH:
    // The DLL is being unmapped from the process's address space.
    break;
}
return(TRUE);
}

```

我们花了好几个小时才发现这个代码中存在的问题。你能够看出这个问题吗？当 DllMain 收到 DLL_PROCESS_ATTACH 通知时，一个新线程就创建了。系统必须用 DLL_THREAD_ATTACH 的值再次调用 DllMain 函数。但是，新线程被暂停运行，因为导致 DLL_PROCESS_ATTACH 被发送给 DllMain 函数的线程尚未完成处理操作。问题是调用 WaitForSingleObject 函数而产生的。这个函数使当前正在运行的线程暂停运行，直到新线程终止运行。但是新线程从未得到机会运行，更不要说终止运行，因为它处于暂停状态，等待当前线程退出 DllMain 函数。这里我们得到的是个死锁条件。两个线程将永远处于暂停状态。

当我刚刚开始考虑如何解决这个问题的时候，我发现了 DisableThreadLibraryCalls 函数：

```
BOOL DisableThreadLibraryCalls(HINSTANCE hinstDll);
```

DisableThreadLibraryCalls 告诉系统说，你不想将 DLL_THREAD_ATTACH 和 DLL_THREAD_DETACH 通知发送给特定的 DLL 的 DllMain 函数。我认为这样做是有道理的，如果我们告诉系统不要将 DLL 通知发送给 DLL，那么就不会发送死锁条件。但是当我测试解决方案时，我很快发现它解决不了问题。请看下面的代码：

```

BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {

    HANDLE hThread;
    DWORD dwThreadId;

    switch (fdwReason) {
case DLL_PROCESS_ATTACH:
    // The DLL is being mapped into the process's address space.

```

```
// Prevent the system from callingDllMain
// when threads are created or destroyed.
DisableThreadLibraryCalls(hinstDll);

// Create a thread to do some stuff.
hThread = CreateThread(NULL, 0, SomeFunction, NULL,
    0, &dwThreadId);

// Suspend our thread until the new thread terminates.
WaitForSingleObject(hThread, INFINITE);

// We no longer need access to the new thread.
CloseHandle(hThread);
break;

case DLL_THREAD_ATTACH:
    // A thread is being created.
    break;

case DLL_THREAD_DETACH:
    // A thread is exiting cleanly.
    break;

case DLL_PROCESS_DETACH:
    // The DLL is being unmapped from the process's address space.
    break;
}
return(TRUE);
}
```

通过进一步的研究,我终于发现了问题。当进程被创建时,系统也创建一个互斥对象。每个进程都有它自己的互斥对象,也就是说多个进程并不共享互斥对象。当线程调用映射到进程的地址空间中的DLL的DllMain函数时,这个互斥对象负责对进程的所有线程实施同步。

当CreateThread函数被调用时,系统首先创建线程的内核对象和线程的堆栈。然后它在内部调用WaitForSingleObject函数,传递进程的互斥对象的句柄。一旦新线程拥有该互斥对象,系统就让新线程用DLL_THREAD_ATTACH的值调用每个DLL的DllMain函数。只有在这个时候,系统才调用ReleaseMutex,释放对进程的互斥对象的所有权。由于系统采用这种方式来运行,因此添加对DisableThreadLibraryCalls的调用,并不会防止线程被暂停运行。防止线程被暂停运行的唯一办法是重新设计这部分源代码,使得WaitForSingleObject不会在任何DLL的DllMain函数中被调用。

20.2.6 DllMain与C/C++运行期库

在上面介绍的DllMain函数中,我假设你使用Microsoft的Visual C++编译器来创建你的DLL。当编写一个DLL时,你需要得到C/C++运行期库的某些初始帮助。例如,如果你创建的DLL包含一个全局变量,而这个全局变量是个C++类的实例。在你顺利地DllMain函数中使用这个全局变量之前,该变量必须调用它的构造函数。这是由C/C++运行期库的DLL启动代码来完成的。

当你链接你的DLL时,链接程序将DLL的进入点函数嵌入产生的DLL文件映像。可以使用链接程序的/ENTRY开关来设定该函数的地址。按照默认设置,当使用Microsoft的链接程序并

且设定/DLL开关时，链接程序假设进入点函数称为 `_DllMainCRTStartup`。该函数包含在C/C++运行期的库文件中，并且在你链接DLL时它被静态链接到你的DLL文件的映像中（即使你使用DLL版本的C/C++运行期库，该函数也是静态链接的）。

当你的DLL文件映像被映射到进程的地址空间中时，系统实际上是调用 `_DllMainCRTStartup`函数，而不是调用 `DllMain`函数。`_DllMainCRTStartup`函数负责对C/C++运行期库进行初始化，并且确保在 `_DllMainCRTStartup`收到 `DLL_PROCESS_ATTACH`通知时创建任何全局或静态C++对象。当执行任何C/C++运行期初始化时，`_DllMainCRTStartup`函数将调用你的 `DllMain`函数。

当DLL收到 `DLL_PROCESS_DETACH`通知时，系统再次调用 `_DllMainCRTStartup`函数。这次该函数调用你的 `DllMain`函数，当 `DllMain`返回时，`_DllMainCRTStartup`就为DLL中的任何全局或静态C++对象调用析构函数。当 `_DllMainCRTStartup`收到 `DLL_THREAD_ATTACH`通知时，`_DllMainCRTStartup`函数并不执行任何特殊的处理操作。但是对于 `DLL_THREAD_DETACH`来说，C/C++运行期将释放线程的 `tiddata`内存块（如果存在这样的内存块的话）。但是，通常情况下，这个 `tiddata`内存块是不应该存在的，因为编写正确的线程函数将返回到内部调用 `_endthreadex`的C/C++运行期的 `_threadstartex`函数（第6章已经介绍），它负责在线程试图调用 `ExitThread`之前释放内存块。

然而，让我们看一看这样一种情况，即用Pascal编写的应用程序调用DLL中用C/C++编写的函数。在这种情况下，Pascal应用程序创建了一个线程，并且不使用 `_beginthreadex`。因此线程对C/C++运行期库的情况一无所知。这时线程调用DLL中的一个函数，该函数又调用一个C运行期函数。当你再次调用该函数时，C运行期函数为该线程创建一个 `tiddata`内存块，并且在创建过程中将它与线程关联起来。这意味着Pascal应用程序能够成功地调用C运行期函数的线程。当用Pascal编写的线程函数返回时，`ExitThread`被调用。C/C++运行期库的DLL收到 `DLL_THREAD_DETACH`通知，并释放 `tiddata`内存块，这样就不会出现任何内存泄漏。这确实是个非常出色的思路。

前面讲过，不必在DLL源代码中实现 `DllMain`函数。如果你并不拥有自己的 `DllMain`函数，可以使用C/C++运行期库的 `DllMain`函数的实现代码，它类似下面的形式（如果静态链接到C/C++运行期库的话）：

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {  
  
    if (fdwReason == DLL_PROCESS_ATTACH)  
        DisableThreadLibraryCalls(hinstDll);  
    return(TRUE);  
}
```

当链接程序链接DLL时，如果链接程序无法找到DLL的.obj文件中的 `DllMain`函数，那么它就链接C/C++运行期库的 `DllMain`函数的实现代码。如果你没有提供自己的 `DllMain`函数，C/C++运行期库就正确地假设你不在乎 `DLL_THREAD_ATTACH`和 `DLL_THREAD_DETACH`通知。为了提高创建和撤消线程的性能，则调用 `DisableThreadLibraryCalls`函数。

20.3 延迟加载DLL

Microsoft Visual C++ 6.0提供了一个出色的新特性，它能够使DLL的操作变得更加容易。这个特性称为延迟加载DLL。延迟加载的DLL是个隐含链接的DLL，它实际上要等到你的代码试图引用DLL中包含的一个符号时才进行加载。延迟加载的DLL在下列情况下是

非常有用的：

- 如果你的应用程序使用若干个 DLL，那么它的初始化时间就比较长，因为加载程序要将所有需要的 DLL 映射到进程的地址空间中。解决这个问题的方法之一是在进程运行的时候分开加载各个 DLL。延迟加载的 DLL 能够更容易地完成这样的加载。
- 如果调用代码中的一个新函数，然后试图在老版本的系统上运行你的应用程序，而该系统中没有该函数，那么加载程序就会报告一个错误，并且不允许该应用程序运行。你需要一种方法让你的应用程序运行，然后，如果（在运行时）发现该应用程序在老的系统上运行，那么你将不调用遗漏的函数。例如，一个应用程序在 Windows 2000 上运行时想要使用 PSAPI 函数，而在 Windows 98 上运行想要使用 ToolHelp 函数（比如 Process32Next）。当该应用程序初始化时，它调用 GetVersionEx 函数来确定主操作系统，并正确地调用相应的其他函数。如果试图在 Windows 98 上运行该应用程序，就会导致加载程序显示一条错误消息，因为 Windows 98 上并不存在 PSAPI.dll 模块。同样，延迟加载的 DLL 能够使你非常容易地解决这个问题。

我花费了相当多的时间来检验 Visual C++ 6.0 中的延迟加载 DLL 特性，必须承认，Microsoft 在实现这个特性方面做了非常出色的工作。它提供了许多特性，并且在 Windows 98 和 Windows 2000 上运行得都很好。

下面让我们从比较容易的操作开始介绍，也就是使延迟加载 DLL 能够运行。首先，你象平常那样创建一个 DLL。也要象平常那样创建一个可执行模块，但是必须修改两个链接程序开关，并且重新链接可执行模块。下面是需要添加的两个链接程序开关：

```
/Lib:DelayImp.lib
```

```
/DelayLoad:MyDll.dll
```

Lib 开关告诉链接程序将一个特殊的函数 `--delayLoadHelper` 嵌入你的可执行模块。第二个开关将下列事情告诉链接程序：

- 从可执行模块的输入节中删除 MyDll.dll，这样，当进程被初始化时，操作系统的加载程序就不会显式加载 DLL。
- 将新的 Delay Import（延迟输入）节（称为 .idata）嵌入可执行模块，以指明哪些函数正在从 MyDll.dll 输入。
- 通过转移到对 `--delayLoadHelper` 函数的调用，转换到对延迟加载函数的调用。

当应用程序运行时，对延迟加载函数的调用实际上是对 `--delayLoadHelper` 函数的调用。该函数引用特殊的 Delay Import 节，并且知道调用 LoadLibrary 之后再调用 GetProcAddress。一旦获得延迟加载函数的地址，`--delayLoadHelper` 就要安排好对该函数的调用，这样，将来的调用就会直接转向对延迟加载函数的调用。注意，当第一次调用同一个 DLL 中的其他函数时，必须对它们做好安排。另外，可以多次设定 `/delayLoad` 链接程序的开关，为想要延迟加载的每个 DLL 设定一次开关。

好了，整个操作过程就这么简单。但是还应该考虑另外两个问题。通常情况下，当操作系统的加载程序加载可执行模块时，它将设法加载必要的 DLL。如果一个 DLL 无法加载，那么加载程序就会显示一条错误消息。如果是延迟加载的 DLL，那么在进行初始化时将不检查是否存在 DLL。如果调用延迟加载函数时无法找到该 DLL，`--delayLoadHelper` 函数就会引发一个软件异常条件。可以使用结构化异常处理（SEH）方法来跟踪该异常条件。如果不跟踪该异常条件，那么你的进程就会终止运行（SEH 将在第 23、24 和 25 章中介绍）。

当 `--delayLoadHelper` 确实找到你的 DLL，但是要调用的函数不在该 DLL 中时，将会出现另

一个问题。比如，如果加载程序找到一个老的 DLL 版本，就会发生这种情况。在这种情况下，--delayLoadHelper 也会引发一个软件异常条件，对这个软件异常条件的处理方法与上面相同。下一节介绍的示例应用程序显示了如何正确地编写 SEH 代码以便处理这些错误。

你会发现代码中有许多其他元素，这些元素与 SEH 和错误处理毫无关系。但是这些元素与你使用延迟加载的 DLL 时可以使用的辅助特性有关。下面将要介绍这些特性。如果你不使用更多的高级特性，可以删除这些额外的代码。

如你所见，Visual C++ 开发小组定义了两个软件异常条件代码，即 `VcppException (ERROR_SEVERITY_ERROR、ERROR_MOD_NOT_FOUND)` 和 `VcppException (ERROR_SEVERITY_ERROR、ERROR_PROC_NOT_FOUND)`。这些代码分别用于指明 DLL 模块没有找到和函数没有找到。我的异常过滤函数 `DelayLoadDllExceptionFilter` 用于查找这两个异常代码。如果两个代码都没有找到，过滤函数将返回 `EXCEPTION_CONTINUE_SEARCH`，这与任何出色的过滤函数返回的值是一样的（对于你不知道如何处理的异常代码，请不要随意删除）。但是如果这两个代码中的一个已经找到，那么 --delayLoadHelper 函数将提供一个指向包含某些辅助信息的 `DelayLoadInfo` 结构的指针。在 Visual C++ 的 `DelayImp.h` 文件中，`DelayLoadInfo` 结构定义为下面的形式：

```
typedef struct DelayLoadInfo {
    DWORD          cb;           // Size of structure
    PCImgDelayDescr pidd;        // Raw data (everything is there)
    FARPROC *       ppfn;        // Points to address of function to load
    LPCSTR          szDll;       // Name of dll
    DelayLoadProc   dlp;         // Name or ordinal of procedure
    HMODULE          hmodCur;    // hInstance of loaded library
    FARPROC          pfnCur;     // Actual function that will be called
    DWORD           dwLastError; // Error received
} DelayLoadInfo, * PDelayLoadInfo;
```

这个数据结构是由 --delayLoadHelper 函数来分配和初始化的。在该函数按步骤动态加载 DLL 并且获得被调用函数的地址的过程中，它将填写该结构的各个成员。在 SEH 结构的内部，成员 `szDll` 指向你要加载的 DLL 的名字，想要查看的函数则在成员 `dlp` 中。由于可以按序号或名字来查看各个函数，因此 `dlp` 成员类似下面的样子：

```
typedef struct DelayLoadProc {
    BOOL fImportByName;
    union {
        LPCSTR szProcName;
        DWORD  dwOrdinal;
    };
} DelayLoadProc;
```

如果 DLL 已经加载成功，但是它不包含必要的函数，也可以查看成员 `hmodCur`，以了解 DLL 被加载到的内存地址。也可以查看成员 `dwLastError`，以了解是什么错误导致了异常条件的引发。不过对于异常过滤函数来说，这是不必要的，因为异常代码能够告诉你究竟发生了什么问题。成员 `pfnCur` 包含了需要的函数的地址。在过滤函数中它总是置为 `NULL`，因为 --delayLoadHelper 无法找到该函数的地址。

在其余的成员中，`cb` 用于确定版本，`pidd` 指向嵌入模块中包含延迟加载的 DLL 和函数的节，`ppfn` 是函数找到时，函数的地址应该放入的地址。最后两个成员供 --delayLoadHelper 函数内部使用。它们有着超高级的用途，现在还没有必要观察或者了解这两个成员。

到现在为止，已经讲述了如何使用延迟加载的 DLL 和正确解决错误条件的基本方法。但是 Microsoft 的延迟加载 DLL 的实现代码超出了迄今为止我已讲述的内容范围。比如，你的应用程序能够卸载延迟加载的 DLL。假如你的应用程序需要一个特殊的 DLL 来打印一个文档，那么这个 DLL 就非常适合作为一个延迟加载的 DLL，因为大部分时间它是不用的。不过，如果用户选择了 Print 命令，你就可以调用该 DLL 中的一个函数，然后它能够自动进行 DLL 的加载。这确实很好，但是，当文档打印后，用户可能不会立即打印另一个文档，因此可以卸载这个 DLL，释放系统的资源。如果用户决定打印另一个文档，那么 DLL 就可以根据用户的要求再次加载。

若要卸载延迟加载的 DLL，必须执行两项操作。首先，当创建可执行文件时，必须设定另一个链接程序开关（/delay:unload）。其次，必须修改源代码，并且在你想要卸载 DLL 时调用 `--FUnloadDelayLoadedDLL` 函数：

```
BOOL __FUnloadDelayLoadedDLL(PCSTR szDll);
```

/Delay:unload 链接程序开关告诉链接程序将另一个节放入文件中。该节包含了你清除已经调用的函数时需要的信息，这样它们就可以再次调用 `--delayLoadHelper` 函数。当调用 `--FUnloadDelayLoadedDll` 时，你将想要卸载的延迟加载的 DLL 的名字传递给它。该函数进入文件中的未卸载节，并清除 DLL 的所有函数地址，然后 `--FUnloadDelayLoadedDll` 调用 `FreeLibrary`，以便卸载该 DLL。

下面要指出一些重要的问题。首先，千万不要自己调用 `FreeLibrary` 来卸载 DLL，否则函数的地址将不会被清除，这样，当下次试图调用 DLL 中的函数时，就会导致访问违规。第二，当调用 `--FUnloadDelayLoadedDll` 时，传递的 DLL 名字不应该包含路径，名字中的字母必须与你将 DLL 名字传递给 /DelayLoad 链接程序开关时使用的字母大小写相同，否则，`--FUnloadDelayLoadedDll` 的调用将会失败。第三，如果永远不打算卸载延迟加载的 DLL，那么请不要设定 /Delay:unload 链接程序开关，并且你的可执行文件的长度应该比较小。最后，如果你不从用 /Delay:unload 开关创建的模块中调用 `--FUnloadDelayLoadedDll`，那么什么也不会发生，`--FUnloadDelayLoadedDll` 什么操作也不执行，它将返回 FALSE。

延迟加载的 DLL 具备的另一个特性是，按照默认设置，调用的函数可以与一些内存地址相链接，在这些内存地址上，系统认为函数将位于一个进程的地址中（本章后面将介绍链接的问题）。由于创建可链接的延迟加载的 DLL 节会使你的可执行文件变得比较大，因此链接程序也支持一个 /Delay:nobind 开关。因为人们通常都喜欢进行链接，因此大多数应用程序不应该使用这个链接开关。

延迟加载的 DLL 的最后一个特性是供高级用户使用的，它真正显示了 Microsoft 的注意力之所在。当 `--delayLoadHelper` 函数执行时，它可以调用你提供的挂钩函数。这些函数将接收 `--delayLoadHelper` 函数的进度通知和错误通知。此外，这些函数可以重载 DLL 如何加载的方法以及如何获取函数的虚拟内存地址的方法。

若要获得通知或重载的行为特性，必须对你的源代码做两件事情。首先必须编写类似清单 20-1 所示的 `DliHook` 函数那样的挂钩函数。`DliHook` 框架函数并不影响 `--delayLoadHelper` 函数的运行。若要改变它的行为特性，可启动 `DliHook` 函数，然后根据需要对它进行修改。接着将函数的地址告诉 `--delayLoadHelper`。

在 `DelayImp.lib` 静态链接库中，定义了两个全局变量，即 `--pfnDliNotifyHook` 和 `--pfnDliFailureHook`。这两个变量均属于 `pfnDliHook` 类型：

```
typedef FARPROC (WINAPI *PfnDliHook)(  
    unsigned dliNotify,  
    PDelayLoadInfo pdli);
```

如你所见，这是个数据类型的函数，与我的 DliHook 函数的原型相匹配。在 DelayImp.lib 文件中，两个变量被初始化为 NULL，它告诉 --delayLoadHelper 不要调用任何挂钩函数。若要使你的函数被调用，必须将这两个函数中的一个设置为挂钩函数的地址。在我的代码中，我只是将下面两行代码添加到全局作用域：

```
PfnDliHook __pfnDliNotifyHook = DliHook;  
PfnDliHook __pfnDliFailureHook = DliHook;
```

如你所见，--delayLoadHelper 实际上是与两个回调函数一道运行的。它调用一个函数以便报告通知，调用另一个函数来报告失败情况。由于这两个函数的原型是相同的，而第一个参数 dliNotify 告诉为什么调用这个函数，因此我总是通过创建单个函数并将两个变量设置为指向我的一个函数，使我的工作变得简单一些。

Visual C++ 6.0 的延迟加载 DLL 的新特性非常出色，许多编程人员几年前就希望使用这个特性。可以想像许多应用程序（尤其是 Microsoft 的应用程序）都将充分利用这个特性。

DelayLoadApp 示例应用程序

清单 20-1 中列出的 DelayLoadApp 应用程序（“20 DelayLoadApp.exe”）显示了在充分利用延迟加载 DLL 时应该做的所有工作。为了演示的需要，必须使用一个简单的 DLL，它的代码位于 20-DelayLoadLib 目录中。

由于该应用程序加载了“20 DelayLoadLib”模块，因此当运行该应用程序时，加载程序不必将该模块映射到进程的地址空间中。在该应用程序中，我定期调用 IsModuleLoaded 函数。该函数只是用来显示一个消息框，通知是否有一个模块加载到了进程的地址空间中。当该应用程序初次启动运行时，“20 DelayLoadLib”模块尚未加载，因此出现图 20-6 所示的消息框。

然后该应用程序调用从 DLL 输入的一个函数，这使得 __delayLoadHelper 函数能够自动加载该 DLL。当该函数返回时，便出现图 20-7 所示的消息框。



图20-6 DelayLoadApp显示“20 DelayLoadLib”
模块尚未加载



图20-7 DelayLoadApp显示“20 DelayLoadLib”
模块已经加载

当这个消息框关闭时，DLL 中的另一个函数被调用。由于该函数是在同一个 DLL 中，因此该 DLL 不必再次加载到该地址空间中去，不过该新函数的地址被转换并调用。

这时，FUnloadDelayLoadedDLL 函数被调用，它负责卸载“20 DelayLoadedLib”。同样，如果调用 IsModuleLoaded 函数，就可以显示图 20-6 所示的消息框。最后，一个输入函数再次被调用，它由于重新加载“20 DelayLoadLib”模块，从而使最后一次对 IsModuleLoaded 的调用能够显示图 20-7 所示的消息框。

如果一切顺利，程序将按描述的那样工作。然而，如果在运行程序之前删除“20 DelayLoadLib”模块或者如果模块不包含导入的一个函数，就会引发异常。示例代码显示了如何从这种情况中恢复过来。

最后，该应用程序显示了如何正确地设置延迟加载的挂钩函数的方法。我的 DliHook 框架函数并没有执行什么新奇的功能。然而它能够捕获各个通知消息，显示收到这些通知时能够执行的操作。

清单20-1 DelayLoadApp示例应用程序



DelayLoadApp.cpp

```

/*****
Module: DelayLoadApp.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

#include "..\CmnHdr.h"      /* See Appendix A. */
#include <Windowsx.h>
#include <tchar.h>

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

#include <Delayimp.h>      // For error handling & advanced features
#include "..\20-DelayLoadLib\DelayLoadLib.h"    // My DLL function prototypes

////////////////////////////////////////////////////////////////
// Statically link __delayLoadHelper/__FUnloadDelayLoadedDLL
#pragma comment(lib, "Delayimp.lib")

// Tell the linker that my DLL should be delay loaded
// Note the 2 (\") because the filename has a space in it
#pragma comment(linker, "/DelayLoad:\"20 DelayLoadLib.dll\")

// Tell the linker that I want to be able to unload my DLL
#pragma comment(linker, "/Delay:unload")

// Tell the linker to make the delay load DLL unbindable
// You usually want this, so I commented out this line
// #pragma comment(linker, "/Delay:nobind")

// The name of the Delay-Load module (only used by this sample app)
TCHAR g_szDelayLoadModuleName[] = TEXT("20 DelayLoadLib");

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

// Forward function prototype
LONG WINAPI DelayLoadDllExceptionFilter(PEXCEPTION_POINTERS pep);

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

void IsModuleLoaded(PCTSTR pszModuleName) {

```



```

    HMODULE hmod = GetModuleHandle(pszModuleName);
    char sz[100];
#ifdef UNICODE
    wprintfA(sz, "Module \"%S\" is %Sloaded.",
        pszModuleName, (hmod == NULL) ? L"not " : L "");
#else
    wprintfA(sz, "Module \"%s\" is %sloaded.",
        pszModuleName, (hmod == NULL) ? "not " : "");
#endif
    chMB(sz);
}

/////////////////////////////////////////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    // Wrap all calls to delay-load DLL functions inside SEH
    __try {
        int x = 0;

        // If you're in the debugger, try the new Debug.Modules menu item to
        // see that the DLL is not loaded prior to executing the line below
        IsModuleLoaded(g_szDelayLoadModuleName);

        x = fnLib(); // Attempt to call delay-load function

        // Use Debug.Modules to see that the DLL is now loaded
        IsModuleLoaded(g_szDelayLoadModuleName);

        x = fnLib2(); // Attempt to call delay-load function

        // Unload the delay-loaded DLL
        // NOTE: Name must exactly match /DelayLoad:(DllName)
        __FUnloadDelayLoadedDLL("20 DelayLoadLib.dll");

        // Use Debug.Modules to see that the DLL is now unloaded
        IsModuleLoaded(g_szDelayLoadModuleName);

        x = fnLib(); // Attempt to call delay-load function

        // Use Debug.Modules to see that the DLL is loaded again
        IsModuleLoaded(g_szDelayLoadModuleName);
    }
    __except (DelayLoadDllExceptionFilter(GetExceptionInformation())) {
        // Nothing to do in here, thread continues to run normally
    }

    // More code can go here...

    return(0);
}

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

LONG WINAPI DelayLoadDllExceptionFilter(PEXCEPTION_POINTERS pep) {

```

```

// Assume we recognize this exception
LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;

// If this is a Delay-load problem, ExceptionInformation[0] points
// to a DelayLoadInfo structure that has detailed error info
PDelayLoadInfo pdli =
    PDelayLoadInfo(pep->ExceptionRecord->ExceptionInformation[0]);

// Create a buffer where we construct error messages
char sz[500] = { 0 };

switch (pep->ExceptionRecord->ExceptionCode) {
case VcppException(ERROR_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND):
    // The DLL module was not found at run time
    wsprintfA(sz, "Dll not found: %s", pdli->szDll);
    break;

case VcppException(ERROR_SEVERITY_ERROR, ERROR_PROC_NOT_FOUND):
    // The DLL module was found, but it doesn't contain the function
    if (pdli->dlp.fImportByName) {
        wsprintfA(sz, "Function %s was not found in %s",
            pdli->dlp.szProcName, pdli->szDll);
    } else {
        wsprintfA(sz, "Function ordinal %d was not found in %s",
            pdli->dlp.dwOrdinal, pdli->szDll);
    }
    break;

default:
    // We don't recognize this exception
    lDisposition = EXCEPTION_CONTINUE_SEARCH;
    break;
}

if (lDisposition == EXCEPTION_EXECUTE_HANDLER) {
    // We recognized this error and constructed a message, show it
    chMB(sz);
}

return(lDisposition);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Skeleton DliHook function that does nothing interesting
FARPROC WINAPI DliHook(unsigned dliNotify, PDelayLoadInfo pdli) {

    FARPROC fp = NULL;    // Default return value

    // NOTE: The members of the DelayLoadInfo structure pointed
    // to by pdli shows the results of progress made so far.

    switch (dliNotify) {
    case dliStartProcessing:
        // Called when __delayLoadHelper attempts to find a DLL/function

```

```

// Return 0 to have normal behavior or nonzero to override
// everything (you will still get dllNoteEndProcessing)
break;

case dllNotePreLoadLibrary:
    // Called just before LoadLibrary
    // Return NULL to have __delayLoadHelper call LoadLibrary
    // or you can call LoadLibrary yourself and return the HMODULE
    fp = (FARPROC) (HMODULE) NULL;
    break;

case dllFailLoadLib:
    // Called if LoadLibrary fails
    // Again, you can call LoadLibrary yourself here and return an HMODULE
    // If you return NULL, __delayLoadHelper raises the
    // ERROR_MOD_NOT_FOUND exception
    fp = (FARPROC) (HMODULE) NULL;
    break;

case dllNotePreGetProcAddress:
    // Called just before GetProcAddress
    // Return NULL to have __delayLoadHelper call GetProcAddress
    // or you can call GetProcAddress yourself and return the address
    fp = (FARPROC) NULL;
    break;

case dllFailGetProcAddress:
    // Called if GetProcAddress fails
    // You can call GetProcAddress yourself here and return an address
    // If you return NULL, __delayLoadHelper raises the
    // ERROR_PROC_NOT_FOUND exception
    fp = (FARPROC) NULL;
    break;

case dllNoteEndProcessing:
    // A simple notification that __delayLoadHelper is done
    // You can examine the members of the DelayLoadInfo structure
    // pointed to by pdli and raise an exception if you desire
    break;
}

return(fp);
}

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

// Tell __delayLoadHelper to call my hook function
PfnDllHook __pfnDllNotifyHook = DllHook;
PfnDllHook __pfnDllFailureHook = DllHook;

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

```

```

/*****
Module: DelayLoadLib.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

```

```

#include "..\CmnHdr.h"      /* See Appendix A. */
#include <Windowsx.h>
#include <tchar.h>

```

```

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

```

```

#define DELAYLOADLIBAPI extern "C" __declspec(dllexport)
#include "DelayLoadLib.h"

```

```

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

```

```

int fnLib() {
    return(321);
}

```

```

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

```

```

int fnLib2() {
    return(123);
}

```

```

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

```

DelayLoadLib.h

```

/*****
Module: DelayLoadLib.h
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

```

```

#ifndef DELAYLOADLIBAPI
#define DELAYLOADLIBAPI extern "C" __declspec(dllimport)
#endif

```

```

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

```

```

DELAYLOADLIBAPI int fnLib();
DELAYLOADLIBAPI int fnLib2();

```

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

20.4 函数转发器

函数转发器是DLL的输出节中的一个项目，用于将对一个函数的调用转至另一个 DLL 中的另一个函数。例如，如果在 Windows 2000 的 Kernel32.dll 上运行 Visual C++ 的 DumpBin 实用程序，那么将看到类似下面的一部分输出：

```
C:\winnt\system32>DumpBin -Exports Kernel32.dll
(some output omitted)
360 167 HeapAlloc (forwarded to NTDLL.RtlAllocateHeap)
361 168 HeapCompact (000128D9)
362 169 HeapCreate (000126EF)
363 16A HeapCreateTagsW (0001279E)
364 16B HeapDestroy (00012750)
365 16C HeapExtend (00012773)
366 16D HeapFree (forwarded to NTDLL.RtlFreeHeap)
367 16E HeapLock (000128ED)
368 16F HeapQueryTagW (000127B8)
369 170 HeapReAlloc (forwarded to NTDLL.RtlReAllocateHeap)
370 171 HeapSize (forwarded to NTDLL.RtlSizeHeap)
(remainder of output omitted)
```

这个输出显示了4个转发函数。每当你的应用程序调用 HeapAlloc、HeapFree、HeapReAlloc 或 HeapSize 时，你的可执行模块就会自动与 Kernel32.dll 相链接。当激活你的可执行模块时，加载程序就加载 Kernel32.dll 并看到转发的函数实际上包含在 NTDLL.dll 中。然后它也加载 NTDLL.dll。当你的可执行模块调用 HeapAlloc 时，它实际上调用的是 NTDLL.dll 中的 RtlAllocateHeap 函数。系统中的任何地方都不存在 HeapAlloc 函数。

如果调用下面的函数，GetProcAddress 就会查看 Kernel32 的输出节，发现 HeapAlloc 是个转发函数，然后按递归方式调用 GetProcAddress 函数，查找 NTDLL.dll 的输出节中的 RtlAllocateHeap。

```
GetProcAddress(GetModuleHandle("Kernel32"), "HeapAlloc");
```

也可以利用 DLL 模块中的函数转发器。最容易的方法是像下面这样使用一个 pragma 指令：

```
// Function forwarders to functions in DllWork
#pragma comment(linker, "/export:SomeFunc=DllWork.SomeOtherFunc")
```

这个 pragma 告诉链接程序，被编译的 DLL 应该输出一个名叫 SomeFunc 的函数。但是 SomeFunc 函数的实现实际上位于另一个名叫 SomeOtherFunc 的函数中，该函数包含在称为 DllWork.dll 的模块中。必须为你想要转发的每个函数创建一个单独的 pragma 代码行。

20.5 已知的DLL

操作系统提供的某些 DLL 得到了特殊的处理。这些 DLL 称为已知的 DLL。它们与其他 DLL 基本相同，但是操作系统总是在同一个目录中查找它们，以便对它们进行加载操作。在注册表中有下面的关键字：

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
Session Manager\KnownDLLs
```

在我的计算机上使用 RegEdit.exe 实用程序时显示的是如图 20-8 所示的对话框。

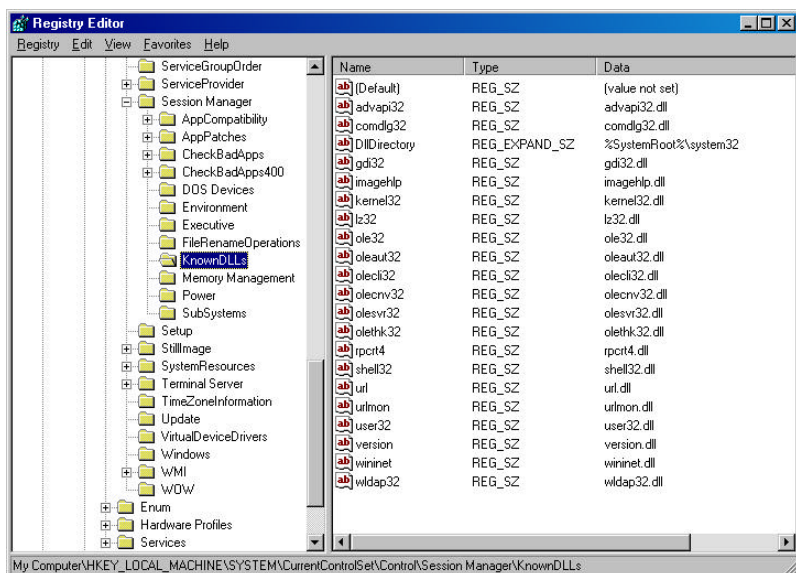


图20-8 使用 RegEdit .exe实用程序时显示的子关键字对话框

如你所见，这个关键字包含一组值的名字，这些名字是某些 DLL 的名字。每个值名字都有一个数据值，该值恰好与带有 .dll 文件扩展名的值名字相同（不过，情况并非完全如此，我将在下面的例子中加以说明）。当 LoadLibrary 或 LoadLibraryEx 被调用时，这些函数首先查看是否传递了包含 .dll 扩展名的 DLL 名字。如果没有传递，那么它们将使用通常的搜索规则来搜索 DLL。

如果确实设定了 .dll 扩展名，那么这些函数将删除扩展名，然后搜索注册表关键字 KnownDLL，以便确定它是否包含匹配的值名字。如果没有找到匹配的名字，便使用通常的搜索规则。但是，如果找到了匹配的值名字，系统将查找相关的值数据，并设法使用值数据来加载 DLL。系统也开始在注册表中的 DllDirectory 值数据指明的目录中搜索 DLL。按照默认设置，Windows 2000 上的 DllDirectory 值的数据是 %SystemRoot%\System32。

为了更好地说明情况，假设将下面的值添加给注册表关键字 KnownDLL：

Value name: SomeLib
Value data: SomeOtherLib.dll

当调用下面的函数时，系统将使用通常的搜索规则来查找该文件：

```
LoadLibrary("SomeLib");
```

但是，如果调用下面的函数，系统将会发现有一个匹配的值名字（记住，当系统检查注册表的值名字时，它将删除扩展名 .dll）。

```
LoadLibrary("SomeLib.dll");
```

这时，系统设法加载称为 SomeOtherLib.dll 的文件，而不是加载 SomeLib.dll。它首先在 %SystemRoot%\System32 目录中查找 SomeOtherLib.dll。如果它在该目录中找到了文件，它就加载该文件。如果文件不在该目录中，LoadLibrary(Ex) 运行失败并返回 NULL，同时，对 GetLastError 的调用将返回 2 (ERROR_FILE_NOT_FOUND)。

20.6 DLL 转移

Windows 98 Windows 98 不支持 DLL 转移。

当Windows刚刚开发成功时，RAM和磁盘空间是非常宝贵的。因此Windows在设计时总是尽可能多地安排资源的共享，以节省宝贵的存储器资源。为了达到这个目的，Microsoft建议，多个应用程序共享的任何模块，如C/C++运行期库和Microsoft基础类（MFC）DLL等，应该放入Windows的系统目录中。这样，系统就能够方便地找到共享文件。

随着时间的推移，这变成一个非常严重的问题，因为安装程序会用旧文件或尚未完全实现向后兼容的新文件来改写该目录中的文件。这将使用户的其他应用程序无法正确地运行。今天，硬盘的容量已经非常大，并且很便宜，RAM的容量也相当富裕，价格也便宜了一些。因此，Microsoft改变了原先的开发策略，非常支持你将应用程序的所有文件放入它们自己的目录中，而不要去碰Windows的系统目录中的任何东西。这样，你的应用程序就不会损坏别的应用程序，别的应用程序也不会损坏你的应用程序。

为了给你提供相应的帮助，Microsoft给Windows 2000增加了一个DLL转移特性。这个特性能够强制操作系统的加载程序首先从你的应用程序目录中加载文件模块。只有当加载程序无法在应用程序目录中找到该文件时，它才搜索其他目录。

为了强制加载程序总是首先查找应用程序的目录，要做的工作就是在应用程序的目录中放入一个文件。该文件的内容可以忽略，但是该文件必须称为AppName.local。

例如，如果有一个可执行文件的名字是 SuperApp.exe，那么转移文件必须称为 SuperApp.exe.local。

在系统内部，LoadLibrary(Ex)已经被修改，以便查看是否存在该文件。如果应用程序的目录中存在该文件，该目录中的模块就已经被加载。如果应用程序的目录中不存在这个模块，LoadLibrary(Ex)将正常运行。

对于已经注册的COM对象来说，这个特性是非常有用的。它使应用程序能够将它的COM对象DLL放入自己的目录，这样，注册了相同COM对象的其他应用程序就无法干扰你的操作。

20.7 改变模块的位置

每个可执行模块和DLL模块都有一个首选的基地址，用于标识模块应该映射到的进程地址空间中的理想内存地址。当创建一个可执行模块时，链接程序将该模块的首选基地址设置为0x00400000。如果是DLL模块，链接程序设置的首选基地址是0x10000000。使用Visual Studio的DumpBin实用程序（带有/Headers开关），可以看到一个映像的首选基地址。下面是使用DumpBin来转储它自己的头文件信息的例子：

```
C:\>DUMPBIN /headers dumpbin.exe
```

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

```
Dump of file dumpbin.exe
```

```
PE signature found
```

```
File Type: EXECUTABLE IMAGE
```

```
FILE HEADER VALUES
```

```
    14C machine (i386)  
      3 number of sections  
3588004A time date stamp Wed Jun 17 10:43:38 1998  
      0 file pointer to symbol table
```

```

    0 number of symbols
    E0 size of optional header
    10F characteristics
        Relocations stripped
        Executable
        Line numbers stripped
        Symbols stripped
        32 bit word machine

OPTIONAL HEADER VALUES
    10B magic #
    6.00 linker version
    1000 size of code
    2000 size of initialized data
        0 size of uninitialized data
    1320 RVA of entry point
    1000 base of code
    2000 base of data
    400000 image base          <-- Module's preferred base address
    1000 section alignment
    1000 file alignment
    4.00 operating system version
    0.00 image version
    4.00 subsystem version
        0 Win32 version
    4000 size of image
    1000 size of headers
    127E2 checksum
        3 subsystem (Windows CUI)
        0 DLL characteristics
    100000 size of stack reserve
    1000 size of stack commit
    :

```

当这个可执行模块被调用时，操作系统加载程序为新进程创建一个虚拟地址。然后该加载程序将可执行模块映射到内存地址 0x00400000，并将 DLL 模块映射到 0x10000000。为什么这个首选基地址这么重要呢？让我们看一看下面的代码：

```

int g_x;

void Func() {
    g_x = 5;    // This is the important line.
}

```

当编译器处理 Func 函数时，该编译器和链接程序创建类似下面的机器代码：

```
MOV    [0x00414540], 5
```

换句话说，编译器和链接程序创建的机器代码实际上是在变量 g_x 的地址 0x00414540 中硬编码的代码。该地址位于机器代码中，用于标识变量在进程的地址空间中的绝对位置。但是，当并且仅当可执行模块加载到它的首选基地址 0x00400000 中时，这个内存地址才是正确的。

如果在一个 DLL 模块中我们拥有与上面完全相同的代码，那将会如何呢？在这种情况下，编译器和链接程序将生成类似下面的机器代码：

```
MOV    [0x10014540], 5
```

同样，注意 DLL 的变量 g_z 的虚拟内存地址是在磁盘驱动器上的 DLL 文件映像中硬编码的代码。而且，如果该 DLL 确实是在它的首选基地址上加载的，那么这个内存地址是绝对正确的。

现在假设你设计的应用程序需要两个DLL。按照默认设置,链接程序将.exe模块的首选基地址设置为0x00400000,同时,链接程序将两个DLL模块的首选基地址均设置为0x10000000。如果想要运行.exe模块,那么加载程序便创建该虚拟地址空间,并将.exe模块映射到内存地址0x00400000中。然后加载程序将第一个DLL映射到内存地址0x10000000中。但是,当加载程序试图将第二个DLL映射到进程的地址空间中去时,它将无法把它映射到该模块的首选基地址中,必须改变该DLL模块的位置,将它放到别的什么地方。

改变可执行(或DLL)模块的位置是个非常可怕的过程,应该采取措施避免这样的操作。为什么要避免这样的操作呢?假设加载程序将第二个DLL的地址改到0x20000000。这时,将变量g_x的值改为5的代码应该是:

```
MOV    [0x20014540], 5
```

但是文件映像中的代码却类似下面的样子:

```
MOV    [0x10014540], 5
```

如果文件映像的代码被允许执行,那么第一个DLL模块中大约有4个字节的值将被值5改写。这是不能允许的。加载程序必须修改这个代码。当链接程序创建你的模块时,它将把一个移位节嵌入产生的文件中。这一节包含一个字节位移的列表。每个字节位移用于标识一个机器代码指令使用的内存地址。如果加载程序能够将一个模块映射到它的首选基地址中,那么系统将永远不会访问模块的移位节。这当然是我们所希望的——你永远不希望使用移位节。

另一方面,如果该模块不能映射到它的首选基地址中,加载程序便打开模块的移位节,并对所有项目重复执行该操作。对于找到的每个项目,加载程序将转至包含要修改机器代码指令的存储器页面。然后它抓取机器指令当前正在使用的内存地址,并将模块的首选基地址与模块实际映射到的地址之间的差与该地址相加。

因此,在上面的例子中,第二个DLL被映射到0x20000000,但是它的首选基地址是0x10000000。它产生的差是0x10000000,然后这个差与机器代码指令的地址相加,产生的结果如下:

```
MOV    [0x20014540], 5
```

现在,第二个DLL中的代码将能够正确地引用它的变量g_x。

当模块不能映射到它的首选基地址中去时,将会出现下面两个主要问题:

- 加载程序必须重复经过移位节,并且要修改模块的许多代码。这会影响到系统的运行速度,并且会增加应用程序的初始化时间。
- 当加载程序将数据写入模块的代码页面时,系统的写入时拷贝(copy-on-write)机制将强制这些页面被系统的页文件拷贝。

上面的第二个问题确实会产生非常不良的作用。它意味着模块的代码页面不再能够从该磁盘上的模块文件映像中删除和重新加载。相反,这些页面将在需要时从系统的页文件中来回倒腾。这也会影响系统的运行速度。但是,还有更为糟糕的事情。由于页文件拷贝了模块的所有代码页面,因此系统只有较少的存储器可供系统中的所有进程来运行。这就限制了用户的电子表格、文字处理文档、CAD图形和位图的大小。

另外,可以创建一个不包含移位节的可执行模块或DLL模块。当创建该模块时,你可以将/FIXED开关传递给链接程序。使用这个开关,能够使模块变得比较小一些,但是这意味着模块不能被改变位置。如果模块不能加载到它的首选基地址,那么它就根本无法加载。如果加载程序必须改变模块的位置,但是却不存在用于该模块的移位节,那么加载程序就会撤消整个进程,并且向用户显示一条“进程异常终止”的消息。

对于只包含资源的DLL来说,这是个问题。只包含资源的DLL中是没有代码的,因此,使用/FIXED开关来链接该DLL是很有意义的。但是,如果只有资源的DLL不能加载到它的首选基地址,那么该模块就根本不能加载。这很奇怪。为了解决这个问题,链接程序允许你用嵌入头文件中的信息来创建一个模块,以指明该模块不包含移位信息,因为不需要这样的信息。Windows 2000加载程序可以使用这个头文件信息,并且允许加载只包含资源的DLL,而不会降低系统的运行速度或者损害页文件的空间。

若要创建不需要进行任何移位的文件映像,请使用/SUBSYSTEM:WINDOWS,5.0开关,或者使用/SUBSYSTEM:CONSOLE,5.0开关,但是不要设定/FIXED开关。如果链接程序确定模块中没有什么东西需要进行移位设置,那么它就从模块中删除移位节,并且关闭头文件中的专用IMAGE_FILE_RELOCS_STRIPPED标志。当Windows 2000加载该模块时,它知道该模块可以移位(因为IMAGE_FILE_RELOCS_STRIPPED标志是关闭的),但是该模块没有移位(因为不存在移位节)。请注意,这是Windows 2000加载程序的一个新特性,它说明了/SUBSYSTEM开关的结尾处为什么需要带有5.0。

现在你已经知道首选基地址的重要性了。所以,如果你有多个模块需要加载到单个地址空间中,必须为每个模块设置不同的首选基地址。Microsoft Visual Studio的Project Settings(项目设置)对话框使得这项操作变得非常容易。你只需要选定Link(链接)选项卡,再选定Output(输出)类别。在Base Address(基地址)域中(该域默认为空),可以输入一个数字。在图20-9中,我将我的DLL模块的基地址设置为0x20000000。

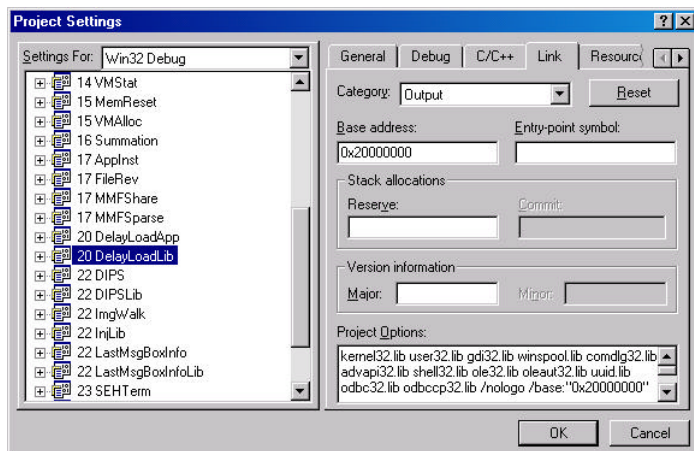


图20-9 Project Settings 对话框

另外,始终都应该从高位内存地址开始加载DLL,然后逐步向下加载到低位内存地址,以减少地址空间中出现的碎片。

注意 首选基地址必须始终从分配粒度边界开始。在迄今为止的所有平台上,系统的分配粒度是64 KB。将来这个分配粒度可能发生变化。第13章已经对分配粒度进行了详细的介绍。

好了,现在已经对所有的内容做了介绍。但是,如果将许多模块加载到单个地址空间中,情况将会如何呢?如果有一种非常容易的方法,可以为所有的模块设置很好的首选基地址,那就好了。幸运的是,这种方法确实有。

Visual Studio配有一个实用程序,称为Rebase.exe。如果运行不带任何命令行参数的Rebase

程序，会得到下面的使用信息：

```
usage:
REBASE [switches]
        [-R image-root [-G filename] [-O filename] [-N filename]]
        image-names...

One of -b and -i switches are mandatory.

[-a] Used with -x.  extract All debug info into .dbg file
[-b InitialBase] specify initial base address
[-c coffbase_filename] generate coffbase.txt
        -C includes filename extensions, -c does not
[-d] top down rebase
[-f] Strip relocs after rebasing the image

[-i coffbase_filename] get base addresses from coffbase_filename
[-l logFilePath] write image bases to log file.
[-p] Used with -x.  Remove private debug info when extracting
[-q] minimal output
[-s] just sum image range
[-u symbol_dir] Update debug info in .DBG along this path
[-v] verbose output
[-x symbol_dir] extract debug info into separate .DBG file first
[-z] allow system file rebasing
[-?] display this message

[-R image_root] set image root for use by -G, -O, -N
[-G filename] group images together in address space
[-O filename] overlay images in address space
[-N filename] leave images at their original address
        -G, -O, -N, may occur multiple times.  File "filename"
        contains a list of files (relative to "image-root")
```

Platform SDK文档对Rebase实用程序进行了介绍，因此在这里就不再对它详细说明了。不过要补充的一点是，这个实用程序并没有什么奇特之处。从内部来说，它只是分别为每个指定的文件调用ReBaseImage函数：

```
BOOL ReBaseImage(
    PSTR CurrentImageName,    // Pathname of file to be rebased
    PSTR SymbolPath,          // Symbol file path so debug info
                                // is accurate
    BOOL fRebase,              // TRUE to actually do the work; FALSE
                                // to pretend
    BOOL fRebaseSysFileOk,     // FALSE to not rebase system images
    BOOL fGoingDown,           // TRUE to rebase the image below
                                // an address
    ULONG CheckImageSize,      // Maximum size that image can grow to
    ULONG* pOldImageSize,      // Receives original image size
    ULONG* pOldImageBase,      // Receives original image base address
    ULONG* pNewImageSize,      // Receives new image size
    ULONG* pNewImageBase,      // Receives new image base address
    ULONG TimeStamp);          // New timestamp for image
```

当你执行Rebase程序，为它传递一组映像文件名时，它将执行下列操作：

- 1) 它能够仿真创建一个进程的地址空间。
- 2) 它打开通常被加载到该地址空间中的所有模块。

- 3) 它仿真改变各个模块在仿真地址空间中的位置, 这样, 各个模块就不会重叠。
- 4) 对于已经移位的模块, 它会分析该模块的移位节, 并修改磁盘上的模块文件中的代码。
- 5) 它会更新每个移位模块的头文件, 以反映新的首选基地址。

Rebase是个非常出色的工具, 建议尽可能使用这个工具。应该在接近你的应用程序模块创建周期结束时运行这个实用程序, 直到所有模块创建完成。另外, 如果使用 Rebase实用程序, 可以忽略 Project Settings对话框中的基地址的设置。链接程序将为 DLL提供一个基地址 0x10000000, 但是Rebase会修改这个地址。

顺便要指出的是, 决不应该改变操作系统配备的任何模块的地址。 Microsoft在销售 Windows操作系统之前, 在操作系统提供的所有文件上运行了 Rebase程序, 因此, 如果将它们映射到单个地址空间中, 所有的操作系统模块都不会重叠。

我给第4章中介绍的ProcessInfo.exe应用程序添加了一个特殊的性质。这个工具显示了一个列表, 里面包括进程的地址空间中存在的所有模块。在 BaseAddr列的下面, 会看到模块加载到的虚拟内存地址。在 BaseAddr列的右边, 是ImagAddr列。通常这个列是空的, 这表示模块加载到了它的首选基地址中。你希望看到所有模块都是这样。但是如果出现带有括号的另一个地址, 就表示该模块没有加载到它的首选基地址中, 这一列中的数字表示从模块的磁盘文件的头文件信息中读取的该模块的首选基地址。

下面是正在查看 Acrord32.exe进程的 ProcessInfo.exe工具。注意, 有些模块确实加载到了它们的首选基地址中, 有些模块则没有。你还会发现所有模块的首选基地址都是0x10000000, 表示它们是DLL模块, 并且这些模块的创建者并没有考虑改变地址的问题 (见图20-10)。

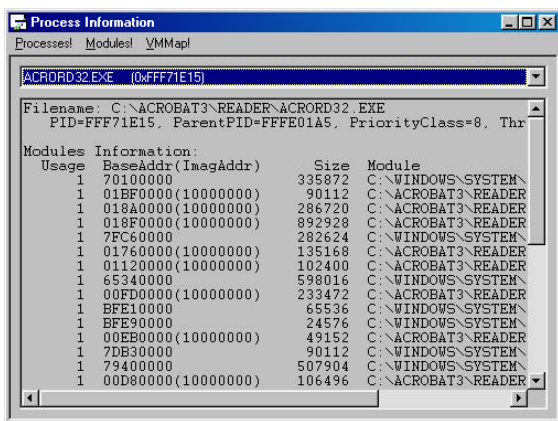


图20-10 Process Information 窗口

20.8 绑定模块

模块的移位是非常重要的, 它能够大大提高整个系统的运行性能。但是, 还可以使用许多别的办法来提高它的性能。比如说, 可以改变应用程序的所有模块的位置。让我们回忆一下第19章中关于加载程序如何查看所有输入符号的地址的情况。加载程序将符号的虚拟地址写入可执行模块的输入节中。这样就可以参考输入的符号, 以便到达正确的内存位置。

让我们进一步考虑一下这个问题。如果加载程序将输入符号的虚拟地址写入 .exe模块的输入节, 那么拷贝输入节的这些页面将被写入虚拟地址。由于这些页面是写入时拷贝的页面, 因此这些页面将被页文件拷贝。这样我们就遇到了一个与改变模块的位置相类似的问题, 即映像文件的各个部分将与系统的页文件之间来回倒腾, 而不是在必要时将它们删除或者从文件的磁盘映像中重新读取。另外, 加载程序必须对 (所有模块的) 所有输入符号的地址进行转换, 这是很费时间的。

可以将模块绑定起来, 使应用程序能够更快的进行初始化, 并且使用较少的存储器。绑定一个模块时, 可以为该模块的输入节配备所有输入符号的虚拟地址。为了缩短初始化的时间和使用较少的存储器, 当然必须在加载模块之前进行这项操作。

Visual Studio配有另一个实用程序，名字是Bind.exe，当运行这个不带任何命令行参数的程序时，它将输出下面的信息：

```
usage: BIND [switches] image-names...
    [-?] display this message
    [-c] no caching of import dlls
    [-o] disable new import descriptors
    [-p dll search path]
    [-s Symbol directory] update any associated .DBG file
    [-u] update the image
    [-v] verbose output
    [-x image name] exclude this image from binding
    [-y] allow binding on images located above 2G
```

Bind（绑定）实用程序在Platform SDK文档中作了描述，这里就不再详细介绍了。但是与Rebase一样，这个实用程序也不是不可思议的程序。从内部来说，它为每个指定的文件重复调用BindImageEx函数：

```
BOOL BindImageEx(
    DWORD dwFlags,          // Flags giving fine control over the function
    PSTR pszImageName,      // Pathname of file to be bound
    PSTR pszDllPath,        // Search path used for locating image files
    PSTR pszSymbolPath,     // Search path used to keep debug info accurate
    PIMAGEHLP_STATUS_ROUTINE StatusRoutine); // Callback function
```

最后一个参数StatusRoutine是一个回调函数的地址，BindImageEx定期调用这个函数，这样就能够监控连接进程。下面是该函数的原型：

```
BOOL WINAPI StatusRoutine(
    IMAGEHLP_STATUS_REASON Reason, // Module/procedure not found, etc.
    PSTR pszImageName,            // Pathname of file being bound
    PSTR pszDllName,              // Pathname of DLL
    ULONG_PTR VA,                 // Computed virtual address
    ULONG_PTR Parameter);         // Additional info depending on Reason
```

当执行Bind程序，传递给它一个映像文件名时，它将执行下列操作：

- 1) 打开指定映像文件的输入节。
- 2) 对于输入节中列出的每个DLL，它打开该DLL文件，查看它的头文件以确定它的首选基地址。
- 3) 查看DLL的输出节中的每个输入符号。
- 4) 取出符号的RVA，并将模块的首选基地址与它相加。将可能产生的输入符号的虚拟地址写入映像文件的输入节中。
- 5) 将某些辅助信息添加到映像文件的输入节中。这些信息包括映像文件绑定到的所有DLL模块的名字和这些模块的时戳。

在第19章中，我们使用DumpBin实用程序来查看Calc.exe的输入节的内容。该输出信息的底部显示了第5个步骤中添加的已链接输入信息。下面是输出信息的有关部分：

```
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
```

可以看到 Calc.exe 连接到了哪些模块，方括号中的数字表示 Microsoft 是在何时创建每个 DLL 模块的。这个 32 位的时戳值在方括号的后面展开了，并以人们能够识别的字符串显示出来。

在执行整个进程期间，Bind 程序做了两个重要的假设：

- 当进程初始化时，需要的 DLL 实际上加载到了它们的首选基地址中。可以使用前面介绍的 Rebase 实用程序来确保这一点。
- 自从绑定操作执行以来，DLL 的输出节中引用的符号的位置一直没有改变。加载程序通过将每个 DLL 的时戳与上面第 5 个步骤中保存的时戳进行核对来核实这个情况。

当然，如果加载程序确定上面的两个假设中有一个是假的，那么 Bind 就没有执行上面所说的有用的操作，加载程序必须通过人工来修改可执行模块的输入节，就像它通常所做的那样。但是，如果加载程序发现模块已经连接，需要的 DLL 已经加载到它们的首选基地址中，而且时戳也匹配，那么它实际上已经无事可做。它不必改变任何模块的位置，也不必查看任何输入函数的虚拟地址。该应用程序只管启动运行就是了。

此外，它不需要来自系统的页文件的任何存储器。这太好了，我们简直拥有世界上最出色的系统。目前销售的商用应用程序许多都不具备相应的移位和绑定特性。

好了，现在你已经知道应该将应用程序配有的所有模块连接起来。但是应该在什么时候进行模块的连接呢？如果你在你的公司连接这些模块，可以将它们与你已经安装的系统 DLL 绑定起来，而这些系统 DLL 并不一定是用户已经安装的。由于不知道用户运行的是 Windows 98 还是 Windows NT，或者是 Windows 2000，也不知道这些操作系统是否已经安装了服务软件包，因此应该将绑定操作作为应用程序的安装操作的一部分来进行。

当然，如果用户能够对 Windows 98 和 Windows 2000 进行双重引导，那么绑定的模块可能对这两个操作系统之一来说是不正确的。另外，如果用户在 Windows 2000 下安装你的应用程序，然后又升级到你的服务软件包，那么模块的绑定也是不正确的。在这些情况下，你和用户都可能无能为力。Microsoft 应该在销售操作系统时配备一个实用程序，使得操作系统升级后能够自动重新绑定每个模块。不过，现在还不存在这样的实用程序。