

2020/07/16_Windows编程_第1课_静态库和动态库

笔记本: Windows编程
创建时间: 2020/7/16 星期四 10:05
作者: ileemi
标签: 静态库和动态库

- [静态库和动态库](#)
- [库的设计](#)
 - [静态库](#)
 - [添加库](#)
 - [使用创建好的库](#)
 - [静态库的通用](#)
 - [IDE中做静态库](#)
 - [动态库 \(动态链接库、共享库、DLL -- Dynamic link library\)](#)
 - [动态库的编写](#)
 - [动态链接库的使用](#)
 - [导出函数](#)
 - [查看导出函数是否导出成功](#)
 - [获取导出函数的地址](#)
 - [卸载, 释放不使用动态链接库](#)
 - [.def文件](#)
 - [导出全局变量](#)
 - [导出类](#)

静态库和动态库

操作系统相关的内容 (系统相关的内容 (内部机制相同)), C库是跨操作系统的。

库的设计

程序的开发都是按照模块进行的, 大型项目, 一个模块一个cpp, 单独做。

静态库

命令行编译.c/.cpp -- 打开VS的本地工具命令提示符, cd 进入到需要编译链接的文件夹内。

编译: 1. cl /c xxx.c(xxx.cpp) 2. cl /c * .cpp

链接: 1. link xxx.obj xxx.obj 2. link * .obj

大多数情况下链接（合并.obj文件）要比编译（检查语法）的效率快。

一般软件一个模块一个.obj文件

将所有同类型功能的 .obj 文件进行打包，用的时候链接器直接从打包好的总 obj 文件中抽取对应的模块进行链接（这个库的功能由编译器去做），库的格式由链接器决定。不同编译器做出的库格式会不一样。现在库的格式就是各个编译器各做各的。每个编译器都提供了库格式的工具。

通用对象文件格式：**COFF**

windows静态库格式的处理工具：-- **LIB** 工具（专门用来做库的）

```
D:\CR37\Works\第二阶段\Windows编程\Codes\20200716 - 静态库和动态库\lib>LIB
Microsoft (R) Library Manager Version 14.26.28806.0
Copyright (C) Microsoft Corporation. All rights reserved.

用法: LIB [选项] [文件]

选项:
    /DEF[: 文件名]
    /ERRORREPORT: {NONE|PROMPT|QUEUE|SEND}
    /EXPORT: 符号
    /EXTRACT: 成员名
    /INCLUDE: 符号
    /LIBPATH: 目录
    /LINKREPRO: dir
    /LINKREPROTARGET: filename
    /LIST[: 文件名]
    /LTCG
    /MACHINE: {ARM|ARM64|ARM64EC|EBC|X64|X86}
    /NAME: 文件名
    /NODEFAULTLIB[: 库]
    /NOLOGO
    /OUT: 文件名
    /REMOVE: 成员名
    /SUBSYSTEM: {BOOT_APPLICATION|CONSOLE|EFI_APPLICATION|
        EFI_BOOT_SERVICE_DRIVER|EFI_ROM|EFI_RUNTIME_DRIVER|
        NATIVE|POSIX|WINDOWS|WINDOWSCE} [, #[: #]]
    /VERBOSE
(按回车键继续)

D:\CR37\Works\第二阶段\Windows编程\Codes\20200716 - 静态库和动态库\lib>lib /LIST conert.lib
Microsoft (R) Library Manager Version 14.26.28806.0
Copyright (C) Microsoft Corporation. All rights reserved.

CONCRT140.dll
CONCRT140.dll
CONCRT140.dll
CONCRT140.dll
CONCRT140.dll
CONCRT140.dll
CONCRT140.dll
CONCRT140.dll
CONCRT140.dll
CONCRT140.dll
```

遍历库中的文件

C库 LIBC.LIB

添加库

使用 lib 的 **/OUT** 创建库，并添加文件到库中，再次添加会覆盖原来库中的数据文件，命令如下：

```
lib /out:demo.lib *.obj
```

```
lib /out:demo.lib module1.obj module2.obj
```

```
D:\CR37\Works\第二阶段\Windows编程\Codes\20200716 - 静态库和动态库\lib>lib /OUT:demo.lib module1.obj module2.obj
Microsoft (R) Library Manager Version 14.26.28806.0
Copyright (C) Microsoft Corporation. All rights reserved.

D:\CR37\Works\第二阶段\Windows编程\Codes\20200716 - 静态库和动态库\lib>lib /LIST demo.lib
Microsoft (R) Library Manager Version 14.26.28806.0
Copyright (C) Microsoft Corporation. All rights reserved.

module1.obj
module2.obj
```

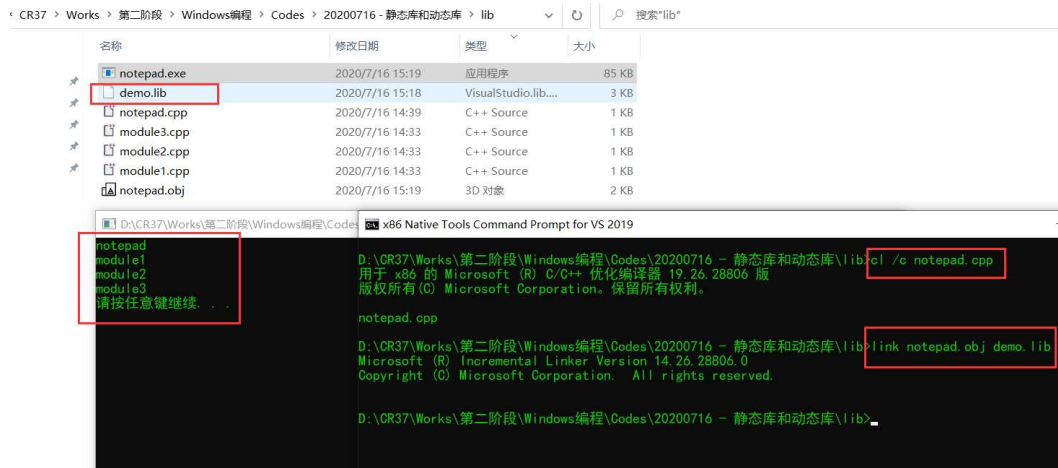
创建一个.lib库，同时添加对应的.obj文件到库中

使用创建好的库

链接文件文件时，添加编译好的lib库即可，命令如下：

```
cl /c notepad.cpp
```

```
link notepad.obj demo.lib
```

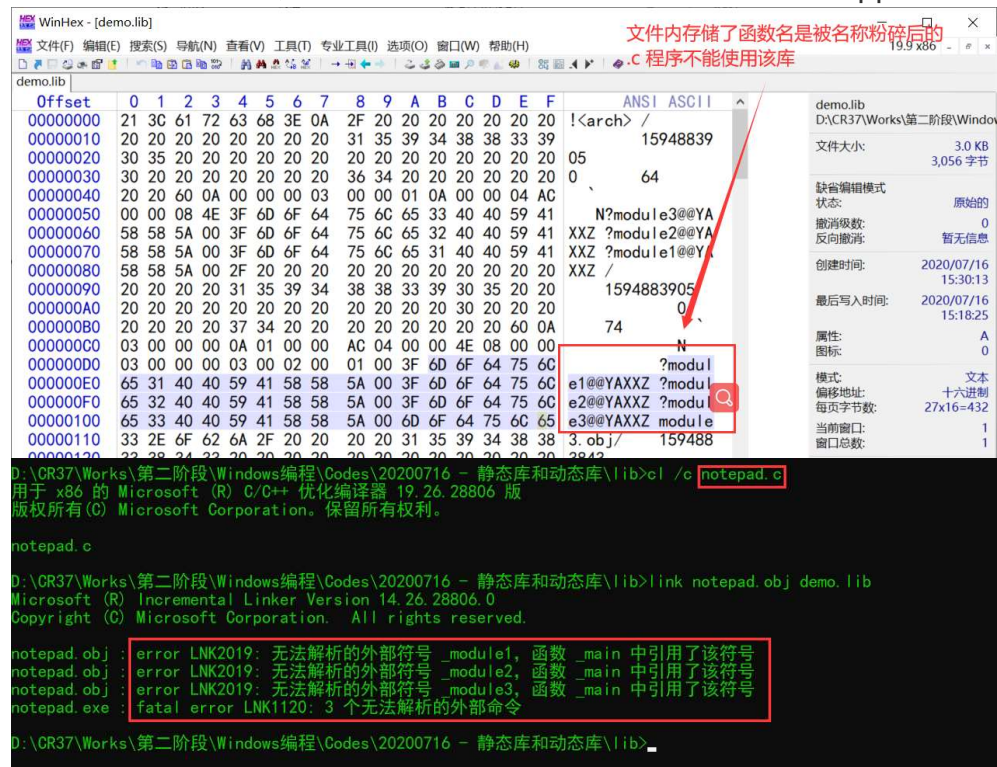


一般情况下，一个库会配备一个头文件，库单独做，哪里需要使用，就将做好的库拿过去给他人使用。头文件方式模块的声明，以及介绍。

静态库的通用

问题：

.c 的程序不能使用函数名称粉碎后的lib库，名称粉碎后的lib库只能.cpp使用



上述问题的解决办法：

1. 在每个模块的函数名前添加 **extern "C"** -- 不进行名称粉碎

```
#include <stdio.h>

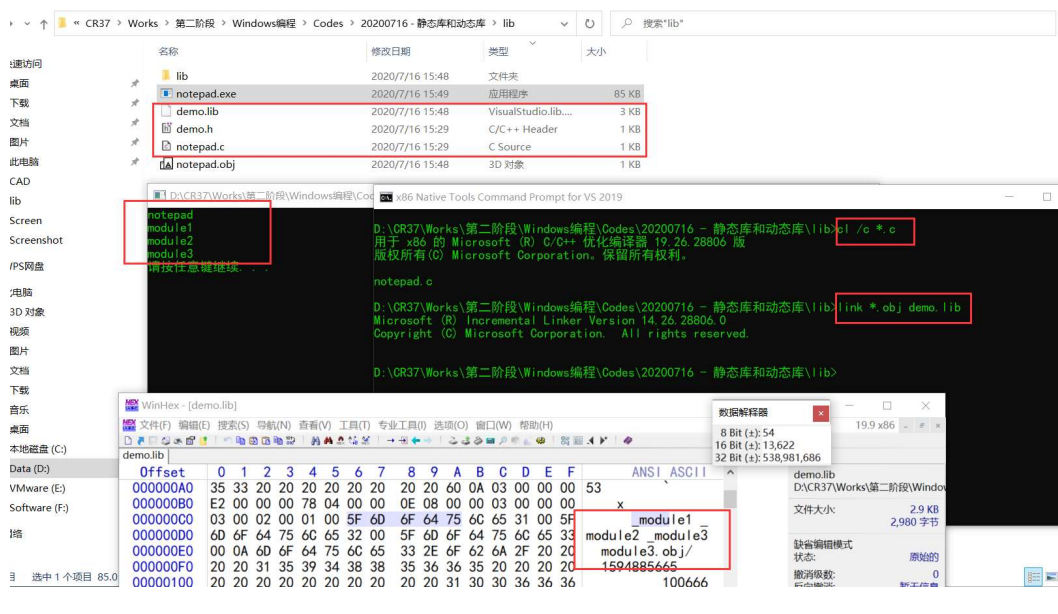
extern "C" void module1()
{
```

```
puts("module1");
}
```

2. 重新将模块进行打包，打包后下面的 .c 文件就可以正常使用

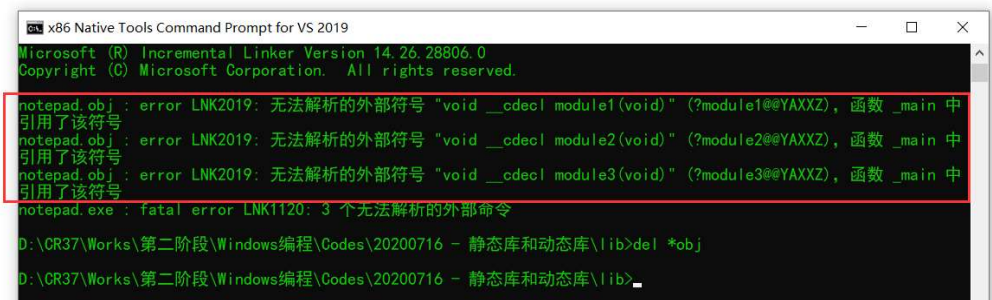
```
#include <stdio.h>
#include <stdlib.h>
#include "demo.h"

int main(int argc, char* argv[])
{
    puts("notepad");
    module1();
    module2();
    module3();
    system("pause");
    return 0;
}
```



但是依然存在问题：.cpp的文件就不能使用这个静态库，库不通用。

lib	2020/7/16 15:52	文件夹	
run.bat	2020/7/16 15:54	Windows 批处理...	1 KB
demo.lib	2020/7/16 15:48	VisualStudio.lib...	3 KB
notepad.cpp	2020/7/16 15:29	C++ Source	1 KB
demo.h	2020/7/16 15:29	C/C++ Header	1 KB



头文件添加extern "C"

```
// __cplusplus 宏是一个标准, 表示 .cpp 文件
#ifdef __cplusplus
    extern "C"
    {
#ifdef __cplusplus
        void module1();
        void module2();
        void module3();
        int g_nNum; // 模块中的全局变量
#ifdef __cplusplus
    }
#endif
#endif
```

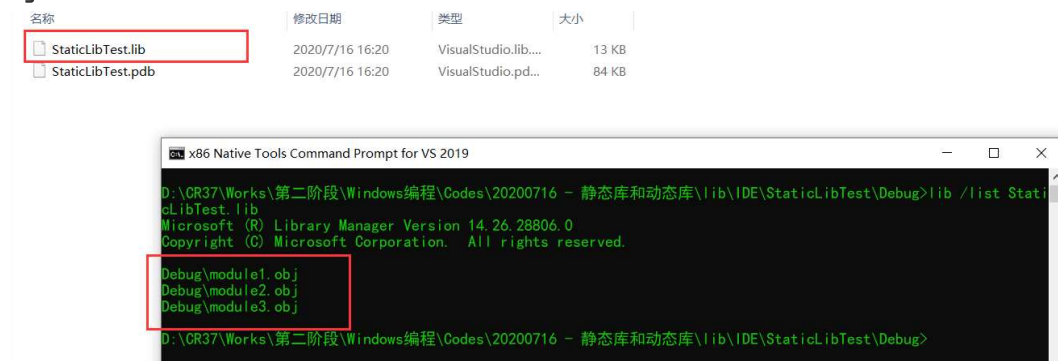
命令行需要添加编译选项, 同时也可以在使用静态库的文件中使用 #pragma comment 命令来包含静态库文件, 使用这个命令后, 再编译的时候就不需要添加静态库名了:

示例 #pragma comment(lib, "mou.lib")

IDE中做静态库

模块中的全局变量也可以做到库中, 需要在库的头文件中声明

打开IDE -- 新建静态库工程 -- 将需要添加到库中的文件放入到工程内 -- 点击运行即可



如何使用:

在需要使用静态库的工程中将库各头文件添加到功能内, 添加编译选项

1. 可以使用 #pragma comment
2. 项目 --> 链接器 --> 所有选项 --> 附加依赖项 (将需要使用的库文件添加进去)

以C++类的形似做静态库, 这个库只能C++的工程可以使用, 模块函数前以及头文件不需要添加extern. 类的声明也要添加到头文件中。

动态库 (动态链接库、共享库、DLL -- Dynamic link library)

静态库存在的问题：

1. 使用静态库的软件有很多的时候，静态库文件中的某个模块存在一个Bug，模块的Bug修复后，所有使用该静态库的软件都需要重新编译链接，可维护性不高（使用静态库程序的程序会加载静态库中所使用的代码到可执行文件中去）。

解决办法：

当可执行程序运行起来的时候，将库内的代码拷贝到内存。库中的模块出现Bug，只需要库进行更新即可。这种做法就是动态链接库的做法。

解决静态库维护性问题，维护性好，占用空间小，但是放模块较多的时候启动速度慢，动态库存在安全问题。

动态库的编写

Link /DLL xxx.obj

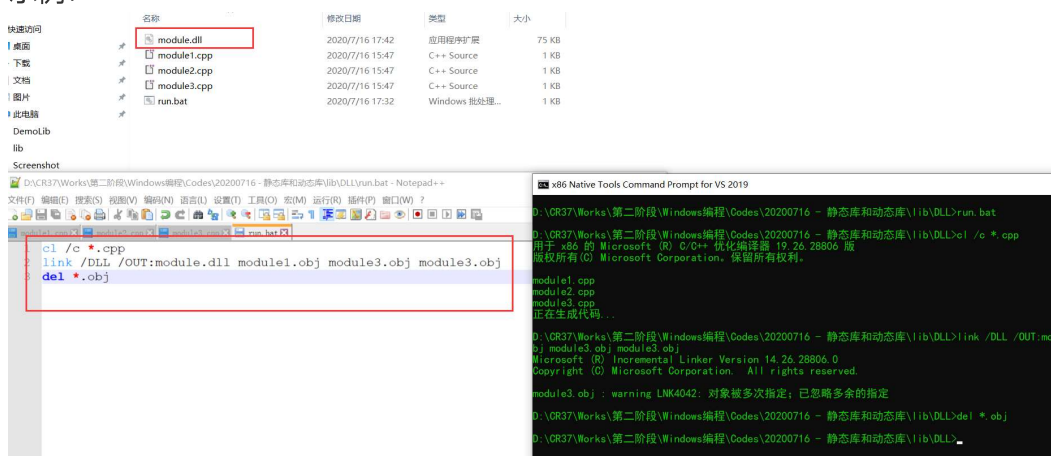
链接后的库就是动态库，静态库用 lib 做，动态库用 Link 做
按照分类做对应的DLL

静态库和动态库的区别：

- 静态库中的模块使用的时候可以一个一个的抽取出来
- 动态库中的模块使用的时候不能一个一个的抽取出来
a.exe 使用 module1, b.exe 使用 module2, module3
demo.dll 中有模块必须要有 module1, module2, module3, 两个exe程序在启动的时候，会将dll中的模块全部加载到自己的内存中。

动态库在操作系统中可以认为是一个应用程序，手动编译代码如下：

示例：



The screenshot shows a Windows file explorer window with the following files and folders:

名称	修改日期	类型	大小
module.dll	2020/7/16 17:42	应用程序扩展	75 KB
module1.cpp	2020/7/16 15:47	C++ Source	1 KB
module2.cpp	2020/7/16 15:47	C++ Source	1 KB
module3.cpp	2020/7/16 15:47	C++ Source	1 KB
run.bat	2020/7/16 17:32	Windows 批处理...	1 KB

Below the file explorer, a command prompt window shows the following commands and output:

```
cl /c *.cpp
link /DLL /OUT:module.dll module1.obj module3.obj module3.obj
del *.obj
```

The output of the linker command is:

```
module1.cpp
module2.cpp
module3.cpp
正在生成代码...
D:\CR37\Works\第二阶段\Windows编程\Codes\20200716 - 静态库和动态库\lib\DLL>link /DLL /OUT:module.dll module1.obj module3.obj module3.obj
Microsoft (R) Incremental Linker Version 14.26.28806.0
Copyright (C) Microsoft Corporation. All rights reserved.
module3.obj : warning LNK4042: 对象被多次指定; 已忽略多余的指定
D:\CR37\Works\第二阶段\Windows编程\Codes\20200716 - 静态库和动态库\lib\DLL>del *.obj
D:\CR37\Works\第二阶段\Windows编程\Codes\20200716 - 静态库和动态库\lib\DLL>
```

动态链接库的使用

用法：

1. 动态加载（动态加载）：
为了支持动态链接 操作系统提供了一个API -- **LoadLibrary**（加载一个库到使用的文件中）
参数：指定路径下的模块，举对路径

返回值：返回一模块句柄，需要包含头文件#include <Windows.h>，**指向动态链接库加载到内存中的首地址。**

代码示例：

```
// 动态连接
HMODULE hDll = LoadLibrary("D:\CR37\Works\第二阶段\Windows编程
\Codes\20200716 - 静态库和动态库\Lib\DLL");

// 当不在需要使用的时候可以使用FreeLibrary() 进行释放
FreeLibrary(hDll);
```

2. 获取动态链接库中的模块（获取函数），由于动态链接库中的模块地址不确定，但是模块在DLL中的偏移地址是确定的。**模块加载到内存的地址就 == 动态链接库加载到内存中的首地址+模块在库中的偏移地址**，由于函数地址和偏移不确定 $\text{base} + \text{offset}$ （不确定），**所以就需要将函数的偏移地址保存到dll文件中**。之后操作系统就可以根据记录的地址算出对应模块在内存中的地址。
3. 将模块的偏移地址记录到 DLL 中，在 Windows 中叫做 **"导出函数"**，将需要使用的函数声明为导出函数，其格式由操作系统和编译器定义的，编译器会提供一个设计导出函数的方法

导出函数

__declspec

在对应的函数名前添加，**__declspec (dllexport)** 告诉编译器该函数是一个导出函数。

代码示例：

```
extern "C" __declspec(dllexport) void module1()
{
    puts("module1");
}
```

查看导出函数是否导出成功

dumpbin /EXPORTS xxx.dll -- 查看导出函数是否成功。

使用示例：

```
x86_x64 Cross Tools Command Prompt for VS 2019
D:\CR37\Works\第二阶段\Windows编程\Codes\20200716 - 静态库和动态库\lib\DLL>dumpbin /exports module.dll

Microsoft (R) COFF/PE Dumper Version 14.26.28806.0
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file module.dll
File Type: DLL

Section contains the following exports for module.dll

 00000000 characteristics
 FFFFFFFF time_date_stamp
 0.00 version
 1 ordinal base
 8 number of functions
 8 number of names

ordinal hint RVA      name
1 0 00001020 ??4CTest@@QAEAAV0@@$QAV0@@Z
2 1 00001020 ??4CTest@@QAEAAV0@@$ABV0@@Z
3 2 00001030 ?Fun1@CTest@@QAE@XZ
4 3 00001050 ?Fun2@CTest@@QAE@XZ
5 4 00018000 g_nNum
6 5 00001000 module1
7 6 000010F0 module2
8 7 00001110 module3

Summary
 2000 .data
 7000 .rdata
10000 .reloc
10000 .text

D:\CR37\Works\第二阶段\Windows编程\Codes\20200716 - 静态库和动态库\lib\DLL>
```

获取导出函数的地址

GetProcAddress //参数1：DLL模块句柄 参数2：函数名

返回值：如果函数成功，返回值是导出函数或变量的地址。使用示例：

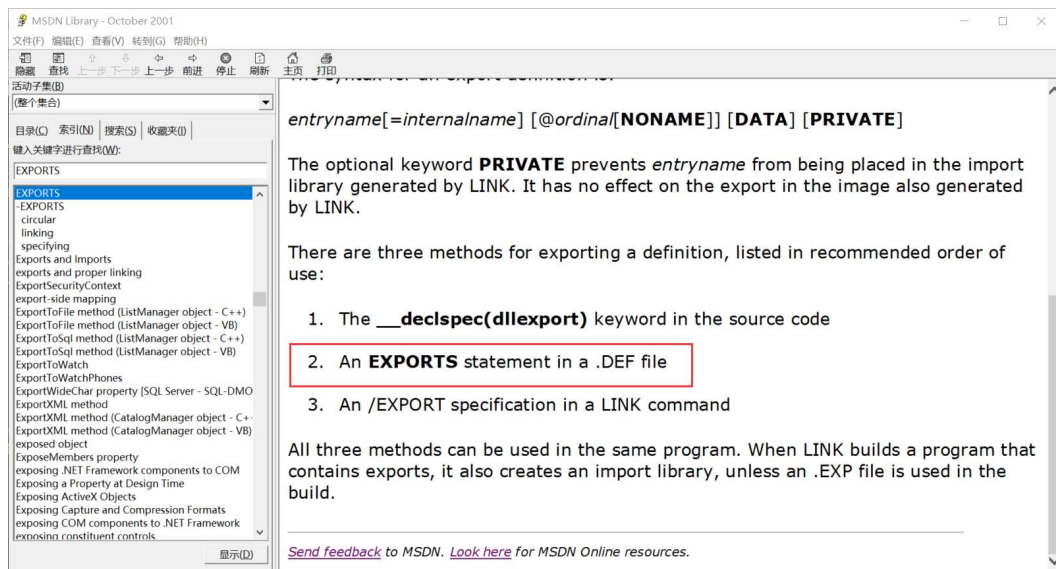
```
notepad.cpp
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <windows.h>
4
5 typedef void (*FUNTYPE)(); // 函数指针
6
7 int main(int argc, char* argv[])
8 {
9     // 动态链接
10    HMODULE hDll = LoadLibrary("D:\\CR37\\Works\\第二阶段\\Windows编程\\Codes\\20200716 - 静态库和动态库\\lib\\DLL\\module.dll");
11    printf("notepad hDll=%p\n", hDll);
12    // 函数地址不确定 base + offset
13    // 获取导出函数地址
14    FUNTYPE pfnMou1 = (FUNTYPE)GetProcAddress(hDll, "module1");
15    printf("pfnMou1=%p\n", pfnMou1);
16    if (pfnMou1 != NULL)
17    {
18        pfnMou1();
19    }
20 }
21
22
```

卸载，释放不使用动态链接库

FreeLibrary(xxx); // 参数为LoadLibrary返回的句柄

.def文件

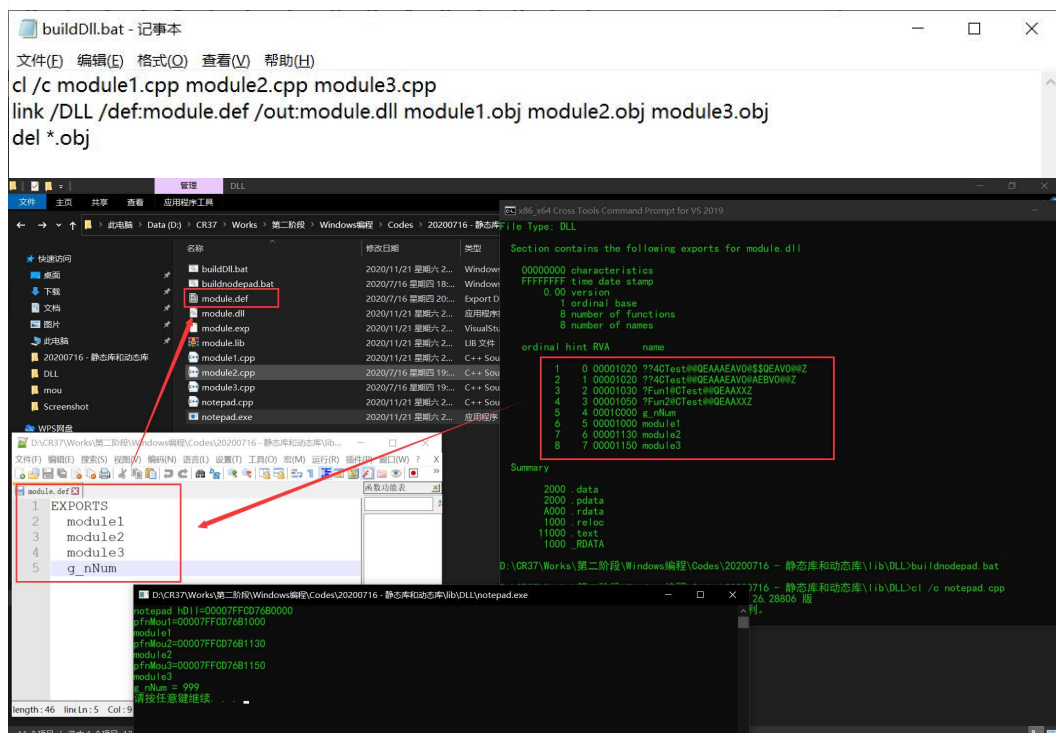
每次导出函数都要在对应的函数名前添加 `__declspec (dllexport)`，为此微软提供了一个名为xxx.def的文件，该文件写导出函数的函数名。



使用方式：不需要在导出的函数名前添加 `__declspec (dllexport)`，在工程目录下新建一个 `xxx.def` 文件，文件开头添加 `EXPORTS`，换行输入需要导出的函数名即可，在链接dll时需要将 `xxx.edf` 文件添加上，可以通过 `dumpbin /exports xxx.dll` 查看导出的函数名。

使用示例：

```
17  /*extern "C" __declspec(dllexport)*/ void module1()
18  {
19      puts("module1");
20  }
```



导出全局变量

同理，在对应的模块中添加全局变量

在 `xxx.def` 中声明，同时在使用地方通过 `GetProcAddress` 获取全局变量的地址，通过返回在内存中的地址，取出内存上的地址即可。

The screenshot displays a Windows desktop with three open windows:

- Top Left Window (Dllvnotepad.cpp):** A C++ source file. It includes `<windows.h>` and `<string.h>`. The `main` function calls `GetProcAddress` to find the address of `g_nNum` in `module1.dll` and prints it. The code is as follows:


```

34 if (pfnMou3 != NULL)
35 {
36     pfnMou3();
37 }
38
39 int *pg_nNum = (int*)GetProcAddress(hDll, "g_nNum");
40 if (pg_nNum != NULL)
41 {
42     printf("g_nNum = %d\r\n", *pg_nNum);
43 }
      
```
- Top Right Window (module1.hpp):** A header file for `module1.dll`. It declares `g_nNum` as a constant integer with the value 999.


```

1 #include <stdio.h>
2
3 int g_nNum = 999;
4
5 /*extern "C" __declspec ( dllexport ) 8*/ void module1()
6 {
7     puts("module1");
8 }
      
```
- Bottom Window (Command Prompt):** A command prompt window titled "D:\C37\Works\第三阶段\Windows编程\Codes\20200716 - 静态库和动态库\lib\DLL\vnnotepad.exe". It shows the execution of the program, which prints the address of `g_nNum` in `module1.dll`.


```

notepad h011=7AE50000
pfnMou1=7AE51000
module1
pfnMou2=7AE51020
module2
pfnMou3=7AE51040
module3
g_nNum = 999
请按任意键继续...
      
```

.def 不支持导出类，导出类必须在类前先添加关键字 `__declspec(dllexport)`

The screenshot illustrates the process of creating a DLL in Visual Studio 2019. The top window displays the source code for 'module1.cpp', which includes a header file and defines two functions, 'Fun1' and 'Fun2', using the 'dllexport' attribute. The bottom window shows the 'x86 Native Tools Command Prompt for VS 2019' with the command 'cl /LD /c module1.cpp' and its output. The output includes a table of exports for 'module1' and 'module2', which are the functions defined in the source code. A red arrow points to the 'Fun1' and 'Fun2' entries in the table, highlighting the exported functions.

使用示例：

```

1  #include <stdio.h>
2
3  int g_nNum = 999;
4  __declspec(dllexport) CTest
5  {
6  public:
7      void Fun1()
8      {
9          printf("Fun1 ()");
10     }
11     void Fun2()
12     {
13         printf("Fun2 ()");
14     }
15 };
16 /*extern "C" __declspec ( dllexport ) &*/
17 puts("module1");
18 }

```

编译命令：

```

cl /LD /c module1.cpp

```

输出结果：

```

***** time date stamp
0.00 version
1 ordinal base
8 number of functions
8 number of names

ordinal hint RVA      name
1 0 00001020 P?4CTest@QAEA0V0$SQA0W0#Z
2 1 00001020 P?4CTest@QAEA0V0$SQA0W0#Z
3 2 00001030 F?Fun1CTest@QAE0XZ
4 3 00001050 F?Fun2CTest@QAE0XZ
5 4 00018000 g_nNum
6 5 00001000 module1
7 6 000010F0 module2
8 7 00001110 module3

Summary
2000 .data
7000 .rdata
1000 .reloc
10000 .text

```

notepad.cpp

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

typedef void (*FUNTYPE)();           // 函数指针

int main(int argc, char* argv[])
{
    //动态链接
    HMODULE hDll = LoadLibrary("D:\\CR37\\Works\\第二阶段\\Windows编程\\Codes\\20200716 - 静态库和动态库\\lib\\DLL\\module.dll");
    printf("notepad hDll=%p\n", hDll);

    //函数地址不确定 base + offset
```

```
//获取导出函数地址
FUNTYPE pfnMou1 = (FUNTYPE)GetProcAddress(hDll, "module1");
printf("pfnMou1=%p\n", pfnMou1);
if (pfnMou1 != NULL)
{
    pfnMou1();
}

FUNTYPE pfnMou2 = (FUNTYPE)GetProcAddress(hDll, "module2");
printf("pfnMou2=%p\n", pfnMou2);
if (pfnMou2 != NULL)
{
    pfnMou2();
}

FUNTYPE pfnMou3 = (FUNTYPE)GetProcAddress(hDll, "module3");
printf("pfnMou3=%p\n", pfnMou3);
if (pfnMou3 != NULL)
{
    pfnMou3();
}

int *pg_nNum = (int*)GetProcAddress(hDll, "g_nNum");
if (pg_nNum != NULL)
{
    printf("g_nNum = %d\r\n", *pg_nNum);
}

//module1();
//module2();
//module3();

//卸载
FreeLibrary(hDll);

system("pause");
return 0;
}
```