zhiweiyouzhishenghuo的专栏

[转][转]为什么在DllMain里不能调用LoadLibrary和FreeLibrary函数？

2012-5-16 阅读3043 评论0

为什么在DllMain里不能调用LoadLibrary和FreeLibrary函数？

MSDN里对这个问题的答案十分的晦涩。不过现在我们已经有了足够的知识来解答这个问题。
考虑下面的情况：

(a)DllB静态链接DllA
(b)DllB在DllMain里调用DllA的一个函数A1()
(c)DllA在DllMain里调用LoadLibrary("DllB.dll")

分析：当执行到DllA中的DllMain的时侯，DllA.dll已经被映射到进程地址空间中，已经加入到了module list中。当它调用LoadLibrary("DllB.dll")时，首先会调用LdrpMapDll把DllB.dll映射到进程地址空间，并加入到InLoadOrderModuleList中。然后会调用LdrpLoadImportModule(...)加载它引用的DllA.dll，而 LdrpLoadImportModule会调用LdrpCheckForLoadedDll检查是否DllA.dll已经被加载。LdrpCheckForLoadedDll会在哈希表LdrpHashTable中查找DllA.dll，而显然它能找到，所以加载DllA.dll这一步被成功调过。DllA在它的DllMain函数里能成功加载DllB，并要执行DllB的DllMain函数对其初始化。站在DllB的角度考虑，当程序运行到它的DllMain的时侯，它完全有理由相信它隐式链接的DllA.dll已经被加载并且成功地初始化。可事实上，此时DllA只是处在"正在初始化"的过程中!这种理想和现实的差距就是可能产生的Bug的根源，就是禁止在DllMain里调用LoadLibrary的理由!

本文附带的例子中说明了这种出错的情况：

```
TestLoad主程序:
int main(int argc, char* argv[])
{
    HINSTANCE hDll = ::LoadLibrary( "DllA.dll" ) ;
    FreeLibrary( hDll ) ;
return 0;
}

DllA:
HANDLE g_hDllB = NULL ;
char *g_buf = NULL ;

BOOL APIENTRY DllMain( HANDLE hModule,
```

```
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved
                    )
{
switch (ul_reason_for_call)
    {
case DLL_PROCESS_ATTACH:
            OutputDebugString( "==>DllA: Initialize begin!\n" ) ;

            g_hDllB = LoadLibrary( "DllB.dll" ) ;

// g_buf在Load DllB.dll之后才初始化，显然它没有料到DllB在初始化时居然会用到g_buf!!
            g_buf = newchar[128] ;
            memset( g_buf, 0, 128 ) ;

            OutputDebugString( "==>DllA: Initialize end!\n" ) ;
break ;

case DLL_THREAD_ATTACH:
case DLL_THREAD_DETACH:
case DLL_PROCESS_DETACH:
break;
    }
returnTRUE;
}

DLLA_API void A1( char *str )
{
    OutputDebugString( "==>DllA: A1()\n" ) ;

// 当DllB.dll在它的DllMain函数里调用A1()时，g_buf还没有初始化，所以必然会出错!
    strcat( g_buf, "==>DllA: " ) ;
    strcpy( g_buf, str ) ;

    OutputDebugString( g_buf ) ;
}

DllB:
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
```

```
                      LPVOID lpReserved
                    )
{
switch (ul_reason_for_call)
    {
case DLL_PROCESS_ATTACH:
            OutputDebugString( "==>DllB: Initialize!\n" ) ;
            OutputDebugString( "==>DllB: DllB depend on DllA.\n" ) ;
            OutputDebugString( "==>DllB: I think DllA has been initialize.\n" ) ;

// 当程序运行到这时，DllB认为它引用的DllA.dll已经加载并初始化了，所以它调用DllA的函数A1()
            A1( "DllB Invoke DllA::A1()\n" ) ;
break ;

case DLL_THREAD_ATTACH:
case DLL_THREAD_DETACH:
case DLL_PROCESS_DETACH:
break;
    }
returnTRUE;
}
```

在调用DllA的函数A1()时，因为DllA里有些变量还没初始化，所以会产生exception。以下是截取的部分LDR的输出，"==>"开头的是程序的输出。

```
LDR: Loading (DYNAMIC) H:\cm\vc6\TestLoad\bin\DllA.dll
    LDR: KERNEL32.dll used by DllA.dll
    LDR: Snapping imports for DllA.dll from KERNEL32.dll
    LDR: Real INIT LIST
         H:\cm\vc6\TestLoad\bin\DllA.dll init routine 10001440
    LDR: DllA.dll loaded. - Calling init routine at 10001440
    ==>DllA: Initialize begin!
    LDR: Loading (DYNAMIC) H:\cm\vc6\TestLoad\bin\DllB.dll
    LDR: DllA.dll used by DllB.dll
    LDR: Snapping imports for DllB.dll from DllA.dll
    LDR: Refcount   DllA.dll (2)
    LDR: Real INIT LIST
         H:\cm\vc6\TestLoad\bin\DllB.dll init routine 371260
    LDR: DllB.dll loaded. - Calling init routine at 371260
    ==>DllB: Initialize!
    ==>DllB: DllB depend on DllA.
```

```
    ==>DllB: I think DllA has been initialize.
    ==>DllA: A1()
    First-chance exception in Test.exe (DLLA.DLL): 0xC0000005: Access Violation.
    ==>DllA: Initialize end!
```

在前面已经说过LdrUnloadDll里对DllMain里调用FreeLibrary的情况进行了特殊处理。此时仍然会对各个相关的Dll引用计数减 1，并移入到unload list中，但然后LdrUnloadDll就返回了!并没有执行Dll的termination code。我构建了一个运行正确的例子TestUnload，说明LdrUnloadDll是怎么处理的。

考虑下面的情况：

(a)DllA依赖于DllC，DllB也依赖于DllC
(b)DllA里调用LoadLibrary("DllB.dll")，并保证其成功
(c)DllA在DllMain的termination code里执行FreeLibrary()，释放DllB
(d)在主程序里动态的加载DllA

下面的代码和注释说明了程序运行的细节：

TestUnload主程序：

```
int main(int argc, char* argv[])
    {
        HINSTANCE hDll = ::LoadLibrary( "DllA.dll" ) ;
// 在调用LoadLibrary之后
// LoadOrderList:   A(1) --> C(2) --> B(1), 括号内的代表LoadCount
// MemoryOrderList: A(1) --> C(2) --> B(1)
// InitOrderList:   C(2) --> A(1) --> B(1)

        FreeLibrary( hDll ) ;
return 0;
    }
```

DllA:

```
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
    {
switch (ul_reason_for_call)
        {
case DLL_PROCESS_ATTACH:
                OutputDebugString( "==>DllA: Initialize!\n" ) ;
```

```
// 这里用LoadLibrary是安全的
                g_hDllB = LoadLibrary( "DllB.dll" ) ;
if (NULL == g_hDllB)
returnFALSE ;
break ;

case DLL_THREAD_ATTACH:
case DLL_THREAD_DETACH:
break ;

case DLL_PROCESS_DETACH:
// 运行到这里时，DllA现在只留在LoadOrderList中，已经从另两个list中删除
// LoadOrderList:   A(0) --> C(1) --> B(1)
// MemoryOrderList: C(1) --> B(1)
// InitOrderList:   C(1) --> B(1)

                OutputDebugString( "==>DllA: Uninitialize begin!\n" ) ;

                FreeLibrary( g_hDllB ) ;

// 运行到这里时，DllB和DllC都从MemoryOrderList和InitOrderList中删除了
// LoadOrderList:   A(0) --> C(0) --> B(0)
// MemoryOrderList:
// InitOrderList:

                OutputDebugString( "==>DllA: Uninitialize end!\n" ) ;
break;
        }
returnTRUE;
    }
```

如果主程序是静态链接DllA又如何呢？LdrUnloadDll同样能判断这种情况：如果进程正在关闭那么LdrUnloadDll直接返回。我也构建了一个运行正确的例子TestUnload2来说明这种情况：

```
TestUnload2主程序：
int main(int argc, char* argv[])
    {
// 此时DllA,DllB,DllC均已load
// LoadOrderList:   A(-1) --> C(-1) --> B(1), 括号内的代表LoadCount
// MemoryOrderList: A(-1) --> C(-1) --> B(1)
// InitOrderList:   C(-1) --> A(-1) --> B(1)
```

```
return 0;
    }

DllA:
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
    {
switch (ul_reason_for_call)
        {
case DLL_PROCESS_ATTACH:
                OutputDebugString( "==>DllA: Initialize!\n" ) ;

// 这里用LoadLibrary是安全的
                g_hDllB = LoadLibrary( "DllB.dll" ) ;
if (NULL == g_hDllB)
returnFALSE ;

break ;

case DLL_THREAD_ATTACH:
case DLL_THREAD_DETACH:
break ;

case DLL_PROCESS_DETACH:
// 运行到这里时，DllB已经被卸载，因为它是InitOrderList中最后一项
// 这里的卸载指的是调用了Init routine，发出了DLL_PROCESS_DETACH通知，而不是指unmap内存中的映像
                OutputDebugString( "==>DllA: Uninitialize begin!\n" ) ;

// 这里不应该再调用DllB的函数!!!

// 尽管DllB已经被卸载，但这里调用FreeLibrary并无危险
// 因为LdrUnloadDll判断出进程正在Shutdown，所以它什么也没做，直接返回
                FreeLibrary( g_hDllB ) ;

                OutputDebugString( "==>DllA: Uninitialize end!\n" ) ;

break;
```

```
        }
returnTRUE;
    }
```

在Jeffrey Richter的"Windows核心编程"和Matt Pietrek在1999年MSJ上的"Under theHood"里都说到，User32.dll在它的initializecode里会用LoadLibrary加载"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit_DLLs" 下的dll，在它的terminate code里会用FreeLibrary卸载它们。跟踪它的FreeLibrary函数，发现同上面的例子一样，LdrUnloadDll发现进程正在 Shutdown中，就直接返回了，没有任何危险。(User32.dll是静态链接的函数，只可能在进程关闭时被卸载。另外，在我调试的时侯，发现即使AppInit_DLLs下为空，User32.dll仍然会加载imm32.dll)。

总而言之，FreeLibrary本身是相当安全的，但MSDN里对它的警告也并非是胡说八道。在DllMain里使用FreeLibrary仍然是具有危险性的，与LoadLibrary一样，它们具有相同的Bug哲学，即理想和现实的差距!
TestUnload2虽然运行正确，但是它具有潜在的危险性。
对DllA而言，释放DllB是它的责任，是它在收到DLL_PROCESS_DETACH通知之后用FreeLibrary卸载的，可事实上如果DllA被主程序静态链接，或者DllA是动态链接但没有用FreeLibrary显式卸载它的话，那么在进程结束时，在DllA卸载DllB之前，DllB就已经被主程序卸载掉了!这种认识上的错误就是养育Bug的沃土。如果DllA没有认识到这种可能性，而在FreeLibrary之前调用DllB的函数，就极可能出错!!!

为了加深理解，我用文章开头提到的那个Bug来说明这种情况，那可是血的教训。问题描述如下：
我用MFC写了一个OCX，OCX里动态加载了一些Plugin Dlls，在OCX的ExitInstance(相当于DllMain里处理DLL_PROCESS_DETACH通知)里调用这些Plugin的 Uninitialize code，然后用FreeLibrary将其释放。在我用MFC编写的一个Doc/View架构的测试程序里运行良好，但不久客户就报告了一个Bug：用 VB写了一个OCX2来包装我的OCX，在一个网页里使用OCX2，然后在IE里打开这个网页，在关掉IE时会当掉!发生在特定条件下的奇怪的错误!当时我可是费了不少功夫来解这个Bug，现在一切都那么清晰了。

下面是我用MFC写的测试程序在关闭时的堆栈：

```
PDFREA_1!CPDFReaderOCXApp::ExitInstance+0x1d
PDFREA_1!DllMain+0x1bb
PDFREA_1!_DllMainCRTStartup+0x80
ntdll!LdrpCallInitRoutine+0x14
ntdll!LdrUnloadDll+0x29a
KERNEL32!FreeLibrary+0x3b
ole32!CClassCache::CDllPathEntry::CFinishObject::Finish+0x2b
ole32!CClassCache::CFinishComposite::Finish+0x19
ole32!CClassCache::FreeUnused+0x192
ole32!CoFreeUnusedLibraries+0x35
MFCO42D!AfxOleTerm+0x7b
MFCO42D!AfxOleTermOrFreeLib+0x12
MFC42D!AfxWinTerm+0xa9
MFC42D!AfxWinMain+0x103
ReaderContainerMFC!WinMain+0x18
ReaderContainerMFC!WinMainCRTStartup+0x1b3
KERNEL32!BaseProcessStart+0x3d
```

可以看到OCX被FreeLibrary显式地释放，抢在Plugin被进程释放之前，所以不会出错。

下面是关闭IE时的堆栈：

```
CPDFReaderOCXApp::ExitInstance() line 44
DllMain(HINSTANCE__ * 0x04e10000, unsigned long 0, void * 0x00000001) line 139
_DllMainCRTStartup(void * 0x04e10000, unsigned long 0, void * 0x00000001) line 273 + 17 bytes
NTDLL! LdrShutdownProcess + 238 bytes
KERNEL32! ExitProcess + 85 bytes
```

可以看到OCX是在LdrShutdownProcess里被释放的，而此时Plugin已经被释放掉了，因为在 InInitializationOrderModuleList表里Plugin Dlls在OCX之后，所以它们被先释放!这种情况要是还不出错真是奇迹了。

总结：虽然MS警告不要在DllMain里不能调用LoadLibrary和FreeLibrary函数，可实际上它还是做了很多的工作来处理这种情况。只不过因为他不想或者懒得说清楚到底哪些情况不能这么用，才干脆一棒子打死统统不许。在你自己的程序里不是绝对不能这么用，只是你必须清楚地知道每件事是怎么发生的，以及潜在的危险。

DllMain函数中不能Load（Unload）别的dll；
DllMain函数中不能调用其它dll暴露的函数！（System32.dll、User32.dll、Advapi32.dll除外）
Dll中声明的全局（或静态）变量的构造和析构函数中同样不能执行以上的操作！因为这些函数甚至在DllMain执行之前就已经执行了！
请各位务必牢记这些原则，不要再犯这样的错误！因为这种错误追查起来非常非常麻烦，因为它的表现受环境影响，缺乏一致性。

上一篇 下一篇