

什么是动态链接库？DLL 三个字母对于你来说一定很熟悉吧，它是 Dynamic Link Library 的缩写形式，动态链接库 (DLL) 是作为共享函数库的可执行文件。动态链接提供了一种方法，使进程可以调用不属于其可执行代码的函数。函数的可执行代码位于一个 DLL 中，该 DLL 包含一个或多个已被编译、链接并与使用它们的进程分开存储的函数。DLL 还有助于共享数据和资源。多个应用程序可同时访问内存中单个 DLL 副本的内容。

本文比较了Visual C++所支持的三种动态链接库，列出了各自不同的特点和应用场合，详细地描述了三种动态链接库的建立和调用的方法。

关键字：动态链接库；导出函数；调用

引言

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。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的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；而且，一些模块的功能可能较为通用，在构造其它软件系统时仍会被使用。

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动态链接库(Dynamic Linkable Library, DLL)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。

动态链接库概述

动态链接库技术是Windows最重要的实现技术之一，Windows的许多新功能、新特性都是通过DLL来实现的。其实，Windows本身就是由许多DLL组成的，它最基本的三大组成模块Kernel、GDI和User都是DLL。

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，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。DLL可以有自己的数据段，但没有自己的堆栈，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，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。

动态链接库的分类

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

1、Non-MFC DLL（非MFC动态库）

这种动态链接库指的是不用MFC的类库结构，直接用C语言写的DLL，其导出的函数是标准的C接口，能被非MFC或MFC编写的应用程序所调用。如果建立的DLL不需要使用MFC，那么应该建立Non-MFC DLL，因为使用MFC会增大用户库的大小，从而浪费用户的磁盘和内存空间。

2、Regular DLL（常规DLL）

这种动态链接库和下述的Extension Dll一样，是用MFC类库编写的，它的一个明显的特点是在源文件里有一个继承CWinApp的类（注意：此类DLL虽然从CWinApp派生，但没有消息循环），被导出的函数是C函数、C++类或者C++成员函数（注意不要把术语C++类与MFC的微软基础C++类相混淆），调用常规DLL的应用程序不必是MFC应用程序，只要是能调用类C函数的应用程序就可以，它们可以是在Visual C++、Delphi、Visual Basic、Borland C等编译环境下利用DLL开发应用程序。常规DLL又可细分成静态链接到MFC和动态链接到MFC两种：

（1）静态链接到MFC的动态链接库只被VC的专业版和企业版所支持。该类DLL里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。输出函数有如下形式：

```
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。

（2）动态链接到 MFC 的常规 DLL 里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。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。

3、Extension Dll（扩展 DLL）

这种动态链接库是使用 MFC 的动态链接版本所创建的，并且它只被用 MFC 类库所编写的应用程序所调用。例如你已经创建了一个从 MFC 的 CToolBar 类的

派生类用于创建一个新的工具栏，为了导出这个类，你必须把它放到一个 MFC 扩展的 DLL 中。扩展 DLL 和常规 DLL 不一样，它没有一个从 CWinApp 继承而来的类的对象，所以，开发人员必须在 DLL 中的 DllMain 函数添加初始化代码和结束代码。与常规 DLL 相比，扩展的 DLL 有如下不同点：

- 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，DLLMain 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。

动态连接库的建立

1、Non-MFC DLL 的建立

每一个 DLL 必须有一个入口点，就象用 C 编写的应用程序时，必须有一个 WINMAIN 函数一样。在 Non-MFC DLL 中 DllMain 是一个缺省的入口函数，你不需要编写自己的 DLL 入口函数，用这个缺省的入口函数就能使动态链接库被调用时得到正确的初始化。如果应用程序的 DLL 需要分配额外的内存或资源，或者说需要对每个进程或线程初始化和清除操作时，需要在相应的 DLL 工程的 .CPP 文件中对 DllMain() 函数按照下面的格式书写。

```

BOOL WINAPI 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;
}

```

参数中，hModule 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链接库的时候，操作系统调用入口函数，并说明动态链接库被调用的原因，它所有的可能值为：

(1)DLL_PROCESS_ATTACH: 进程被调用或调用 Load Library，DLL 被链接到当前进程的地址空间并被初始化；

(2)DLL_THREAD_ATTACH: 当前进程创建一个新线程，DLL 在新线程正文内被调用；

(3)DLL_PROCESS_DETACH: 调用 DLL 的进程被终止, DLL 被卸载;

(4)DLL_THREAD_DETACH: 调用 DLL 的线程被终止, DLL 被卸载;

lpReserved 为保留参数。

如果在 DLL 中加入想要输出的函数、变量、C++类或其它函数, 可以调用 VC 的关键字 `_declspec(dllexport)`。

2、MFC AppWizard[dll]方式下 Regular DLL 和 Extension DLL 的建立

在 MFC AppWizard[dll]下生成的 DLL 文件有三种方式: 静态链接到 MFC 的常规 DLL、动态链接到 MFC 的常规 DLL 以及 MFC 扩展 DLL, 在创建 DLL 是, 要根据实际情况选择创建 DLL 的方式。

静态链接到 MFC 的常规 DLL 和静态连接到 MFC 常规 DLL 的区别是: 前者使用的是 MFC 的静态链接库, 生成的 DLL 文件长度大, 一般不使用这种方式; 后者使用 MFC 的动态链接库, 生成的 DLL 文件长度小; 动态链接到 MFC 的常规 DLL 所有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
```

MFC 扩展 DLL 的特点是用来建立 MFC 的派生类, Dll 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, 编译器默认了一个 DLL 入口函数 `DLLMain()` 作为对 DLL 的初始化, 你可以在此函数中实现初始化, 代码如下:

```
BOOL WINAPI APIENTRY DLLMain(HINSTANCE hinstDll, DWORD reason ,
```

```

LPVOID flmpload)
{
switch(reason)
{
.....//初始化代码;
}
return true;
}

```

参数hinstDll存放DLL的句柄, 参数reason指明调用函数的原因, lpReserved是一个被系统所保留的参数。对于隐式链接是一个非零值, 对于显式链接值是零。

动态连接库的调用

动态连接库的调用可以分为两种：一种是隐式调用，一种是显示调用。

1、隐式的调用

这种调用方式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, 在使用DLL中的函数时, 只须说明一下后就可以直接通过函数名调用DLL的输出。 饕梅椒ê统绦蚰诌科渌 暮 且谎 摹 R 降饕貌恍枰 饕胞oad Library() 和Free Library()。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。

当程序员通过隐式调用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, 这些符号或标识号被写入到生成的EXE文件中。LIB文件也包含了对应的DLL文件名(但不是完全的路径名), 链接程序也将其存

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文件加载时被加载到内存中。

2、显式调用

这种调用方式是指在应用程序中用Load Library或MFC提供的AfxLoadLibrary显式的将自己所做的动态连接库调进来，并指定DLL的路径作为参数。LoadLibrary返回HINSTANCE参数，应用程序在调用GetProcAddress函数时使用这一参数。当完成对动态链接库的导入以后，再使用GetProcAddress()获取想要引入的函数，该函数将符号名或标识号转换为DLL内部的地址，之后就可以象使用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Free Library或MFC提供的AfxFreeLibrary释放动态连接库。

使用显式调用方式可以让程序员来决定DLL文件何时加载或不加载，而操作系统在载入应用程序时不必要将所有该应用程序所引用的DLL都一起加载到内存中，只要在使用某个DLL时再将其载入，这样就可以减少应用程序在初始加载时所使用的时间和对内存的消耗。在对DLL加载的过程中，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 ；
- ③Windows系统目录 ；
- ④Windows目录 ；
- ⑤列在Path环境变量中的一系列目录。

总结

在Windows操作系统中使用动态链接库（DLL）有很多优点，最主要的一点是多个应用程序、甚至是不同语言编写的应用程序可以共享一个DLL文件，真正实现了资源“共享”，大大缩小了应用程序的执行代码，更加有效地利用了内存；使用DLL的另一个优点是DLL文件作为一个单独的程序模块，封装性、独立性好，在软件需要升级的时候，开发人员只需要修改相应的DLL文件就可以了，而且，当DLL中的函数改变后，如果没有修改参数，程序代码并不需要重新编译。这在编程时十分有用，大大提高了软件开发和维护的效率。

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。

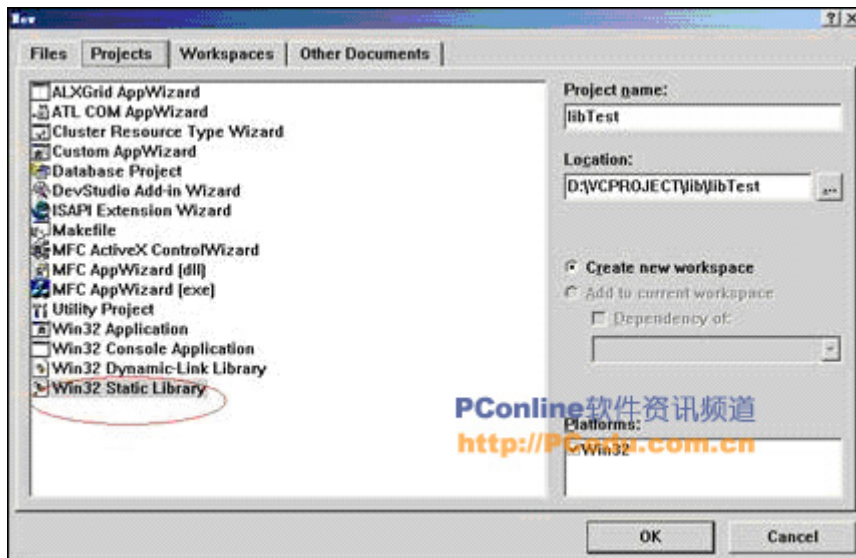


图 1 建立一个静态链接库

如图 1，在 VC++6.0 中新建一个名称为 libTest 的 static library 工程（单击此处下载本工程附件），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。

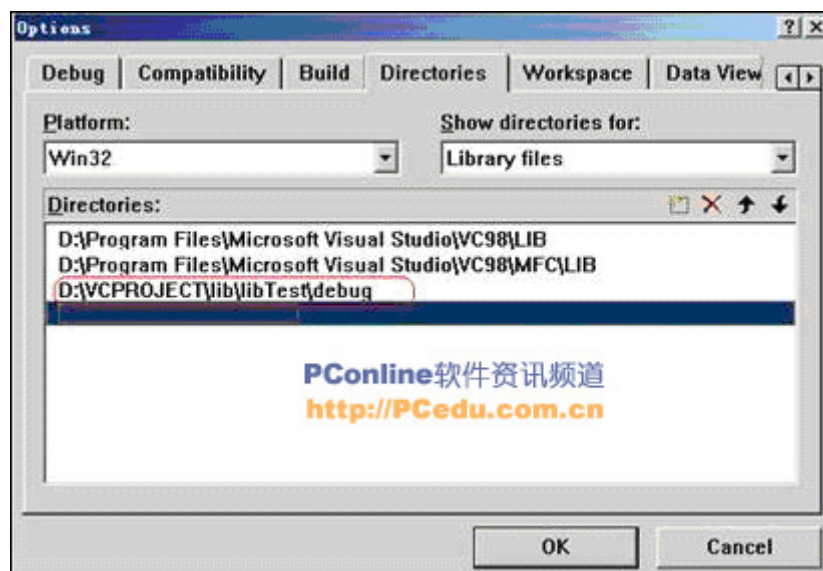


图 2 在 VC 中设置库文件路径

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。

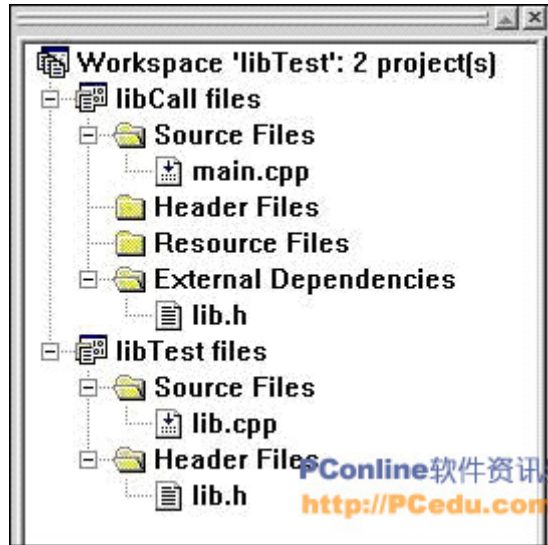


图 4 把库工程和调用库的工程放入同一工作区进行调试

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！

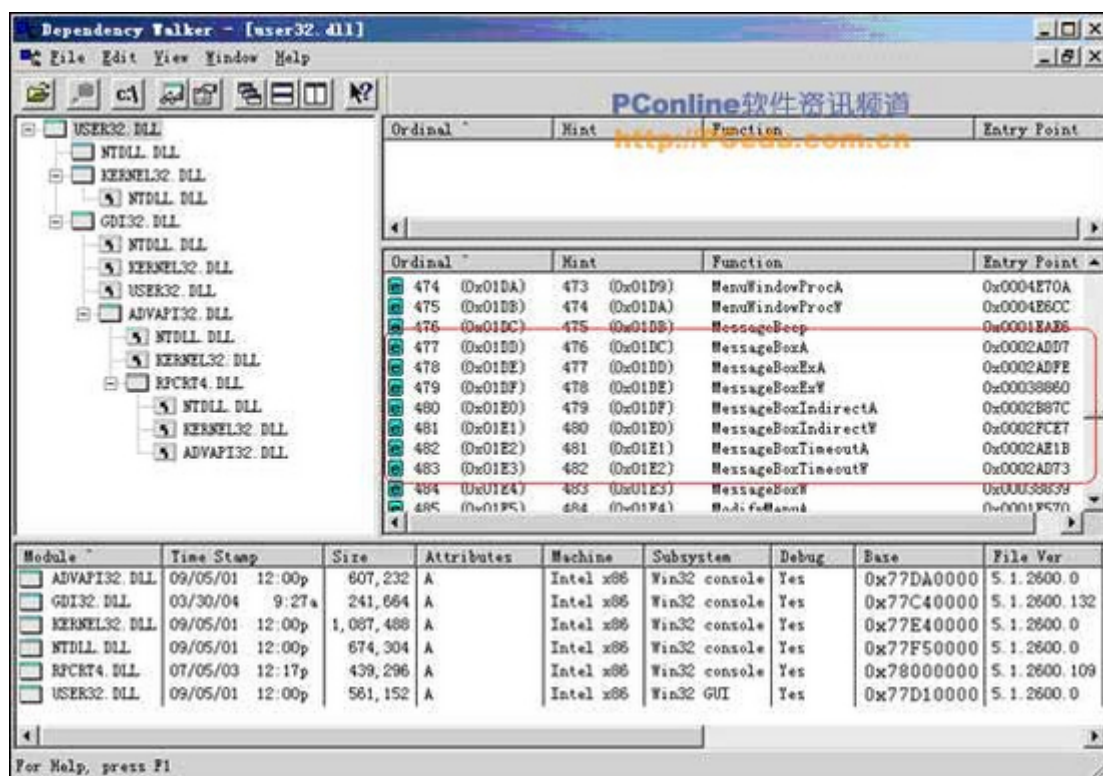


图5 用 Depends 查看 DLL

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。

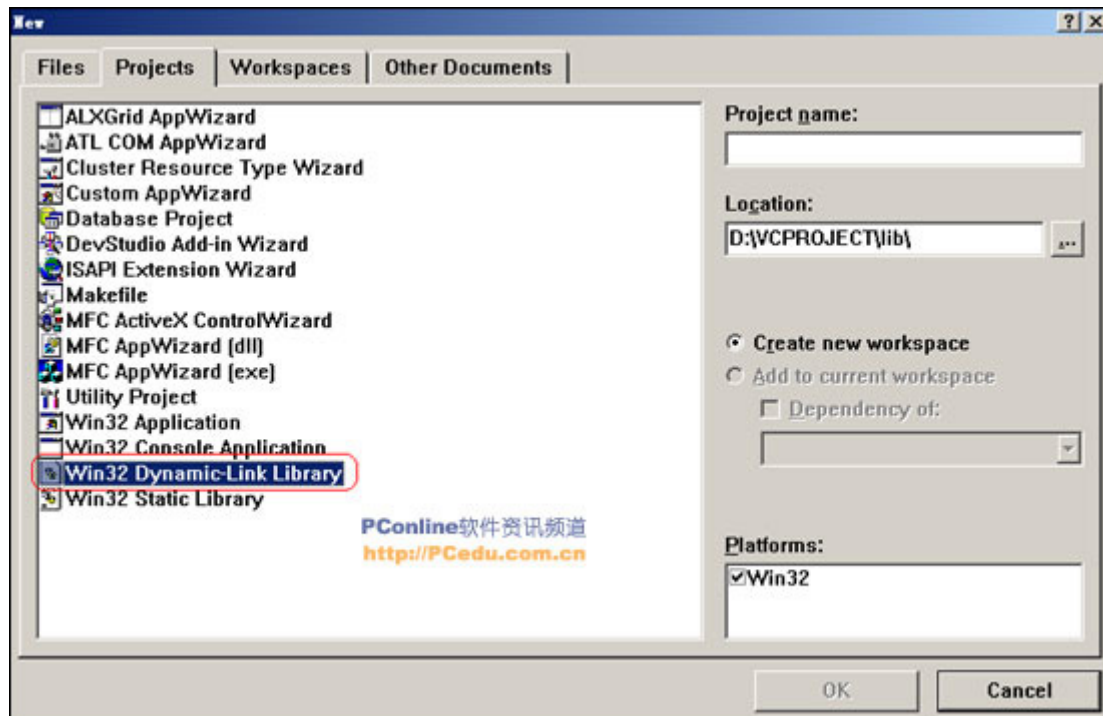


图 6 建立一个非 MFC DLL

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); /*+】

{

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。

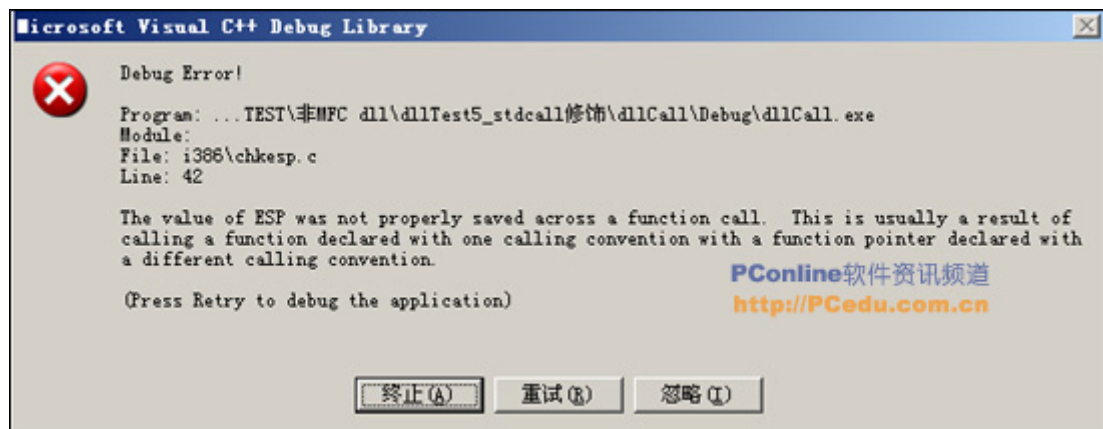


图 7 调用约定不匹配时的运行错误

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(dllexport) class_name //导入类

{

...

}
```

配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)` 配对的问题。

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，其思维方式可以直接过渡。

接下来，我们将要**研究MFC规则DLL**第4节我们对非MFC DLL进行了介绍，这一节将详细地讲述MFC规则DLL的创建与使用技巧。

另外，自从本文开始连载后，收到了一些读者的e-mail。有的读者提出了一些问题，笔者将在本文的最后一次连载中选取其中的典型问题进行解答。由于时间的关系，对于读者朋友的来信，笔者暂时不能一一回复，还望海涵！由于笔者的水平有限，文中难免有错误和纰漏，也热诚欢迎读者朋友不吝指正！

5. MFC规则DLL

5.1 概述

MFC规则DLL的概念体现在两方面：

(1) 它是MFC的

“是MFC的”意味着可以在这种DLL的内部使用MFC；

(2) 它是规则的

“是规则的”意味着它不同于MFC扩展DLL，在MFC规则DLL的内部虽然可以使用MFC，但是其与应用程序的接口不能是MFC。而MFC扩展DLL与应用程序的接口可以是MFC，可以从MFC扩展DLL中导出一个MFC类的派生类。

Regular DLL能够被所有支持DLL技术的语言所编写的应用程序调用，当然也包括使用MFC的应用程序。在这种动态连接库中，包含一个从CWinApp继承下来的类，DllMain函数则由MFC自动提供。

Regular DLL分为两类:

(1) 静态链接到MFC 的规则DLL

静态链接到MFC的规则DLL与MFC库（包括MFC扩展 DLL）静态链接，将MFC库的代码直接生成在.dll文件中。在调用这种DLL的接口时，MFC使用DLL的资源。因此，在静态链接到MFC的规则DLL中不需要进行模块状态的切换。

使用这种方法生成的规则DLL其程序较大，也可能包含重复的代码。

(2) 动态链接到MFC 的规则DLL

动态链接到MFC 的规则DLL 可以和使用它的可执行文件同时动态链接到 MFC DLL 和任何MFC扩展 DLL。在使用了MFC共享库的时候，默认情况下，MFC使用主应用程序的资源句柄来加载资源模板。这样，当DLL和应用程序中存在相同ID的资源时（即所谓的资源重复问题），系统可能不能获得正确的资源。因此，对于共享MFC DLL的规则DLL，我们必须进行模块切换以使得MFC能够找到正确的资源模板。我们可以在Visual C++中设置MFC规则DLL是静态链接到MFC DLL还是动态链接到MFC DLL。如图 8，依次选择Visual C++的project -> Settings -> General菜单或选项，在Microsoft Foundation Classes中进行设置。

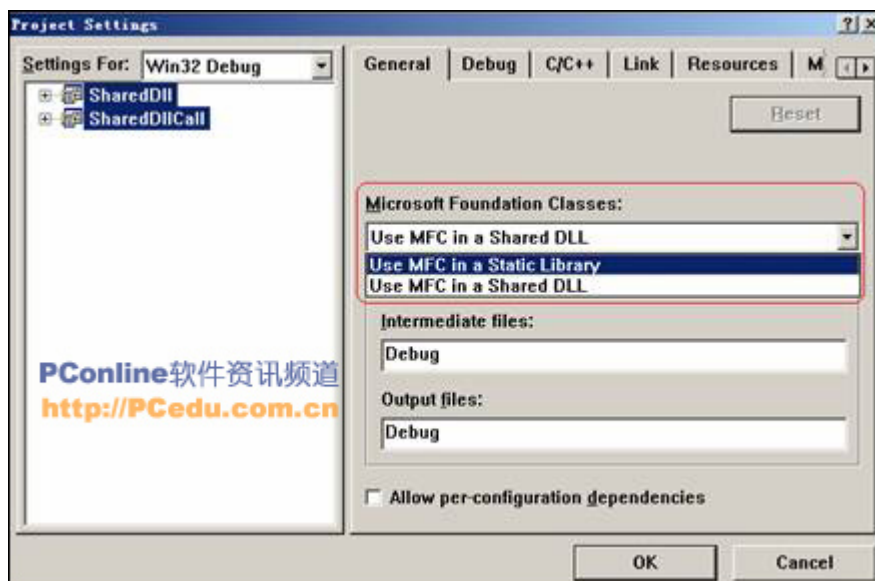


图 8 设置动态/静态链接 MFC DLL

5.2 MFC 规则 DLL 的创建

我们来一步步讲述使用 MFC 向导创建 MFC 规则 DLL 的过程，首先新建一个 project，如图 9，选择 project 的类型为 MFC AppWizard(dll)。点击 OK 进入如图 10 所示的对话框。

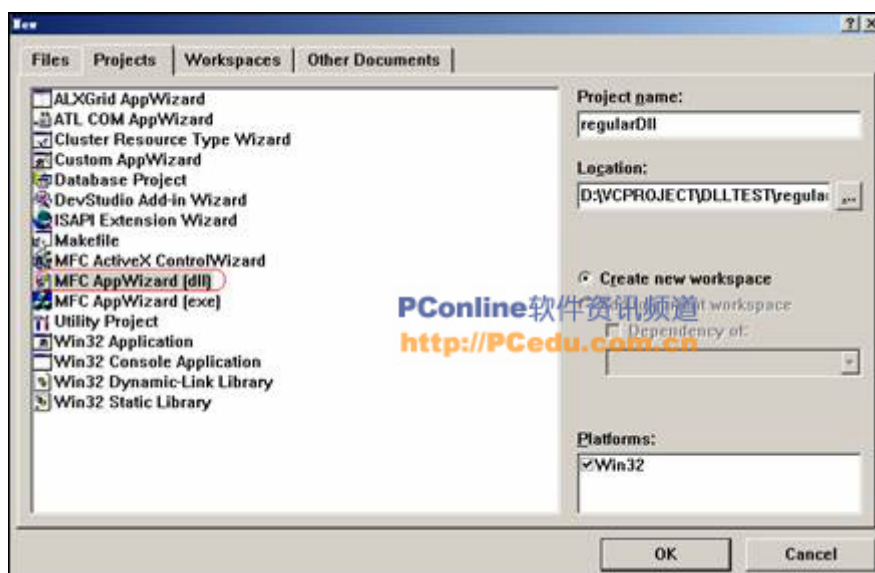


图 9 MFC DLL 工程的创建

图 10 所示对话框中的 1 区选择 MFC DLL 的类别。

2 区选择是否支持 automation (自动化) 技术, automation 允许用户在一个应用程序中操纵另外一个应用程序或组件。例如, 我们可以在应用程序中利用 Microsoft Word 或 Microsoft Excel 的工具, 而这种使用对用户而言是透明的。自动化技术可以大大简化和加快应用程序的开发。

3 区选择是否支持 Windows Sockets, 当选择此项目时, 应用程序能在 TCP/IP 网络上进行通信。CWinApp 派生类的 InitInstance 成员函数会初始化通讯端的支持, 同时工程中的 StdAfx.h 文件会自动 include <AfxSock.h>头文件。

添加 socket 通讯支持后的 InitInstance 成员函数如下:

```
BOOL CRegularDllSocketApp::InitInstance()
{
    if (!AfxSocketInit())
    {
        AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
        return FALSE;
    }

    return TRUE;
}
```

4 区选择是否由 MFC 向导自动在源代码中添加注释, 一般我们选择 “Yes, please”。

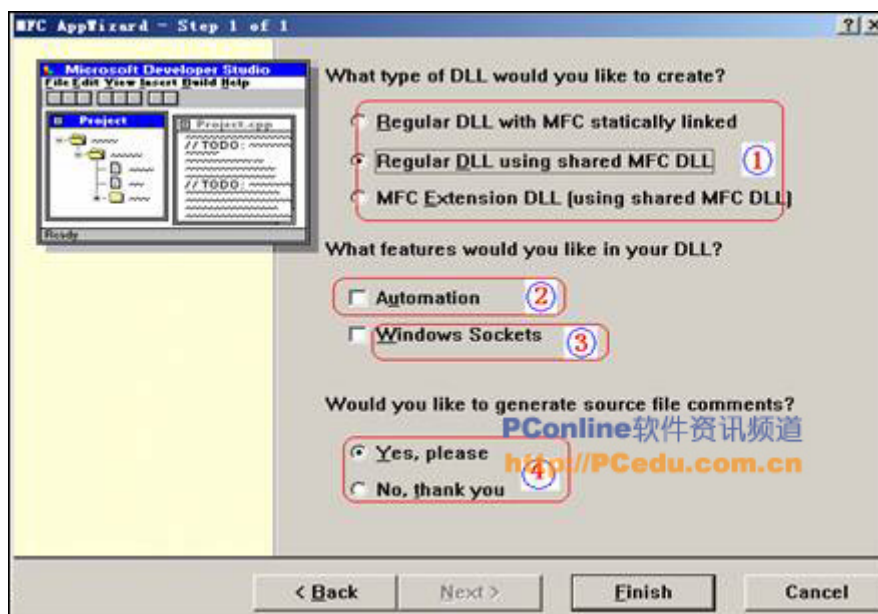


图 10 MFC DLL 的创建选项

5.3 一个简单的MFC规则DLL

这个DLL的例子（属于静态链接到MFC 的规则DLL）中提供了一个如图 11 所示的对话框。



图 11 MFC 规则 DLL 例子

在DLL中添加对话框的方式与在MFC应用程序中是一样的。

在图 11 所示DLL中的对话框的Hello按钮上点击时将MessageBox一个“Hello, pconline的网友”对话框，下面是相关的文件及源代码，其中删除了MFC向导自动生成的绝大多数注释（下载本工程附件）：

第一组文件：CWinApp 继承类的声明与实现

```
// RegularDll.h : main header file for the REGULARDLL DLL

#ifndef AFX_REGULARDLL_H__3E9CB22B_588B_4388_B778_B3416ADB79B3__INCLUDED_
#define AFX_REGULARDLL_H__3E9CB22B_588B_4388_B778_B3416ADB79B3__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h" // main symbols

class CRegularDllApp : public CWinApp
{
public:
    CRegularDllApp();

    DECLARE_MESSAGE_MAP()
};
```

```
#endif

// RegularDll.cpp : Defines the initialization routines for the DLL.

#include "stdafx.h"

#include "RegularDll.h"

#ifdef _DEBUG

#define new DEBUG_NEW

#undef THIS_FILE

static char THIS_FILE[] = __FILE__;

#endif

BEGIN_MESSAGE_MAP(CRegularDllApp, CWinApp)

END_MESSAGE_MAP()

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

// CRegularDllApp construction

CRegularDllApp::CRegularDllApp()

{

}
```

```

////////////////////////////////////
// The one and only CRegularDllApp object

CRegularDllApp theApp;

```

分析：

在这一组文件中定义了一个继承自 CWinApp 的类 CRegularDllApp，并同时定义了其的一个实例 theApp。乍一看，您会以为它是一个 MFC 应用程序，因为 MFC 应用程序也包含这样的在工程名后添加“App”组成类名的类（并继承自 CWinApp 类），也定义了这个类的一个全局实例 theApp。我们知道，在 MFC 应用程序中 CWinApp 取代了 SDK 程序中 WinMain 的地位，SDK 程序 WinMain 所完成的工作由 CWinApp 的三个函数完成：

```

virtual BOOL InitApplication( );

virtual BOOL InitInstance( );

virtual BOOL Run( ); //传说中 MFC 程序的“活水源头”

```

但是MFC规则DLL并不是MFC应用程序，它所继承自CWinApp的类不包含消息循环。这是因为，MFC规则DLL不包含CWinApp::Run机制，主消息泵仍然由应用程序拥有。如果DLL 生成无模式对话框或有自己的主框架窗口，则应用程序的主消息泵必须调用从DLL 导出的函数来调用PreTranslateMessage成员函数。

另外，MFC规则DLL与MFC 应用程序中一样，需要将所有 DLL中元素的初始化放到InitInstance 成员函数中。

第二组文件 自定义对话框类声明及实现(点击查看附件)

分析：

这一部分的编程与一般的应用程序根本没有什么不同，我们照样可以利用MFC类向导来自动为对话框上的控件添加事件。MFC类向导照样会生成类似ON_BN_CLICKED(IDC_HELLO_BUTTON, OnHelloButton)的消息映射宏。

第三组文件 DLL中的资源文件

```
//{{NO_DEPENDENCIES}}  
  
// Microsoft Developer Studio generated include file.  
  
// Used by RegularDll.rc  
  
//  
  
#define IDD_DLL_DIALOG 1000  
  
#define IDC_HELLO_BUTTON 1000
```

分析：

在MFC规则DLL中使用资源也与在MFC应用程序中使用资源没有什么不同，我们照样可以用Visual C++的资源编辑工具进行资源的添加、删除和属性的更改。

第四组文件 MFC 规则 DLL 接口函数

```
#include "StdAfx.h"  
  
#include "DllDialog.h"  
  
extern "C" __declspec(dllexport) void ShowDlg(void)  
  
{  
  
    CDllDialog dllDialog;  
  
    dllDialog.DoModal();  
}
```

```
}
```

分析：

这个接口并不使用MFC，但是在其中却可以调用MFC扩展类CdllDialog的函数，这体现了“规则”的概类。

与非MFC DLL完全相同，我们可以使用__declspec(dllexport)声明或在.def中引出的方式导出MFC规则DLL中的接口。

5.4 MFC规则DLL的调用

笔者编写了如图 12 的对话框MFC程序（下载本工程附件）来调用 5.3 节的MFC规则DLL，在这个程序的对话框上点击“调用DLL”按钮时弹出 5.3 节MFC规则DLL中的对话框。



图 12 MFC 规则 DLL 的调用例子

下面是“调用 DLL”按钮单击事件的消息处理函数：

```
void CRegularDllCallDlg::OnCalldllButton()
{
    typedef void (*lpFun)(void);

    HINSTANCE hDll; //DLL 句柄
```

```

hDll = LoadLibrary("RegularDll.dll");

if (NULL==hDll)

{

MessageBox("DLL 加载失败");

}


lpFun addFun; //函数指针

lpFun pShowDlg = (lpFun)GetProcAddress(hDll, "ShowDlg");

if (NULL==pShowDlg)

{

MessageBox("DLL 中函数寻找失败");

}

pShowDlg();

}

```

上述例子中给出的是显示调用的方式，可以看出，其调用方式与第 4 节中非 MFC DLL 的调用方式没有什么不同。我们照样可以在 EXE 程序中隐式调用 MFC 规则 DLL，只需要将 DLL 工程生成的 .lib 文件和 .dll 文件拷入当前工程所在的目录，并在 RegularDllCallDlg.cpp 文件（图 12 所示对话框类的实现文件）的顶部添加：

```

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

void ShowDlg(void);

```

并将 void CRegularDllCallDlg::OnCallDllButton() 改为：


```
void CRegularDllCallDlg::OnCallDllButton()
{
    ShowDlg();
}
```

5.5 共享MFC DLL的规则DLL的模块切换

应用程序进程本身及其调用的每个DLL模块都具有一个全局唯一的HINSTANCE句柄，它们代表了DLL或EXE模块在进程虚拟空间中的起始地址。进程本身的模块句柄一般为 0x400000，而DLL模块的缺省句柄为 0x10000000。如果程序同时加载了多个DLL，则每个DLL模块都会有不同的HINSTANCE。应用程序在加载DLL时对其进行了重定位。

共享MFC DLL（或MFC扩展DLL）的规则DLL涉及到HINSTANCE句柄问题，HINSTANCE句柄对于加载资源特别重要。EXE和DLL都有其自己的资源，而且这些资源的ID可能重复，应用程序需要通过资源模块的切换来找到正确的资源。如果应用程序需要来自于DLL的资源，就应将资源模块句柄指定为DLL的模块句柄；如果需要EXE文件中包含的资源，就应将资源模块句柄指定为EXE的模块句柄。

这次我们创建一个动态链接到MFC DLL的规则DLL（下载本工程附件），在其中包含如图13的对话框。



图 13 DLL 中的对话框

另外，在与这个 DLL 相同的工作区中生成一个基于对话框的 MFC 程序，其对话框与图 12 完全一样。但是在此工程中我们另外添加了一个如图 14 的对话框。



图 14 EXE 中的对话框

图 13 和图 14 中的对话框除了 caption 不同（以示区别）以外，其它的都相同。

尤其值得特别注意，在 DLL 和 EXE 中我们对图 13 和图 14 的对话框使用了相同的资源 ID=2000，在 DLL 和 EXE 工程的 resource.h 中分别有如下的宏：

```
//DLL 中对话框的 ID  
  
#define IDD_DLL_DIALOG 2000  
  
//EXE 中对话框的 ID
```

```
#define IDD_EXE_DIALOG 2000
```

与 5.3 节静态链接 MFC DLL 的规则 DLL 相同，我们还是在规则 DLL 中定义接口函数 ShowDlg，原型如下：

```
#include "StdAfx.h"

#include "SharedDll.h"

void ShowDlg(void)

{

CDialog dlg(IDD_DLL_DIALOG); //打开 ID 为 2000 的对话框

dlg.DoModal();

}
```

而为应用工程主对话框的“调用 DLL”的单击事件添加如下消息处理函数：

```
void CSharedDllCallDlg::OnCallDllButton()

{

ShowDlg();

}
```

我们以为单击“调用 DLL”会弹出如图 13 所示 DLL 中的对话框，可是可怕的事情发生了，我们看到是图 14 所示 EXE 中的对话框！

惊讶？

产生这个问题的根源在于应用程序与 MFC 规则 DLL 共享 MFC DLL（或 MFC 扩展 DLL）的程序总是默认使用 EXE 的资源，我们必须进行资源模块句柄的切换，其实现方法有三：

方法一 在 DLL 接口函数中使用：

```
AFX_MANAGE_STATE(AfxGetStaticModuleState());
```

我们将 DLL 中的接口函数 ShowDlg 改为：

```
void ShowDlg(void)

{

//方法 1:在函数开始处变更，在函数结束时恢复

//将 AFX_MANAGE_STATE(AfxGetStaticModuleState());作为接口函数的第一//条语句进行模块状态切换

AFX_MANAGE_STATE(AfxGetStaticModuleState());


CDialog dlg(IDD_DLL_DIALOG); //打开 ID 为 2000 的对话框

dlg.DoModal();

}
```

这次我们再点击 EXE 程序中的“调用 DLL”按钮，弹出的是 DLL 中的如图 13 的对话框！
嘿嘿，弹出了正确的对话框资源。

AfxGetStaticModuleState 是一个函数，其原型为：

```
AFX_MODULE_STATE* AFXAPI AfxGetStaticModuleState( );
```

该函数的功能是在栈上（这意味着其作用域是局部的）创建一个 AFX_MODULE_STATE 类（模块全局数据也就是模块状态）的实例，对其进行设置，并将其指针 pModuleState 返回。

AFX_MODULE_STATE 类的原型如下：

```

// AFX_MODULE_STATE (global data for a module)

class AFX_MODULE_STATE : public CNoTrackObject
{
public:

#ifdef _AFXDLL

AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfnAfxWndProc, DWORD dwVersion);

AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfnAfxWndProc, DWORD dwVersion, BOOL bSystem);

#else

AFX_MODULE_STATE(BOOL bDLL);

#endif

~AFX_MODULE_STATE();

CWinApp* m_pCurrentWinApp;

HINSTANCE m_hCurrentInstanceHandle;

HINSTANCE m_hCurrentResourceHandle;

LPCTSTR m_lpszCurrentAppName;

... //省略后面的部分

}

```

AFX_MODULE_STATE 类利用其构造函数和析构函数进行存储模块状态现场及恢复现场的工作，类似汇编中 call 指令对 pc 指针和 sp 寄存器的保存与恢复、中断服务程序的中断现场压栈与恢复以及操作系统线程调度的任务控制块保存与恢复。

许多看似不着边际的知识点居然有惊人的相似！AFX_MANAGE_STATE 是一个宏，其原型

为：

```
AFX_MANAGE_STATE( AFX_MODULE_STATE* pModuleState )
```

该宏用于将 pModuleState 设置为当前的有效模块状态。当离开该宏的作用域时（也就离开了 pModuleState 所指向栈上对象的作用域），先前的模块状态将由 AFX_MODULE_STATE 的析构函数恢复。

方法二 在 DLL 接口函数中使用：

```
AfxGetResourceHandle();  
  
AfxSetResourceHandle(HINSTANCE xxx);
```

AfxGetResourceHandle 用于获取当前资源模块句柄，而 AfxSetResourceHandle 则用于设置程序目前要使用的资源模块句柄。

我们将 DLL 中的接口函数 ShowDlg 改为：

```
void ShowDlg(void)  
{  
  
    //方法 2 的状态变更  
  
    HINSTANCE save_hInstance = AfxGetResourceHandle();  
  
    AfxSetResourceHandle(theApp.m_hInstance);  
  
  
    CDialog dlg(IDD_DLL_DIALOG); //打开 ID 为 2000 的对话框  
  
    dlg.DoModal();  
  
    //方法 2 的状态还原  
  
    AfxSetResourceHandle(save_hInstance);  
}
```

```
}
```

通过 `AfxGetResourceHandle` 和 `AfxSetResourceHandle` 的合理变更,我们能够灵活地设置程序的资源模块句柄,而方法一则只能在 DLL 接口函数退出的时候才会恢复模块句柄。方法二则不同,如果将 `ShowDlg` 改为:

```
extern CSharedDllApp theApp; //需要声明 theApp 外部全局变量

void ShowDlg(void)

{

//方法 2 的状态变更

HINSTANCE save_hInstance = AfxGetResourceHandle();

AfxSetResourceHandle(theApp.m_hInstance);

CDialog dlg(IDD_DLL_DIALOG); //打开 ID 为 2000 的对话框

dlg.DoModal();

//方法 2 的状态还原

AfxSetResourceHandle(save_hInstance);

//使用方法 2 后在此处再进行操作针对的将是应用程序的资源

CDialog dlg1(IDD_DLL_DIALOG); //打开 ID 为 2000 的对话框

dlg1.DoModal();

}
```

在应用程序主对话框的“调用DLL”按钮上点击,将看到两个对话框,相继为DLL中的对话框(图 13)和EXE中的对话框(图 14)。方法三 由应用程序自身切换

资源模块的切换除了可以由DLL接口函数完成以外，由应用程序自身也能完成（下载本工程附件）。

现在我们把DLL中的接口函数改为最简单的：

```
void ShowDlg(void)

{

    CDialog dlg(IDD_DLL_DIALOG); //打开 ID 为 2000 的对话框

    dlg.DoModal();

}
```

而将应用程序的 OnCallDllButton 函数改为：

```
void CSharedDllCallDlg::OnCallDllButton()

{

    //方法 3：由应用程序本身进行状态切换

    //获取 EXE 模块句柄

    HINSTANCE exe_hInstance = GetModuleHandle(NULL);

    //或者 HINSTANCE exe_hInstance = AfxGetResourceHandle();

    //获取 DLL 模块句柄

    HINSTANCE dll_hInstance = GetModuleHandle("SharedDll.dll");

    AfxSetResourceHandle(dll_hInstance); //切换状态

    ShowDlg(); //此时显示的是 DLL 的对话框
```



```
AfxSetResourceHandle(exe_hInstance); //恢复状态

//资源模块恢复后再调用 ShowDlg

ShowDlg(); //此时显示的是 EXE 的对话框

}
```

方法三中的 Win32 函数 `GetModuleHandle` 可以根据 DLL 的文件名获取 DLL 的模块句柄。

如果需要得到 EXE 模块的句柄，则应调用带有 `Null` 参数的 `GetModuleHandle`。

方法三与方法二的不同在于方法三是在应用程序中利用 `AfxGetResourceHandle` 和 `AfxSetResourceHandle` 进行资源模块句柄切换的。同样地，在应用程序主对话框的“调用 DLL”按钮上点击，也将看到两个对话框，相继为 DLL 中的对话框（图 13）和 EXE 中的对话框（图 14）。

在下一节我们将对 MFC 扩展 DLL 进行详细分析和实例讲解，欢迎您继续关注本系列连载。MFC 扩展 DLL 的内涵为 MFC 的扩展，用户使用 MFC 扩展 DLL 就像使用 MFC 本身的 DLL 一样。除了可以在 MFC 扩展 DLL 的内部使用 MFC 以外，MFC 扩展 DLL 与应用程序的接口部分也可以是 MFC。我们一般使用 MFC 扩展 DLL 来包含一些 MFC 的增强功能，譬如扩展 MFC 的 `CStatic`、`CButton` 等类使之具备更强大的能力。

使用 Visual C++ 向导生产 MFC 扩展 DLL 时，MFC 向导会自动增加 DLL 的入口函数 `DllMain`：

```
extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    // Remove this if you use lpReserved
    UNREFERENCED_PARAMETER(lpReserved);

    if (dwReason == DLL_PROCESS_ATTACH)
    {
```

```

        TRACE0("MFCEXPENDDLL.DLL Initializing!\n");

        // Extension DLL one-time initialization
        if (!AfxInitExtensionModule(MfcexpenddllDLL, hInstance))
            return 0;

        // Insert this DLL into the resource chain
        // NOTE: If this Extension DLL is being implicitly linked to by
        // an MFC Regular DLL (such as an ActiveX Control)
        // instead of an MFC application, then you will want to
        // remove this line from DllMain and put it in a separate
        // function exported from this Extension DLL. The Regular DLL
        // that uses this Extension DLL should then explicitly call that
        // function to initialize this Extension DLL. Otherwise,
        // the CDynLinkLibrary object will not be attached to the
        // Regular DLL's resource chain, and serious problems will
        // result.

        new CDynLinkLibrary(MfcexpenddllDLL);
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        TRACE0("MFCEXPENDDLL.DLL Terminating!\n");
        // Terminate the library before destructors are called
        AfxTermExtensionModule(MfcexpenddllDLL);
    }
    return 1;    // ok
}

```

上述代码完成 MFC 扩展 DLL 的初始化和终止处理。

由于 MFC 扩展 DLL 导出函数和变量的方式与其它 DLL 没有什么区别,我们不再细致讲解。

下面直接给出一个 MFC 扩展 DLL 的创建及在应用程序中调用它的例子。

6.1 MFC扩展DLL的创建

下面我们将在 MFC 扩展 DLL 中导出一个按钮类 CSXButton(扩展自 MFC 的 CButton 类), 类 CSXButton 是一个用以取代 CButton 的类, 它使你能在同一个按钮上显示位图和文字, 而 MFC 的按钮仅可显示二者之一。类 CSXbutton 的源代码在 Internet 上广泛流传, 有很好的“群众基础”, 因此用这个类来讲解 MFC 扩展 DLL 有其特殊的功效。

MFC 中包含一些宏，这些宏在 DLL 和调用 DLL 的应用程序中被以不同的方式展开，这使得在 DLL 和应用程序中，使用统一的一个宏就可以表示出输出和输入的不同意思：

```
// for data
#ifndef AFX_DATA_EXPORT
#define AFX_DATA_EXPORT __declspec(dllexport)
#endif

#ifndef AFX_DATA_IMPORT
#define AFX_DATA_IMPORT __declspec(dllimport)
#endif

// for classes
#ifndef AFX_CLASS_EXPORT
#define AFX_CLASS_EXPORT __declspec(dllexport)
#endif

#ifndef AFX_CLASS_IMPORT
#define AFX_CLASS_IMPORT __declspec(dllimport)
#endif

// for global APIs
#ifndef AFX_API_EXPORT
#define AFX_API_EXPORT __declspec(dllexport)
#endif

#ifndef AFX_API_IMPORT
#define AFX_API_IMPORT __declspec(dllimport)
#endif

#ifndef AFX_EXT_DATA
#ifdef _AFXEXT
#define AFX_EXT_CLASS      AFX_CLASS_EXPORT
#define AFX_EXT_API        AFX_API_EXPORT
#define AFX_EXT_DATA       AFX_DATA_EXPORT
#define AFX_EXT_DATADEF
#else
#define AFX_EXT_CLASS      AFX_CLASS_IMPORT
#define AFX_EXT_API        AFX_API_IMPORT
#define AFX_EXT_DATA       AFX_DATA_IMPORT
#define AFX_EXT_DATADEF
#endif
#endif
```

导出一个类，直接在类声明头文件中使用 AFX_EXT_CLASS 即可，以下是导出 CSXButton

类的例子：

```
#ifndef _SXBUTTON_H
#define _SXBUTTON_H

#define SXBUTTON_CENTER -1

class AFX_EXT_CLASS CSXButton : public CButton
{
    // Construction
    public:
        CSXButton();

    // Attributes
    private:
        // Positioning
        BOOL m_bUseOffset;
        CPoint m_pointImage;
        CPoint m_pointText;
        int m_nImageOffsetFromBorder;
        int m_nTextOffsetFromImage;

        // Image
        HICON m_hIcon;
        HBITMAP m_hBitmap;
        HBITMAP m_hBitmapDisabled;
        int m_nImageWidth, m_nImageHeight;

        // Color Tab
        char m_bColorTab;
        COLORREF m_crColorTab;

        // State
        BOOL m_bDefault;
        UINT m_nOldAction;
        UINT m_nOldState;

        // Operations
    public:
        // Positioning
        int SetImageOffset( int nPixels );
        int SetTextOffset( int nPixels );
```

```

        CPoint SetImagePos( CPoint p );
        CPoint SetTextPos( CPoint p );

        // Image
        BOOL SetIcon( UINT nID, int nWidth, int nHeight );
        BOOL SetBitmap( UINT nID, int nWidth, int nHeight );
        BOOL SetMaskedBitmap( UINT nID, int nWidth, int nHeight, COLORREF crTransparentMask );
        BOOL HasImage() { return (BOOL)( m_hIcon != 0 | m_hBitmap != 0 ); }

        // Color Tab
        void SetColorTab(COLORREF crTab);

        // State
        BOOL SetDefaultButton( BOOL bState = TRUE );
        private:
        BOOL SetBitmapCommon( UINT nID, int nWidth, int nHeight, COLORREF crTransparentMask, BOOL
            bUseMask );
        void CheckPointForCentering( CPoint &p, int nWidth, int nHeight );
        void Redraw();

        // Overrides
        // ClassWizard generated virtual function overrides
        //{AFX_VIRTUAL(CSXButton)
        public:
        virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
        //}}AFX_VIRTUAL

        // Implementation
        public:
        virtual ~CSXButton();

        // Generated message map functions
        protected:
        //{AFX_MSG(CSXButton)
        afx_msg LRESULT OnGetText(WPARAM wParam, LPARAM lParam);
        //}}AFX_MSG

        DECLARE_MESSAGE_MAP()
    };

#endif

```

把 SXBUTTON.CPP 文件直接添加到工程，编译工程，得到“mfccxpendd11.lib”和“mfccxpendd11.dll”两个文件。我们用 Visual Studio 自带的 Depends 工具可以查看这个.dll，发现其导出了众多符号（见图 15）。

Ordinal	Mint	Function	Entry Point
1 (0x0001)	0 (0x0000)	?0CSXButton@9QAEHXZ	0x00001028
2 (0x0002)	1 (0x0001)	??1CSXButton@9QAEHXZ	0x00001023
3 (0x0003)	2 (0x0002)	??_TCSXButton@9QAEHXZ	0x000015098
19 (0x0013)	18 (0x0012)	?_GetBaseMessageMap@CSXButton@9QAEHXZ	0x00001050
20 (0x0014)	19 (0x0013)	?_messageEntries@CSXButton@9QAEHXZ	0x000015078
4 (0x0004)	3 (0x0003)	?CheckPointForCentering@CSXButton@9QAEHXZ	0x00001014
5 (0x0005)	4 (0x0004)	?DrawItem@CSXButton@9QAEHXZ	0x00001041
6 (0x0006)	5 (0x0005)	?GetMessageMap@CSXButton@9QAEHXZ	0x00001073
7 (0x0007)	6 (0x0006)	?HasImage@CSXButton@9QAEHXZ	0x0000107D
21 (0x0015)	20 (0x0014)	?messageMap@CSXButton@9QAEHXZ	0x000015070
8 (0x0008)	7 (0x0007)	?Redraw@CSXButton@9QAEHXZ	0x0000104B
9 (0x0009)	8 (0x0008)	?SetBitmap@CSXButton@9QAEHXZ	0x00001037
10 (0x000A)	9 (0x0009)	?SetBitmapCommon@CSXButton@9QAEHXZ	0x00001069
11 (0x000B)	10 (0x000A)	?SetColorTab@CSXButton@9QAEHXZ	0x0000103C
12 (0x000C)	11 (0x000B)	?SetDefaultButton@CSXButton@9QAEHXZ	0x0000105F
13 (0x000D)	12 (0x000C)	?SetIcon@CSXButton@9QAEHXZ	0x00001055
14 (0x000E)	13 (0x000D)	?SetImageOffset@CSXButton@9QAEHXZ	0x00001046
15 (0x000F)	14 (0x000E)	?SetImagePos@CSXButton@9QAEHXZ	0x0000101E
16 (0x0010)	15 (0x000F)	?SetMaskedBitmap@CSXButton@9QAEHXZ	0x0000105A
17 (0x0011)	16 (0x0010)	?SetTextOffset@CSXButton@9QAEHXZ	0x0000102D
18 (0x0012)	17 (0x0011)	?SetTextPos@CSXButton@9QAEHXZ	0x00001064

图 15 导出类时导出的大量符号（+放大该图片）

这些都是类的构造函数、析构函数及其它成员函数和变量经编译器处理过的符号，我们直接用__declspec(dllexport)语句声明类就导出了这些符号。

如果我们想用.lib文件导出这些符号，是非常困难的，我们需要在工程中生成.map文件，查询.map文件的符号，然后将其一一导出。如图 16，打开 DLL 工程的 settings 选项，再选择 Link，勾选其中的产生 MAP 文件（Generate mapfile）就可以产生.map文件了。

打开 mfccxpendd11 工程生成的.map文件，我们发现其中包含了图 15 中所示的符号

(symbol)

```

0001:00000380 ?HasImage@CSXButton@@QAEHXZ 10001380 f i SXBUTTON.OBJ
0001:000003d0 ??0CSXButton@@QAEHXZ 100013d0 f SXBUTTON.OBJ
0001:00000500 ??_GCSXButton@@UAEPAHXZ 10001500 f i SXBUTTON.OBJ
0001:00000570 ??_ECSXButton@@UAEPAHXZ 10001570 f i SXBUTTON.OBJ
0001:00000630 ??1CSXButton@@UAEHXZ 10001630 f SXBUTTON.OBJ
0001:00000700 ?_GetBaseMessageMap@CSXButton@@KGPBUAFX_MSGMAP@@XZ 10001700 f SXBUTTON.OBJ
0001:00000730 ?GetMessageMap@CSXButton@@MBEPBUAFX_MSGMAP@@XZ 10001730 f SXBUTTON.OBJ
0001:00000770 ?Redraw@CSXButton@@AAEXXZ 10001770 f i SXBUTTON.OBJ
0001:000007d0 ?SetIcon@CSXButton@@QAEHIHXZ 100017d0 f SXBUTTON.OBJ

```

......./省略

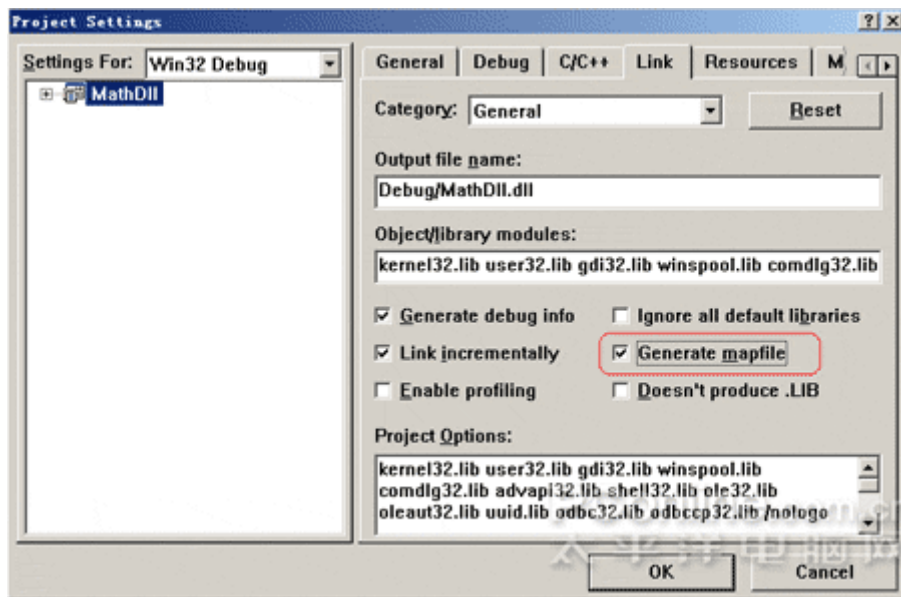


图 16 产生.map文件（+放大该图片）

所以，对于 MFC 扩展 DLL，我们不宜以.lib 文件导出类。

6.2 MFC扩展DLL的调用

在 DLL 所在工作区新增一个 dllcall 工程，它是一个基于对话框的 MFC EXE 程序。在其中增加两个按钮 SXBUTTON1、SXBUTTON2，并设置其属性为“Owner draw”，如图 17。

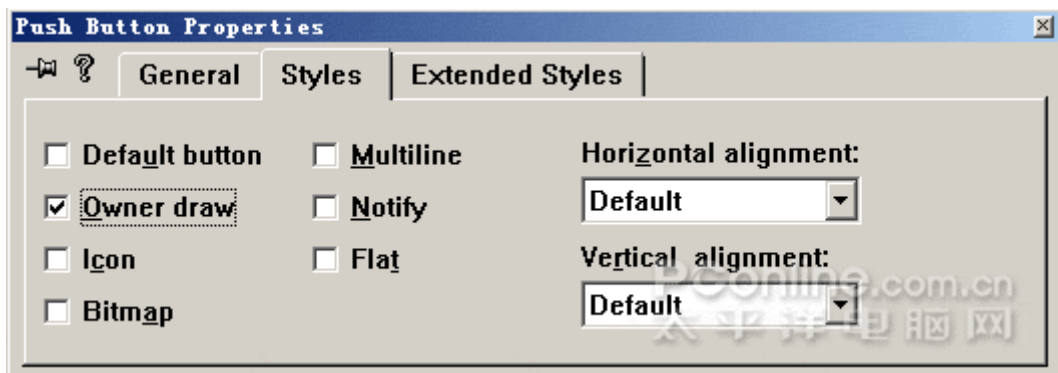


图 17 设置按钮属性为“Owner draw”

在工程中添加两个 ICON 资源：IDI_MSN_ICON (MSN 的图标)、IDI_REFBAR_ICON (Windows 的系统图标)。

修改工程的“calldllDlg.h”头文件为：

```
#include "..\..\mfccexpnddll\SXBUTTON.h" //包含 dll 的导出类头文件
#pragma comment(lib, "mfccexpnddll.lib") //隐式链接 dll
////////////////////////////////////
// CCalldllDlg dialog

class CCalldllDlg : public CDialog
{
// Construction
public:
CCalldllDlg(CWnd* pParent = NULL); // standard constructor

// Dialog Data
//{{AFX_DATA(CCalldllDlg)
enum { IDD = IDD_CALLDLL_DIALOG };
//增加与两个按钮对应的成员变量
CSXButton m_button1;
CSXButton m_button2;
...
}
```

同时，修改“calldllDlg.cpp”文件，使得 m_button1、m_button2 成员变量与对话框上的按钮控件建立关联：

```
void CCalldllDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CCalldllDlg)
DDX_Control(pDX, IDC_BUTTON2, m_button2);
DDX_Control(pDX, IDC_BUTTON1, m_button1);
//}}AFX_DATA_MAP
}
```

修改 BOOL CCalldllDlg::OnInitDialog() 函数，在其中增加对两个按钮设置 ICON 的代码：


```

        BOOL CCallDllDlg::OnInitDialog()
        {
            CDialog::OnInitDialog();

            // Add "About..." menu item to system menu.

            // IDM_ABOUTBOX must be in the system command range.
            ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
            ASSERT(IDM_ABOUTBOX < 0xF000);

            CMenu* pSysMenu = GetSystemMenu(FALSE);
            if (pSysMenu != NULL)
            {
                CString strAboutMenu;
                strAboutMenu.LoadString(IDS_ABOUTBOX);
                if (!strAboutMenu.IsEmpty())
                {
                    pSysMenu->AppendMenu(MF_SEPARATOR);
                    pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
                }
            }

            // Set the icon for this dialog. The framework does this automatically
            // when the application's main window is not a dialog
            SetIcon(m_hIcon, TRUE); // Set big icon
            SetIcon(m_hIcon, FALSE); // Set small icon

            // TODO: Add extra initialization here
            m_button1.SetIcon(IDI_MSN_ICON, 16, 16);
            m_button2.SetIcon(IDI_REFBAR_ICON, 16, 16);

            return TRUE; // return TRUE unless you set the focus to a control
        }

```

运行程序，将出现如图 18 的对话框，图形和文字同时出现在按钮上，这说明我们正确地调用了 MFC 扩展 DLL。



图 18 DLL 扩展的按钮被显示

如果我们不修改 `void CCallDllDlg::DoDataExchange(CDataExchange* pDX)`，即不增加下列代码：

```
DDX_Control(pDX, IDC_BUTTON2, m_button2);  
DDX_Control(pDX, IDC_BUTTON1, m_button1);
```

我们也可以在 `BOOL CCallDllDlg::OnInitDialog()` 函数中添加如下代码实现

`m_button1`、`m_button2` 与 `IDC_BUTTON1`、`IDC_BUTTON2` 的关联：

```
m_button1.SubclassDlgItem(IDC_BUTTON1, this);  
m_button2.SubclassDlgItem(IDC_BUTTON2, this);
```

但是，`DDX_Control` 与按钮类的 `SubclassDlgItem` 成员函数不能同时存在，否则程序会出错。

6.3 总结

由以上分析可知，MFC 扩展 DLL 的导出与引用方式与前几节所讲述的方式没有太大的差别，MFC 扩展 DLL 主要强调对 MFC 进行功能扩展。因此，如果 DLL 的目标不是增强 MFC 的功能，其与应用程序的接口也不是 MFC，请不要将 DLL 建立为 MFC 扩展 DLL。

下载本文相关源代码。（58.3KB，Winzip压缩）