1.概论

先来阐述一下 DLL(Dynamic Linkable Library)的概念，你可以简单的把 DLL 看成一种仓库，它提供给你一些可以直接拿来用的变量、函数或类。在仓库的发展史上经历了“无库－静态链接库－动态链接库”的时代。静态链接库与动态链接库都是共享代码的方式，如果采用静态链接库，则无论你愿不愿意，lib 中的指令都被直接包含在最终生成的 EXE 文件中了。但是若使用 DLL，该 DLL 不必被包含在最终 EXE 文件中，EXE 文件执行时可以“动态”地引用和卸载这个与 EXE 独立的 DLL 文件。静态链接库和动态链接库的另外一个区别在于静态链接库中不能再包含其他的动态链接库或者静态库，而在动态链接库中还可以再包含其他的动态或静态链接库。

对动态链接库，我们还需建立如下概念：

（1）DLL 的编制与具体的编程语言及编译器无关

只要遵循约定的 DLL 接口规范和调用方式，用各种语言编写的 DLL 都可以相互调用。譬如 Windows 提供的系统 DLL（其中包括了 Windows 的 API），在任何开发环境中都能被调用，不在乎其是 Visual Basic、Visual C++还是 Delphi。

（2）动态链接库随处可见

我们在 Windows 目录下的 system32 文件夹中会看到 kernel32.dll、user32.dll 和 gdi32.dll，windows 的大多数 API 都包含在这些 DLL 中。kernel32.dll 中的函数主要处理内存管理和进程调度；user32.dll 中的函数主要控制用户界面；gdi32.dll 中的函数则负责图形方面的操作。

一般的程序员都用过类似 MessageBox 的函数，其实它就包含在 user32.dll 这个动态链接库中。由此可见 DLL 对我们来说其实并不陌生。

(3)VC 动态链接库的分类

Visual C++支持三种 DLL，它们分别是 Non-MFC DLL（非 MFC 动态库）、MFC Regular DLL（MFC 规则 DLL）、MFC Extension DLL（MFC 扩展 DLL）。

非 MFC 动态库不采用 MFC 类库结构，其导出函数为标准的 C 接口，能被非 MFC 或 MFC 编写的应用程序所调用；MFC 规则 DLL 包含一个继承自 CWinApp 的类，但其无消息循环；MFC 扩展 DLL 采用 MFC 的动态链接版本创建，它只能被用 MFC 类库所编写的应用程序所调用。

由于本文篇幅较长，内容较多，势必需要先对阅读本文的有关事项进行说明，下面以问答形式给出。

问：本文主要讲解什么内容？

答：本文详细介绍了 DLL 编程的方方面面，努力学完本文应可以对 DLL 有较全面的掌握，并能编写大多数 DLL 程序。

问：如何看本文？

答：本文每一个主题的讲解都附带了源代码例程，可以随文下载（每个工程都经 WINRAR 压缩）。所有这些例程都由笔者编写并在 VC++6.0 中调试通过。

当然看懂本文不是读者的最终目的，读者应亲自动手实践才能真正掌握 DLL 的奥妙。

问：学习本文需要什么样的基础知识？

答：如果你掌握了 C，并大致掌握了 C++，了解一点 MFC 的知识，就可以轻松地看懂本文。

2.静态链接库

对静态链接库的讲解不是本文的重点，但是在具体讲解 DLL 之前，通过一个静态链接库的例子可以快速地帮助我们建立“库”的概念。

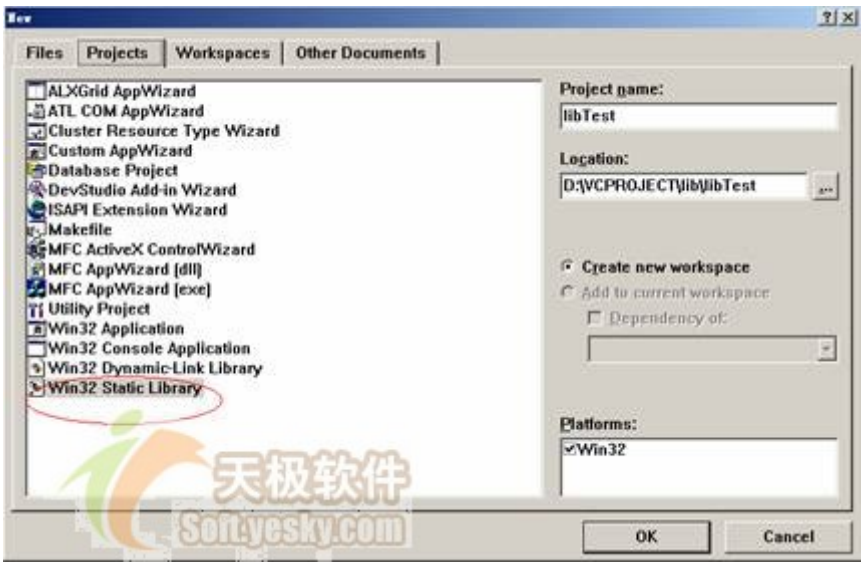

图 1 建立一个静态链接库

如图 1，在 VC++6.0 中 new 一个名称为 libTest 的 static library 工程（单击此处下载本工程），并新建 lib.h 和 lib.cpp 两个文件，lib.h 和 lib.cpp 的源代码如下：

```
//文件：lib.h

#ifndef LIB_H
#define LIB_H
extern "C" int add(int x,int y);          //声明为 C 编译、连接方式的外部函数
#endif

//文件：lib.cpp

#include "lib.h"
int add(int x,int y)
{
   return x + y;
}
```

编译这个工程就得到了一个.lib 文件，这个文件就是一个函数库，它提供了 add 的功能。将头文件和.lib 文件提交给用户后，用户就可以直接使用其中的 add 函数了。

标准 Turbo C2.0 中的 C 库函数（我们用来的 scanf、printf、memcpy、strcpy 等）就来自这种静态库。

下面来看看怎么使用这个库，在 libTest 工程所在的工作区内 new 一个 libCall 工程。libCall 工程仅包含一个 main.cpp 文件，它演示了静态链接库的调用方法，其源代码如下：

```
#include <stdio.h>
#include "..\lib.h"
#pragma comment( lib, "..\\debug\\libTest.lib" )     //指定与静态库一起连接

int main(int argc, char* argv[])
{
   printf( "2 + 3 = %d", add( 2, 3 ) );
}
```

静态链接库的调用就是这么简单，或许我们每天都在用，可是我们没有明白这个概念。代码中#pragma comment(lib , "..\\debug\\libTest.lib")的意思是指本文件生成的.obj 文件应与 libTest.lib 一起连接。如果不用 #pragma comment 指定，则可以直接在 VC++中设置，如图 2，依次选择 tools、options、directories、library files 菜单或选项，填入库文件路径。图 2 中加红圈的部分为我们添加的 libTest.lib 文件的路径。

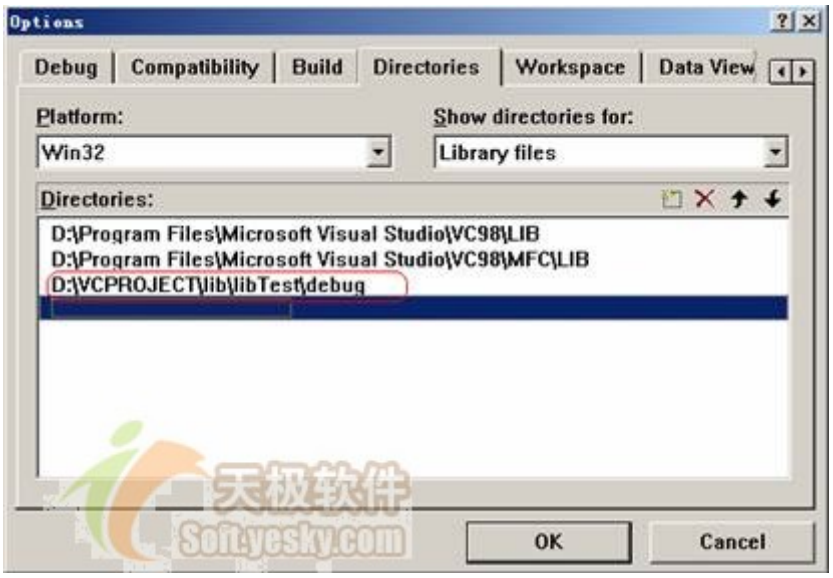

图 2 在 VC 中设置库文件路径

这个静态链接库的例子至少让我们明白了库函数是怎么回事，它们是哪来的。我们现在有下列模糊认识了：

（1）库不是个怪物，编写库的程序和编写一般的程序区别不大，只是库不能单独执行；

（2）库提供一些可以给别的程序调用的东东，别的程序要调用它必须以某种方式指明它要调用之。

以上从静态链接库分析而得到的对库的懵懂概念可以直接引申到动态链接库中，动态链接库与静态链接库在编写和调用上的不同体现在库的外部接口定义及调用方式略有差异。

3.库的调试与查看

在具体进入各类 DLL 的详细阐述之前，有必要对库文件的调试与查看方法进行一下介绍，因为从下一节开始我们将面对大量的例子工程。

由于库文件不能单独执行，因而在按下 F5（开始 debug 模式执行）或 CTRL+F5（运行）执行时，其弹出如图 3 所示的对话框，要求用户输入可执行文件的路径来启动库函数的执行。这个时候我们输入要调用该库的 EXE 文件的路径就可以对库进行调试了，其调试技巧与一般应用工程的调试一样。

图 3 库的调试与“运行”

通常有比上述做法更好的调试途径，那就是将库工程和应用工程（调用库的工程）放置在同一 VC 工作区，只对应用工程进行调试，在应用工程调用库中函数的语句处设置断点，执行后按下 F11，这样就单步进入了库中的函数。第 2 节中的 libTest 和 libCall 工程就放在了同一工作区，其工程结构如图 4 所示。

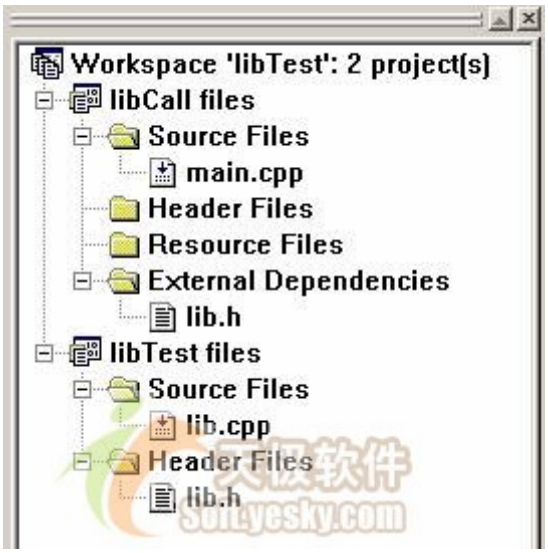

图 4 把库工程和调用库的工程放入同一工作区进行调试

上述调试方法对静态链接库和动态链接库而言是一致的。所以本文提供下载的所有源代码中都包含了库工程和调用库的工程，这二者都被包含在一个工作区内，这是笔者提供这种打包下载的用意所在。

动态链接库中的导出接口可以使用 Visual C++的 Depends 工具进行查看，让我们用 Depends 打开系统目录中的 user32.dll，看到了吧？红圈内的就是几个版本的 MessageBox 了！原来它真的在这里啊，原来它就在这里啊！

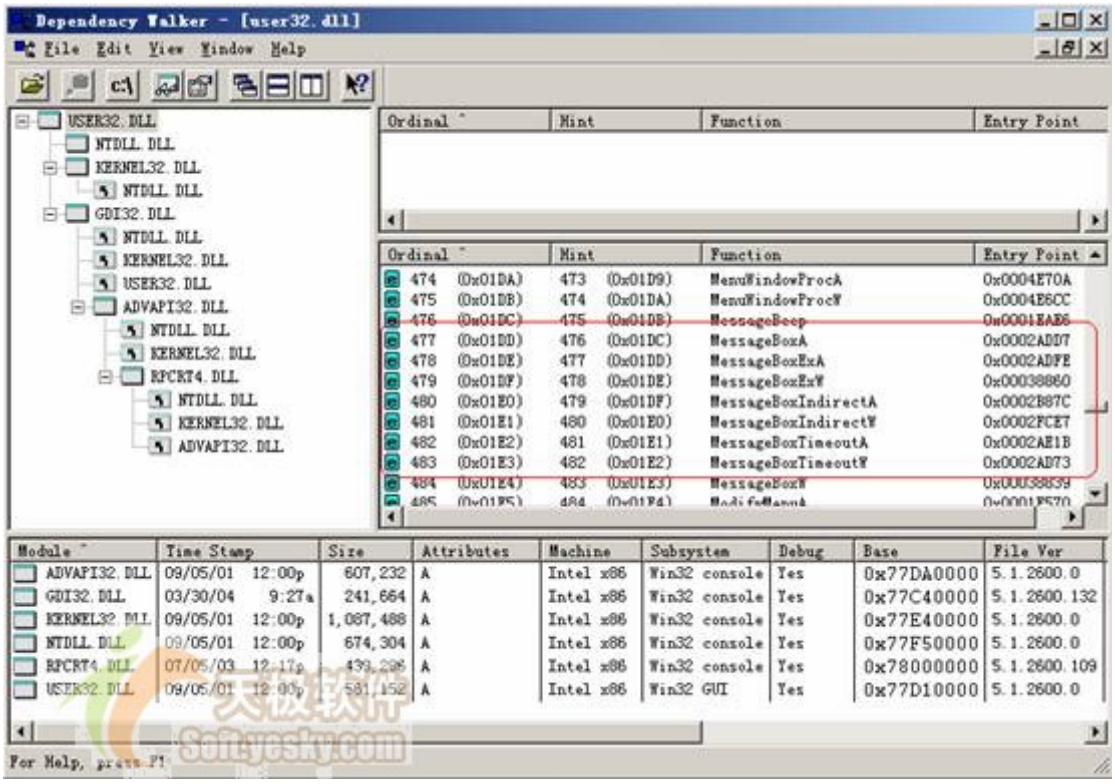

图 5　用 Depends 查看 DLL

当然 Depends 工具也可以显示 DLL 的层次结构，若用它打开一个可执行文件则可以看出这个可执行文件调用了哪些 DLL。

好，让我们正式进入动态链接库的世界，先来看看最一般的 DLL，即非 MFC DLL。

4.非 MFC DLL

4.1 一个简单的 DLL

第 2 节给出了以静态链接库方式提供 add 函数接口的方法，接下来我们来看看怎样用动态链接库实现一个同样功能的 add 函数。

如图 6，在 VC++中 new 一个 Win32 Dynamic-Link Library 工程 dllTest（单击此处下载本工程附件）。注意不要选择 MFC AppWizard(dll)，因为用 MFC AppWizard(dll)建立的将是第 5、6 节要讲述的 MFC 动态链接库。

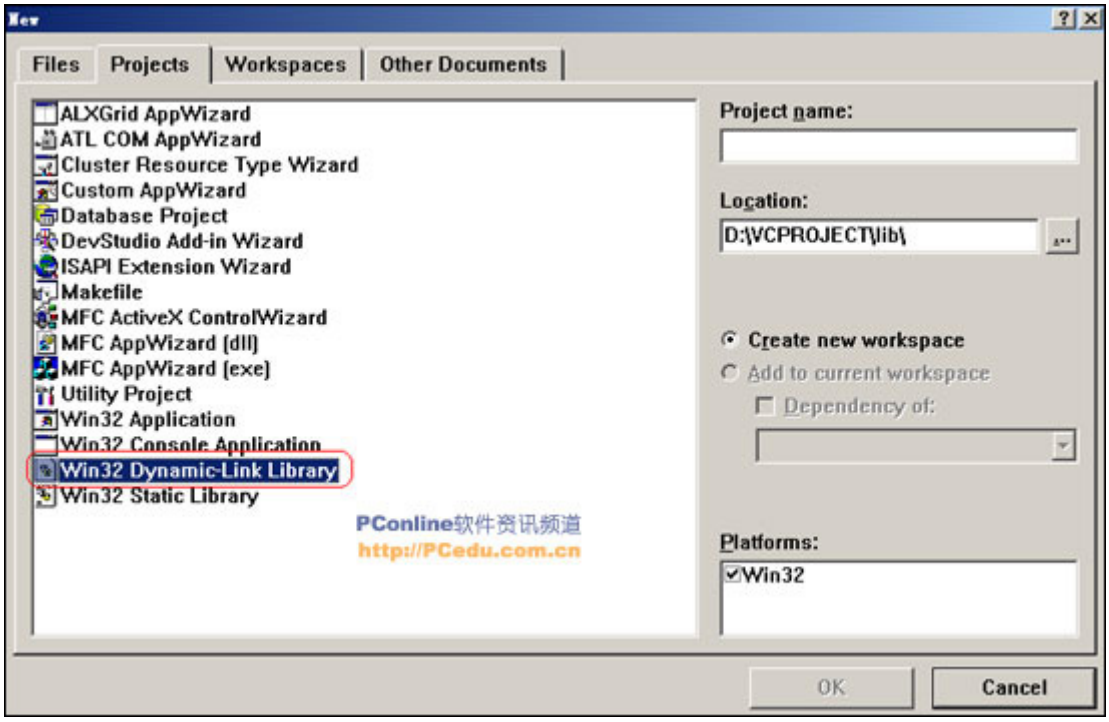

图 6 建立一个非 MFC DLL

在建立的工程中添加 lib.h 及 lib.cpp 文件，源代码如下：

```
/* 文件名：lib.h */

#ifndef LIB_H

#define LIB_H

extern "C" int __declspec(dllexport)add(int x, int y);

#endif


/* 文件名：lib.cpp */

#include "lib.h"

int add(int x, int y)

{

return x + y;

}
```

与第 2 节对静态链接库的调用相似，我们也建立一个与 DLL 工程处于同一工作区的应用工程 dllCall，它调用 DLL 中的函数 add，其源代码如下：

```
#include <stdio.h>

#include <windows.h>

typedef int(*lpAddFun)(int, int); //宏定义函数指针类型
```

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

{

HINSTANCE hDll; //DLL 句柄

lpAddFun addFun; //函数指针

hDll = LoadLibrary("..\\Debug\\dllTest.dll");

if (hDll != NULL)

{

addFun = (lpAddFun)GetProcAddress(hDll, "add");

if (addFun != NULL)

{

int result = addFun(2, 3);

printf("%d", result);

}

FreeLibrary(hDll);

}

return 0;

}
```

分析上述代码，dllTest 工程中的 lib.cpp 文件与第 2 节静态链接库版本完全相同，不同在于 lib.h 对函数 add 的声明前面添加了__declspec(dllexport)语句。这个语句的含义是声明函数 add 为 DLL 的导出函数。DLL 内的函数分为两种：

(1)DLL 导出函数，可供应用程序调用；

(2) DLL 内部函数，只能在 DLL 程序使用，应用程序无法调用它们。

而应用程序对本 DLL 的调用和对第 2 节静态链接库的调用却有较大差异，下面我们来逐一分析。

首先，语句 typedef int (* lpAddFun)(int,int)定义了一个与 add 函数接受参数类型和返回值均相同的函数指针类型。随后，在 main 函数中定义了 lpAddFun 的实例 addFun；

其次，在函数 main 中定义了一个 DLL HINSTANCE 句柄实例 hDll，通过 Win32 Api 函数 LoadLibrary

动态加载了 DLL 模块并将 DLL 模块句柄赋给了 hDll；

再次，在函数 main 中通过 Win32 Api 函数 GetProcAddress 得到了所加载 DLL 模块中函数 add 的地址并赋给了 addFun。经由函数指针 addFun 进行了对 DLL 中 add 函数的调用；

最后，应用工程使用完 DLL 后，在函数 main 中通过 Win32 Api 函数 FreeLibrary 释放了已经加载的 DLL 模块。

通过这个简单的例子，我们获知 DLL 定义和调用的一般概念：

(1)DLL 中需以某种特定的方式声明导出函数（或变量、类）；

(2)应用工程需以某种特定的方式调用 DLL 的导出函数（或变量、类）。

下面我们来对“特定的方式进行”阐述。

4.2 声明导出函数

DLL 中导出函数的声明有两种方式：一种为 4.1 节例子中给出的在函数声明中加上__declspec(dllexport)，这里不再举例说明；另外一种方式是采用模块定义(.def) 文件声明，.def 文件为链接器提供了有关被链接程序的导出、属性及其他方面的信息。

下面的代码演示了怎样同.def 文件将函数 add 声明为 DLL 导出函数（需在 dllTest 工程中添加 lib.def 文件）：

```
; lib.def :  导出 DLL 函数

LIBRARY dllTest

EXPORTS

add @ 1
```

.def 文件的规则为：

(1)LIBRARY 语句说明.def 文件相应的 DLL；

(2)EXPORTS 语句后列出要导出函数的名称。可以在.def 文件中的导出函数名后加@n，表示要导出函数的序号为 n（在进行函数调用时，这个序号将发挥其作用）；

(3).def 文件中的注释由每个注释行开始处的分号 (;) 指定，且注释不能与语句共享一行。

由此可以看出，例子中 lib.def 文件的含义为生成名为“dllTest”的动态链接库，导出其中的 add 函数，并指定 add 函数的序号为 1。

4.3 DLL 的调用方式

在 4.1 节的例子中我们看到了由“LoadLibrary-GetProcAddress-FreeLibrary”系统 Api 提供的三位一体“DLL 加载-DLL 函数地址获取-DLL 释放”方式，这种调用方式称为 DLL 的动态调用。

动态调用方式的特点是完全由编程者用 API 函数加载和卸载 DLL，程序员可以决定 DLL 文件何时加载或不加载，显式链接在运行时决定加载哪个 DLL 文件。

与动态调用方式相对应的就是静态调用方式，“有动必有静”，这来源于物质世界的对立统一。“动与静”，其对立与统一竟无数次在技术领域里得到验证，譬如静态 IP 与 DHCP、静态路由与动态路由等。从前文我们已经知道，库也分为静态库与动态库 DLL，而想不到，深入到 DLL 内部，其调用方式也分为静态与动态。“动与静”，无处不在。《周易》已认识到有动必有静的动静平衡观，《易．系辞》曰：“动静有常，刚柔断矣”。哲学意味着一种普遍的真理，因此，我们经常可以在枯燥的技术领域看到哲学的影子。

静态调用方式的特点是由编译系统完成对 DLL 的加载和应用程序结束时 DLL 的卸载。当调用某 DLL 的应用程序结束时，若系统中还有其它程序使用该 DLL，则 Windows 对 DLL 的应用记录减 1，直到所有使用该 DLL 的程序都结束时才释放它。静态调用方式简单实用，但不如动态调用方式灵活。

下面我们来看看静态调用的例子（单击此处下载本工程附件），将编译 dllTest 工程所生成的.lib 和.dll 文件拷入 dllCall 工程所在的路径，dllCall 执行下列代码：

```
#pragma comment(lib,"dllTest.lib")

//.lib 文件中仅仅是关于其对应 DLL 文件中函数的重定位信息

extern "C" __declspec(dllimport) add(int x,int y);

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

{

int result = add(2,3);

printf("%d",result);

return 0;

}
```

由上述代码可以看出，静态调用方式的顺利进行需要完成两个动作：

(1)告诉编译器与 DLL 相对应的.lib 文件所在的路径及文件名，#pragma comment(lib,"dllTest.lib")就是起这个作用。

程序员在建立一个 DLL 文件时，连接器会自动为其生成一个对应的.lib 文件，该文件包含了 DLL 导出函数的符号名及序号（并不含有实际的代码）。在应用程序里，.lib 文件将作为 DLL 的替代文件参与编译。

(2)声明导入函数，extern "C" __declspec(dllimport) add(int x,int y)语句中的__declspec(dllimport)发挥这个作用。

静态调用方式不再需要使用系统 API 来加载、卸载 DLL 以及获取 DLL 中导出函数的地址。这是因为，当程序员通过静态链接方式编译生成应用程序时，应用程序中调用的与.lib 文件中导出符号相匹配的函数符号将进入到生成的 EXE 文件中，.lib 文件中所包含的与之对应的 DLL 文件的文件名也被编译器存储在 EXE 文件内部。当应用程序运行过程中需要加载 DLL 文件时，Windows 将根据这些信息发现并加载 DLL，然后通过符号名实现对 DLL 函数的动态链接。这样，EXE 将能直接通过函数名调用 DLL 的输出函数，就象调用程序内部的其他函数一样。

4.4 DllMain 函数

Windows 在加载 DLL 的时候，需要一个入口函数，就如同控制台或 DOS 程序需要 main 函数、WIN32 程序需要 WinMain 函数一样。在前面的例子中，DLL 并没有提供 DllMain 函数，应用工程也能成功引用 DLL，这是因为 Windows 在找不到 DllMain 的时候，系统会从其它运行库中引入一个不做任何操作的缺省 DllMain 函数版本，并不意味着 DLL 可以放弃 DllMain 函数。

根据编写规范，Windows 必须查找并执行 DLL 里的 DllMain 函数作为加载 DLL 的依据，它使得 DLL 得以保留在内存里。这个函数并不属于导出函数，而是 DLL 的内部函数。这意味着不能直接在应用工程中引用 DllMain 函数，DllMain 是自动被调用的。

我们来看一个 DllMain 函数的例子（单击此处下载本工程附件）。

```
BOOL APIENTRY DllMain( HANDLE hModule,

DWORD ul_reason_for_call,

LPVOID lpReserved

)

{

switch (ul_reason_for_call)

{

case DLL_PROCESS_ATTACH:

printf("\nprocess attach of dll");

break;

case DLL_THREAD_ATTACH:

printf("\nthread attach of dll");

break;

case DLL_THREAD_DETACH:

printf("\nthread detach of dll");

break;

case DLL_PROCESS_DETACH:

printf("\nprocess detach of dll");

break;

}
```

```
return TRUE;

}
```

DllMain 函数在 DLL 被加载和卸载时被调用，在单个线程启动和终止时，DLLMain 函数也被调用，ul_reason_for_call 指明了被调用的原因。原因共有 4 种，即 PROCESS_ATTACH、PROCESS_DETACH、THREAD_ATTACH 和 THREAD_DETACH，以 switch 语句列出。
来仔细解读一下 DllMain 的函数头 BOOL APIENTRY DllMain(HANDLE hModule, WORD ul_reason_for_call, LPVOID lpReserved)。

APIENTRY 被定义为__stdcall，它意味着这个函数以标准 Pascal 的方式进行调用，也就是 WINAPI 方式；

进程中的每个 DLL 模块被全局唯一的 32 字节的 HINSTANCE 句柄标识，只有在特定的进程内部有效，句柄代表了 DLL 模块在进程虚拟空间中的起始地址。在 Win32 中，HINSTANCE 和 HMODULE 的值是相同的，这两种类型可以替换使用，这就是函数参数 hModule 的来历。

执行下列代码：

```
hDll = LoadLibrary("..\\Debug\\dllTest.dll");

if (hDll != NULL)

{

addFun = (lpAddFun)GetProcAddress(hDll, MAKEINTRESOURCE(1));

//MAKEINTRESOURCE 直接使用导出文件中的序号

if (addFun != NULL)

{

int result = addFun(2, 3);

printf("\ncall add in dll:%d", result);

}

FreeLibrary(hDll);

}
```

我们看到输出顺序为：

process attach of dll

call add in dll:5

process detach of dll

这一输出顺序验证了 DllMain 被调用的时机。

代码中的 GetProcAddress (hDll, MAKEINTRESOURCE (1))值得留意，它直接通过.def 文件中为 add 函数指定的顺序号访问 add 函数，具体体现在 MAKEINTRESOURCE (1)，MAKEINTRESOURCE 是一个通过序号获取函数名的宏，定义为（节选自 winuser.h）：

```
#define MAKEINTRESOURCEA(i) (LPSTR)((DWORD)((WORD)(i)))

#define MAKEINTRESOURCEW(i) (LPWSTR)((DWORD)((WORD)(i)))

#ifdef UNICODE

#define MAKEINTRESOURCE MAKEINTRESOURCEW

#else

#define MAKEINTRESOURCE MAKEINTRESOURCEA
```

4.5 __stdcall 约定

如果通过 VC++编写的 DLL 欲被其他语言编写的程序调用，应将函数的调用方式声明为__stdcall 方式，WINAPI 都采用这种方式，而 C/C++缺省的调用方式却为__cdecl。__stdcall 方式与__cdecl 对函数名最终生成符号的方式不同。若采用 C 编译方式(在 C++中需将函数声明为 extern "C")，__stdcall 调用约定在输出函数名前面加下划线，后面加“@”符号和参数的字节数，形如_functionname@number；而__cdecl 调用约定仅在输出函数名前面加下划线，形如_functionname。

Windows 编程中常见的几种函数类型声明宏都是与__stdcall 和__cdecl 有关的（节选自 windef.h）：

```
#define CALLBACK __stdcall //这就是传说中的回调函数

#define WINAPI __stdcall //这就是传说中的 WINAPI

#define WINAPIV __cdecl

#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为__stdcall 调用，而应用工程中仍使用 typedef int (* lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的__cdecl 调用），弹出如图 7 所示的对话框。

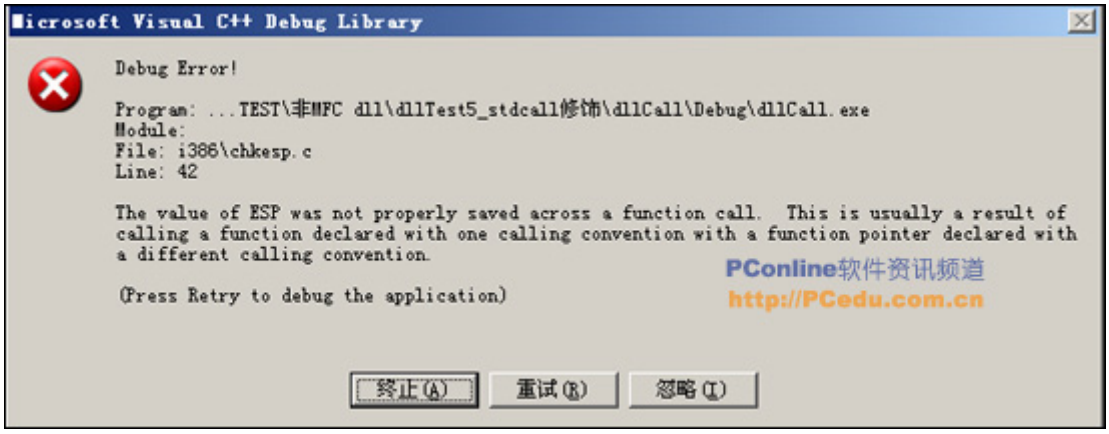

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即“This is usually a result of …”。

单击此处下载__stdcall 调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问；DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件）。

```
/* 文件名：lib.h */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;

#endif


/* 文件名：lib.cpp */

#include "lib.h"

#include <windows.h>


int dllGlobalVar;


BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)

{

switch (ul_reason_for_call)

{

case DLL_PROCESS_ATTACH:
```

```
dllGlobalVar = 100; //在 dll 被加载时，赋全局变量为 100

break;

case DLL_THREAD_ATTACH:

case DLL_THREAD_DETACH:

case DLL_PROCESS_DETACH:

break;

}

return TRUE;

}


;文件名：lib.def

;在 DLL 中导出变量

LIBRARY "dllTest"

EXPORTS

dllGlobalVar CONSTANT

;或 dllGlobalVar DATA

GetGlobalVar
```

从 lib.h 和 lib.cpp 中可以看出，全局变量在 DLL 中的定义和使用方法与一般的程序设计是一样的。若要导出某全局变量，我们需要在.def 文件的 EXPORTS 后添加：

```
变量名    CONSTANT          //过时的方法
```

或

```
变量名    DATA              //VC++提示的新方法
```

在主函数中引用 DLL 中定义的全局变量：

```
#include <stdio.h>

#pragma comment(lib,"dllTest.lib")

extern int dllGlobalVar;

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

{
```

```
printf("%d ", *(int*)dllGlobalVar);

*(int*)dllGlobalVar = 1;

printf("%d ", *(int*)dllGlobalVar);


return 0;

}
```

特别要注意的是用 extern int dllGlobalVar 声明所导入的并不是 DLL 中全局变量本身，而是其地址，应用程序必须通过强制指针转换来使用 DLL 中的全局变量。这一点，从*(int*)dllGlobalVar 可以看出。因此在采用这种方式引用 DLL 全局变量时，千万不要进行这样的赋值操作：
dllGlobalVar = 1;

其结果是 dllGlobalVar 指针的内容发生变化，程序中以后再也引用不到 DLL 中的全局变量了。

在应用工程中引用 DLL 中全局变量的一个更好方法是：

```
#include <stdio.h>

#pragma comment(lib,"dllTest.lib")

extern int _declspec(dllimport) dllGlobalVar; //用_declspec(dllimport)导入

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

{

printf("%d ", dllGlobalVar);

dllGlobalVar = 1; //这里就可以直接使用，无须进行强制指针转换

printf("%d ", dllGlobalVar);

return 0;

}
```

通过_declspec(dllimport)方式导入的就是 DLL 中全局变量本身而不再是其地址了，笔者建议在一切可能的情况下都使用这种方式。

4.7 DLL 导出类
DLL 中定义的类可以在应用工程中使用。

下面的例子里，我们在 DLL 中定义了 point 和 circle 两个类，并在应用工程中引用了它们（单击此处下载本工程附件）。

```
//文件名：point.h，point 类的声明
```

```
#ifndef POINT_H

#define POINT_H

#ifdef DLL_FILE

class _declspec(dllexport) point //导出类 point

#else

class _declspec(dllimport) point //导入类 point

#endif

{

public:

float y;

float x;

point();

point(float x_coordinate, float y_coordinate);

};

#endif


//文件名：point.cpp，point 类的实现

#ifndef DLL_FILE

#define DLL_FILE

#endif

#include "point.h"

//类 point 的缺省构造函数

point::point()

{

x = 0.0;
```

```
y = 0.0;
}
//类 point 的构造函数
point::point(float x_coordinate, float y_coordinate)
{
x = x_coordinate;
y = y_coordinate;
}

//文件名：circle.h，circle 类的声明
#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h"
#ifdef DLL_FILE
class _declspec(dllexport)circle //导出类 circle
#else
class _declspec(dllimport)circle //导入类 circle
#endif
{
public:
void SetCentre(const point ¢rePoint);
void SetRadius(float r);
float GetGirth();
float GetArea();
circle();
```

```
private:
float radius;
point centre;
};
#endif

//文件名：circle.cpp，circle 类的实现
#ifndef DLL_FILE
#define DLL_FILE
#endif
#include "circle.h"
#define PI 3.1415926
//circle 类的构造函数
circle::circle()
{
centre = point(0, 0);
radius = 0;
}
//得到圆的面积
float circle::GetArea()
{
return PI *radius * radius;
}
//得到圆的周长
float circle::GetGirth()
```

```
{

return 2 *PI * radius;

}

//设置圆心坐标

void circle::SetCentre(const point ¢rePoint)

{

centre = centrePoint;

}

//设置圆的半径

void circle::SetRadius(float r)

{

radius = r;

}
```

类的引用：

```
#include "..\circle.h"       //包含类声明头文件

#pragma comment(lib,"dllTest.lib");


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

{

circle c;

point p(2.0, 2.0);

c.SetCentre(p);

c.SetRadius(1.0);

printf("area:%f girth:%f", c.GetArea(), c.GetGirth());


return 0;
```

}

从上述源代码可以看出，由于在 DLL 的类实现代码中定义了宏 DLL_FILE，故在 DLL 的实现中所包含的类声明实际上为：

class _declspec(dllexport) point //导出类 point

{

…

}

和

class _declspec(dllexport) circle //导出类 circle

{

…

}

而在应用工程中没有定义 DLL_FILE，故其包含 point.h 和 circle.h 后引入的类声明为：

class _declspec(dllimport) point //导入类 point

{

…

}

和

class _declspec(dllimport) circle //导入类 circle

{

…

}

不错，正是通过 DLL 中的

class _declspec(dllexport) class_name //导出类 circle

{

…

}

与应用程序中的

class _declspec(dllimport) class_name //导入类

{

…

}

匹对来完成类的导出和导入的！

我们往往通过在类的声明头文件中用一个宏来决定使其编译为 class _declspec(dllexport) class_name 还是 class _declspec(dllimport) class_name 版本，这样就不再需要两个头文件。本程序中使用的是：

#ifdef DLL_FILE

class _declspec(dllexport) class_name //导出类

#else

class _declspec(dllimport) class_name //导入类

#endif

实际上，在 MFC DLL 的讲解中，您将看到比这更简便的方法，而此处仅仅是为了说明_declspec(dllexport)与_declspec(dllimport)匹对的问题。

由此可见，应用工程中几乎可以看到 DLL 中的一切，包括函数、变量以及类，这就是 DLL 所要提供的强大能力。只要 DLL 释放这些接口，应用程序使用它就将如同使用本工程中的程序一样！

本章虽以 VC++为平台讲解非 MFC DLL，但是这些普遍的概念在其它语言及开发环境中也是相同的，

比较大的应用程序都由很多模块组成，这些模块分别完成相对独立的功能，它们彼此协作来完成整个软件系统的工作。可能存在一些模块的功能较为通用，在构造其它软件系统时仍会被使用。在构造软件系统时，如果将所有模块的源代码都静态编译到整个应用程序 EXE 文件中，会产生一些问题：一个缺点是增加了应用程序的大小，它会占用更多的磁盘空间，程序运行时也会消耗较大的内存空间，造成系统资源的浪费；另一个缺点是，在编写大的 EXE 程序时，在每次修改重建时都必须调整编译所有源代码，增加了编译过程的复杂性，也不利于阶段性的单元测试。

Windows 系统平台上提供了一种完全不同的较有效的编程和运行环境，你可以将独立的程序模块创建为较小的 DLL(Dynamic Linkable Library)文件，并可对它们单独编译和测试。在运行时，只有当 EXE 程序确实要调用这些 DLL 模块的情况下，系统才会将它们装载到内存空间中。这种方式不仅减少了 EXE 文件的大小和对内存空间的需求，而且使这些 DLL 模块可以同时被多个应用程序使用。Windows 自己就将一些主要的系统功能以 DLL 模块的形式实现。

一般来说，DLL 是一种磁盘文件，以.DLL、.DRV、.FON、.SYS 和许多以.EXE 为扩展名的系统文件都可以是 DLL。它由全局数据、服务函数和资源组成，在运行时被系统加载到进程的虚拟空间中，成为调用进程的一部分。如果与其它 DLL 之间没有冲突，该文件通常映射到进程虚拟空间的同一地址上。DLL 模块中包含各种导出函数，用于向外界提供服务。DLL 可以有自己的数据段，但没有自己的堆栈，使用与调用它的应用程序相同的堆栈模式；一个 DLL 在内存中只有一个实例；DLL 实现了代码封装性；DLL 的编制与具体的编程语言及编译器无关。

在 Win32 环境中，每个进程都复制了自己的读/写全局变量。如果想要与其它进程共享内存，必须使用内存映射文件或者声明一个共享数据段。DLL 模块需要的堆栈内存都是从运行进程的堆栈中分配出来的。Windows 在加载 DLL 模块时将进程函数调用与 DLL 文件的导出函数相匹配。Windows 操作系统对 DLL 的操作

仅仅是把 DLL 映射到需要它的进程的虚拟地址空间里去。DLL 函数中的代码所创建的任何对象（包括变量）都归调用它的线程或进程所有.

一、关于调用方式：

1、静态调用方式：由编译系统完成对 DLL 的加载和应用程序结束时 DLL 卸载的编码（如还有其它程序使用该 DLL，则 Windows 对 DLL 的应用记录减 1，直到所有相关程序都结束对该 DLL 的使用时才释放它），简单实用，但不够灵活，只能满足一般要求。

隐式的调用：

需要把产生动态连接库时产生的.LIB 文件加入到应用程序的工程中，想使用 DLL 中的函数时，只须说明一下。隐式调用不需要调用 LoadLibrary()和 FreeLibrary()。程序员在建立一个 DLL 文件时，链接程序会自动生成一个与之对应的 LIB 导入文件。该文件包含了每一个 DLL 导出函数的符号名和可选的标识号，但是并不含有实际的代码。LIB 文件作为 DLL 的替代文件被编译到应用程序项目中。当程序员通过静态链接方式编译生成应用程序时，应用程序中的调用函数与 LIB 文件中导出符号相匹配，这些符号或标识号进入到生成的 EXE 文件中。LIB 文件中也包含了对应的 DLL 文件名（但不是完全的路径名），链接程序将其存储在 EXE 文件内部。当应用程序运行过程中需要加载 DLL 文件时，Windows 根据这些信息发现并加载 DLL，然后通过符号名或标识号实现对 DLL 函数的动态链接。所有被应用程序调用的 DLL 文件都会在应用程序 EXE 文件加载时被加载在到内存中。可执行程序链接到一个包含 DLL 输出函数信息的输入库文件(.LIB 文件)。操作系统在加载使用可执行程序时加载 DLL。可执行程序直接通过函数名调用 DLL 的输出函数，调用方法和程序内部其他的函数是一样的。

2、动态调用方式：

是由编程者用 API 函数加载和卸载 DLL 来达到调用 DLL 的目的，使用上较复杂，但能更加有效地使用内存，是编制大型应用程序时的重要方式。
显式的调用：

是指在应用程序中用 LoadLibrary 或 MFC 提供的 AfxLoadLibrary 显式的将自己所做的动态连接库调进来，动态连接库的文件名即是上面两个函数的参数，再用 GetProcAddress()获取想要引入的函数。自此，你就可以象使用如同本应用程序自定义的函数一样来调用此引入函数了。在应用程序退出之前，应该用 FreeLibrary 或 MFC 提供的 AfxFreeLibrary 释放动态连接库。直接调用 Win32 的 LoadLibary 函数，并指定 DLL 的路径作为参数。LoadLibary 返回 HINSTANCE 参数，应用程序在调用 GetProcAddress 函数时使用这一参数。GetProcAddress 函数将符号名或标识号转换为 DLL 内部的地址。程序员可以决定 DLL 文件何时加载或不加载，显式链接在运行时决定加载哪个 DLL 文件。使用 DLL 的程序在使用之前必须加载（LoadLibrary）加载 DLL 从而得到一个 DLL 模块的句柄，然后调用 GetProcAddress 函数得到输出函数的指针，在退出之前必须卸载 DLL(FreeLibrary)。

Windows 将遵循下面的搜索顺序来定位 DLL：
1. 包含 EXE 文件的目录，
2. 进程的当前工作目录，
3. Windows 系统目录，
4. Windows 目录，
5. 列在 Path 环境变量中的一系列目录。

二、MFC 中的 dll：

a、Non-MFC DLL:指的是不用 MFC 的类库结构，直接用 C 语言写的 DLL，其输出的函数一般用的是标准 C 接口，并能被非 MFC 或 MFC 编写的应用程序所调用。

b、Regular DLL:和下述的 Extension Dlls 一样，是用 MFC 类库编写的。明显的特点是在源文件里有一个继承 CWinApp 的类。其又可细分成静态连接到 MFC 和动态连接到 MFC 上的。
静态连接到 MFC 的动态连接库只被 VC 的专业般和企业版所支持。该类 DLL 应用程序里头的输出函数可以被任意 Win32 程序使用，包括使用 MFC 的应用程序。输入函数有如下形式：
extern "C" EXPORT YourExportedFunction( );

如果没有 extern “C”修饰，输出函数仅仅能从 C++代码中调用。
DLL 应用程序从 CWinApp 派生，但没有消息循环。
动态链接到 MFC 的规则 DLL 应用程序里头的输出函数可以被任意 Win32 程序使用,包括使用 MFC 的应用程序。但是，所有从 DLL 输出的函数应该以如下语句开始：
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
此语句用来正确地切换 MFC 模块状态。
Regular DLL 能够被所有支持 DLL 技术的语言所编写的应用程序所调用。在这种动态连接库中，它必须有一个从 CWinApp 继承下来的类，DllMain 函数被 MFC 所提供，不用自己显式的写出来。

c、Extension DLL:用来实现从 MFC 所继承下来的类的重新利用，也就是说，用这种类型的动态连接库，可以用来输出一个从 MFC 所继承下来的类。它输出的函数仅可以被使用 MFC 且动态链接到 MFC 的应用程序使用。可以从 MFC 继承你所想要的、更适于你自己用的类，并把它提供给你的应用程序。你也可随意的给你的应用程序提供 MFC 或 MFC 继承类的对象指针。Extension DLL 使用 MFC 的动态连接版本所创建的，并且它只被用 MFC 类库所编写的应用程序所调用。Extension DLLs 和 Regular DLLs 不一样，它没有一个从 CWinApp 继承而来的类的对象，所以，你必须为自己 DllMain 函数添加初始化代码和结束代码。
和规则 DLL 相比，有以下不同：
1、它没有一个从 CWinApp 派生的对象；
2、它必须有一个 DllMain 函数；
3、DllMain 调用 AfxInitExtensionModule 函数，必须检查该函数的返回值，如果返回 0，DllMmain 也返回 0；
4、如果它希望输出 CRuntimeClass 类型的对象或者资源(Resources)，则需要提供一个初始化函数来创建一个 CDynLinkLibrary 对象。并且，有必要把初始化函数输出；
5、使用扩展 DLL 的 MFC 应用程序必须有一个从 CWinApp 派生的类，而且，一般在 InitInstance 里调用扩展 DLL 的初始化函数。

三、dll 入口函数：

1、每一个 DLL 必须有一个入口点，DllMain 是一个缺省的入口函数。DllMain 负责初始化(Initialization)和结束(Termination)工作，每当一个新的进程或者该进程的新的线程访问 DLL 时，或者访问 DLL 的每一个进程或者线程不再使用 DLL 或者结束时,都会调用 DllMain。但是,使用 TerminateProcess 或 TerminateThread 结束进程或者线程，不会调用 DllMain。
DllMain 的函数原型：

```
BOOL APIENTRY DllMain(HANDLE hModule,DWORD ul_reason_for_call,LPVOID lpReserved)
{
switch(ul_reason_for_call)
{
case DLL_PROCESS_ATTACH:
.......
case DLL_THREAD_ATTACH:
.......
```

```
case DLL_THREAD_DETACH:
.......
case DLL_PROCESS_DETACH:
.......
return TRUE;
}
}
```

参数：

hMoudle：是动态库被调用时所传递来的一个指向自己的句柄(实际上，它是指向_DGROUP 段的一个选择符)；

ul_reason_for_call：是一个说明动态库被调原因的标志。当进程或线程装入或卸载动态连接库的时候，操作系统调用入口函数，并说明动态连接库被调用的原因。它所有的可能值为：

DLL_PROCESS_ATTACH: 进程被调用；

DLL_THREAD_ATTACH: 线程被调用；

DLL_PROCESS_DETACH: 进程被停止；

DLL_THREAD_DETACH: 线程被停止；

lpReserved：是一个被系统所保留的参数。

2、_DllMainCRTStartup

为了使用“C”运行库(CRT，C Run time Library)的 DLL 版本（多线程），一个 DLL 应用程序必须指定 _DllMainCRTStartup 为入口函数，DLL 的初始化函数必须是 DllMain。

_DllMainCRTStartup 完成以下任务：当进程或线程捆绑(Attach)到 DLL 时为“C”运行时的数据(C Runtime Data)分配空间和初始化并且构造全局“C++”对象，当进程或者线程终止使用 DLL(Detach)时，清理 C Runtime Data 并且销毁全局“C++”对象。它还调用 DllMain 和 RawDllMain 函数。

RawDllMain 在 DLL 应用程序动态链接到 MFC DLL 时被需要，但它是静态的链接到 DLL 应用程序的。在讲述状态管理时解释其原因。

谢谢大家参阅！