# 学习大纲

|  |
| --- |
|  |

## 第一种方式我们上一节已经学习了，这里我们来学习第二种方式

## LoadLibrary函数的用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 将指定的模块加载到调用进程的地址空间中。 指定的模块可能会导致加载其他模块。  有关其他加载选项，请使用 [LoadLibraryEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibraryexa) 函数。 语法 C++复制  HMODULE LoadLibraryA(  [in] LPCSTR lpLibFileName  ); parameters [in] lpLibFileName  模块的名称。 可以是库模块 (.dll 文件) ，也可以是可执行模块 (.exe 文件) 。 如果指定的模块是可执行模块，则不会加载静态导入;相反，模块就像由 [LoadLibraryEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibraryexa) 使用 标志一 DONT\_RESOLVE\_DLL\_REFERENCES 样加载。  指定的名称是模块的文件名，与库模块本身中存储的名称无关，由 **module-definition** (.def) 文件中的 LIBRARY 关键字 (keyword) 指定。  如果字符串指定完整路径，则函数仅搜索模块的该路径。  如果字符串指定相对路径或模块名称而不指定路径，则函数使用标准搜索策略来查找模块;有关详细信息，请参阅备注。  如果函数找不到模块，则函数将失败。 指定路径时，请确保使用反斜杠 (\) ，而不是 (/) 正斜杠。 有关路径的详细信息，请参阅 [命名文件或目录](https://learn.microsoft.com/zh-cn/windows/desktop/FileIO/naming-a-file)。  如果字符串指定模块名称而不指定路径，并且省略了文件扩展名，则函数会将默认库扩展名“.DLL”追加到模块名称。 若要防止函数将“.DLL”追加到模块名称，请在模块名称字符串中包含尾随点字符 (.) 。 返回值 如果函数成功，则返回值是模块的句柄。  如果函数失败，则返回值为 NULL。 要获得更多的错误信息，请调用 GetLastError。 注解 若要启用或禁用加载程序在 DLL 加载期间显示的错误消息，请使用 [SetErrorMode](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-seterrormode) 函数。  **LoadLibrary** 可用于将库模块加载到进程的地址空间中，并返回一个句柄，该句柄可用于 [在 GetProcAddress 中](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-getprocaddress) 获取 DLL 函数的地址。 **LoadLibrary** 还可用于加载其他可执行模块。 例如，函数可以指定 .exe 文件以获取可在 [FindResource](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-findresourcea) 或 [LoadResource](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-loadresource) 中使用的句柄。 但是，请勿使用 **LoadLibrary** 运行 .exe 文件。 请改用 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) 函数。  如果指定的模块是尚未为调用进程加载的 DLL，则系统会使用**DLL\_PROCESS\_ATTACH**值调用 DLL 的 [DllMain](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/dllmain) 函数。 如果 **DllMain** 返回 **TRUE**， **则 LoadLibrary** 将返回模块的句柄。 如果 **DllMain** 返回 **FALSE**，则系统会从进程地址空间中卸载 DLL， **而 LoadLibrary** 返回 **NULL**。 从 **DllMain** 调用 **LoadLibrary** 是不安全的。 有关详细信息，请参阅 **DllMain** 中的“备注”部分。  模块句柄不是全局句柄，也不是可继承的。 一个进程调用 **LoadLibrary** 不会生成另一个进程可以使用的句柄，例如，在调用 [GetProcAddress](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-getprocaddress) 时。 在调用 **GetProcAddress** 之前，另一个进程必须自行调用模块的 **LoadLibrary**。  如果 *lpFileName* 不包含路径，并且有多个具有相同基名称和扩展名的已加载模块，则函数将返回首先加载的模块的句柄。  如果未在 *lpFileName* 参数中指定文件扩展名，则会追加默认库扩展名 .dll。 但是，文件名字符串可以包含尾随点字符 (.) 以指示模块名称没有扩展名。 如果未指定路径，函数将搜索其基名称与要加载的模块的基名称匹配的已加载模块。 如果名称匹配，则加载成功。 否则，函数将搜索 文件。  搜索的第一个目录是包含用于创建调用进程 (的图像文件的目录，有关详细信息，请参阅 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) 函数) 。 这样做可以在不将进程的已安装目录添加到 PATH 环境变量的情况下，找到与进程关联的专用动态链接库 (DLL) 文件。 如果指定了相对路径，则整个相对路径将追加到 DLL 搜索路径列表中的每个标记。 若要从相对路径加载模块而不搜索任何其他路径，请使用 [GetFullPathName](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-getfullpathnamea) 获取非关系路径，并使用非关系路径调用 **LoadLibrary** 。 有关 DLL 搜索顺序的详细信息，请参阅 [动态链接库搜索顺序](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/dynamic-link-library-search-order)。  可以使用 [SetDllDirectory](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-setdlldirectorya) 函数更改搜索路径。 建议使用此解决方案，而不是使用 [SetCurrentDirectory](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-setcurrentdirectory) 或硬编码 DLL 的完整路径。  如果指定了路径，并且应用程序有重定向文件，则函数在应用程序的目录中搜索模块。 如果模块存在于应用程序的目录中， **则 LoadLibrary** 将忽略指定的路径，并从应用程序的目录中加载该模块。 如果应用程序的目录中不存在模块， **则 LoadLibrary** 从指定的目录加载模块。 有关详细信息，请参阅 [动态链接库重定向](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/dynamic-link-library-redirection)。  如果使用程序集的名称调用 **LoadLibrary** ，但没有路径规范，并且程序集列在系统兼容清单中，则调用会自动重定向到并行程序集。  系统在所有加载的模块上维护每进程引用计数。 调用 **LoadLibrary** 会递增引用计数。 调用 [FreeLibrary](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-freelibrary) 或 [FreeLibraryAndExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-freelibraryandexitthread) 函数会减少引用计数。 当模块的引用计数达到零或进程终止时，无论引用计数) 如何，系统都会卸载模块 (。  **Windows Server 2003 和 Windows XP：**Visual C++ 编译器支持用于声明线程局部变量的语法： **\_declspec (线程)**。 如果在 DLL 中使用此语法，将无法在 Windows Vista 之前的 Windows 版本上使用 **LoadLibrary** 显式加载 DLL。 如果 DLL 将被显式加载，则必须使用线程本地存储函数，而不是 **\_declspec (线程)**。 有关示例，请参阅 [在动态链接库中使用线程本地存储](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/using-thread-local-storage-in-a-dynamic-link-library)。 安全备注 请勿使用 [SearchPath](https://learn.microsoft.com/zh-cn/windows/desktop/api/processenv/nf-processenv-searchpathw) 函数检索 DLL 的路径，以便进行后续 **LoadLibrary** 调用。 **SearchPath** 函数使用的搜索顺序与 **LoadLibrary** 不同，并且它不使用安全进程搜索模式，除非通过调用**具有 BASE\_SEARCH\_PATH\_ENABLE\_SAFE\_SEARCHMODE** 的 [SetSearchPathMode](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-setsearchpathmode) 显式启用此模式。 因此， **SearchPath** 可能首先搜索用户的当前工作目录以查找指定的 DLL。 如果攻击者将恶意的 DLL 版本复制到当前工作目录中， **SearchPath** 检索到的路径将指向恶意 DLL， **然后 LoadLibrary** 将加载该 DLL。  不要基于搜索 DLL 的 **LoadLibrary** 调用来假设操作系统版本。 如果应用程序在 DLL 合法不存在的环境中运行，但该 DLL 的恶意版本位于搜索路径中，则可能会加载该 DLL 的恶意版本。 请改用 [获取系统版本](https://learn.microsoft.com/zh-cn/windows/desktop/SysInfo/getting-the-system-version)中所述的推荐技术。 示例 有关示例，请参阅 [使用 Run-Time 动态链接](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/using-run-time-dynamic-linking)。  **备注**  libloaderapi.h 标头将 LoadLibrary 定义为别名，该别名根据 UNICODE 预处理器常量的定义自动选择此函数的 ANSI 或 Unicode 版本。 将非特定编码别名的使用与非非特定编码的代码混合使用可能会导致不匹配，从而导致编译或运行时错误。 有关详细信息，请参阅 [**函数原型的约定**](https://learn.microsoft.com/zh-cn/windows/win32/intl/conventions-for-function-prototypes)。 要求  |  |  | | --- | --- | | **标头** | libloaderapi.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [DllMain](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/dllmain)  [动态链接库函数](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/dynamic-link-library-functions)  [FindResource](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-findresourcea)  [FreeLibrary](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-freelibrary)  [GetProcAddress](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-getprocaddress)  [GetSystemDirectory](https://learn.microsoft.com/zh-cn/windows/desktop/api/sysinfoapi/nf-sysinfoapi-getsystemdirectorya)  [GetWindowsDirectory](https://learn.microsoft.com/zh-cn/windows/desktop/api/sysinfoapi/nf-sysinfoapi-getwindowsdirectorya)  [LoadLibraryEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibraryexa)  [LoadResource](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-loadresource)  [运行时动态链接](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/run-time-dynamic-linking)  [SetDllDirectory](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-setdlldirectorya)  [SetErrorMode](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-seterrormode) |

## GetProcAddress函数的用法

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 从指定的动态链接库 (DLL) 检索导出函数 (也称为过程) 或变量的地址。 语法 C++复制  FARPROC GetProcAddress(  [in] HMODULE hModule,  [in] LPCSTR lpProcName  ); 参数 [in] hModule  包含函数或变量的 DLL 模块的句柄。 [LoadLibrary](https://learn.microsoft.com/zh-cn/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya)、[LoadLibraryEx](https://learn.microsoft.com/zh-cn/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryexa)、[LoadPackagedLibrary](https://learn.microsoft.com/zh-cn/windows/win32/api/winbase/nf-winbase-loadpackagedlibrary) 或 [GetModuleHandle](https://learn.microsoft.com/zh-cn/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea) 函数返回此句柄。  **GetProcAddress** 函数不会从使用 **LOAD\_LIBRARY\_AS\_DATAFILE** 标志加载的模块中检索地址。 有关详细信息，请参阅 [LoadLibraryEx](https://learn.microsoft.com/zh-cn/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryexa)。  [in] lpProcName  函数或变量名称，或函数的序号值。 如果此参数是序号值，则它必须在低序位字中；高序位字必须为零。 返回值 如果函数成功，则返回值是导出的函数或变量的地址。  如果函数失败，则返回值为 NULL。 要获得更多的错误信息，请调用 GetLastError。 注解 lpProcName 指向的函数名称的拼写和大小写必须与源 DLL 的模块定义 (.def) 文件的 **EXPORTS** 语句中的拼写和大小写相同。 函数的导出名称可能与在代码中调用这些函数时使用的名称不同。 SDK 头文件中使用的宏会隐藏此差异。 有关详细信息，请参阅 [函数原型的约定](https://learn.microsoft.com/zh-cn/windows/win32/Intl/conventions-for-function-prototypes)。  lpProcName 参数可以通过在 **EXPORTS** 语句中指定与函数关联的序号值来标识 DLL 函数。 **GetProcAddress** 验证指定的序号是否在 .def 文件中导出的最高序号值范围内。 然后，函数使用序号作为索引，从函数表中读取函数的地址。  如果 .def 文件不连续对函数进行编号，从 1 到 N (其中 N 是) 导出的函数数，则可能会出现错误，即 **GetProcAddress** 返回无效的非 NULL 地址，即使没有具有指定序号的函数也是如此。  如果 DLL 模块中不存在函数（例如，如果函数仅在 Windows Vista 上可用，但应用程序可能在 Windows XP 上运行），则按名称（而不是序号值）指定函数，并将应用程序设计为处理函数不可用时的情况，如以下代码片段所示。  C++复制  typedef void (WINAPI \*PGNSI)(LPSYSTEM\_INFO);  // Call GetNativeSystemInfo if supported or GetSystemInfo otherwise.  PGNSI pGNSI;  SYSTEM\_INFO si;  ZeroMemory(&si, sizeof(SYSTEM\_INFO));    pGNSI = (PGNSI) GetProcAddress(  GetModuleHandle(TEXT("kernel32.dll")),  "GetNativeSystemInfo");  if(NULL != pGNSI)  {  pGNSI(&si);  }  else  {  GetSystemInfo(&si);  }  有关包含此代码片段的完整示例，请参阅 [获取系统版本](https://learn.microsoft.com/zh-cn/windows/win32/SysInfo/getting-the-system-version)。 示例 有关示例，请参阅 [使用 Run-Time 动态链接](https://learn.microsoft.com/zh-cn/windows/win32/Dlls/using-run-time-dynamic-linking)。   |  | | --- | | // A simple program that uses LoadLibrary and  // GetProcAddress to access myPuts from Myputs.dll.    #include <windows.h>  #include <stdio.h>    typedef int (\*MYPROC)(LPCWSTR);    int main( void )  {  HINSTANCE hinstLib;  MYPROC ProcAdd;  BOOL fFreeResult, fRunTimeLinkSuccess = FALSE;    // Get a handle to the DLL module.    hinstLib = LoadLibrary(TEXT("MyPuts.dll"));    // If the handle is valid, try to get the function address.    if (hinstLib != NULL)  {  ProcAdd = (MYPROC) GetProcAddress(hinstLib, "myPuts");    // If the function address is valid, call the function.    if (NULL != ProcAdd)  {  fRunTimeLinkSuccess = TRUE;  ProcAdd(L"Message sent to the DLL function\n");  }  // Free the DLL module.    fFreeResult = FreeLibrary(hinstLib);  }  // If unable to call the DLL function, use an alternative.  if (! fRunTimeLinkSuccess)  printf("Message printed from executable\n");  return 0;  } |  要求  |  |  | | --- | --- | | **标头** | libloaderapi.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [动态链接库函数](https://learn.microsoft.com/zh-cn/windows/win32/Dlls/dynamic-link-library-functions)  [FreeLibrary](https://learn.microsoft.com/zh-cn/windows/win32/api/libloaderapi/nf-libloaderapi-freelibrary)  [GetModuleHandle](https://learn.microsoft.com/zh-cn/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea)  [LoadLibrary](https://learn.microsoft.com/zh-cn/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya)  [LoadLibraryEx](https://learn.microsoft.com/zh-cn/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryexa)  [LoadPackagedLibrary](https://learn.microsoft.com/zh-cn/windows/win32/api/winbase/nf-winbase-loadpackagedlibrary)  [运行时动态链接](https://learn.microsoft.com/zh-cn/windows/win32/Dlls/run-time-dynamic-linking)  [VBS enclave 中可用的 Vertdll API](https://learn.microsoft.com/zh-cn/windows/win32/trusted-execution/enclaves-available-in-vertdll) |

## FreeLibrary函数的用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 释放加载的动态链接库 (DLL) 模块，并在必要时递减其引用计数。 当引用计数达到零时，模块将从调用进程的地址空间中卸载，句柄不再有效。 语法 C++  BOOL FreeLibrary(  [in] HMODULE hLibModule  ); parameters [in] hLibModule  已加载的库模块的句柄。 [LoadLibrary](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibrarya)、[LoadLibraryEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibraryexa)、 [GetModuleHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-getmodulehandlea) 或 [GetModuleHandleEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-getmodulehandleexa) 函数返回此句柄。 返回值 如果该函数成功，则返回值为非零值。  如果函数失败，则返回值为零。 若要获得更多的错误信息，请调用 GetLastError 函数。 注解 系统为每个加载的模块维护每个进程引用计数。 由于加载时动态链接，在进程初始化时加载的模块的引用计数为 1。 每次调用 [LoadLibrary](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibrarya) 加载模块时，模块的引用计数都会递增。 引用计数也会通过对 [LoadLibraryEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibraryexa) 的调用递增，除非模块是首次加载并且作为数据或图像文件加载的。  每次为模块调用 **FreeLibrary** 或 [FreeLibraryAndExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-freelibraryandexitthread) 函数时，引用计数都会递减。 当模块的引用计数达到零或进程终止时，系统会从进程的地址空间中卸载模块。 在卸载库模块之前，系统允许模块通过调用模块的 [DllMain](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/dllmain) 函数（如果有），并使用 DLL\_PROCESS\_DETACH 值从进程分离。 这样做使库模块有机会清理代表当前进程分配的资源。 在入口点函数返回之后，将从当前进程的地址空间移除库模板。  从 [DllMain](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/dllmain) 调用 **FreeLibrary** 是不安全的。 有关详细信息，请参阅 [DllMain](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/dllmain) 中的“备注”部分。  调用 **FreeLibrary** 不会影响使用同一模块的其他进程。  使用 [GetModuleHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-getmodulehandlea) 返回的句柄调用 **FreeLibrary** 时要小心。 **GetModuleHandle** 函数不会递增模块的引用计数，因此将此句柄传递给 **FreeLibrary** 可能会导致模块过早卸载。  必须卸载它正在执行的 DLL 然后终止自己的线程应调用 [FreeLibraryAndExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-freelibraryandexitthread) ，而不是单独调用 **FreeLibrary** 和 **ExitThread** 。 否则，可能会出现争用条件。 有关详细信息，请参阅 [FreeLibraryAndExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-freelibraryandexitthread) 的“备注”部分。 示例 有关示例，请参阅 [使用 Run-Time 动态链接](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/using-run-time-dynamic-linking)。 要求  |  |  | | --- | --- | | **标头** | libloaderapi.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [DllMain](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/dllmain)  [动态链接库函数](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/dynamic-link-library-functions)  [FreeLibraryAndExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-freelibraryandexitthread)  [GetModuleHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-getmodulehandlea)  [GetModuleHandleEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-getmodulehandleexa)  [LoadLibrary](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibrarya)  [运行时动态链接](https://learn.microsoft.com/zh-cn/windows/desktop/Dlls/run-time-dynamic-linking) |

# 演练

## 1.新建一个文件夹取名Lesson49-dll2，然后把上一节的两个项目复制粘贴到里面，然后我们用vs打开他们

|  |
| --- |
|  |
|  |

### 需要注意的是：DllMain函数既是入口函数，也是出口函数，加载dll的时候会调用它，卸载dll的时候 也会调用它。

## 2. DllMain函数的第二个参数可以指明是在入口的时候调用还是出口的时候调用。所以在DllMain函数里面一般都需要根据这个参数的值做switch..case语句，可以这么写，有四个步骤，1.创建进程的时候的dll附加到进程里,每一个新进程只初始化一次，他是作为一个独立的线程来执行，2. 创建线程3. 线程退出4. 释放dll，1和2是入口，3和4是出口顺序1-2-3-4，一般入口时候需要做一些初始化工作，退出的时候做一下清理工作。由于我们的程序非常简单，不需要上面初始化工作，我们就利用MessageBox来测试一下

|  |
| --- |
|  |

## 3.然后我们给我们的drawutil项目再添加2个函数，先在头文件里面声明并且做宏定义

|  |
| --- |
|  |

## 4.然后我们就来写具体实现，只是调用一下消息框函数作为测试使用

|  |
| --- |
|  |

## 5.先执行清理项目，然后再构建项目,成功后如图

|  |
| --- |
|  |

## 6、然后把test-drawutil下面的源代码文件夹里面的所有drawutil文件删除，注意：是Lesson49文件夹里面的，把我们Lesson49drawutil项目生成的dll文件粘贴进来，注意此时我们不要头文件，要不要lib文件

|  |
| --- |
|  |

## 7.把test-drawutil项目删除，新建一个叫做DynamicLoadDrawutil的MFC项目，应用程序类型选择基于对话框的应用程序，点击完成

|  |
| --- |
|  |

## 8.然后把drawutil.dll拷贝粘贴到源码所在的文件夹里面

|  |
| --- |
|  |

## 9.为了防止用户按下回车应用程序马上结束，我们保留确定按钮，改名退出，然后在他的点击事件处理里面添加退出确认代码

|  |
| --- |
|  |

### 注意：需要使用win32的代码，因为这样子才可以有按钮可以选择

## 10.给对话框添加2个按钮一个用来动态加载dll，ID：IDC\_BTN\_LOAD，另外一个调用dll里面的函数,ID: IDC\_BTN\_CALL，然后文件点击加载按钮进入事件处理函数,点击第一个按钮加载dll，点击第二个按钮调用指定的函数。注意：这里需要两个全局变量一个是用来保存加载后的dll句柄，另外一个是函数指针用来接收被调用的dll里面的函数

|  |
| --- |
|  |

## 11.给对话框添加一个按钮，用来释放dll，其实我们需要这么做，先把调用函数的按钮禁用，当dll文件被成功加载了，就使得调用按钮可用，当dll文件被卸载了，就要让调用按钮不可以，这样子可以避免一下错误

|  |
| --- |
|  |

# 小结：

## 动态加载dll文件需要5个步骤

## 1.将dll文件复制到调用代码文件夹下面

## 2.调用LoadLibrary函数把dll加载近进程空间

## 3.调用GetProcAddess在dll里面获取需要调用的函数地址保存到一个函数指针里面

## 4.调用这个函数

## 5.调用完毕后需要释放dll

# 这一节的学习到此为止，完整代码如下

## Lesson49-dll2/drawutil/drawutil/drawutil.h

|  |
| --- |
| #pragma once  #ifdef \_\_cplusplus  #define EXPORT extern "C" \_\_declspec(dllexport) //如果是c++就定义这个  #else  #define EXPORT \_\_declspec(dllexport)  #endif  EXPORT BOOL CALLBACK DwuCenterTextA(HDC hdc,PRECT prc,PCSTR pstr);  EXPORT BOOL CALLBACK DwuCenterTextW(HDC hdc,PRECT prc,PCWSTR pstr);  //new functions  EXPORT DWORD FuncA();  EXPORT DWORD FuncW();  //根据程序是否使用UNICODE来决定调用哪一个版本的函数，也就是定义一个宏这个宏既不需要A也不需要W  #ifdef UNICODE  #define DwuCenterText DwuCenterTextW  #define Func FuncW  #else  #define DwuCenterText DwuCenterTextA  #define Func FuncA  #endif |

## Lesson49-dll2/drawutil/drawutil/drawutil.c

|  |
| --- |
| #include<Windows.h>  #include"drawutil.h"  int WINAPI DllMain(HINSTANCE hinstDLL, // handle to DLL module  DWORD fdwReason, // reason for calling function  LPVOID lpvReserved )  {  switch(fdwReason) //四个步骤，1和2是入口，3和4是出口顺序1-2-3-4，一般入口时候需要做一些初始化工作，退出的时候做一下清理工作  {  case DLL\_PROCESS\_ATTACH: //1.创建进程的时候的dll附加到进程里,每一个新进程只初始化一次，他是作为一个独立的线程来执行  MessageBox(NULL,TEXT("Dll Loaded..."),TEXT("info"),0);  break;  case DLL\_THREAD\_ATTACH: //2.创建线程  //MessageBox(NULL,TEXT("THREAD ATTACH"),TEXT("info"),0);  break;  case DLL\_THREAD\_DETACH: //3.线程退出  //MessageBox(NULL,TEXT("THREAD ENDING"),TEXT("info"),0);  break;  case DLL\_PROCESS\_DETACH: //4.释放dll  //MessageBox(NULL,TEXT("Release Dll"),TEXT("info"),0);  if (lpvReserved != NULL)  {  break; // do not do cleanup if process termination scenario  }    break;    }  return TRUE;  }  //通用调用方法，注意EXPORT是在头文件声明的  EXPORT BOOL CALLBACK DwuCenterTextA(HDC hdc,PRECT prc,PCSTR pstr)  {  int length,centerx,centery;  SIZE sz;  length = lstrlenA(pstr);//获取字符串里面的字符有多少个  GetTextExtentPoint32A(hdc,pstr,length,&sz);//GetTextExtentPoint32 函数计算指定文本字符串的宽度和高度。  centerx = (prc->right - prc->left-sz.cx)/2;  centery = (prc->bottom - prc->top-sz.cy)/2;  return TextOutA(hdc,centerx,centery,pstr,length);  }  EXPORT BOOL CALLBACK DwuCenterTextW(HDC hdc,PRECT prc,PCWSTR pstr)  {  int length,centerx,centery;  SIZE sz;  length = lstrlenW(pstr);//获取字符串里面的字符有多少个  GetTextExtentPoint32W(hdc,pstr,length,&sz);//GetTextExtentPoint32 函数计算指定文本字符串的宽度和高度。  centerx = (prc->right - prc->left-sz.cx)/2;  centery = (prc->bottom - prc->top-sz.cy)/2;  return TextOutW(hdc,centerx,centery,pstr,length);  }  //new functions,这里不是给窗口用的，不需要CALLBACK关键字  EXPORT DWORD FuncA()  {  MessageBox(NULL,TEXT("FuncA Called..."),TEXT("info"),0);  return TRUE;  }  EXPORT DWORD FuncW()  {  MessageBox(NULL,TEXT("FuncW Called..."),TEXT("info"),0);  return TRUE;  } |

## Lesson49-dll2\DynamicLoadDrawutil\DynamicLoadDrawutil/ DynamicLoadDrawutilDlg.cpp

|  |
| --- |
| // DynamicLoadDrawutilDlg.cpp : 实现文件  //  #include "stdafx.h"  #include "DynamicLoadDrawutil.h"  #include "DynamicLoadDrawutilDlg.h"  #include "afxdialogex.h"  #ifdef \_DEBUG  #define new DEBUG\_NEW  #endif  // 用于应用程序“关于”菜单项的 CAboutDlg 对话框  class CAboutDlg : public CDialogEx  {  public:  CAboutDlg();  // 对话框数据  enum { IDD = IDD\_ABOUTBOX };  protected:  virtual void DoDataExchange(CDataExchange\* pDX); // DDX/DDV 支持  // 实现  protected:  DECLARE\_MESSAGE\_MAP()  };  CAboutDlg::CAboutDlg() : CDialogEx(CAboutDlg::IDD)  {  }  void CAboutDlg::DoDataExchange(CDataExchange\* pDX)  {  CDialogEx::DoDataExchange(pDX);  }  BEGIN\_MESSAGE\_MAP(CAboutDlg, CDialogEx)  END\_MESSAGE\_MAP()  // CDynamicLoadDrawutilDlg 对话框  CDynamicLoadDrawutilDlg::CDynamicLoadDrawutilDlg(CWnd\* pParent /\*=NULL\*/)  : CDialogEx(CDynamicLoadDrawutilDlg::IDD, pParent)  {  m\_hIcon = AfxGetApp()->LoadIcon(IDR\_MAINFRAME);  }  void CDynamicLoadDrawutilDlg::DoDataExchange(CDataExchange\* pDX)  {  CDialogEx::DoDataExchange(pDX);  }  BEGIN\_MESSAGE\_MAP(CDynamicLoadDrawutilDlg, CDialogEx)  ON\_WM\_SYSCOMMAND()  ON\_WM\_PAINT()  ON\_WM\_QUERYDRAGICON()  ON\_BN\_CLICKED(IDOK, &CDynamicLoadDrawutilDlg::OnBnClickedOk)  ON\_BN\_CLICKED(IDC\_BTN\_LOAD, &CDynamicLoadDrawutilDlg::OnBnClickedBtnLoad)  ON\_BN\_CLICKED(IDC\_BTN\_CALL, &CDynamicLoadDrawutilDlg::OnBnClickedBtnCall)  ON\_BN\_CLICKED(IDC\_BTN\_FREE, &CDynamicLoadDrawutilDlg::OnBnClickedBtnFree)  END\_MESSAGE\_MAP()  // CDynamicLoadDrawutilDlg 消息处理程序  BOOL CDynamicLoadDrawutilDlg::OnInitDialog()  {  CDialogEx::OnInitDialog();  // 将“关于...”菜单项添加到系统菜单中。  // IDM\_ABOUTBOX 必须在系统命令范围内。  ASSERT((IDM\_ABOUTBOX & 0xFFF0) == IDM\_ABOUTBOX);  ASSERT(IDM\_ABOUTBOX < 0xF000);  CMenu\* pSysMenu = GetSystemMenu(FALSE);  if (pSysMenu != NULL)  {  BOOL bNameValid;  CString strAboutMenu;  bNameValid = strAboutMenu.LoadString(IDS\_ABOUTBOX);  ASSERT(bNameValid);  if (!strAboutMenu.IsEmpty())  {  pSysMenu->AppendMenu(MF\_SEPARATOR);  pSysMenu->AppendMenu(MF\_STRING, IDM\_ABOUTBOX, strAboutMenu);  }  }  // 设置此对话框的图标。当应用程序主窗口不是对话框时，框架将自动  // 执行此操作  SetIcon(m\_hIcon, TRUE); // 设置大图标  SetIcon(m\_hIcon, FALSE); // 设置小图标  // TODO: 在此添加额外的初始化代码  return TRUE; // 除非将焦点设置到控件，否则返回 TRUE  }  void CDynamicLoadDrawutilDlg::OnSysCommand(UINT nID, LPARAM lParam)  {  if ((nID & 0xFFF0) == IDM\_ABOUTBOX)  {  CAboutDlg dlgAbout;  dlgAbout.DoModal();  }  else  {  CDialogEx::OnSysCommand(nID, lParam);  }  }  // 如果向对话框添加最小化按钮，则需要下面的代码  // 来绘制该图标。对于使用文档/视图模型的 MFC 应用程序，  // 这将由框架自动完成。  void CDynamicLoadDrawutilDlg::OnPaint()  {  if (IsIconic())  {  CPaintDC dc(this); // 用于绘制的设备上下文  SendMessage(WM\_ICONERASEBKGND, reinterpret\_cast<WPARAM>(dc.GetSafeHdc()), 0);  // 使图标在工作区矩形中居中  int cxIcon = GetSystemMetrics(SM\_CXICON);  int cyIcon = GetSystemMetrics(SM\_CYICON);  CRect rect;  GetClientRect(&rect);  int x = (rect.Width() - cxIcon + 1) / 2;  int y = (rect.Height() - cyIcon + 1) / 2;  // 绘制图标  dc.DrawIcon(x, y, m\_hIcon);  }  else  {  CDialogEx::OnPaint();  }  }  //当用户拖动最小化窗口时系统调用此函数取得光标  //显示。  HCURSOR CDynamicLoadDrawutilDlg::OnQueryDragIcon()  {  return static\_cast<HCURSOR>(m\_hIcon);  }  void CDynamicLoadDrawutilDlg::OnBnClickedOk()  {  // TODO: 在此添加控件通知处理程序代码  if(IDOK==::MessageBox(this->m\_hWnd,\_T("退出程序？"),\_T("退出确认"),MB\_OKCANCEL))  {  CDialogEx::OnOK();  }  }  HINSTANCE hinstDll;//全局变量  typedef DWORD (\*MyFunc)();//函数指针用来接收GetProcAddress的返回值  void CDynamicLoadDrawutilDlg::OnBnClickedBtnLoad()  {  // TODO: 在此添加控件通知处理程序代码  //动态加载drawutil.dl  hinstDll = LoadLibrary(\_T("drawutil.dll"));  GetDlgItem(IDC\_BTN\_CALL)->EnableWindow(TRUE);//加载dll后让调用按钮可用  }  void CDynamicLoadDrawutilDlg::OnBnClickedBtnCall()  {  // TODO: 在此添加控件通知处理程序代码  MyFunc Func;  Func = (MyFunc)GetProcAddress(hinstDll,("FuncW"));//这里不能用宏只能用A版函数或者W版  Func();  }  void CDynamicLoadDrawutilDlg::OnBnClickedBtnFree()  {  // TODO: 在此添加控件通知处理程序代码  FreeLibrary(hinstDll);  MessageBox(\_T("Dll被卸载了"));  GetDlgItem(IDC\_BTN\_CALL)->EnableWindow(FALSE);//卸载dll后禁用调用按钮  } |