原创 VC++动态链接库(DLL)编程（一）——理解库

2005-10-20 23:36:00

标签：VC++ 编程 DLL [推送到技术圈]

VC++动态链接库(DLL)编程（一）

——理解库

作者： 宋宝华 **e-mail:** 21cnbao@21cn.com

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。

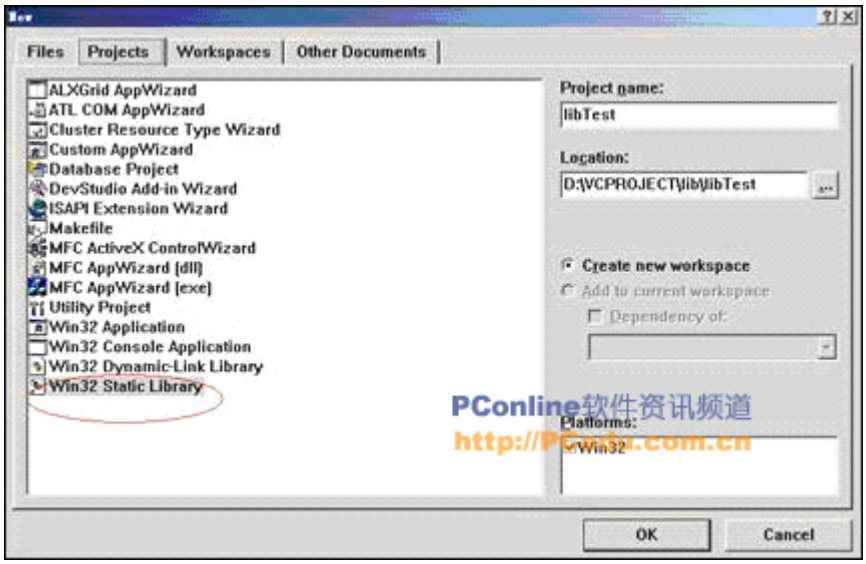

图1建立一个静态链接库

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。

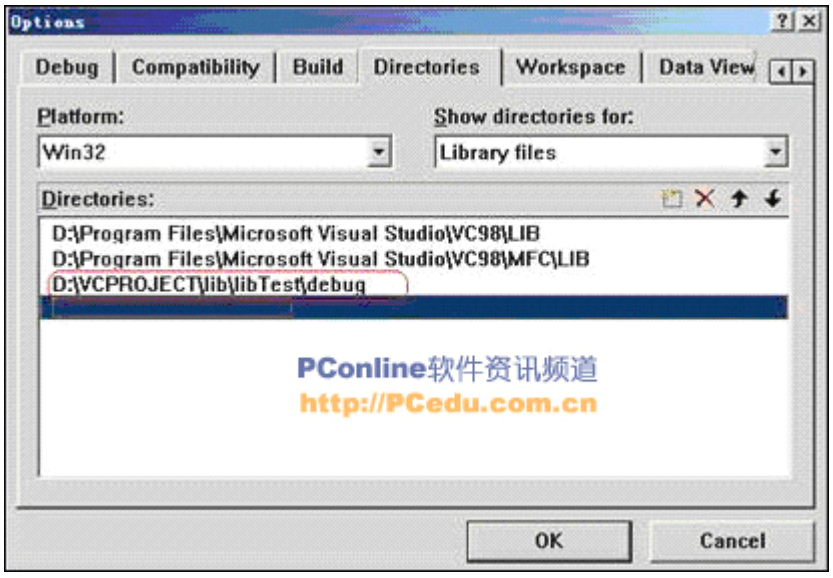

图2在 VC 中设置库文件路径

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。

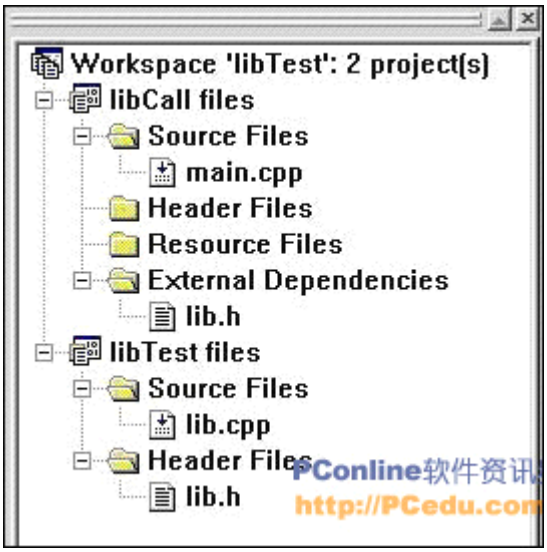

图4　把库工程和调用库的工程放入同一工作区进行调试

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！

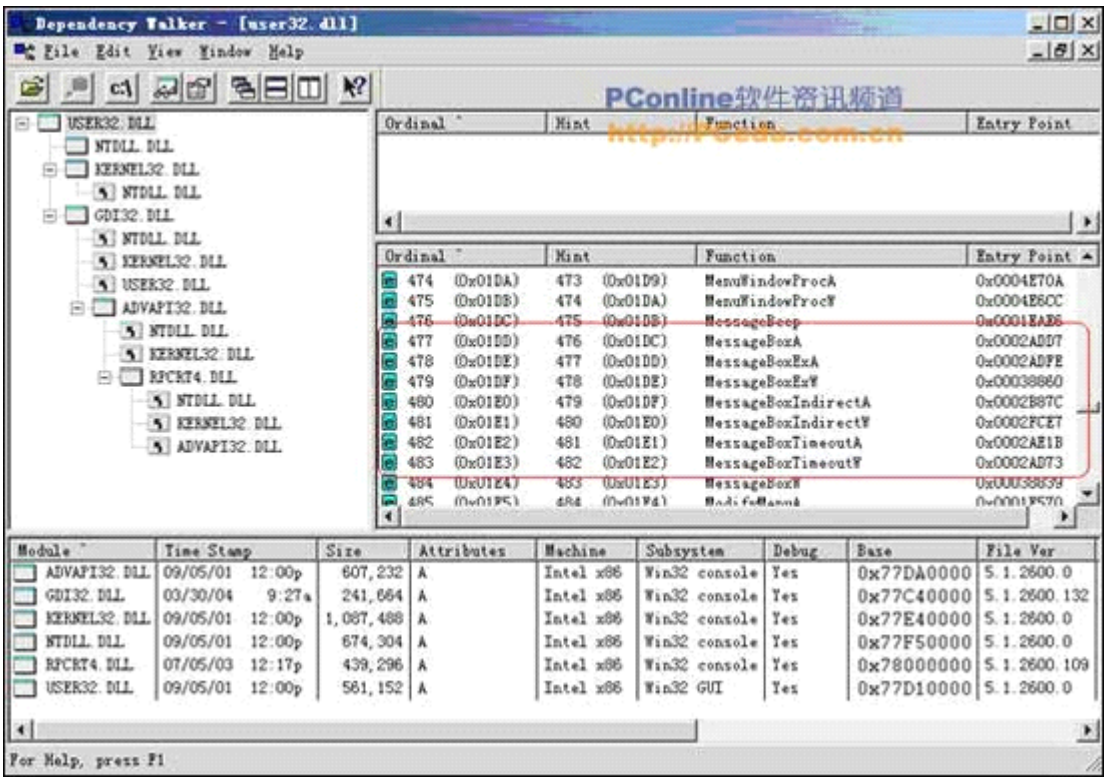

图5　用 Depends 查看 DLL

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。

原创 VC++动态链接库(DLL)编程（二）——非 MFC DLL

2005-10-20 23:52:00

标签：VC++ 编程 DLL　　[推送到技术圈]

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。

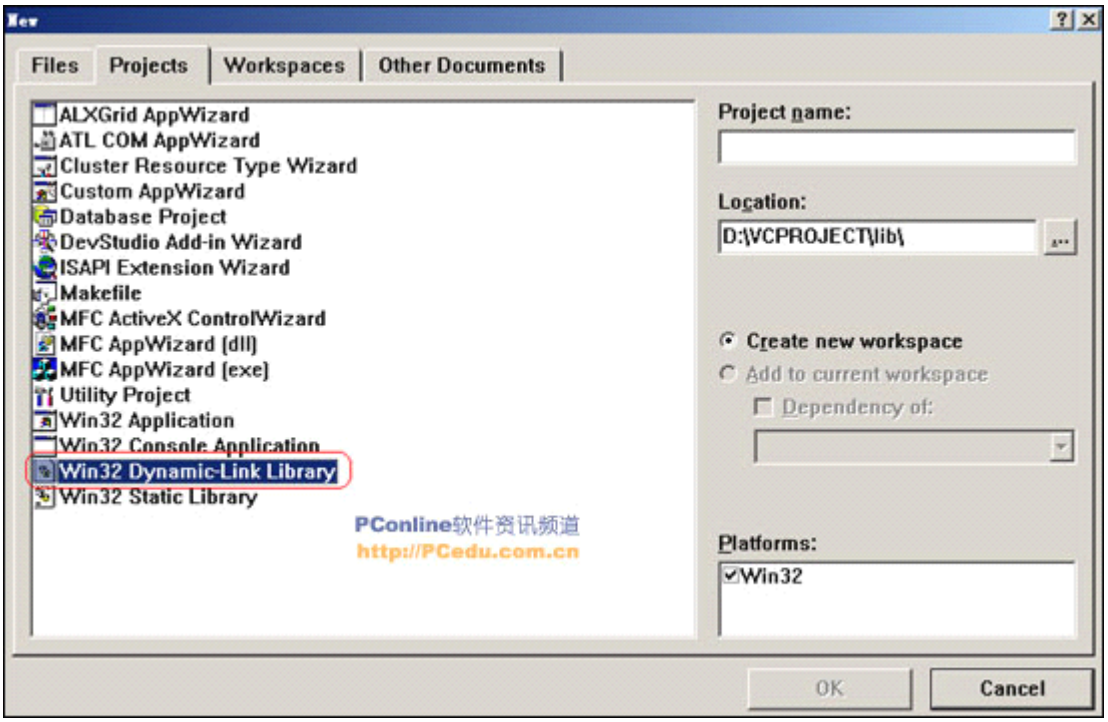

图6 建立一个非 MFC DLL

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。

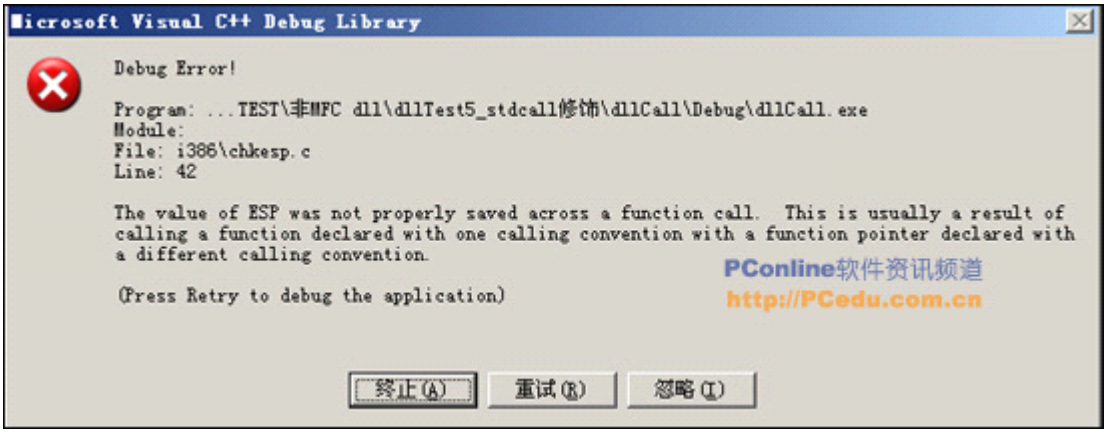

图7 调用约定不匹配时的运行错误

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，其思维方式可以直接过渡。

原创 VC++动态链接库(DLL)编程（三）——MFC 规则 DLL

2005-10-21 12:51:00

标签：VC++ 编程 DLL [推送到技术圈]

VC++动态链接库(DLL)编程（三）

——MFC 规则 DLL

作者: 宋宝华 **e-mail:**21cnbao@21cn.com

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。

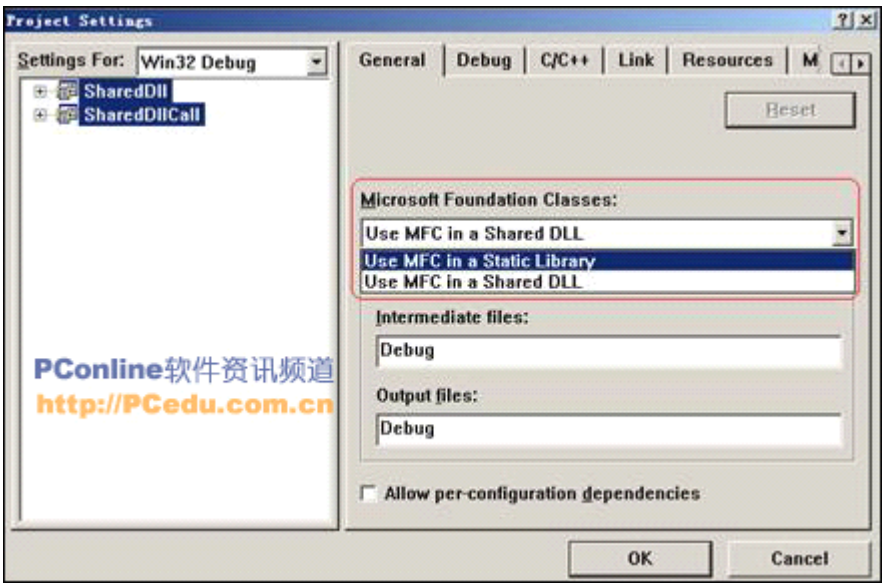

图8设置动态/静态链接 MFC DLL

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。

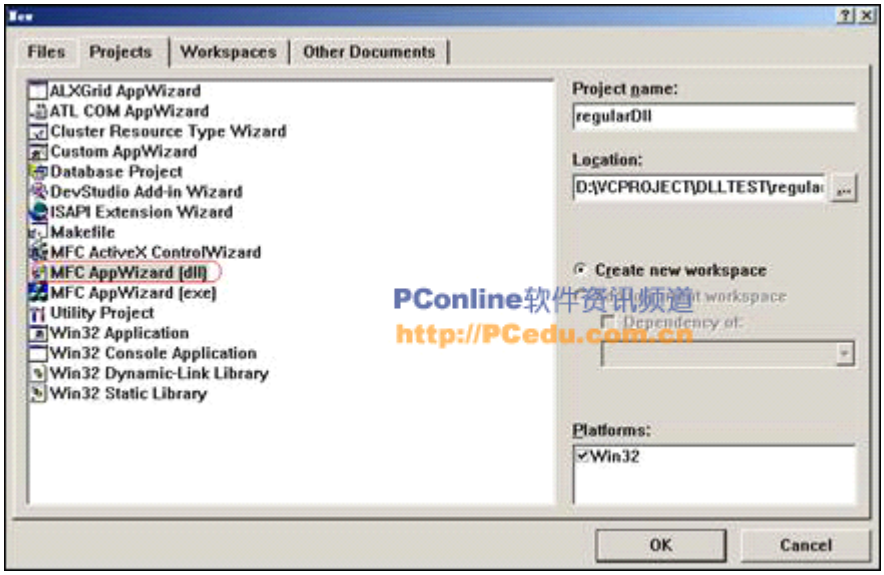

图9　MFC DLL 工程的创建

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”。

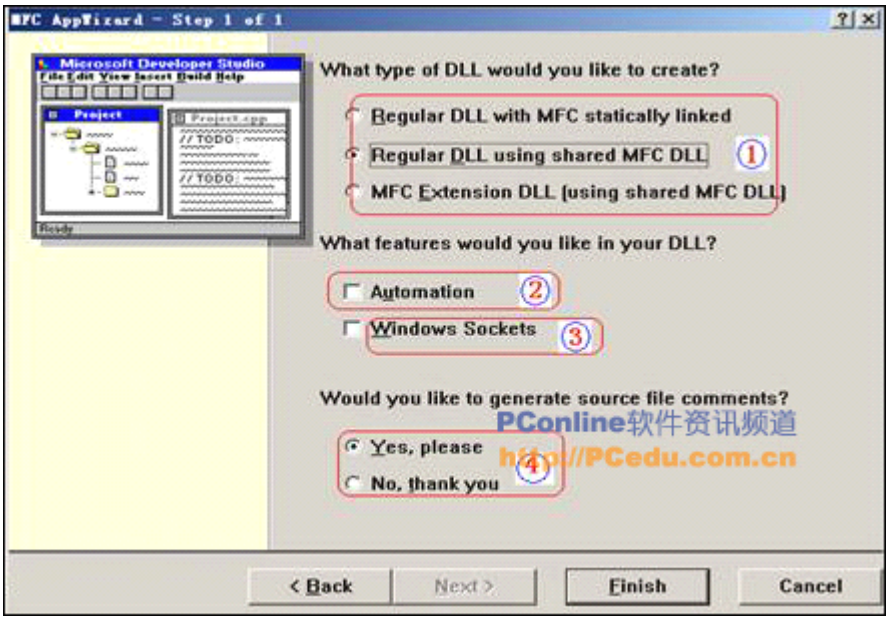

图10　MFC DLL 的创建选项

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）：

第一组文件：CWinApp 继承类的声明与实现

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

#if

!defined(AFX_REGULARDLL_H__3E9CB22B_588B_4388_B778_B3416ADB79B3__I
NCLUDED_)
#define
AFX_REGULARDLL_H__3E9CB22B_588B_4388_B778_B3416ADB79B3__INCLUDE
D_

#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

```
#if

!defined(AFX_DLLDIALOG_H__CEA4C6AF_245D_48A6_B11A_A5521EAD7C4E__I
NCLUDED_)
#define
AFX_DLLDIALOG_H__CEA4C6AF_245D_48A6_B11A_A5521EAD7C4E__INCLUDE
D_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DllDialog.h : header file

/////////////////////////////////////////////////////////////////////////////
// CDllDialog dialog
```

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

	enum { IDD = IDD_DLL_DIALOG };

	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV support

// Implementation
protected:

	afx_msg void OnHelloButton();
	DECLARE_MESSAGE_MAP()
};

#endif

// DllDialog.cpp : implementation file

#include "stdafx.h"
#include "RegularDll.h"
#include "DllDialog.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDllDialog dialog

CDllDialog::CDllDialog(CWnd* pParent /*=NULL*/)
	: CDialog(CDllDialog::IDD, pParent)
{
}

void CDllDialog::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
```

```
}

BEGIN_MESSAGE_MAP(CDllDialog, CDialog)
        ON_BN_CLICKED(IDC_HELLO_BUTTON, OnHelloButton)
END_MESSAGE_MAP()

//////////////////////////////////////////////////////////////////////////
// CDllDialog message handlers

void CDllDialog::OnHelloButton()
{
        MessageBox("Hello,pconline 的网友","pconline");
}
```

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）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），在其中包含如图13的对话框。

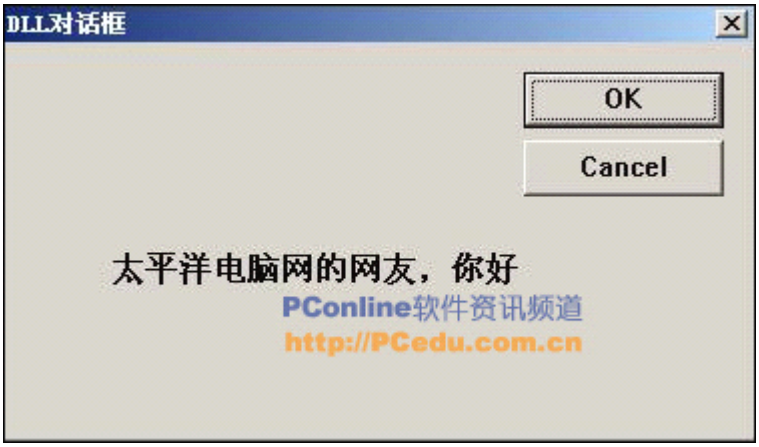

图13　DLL 中的对话框

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。

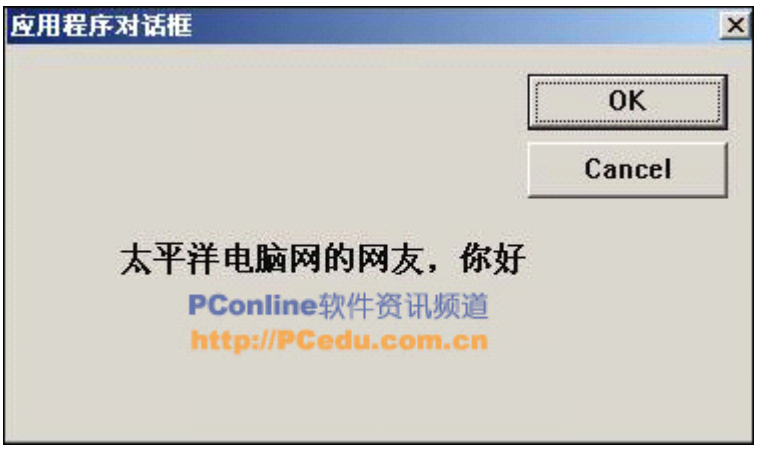

图14　EXE 中的对话框

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）。

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。

原创 VC++动态链接库(DLL)编程（四）——MFC扩展 DLL

2005-10-21 23:09:00

标签：VC++ 编程 异常处理 [推送到技术圈]

VC++动态链接库(DLL)编程（四）

——MFC扩展DLL

作者： 宋宝华 **e-mail:** 21cnbao@21cn.com

前文我们对非 MFC DLL 和 MFC 规则 DLL 进行了介绍，现在开始详细分析 DLL 的最后一种类型——MFC 扩展 DLL。

6.1概论

MFC 扩展 DLL 与 MFC 规则 DLL 的相同点在于在两种 DLL 的内部都可以使用 MFC 类库，其不同点在于 MFC 扩展 DLL 与应用程序的接口可以是 MFC 的。MFC 扩展 DLL 的含义在于它是 MFC 的扩展，其主要功能是实现从现有 MFC 库类中派生出可重用的类。MFC 扩展 DLL 使用 MFC 动态链接库版本，因此只有用共享 MFC 版本生成的 MFC 可执行文件（应用程序或规则 DLL）才能使用 MFC 扩展 DLL。

从前文可知，MFC 规则 DLL 被 MFC 向导自动添加了一个 CWinApp 的对象，而 MFC 扩展 DLL 则不包含该对象，它只是被自动添加了 DllMain 函数。对于 MFC 扩展 DLL，开发人员必须在 DLL 的 DllMain 函数中添加初始化和结束代码。

从下表我们可以看出三种 DLL 对 DllMain 入口函数的不同处理方式：

DLL 类型	入口函数
非 MFC DLL	编程者提供 DllMain 函数
MFC 规则 DLL	CWinApp 对象的 InitInstance 和 ExitInstance
MFC 扩展 DLL	MFC DLL 向导生成 DllMain 函数

对于 MFC 扩展 DLL，系统会自动在工程中添加如下表所示的宏，这些宏为 DLL 和应用程序的编写提供了方便。像 AFX_EXT_CLASS、AFX_EXT_API、AFX_EXT_DATA 这样的宏，在 DLL 和应用程序中将具有不同的定义，这取决于_AFXEXT 宏是否被定义。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。在 DLL 中，表示输出（因为_AFXEXT 被定义，通常是在编译器的标识参数中指定/D_AFXEXT）；在应用程序中，则表示输入（_AFXEXT 没有定义）。

宏	定义
AFX_CLASS_IMPORT	__declspec(dllexport)
AFX_API_IMPORT	__declspec(dllexport)
AFX_DATA_IMPORT	__declspec(dllexport)
AFX_CLASS_EXPORT	__declspec(dllexport)
AFX_API_EXPORT	__declspec(dllexport)
AFX_DATA_EXPORT	__declspec(dllexport)
AFX_EXT_CLASS	#ifdef _AFXEXT AFX_CLASS_EXPORT #else AFX_CLASS_IMPORT
AFX_EXT_API	#ifdef _AFXEXT AFX_API_EXPORT #else AFX_API_IMPORT
AFX_EXT_DATA	#ifdef _AFXEXT

```
AFX_DATA_EXPORT
#else
AFX_DATA_IMPORT
```

6.2 MFC 扩展 DLL 导出 MFC 派生类

在这个例子中，我们将产生一个名为“ExtDll”的 MFC 扩展 DLL 工程，在这个 DLL 中导出一个对话框类，这个对话框类派生自 MFC 类 CDialog。

使用 MFC 向导生成 MFC 扩展 DLL 时，系统会自动添加如下代码：

```
static AFX_EXTENSION_MODULE ExtDllDLL = { NULL, NULL };

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

	if (dwReason == DLL_PROCESS_ATTACH)
	{
		TRACE0( "EXTDLL.DLL Initializing!\n" );

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

		// Insert this DLL into the resource chain
		new CDynLinkLibrary( ExtDllDLL );
	}
	else if (dwReason == DLL_PROCESS_DETACH)
	{
		TRACE0( "EXTDLL.DLL Terminating!\n" );
		// Terminate the library before destructors are called
		AfxTermExtensionModule( ExtDllDLL );
	}
	return 1;   // ok
}
```

这一段代码含义晦涩，我们需要对其进行解读：

（1）上述代码完成 MFC 扩展 DLL 的初始化和终止处理；

（2）初始化期间所创建的 CDynLinkLibrary 对象使 MFC 扩展 DLL 可以将 DLL 中的 CRuntimeClass 对象或资源导出到应用程序；

（3）AfxInitExtensionModule 函数捕获模块的 CRuntimeClass 结构和在创建 CDynLinkLibrary 对象时使用的对象工厂（COleObjectFactory 对象）；

（4）AfxTermExtensionModule 函数使 MFC 得以在每个进程与扩展 DLL 分离时（进程退出或使用 AfxFreeLibrary 卸载 DLL 时）清除扩展 DLL；

（5）第一条语句 static AFX_EXTENSION_MODULE ExtDllDLL = { NULL, NULL }; 定义了一个 AFX_EXTENSION_MODULE 类的静态全局对象，

AFX_EXTENSION_MODULE 的定义如下：

```
struct AFX_EXTENSION_MODULE
{
    BOOL bInitialized;
    HMODULE hModule;
    HMODULE hResource;
    CRuntimeClass* pFirstSharedClass;
    COleObjectFactory* pFirstSharedFactory;
};
```

由 AFX_EXTENSION_MODULE 的定义我们可以更好的理解（2）、（3）、（4）点。

在资源编辑器中添加一个如图15所示的对话框，并使用 MFC 类向导为其添加一个对应的类 CExtDialog，系统自动添加了 ExtDialog.h 和 ExtDialog.cpp 两个头文件。

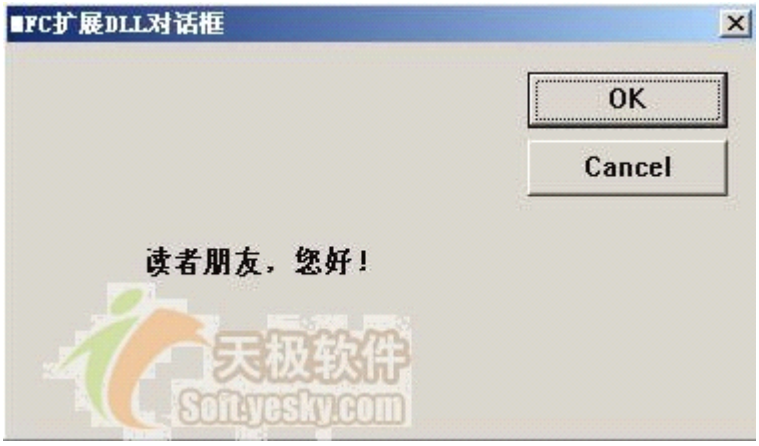

图15 MFC 扩展 DLL 中的对话框

修改 ExtDialog.h 中 CExtDialog 类的声明为：

```
class AFX_EXT_CLASS CExtDialog : public CDialog
{
public:
    CExtDialog( CWnd* pParent = NULL );

    enum { IDD = IDD_DLL_DIALOG };

    protected:
    virtual void DoDataExchange( CDataExchange* pDX );

    DECLARE_MESSAGE_MAP()
};
```

这其中最主要的改变是我们在 class AFX_EXT_CLASS CExtDialog 语句中添加了“AFX_EXT_CLASS”宏，则使得 DLL 中的 CExtDialog 类被导出。

6.3 MFC扩展DLL的加载

6.3.1隐式加载

我们在6.2工程所在的工作区中添加一个LoadExtDllDlg工程，用于演示MFC扩展DLL的加载。在LoadExtDllDlg工程中添加一个如图16所示的对话框，这个对话框上包括一个“调用DLL”按钮。

图16 MFC扩展DLL调用工程中的对话框

在与图16对应对话框类实现文件的头部添加：

```
// LoadExtDllDlg.cpp : implementation file
//
#include "..\ExtDialog.h"
#pragma comment( lib, "ExtDll.lib" )
```

而“调用DLL”按钮的单击事件的消息处理函数为：

```
void CLoadExtDllDlg::OnDllcallButton()
{
        CExtDialog extDialog;
        extDialog.DoModal();
}
```

当我们单击“调用DLL”的时候，弹出了如图15的对话框。

为提供给用户隐式加载（MFC扩展DLL一般使用隐式加载，具体原因见下节），MFC扩展DLL需要提供三个文件：

（1）描述DLL中扩展类的头文件；

（2）与动态链接库对应的.LIB文件；

（3）动态链接库.DLL文件本身。

有了这三个文件，应用程序的开发者才可充分利用MFC扩展DLL。

6.3.2显示加载

显示加载MFC扩展DLL应使用MFC全局函数AfxLoadLibrary而不是WIN32 API中的LoadLibrary。AfxLoadLibrary最终也调用了LoadLibrary这个API，但是在调用之前进行了线程同步的处理。

AfxLoadLibrary的函数原型与LoadLibrary完全相同，为：

```
HINSTANCE AFXAPI AfxLoadLibrary( LPCTSTR lpszModuleName );
```

与之相对应的是，MFC应用程序应使用AfxFreeLibrary而非FreeLibrary卸载MFC扩展DLL。AfxFreeLibrary的函数原型也与FreeLibrary完全相同，为：

```
BOOL AFXAPI AfxFreeLibrary( HINSTANCE hInstLib );
```

如果我们把上例中的“调用DLL”按钮单击事件的消息处理函数改为：

```
void CLoadExtDllDlg::OnDllcallButton()
{
    HINSTANCE hDll = AfxLoadLibrary( "ExtDll.dll" );
        if(NULL == hDll)
        {
                AfxMessageBox( "MFC 扩展 DLL 动态加载失败" );
                return;
        }

        CExtDialog extDialog;
        extDialog.DoModal();

        AfxFreeLibrary(hDll);
}
```

则工程会出现 link 错误：

LoadExtDllDlg.obj : error LNK2001: unresolved external symbol "__declspec(dllimport) public: virtual __thiscall CExtDialog::~CExtDialog(void)" (__imp_??1CExtDialog@@UAE@XZ)

LoadExtDllDlg.obj : error LNK2001: unresolved external symbol "__declspec(dllimport) public: __thiscall CExtDialog::CExtDialog(class CWnd *)" (__imp_??0CExtDialog@@QAE@PAVCWnd@@@Z)

提示 CExtDialog 的构造函数和析构函数均无法找到！是的，对于派生 MFC 类的 MFC 扩展 DLL，当我们要在应用程序中使用 DLL 中定义的派生类时，我们不宜使用动态加载 DLL 的方法。

6.4 MFC 扩展 DLL 加载 MFC 扩展 DLL

我们可以在 MFC 扩展 DLL 中再次使用 MFC 扩展 DLL，但是，由于在两个 DLL 中对于 AFX_EXT_CLASS、AFX_EXT_API、AFX_EXT_DATA 宏的定义都是输出，这会导致调用的时候出现问题。

我们将会在调用 MFC 扩展 DLL 的 DLL 中看到 link 错误：

error LNK2001: unresolved external symbol ….....

因此，在调用 MFC 扩展 DLL 的 MFC 扩展 DLL 中，在包含被调用 DLL 的头文件之前，需要临时重新定义 AFX_EXT_CLASS 的值。下面的例子显示了如何实现：

```
//临时改变宏的含义“输出”为“输入”
#undef AFX_EXT_CLASS
#undef AFX_EXT_API
#undef AFX_EXT_DATA
#define AFX_EXT_CLASS AFX_CLASS_IMPORT
#define AFX_EXT_API AFX_API_IMPORT
#define AFX_EXT_DATA AFX_DATA_IMPORT

//包含被调用 MFC 扩展 DLL 的头文件
#include "CalledDLL.h"

//恢复宏的含义为输出
#undef AFX_EXT_CLASS
```

```
#undef AFX_EXT_API
#undef AFX_EXT_DATA
#define AFX_EXT_CLASS AFX_CLASS_EXPORT
#define AFX_EXT_API AFX_API_EXPORT
#define AFX_EXT_DATA AFX_DATA_EXPORT
```

6.5 MFC 扩展 DLL 导出函数和变量

MFC 扩展 DLL 导出函数和变量的方法也十分简单，下面我们给出一个简单的例子。我们在 MFC 向导生成的 MFC 扩展 DLL 工程中添加 gobal.h 和 global.cpp 两个文件：

```
//global.h:MFC 扩展 DLL 导出变量和函数的声明
extern "C"
{       int AFX_EXT_DATA total; //导出变量
          int AFX_EXT_API add( int x, int y ); //导出函数
}

//global.cpp:MFC 扩展 DLL 导出变量和函数定义
#include "StdAfx.h"
#include "global.h"

extern "C" int total;
int add(int x,int y)
{
          total = x + y;
          return total;
}
```

编写一个简单的控制台程序来调用这个 MFC 扩展 DLL：

```
#include <iostream.h>
#include <afxver_.h>
          //AFX_EXT_DATA、AFX_EXT_API 宏的定义在 afxver_.h 头文件中

#pragma comment ( lib, "ExtDll.lib" )
#include "..\global.h"

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

          cout << add(2,3) << endl;
          cout << total;

          return 0;
}
```

运行程序，在控制台上看到：

```
5
5
```

另外，在 Visual C++下建立 MFC 扩展 DLL 时，MFC DLL 向导会自动生成.def 文件。因此，对于函数和变量，我们除了可以利用 AFX_EXT_DATA、AFX_EXT_API 宏导出以外，在.def 文件中定义导出也是一个很好的办法。与之相比，在.def 文件中导出类却较麻烦。通常需要从工程生成的.map 文件中获得类的所有成员函数被 C++编译器更改过的标识符，并且在.def 文件中导出这些“奇怪”的标识符。因此，MFC 扩展 DLL 通常以 AFX_EXT_CLASS 宏直接声明导出类。

6.6 MFC 扩展 DLL 的应用

上述各小节所举 MFC 扩展 DLL 的例子均只是为了说明某方面的问题，没有真实地体现“MFC 扩展”的内涵，譬如6.2派生自 CDialog 的类也不具备比 CDialog 更强的功能。MFC 扩展 DLL 的真实内涵体现在它提供的类虽然派生自 MFC 类，但是提供了比 MFC 类更强大的功能、更丰富的接口。下面我们来看一个具体的例子。

我们知道 static 控件所对应的 CStatic 类不具备设置背景和文本颜色的接口，这使得我们不能在对话框或其它用户界面上自由灵活地修改 static 控件的颜色风格，因此我们需要一个提供了 SetBackColor 和 SetTextColor 接口的 CStatic 派生类 CMultiColorStatic。

这个类的声明如下：

```
class AFX_EXT_CLASS CMultiColorStatic : public CStatic
{
        // Construction
public:
        CMultiColorStatic();
        virtual ~CMultiColorStatic();
        // Attributes
protected:
        CString m_strCaption;
        COLORREF m_BackColor;
        COLORREF m_TextColor;
        // Operations
public:
    void SetTextColor( COLORREF TextColor );
    void SetBackColor( COLORREF BackColor );
          void SetCaption( CString strCaption );
        // Generated message map functions
protected:
    afx_msg void OnPaint();
        DECLARE_MESSAGE_MAP()
};
```

在这个类的实现文件中，我们需要为它提供 WM_PAINT 消息的处理函数（这是因为颜色的设置依赖于 WM_PAINT 消息）：

```
BEGIN_MESSAGE_MAP(CMultiColorStatic, CStatic)
//{{AFX_MSG_MAP(CMultiColorStatic)
        ON_WM_PAINT() //为这个类定义 WM_PAINT 消息处理函数
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

下面是这个类中的重要成员函数：

```
//为 CMultiColorStatic 类添加“设置文本颜色”接口
void CMultiColorStatic::SetTextColor( COLORREF TextColor )
{
        m_TextColor = TextColor;          //设置文字颜色
}
//为 CMultiColorStatic 类添加“设置背景颜色”接口
void CMultiColorStatic::SetBackColor( COLORREF BackColor )
{
    m_BackColor = BackColor;          //设置背景颜色
}
//为 CMultiColorStatic 类添加“设置标题”接口
void CMultiColorStatic::SetCaption( CString strCaption )
{
        m_strCaption = strCaption;
}
//重画 Static，颜色和标题的设置都依赖于这个函数
void CMultiColorStatic::OnPaint()
{
    CPaintDC dc(this); // device context for painting
    CRect rect;
    GetClientRect( &rect );
    dc.SetBkColor( m_BackColor );
    dc.SetBkMode( TRANSPARENT );
    CFont *pFont = GetParent()->GetFont();//得到父窗体的字体
    CFont *pOldFont;
    pOldFont = dc.SelectObject( pFont );//选用父窗体的字体
    dc.SetTextColor( m_TextColor );//设置文本颜色
    dc.DrawText( m_strCaption, &rect, DT_CENTER );//文本在 Static 中央
    dc.SelectObject( pOldFont );
}
```

为了验证 CMultiColorStatic 类，我们制作一个基于对话框的应用程序，它包含一个如图17所示的对话框。该对话框上包括一个 static 控件和三个按钮，这三个按钮可分别把 static 控件设置为“红色”、“蓝色”和“绿色”。

图17扩展的 CStatic 类调用演示

下面看看应如何编写与这个对话框对应的类。

包含这种 Static 的对话框类的声明如下：

```
#include "..\MultiColorStatic.h"
#pragma comment ( lib, "ColorStatic.lib" )

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

	enum { IDD = IDD_CALLDLL_DIALOG };
	CMultiColorStatic	m_colorstatic; //包含一个 CMultiColorStatic 的实例

	protected:
	virtual void DoDataExchange(CDataExchange* pDX);//DDX/DDV support

	HICON m_hIcon;

	// Generated message map functions
	//{{AFX_MSG(CCallDllDlg)
	virtual BOOL OnInitDialog();
	afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	afx_msg void OnRedButton();
	afx_msg void OnBlueButton();
	afx_msg void OnGreenButton();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

下面是这个类中与使用 CMultiColorStatic 相关的主要成员函数：

```
void CCallDllDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CCallDllDlg)
	DDX_Control(pDX, IDC_COLOR_STATIC, m_colorstatic);
	 //使 m_colorstatic 与 IDC_COLOR_STATIC 控件关联
	//}}AFX_DATA_MAP
}

BOOL CCallDllDlg::OnInitDialog()
```

```
{
	…
	// TODO: Add extra initialization here

	//初始 static 控件的显示
	m_colorstatic.SetCaption("最开始为黑色");
	m_colorstatic.SetTextColor(RGB(0,0,0));

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

//设置 static 控件文本颜色为红色
void CCallDllDlg::OnRedButton()
{
	m_colorstatic.SetCaption( "改变为红色" );
	m_colorstatic.SetTextColor( RGB( 255, 0, 0 ) );
	Invalidate( TRUE );	//导致发出 WM_PAINT 消息
}

//设置 static 控件文本颜色为蓝色
void CCallDllDlg::OnBlueButton()
{
	m_colorstatic.SetCaption( "改变为蓝色" );
	m_colorstatic.SetTextColor( RGB( 0, 0, 255 ) );
	Invalidate( TRUE );	//导致发出 WM_PAINT 消息
}

//设置 static 控件文本颜色为绿色
void CCallDllDlg::OnGreenButton()
{
	m_colorstatic.SetCaption( "改变为绿色" );
	m_colorstatic.SetTextColor( RGB(0,255,0) );
	Invalidate( TRUE );	//导致发出 WM_PAINT 消息
}
```

至此，我们已经讲解完成了所有类型的动态链接库，即非 MFC DLL、MFC 规则 DLL 和 MFC 扩展 DLL。下一节将给出 DLL 的三个工程实例，与读者朋友们共同体会 DLL 的应用范围和使用方法。

原创 VC++动态链接库(DLL)编程（五）——DLL 典型实例

2005-10-24 18:21:00

标签：VC++ 编程 DLL [推送到技术圈]

VC++动态链接库(DLL)编程（五）

——DLL 典型实例

作者：宋宝华 **e-mail:**21cnbao@21cn.com

动态链接库 DLL 实现了库的共享，体现了代码重用的思想。我们可以把广泛的、具有共性的、能够多次被利用的函数和类定义在库中。这样，在再次使用这些函数和类的时候，就不再需要重新添加与这些函数和类相关的代码。具有共性的问题大致有哪些呢？笔者归纳如下：

（1）通用的算法

图像处理、视频音频解码、压缩与解压缩、加密与解密通常采用某些特定的算法，这些算法较固定且在这类程序中往往经常被使用。

（2）纯资源 DLL

我们可以从 DLL 中获取资源，对于一个支持多种语言的应用程序而言，我们可以判断操作系统的语言，并自动为应用程序加载与 OS 对应的语言。这是多语言支持应用程序的一般做法。

（3）通信控制 DLL

串口、网口的通信控制函数如果由 DLL 提供则可以使应用程序轻松不少。在工业控制、modem 程序甚至 socket 通信中，经常使用通信控制 DLL。

本节将给出 DLL 的三个典型应用实例。

7.1算法 DLL

我们直接用读者的一个提问作为例子。

宋宝华先生，您好!

我在 pconline 上看到你连载的《VC++动态链接库(DLL)编程深入浅出》，觉得非常好。我以前主要是用 Delphi 的，C/C++学过，对 Win32和 VCL 比较熟悉，但是没有接触过 VC++，对 MFC 很陌生。这段时间和一个同学合作做光学成像的计算机模拟，用到傅立叶变换，手里面有例程是 VC++写的。我们的界面是用 Delphi 开发，需要将其傅立叶变换功能提出做一个 DLL 供 Delphi 调用。苦于不懂 MFC，试了很多方法，都不成功，最后只得采用折衷方案，简单修改一下程序，传一个参数进去，当作 exe 来调用，才没有耽搁后续进程。

……

谢谢!

致

礼!

某某

学习过较高级别数学（概率统计与随机过程）、信号与线性系统及数字信号处理的读者

应该知道，傅立叶变换是一种在信号分析中常用的算法，用于时域和频域的相互转换。FFT 变换算法通用而有共性，我们适宜把它集成在一个 DLL 中。

随后，这位读者提供了这样的一个函数：

```
/*  函数名称：FFT()
*   参数:
*   complex<double> * TD      -指向时域数组的指针
*   complex<double> * FD      -指向频域数组的指针
*   r                                                    －2的幂数，即迭代次数
*   返回值:无。
*   说明:该函数用来实现快速傅立叶变换
*/
void FFT(complex<double> * TD, complex<double> * FD, int r)
{
        LONG    count; //傅立叶变换点数
        int             i,j,k; //循环变量
        int             bfsize,p; //中间变量
        double      angle; //角度
        complex<double> *W,*X1,*X2,*X;

        count = 1 << r; //傅立叶变换点数

        //分配运算所需存储器
        W = new complex<double>[count / 2];
        X1 = new complex<double>[count];
        X2 = new complex<double>[count];

        //计算加权系数
        for(i = 0; i < count / 2; i++)
        {
                angle = -i * PI * 2 / count;
                W[i] = complex<double> (cos(angle), sin(angle));
        }

        //将时域点写入 X1
        memcpy(X1, TD, sizeof(complex<double>) * count);

        //采用蝶形算法进行快速傅立叶变换
        for(k = 0; k < r; k++)
        {
                for(j = 0; j < 1 << k; j++)
                {
                        bfsize = 1 << (r-k);
                        for(i = 0; i < bfsize / 2; i++)
                        {
```

```
                                        p = j * bfsize;
                                        X2[i + p] = X1[i + p] + X1[i + p + bfsize / 2];
                                        X2[i + p + bfsize / 2] = (X1[i + p] - X1[i + p + bfsize / 2])
* W[i * (1<<k)];
                                }
                        }
                        X = X1;
                        X1 = X2;
                        X2 = X;
                }

                //重新排序
                for(j = 0; j < count; j++)
                {
                        p = 0;
                        for(i = 0; i < r; i++)
                        {
                                if (j&(1<<i))
                                {
                                        p+=1<<(r-i-1);
                                }
                        }
                        FD[j]=X1[p];
                }

                //释放内存
                delete W;
                delete X1;
                delete X2;
        }
```

既然有了 FFT 这个函数，我们要把它做在 DLL 中，作为 DLL 的一个接口将是十分简单的，其步骤如下：

（1）利用 MFC 向导建立一个非 MFC DLL；

（2）在工程中添加 fft.h 和 fft.cpp 两个文件；

fft.h 的源代码为：

```
#ifndef FFT_H
 #define FFT_H

 #include <complex>
 using namespace std;
 extern "C" void __declspec(dllexport) __stdcall FFT(complex<double> * TD,
complex<double> * FD, int r);

 #define PI 3.1415926
```

```
#endif
```

fft.cpp 的源代码为：

```
/*文件名：fft.cpp   */
#include "fft.h"
void __stdcall FFT(complex<double> * TD, complex<double> * FD, int r)
{
  …//读者提供的函数代码
}
```

在任何编程语言中使用 Win32 API LoadLibrary 都可以加载这个 DLL，而使用 GetProcAddress(hDll, "FFT")则可以获得函数 FFT 的地址，读者所提到的 Delphi 当然也不例外。

这个 DLL 中有两点需要注意：

（1）使用 extern "C"修饰函数声明，否则，生成的 DLL 只能供 C++调用；

（2）使用__stdcall 修饰函数声明及定义，__stdcall 是 Windows API 的函数调用方式。

7.2纯资源 DLL

我们在应用程序中产生如图18所示的资源（对话框）。

图18中文对话框

在与这个应用程序相同的工作区里利用 MFC 向导建立两个简单的 DLL，把应用工程中的资源全选后分别拷贝到 ChineseDll 和 EngLishDll，在 EnglishDll 工程的资源文件中搜索下面的语句：

```
//////////////////////////////////////////////////////////////////////////
// Chinese (P.R.C.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_CHS)
#ifdef _WIN32
LANGUAGE LANG_CHINESE, SUBLANG_CHINESE_SIMPLIFIED
#pragma code_page(936)
#endif //_WIN32
```

将其改为：

```
//////////////////////////////////////////////////////////////////////////
// English (U.S.) resources
```

```
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32
```

并将其中所有的中文翻译为英文。这个 DLL 为我们提供了如图19所示的对话框资源。

图19英文对话框

修改应用工程的 InitInstance()函数，在

```
CResourceDllCallDlg dlg;
m_pMainWnd = &dlg;
int nResponse = dlg.DoModal();
```

之前（即对话框显示之前）添加如下代码：

```
    //获取操作系统的语言
  WORD wLangPID = PRIMARYLANGID( GetSystemDefaultLangID() );
  if( LANG_CHINESE == wLangPID )
  {
      hLanguageDll = LoadLibrary( "ChineseDll.dll" ); //加载中文资源
  }
  else
  {
      hLanguageDll = LoadLibrary( "EnglishDll.dll" );          //加载英文资源
  }

    if( NULL == hLanguageDll )
    {
            AfxMessageBox( "Load  DLL failure" );
            return FALSE;
    }
   AfxSetResourceHandle( hLanguageDll );    //设置当前的资源句柄
```

这样的应用程序将具有自适应性质，在中文 OS 中显示中文资源，在英文 OS 中则显示英文资源。

7.3通信控制 DLL

我们在这里举一个串口通信类的例子。

也许您需要了解一点串口通信的背景知识，其实串口到处都看得到，譬如 PC 机的 COM

口即为串行通讯口（简称串口）。如图20，打开 Windows 的设备管理器，我们看到了 COM 口。

在 Windows 系统，需通过 DCB(Device Control Block)对串口进行配置。利用 Windows API GetCommState 函数可以获取串口当前配置；利用 SetCommState 函数则可以设置串口通讯的参数。

串行通信通常按以下四步进行：

(1)打开串口；

(2)配置串口；

(3)数据传送；

(4)关闭串口。

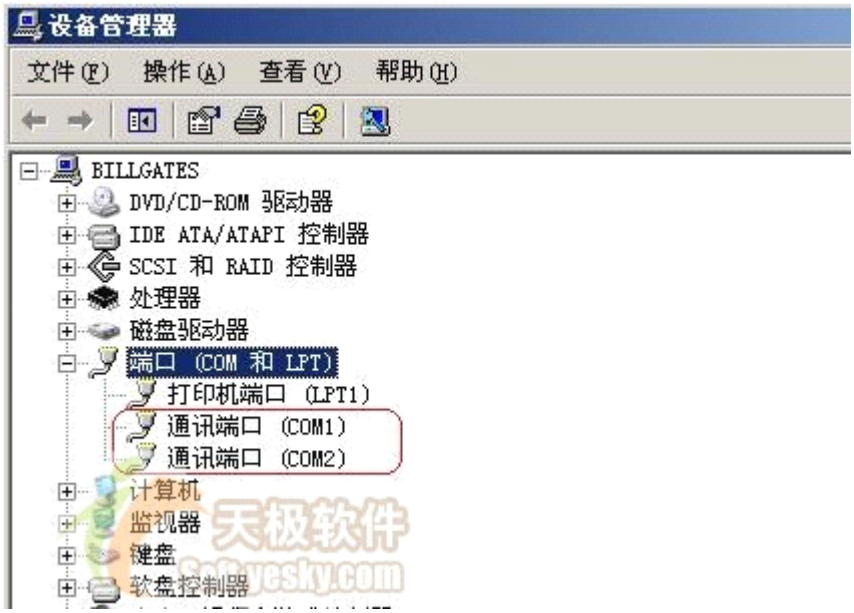

图20 PC 的串口

由此可见，我们需要给串口控制 DLL 提供如下四个接口函数：

```
//打开指定的串口，其参数 port 为端口号
BOOL ComOpen(int port);  //在这个函数里使用默认的参数设置串口

//将打开的串口关闭
void ComClose(int port);

//将串口接收缓冲区中的数据放到 buffer 中
int GetComData(char *buf, int buf_len);

//将指定长度的数据发送到串口
int SendDataToCom(LPBYTE buf,int buf_Len);
```

下面给出了 DLL 接口的主要源代码框架：

```
//com.h：com 类通信接口
class AFX_EXT_CLASS com
{
```

```
public:
        ComOpen(int port)
        {
                …
        }

        int SendDataToCom(LPBYTE buf,int buf_Len)
        {
                …
        }

        int GetComData(char *buf, int buf_len)
        {
                …
        }

        void ComClose()
        {
                …
        }
}
```

我们编写一控制台程序来演示 DLL 的调用：

```
#include <iostream>
#include <exception>
using namespace std;

#include <windows.h>
#include "com.h" //包含 DLL 中导出类的头文件
int main(int argc, char *argv[])
{
        try
        {
                char str[] = "com_class test";
                com com1;
                com1.ComOpen (1);
                for(int i=0; i<100; i++)   //以同步方式写 com 的 buffer
                {
                        Sleep(500);
                        com1.SendDataToCom (str,strlen(str));
                }
                com1.ComClose ();
        }
        catch(exception &e)
        {
```

```
                cout << e.what() << endl;
        }
        return 0;
}
```

DLL 的编写与调用方法及主要应用皆已讲完，在下一节里，我们将看到比较“高深”的主题－－DLL 木马。曾几何时，DLL 木马成为了病毒的一种十分重要的形式，是 DLL 的什么特性使得它能够成为一种病毒？下一节我们将揭晓谜底。

原创 VC++动态链接库(DLL)编程（六）——DLL 木马

2005-10-29 12:23:00

标签：VC++ 编程 DLL [推送到技术圈]

VC++动态链接库(DLL)编程（六）

——DLL 木马

作者： 宋宝华 **e-mail:** 21cnbao@21cn.com

从前文可知，DLL 在程序编制中可作出巨大贡献，它提供了具共性代码的复用能力。但是，正如一门高深的武学，若被掌握在正义之侠的手上，便可助其仗义江湖；但若被掌握在邪恶之徒的手上，则必然在江湖上掀起腥风血雨。DLL 正是一种这样的武学。DLL 一旦染上了魔性，就不再是正常的 DLL 程序，而是 DLL 木马，一种恶贯满盈的病毒，令特洛伊一夜之间国破家亡。

8.1 DLL 木马的原理

DLL 木马的实现原理是编程者在 DLL 中包含木马程序代码，随后在目标主机中选择特定目标进程，以某种方式强行指定该进程调用包含木马程序的 DLL，最终达到侵袭目标系统的目的。

正是 DLL 程序自身的特点决定了以这种形式加载木马不仅可行，而且具有良好的隐藏性：

（1）DLL 程序被映射到宿主进程的地址空间中，它能够共享宿主进程的资源，并根据宿主进程在目标主机的级别非法访问相应的系统资源；

（2）DLL 程序没有独立的进程地址空间，从而可以避免在目标主机中留下“蛛丝马迹”

，达到隐蔽自身的目的。

DLL 木马实现了“真隐藏”，我们在任务管理器中看不到木马“进程”，它完全溶进了系统的内核。与“真隐藏”对应的是“假隐藏”，“假隐藏”木马把自己注册成为一个服务。虽然在任务管理器中也看不到这个进程，但是“假隐藏”木马本质上还具备独立的进程空间。“假隐藏”只适用于 Windows9x 的系统，对于基于 WINNT 的操作系统，通过服务管理器，我们可以发现系统中注册过的服务。

DLL 木马注入其它进程的方法为远程线程插入。

远程线程插入技术指的是通过在另一个进程中创建远程线程的方法进入那个进程的内存地址空间。将木马程序以 DLL 的形式实现后，需要使用插入到目标进程中的远程线程将该木马 DLL 插入到目标进程的地址空间，即利用该线程通过调用 Windows API LoadLibrary 函数来加载木马 DLL，从而实现木马对系统的侵害。

8.2 DLL 木马注入程序

这里涉及到一个非常重要的 Windows API－－CreateRemoteThread。与之相比，我们所习惯使用的 CreateThread API 函数只能在进程自身内部产生一个新的线程，而且被创建的新线程与主线程共享地址空间和其他资源。而 CreateRemoteThread 则不同，它可以在另外的进程中产生线程！CreateRemoteThread 有如下特点：

（1）CreateRemoteThread 较 CreateThread 多一个参数 hProcess，该参数用于指定要创建线程的远程进程，其函数原型为：

```
HANDLE CreateRemoteThread(
  HANDLE hProcess,          //远程进程句柄
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

（2）线程函数的代码不能位于我们用来注入 DLL 木马的进程所在的地址空间中。也就是说，我们不能想当然地自己写一个函数，并把这个函数作为远程线程的入口函数；

（3）不能把本进程的指针作为 CreateRemoteThread 的参数，因为本进程的内存空间与远程进程的不一样。

以下程序由作者 Shotgun 的 DLL 木马注入程序简化而得（在经典书籍《Windows 核心编程》中我们也可以看到类似的例子），它将 d 盘根目录下的 troydll.dll 插入到 ID 为4000 的进程中：

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>

void CheckError ( int, int, char *);          //出错处理函数

PDWORD pdwThreadId;
HANDLE hRemoteThread, hRemoteProcess;
DWORD fdwCreate, dwStackSize, dwRemoteProcessId;
```

```
PWSTR pszLibFileRemote=NULL;

void main(int argc,char **argv)
{
    int iReturnCode;
    char lpDllFullPathName[MAX_PATH];
    WCHAR pszLibFileName[MAX_PATH]={0};

        dwRemoteProcessId = 4000;
        strcpy( lpDllFullPathName, "d:\\troydll.dll");
        //将 DLL 文件全路径的 ANSI 码转换成 UNICODE 码
        iReturnCode = MultiByteToWideChar(CP_ACP, MB_ERR_INVALID_CHARS,
                lpDllFullPathName, strlen( lpDllFullPathName),
                pszLibFileName, MAX_PATH);
        CheckError( iReturnCode, 0, "MultByteToWideChar");
    //打开远程进程
    hRemoteProcess = OpenProcess(PROCESS_CREATE_THREAD | //允许创建线程
                PROCESS_VM_OPERATION | //允许 VM 操作
                PROCESS_VM_WRITE,          //允许 VM 写
                FALSE, dwRemoteProcessId );
    CheckError( (int) hRemoteProcess, NULL,
                "Remote Process not Exist or Access Denied!");
    //计算 DLL 路径名需要的内存空间
    int cb = (1 + lstrlenW(pszLibFileName)) * sizeof(WCHAR);
    pszLibFileRemote = (PWSTR) VirtualAllocEx( hRemoteProcess, NULL, cb,
                MEM_COMMIT, PAGE_READWRITE);
    CheckError((int)pszLibFileRemote, NULL, "VirtualAllocEx");
    //将 DLL 的路径名复制到远程进程的内存空间
    iReturnCode = WriteProcessMemory(hRemoteProcess,
          pszLibFileRemote, (PVOID) pszLibFileName, cb, NULL);
    CheckError( iReturnCode, false, "WriteProcessMemory");
    //计算 LoadLibraryW 的入口地址
    PTHREAD_START_ROUTINE pfnStartAddr = (PTHREAD_START_ROUTINE)
          GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");
    CheckError((int)pfnStartAddr, NULL, "GetProcAddress");
    //启动远程线程，通过远程线程调用用户的 DLL 文件
    hRemoteThread = CreateRemoteThread( hRemoteProcess, NULL, 0, pfnStartAddr,
pszLibFileRemote, 0, NULL);
    CheckError((int)hRemoteThread, NULL, "Create Remote Thread");
    //等待远程线程退出
    WaitForSingleObject(hRemoteThread, INFINITE);
    //清场处理
    if (pszLibFileRemote != NULL)
        {
```

```
            VirtualFreeEx(hRemoteProcess, pszLibFileRemote, 0, MEM_RELEASE);
        }
    if (hRemoteThread != NULL)
        {
            CloseHandle(hRemoteThread );
        }
    if (hRemoteProcess!= NULL)
        {
            CloseHandle(hRemoteProcess);
        }
}

//错误处理函数 CheckError()
void CheckError(int iReturnCode, int iErrorCode, char *pErrorMsg)
{
    if(iReturnCode==iErrorCode)
        {
         printf("%s Error:%d\n\n", pErrorMsg, GetLastError());
         //清场处理
         if (pszLibFileRemote != NULL)
                {   VirtualFreeEx(hRemoteProcess, pszLibFileRemote, 0,
MEM_RELEASE);
         }
                if (hRemoteThread != NULL)
                {
                        CloseHandle(hRemoteThread );
                }
         if (hRemoteProcess!= NULL)
                {
                        CloseHandle(hRemoteProcess);
                }
         exit(0);
    }
}
```

从 DLL 木马注入程序的源代码中我们可以分析出 DLL 木马注入的一般步骤为：

（1）取得宿主进程（即要注入木马的进程）的进程 ID dwRemoteProcessId；

（2）取得 DLL 的完全路径，并将其转换为宽字符模式 pszLibFileName；

（3）利用 Windows API OpenProcess 打开宿主进程，应该开启下列选项：

a.PROCESS_CREATE_THREAD：允许在宿主进程中创建线程；

b.PROCESS_VM_OPERATION：允许对宿主进程中进行 VM 操作；

c.PROCESS_VM_WRITE：允许对宿主进程进行 VM 写。

（4）利用 Windows API VirtualAllocEx 函数在远程线程的 VM 中分配 DLL 完整路径宽字符所需的存储空间，并利用 Windows API WriteProcessMemory 函数将完整路径写入该存储空间；

（5）利用 Windows API GetProcAddress 取得 Kernel32 模块中 LoadLibraryW 函数的地址，这个函数将作为随后将启动的远程线程的入口函数；

（6）利用 Windows API CreateRemoteThread 启动远程线程，将 LoadLibraryW 的地址作为远程线程的入口函数地址，将宿主进程里被分配空间中存储的完整 DLL 路径作为线程入口函数的参数以另其启动指定的 DLL；

（7）清理现场。

8.3 DLL 木马的防治

从 DLL 木马的原理和一个简单的 DLL 木马程序中我们学到了 DLL 木马的工作方式，这可以帮助我们更好地理解 DLL 木马病毒的防治手段。

一般的木马被植入后要打开一网络端口与攻击程序通信，所以防火墙是抵御木马攻击的最好方法。防火墙可以进行数据包过滤检查，我们可以让防火墙对通讯端口进行限制，只允许系统接受几个特定端口的数据请求。这样，即使木马植入成功，攻击者也无法进入到受侵系统，防火墙把攻击者和木马分隔开来了。

对于 DLL 木马，一种简单的观察方法也许可以帮助用户发现之。我们查看运行进程所依赖的 DLL，如果其中有一些莫名其妙的 DLL，则可以断言这个进程是宿主进程，系统被植入了 DLL 木马。“道高一尺，魔高一丈”，现如今，DLL 木马也发展到了更高的境界，它们看起来也不再“莫名其妙”。在最新的一些木马里面，开始采用了先进的 DLL 陷阱技术，编程者用特洛伊 DLL 替换已知的系统 DLL。特洛伊 DLL 对所有的函数调用进行过滤，对于正常的调用，使用函数转发器直接转发给被替换的系统 DLL；对于一些事先约定好的特殊情况，DLL 会执行一些相应的操作。

本文给出的只是 DLL 木马最简单情况的介绍，读者若有兴趣深入研究，可以参考其它资料。

此后将是本系列文章的最后一次连载，即读者来信与反馈。

原创 VC++动态链接库(DLL)编程（七）——读者反馈与答复

2005-10-29 12:27:00

标签：VC++ 编程 DLL [推送到技术圈]

VC++动态链接库(DLL)编程（七）

——读者反馈与答复

作者： 宋宝华 **e-mail:** 21cnbao@21cn.com

1.关于文章的获取

许多读者发来 e-mail 询问本系列文章的相关事宜，如:

（1） 是否已出版?

（2） 哪里可以下载打包版?

（3） 哪里可以下载笔者的其它文章?

还有一些读者对日前笔者在天极网发表的《C 语言嵌入式系统编程修炼之道》非常喜爱,给予了热情洋溢的赞扬，询问笔者能否继续创作嵌入式编程方面的文章。

对于这些问题，统一作答如下：

（1）本系列文章暂时尚未出版；

（2）您可以在天极网或 pconline 软件频道下载笔者的多数拙作。另外，我也将不定期将这些文章上传到我的博客（[url]http://blog.donews.com/21cnbao/[/url]）。所有文章中的例程源代码均经过亲手调试，验证无误；

（3）就嵌入式系统开发，笔者将继续进行此方面的创作，新近将推出《基于嵌入式实时 OS VxWorks 的多任务程序设计》及《领悟：从 Windows 多线程到 VxWorks 的多任务》。

非常感谢读者朋友对这些文章的喜爱，在下将竭尽所能地为您提供更多的好文章。

2.关于 DLL 的疑问

你好，看了你写的“VC++动态链接库(DLL)编程深入浅出”，特别有收获。 只是有个地方我老搞不明白，就是用 DLL 导出全局变量时，指定了.lib 的路径（#pragma comment(lib,"dllTest.lib")），那么.dll 的文件的路径呢，我尝试着把.dll 文件移到别的地方程序就无法正常运行了，请问.dll 在这里怎么指定。

希望您能在百忙中抽空给我解答一下，不胜感激!

一位编程爱好者

回答：

Windows 按下列顺序搜索 DLL：

（1）当前进程的可执行模块所在的目录；

（2）当前目录；

（3）Windows 系统目录，通过 GetSystemDirectory 函数可获得此目录的路径；

（4）Windows 目录，通过 GetWindowsDirectory 函数可获得此目录的路径；

（5）PATH 环境变量中列出的目录。

因此，隐式链接时，DLL 文件的路径不需要指定也不能指定，系统指定按照1～5的步骤寻找 DLL，但是对应的.lib 文件却需要指定路径；如果使用 Windows API 函数 LoadLibrary 动态加载 DLL，则可以指定 DLL 的路径。

你好,我是一位 C++初学者,我在 PCONLINE 看了教学之后,受益不浅。我想问一下能否在 DLL 里使用多线程?MSDN 上用#using <mscorlib.dll>这个指令之后实现了多线程,不过好象不支持 DLL..

请问有什么办法支持制作多线程 DLL??能否给一个源码来?

回答：

在 DLL 中可以处理多线程，WIN32对于多线程的支持是操作系统本身提供的一种能力，并不在于用户编写的是哪一类程序。即便是一个控制台程序，我们都可以使用多线程：

```
#include <stdio.h>
```

```
#include <windows.h>
void ThreadFun(void)
{
	while(1)
	{
		printf( "this is new thread\n" );
		Sleep( 1000 );
	}
}
int main()
{
	DWORD threadID;
	CreateThread( NULL, 0, (LPTHREAD_START_ROUTINE)ThreadFun,
		NULL, 0, &threadID );
	while(1)
	{
		printf( "this is main thread\n" );
		Sleep( 1000 );
	}
}
```

观察程序运行的结果为在控制台窗口上交替输出 this is main thread、this is new thread。

我们来看下面的一个多线程 DLL 的例子。

DLL 程序提供一个接口函数 SendInit，在此接口中启动发送线程 SendThreadFunc，在这个线程的对应工作函数中我们使用原始套接字 socket 发送报文。参考微软出版的经典书籍《Windows 核心编程》，我们发现，不宜在 DLL 被加载的时候（即进程绑定时）启动一个新的线程。

这个线程等待一个 CEvent 事件（用于线程间通信），应用程序调用 DLL 中的接口函数 SendMsg(InterDataPkt sendData)可以释放此事件。下面是相关的源代码：

（1）发送报文线程入口函数

```
//////////////////////////////////////////////////////////////////////////
//函数名：SendThreadFunc
//函数功能：发送报文工作线程入口函数，使用 UDP 协议
//////////////////////////////////////////////////////////////////////////
DWORD WINAPI SendThreadFunc( LPVOID lpvThreadParm )
//提示：对于线程函数应使用 WINAPI 声明，WINAPI 被宏定义为__stdcall
{
	/*创建 socket */
	sendSock = socket ( AF_INET, SOCK_DGRAM, 0 );
	if ( sendSock == INVALID_SOCKET )
	{
		AfxMessageBox ( "Socket 创建失败" );
		closesocket ( recvSock );
	}
```

```
        /*获得目标节点端口与地址*/
        struct sockaddr_in desAddr;
        desAddr.sin_family=AF_INET;
        desAddr.sin_port=htons( DES_RECV_PORT );     //目标节点接收端口
        desAddr.sin_addr.s_addr = inet_addr( DES_IP );

  /*发送数据*/
    while(1)
        {
                WaitForSingleObject( hSendEvent, 0xffffffffL );//无限等待事件发生
                ResetEvent( hSendEvent );

                sendto( sendSock, (char *)sendSockData.data, sendSockData.len,
                        0, (struct sockaddr*)&desAddr, sizeof(desAddr) );
        }

        return -1;
}
```

（2）MFC 规则 DLL 的 InitInstance 函数

```
////////////////////////////////////////////////////////////////////////////
// CMultiThreadDllApp initialization
BOOL CMultiThreadDllApp::InitInstance()
{
        if ( !AfxSocketInit() ) //初始化 socket
        {
                AfxMessageBox( IDP_SOCKETS_INIT_FAILED );
                return FALSE;
        }

        return TRUE;
}
```

（3）启动发送线程

```
////////////////////////////////////////////////////////////////////////////
//函数名：SendInit
//函数功能：DLL 提供给应用程序调用接口，用于启动发送线程
////////////////////////////////////////////////////////////////////////////
void SendInit(void)
{
        hSendThread = CreateThread( NULL, 1000, SendThreadFunc,
                                this, 1, &uSendThreadID );
}
```

（4）SendMsg 函数

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

```
//函数名：SendMsg
//函数功能：DLL 提供给应用程序调用接口，用于发送报文
////////////////////////////////////////////////////////////////////////
extern "C" void WINAPI SendMsg( InterDataPkt sendData )
{
        sendSockData = sendData;
        SetEvent( hSendEvent );     //释放发送事件
}
```

以上程序仅仅是一个简单的例子，其实在许多工程应用中，我们经常看到这样的处理方式。这个 DLL 对用户而言仅仅使一个简单的接口函数 SendMsg，对调用它的应用程序屏蔽了多线程的技术细节。与之类似，MFC 提供的 CSocket 类在底层自己采用了多线程机制，所以使我们免去了对多线程的使用。

您好,看了您的 DLL 文章，发现导出函数可以直接用_declspec(dllexport)声明或在.def 文件中定义，变量的导出也一样。我想知道类是否也可以在.def 文件中导出？您的文章中只讲了在类前添加_declspec(dllexport)导出类的方法。请您指教!

回答：

一般我们不采用.def 文件导出类，但是这并不意味着类不能用.def 文件导出类。

使用 Depends 查看连载2的“导出类”例程生成的 DLL，我们发现其导出了如图21的众多“怪”symbol，这些 symbol 都是经过编译器处理的。因此，为了以.def 文件导出类，我们必须把这些“怪”symbol 全部导出，实在是不划算啊！所以对于类，我们最好直接以_declspec(dllexport)导出。

Ordinal ^	Hint	Function	Entry Point
1 (0x0001)	0 (0x0000)	??0circle@@QAE@XZ	0x0000101E
2 (0x0002)	1 (0x0001)	??0point@@QAE@MM@Z	0x00001032
3 (0x0003)	2 (0x0002)	??0point@@QAE@XZ	0x0000100F
4 (0x0004)	3 (0x0003)	??4circle@@QAEAAV0@ABV0@@Z	0x0000102D
5 (0x0005)	4 (0x0004)	??4point@@QAEAAV0@ABV0@@Z	0x00001023
6 (0x0006)	5 (0x0005)	?GetArea@circle@@QAEMXZ	0x0000100A
7 (0x0007)	6 (0x0006)	?GetGirth@circle@@QAEMXZ	0x00001014
8 (0x0008)	7 (0x0007)	?SetCentre@circle@@QAEXABVpoint@@@Z	0x00001028
9 (0x0009)	8 (0x0008)	?SetRadius@circle@@QAEXM@Z	0x00001019

图21导出类时导出的 symbol

您好,看了您的 DLL 文章，知道怎么创建 DLL 了，但是面对一个具体的工程，我还是不知道究竟应该把什么做成 DLL？您能给一些这方面的经验吗?

回答：

DLL 一般用于软件模块中较固定、较通用的可以被复用的模块，这里有一个非常好的例子，就是豪杰超级解霸。梁肇新大师把处理视频和音频的算法模块专门做成了两个 DLL，供超级解霸的用户界面 GUI 程序调用，实在是 DLL 设计的模范教程。所谓“万变不离其宗”，超级解霸的界面再 cool，用到的还是那几个 DLL！具体请参考《编程高手箴言》一书。

您好,您的 DLL 文章讲的都是 Windows 的，请问 Linux 操作系统上可以制作 DLL 吗？如

果能，和 Windows 有什么不一样？谢谢!

回答：

在 Linux 操作系统中，也可以采用动态链接技术进行软件设计，但与 Windows 下 DLL 的创建和调用方式有些不同。

Linux 操作系统中的共享对象技术（Shared Object）与 Windows 里的 DLL 相对应，但名称不一样，其共享对象文件以.so 作为后缀。与 Linux 共享对象技术相关的一些函数如下：

(1)打开共享对象，函数原型：

```
//打开名为 filename 共享对象，并返回操作句柄；
void *dlopen (const char *filename, int flag);
```

(2)取函数地址，函数原型:

```
//获得接口函数地址
void *dlsym(void *handle, char *symbol);
```

(3)关闭共享对象，函数原型:

```
//关闭指定句柄的共享对象
int dlclose (void *handle);
```

(4)动态库错误函数，函数原型:

```
//共享对象操作函数执行失败时，返回出错信息
const char *dlerror(void);
```

从这里我们分明看到 Windows API——LoadLibrary、FreeLibrary 和 GetProcAddress 的影子！又一个“万变不离其宗”！

本系列文章的连载暂时告一段落，您可以继续给笔者发送 email（mailto: 21cnbao@21cn.com）讨论 DLL 的编程问题。对于文中的错误和纰漏，也热诚欢迎您指正。