**动态链接库**

动态链接库（DLL）是Windows OS基础。DLL不能直接运行，也不能接收消息。DLL包含可被可执行程序或其它DLL调用，以完成某项工作的函数。Windows API中所有函数都包含在DLL中。

1. 概述

**1.1 静态库和动态库**

* **静态库**

函数和数据被编译进一个二进制文件（扩展名常为.lib）。使用静态库时，编译链接主应用程序可执行文件时，链接器从静态库中复制静态库中的函数、数据，并将它们和主应用程序的其它模块链接在一起，创建最终的exe文件。产品发布时，不需静态库文件。

* **动态库**

使用动态库时，一般需要两个文件：一个引入库文件（也称导入库文件），一个DLL（.dll）文件。动态库的引入库文件扩展名也是.lib，但是两者有本质区别，DLL的引入库文件包含DLL导出的函数、变量的符号名，DLL文件则包含DLL实际的函数和数据。使用DLL时，编译链接主应用程序的可执行文件时，只连接该DLL的引入库文件，DLL中的函数代码和数据并不复制到可执行文件中，直到主程序可执行文件运行时，才加载所需的DLL，将该DLL映射到进程的地址空间中，然后访问DLL中导出的函数。产品发布时，除了发布主应用程序的可执行文件外，还要发布该程序将要调用的动态链接库（引入库文件在链接时使用，不需要发布）。

|  |  |  |
| --- | --- | --- |
|  | 静态库 | 动态库 |
|  | 函数和数据被编译进二进制lib库文件。 | 引入库lib文件包含dll导出的函数和变量的符号名，dll包含实际的函数、数据。 |
| 主程序编译链接时 | 链接器从静态库（lib）中复制函数、数据，并将它们和主应用程序的其它模块链接在一起，创建最终的exe文件。 | 只需要引入库（lib），dll中的函数、数据并不复制到exe中。  不需要dll文件。 |
| 主程序运行时 | 不需要静态库。 | 加载dll，将该dll映射到进程的地址空间中，访问dll中导出的函数。  需要dll文件。 |
| 程序发布 | 只需要发布主应用程序的exe，不需要发布静态库。 | 除发布主程序exe文件以外，还需要发布该exe调用的dll。 |

**1.2 使用动态链接库的好处**

可使用多种编程语言缩写。

简化项目管理。

节省磁盘空间和内存。

DLL可以包含对话框模板、字符串、图标、位图等多种资源，多个应用程序可以使用DLL来共享这些资源。可以编写纯资源的动态链接库，供其它应用程序访问。

可以使用DLL支持多语言，即为每种语言创建只支持某一种语言的动态链接库。

**1.3 动态链接库的加载**

两种加载动态链接库方式：隐式链接、显式加载。

1. DLL的创建和使用

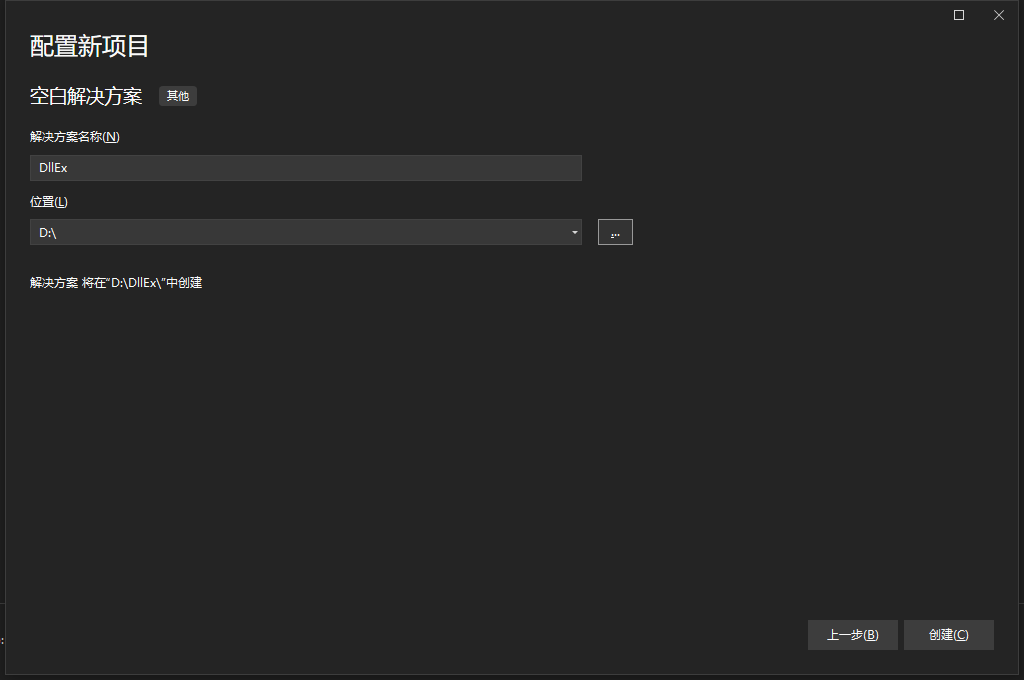
**2.1 创建第一个DLL**

Step1：创建一个“空白解决方案”。

在下图上部搜索框中输入“空白解决方案”，在搜索结果中选择“空白解决方案”。



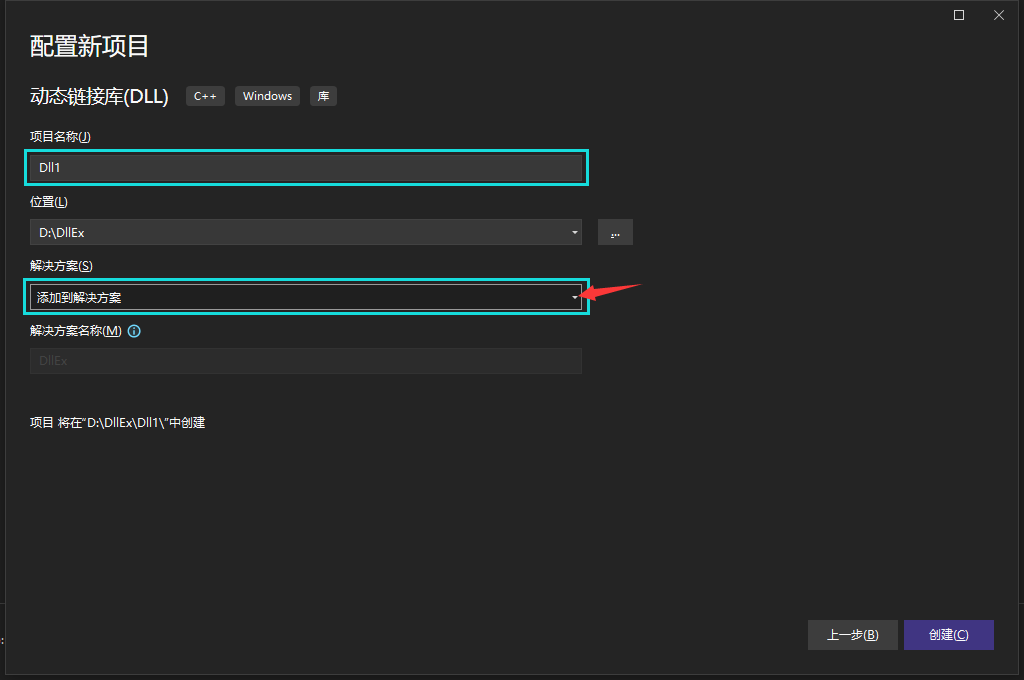
在下图中输入解决方案名称“DllEx”，选择解决方案存储路径，创建空白解决方案。

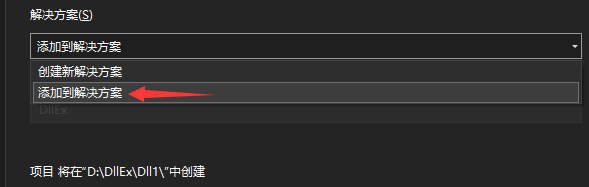


Step2：在空白解决方案中添加一个“动态链接库(DLL)”类型项目。

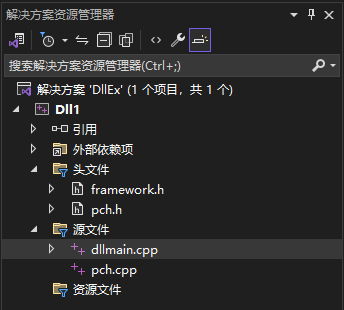


在下图“项目名称”中输入“Dll1”，在“解决方案”下拉列表中选择“添加到解决方案”，在解决方案“DllEx”中创建动态链接库项目Dll1。





Visual Studio（以下简称为VS）自动为创建的项目Dll1添加如下图所示4个文件：



* framework.h文件内容：

*#pragma once*

*#define WIN32\_LEAN\_AND\_MEAN // 从 Windows 头文件中排除极少使用的内容*

*// Windows 头文件*

*#include <windows.h>*

* pch.h文件内容：

*// pch.h: 这是预编译标头文件。*

*// 下方列出的文件仅编译一次，提高了将来生成的生成性能。*

*// 这还将影响 IntelliSense 性能，包括代码完成和许多代码浏览功能。*

*// 但是，如果此处列出的文件中的任何一个在生成之间有更新，它们全部都将被重新编译。*

*// 请勿在此处添加要频繁更新的文件，这将使得性能优势无效。*

*#ifndef PCH\_H*

*#define PCH\_H*

*// 添加要在此处预编译的标头*

*#include "framework.h"*

*#endif //PCH\_H*

* pch.cpp文件内容：

*// pch.cpp: 与预编译标头对应的源文件*

*#include "pch.h"*

*// 当使用预编译的头时，需要使用此源文件，编译才能成功。*

* dllmain.cpp文件内容：

*// dllmain.cpp : 定义 DLL 应用程序的入口点。*

*#include "pch.h"*

*BOOL APIENTRY DllMain( HMODULE 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:*

*break;*

*}*

*return TRUE;*

*}*

Step3：在dllmain.cpp文件中添加如下代码：

int add(int a, int b)

{

return a + b;

}

int subtract(int a, int b)

{

return a - b;

}

利用【生成\生成 DLL1(U)】菜单命令，生成Dll1.dll动态链接库。在目录D:\DllEx\x64\Debug下，可以看到生成了Dll1.dll、Dll1.pdb两个文件。其中，Dll1.dll即为动态链接库文件；Dll1.pdb为程序数据库文件（也称为符号文件）。（pbd文件将项目源代码中的标识符和语句映射到已编译应用中的相应标识符和说明。这些映射文件将调试器链接到源代码，以进行调试。使用标准调试生成配置从 Visual Studio IDE 生成项目时，编译器会创建相应的符号文件。程序员一般不需要直接关注该文件。）

**2.2 dumpbin命令**

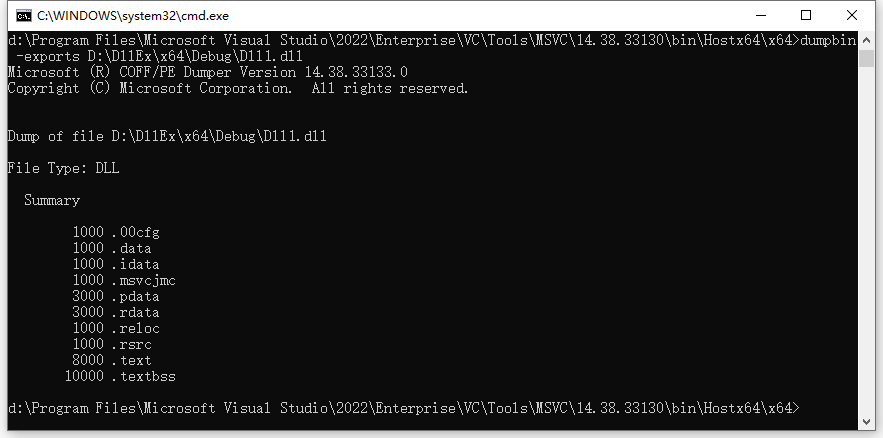
dumpbin命令由VS提供，位于X:\Program Files\Microsoft Visual Studio\2022\Enterprise\VC\Tools\MSVC\14.38.33130\bin\Hostx64\x64目录下。打开Windows的“命令提示符”窗口，进入dumpbin.exe所在目录，输入如下命令：

dumpbin -exports D:\DllEx\x64\Debug\Dll1.dll

查看Dll1.dll导出的函数。

为了方便使用dumpbin命令，可以将dumpbin.exe所在的文件路径添加到Windows的环境变量PATH中。这样，可以先进入Dll1.dll文件所在目录，输入如下命令后直接执行dumpbin命令。

dumpbin -exports Dll1.dll



以上命令输出信息显示，Dll1.dll没有导出函数。

**2.3 从DLL中导出函数**

为了导出DLL中的函数，需要在要导出的函数前添加标识符：\_declspec(dllexport)。修改add、subtract函数的定义：

\_declspec(dllexport) int add(int a, int b)

{

return a + b;

}

\_declspec(dllexport) int subtract(int a, int b)

{

return a - b;

}

重新生成Dll1.dll，这时VS的“输出”窗口中显示如下信息：

*生成开始于 15:29...*

*1>------ 已启动生成: 项目: Dll1, 配置: Debug x64 ------*

*1>dllmain.cpp*

*1> 正在创建库 D:\DllEx\x64\Debug\Dll1.lib 和对象 D:\DllEx\x64\Debug\Dll1.exp*

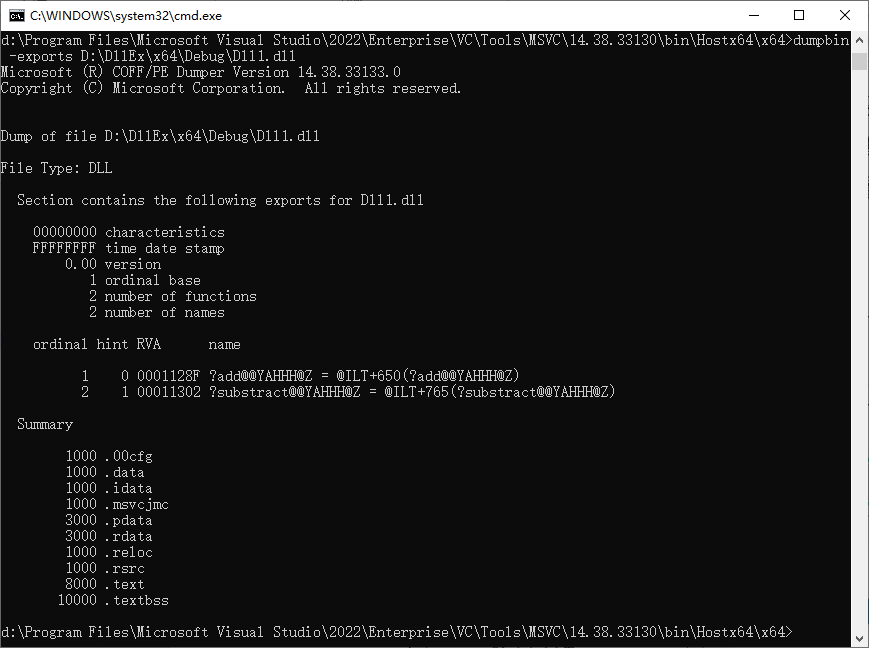
*1>Dll1.vcxproj -> D:\DllEx\x64\Debug\Dll1.dll*

*========== 生成: 1 成功，0 失败，0 最新，0 已跳过 ==========*

*========== 生成 于 15:29 完成，耗时 01.560 秒 ==========*

以上信息显示，VS又生成了两个新文件：Dll1.lib、Dll1.exp。Dll1.lib为DLL的引入库文件，保存Dll1.dll中导出的函数、变量的符号名；Dll1.exp为导出文件，包含有关导出函数、数据项的信息。程序员一般不需要直接关注exp文件。

再次运行之前的dumpbin命令查看Dll1.dll导出函数函数的信息，结果如下图。



从上图看出，这时多了一些信息，我们关注一下其中的如下内容：

*ordinal hint RVA name*

*1 0 0001128F ?add@@YAHHH@Z = @ILT+650(?add@@YAHHH@Z)*

*2 1 00011302 ?subtract@@YAHHH@Z = @ILT+765(?subtract@@YAHHH@Z)*

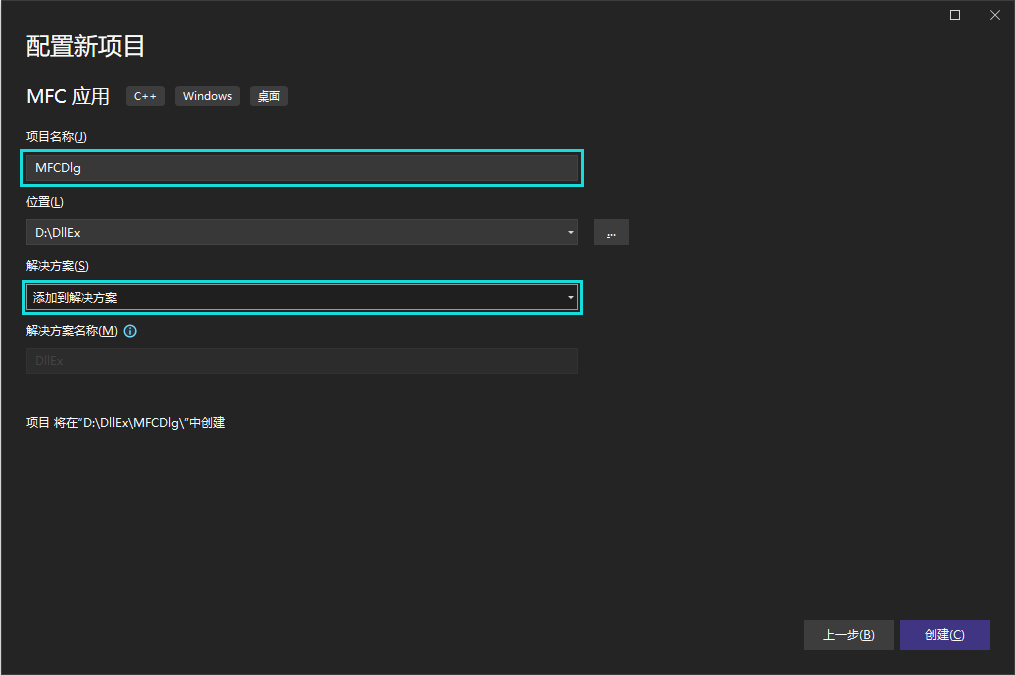
这段信息中，“ordinal”列中的“1”、“2”为导出函数的序号；“hint”列为提示码，该信息不需要关注；“RVA”列列出的内存地址是导出的函数在DLL模块中的内存位置，即，通过该地址，可以在DLL中找到这些函数；“name”列是导出的函数的名称，add导出函数的名称为“?add@@YAHHH@Z”，subtract导出函数的名称为“?subtract@@YAHHH@Z”。

解释：C++支持函数重载，对于重载的多个函数而言，其函数名一样，为了加以区分，编译链接时时，C++会根据某种规则篡改函数名，这个过程称之为“名字改编”（或“名字粉碎”）。

1. 隐式链接方式加载DLL

在“DllEx”解决方案中添加一个基于对话框的MFC应用程序，项目名为“MFC”。

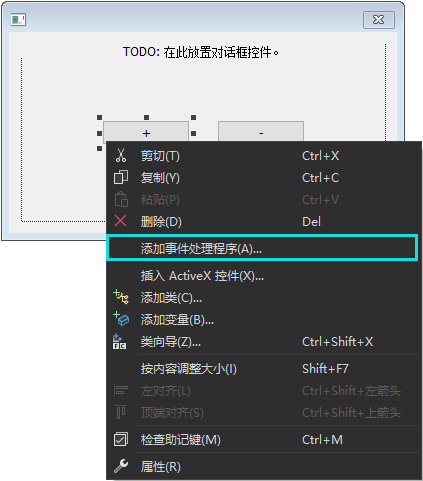




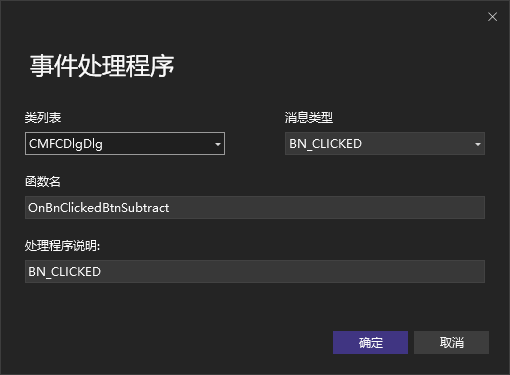


在“MFCDlg”项目的主对话框资源上放置两个Button控件，其资源ID、Caption属性以及鼠标单击消息响应函数名、函数功能如下表。

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Caption | 消息响应函数名 | 函数功能 |
| IDC\_BTN\_ADD | + | OnBnClickedBtnAdd | 调用Dll1.dll的add函数 |
| IDC\_BTN\_SUBTRACT | - | OnBnClickedBtnSubtract | 调用Dll1.dll的subtract函数 |







**3.1 函数声明**

在“+”、“-”按钮的单击消息响应函数OnBnClickedBtnAdd、OnBnClickedBtnSubtract中，调用Dll1.dll的add、subtract函数，需要在调用前对这两个函数进行声明：

extern int add(int a, int b);

extern int subtract(int a, int b);

也可以写成

int add(int a, int b);

int subtract(int a, int b);

OnBnClickedBtnAdd、OnBnClickedBtnSubtract函数代码如下：

void CMFCDlgDlg::OnBnClickedBtnAdd()

{

CString str;

str.Format(\_T("7+3=%d"), add(7, 3));

MessageBox(str);

}

void CMFCDlgDlg::OnBnClickedBtnSubtract()

{

CString str;

str.Format(\_T("7-3=%d"), subtract(7, 3));

MessageBox(str);

}

生成项目“MFCDlg”，“输出”信息栏反馈信息有如下内容：

*1>正在生成代码...*

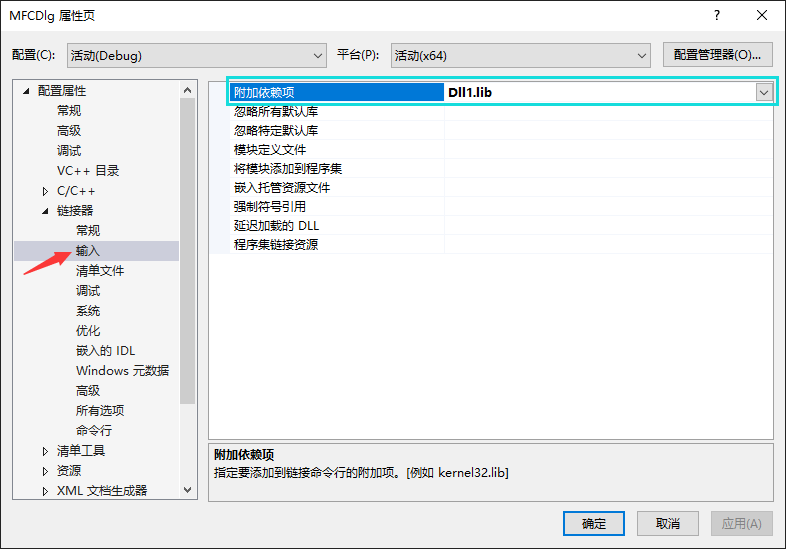
*1>MFCDlgDlg.obj : error LNK2019: 无法解析的外部符号 "int \_\_cdecl add(int,int)" (?add@@YAHHH@Z)，函数 "public: void \_\_cdecl CMFCDlgDlg::OnBnClickedBtnAdd(void)" (?OnBnClickedBtnAdd@CMFCDlgDlg@@QEAAXXZ) 中引用了该符号*

*1>MFCDlgDlg.obj : error LNK2019: 无法解析的外部符号 "int \_\_cdecl subtract(int,int)" (?subtract@@YAHHH@Z)，函数 "public: void \_\_cdecl CMFCDlgDlg::OnBnClickedBtnSubtract(void)" (?OnBnClickedBtnSubtract@CMFCDlgDlg@@QEAAXXZ) 中引用了该符号*

*1>D:\DllEx\x64\Debug\MFCDlg.exe : fatal error LNK1120: 2 个无法解析的外部命令*

*1>已完成生成项目“MFCDlg.vcxproj”的操作 - 失败。*

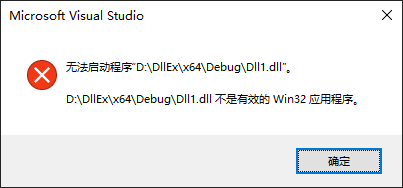
因为在调用add、subtract前进行了它们的声明，因此在项目“MFCDlg”编译阶段正常通过。以上两个error为链接时的错误，其原因是链接器没有找到add、subtract的实现（定义）。此时需要使用Dll1.dll的引入库文件Dll1.lib来解决这个问题。从D:\DllEx\x64\Debug\目录下复制Dll1.lib文件，粘贴至D:\DllEx\MFCDlg\目录中。在“MFCDlg”的项目属性中，在“配置属性”\“链接器”\“输入”\“附加依赖项”中，输入“Dll1.lib”。



再次生成“MFCDlg”项目，编译、链接都正常通过，成功生成MFCDlg.exe文件。

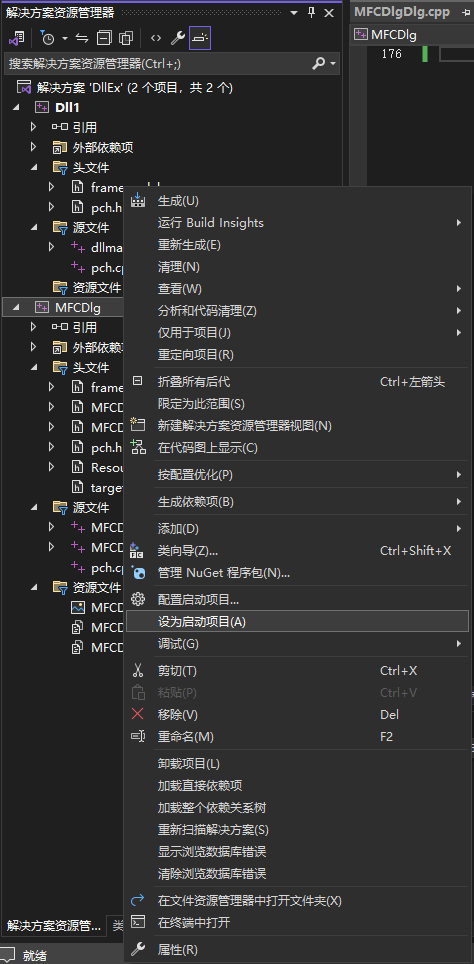
应用程序调用到一个DLL导出的函数，编译时要确保函数调用前的函数原型声明，链接时只需要该DLL的引入库lib文件，不需要dll文件。lib文件并没有被调用的函数的实际代码，只是用来为链接器链接程序提供必要的信息。

在VS中运行调试运行，失败！报如下错误。



这是因为解决方案“DllEx”中目前有两个项目“Dll1”、“MFCDlg”，其中“Dll1”是解决方案中第一个项目，默认它为解决方案的启动项目，调试运行时VS将试图执行项目“Dll1”生成的exe，但“Dll1”是一个动态链接库项目，它生成的Dll1.dll不能直接运行，因而报错。

将项目“MFCDlg”设置为解决方案的启动项目，如下图所示。



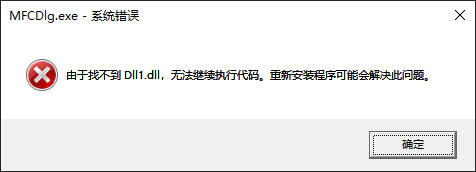
“MFCDlg”被设置为解决方案的启动项目后，可以看到，VS解决方案资源管理器中，由原来项目“Dll1”以粗体显示，变为“MFCDlg”以粗体显示。再次执行调试运行，运行正常，弹出如下对话框。



点出“+”、“-”按钮，分别弹出如下两个消息对话框。

说明：项目Dll1未正常完成生成，则以上调试运行会失败，弹出如下错误提示对话框。



VS调试运行，需要加载DLL时，将按以下顺序搜索：

* + exe的执行目录

D:\DllEx\x64\Debug

* + 当前目录

D:\DllEx\MFCDlg

* + 系统目录

C:\Windows\system32 、C:\Windows\system 、C:\Windows等。

* + Windows环境变量path中所列出的路径

MFCDlg.exe调试运行时，按如下顺序搜索Dll1.dll以加载，本示例由于项目“Dll1”、“MFCDlg”同属于“DllEx”解决方案，因此Dll1.dll、MFCDlg.exe文件都是生成在D:\DllEx\x64\Debug中，即两个项目都正常完成了生成的情况下，在VS启动调试运行MFCDlg不会报如上错误。

如果项目项目“Dll1”、“MFCDlg”不属于同一解决方案，MFCDlg.exe试图加载Dll1.dll时，搜索不到Dll1.dll文件，则会运行失败，报如上错误，此时，将“Dll1.dll”文件复制到MFCDlg.exe可以搜索到的目录中即可。

**3.2 利用\_declspec(dllimport)声明函数**

为了表明函数是从动态链接库中引入的，可以在声明函数时使用\_declspec(dllimport)。在MFCDlg中MFCDlgDlg.cpp文件中，将add、subtract函数的声明改成如下形式：

\_declspec(dllimport) int add(int a, int b);

\_declspec(dllimport) int subtract(int a, int b);

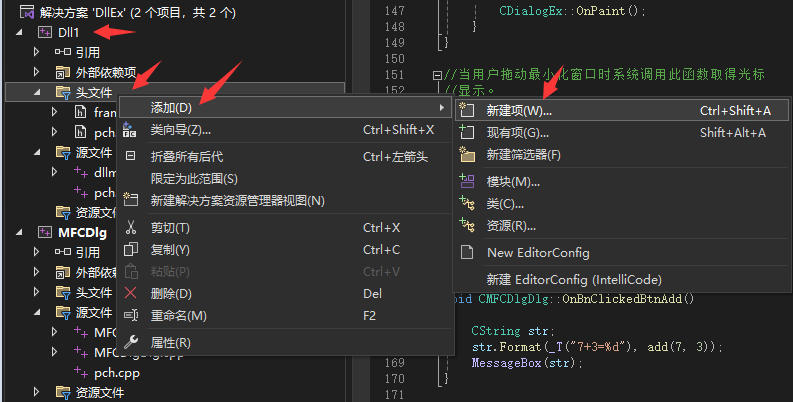
编译运行“MFCDlg”，运行情况没有变化。

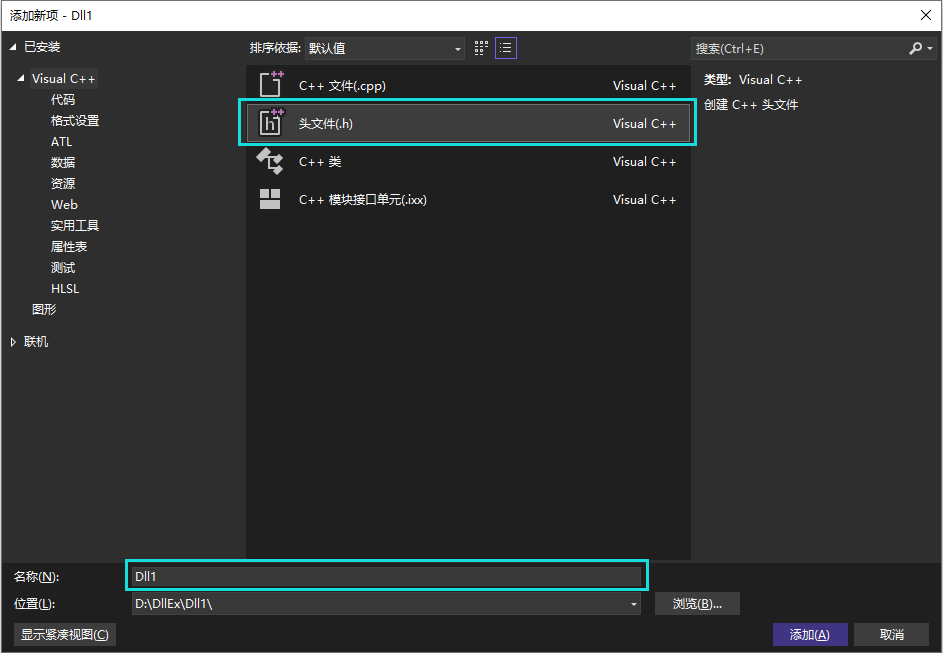
使用\_declspec(dllimport)声明函数，明确告知编译器该函数从DLL中引入，VS生成运行效率更高的代码。

1. 完善DLL示例

为便于客户端程序使用DLL，编写DLL时，一般提供一个头文件，以提供DLL导出函数的函数原型声明以及有关注释说明。

在项目“Dll1”中添加头文件“Dll1.h”。





在“Dll1.h”中添加如下代码：

\_declspec(dllimport) int add(int a, int b);

\_declspec(dllimport) int subtract(int a, int b);

注意：头文件是“Dll1.h”是提供给客户端程序，调用Dll1.dll中导出的函数时使用的，因此声明add、subtract函数时使用关键字“dllimport”，向客户端程序表明函数是从DLL导入的。

将项目“MFCDlg”的“MFCDlgDlg.cpp”文件中，原来声明add、subtract函数的代码删除，以如下头文件包含指令替代。

#include "..\Dll1\Dll1.h"

编译运行“MFCDlg”，运行情况没有变化。这样，在向客户端发布DLL时，可以将头文件“Dll1.h”一起提供，客户端程序通过包含该头文件实现函数原型声明。

说明：注意以上头文件包含时Dll1.h的文件路径。由于头文件“Dll1.h”是创建在项目Dll1中，其存储路径为：D:\DllEx\Dll1。而MFCDlgDlg.cpp文件的存储路径为：D:\DllEx\MFCDlg。因此Dll1.h对于MFCDlgDlg.cpp的相对路径为“..\Dll1\”。当然，一种简单的处理方式就是，将Dll1.h复制到不需要指定路径的目录下，如“D:\DllEx\MFCDlg”。

正如以上提到的，以上的Dll1.h头文件中，声明函数时使用的是关键字“dllimport”，因此，只适合提供给客户端程序使用。对于DLL自身来说，声明add、subtract函数时，应该使用关键字“dllexport”，以告诉编译器该函数为导出的函数。

将Dll1.h头文件修改为如下：

#pragma once

#ifdef DLL1\_HIE

#else

#define DLL1\_HIE \_declspec(dllimport)

#endif

DLL1\_HIE int add(int a, int b);

DLL1\_HIE int subtract(int a, int b);

文件第一行为确保该头文件在一个项目不被重包含的预编译指令。

其后的条件编译指令段，首先判断是否定义了宏DLL1\_HIE，如果定义过，则不做任何处理，如果尚未定义，则定义宏DLL1\_HIE为\_declspec(dllimport)。

再其后的add函数原型声明前面，以宏DLL1\_HIE替代原来的\_declspec(dllimport)。当客户端程序包含头文件“Dll1.h”时，因为没定义宏DLL1\_HIE，根据条件编译指令段，定义DLL1\_HIE为\_declspec(dllimport)，其后add、subtract函数声明前的宏名DLL1\_HIE在编译预处理时将会被替换为\_declspec(dllimport)。

在“Dll1”项目中，将dllmain.cpp源文件中，定义add、subtract函数的原如下代码进行修改：

\_declspec(dllexport) int add(int a, int b)

{

return a + b;

}

\_declspec(dllexport) int subtract(int a, int b)

{

return a - b;

}

修改为：

#define DLL1\_HIE \_declspec(dllexport)

#include "Dll1.h"

int add(int a, int b)

{

return a + b;

}

int subtract(int a, int b)

{

return a - b;

}

将以上#include "Dll1.h"指令用“Dll1.h”的内容替换：

#define DLL1\_HIE \_declspec(dllexport)

#ifdef DLL1\_HIE

#else

#define DLL1\_HIE \_declspec(dllimport)

#endif

DLL1\_HIE int add(int a, int b);

DLL1\_HIE int subtract(int a, int b);

int add(int a, int b)

{

return a + b;

}

int subtract(int a, int b)

{

return a - b;

}

本段代码一开始将宏DLL1\_HIE定义为\_declspec(dllexport)，因此以上标红部分预编译指令无任何影响效果，即以下两条语句前的宏DLL1\_HIE，在编译预处理时均将被\_declspec(dllexport)替换。

DLL1\_HIE int add(int a, int b);

DLL1\_HIE int subtract(int a, int b);

替换后的形式为：

\_declspec(dllexport) int add(int a, int b);

\_declspec(dllexport) int subtract(int a, int b);

也就是说，在函数add、subtract函数定义前，如上函数原型声明已经将函数声明为从DLL中导出的函数，因此在这两个函数定义时，其函数原型前就可以省略\_declspec(dllexport)了。

如果在项目“Dll1”的其它源文件中调用了add、subtract函数，需要通过包含头文件“Dll1.h”实现函数原型声明，则应在头文件包含指令前添加如下指令：

#define DLL1\_HIE

即在包含头文件前，将DLL1\_HIE定义为一个空的宏，这样，头文件“Dll1.h”内容编译预处理前后内容对比如下表：

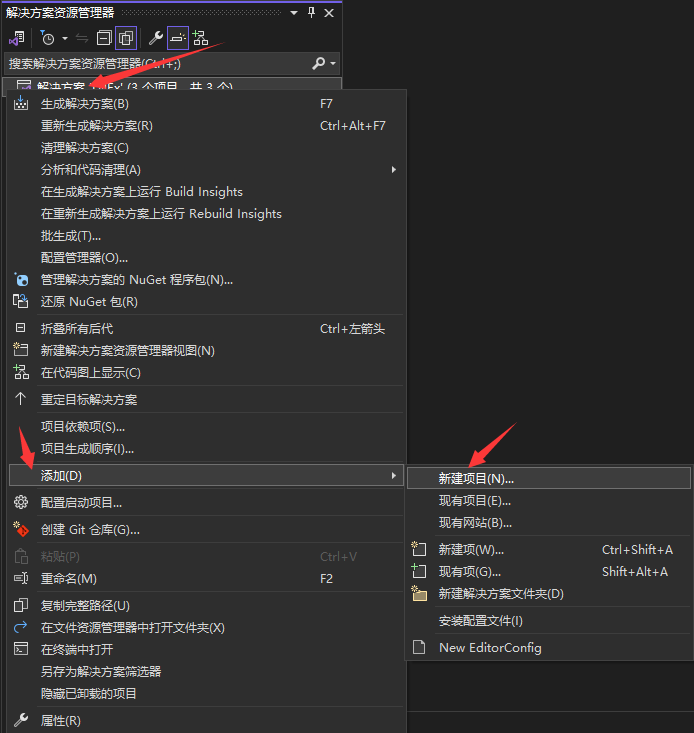
|  |  |
| --- | --- |
| 编译预处理前 | 编译预处理后 |
| #ifdef DLL1\_HIE  #else  #define DLL1\_HIE \_declspec(dllimport)  #endif  DLL1\_HIE int add(int a, int b);  DLL1\_HIE int subtract(int a, int b); | int add(int a, int b);  int subtract(int a, int b); |

即，通过包含头文件“Dll1.h”，引入的add、subtract的函数原型声明，既没有\_declspec(dllimport)，也没有\_declspec(dllexport)，恰好是一般形式的函数原型声明，符合要求。

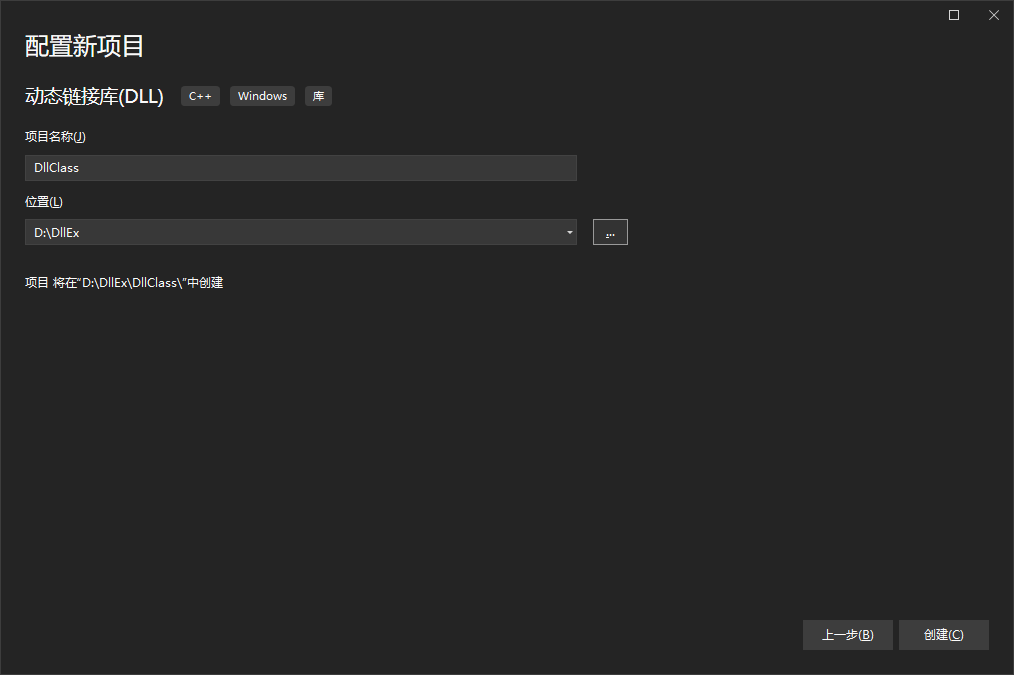
通过以上处理，不管是在“Dll1”项目内部，还是外部，都能使用同一个头文件“Dll1.h”，实现各处需要的，add、subtract正确的函数原型声明语法形式。

1. 从DLL中导出C++类

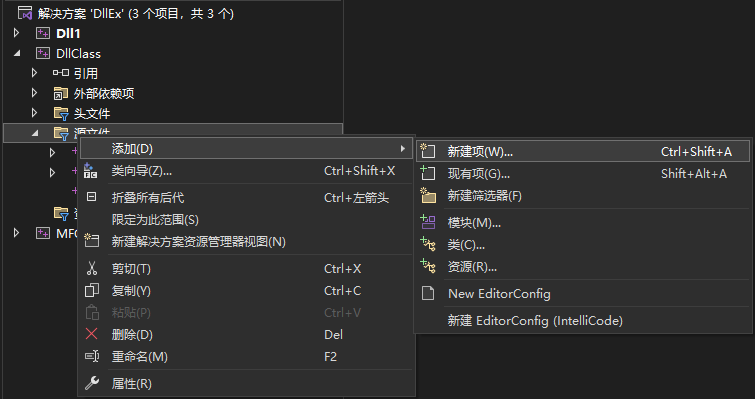
DLL可以导出C++类。往解决方案“DllEx”中再添加一个动态链接库项目“DllClass”。







往项目“DllClass”中添加一个源文件“DllClass.cpp”、一个头文件“DllClass.h”。



DllClass.h文件：

#pragma once

#ifdef DLL\_CLASS\_HIE

#else

#define DLL\_CLASS\_HIE \_declspec(dllimport)

#endif

class DLL\_CLASS\_HIE Point

{

public:

void output(int x, int y);

};

DllClass.dll文件：

#include "pch.h"

#define DLL\_CLASS\_HIE \_declspec(dllexport)

#include "DllClass.h"

#include <tchar.h>

void Point::output(int x, int y)

{

HWND hwnd = GetForegroundWindow();

HDC hdc = GetDC(hwnd);

TCHAR buf[20] = { 0 };

\_stprintf\_s(buf, \_T("x=%d,y=%d"), x, y);

TextOut(hdc, x, y, buf, static\_cast<int>(\_tcslen(buf)));

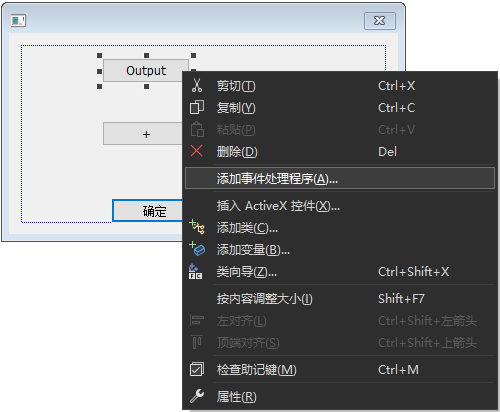
ReleaseDC(hwnd, hdc);

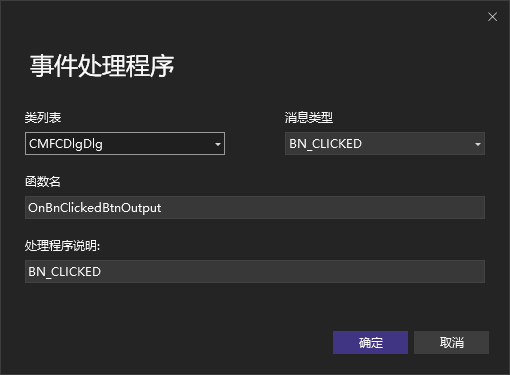
}

定义了一个Point类，其成员函数output为public可访问属性。类似于从DLL导出函数，为了导出Point类，需要在class关键字与类名之间加入\_declspec(dllexport)，在客户端程序中导入Point，使用\_declspec(dllimport)。

Point的成员函数output的功能是在调用程序的窗口中显示x、y的值。GetForegroundWindow是一个Windows API函数，它返回前景窗口的句柄，这个前景窗口就是当前用户正在使用的那个程序窗口。因此在上述示例代码中，通过调用GetForegroundWindow 函数，获得调用者进程当前正在使用的那个窗口的句柄。对于上述示例来说，DllClass.dll的客户端程序是MFCDlg程序，于是调用GetForegroundWindow函数后得到的就是MFCDlg程序主对话框窗口的句柄。在得到了窗口句柄之后，该窗口的DC也就得到了（通过调用GetDC函数得到）。之后定义了一个字符缓冲区（buf），并将该字符数组中的元素都初始化为0。然后调用\_stprintf\_s函数将Point的坐标（x, y）格式化到该字符数组中。最后，就可以调用TextOut函数在窗口(0, 0)位置处输出坐标了。示例代码最后调用ReleaseDC函数释放设备句柄。因为示例代码中使用了与字符集无关的\_stprintf\_s宏，所以程序需要包含相应的头文件：tchar.h。

在项目“MFCDlg”的主对话框上放置一个Button控件，ID: IDC\_BTN\_OUTPUT，Caption: Output。添加鼠标单击消息处理函数：OnBnClickedBtnOutput，其功能是调用DllClass.dll导出的Point类的output成员函数。





OnBnClickedBtnOutput函数代码如下：

#include "..\DllClass\DllClass.h"

void CMFCDlgDlg::OnBnClickedBtnOutput()

{

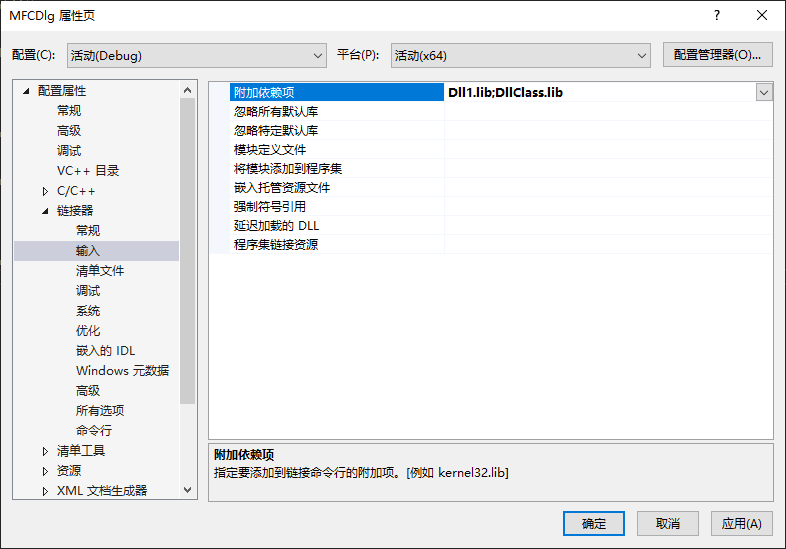
Point pt;

pt.output(100, 120);

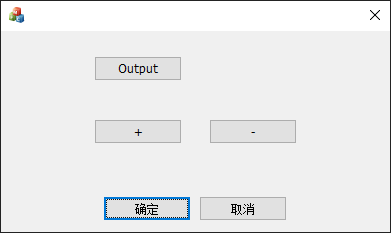
}

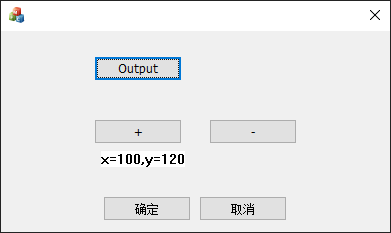
注意包含头文件DllClass.h，并留意其路径。

生成项目“DllClass”后，将在D:\DllEx\x64\Debug\中生成的“DllClass.lib”文件复制粘贴至项目“MFCDlg”的当前目录D:\DllEx\MFCDlg\中。如下图所示，配置项目“MFCDlg”的属性，将“DllClass.lib”添加到项目属性中的“附加依赖项”里。



生成项目“MFCDlg”，调试运行。





点击“Output”按钮，调用DllClass.dll中导出的Point类output成员函数，在MFCDlg的主窗体中显示运行结果。

利用dumpbin命令的exports选项查看DllClass.dll的导出情况：

*Section contains the following exports for DllClass.dll*

*00000000 characteristics*

*FFFFFFFF time date stamp*

*0.00 version*

*1 ordinal base*

*3 number of functions*

*3 number of names*

*ordinal hint RVA name*

*1 0 000112E4 ??4Point@@QEAAAEAV0@$$QEAV0@@Z = @ILT+735(??4Point@@QEAAAEAV0@$$QEAV0@@Z)*

*2 1 000110EB ??4Point@@QEAAAEAV0@AEBV0@@Z = @ILT+230(??4Point@@QEAAAEAV0@AEBV0@@Z)*

*3 2 00011168 ?output@Point@@QEAAXHH@Z = @ILT+355(?output@Point@@QEAAXHH@Z)*

DLL中的类，可以不导出整个类，而只导出部分成员函数。修改DllClass的“DllClass.h”、“DllClass.cpp”。

DllClass.h文件：

#pragma once

#ifdef DLL\_CLASS\_HIE

#else

#define DLL\_CLASS\_HIE \_declspec(dllimport)

#endif

class /\*DLL\_CLASS\_HIE\*/ Point

{

public:

DLL\_CLASS\_HIE void output(int x, int y);

void TestMethod();

};

DllClass.cpp文件：

#include "pch.h"

#define DLL\_CLASS\_HIE \_declspec(dllexport)

#include "DllClass.h"

#include <tchar.h>

void Point::output(int x, int y)

{

HWND hwnd = GetForegroundWindow();

HDC hdc = GetDC(hwnd);

TCHAR buf[20] = { 0 };

\_stprintf\_s(buf, \_T("x=%d,y=%d"), x, y);

TextOut(hdc, x, y, buf, static\_cast<int>(\_tcslen(buf)));

ReleaseDC(hwnd, hdc);

}

void Point::TestMethod()

{

}

生成解决方案，调试运行MFCGlg，没有变化。再使用dumpbin命令的exports选项查看DllClass.dll的导出情况：

*Section contains the following exports for DllClass.dll*

*00000000 characteristics*

*FFFFFFFF time date stamp*

*0.00 version*

*1 ordinal base*

*1 number of functions*

*1 number of names*

*ordinal hint RVA name*

*1 0 00011168 ?output@Point@@QEAAXHH@Z = @ILT+355(?output@Point@@QEAAXHH@Z)*

Point类只导出了output成员函数。

最后注意，只能导出类的public成员函数，protected、private成员函数导出了不能被其它程序访问。

1. 名字改编问题

**6.1 extern “C”**

如前提到，C++编译时，会对函数名进行名字改编，而且不同编译器使用的改编规则可能不同。因此，当不同编译器分别生成程序的不同模块时，可能会造成名字不一致的问题。为了避免名字不一致，定义导出函数时，需要加限定符：extern “C”，它的作用是告诉编译器，函数名以C语言的形式进行extern扩展（外部扩展），以保证编译过后函数名被“翻译”为相同的形式，进而确保最终链接正常。

将项目“Dll1”的“Dll1.h”、“dllmain.cpp”修改为如下形式：

Dll1.h文件：

#pragma once

#ifdef DLL1\_HIE

#else

#define DLL1\_HIE **extern "C"** \_declspec(dllimport)

#endif

DLL1\_HIE int add(int a, int b);

DLL1\_HIE int subtract(int a, int b);

dllmain.cpp文件：

// dllmain.cpp : 定义 DLL 应用程序的入口点。

#include "pch.h"

#define DLL1\_HIE **extern "C"** \_declspec(dllexport)

#include "Dll1.h"

BOOL APIENTRY DllMain(HMODULE 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:

break;

}

return TRUE;

}

int add(int a, int b)

{

return a + b;

}

int subtract(int a, int b)

{

return a - b;

}

生成项目“Dll1”，调试运行“MFCDlg”，没有变化。再次利用dumpbin命令查看Dll1.dll的导出情况：

*ordinal hint RVA name*

*1 0 00011249 add = @ILT+580(add)*

*2 1 000110BE subtract = @ILT+185(subtract)*

Dll1.dll导出的函数名没有被改编。

注意，如果使用extern “C”来声明一个函数以C语言的形式extern，则这个函数不能被重载。

*extern "C" int Fun(int a)*

*{*

*return a + 2;*

*}*

*extern "C" int Fun(int a, int b)*

*{*

*return a + b;*

*}*

如果程序中有以上函数重载的形式，编译时VS将报如下错误：

*error C2733: “Fun”: 无法重载具有外部 "C" 链接的函数*

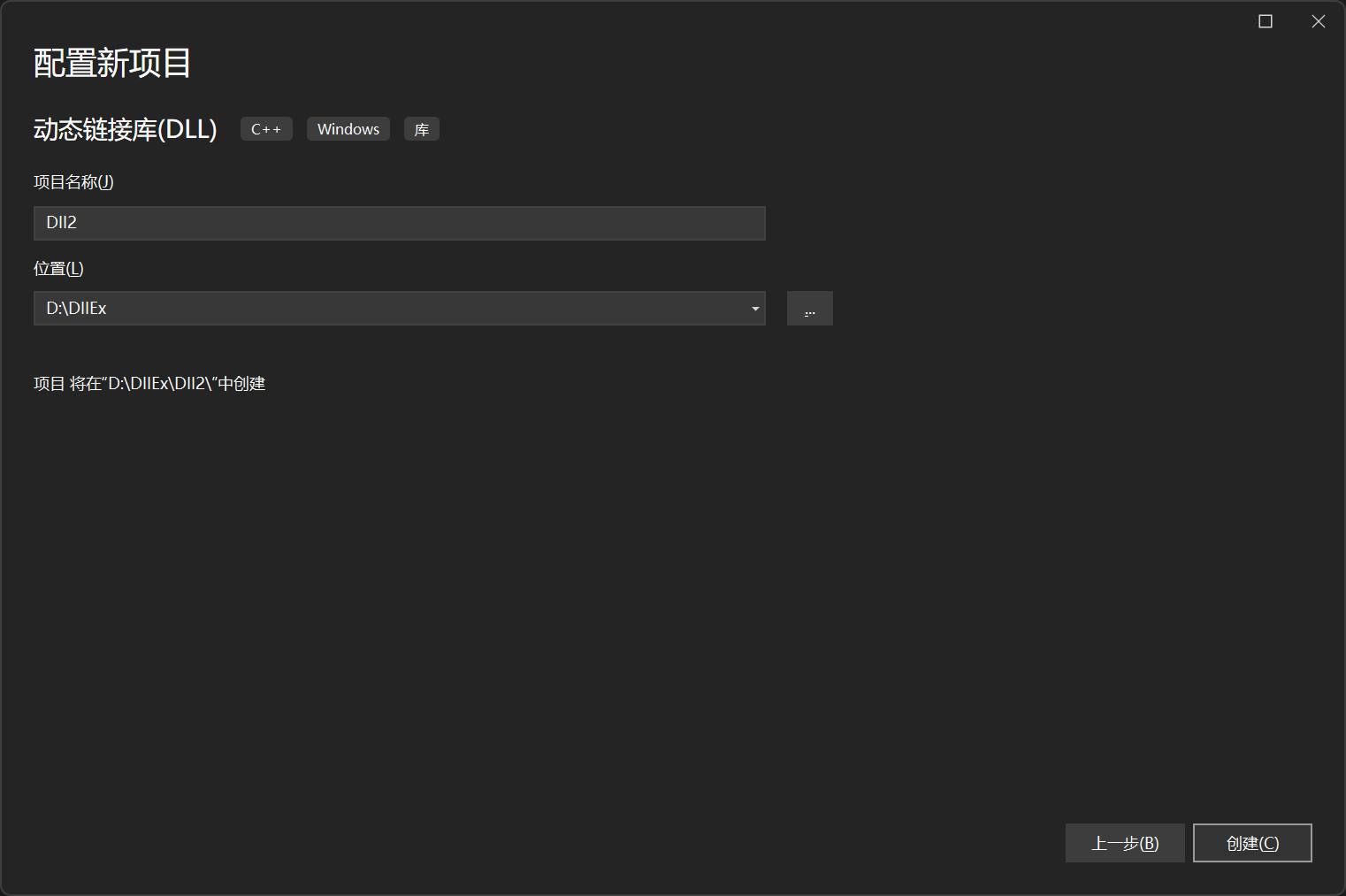
extern “C”还有一个缺陷，它不能用于导出一个类的成员函数，只能用于导出全局函数。

有关名字改编问题比较复杂，它甚至还会关系到调用约定的问题。因此不同语言之间的混合编程，不同编译器混用等，涉及的问题比较复杂，这里先略过。

**6.2 模块定义文件def**

可以使用模块定义文件（def文件）来解决名字改编问题。

在“DllEx”解决方案中添加一个动态链接库项目“Dll2”。



为Dll2项目新建“Dll2.cpp”文件，其代码如下。

#include "pch.h"

extern "C" int add(int a, int b)

{

return a + b;

}

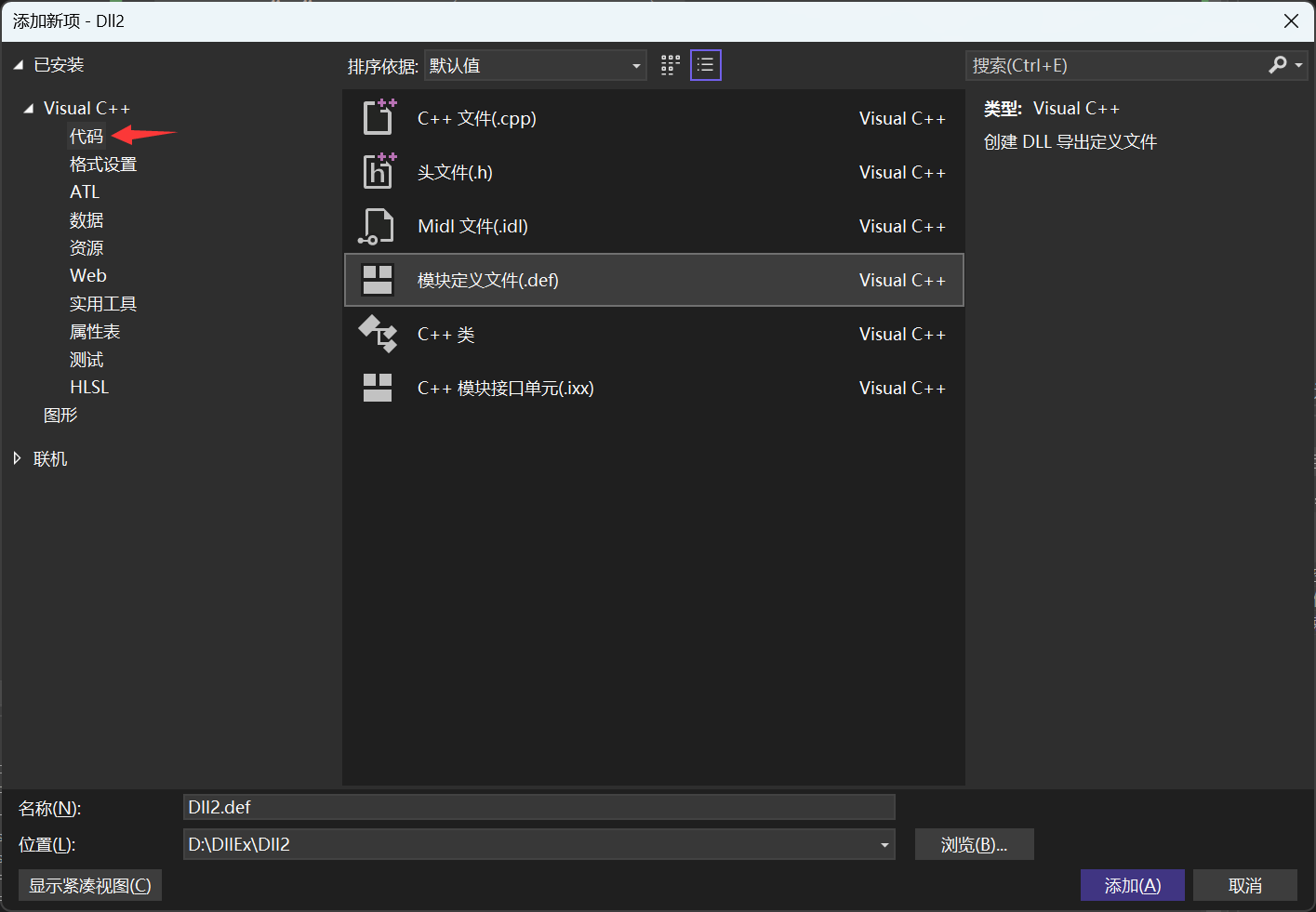
int subtract(int a, int b)

{

return a - b;

}

添加Dll2.def文件。



Dll2.def内容如下：

LIBRARY Dll2

EXPORTS

add

subtract

说明：

LIBRARY语句指定动态链接库的内部名称，该名称与生成的动态链接库名称一定要匹配。LIBRARY语句并非必需。

EXPORTS语句给出DLL要导出的函数，以及为这些导出函数指定符号名。

当链接器链接时，分析def文件，当发现EXPORTS语句下有add、subtract两个符号，并且它们与源文件中定义的add、subtract函数名一致时，编译器就会以add、subtract符号名导出相应的函数。如果将要导出的符号名与源文件中定义的函数名不同，则可以按照如下语法指定导出的函数：

entryname=internalname

等号左边entryname项是导出的符号名，右边internalname项是DLL中将要导出的函数的名字。

生成Dll2，利用dumpbib命令查看Dll2.dll的导出信息。

*Section contains the following exports for Dll2.dll*

*00000000 characteristics*

*FFFFFFFF time date stamp*

*0.00 version*

*1 ordinal base*

*2 number of functions*

*2 number of names*

*ordinal hint RVA name*

*1 0 00011294 add = @ILT+655(?add@@YAHHH@Z)*

*2 1 00011096 subtract = @ILT+145(?subtract@@YAHHH@Z)*

Dll2.dll按照Dll2.def文件给出的函数名导出。

1. 显示加载方式加载DLL

**7.1 LoadLibrary函数**

使用LoadLibrary函数动态加载DLL，它的作用是将指定的可执行模块映射到调用进程的地址空间。LoadLibrary函数的原型如下：

HMODULE LoadLibrary(LPCTSTR lpLibFileName);

LoadLibrary函数不仅能加载DLL，也可以加载可执行模块（exe）。当加载可执行模块时，主要是为了访问模块内的一些资源，如对话框、位图、图标资源等。形参lpLibFileName为字符串类型，指定可执行模块的名称，一个dll文件，或一个exe文件。加载成功，函数返回所加载的模块的句柄，其类型是HMODULE。HMODULE和HINSTANCE类型可以通用。为了字符集兼容，现在的LoadLibrary是一个宏。

*WINBASEAPI*

*\_Ret\_maybenull\_*

*HMODULE*

*WINAPI*

*LoadLibraryA(*

*\_In\_ LPCSTR lpLibFileName*

*);*

*WINBASEAPI*

*\_Ret\_maybenull\_*

*HMODULE*

*WINAPI*

*LoadLibraryW(*

*\_In\_ LPCWSTR lpLibFileName*

*);*

*#ifdef UNICODE*

*#define LoadLibrary LoadLibraryW*

*#else*

*#define LoadLibrary LoadLibraryA*

*#endif // !UNICODE*

（以上代码摘自libloaderapi.h）

修改“MFCDlg”项目的MFCDlgDlg.cpp文件中的OnBnClickedBtnAdd函数：

void CMFCDlgDlg::OnBnClickedBtnAdd()

{

HMODULE hMoudule = LoadLibrary(\_T("Dll2.dll"));

if (!hMoudule)

{

MessageBox(\_T("DLL加载失败！"));

return;

}

else

{

typedef int (\*FunAdd)(int a, int b);

FunAdd pAdd = (FunAdd)GetProcAddress(hMoudule, "add");

if (!pAdd)

{

MessageBox(\_T("获取函数地址失败！"));

return;

}

CString str;

str.Format(\_T("7+3=%d"), pAdd(7, 3));

MessageBox(str);

FreeLibrary(hMoudule);

}

}

重新生成“MFCDlg”，调试运行，结果不变。即以上代码利用LoadLibrary函数显式将“Dll2.dll”成功加载。在动态加载DLL时，客户端程序不需要包含导出函数声明的头文件和引入库文件，只需要.dll文件即可。

DLL动态加载一般包括几个步骤：（1）调用LoadLibrary函数加载DLL文件，获取DLL句柄。（2）通过GetProcAddress函数获取DLL中导出函数的地址。（3）通过函数指针调用DLL中的导出函数。（4）调用FreeLibrary函数释放DLL句柄。

注意，使用LoadLibrary函数加载DLL后，应该调用FreeLibrary函数来释放DLL句柄。

**7.2 调用约定**

略…

**7.3 根据序号访问DLL中导出的函数**

略…

1. Debug、Release版本及X64、X86平台

VS中生成项目/解决方案时有Debug、Release两个版本的选择。

Release为发布版本，即最终提交的版本。VS在生成Release版本时会进行优化，以追求程序大小、运行速度上的最优。

Debug版本为调试版本。生成Debug版本时，程序会包含调试信息，相比Release版本，Debug版本文件比较大，运行效率稍慢，但为程序员提供调试断点调试、单步调试等能力。

Debug版本调试通过后，再发布Release版本。

目前PC机主流为64位，但仍有部分还是32位，VS生成时可以选择其运行平台。

因此，在VS中生成项目/解决方案时，如下四种版本选择：

64位Debug

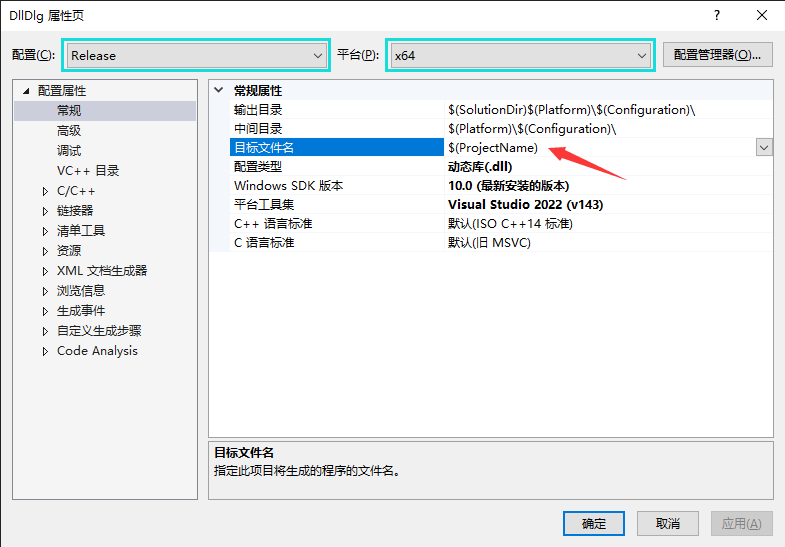
64位Release

32位Debug

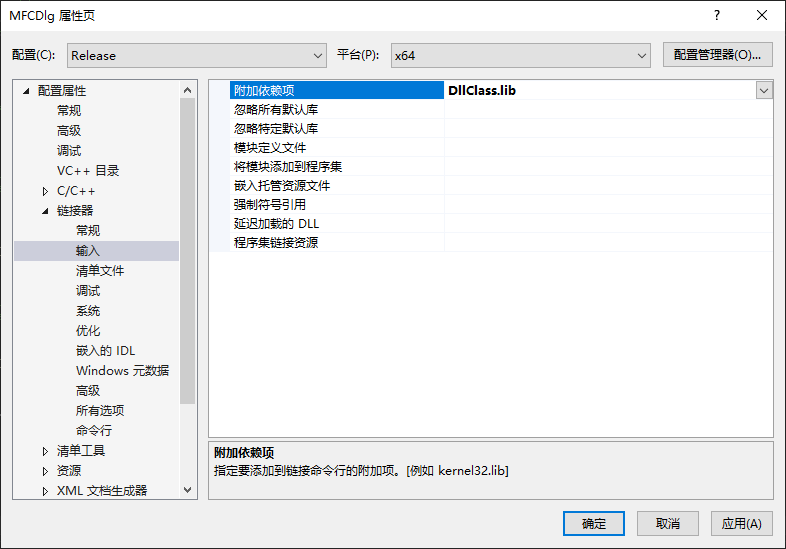
32位Release

编写DLL时，要注意DLL与其客户应用程序的版本应一致，否则会无法正常加载DLL。为了方便，不同版本（Debug、Release、x64、x86）生成时，可以生成不同的文件名。

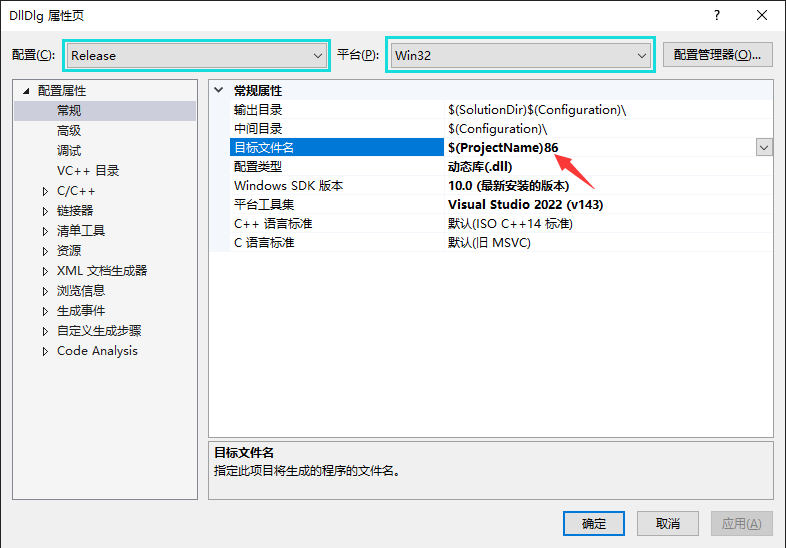
* Release/x64



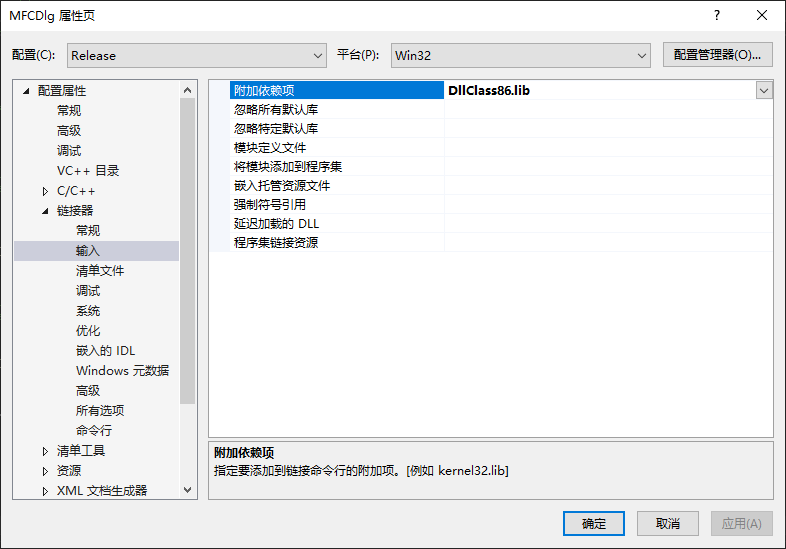
生成的文件：DllClass.dll、DllClass.exp、DllClass.lib、DllClass.pdb。DllClass——项目名。



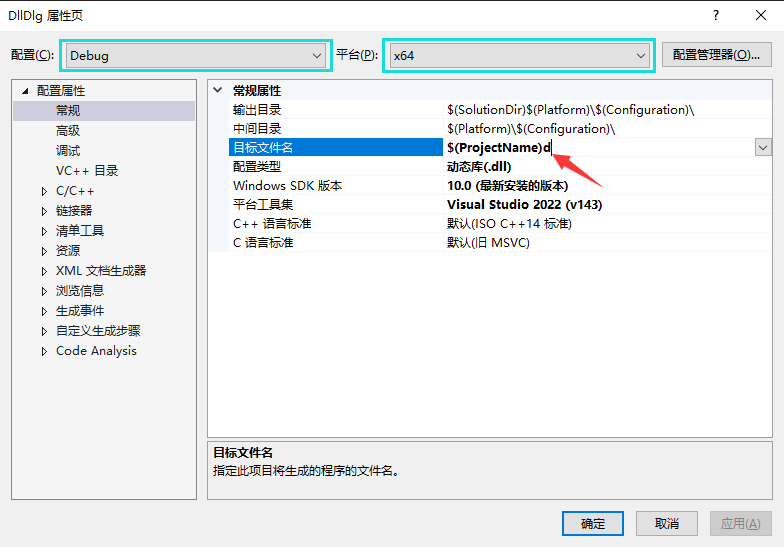
* Release/x86



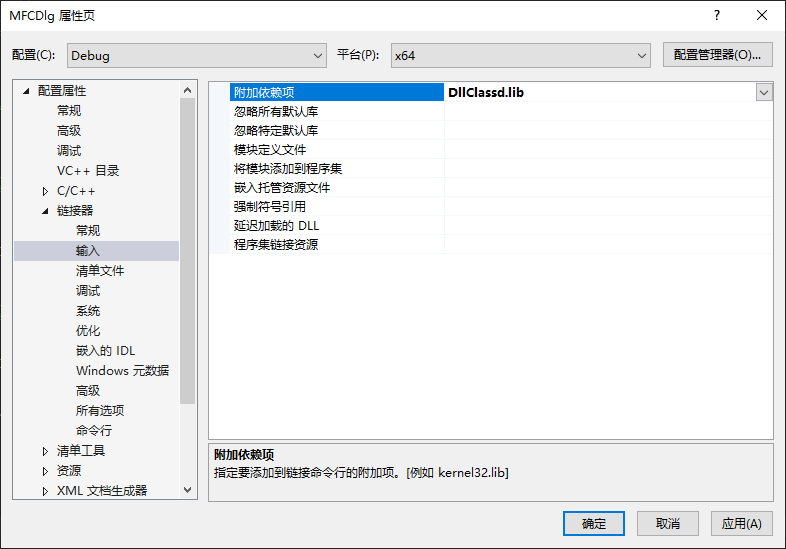
生成的文件：DllClass86.dll、DllClass86.exp、DllClass86.lib、DllClass86.pdb。



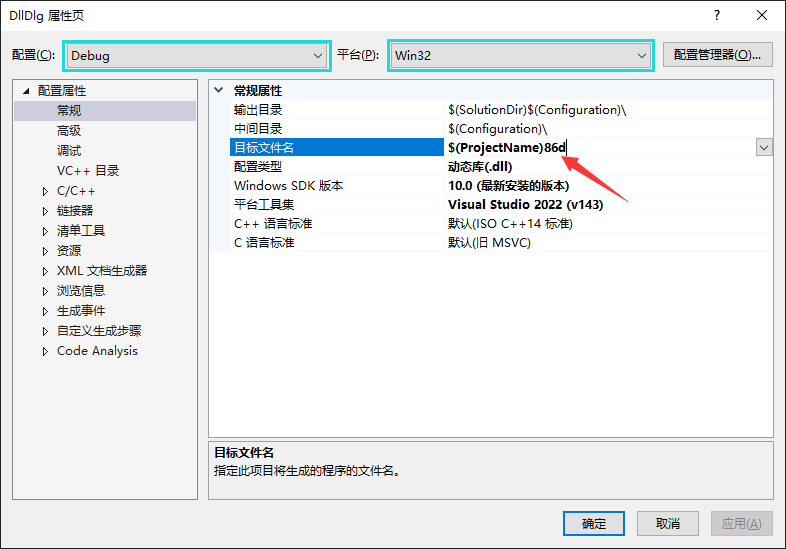
* Debug/x64



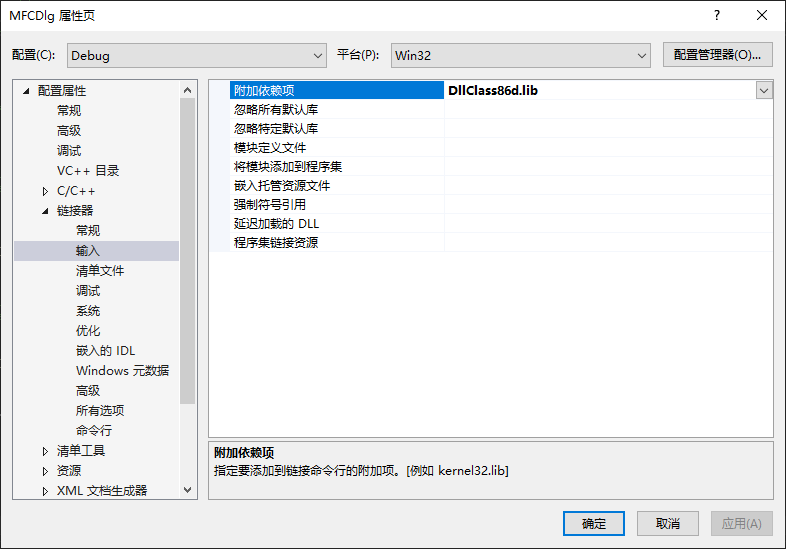
生成的文件：DllClassd.dll、DllClassd.exp、DllClassd.lib、DllClassd.pdb。



* Debug/x86



生成的文件：DllClass86d.dll、DllClass86d.exp、DllClass86d.lib、DllClass86d.pdb。



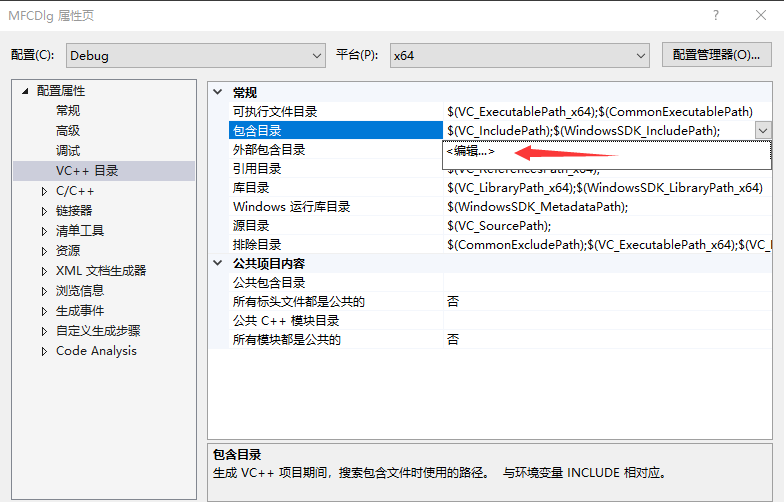
通过类似以上的处理方式，生成目标文件时在文件名后面添加一个后缀，以区分不同的版本。客户端程序的项目属性中，不同版本的“附加依赖项”对应给出相应版本的引入库文件名。最后，将不同版本的dll文件配置相应版本的客户端程序，保证编译、链接、运行时不同版本文件的匹配。

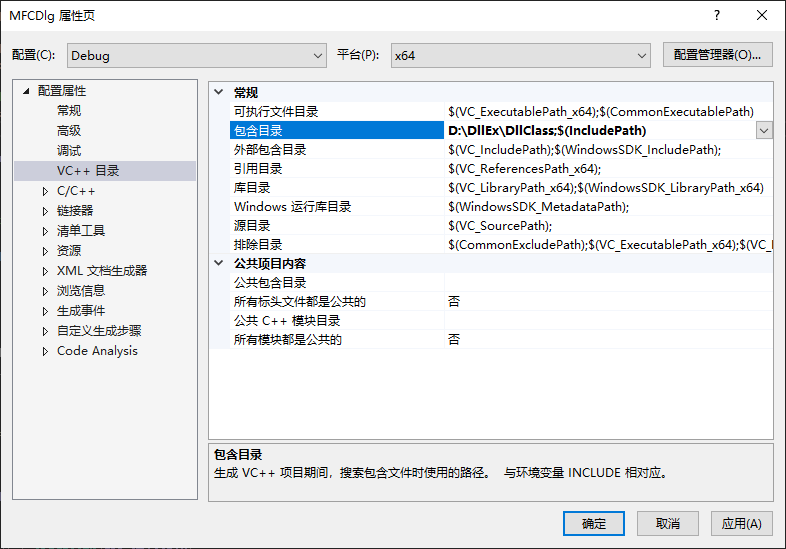
1. 项目属性包含目录、引用目录设置

隐匿链接方式加载DLL时，客户端应用程序，编译时可能需要DLL的头文件，链接时需要DLL的引入库文件。

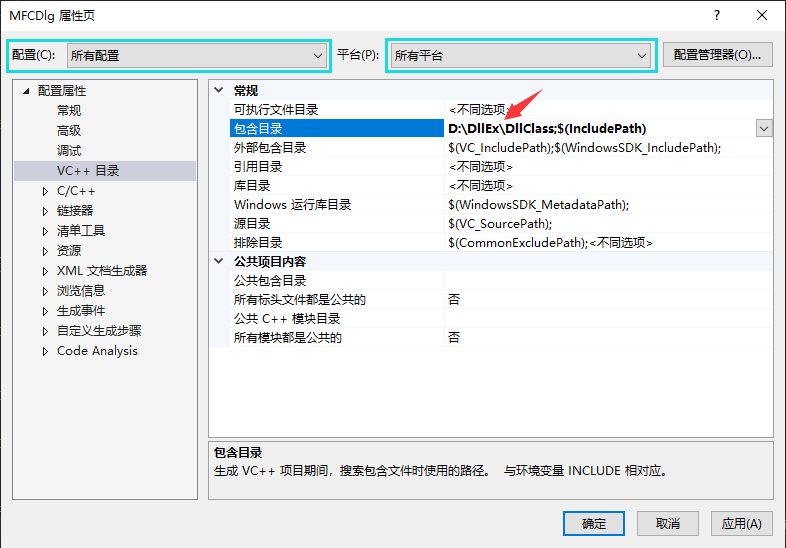
* 头文件

可以在包含头文件时指定文件路径，也可以在客户端项目的属性设置里指定包含目录。



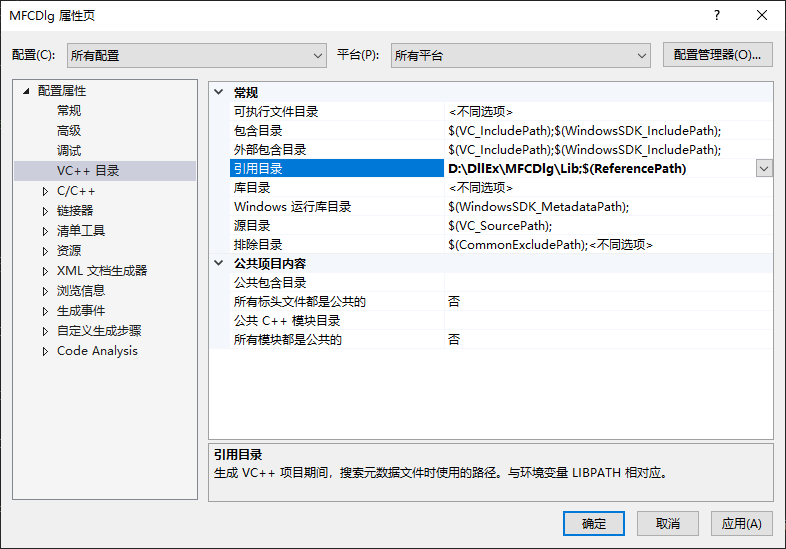


因为对于头文件，Debug、Release、X64、X86都是统一的，因此可以如下图，在“配置”、“平台”中选择“所有配置”与“所有平台”，即一次完全四种情况的统一设置。



* 引入库文件

DLL的lib、dll文件和客户端程序的版本、运行平台必须一致，因此可以为客户端项目的项目属性分不同“配置”、“平台”指定不同的引入库文件目录。也可以指定统一的引入库文件目录，但这时最好是如第8节所述，将不同版本的lib文件通过加不同的后缀以区分。



* dll文件

VS中调试运行程序，需要加载DLL时，会按照如下顺序搜索dll文件：

* + exe的执行目录

如，D:\DllEx\x64\Debug

* + 当前目录

如，D:\DllEx\MFCDlg

* + 系统目录

C:\Windows\system32 、C:\Windows\system 、C:\Windows等。

* + Windows环境变量path中所列出的路径

因此如果有必要，可以通过配置Windows环境变量path来指定dll文件所在目录。