部分 1_ COM 原理

哈尔滨顺时针电脑学校 于凯

01_01_DLL 基础

哈尔滨顺时针电脑学校
地址：哈尔滨市南岗区复兴街 16 号
电话：0451-86220686 86228969 86220769
网址：http://www.sszkj.com
EMAIL: sszkj@tom.com

DLL(Dynamic Link Libraries)专题

目录

引言

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

Windows 系统平台上提供了一种完全不同的较有效的编程和运行环境，你可以将独立的程序模块创建为较小的 DLL(Dynamic Linkable Library)文件，并可对它们单独编译和测试。在运行时，只有当 EXE 程

序确实要调用这些 DLL 模块的情况下，系统才会将它们装载到内存空间中。这种方式不仅减少了 EXE 文件的大小和对内存空间的需求，而且使这些 DLL 模块可以同时被多个应用程序使用。Windows 自己就将一些主要的系统功能以 DLL 模块的形式实现。

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

在 Win32 环境中，每个进程都复制了自己的读/写全局变量。如果想要与其它进程共享内存，必须使用内存映射文件或者声明一个共享数据段。DLL 模块需要的堆栈内存都是从运行进程的堆栈中分配出来的。Windows 在加载 DLL 模块时将进程函数调用与 DLL 文件的导出函数相匹配。Windows 操作系统对 DLL 的操作仅仅是把 DLL 映射到需要它的进程的虚拟地址空间里去。DLL 函数中的代码所创建的任何对象（包括变量）都归调用它的线程或进程所有.

调用方式

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

隐式的调用：需要把产生动态连接库时产生的.LIB 文件加入到应用程序的工程中，想使用 DLL 中的函数时，只须说明一下。隐式调用不需要调用 LoadLibrary()和 FreeLibrary()。程序员在建立一个 DLL 文件时，链接程序会自动生成一个与之对应的 LIB 导入文件。该文件包含了每一个 DLL 导出函数的符号名和可选的标识号，但是并不含有实际的代码。LIB 文件作为 DLL 的替代文件被编译到应用程序项目中。

当程序员通过静态链接方式编译生成应用程序时，应用程序中的调用函数与 LIB 文件中导出符号相匹配，这些符号或标识号进入到生成的 EXE 文件中。LIB 文件中也包含了对应的 DLL 文件名（但不是完全的路径名），链接程序将其存储在 EXE 文件内部。

当应用程序运行过程中需要加载 DLL 文件时，Windows 根据这些信息发现并加载 DLL，然后通过符号名或标识号实现对 DLL 函数的动态链接。所有被应用程序调用的 DLL 文件都会在应用程序 EXE 文件加载时被加载在到内存中。可执行程序链接到一个包含 DLL 输出函数信息的输入库文件(.LIB 文件)。操作系统在加载使用可执行程序时加载 DLL。可执行程序直接通过函数名调用 DLL 的输出函数，调用方法和程序内部其他的函数是一样的。

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

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

Windows 将遵循下面的搜索顺序来定位 DLL：

1．包含 EXE 文件的目录

2．进程的当前工作目录

3．Windows 系统目录

4．Windows 目录

5．列在 Path 环境变量中的一系列目录

MFC 中的 DLL

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

b、Regular DLL:和下述的 Extension DLLs 一样，是用 MFC 类库编写的。明显的特点是在源文件里有一个继承 CWinApp 的类。其又可细分成静态连接到 MFC 和动态连接到 MFC 上的。

静态连接到 MFC 的动态连接库只被 VC 的专业般和企业版所支持。该类 DLL 应用程序里头的输出函数可以被任意 Win32 程序使用，包括使用 MFC 的应用程序。输入函数有如下形式：

extern "C" EXPORT YourExportedFunction( );

如果没有 extern "C"修饰，输出函数仅仅能从 C++代码中调用。

DLL 应用程序从 CWinApp 派生，但没有消息循环。

动态链接到 MFC 的规则 DLL 应用程序里头的输出函数可以被任意 Win32 程序使用，包括使用 MFC 的应用程序。但是，所有从 DLL 输出的函数应该以如下语句开始：

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

此语句用来正确地切换 MFC 模块状态。

Regular DLL 能够被所有支持 DLL 技术的语言所编写的应用程序所调用。在这种动态连接库中，它必须有一个从 CWinApp 继承下来的类，DLLMain 函数被 MFC 所提供，不用自己显式的写出来。

c、Extension DLL:用来实现从 MFC 所继承下来的类的重新利用，也就是说，用这种类型的动态连接库，可以用来输出一个从 MFC 所继承下来的类。它输出的函数仅可以被使用 MFC 且动态链接到 MFC 的应用程序使用。可以从 MFC 继承你所想要的、更适于你自己用的类，并把它提供给你的应用程序。你也可随意的给你的应用程序提供 MFC 或 MFC 继承类的对象指针。Extension DLL 使用 MFC 的动态连接版本所创建的，并且它只被用 MFC 类库所编写的应用程序所调用。Extension DLLs 和 Regular DLLs 不一样，它没有一个从 CWinApp 继承而来的类的对象，所以，你必须为自己 DLLMain 函数添加初始化代码和结束代码。

和规则 DLL 相比，有以下不同：

1、它没有一个从 CWinApp 派生的对象；

2、它必须有一个 DLLMain 函数；

3、DLLMain 调用 AfxInitExtensionModule 函数，必须检查该函数的返回值，如果返回 0，DLLMmain 也返回 0；

4、如果它希望输出 CRuntimeClass 类型的对象或者资源(Resources)，则需要提供一个初始化函数来创建一个 CDynLinkLibrary 对象。并且，有必要把初始化函数输出；

5、使用扩展 DLL 的 MFC 应用程序必须有一个从 CWinApp 派生的类，而且，一般在 InitInstance 里调用扩展 DLL 的初始化函数。

DLL 入口函数

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

DLLMain 的函数原型：

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

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

参数：

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

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

DLL_PROCESS_ATTACH：进程被调用；

DLL_THREAD_ATTACH：线程被调用；

DLL_PROCESS_DETACH：进程被停止；

DLL_THREAD_DETACH：线程被停止；

lpReserved：是一个被系统所保留的参数。

2、_DLLMainCRTStartup

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

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

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

关于约定

动态库输出函数的约定有两种：调用约定和名字修饰约定。

1)调用约定(Calling convention)：决定函数参数传送时入栈和出栈的顺序，由调用者还是被调用者把参数弹出栈，以及编译器用来识别函数名字的修饰约定。

函数调用约定有多种，这里简单说一下：

1、__stdcall 调用约定相当于 16 位动态库中经常使用的 PASCAL 调用约定。在 32 位的 VC++5.0 中 PASCAL 调用约定不再被支持（实际上它已被定义为__stdcall。除了__pascal 外，__fortran 和__syscall 也不被支持），取而代之的是__stdcall 调用约定。两者实质上是一致的，即函数的参数自右向左

通过栈传递，被调用的函数在返回前清理传送参数的内存栈，但不同的是函数名的修饰部分（关于函数名的修饰部分在后面将详细说明）。

_stdcall 是 Pascal 程序的缺省调用方式，通常用于 Win32 Api 中，函数采用从右到左的压栈方式，自己在退出时清空堆栈。VC 将函数编译后会在函数名前面加上下划线前缀，在函数名后加上"@"和参数的字节数。

2、C 调用约定（即用__cdecl 关键字说明）按从右至左的顺序压参数入栈，由调用者把参数弹出栈。对于传送参数的内存栈是由调用者来维护的（正因为如此，实现可变参数的函数只能使用该调用约定）。另外，在函数名修饰约定方面也有所不同。

_cdecl 是 C 和 C＋＋程序的缺省调用方式。每一个调用它的函数都包含清空堆栈的代码，所以产生的可执行文件大小会比调用_stdcall 函数的大。函数采用从右到左的压栈方式。VC 将函数编译后会在函数名前面加上下划线前缀。是 MFC 缺省调用约定。

3、__fastcall 调用约定是"人"如其名，它的主要特点就是快，因为它是通过寄存器来传送参数的（实际上，它用 ECX 和 EDX 传送前两个双字（DWORD）或更小的参数，剩下的参数仍旧自右向左压栈传送，被调用的函数在返回前清理传送参数的内存栈），在函数名修饰约定方面，它和前两者均不同。

_fastcall 方式的函数采用寄存器传递参数，VC 将函数编译后会在函数名前面加上"@"前缀，在函数名后加上"@"和参数的字节数。

4、thiscall 仅仅应用于"C++"成员函数。this 指针存放于 CX 寄存器，参数从右到左压。thiscall 不是关键词，因此不能被程序员指定。

5、naked call 采用 1-4 的调用约定时，如果必要的话，进入函数时编译器会产生代码来保存 ESI，EDI，EBX，EBP 寄存器，退出函数时则产生代码恢复这些寄存器的内容。

naked call 不产生这样的代码。naked call 不是类型修饰符，故必须和_declspec 共同使用。

关键字 __stdcall、__cdecl 和__fastcall 可以直接加在要输出的函数前，也可以在编译环境的 Setting... \C/C++ \Code Generation 项选择。当加在输出函数前的关键字与编译环境中的选择不同时，直接加在输出函数前的关键字有效。它们对应的命令行参数分别为/Gz、/Gd 和/Gr。缺省状态为/Gd，即__ cdecl。

要完全模仿 PASCAL 调用约定首先必须使用__stdcall 调用约定，至于函数名修饰约定，可以通过其它方法模仿。还有一个值得一提的是 WINAPI 宏，Windows.h 支持该宏，它可以将出函数翻译成适当的调用约定，在 WIN32 中，它被定义为__stdcall。使用 WINAPI 宏可以创建自己的 APIs。

2)名字修饰约定

1、修饰名(Decoration name)

"C"或者"C++"函数在内部（编译和链接）通过修饰名识别。修饰名是编译器在编译函数定义或者原型时生成的字符串。有些情况下使用函数的修饰名是必要的，如在模块定义文件里头指定输出"C++"重载函数、构造函数、析构函数，又如在汇编代码里调

用"C""或"C++"函数等。

修饰名由函数名、类名、调用约定、返回类型、参数等共同决定。

2、名字修饰约定随调用约定和编译种类(C 或 C++)的不同而变化。函数名修饰约定随编译种类和调用约定的不同而不同，下面分别说明。

a、C 编译时函数名修饰约定规则：

__stdcall 调用约定在输出函数名前加上一个下划线前缀，后面加上一个"@"符号和其参数的字节数，格式为_functionname@number。

__cdecl 调用约定仅在输出函数名前加上一个下划线前缀，格式为_functionname。

__fastcall 调用约定在输出函数名前加上一个"@"符号，后面也是一个"@"符号和其参数的字节数，格式为@functionname@number。

它们均不改变输出函数名中的字符大小写，这和 PASCAL 调用约定不同，PASCAL 约定输出的函数名无任何修饰且全部大写。

b、C++编译时函数名修饰约定规则：

__stdcall 调用约定：

1、以"?"标识函数名的开始，后跟函数名；

2、函数名后面以"@@YG"标识参数表的开始，后跟参数表；

3、参数表以代号表示：

X--void ，

D--char，

E--unsigned char，

F--short，

H--int，

I--unsigned int，

J--long，

K--unsigned long，

M--float，

N--double，

_N--bool，

....

PA--表示指针，后面的代号表明指针类型，如果相同类型的指针连续出现，以"0"代替，一个"0"代表一次重复；

4、参数表的第一项为该函数的返回值类型，其后依次为参数的数据类型,指针标识在其所指数据类型前；

5、参数表后以"@Z"标识整个名字的结束，如果该函数无参数，则以"Z"标识结束。

其格式为"?functionname@@YG*****@Z"或"?functionname@@YG*XZ"，例如

```
int Test1（char *var1,unsigned long）-----“?Test1@@YGHPADK@Z”
void Test2（）                    -----“?Test2@@YGXXZ”
```

__cdecl 调用约定：

规则同上面的_stdcall 调用约定，只是参数表的开始标识由上面的"@@YG"变为"@@YA"。

__fastcall 调用约定：

规则同上面的_stdcall 调用约定，只是参数表的开始标识由上面的"@@YG"变为"@@YI"。

VC++对函数的省缺声明是"__cedcl",将只能被 C/C++调用.

关于 DLL 的函数

动态链接库中定义有两种函数：导出函数(export function)和内部函数(internal function)。导出函数可以被其它模块调用，内部函数在定义它们的 DLL 程序内部使用。

输出函数的方法有以下几种：

1、传统的方法

在模块定义文件的 EXPORT 部分指定要输入的函数或者变量。语法格式如下：

```
entryname[=internalname] [@ordinal[NONAME]] [DATA] [PRIVATE]
```

其中：

entryname 是输出的函数或者数据被引用的名称；

internalname 同 entryname;

@ordinal 表示在输出表中的顺序号(index)；

NONAME 仅仅在按顺序号输出时被使用（不使用 entryname）；

DATA 表示输出的是数据项，使用 DLL 输出数据的程序必须声明该数据项为_declspec(DLLimport)。

上述各项中，只有 entryname 项是必须的，其他可以省略。

对于"C"函数来说，entryname 可以等同于函数名；但是对"C++"函数（成员函数、非成员函数）来说，entryname 是修饰名。可以从.map 映像文件中得到要输出函数的修饰名，或者使用 DUMPBIN /SYMBOLS 得到，然后把它们写在.def 文件的输出模块。DUMPBIN 是 VC 提供的一个工具。

如果要输出一个"C++"类，则把要输出的数据和成员的修饰名都写入.def 模块定义文件。

2、在命令行输出

对链接程序 LINK 指定/EXPORT 命令行参数，输出有关函数。

3、使用 MFC 提供的修饰符号_declspec(DLLexport)

在要输出的函数、类、数据的声明前加上_declspec(DLLexport)的修饰符，表示输出。__declspec(DLLexport)在 C 调用约定、C 编译情况下可以去掉输出函数名的下划线前缀。extern "C"使得在 C++中使用 C 编译方式成为可能。在"C++"下定义"C"函数，

需要加 extern "C"关键词。用 extern "C"来指明该函数使用 C 编译方式。输出的"C"函数可以从"C"代码里调用。

例如，在一个 C++文件中，有如下函数：

```
   extern "C" {void  __declspec(DLLexport) __cdecl Test(int var);}
```

其输出函数名为：Test

MFC 提供了一些宏，就有这样的作用。

```
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
```

像 AFX_EXT_CLASS 这样的宏，如果用于 DLL 应用程序的实现中，则表示输出（因为_AFX_EXT 被定义，通常是在编译器的标识参数中指定该选项/D_AFX_EXT）；如果用于使用 DLL 的应用程序中，则表示输入（_AFX_EXT 没有定义）。

要输出整个的类，对类使用_declspec(_DLLexpot)；要输出类的成员函数，则对该函数使用_declspec(_DLLexport)。如：

```
class AFX_EXT_CLASS CTextDoc : public CDocument
{
        …
}

extern "C" AFX_EXT_API void WINAPI InitMYDLL();
```

这几种方法中，最好采用第三种，方便好用；其次是第一种，如果按顺序号输出，调用效率会高些；最次是第二种。

模块定义文件(.DEF)

模块定义文件(.DEF)是一个或多个用于描述 DLL 属性的模块语句组成的文本文件，每个 DEF 文件至少必须包含以下模块定义语句：

* 第一个语句必须是 LIBRARY 语句，指出 DLL 的名字；

* EXPORTS 语句列出被导出函数的名字；将要输出的函数修饰名罗列在 EXPORTS 之下，这个名字必须与定义函数的名字完全一致，如此就得到一个没有任何修饰的函数名了。

* 可以使用 DESCRIPTION 语句描述 DLL 的用途(此句可选)；

* ";"对一行进行注释(可选)。

DLL 程序和调用其输出函数的程序的关系

1、DLL 与进程、线程之间的关系

DLL 模块被映射到调用它的进程的虚拟地址空间。

DLL 使用的内存从调用进程的虚拟地址空间分配，只能被该进程的线程所访问。

DLL 的句柄可以被调用进程使用；调用进程的句柄可以被 DLL 使用。

DLL 使用调用进程的栈。

2、关于共享数据段

DLL 定义的全局变量可以被调用进程访问；DLL 可以访问调用进程的全局数据。使用同一 DLL 的每一个进程都有自己的 DLL 全局变量实例。如果多个线程并发访问同一变量，则需要使用同步机制；对一个 DLL 的变量，如果希望每个使用 DLL 的线程都有自己的值，则应该使用线程局部存储(TLS，Thread Local Strorage)。

在程序里加入预编译指令，或在开发环境的项目设置里也可以达到设置数据段属性的目的.必须给这些变量赋初值,否则编译器会把没有赋初始值的变量放在一个叫未被初始化的数据段中。

Adll 与 lib

目标：写几个比较简单的 dll 并了解**.dll 与**.lib 的关系。

一：没有 lib 的 dll

1.1 建一个没有 lib 的 dll

- 1） 新建一个 WIN32 工程→空项目，加入一个 com_1.cpp 文件（注意此 dll 根本没有什么用）
- 2） 在 com_1.cpp 写下下面的代码
- 3） 按下 F5 运行，所有的东西都按确定。
- 4） 应该出现如下错误：

```
  Linking...
     Creating library Debug/COM_1.lib and object Debug/COM_1.exp
  LIBCD.lib(crt0.obj) : error LNK2001: unresolved external
symbol _main
Debug/COM_1.exe : fatal error LNK1120: 1 unresolved externals
```

- 5)进入工程属性，在 "C/C++" 属性框的 "预处理器→预处理器定义" 里把 "_console" 修改成"_WINDOWS”。 或者在命令行中加入“/D"_WINDOWS"”
- 6)在 "链接器" 属性框的 "命令行" 里增加下面的编译开关 "/dll "
- 7）把"链接器" 属性框的 "常规" 中的输出文件改为.dll

增加的编译开关大致如下：

```
kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib
advapi32.lib shell32.lib
ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib /nologo /dll
/incremental:yes
/pdb:"Debug/COM_1.pdb" /debug /machine:I386 /out:"Debug/COM_1.dll"
/implib:"Debug/COM_1.lib"
/pdbtype:sept
```

注意："/dll"应该与后面的开关之间有一个空格

```
//com_1.cpp
#include <objbase.h>
BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, void*
lpReserved)
{
                HANDLE g_hModule;
        switch(dwReason)
        {
```

```
        case DLL_PROCESS_ATTACH:
                g_hModule = (HINSTANCE)hModule;
                break;
        case DLL_PROCESS_DETACH:
                 g_hModule=NULL;
                 break;
        }
}
```

现在可以编译了，这小片段代码将会生成一个 dll,但这个 dll 是没有用的。没有引出函数和变量。

1.2 调试没有 lib 的 dll

1) 新建一个工程 Client，工程类型为 console，将上面创建的 dll copy 到 client 工程目录下
2) 增加 Client.cpp（代码见下）到工程 Client 中去
3) 选中 Client 工程，并在 project|setting|debug|Category 下拉框，如图：

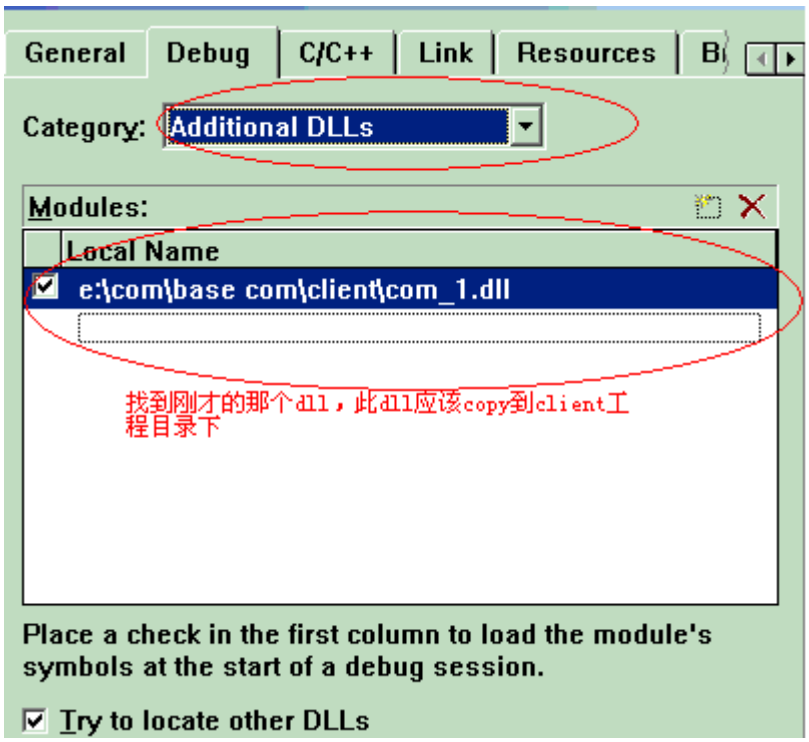

图 1.4 调试

注意这是一种调试 dll 的方法

5） 现在可以在 Client 和 COM_1.dll 里打断点调试了。
在这里我们只能调试 DllMain()函数，因为那个 dll 里除了就没别的东西了，下面我开始 增加一点东西。

二：带有 lib 的 dll

2.1 创建一个带有 lib 的 dll

我们在原来的基础上让上面的代码产生一个 lib 了。新的代码如下：

```
#include <objbase.h>

extern "C" __declspec(dllexport)  void tulip (void)
{
        ::MessageBox(NULL,"ok","I''am fine",MB_OK);
}

BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, void*
lpReserved)
{
        HANDLE g_hModule;
        switch(dwReason)
        {
        case DLL_PROCESS_ATTACH:
                g_hModule = (HINSTANCE)hModule;
                break;
        case DLL_PROCESS_DETACH:
                 g_hModule=NULL;
                 break;
        }

        return TRUE;
}
```

在这个 dll 里，我们引出一个 tulip 函数。如果此时我们想要在客户调用此函数应该用什么方法呢？

上面的代码除了生成 dll 外，他比第一个程序多产生一个 lib 文件，现在应该知道 dll 与 lib 的关系吧。Lib 文件是 dll 输出符号文件。如果一个 dll 没有任何东西输出那么不会有对应的 lib 文件，但只要一个 dll 输出一个变量或函数就会相应的 lib 文件。总的说来，dll 与 lib 是相互配套的。
当某个 dll 他有输出函数（或变量）而没有 lib 文件时，我们应该怎么调用 dll 的函数呢？请看下面的方法。

2.2 调试带有引用但没有头文件的 dll

注意：本方法根本没有用 COM_1.lib 文件，你可以把 COM_1.lib 文件删除而不影响。
此时的客户端代码如果下：

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

```
int main(void)
{
	//定义一个函数指针
	typedef void (  * TULIPFUNC )(void);

	//定义一个函数指针变量
	TULIPFUNC tulipFunc;

	//加载我们的 dll
	HINSTANCE hinst=::LoadLibrary("COM_1.dll");

	//找到 dll 的 tulip 函数
	tulipFunc=(TULIPFUNC)GetProcAddress(hinst,"tulip");

	//调用 dll 里的函数
	tulipFunc();

	return 0;
}
```

对于调用系统函数用上面的方法非常方便，因为对于 User32.dll，GUI32.dll 这种 dll,我没有对应的 lib，所以一般用上面的方法。

三：带有头文件的 dll

3.1 创建一个带有引出信息头文件的 dll

如果用上面的方法调用我们自己创建的 dll 那太烦了！因为我们的 dll 可能没有像 window 这样标准化的文档。可能过了一段时间后，我们都会忘记 dll 内部函数的格式。再如当我们把此 dll 发布客户时，那个客户肯定会在背后骂你的！

这时我们需要一个能了解 dll 引出信息途径。我创建一个.h 文件。继续我们旅途。
我们的 dll 代码只需要修改一点点，代码如下：

```
#include <objbase.h>
#include "header.h"//看到没有，这就是我们增加的头文件

extern "C" __declspec(dllexport)  void tulip (void)
{
	::MessageBox(NULL,"ok","I''am fine",MB_OK);
}

BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, void*
lpReserved)
{
	HANDLE g_hModule;
```

```
        switch(dwReason)
        {
        case DLL_PROCESS_ATTACH:
                g_hModule = (HINSTANCE)hModule;
                break;
        case DLL_PROCESS_DETACH:
                 g_hModule=NULL;
                 break;
        }

        return TRUE;
}
```

而 header.h 文件只有一行代码：

```
extern "C" __declspec(dllexport)  void tulip (void);
```

3.2 调试带有头文件的 dll

而此时我们的客户程序应该变成如下样子：（比第二要简单多了）

```
#include <windows.h>
#include "..\header.h"//注意路径

//注意路径，加载 COM_1.lib 的另一种方法是 Project | setting | link 设置里
#pragma comment(lib,"COM_1.lib")

int main(void)
{
        tulip();//只要这样我们就可以调用 dll 里的函数了

        return 0;
}
```

四：小结

今天讲了三种 dll 形式，第一种是没有什么实用价值的，但能讲清楚 dll 与 lib 的关系。我们遇到的情况大多数是第三种，dll 的提供一般会提供 **.lib 和 **.h 文件，而第二种方法适用于系统函数。

几点说明：

1、 DLL 中可以没有 DllMain()，如果你不需要在加载和卸载时做什么的话。

2、 可以指定项目依赖项，把 EXE 项目指定依赖 DLL 项目。这样就不用再管 LIB 的问题了。

3、 如果实现文件的引出函数没有__declspec(dllexport)的话，就不会产生.lib 文件。