Windows 加载器与模块初始化

作者：Matt Pietrek

在最近的 MSJ 专栏（1999 年六月）中，我讨论了 COM 类型库和数据库访问层，例如 ActiveX® 数据对象（ADO）和 OLE DB。MSJ 专栏的长期读者可能认为我已经不行了（写不出技术层次比较高的文章了）。为了重振雄风，这个月我要讲解一部分 Windows NT®加载器代码，它是操作系统和你的代码接合的地方。同时，我也会向你演示一些获取加载器状态信息的高超技巧，以及可以用在 Developer Studio®调试器中的相关技巧。

考虑一下你对 EXE、DLL 以及它们是如何被加载的和初始化的到底知道多少。你可能知道当一个用 C++写成的 DLL 被加载时，它的 DllMain 函数会被调用。想一想当你的 EXE 隐含链接到一些 DLL（例如，KERNEL32.DLL 和 USER32.DLL）时到底发生了什么。这些 DLL 是以什么顺序被初始化的？某个 DLL 将要被初始化，而它所依赖的其它 DLL 还未被初始化，这可能吗？Platform SDK 在“Dynamic Link Library Entry Point Function（动态链接库入口点函数）”一节中对此描述如下：

“你的函数应该仅进行一些简单的初始化任务，例如设置线程局部存储（TLS），创建同步对象和打开文件等。它绝对不能调用 LoadLibrary 函数，因为这可能在 DLL 加载顺序上造成循环依赖。这可能导致即将使用一个 DLL 但是系统还未对它进行初始化。同样，你也不能在入口点函数中调用 FreeLibrary 函数，因为这可能导致即将使用一个 DLL 但是系统已经执行完了它的终止代码。”

“调用除 TLS 函数、同步函数和文件函数之外的 Win32®函数也可能引起很难诊断的问题。例如，调用 User 函数、Shell 函数和 COM 函数可能引起访问违规，因为这些 DLL 中一些函数调用 LoadLibrary 加载其它系统组件。”

看了上述文档后我的第一反应是它太含糊了。例如你想在自己的 DllMain 函数中读取注册表是再正常不过的事了，它当然可以作为初始化的一部分。但不幸的是，在你的 DllMain 代码开始执行时 ADVAPI32.DLL 还没有初始化。这样，对注册表 API 的调用将会失败。

在上述文档中对使用 LoadLibrary 给出了严厉的警告。但非常有趣的是，Windows NT 的 USER32.DLL 却明确地忽略前面的忠告。你可能知道 Widnows NT 上的一个注册表键 AppInit_Dlls，它用来加载一系列 DLL 到每个进程。事实表明，是 USER32 在初始化时加载这些 DLL 的。USER32 在它的 DllMain 代码中查看这个注册表键并调用 LoadLibrary 加载这些 DLL。稍微思考一下就会知道，如果你的应用程序不使用 USER32.DLL 的话，AppInit_Dlls 这个技巧就不能发挥作用。不过，这有点跑题了。

我之所以要讲解这方面的内容是因为 DLL 的加载与初始化还是一片盲区。在大多数情况下，对操作系统加载器是如何工作的有一个简单的印象就足够了。然而，在极少数情况下，除非你对操作系统加载器的行为方式有比较详细的了解，否则就会陷入困境之中。

加载器醒来！

大多数程序员所认为的模块加载过程实际上分为两个截然不同的步骤。**第一步是把 EXE 或 DLL 映射进内存。**此时加载器查看模块的导入地址表(IAT)来判断这个模块是否依赖于其它 DLL。如果它依赖的 DLL 还未被加载进那个进程，加载器也将它们映射进内存。这个过程递归进行，直到所有依赖的模块都被映射进内存。要查看一个可执行文件隐含依赖的所有 DLL，最好的方法是使用 Platform SDK 附带的 DEPENDS 程序。

第二步是初始化所有 DLL。在第一步中，当操作系统把 EXE 和 DLL 映射进内存时，它并不调用相应的初始化例程。初始化例程是在所有模块都被映射进内存之后才被调用的。关键是：**DLL 被映射进内存的顺序并不需要与它们被初始化的顺序一样**。我曾经见到有人看到 Developer Studio 调试器中对 DLL 映射时的通知而误认为 DLL 是以相同的顺序被初始化的。

在Windows NT中，调用EXE和DLL入口点代码的例程被称为**LdrpRunInitializeRoutines**。在平常的工作中，我已经多次跟踪到LdrpRunIntializeRoutines的汇编代码中。但是，看着大堆的汇编代码并不是理解它的好方法。因此，我用类似C++的伪代码重写了Windows NT 4.0 SP3 的LdrpRunInitializeRoutines函数，如图 1所示。实际上，在NTDLL.DBG中这个例程的名字按__stdcall调用约定被粉碎成了_LdrpRunInitializeRoutines@4。在伪代码中，除了那些名字前面加了下划线的，其余的都是我起的名字。

在 Windows NT 加载器代码中，LdrpRunInitializeRoutines 是调用 EXE 或 DLL 的指定入口点代码之前的最后一站。(在下面的讨论中，我将把 “入口点”和“初始化例程”互换着使用。)这段加载器代码在被加载的 DLL 所在的那个进程环境中执行。也就是说，它并不是什么特别的加载器进程的一部分。在进程启动过程中处理隐含加载的 DLL 时，LdrpRunInitializeRoutines 至少被调用一次。同时，每当动态加载一个或多个 DLL(一般是通常调用 LoadLibrary 实现的)时，都要调用它，

每当 LdrpRunInitializeRoutines 执行时，它就查找并调用已经被映射进内存但还未被初始化的所有 DLL 的入口点代码。在看上面的伪代码时，注意所有提供跟踪输出的额外代码(也就是上面的伪代码中使用_ShowSnaps 变量和_DbgPrint 函数的代码)，它们甚至存在于非调试版的 Windows NT 中。稍候我会接着说这一点。

这个函数大体上分为四个不同的部分。第一部分调用_LdrpClearLoadInProgress 函数。这个 NTDLL 函数返回刚才映射进内存的 DLL 的数目。例如，如果你在 FOO.DLL 中调用 LoadLibrary 函数，而 FOO 隐含链接到了 BAR.DLL 和 BAZ.DLL，那么_LdrpClearLoadInProgress 将返回 3，因为有三个 DLL 被映射进内存中。

在知道了相关的 DLL 数目之后，LdrpRunInitializeRoutines 调用_RtlAllocateHeap(也被称为 HeapAlloc)来为一个指针数组分配内存。在伪代码中我把这个数组称为 pInitNodeArray。这个数组中的每个元素(指针)最终分别指向一个包含有关最近加载(但尚未初始化)的 DLL 的信息的结构。

在 LdrpRunInitializeRoutines 的第二部分中，它使用内部进程数据结构来获取一个包含最近加载的 DLL 的链表。然后它遍历这个链表来确定加载器是否曾经加载过这个 DLL。接下来确定 DLL 是否有入口点函数。如果这两个测试都通过了，它就将指向相应模块信息的指针添加到 pInitNodeArray 数组中。在伪代码中我称这个模块信息为 pModuleLoaderInfo。一定要注意：一

个 DLL 完全有可能不包含入口点函数——例如纯资源 DLL。因此，pInitNodeArray 中的元素数可能比前面由_LdrpClearLoadInProgress 函数返回的值小。

LdrpRunInitializeRoutines 例程的第三部分（也是最大的一部分）才是真正的重头戏。它的任务就是枚举 pInitNodeArray 数组中的每个元素并调用相应的入口点函数。由于 DLL 的初始化代码可能会出错，因此这部分代码整个用一个__try 块包装着。这就是动态加载 DLL 时虽然 DllMain 中出现错误但并不会导致整个进程终止的原因。

遍历一个数组并调用其中的每个入口点函数应该是小菜一碟。然而由于Windows NT中一些灰暗不明的特性使它变得复杂起来。首先需要考虑进程是否正在被像MSDEV.EXE之类的Win32 调试器调试。Windows NT有一个选项允许你在DLL初始化之前将一个进程挂起并把控制权发送到调试器。这个功能是基于DLL的，可以通过向注册表中一个以DLL名称命名的键中添加一个字符串值（BreakOnDllLoad）来实现。详细信息可以参考 图 1 的伪代码中函数_LdrQueryImageFileExecutionOptions的调用代码上面的注释。

在调用 DLL 的入口点函数之前可能需要执行的另一块代码是 TLS 的初始化代码。当你使用__declspec(thread)定义 TLS 变量时，链接器会包含触发这个条件的数据。在 DLL 的入口点函数被调用之前，LdrpRunInitializeRoutines 要确定是否需要初始化 TLS。如果需要，它就调用_LdrpCallTlsInitializers。后面会详细讨论。

LdrpRunInitializeRoutines 中真正调用 DLL 入口点函数的代码终于到来了。我有意用汇编语言来表示这部分代码。原因一会儿就清楚了。这里面最关键的一条指令是 CALL EDI。在这里，EDI 指向 DLL 的入口点函数，而入口点函数是由 DLL 的 PE 文件头指定的。当 CALL EDI 返回时，DLL 已经完成了它的初始化。对于用 C++写的 DLL 来说，这意味着它的 DllMain 函数已经执行完了与 DLL_PROCESS_ATTACH 相应的那部分代码。同时要注意传递给入口点函数的第三个参数，它通常被称为 lpvReserved。事实上，对于可执行文件隐含链接（直接链接或通过其它 DLL 间接链接）到的 DLL 来说，这个参数非 0。对于其它 DLL（即通过调用 LoadLibrary 动态加载的 DLL）来说，这个参数为 0。

DLL 的入口点函数被调用之后,LdrpRunInitializeRoutines 开始进行安全性检查以确保 DLL 的入口点代码中没有错误。它比较调用入口点代码前后堆栈指针（ESP）的值。如果它们不同，那就表明 DLL 的初始化函数出现了错误。由于大多数程序员从未定义过真正的 DLL 入口点函数，这种情况很少发生。但是它一旦发生,Windows 就会用一个对话框通知你这个问题(如图 2 所示)。我不得不使用调试器并在恰当的地方修改寄存器的值才产生了这个对话框。

图 2 非法 DLL 入口点

堆栈检查完毕之后，LdrpRunInitializeRoutines 检查入口点函数的返回值。对于用 C++写的 DLL 来说，它就是 DllMain 的返回值。如果 DLL 返回 0，它通常表示出现了错误，不能继续加

载这个 DLL 了。如果发生这种情况，你就会得到一个令人害怕的“DLL 初始化失败”对话框。

在所有的 DLL 初始化完毕之后开始执行 LdrpRunInitializeRoutines 函数的第三部分中的最后一些代码。如果进程本身的 EXE 文件包含 TLS 数据，并且如果隐含链接到的 DLL 已经被初始化，那它就调用_LdrpCallTlsInitializers。

LdrpRunInitializeRoutines 函数的第四部分（也是最后一部分）是清理代码。还记得前面_RtlAllocateHeap 创建的 pInitNodeArray 数组吗？这部分内存需要被释放，释放它的代码在__finally 块中。这样，即使这些 DLL 中可能有的初始化时会失败，__try/__finally 代码也能保证会调用_RtlFreeHeap 来释放 pInitNodeArray。

我们的 LdrpRunInitializeRoutines 之旅就此结束了，现在让我们来看一下与此相关的一些问题。

调试初始化例程

我也曾经遇到过 DLL 在初始化时失败的情况。不幸的是，错误可能是好几个 DLL 中的一个，而操作系统并没有告诉我到底哪一个才是罪魁祸首。在这种情况下，你就可以使用调试器断点来解决问题。

大多数调试器都直接跳过静态链接的 DLL 的初始化例程。它们把注意力放在 EXE 文件的第一条指令或第一行上。但是知道了 LdrpRunInitializeRoutines 的内部情况之后，你就可以在 CALL EDI 这条指令上设置一个断点，此时正要执行的是 DLL 的入口点代码。一旦设置了这个断点，每次 DLL 将要接到 DLL_PROCESS_ATTACH 通知时，就会中断在 NTDLL 的 CALL 指令上。图 3 是在 Visual C++® 6.0 IDE（MSDEV.EXE）中的情况。

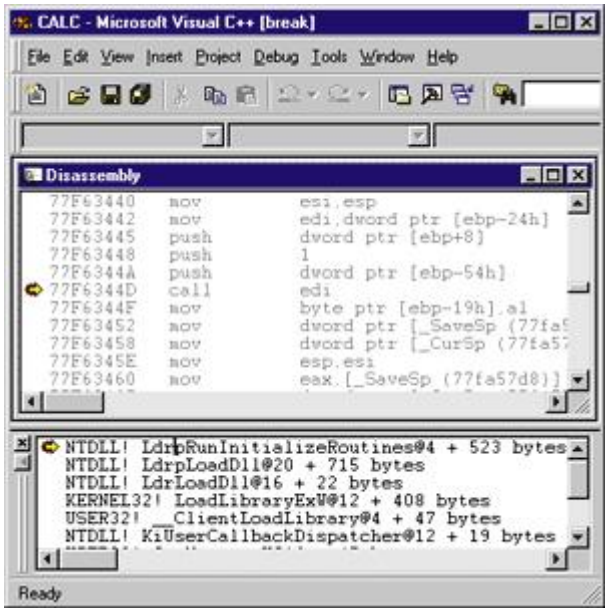

图 3 在 CALL EDI 指令上设置断点

如果单步跟踪 CALL 指令，你会遇到 DLL 入口点代码的第一条指令。意识到这段代码绝大多

数情况下都不是你自己写的这一点很重要。因为它通常是运行时库中的代码，这段代码先做一些准备工作，然后再调用你的初始化代码。例如，在用 Visual C++写的 DLL 中，它的入口点函数是_DllMainCRTStratup,这个函数在 CRTDLL.C 中。在没有调试符号和源代码的情况下,你在 MSDEV 的汇编窗口中看到内容类似下面这个样子（图 4）：

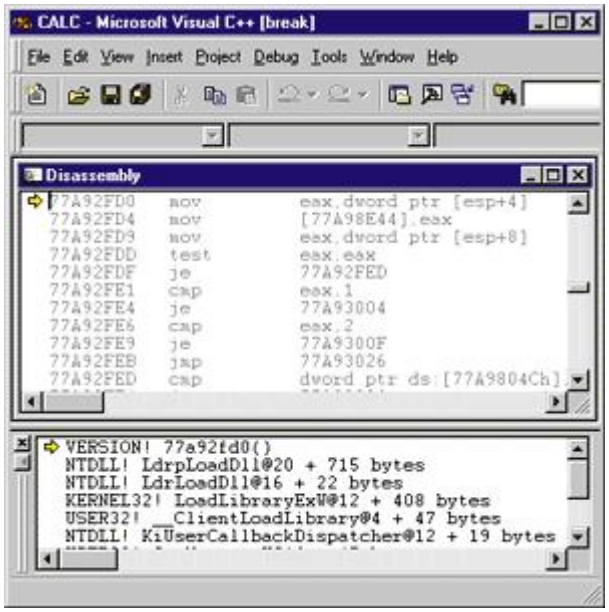

图 4 单步跟踪 CALL 指令

通常我在调试时会按照下面这个过程进行。第一步就是找出哪个 DLL 出现了错误。通常设置前面讲的断点然后单步跟踪到每个 DLL 的初始化例程就可以了。使用调试器找出你当前正处于哪个 DLL 中，并把它记下来。一种方法就是使用调试器的内存窗口来观察堆栈（ESP），获取你进入的 DLL 的 HMODULE。

当你知道进入到了哪个 DLL 之后，让进程继续运行（一般是 Go 命令）。很快就会在下一个 DLL 中再次触发断点。重复这个过程直到你找到有问题的 DLL 为止。你很容易就能找到出错的 DLL，因为它的初始化代码被调用了,但在这个初始化代码返回之前,进程却意外终止了(因为出错了)。

第二步就是仔细检查出错的 DLL。如果你有那个 DLL 的源代码，你最好在 DllMain 上设置一个断点，然后让进程运行等待断点被触发。如果你没有源代码，只管让进程运行，等待你在 CALL EDI 指令上设置的断点在那个位置触发。继续运行直到你碰到出错的指令。单步跟踪进入这个入口点代码并一直单步跟踪下去直到你确定问题所在。这通常需要跟踪大量汇编代码！我从没有说过这很容易，但有时候这是解决问题的惟一方法。

找出 CALL EDI 指令需要一些技巧（至少是在当前的 Microsoft®调试器上）。你现在就能理解我为什么在上面的伪代码中用汇编语言表示这部分代码了。首先，很明显你需要把 NDDLL.DLL 配套的 NTDLL.DBG 文件（现在当然是 NTDLL.PDB 文件）放在你的 SYSTEM32 目录中。当你开始单步跟踪你的程序时，调试器应该会自动加载调试符号。

在 Visual C++的汇编窗口，原理上你可以使用符号名作为地址。在这里，你当然是想转到_LdrpRunInitializeRoutines@4，然后滚动窗口直到你看到 CALL EDI 这条指令。不幸的是，除非你中断在 NTDLL.DLL 中，否则 Visual C++调试器并不能识别 NTDLL 中的符号名。

如果你碰巧知道_LdrpRunInitializeRoutines@4 的地址（在 Intel 平台的 Windows NT 4.0 SP3 上这个地址为 0x77F63242），你可以键入那个地址，汇编窗口很容易就会显示它。IDE 甚至会显示这个函数的名称为_LdrpRunInitializeRoutines@4。如果你不是调试器老手，符号名识别失败让人很困惑。如果你和我一样是个调试器爱好者，这是非常讨厌的，因为你不知道到底问题出在哪里。

Platform SDK 中的 WinDBG 在识别符号名方面稍好一些。一旦你启动了目标进程，你就可以用_LdrpRunInitializeRoutines@4 的名称在这个函数上设置一个断点。不幸的是，当你首次执行这个进程时，你还没来得及在_LdrpRunInitializeRoutines@4 上设置断点，执行流程已经过了这个函数了。为了解决这个问题，启动 WinDBG 后，先单步跟踪一步，然后设置断点并停止调试，仍然保留调试器。然后你可以重启被调试程序，这次断点就会在每一次调用 _LdrpRunInitializeRoutines@4 时被触发。这个技巧也可以用在 Visual C++调试器中。

ShowSnaps 是什么？

_ShowSnaps 这个全局变量是我在查看 LdrpRunInitializeRoutines 的代码时首先注意到的内容之一。趁这个好机会简要地解释一下有关 GlobalFlag 和 GFlags.EXE 方面的内容。

Windows NT 注册表中包含了影响系统代码某些行为的 DWORD 值。它们大部分与堆和调试有关。注册表中 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager 子键下的 GlobalFlag 值是一组位域。知识库文章 Q147314 描述了这些域中的大部分，因此我在这里就不详细讲了。除了系统范围内的 GlobalFlag 值外，各个可执行文件也可以有它们自己的 GlobalFlag 值。与单个进程相关的 GlobalFlag 值被保存在注册表中 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\imagename 这个子键下，这里的 imagename 是可执行文件的名字（例如 WinWord.exe）。所有这些对于编制文档来说都是极大挑战的位域以及嵌套极深的注册表键急需有一个程序来简化。实际上，Microsoft 就提供了一个这样的程序（GFlags.EXE）。（关于 GFlags 工具以及各个标志位的详细含义，可以参考最新的 Microsoft® Debugging Tools for Windows®帮助文档。）

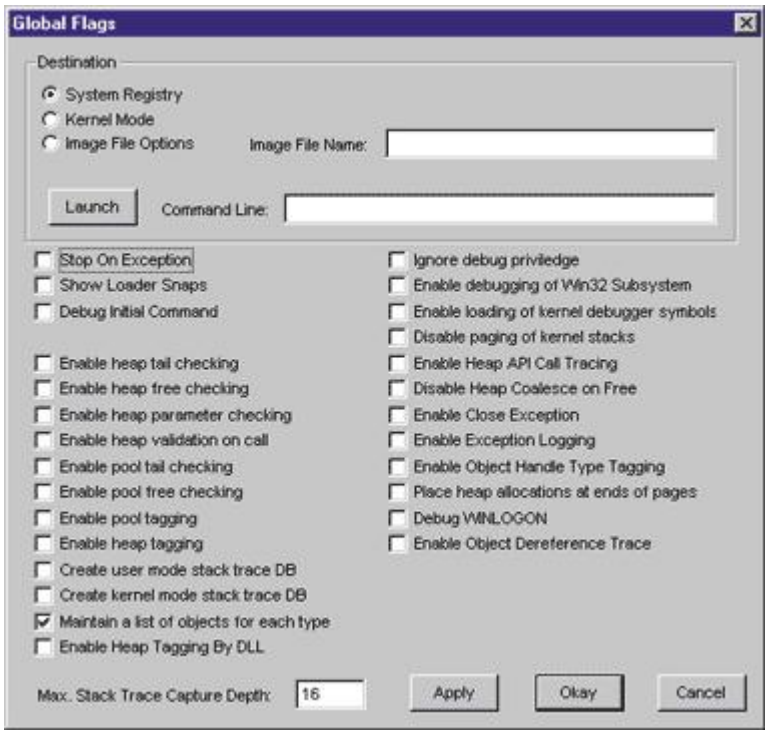

图 5 GFlags.EXE

图 5 显示的是 GFlags.EXE，它来自于 Windows NT 4.0 资源工具包。GFlags.EXE 左上角是三个单选按钮。选择最上面的两个（System Registry 或 Kernel Mode）中任意一个就可以改变 Session Manager 中的 GlobalFlags 的值。如果你选择第三个单选按钮（Image File Options）的话，那么许多选项就会消失。这是因为一些 GlobalFlag 选项只影响内核模式代码，对每个进程来说并无多大意义。需要注意的一点是，大多数只用于内核模式的选项都假定你使用的是诸如 i386kd 之类的系统级调试器。如果不使用这样的调试器深入内部或接收输出信息，那使用这些选项也就没有什么意义了。(最新版本的 GFlags.EXE 除了使用了三个选项卡而不是三个单选按钮外，基本与此类似。)

这些标志中与_ShowSnaps 相关的就是 Show loader snaps 选项。如果它被选中，那么 NTDLL.DLL 中的_ShowSnaps 变量就会被设置成一个非 0 值。在注册表中，这个位是 0x00000002，它被定义为 FLG_SHOW_LDR_SNAPS。幸运的是，这个标志是 GlobalFlag 中可以被设置为针对于每个线程的一些标志中的一个。要不然你要是在系统范围使用这个标志的话，那输出内容会相当多。

检查 ShowSnaps 输出

现在让我们看一下选中Show loader snaps标志后会输出什么类型的信息。我发现没有讨论到的Windows NT加载器的其它部分也会检查这个标志并输出一些信息。图 6就是运行CALC.EXE时输出内容的一部分。要获得这个文本，我首先运行GFlags打开CALC.EXE的Show loader snaps标志。然后我在MSDEV.EXE的控制下运行CALC.EXE，这样就从输出窗口中获得了那些内容。

在图 6 中，注意所有从 NTDLL 中输出的内容前面都加了 LDR:前缀。其它行（例如“Loaded symbols for XXX”）是由 MSDEV 进程插入的。在查看带有 LDR:的行时会发现一些有价值的信息。

例如在进程启动时给出了 EXE 文件的完整路径以及当前目录和搜索路径。

由于 NTDLL 加载各个 DLL 并修正导入函数的地址，因此你会看到类似下面的信息：

```
LDR: ntdll.dll used by SHELL32.dll
LDR: Snapping imports for SHELL32.dll from ntdll.dll
```

第一行表明 SHELL32.DLL 链接到了 NTDLL 中的 API 上。第二行表明了从 NTDLL 导入的 API 正常被“snapped（快照）”。当可执行模块从其它 DLL 导入函数时，在它里面就有一个函数指针数组。这个函数指针数组就是 IAT。加载器的工作之一就是定位导入函数的地址并把它们填入 IAT 中。因此，术语“snapping”就出现在了 LDR：输出中。

输出内容中另一个引起我注意的是正在被处理的 DLL 的绑定信息。

```
LDR: COMCTL32.dll bound to KERNEL32.dll
LDR: COMCTL32.dll has correct binding to KERNEL32.dll
```

在以前的专栏中，我曾经讲过使用 BIND.EXE 程序或 IMAGEHLP.DLL 导出的 BindImageEx 这个 API 来绑定程序。将一个可执行文件绑定到某个 DLL 上实际就是查找导入函数的地址并把它们写入到磁盘上的可执行文件中。这可以加速加载过程，因为加载时不再需要查找导入函数的地址了。

上面的第一行表明 COMCTL32 绑定到了 KERNEL32.DLL 上。第二行表明绑定的地址是正确的。加载器通过比较时间戳来确定这一点。如果时间戳不匹配，那么绑定就是无效的。在这种情况下，加载器就重新查找导入函数的地址，就好像这个可执行文件并没有绑定一样。

TLS 初始化

最后我以另一个例程的伪代码来结束本期专栏。在LdrpRunInitializeRoutines函数中，在调用模块的入口点代码前的最后一刻，NTDLL检查这个模块是否需要初始化TLS。如果需要，它就调用LdrpCallTlsInitializers函数来进行初始化。图 7是我为这个例程写的伪代码。

这个函数相当简单。PE 文件头中保存了 IMAGE_TLS_DIRECTORY 结构（在 WINNT.H 中定义）的偏移（RVA）。这个函数调用 RtlImageDirecotoryEntryToData 来获取指向这个结构的指针。IMAGE_TLS_DIRECTORY 结构中保存了一个指针，它指向一个由回调函数的地址组成的数组。这些回调函数被声明为 PIMAGE_TLS_CALLBACK 类型的函数，这个类型在 WINNT.H 中定义。TLS 初始化回调函数与 DllMain 函数非常相似。实际上在使用__declspec(thread)定义变量时，Visual C++ 生成了一些导致这些函数会被调用的数据。但是当前运行时库并未定义实际的回调函数，因此这个函数指针数组只有一个值为 NULL 的元素。

总结

我对 Windows NT 模块初始化方面的讨论已经结束了。很明显我跳过了许多相关内容。例如确定模块初始化顺序的算法是什么？Windows NT 上的这个算法至少已经改变过一次，如果有 Microsoft technical note 就好了，至少它可以给我们一些指导。同样，我也没有讨论与模块加载对应的话题：模块卸载。然而，我希望我对 Windows NT 加载器内部工作过程的“一瞥”能够为你更深层次的探索提供一些材料。

附录

图 1 RunInit.cpp

```
//============================================================================
// Matt Pietrek, September 1999 Microsoft Systems Journal
//
// NTDLL.DLL 中 LdrpRunInitializeRoutines 例程的伪代码（NT 4，SP3）
//
// 在一个进程中首次调用 LdrpRunInitializeRoutines（也就是在初始化
// 隐含链接的模块）时，bImplicitLoad 参数不为 0；在后续的调用
// (通过调用 LoadLibrary 而间接调用此例程)中，bImplicitLoad 为 0。
//
//============================================================================

#include <ntexapi.h>    // 用于函数末尾的 HardError 定义

// 以下是全局符号（这些名字是准确的，它们来自 NTDLL.DBG 文件）
//  _NtdllBaseTag
//  _ShowSnaps
//  _SaveSp
//  _CurSp
//  _LdrpInLdrInit
//  _LdrpFatalHardErrorCount
//  _LdrpImageHasTls

NTSTATUS
LdrpRunInitializeRoutines( DWORD bImplicitLoad )
{
    // 获取可能需要被初始化的模块数。其中的一些可能已经被初始化了
    unsigned nRoutinesToRun = _LdrpClearLoadInProgress();

    if ( nRoutinesToRun )
    {
        // 如果存在需要初始化的模块，就为保存模块相关信息的数组分配内存
        pInitNodeArray = _RtlAllocateHeap(GetProcessHeap(),
                                          _NtdllBaseTag + 0x60000,
                                          nRoutinesToRun * 4 );

        if ( 0 == pInitNodeArray )    // 确保内存分配成功
            return STATUS_NO_MEMORY;
    }
    else
        pInitNodeArray = 0;
```

```
//
// 进程环境块（Process Environment Block，Peb）中保存了一个指向已加载
// 模块链表的指针。现在获取这个指针。
//
pCurrNode = *(pCurrentPeb->ModuleLoaderInfoHead);
ModuleLoaderInfoHead = pCurrentPeb->ModuleLoaderInfoHead;

if ( _ShowSnaps )
{
    _DbgPrint( "LDR: Real INIT LIST\n" );
}

nModulesInitedSoFar = 0;

if ( pCurrNode != ModuleLoaderInfoHead )
{
    //
    // 遍历链表
    //
    while ( pCurrNode != ModuleLoaderInfoHead )
    {
        ModuleLoaderInfo  pModuleLoaderInfo;

        //
        // 显然指向下一个结点的指针在 ModuleLoaderInfo 结构中的 0x10 字节处
        //
        pModuleLoaderInfo = &NextNode - 0x10;

        // 这条语句看起来好像没有什么作用
        localVar3C = pModuleLoaderInfo;

        //
        // 确定模块是否已经被初始化。如果是，就跳过它
        //
        // X_LOADER_SAW_MODULE = 0x40
        if ( !(pModuleLoaderInfo->Flags35 & X_LOADER_SAW_MODULE) )
        {
            //
            // 此模块还未被初始化。检查它是否有入口点函数
            //
            if ( pModuleLoaderInfo->EntryPoint )
            {
                //
```

```
                // 这个未初始化的模块有入口点函数。将它添加到
                // pInitNodeArray 数组中。此函数会在后面初始化
                // 这个数组中的模块。
                //
                pInitNodeArray[nModulesInitedSoFar] =pModuleLoaderInfo;

                // 若 ShowSnaps 不为 0，输出模块的路径及其入口点地址。例如：
                //
                // C:\WINNT\system32\KERNEL32.dll init routine 77f01000
                if ( _ShowSnaps )
                {
                    _DbgPrint(  "%wZ init routine %x\n",
                                &pModuleLoaderInfo->24,
                                pModuleLoaderInfo->EntryPoint );
                }

                nModulesInitedSoFar++;
            }
        }

        // 设置此模块的 X_LOADER_SAW_MODULE 标志。注意：此时模块实际
        // 并未被初始化。要等到这个函数快结束时才初始化
        pModuleLoaderInfo->Flags35 &= X_LOADER_SAW_MODULE;

        // 移向模块列表中的下一个结点
        pCurrNode = pCurrNode->pNext
    }
}
else
{
    pModuleLoaderInfo = localVar3C;     // 可能未被初始化吗???
}

if ( 0 == pInitNodeArray )
    return STATUS_SUCCESS;



// ************************ MSJ 建议代码布局！ *****************
// 如果由于页面限制需要将代码分开，这里是最好的分隔点。
// 不过要记住移去这个注释
// ************************ MSJ 建议代码布局！ *****************
```

```
//
// 现在 pInitNodeArray 数组中包含的是未初始化的模块的信息的指针。
// 是调用它们的初始化例程的时候了。
//
try     // 用 try 块将整个代码包装起来，以防初始化例程失败。
{
    nModulesInitedSoFar = 0;  // 从索引为 0 的数组元素开始

    //
    // 开始遍历整个模块数组
    //
    while ( nModulesInitedSoFar < nRoutinesToRun )
    {
        // 获取有关模块信息的指针
        pModuleLoaderInfo = pInitNodeArray[ nModulesInitedSoFar ];

        // 这条语句好像没什么作用
        localVar3C = pModuleLoaderInfo;

        nModulesInitedSoFar++;

        // 将初始化例程的地址保存在一个局部变量中
        pfnInitRoutine = pModuleLoaderInfo->EntryPoint;

        fBreakOnDllLoad = 0;    // 默认加载时不中断

        //
        // 如果进程正处于被调试状态，确认一下是否应该在调用
        // 初始化例程之前中断在调试器中
        //
        // DebuggerPresent(在 PEB 结构中的偏移 2 处)是 IsDebuggerPresent()
        // 返回的内容。这个 API 仅存在于 Windows NT 上
        //
        if ( pCurrentPeb->DebuggerPresent || pCurrentPeb->1 )
        {
            LONG retCode;

            //
            // 查询注册表中的“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
            // Windows NT\CurrentVersion\Image File Execution Options”
            // 这个键。如果它下面存在一个以这个可执行文件名命名的子键，
            // 就检查这个子键下的 BreakOnDllLoad 值
            //
            retCode =
```

```
        _LdrQueryImageFileExecutionOptions(
                    pModuleLoaderInfo->pwszDllName,
                    "BreakOnDllLoad",
                    REG_DWORD,
                    &fBreakOnDllLoad,
                    sizeof(DWORD),
                    0 );

    // 如果未找到这个值（通常是这样），在初始化 DLL 时就不中断
    if ( retCode <= STATUS_SUCCESS )
        fBreakOnDllLoad = 0;
}

if ( fBreakOnDllLoad )
{
    if ( _ShowSnaps )
    {
        // 在实际中断进调试器之前，输出模块名称和初始化例程的地址
        _DbgPrint(  "LDR: %wZ loaded.",
                    &pModuleLoaderInfo->pModuleLoaderInfo );

        _DbgPrint(  "- About to call init routine at %lx\n",
                    pfnInitRoutine )
    }

    // 中断进调试器
    _DbgBreakPoint();   // 它实际是一条 INT 3 指令，后面跟着 RET 指令
}
else if ( _ShowSnaps && pfnInitRoutine )
{
    // 在调用初始化例程之前输出模块名称和初始化例程的地址
    _DbgPrint(  "LDR: %wZ loaded.",
                pModuleLoaderInfo->pModuleLoaderInfo );

    _DbgPrint("- Calling init routine at %lx\n", pfnInitRoutine);
}

if ( pfnInitRoutine )
{
    // 设置标志来表明已将 DLL_PROCESS_ATTACH 通知发送给了 DLL
    //
    // （难道这不应该是在实际调用初始化例程之后才设置吗？）
    //
    // X_LOADER_CALLED_PROCESS_ATTACH = 0x8
```

```
pModuleLoaderInfo->Flags36 |= X_LOADER_CALLED_PROCESS_ATTACH;

//
// 如果此模块使用了线程局部存储（TLS），现在调用 TLS 初始化函数
// *** 注意 ***
// 这仅发生在一个进程首次调用此函数时（也就是在初始化隐含
// 链接的 DLL 时）。动态加载的 DLL 不应该使用 TLS 变量，正如
// SDK 文档所说的那样
//
if ( pModuleLoaderInfo->bHasTLS && bImplicitLoad )
{
    _LdrpCallTlsInitializers(   pModuleLoaderInfo->hModDLL,
                                DLL_PROCESS_ATTACH );
}


hModDLL = pModuleLoaderInfo->hModDLL

MOV     ESI,ESP // 将 ESP 寄存器的值保存到 ESI 中

MOV     EDI,DWORD PTR [pfnInitRoutine]      // 将模块的入口点
                                            //地址加载到 EDI 中

//以下的汇编语言代码用 C++代码表示就是：
//
// initRetValue =
// pfnInitRoutine(hInstDLL,DLL_PROCESS_ATTACH,bImplicitLoad);
//

PUSH    DWORD PTR [bImplicitLoad]
PUSH    DLL_PROCESS_ATTACH
PUSH    DWORD PTR [hModDLL]

CALL    EDI     // 调用初始化例程。这是设置断点的最佳位置。
                // 单步跟踪这个调用就进入到了 DLL 的入口点中

MOV     BYTE PTR [initRetValue],AL  // 保存入口点函数的返回值

MOV     DWORD PTR [_SaveSp],ESI // 保存入口点函数返回后的
MOV     DWORD PTR [_CurSp],ESP  // 堆栈指针的值

MOV     ESP,ESI     // 恢复调用入口点函数之前 ESP 中的值

//
```

```
// 校验调用前后堆栈指针（ESP）的值。如果它们不同，这
// 表明 DLL 的初始化例程并没有正确地清理堆栈。例如，它的
// 入口点函数可能定义的不正确。尽管这极少发生，但是如果
// 它确实发生了，我们要通知用户并让他们决定是否继续执行
//
if ( _CurSP != _SavSP )
{
    hardErrorParam = pModuleLoaderInfo->FullDllPath;

    hardErrorRetCode =
        _NtRaiseHardError(
            STATUS_BAD_DLL_ENTRYPOINT | 0x10000000,
            1,  // 参数个数
            1,  // UnicodeStringParametersMask,
            &hardErrorParam,
            OptionYesNo,    // 让用户决定
            &hardErrorResponse );

    if ( _LdrpInLdrInit )
        _LdrpFatalHardErrorCount++;

    if (    (hardErrorRetCode >= STATUS_SUCCESS)
        &&  (ResponseYes == hardErrorResponse) )
    {
        return STATUS_DLL_INIT_FAILED;
    }
}

//
// 如果 DLL 的入口点函数返回 0（表示失败），通知用户
//
if ( 0 == initRetValue )
{
    DWORD hardErrorParam2;
    DWORD hardErrorResponse2;

    hardErrorParam2 = pModuleLoaderInfo->FullDllPath;

    _NtRaiseHardError(  STATUS_DLL_INIT_FAILED,
                        1,  // 参数个数
                        1,  // UnicodeStringParametersMask
                        &hardErrorParam2,
                        OptionOk,   // 只能以“确定”作为响应
                        &hardErrorResponse2 );
```

```
                    if ( _LdrpInLdrInit )
                        _LdrpFatalHardErrorCount++;

                    return STATUS_DLL_INIT_FAILED;
                }
            }
        }

        //
        // 如果这个进程自身的 EXE 文件定义了 TLS 变量，现在调用 TLS 初始化例程。
        // 要获取更详细的信息，参考前面调用_LdrpCallTlsInitializers 时的注释
        //
        if ( _LdrpImageHasTls && bImplicitLoad )
        {
            _LdrpCallTlsInitializers(   pCurrentPeb->ProcessImageBase,
                                        DLL_PROCESS_ATTACH );
        }
    }
    __finally
    {
        //
        // 在这个函数退出之前，确保它在前面分配的内存被释放
        //
        _RtlFreeHeap( GetProcessHeap(), 0, pInitNodeArray );
    }

    return STATUS_SUCCESS;
}
```

返回

图 6　CALC.EXE的ShowSnaps输出信息

```
## 以##开头的是我的注释
Loaded 'C:\WINNT\system32\CALC.EXE', no matching symbolic information found.
Loaded symbols for 'C:\WINNT\system32\ntdll.dll'
LDR: PID: 0x3a started - '"C:\WINNT\system32\CALC.EXE"'
LDR: NEW PROCESS
     Image Path: C:\WINNT\system32\CALC.EXE (CALC.EXE)
     Current Directory: C:\WINNT\system32
     Search Path: C:\WINNT\system32;.;C:\WINNT\System32;C:\WINNT\system;...
LDR: SHELL32.dll used by CALC.EXE
Loaded 'C:\WINNT\system32\SHELL32.DLL', no matching symbolic information found.
LDR: ntdll.dll used by SHELL32.dll
LDR: Snapping imports for SHELL32.dll from ntdll.dll
```

```
LDR: KERNEL32.dll used by SHELL32.dll
Loaded symbols for 'C:\WINNT\system32\KERNEL32.DLL'
LDR: ntdll.dll used by KERNEL32.dll
LDR: Snapping imports for KERNEL32.dll from ntdll.dll
LDR: Snapping imports for SHELL32.dll from KERNEL32.dll
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlEnterCriticalSection
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlDeleteCriticalSection//其余部分省略....
LDR: GDI32.dll used by SHELL32.dll
Loaded symbols for 'C:\WINNT\system32\GDI32.DLL'
LDR: ntdll.dll used by GDI32.dll
LDR: Snapping imports for GDI32.dll from ntdll.dll
LDR: KERNEL32.dll used by GDI32.dll
LDR: Snapping imports for GDI32.dll from KERNEL32.dll
LDR: USER32.dll used by GDI32.dll
Loaded symbols for 'C:\WINNT\system32\USER32.DLL'
LDR: ntdll.dll used by USER32.dll
LDR: Snapping imports for USER32.dll from ntdll.dll
LDR: KERNEL32.dll used by USER32.dll
LDR: Snapping imports for USER32.dll from KERNEL32.dll
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlSizeHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlReAllocateHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlFreeHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlAllocateHeap//其余部分省略....
## 注意加载器开始查找并校验 COMCTL32 导入并绑定的 DLL
Loaded 'C:\WINNT\system32\COMCTL32.DLL', no matching symbolic information found.
LDR: COMCTL32.dll bound to ntdll.dll
LDR: COMCTL32.dll has correct binding to ntdll.dll
LDR: COMCTL32.dll bound to GDI32.dll
LDR: COMCTL32.dll has correct binding to GDI32.dll
LDR: COMCTL32.dll bound to KERNEL32.dll
LDR: COMCTL32.dll has correct binding to KERNEL32.dll
LDR: COMCTL32.dll bound to ntdll.dll via forwarder(s) from KERNEL32.dll
LDR: COMCTL32.dll has correct binding to ntdll.dll
LDR: COMCTL32.dll bound to USER32.dll
LDR: COMCTL32.dll has correct binding to USER32.dll
LDR: COMCTL32.dll bound to ADVAPI32.dll
LDR: COMCTL32.dll has correct binding to ADVAPI32.dll//其余部分省略....
LDR: Refcount   COMCTL32.dll (1)
```

```
LDR: Refcount   GDI32.dll (3)
LDR: Refcount   KERNEL32.dll (6)
LDR: Refcount   USER32.dll (4)
LDR: Refcount   ADVAPI32.dll (5)
LDR: Refcount   KERNEL32.dll (7)
LDR: Refcount   GDI32.dll (4)
LDR: Refcount   USER32.dll (5)## List of implicit link DLLs to be init'ed.
LDR: Real INIT LIST
     C:\WINNT\system32\KERNEL32.dll init routine 77f01000
     C:\WINNT\system32\RPCRT4.dll init routine 77e1b6d5
     C:\WINNT\system32\ADVAPI32.dll init routine 77dc1000
     C:\WINNT\system32\USER32.dll init routine 77e78037
     C:\WINNT\system32\COMCTL32.dll init routine 71031a18
     C:\WINNT\system32\SHELL32.dll init routine 77c41094
## 开始实际调用隐含链接的 DLL 的初始化例程
LDR: KERNEL32.dll loaded. - Calling init routine at 77f01000
LDR: RPCRT4.dll loaded. - Calling init routine at 77e1b6d5
LDR: ADVAPI32.dll loaded. - Calling init routine at 77dc1000
LDR: USER32.dll loaded. - Calling init routine at 77e78037
## USER32 开始做与 AppInit_DLLs 有关的工作，因此静态初始化被暂时中断
## 这个例子中，“globaldll.dll”是在 USER32 的初始化代码中由 LoadLibrary 加载的
LDR: LdrLoadDll, loading c:\temp\globaldll.dll from C:\WINNT\system32;.;
LDR: Loading (DYNAMIC) c:\temp\globaldll.dll
Loaded 'C:\TEMP\GlobalDLL.dll', no matching symbolic information found.
LDR: KERNEL32.dll used by globaldll.dll//其余部分省略....
LDR: Real INIT LIST
     c:\temp\globaldll.dll init routine 10001310
LDR: globaldll.dll loaded. - Calling init routine at 10001310
## 现在接着初始化隐含链接的 DLL
LDR: COMCTL32.dll loaded. - Calling init routine at 71031a18
LDR: LdrGetDllHandle, searching for USER32.dll from
LDR: LdrGetProcedureAddress by NAME - GetSystemMetrics
LDR: LdrGetProcedureAddress by NAME - MonitorFromWindow
LDR: SHELL32.dll loaded. - Calling init routine at 77c41094
//其余部分省略....
```

返回

图 7 TLSInit.cpp

```
void _LdrpCallTlsInitializers( HMODULE hModule, DWORD fdwReason )
{
    PIMAGE_TLS_DIRECTORY pTlsDir;
    DWORD size

    // 从 IMAGE_OPTIONAL_HEADER.DataDirectory 中查找 TLS 目录
```

```
    pTlsDir = _RtlImageDirectoryEntryToData(hModule,
                                            1,
                                            IMAGE_DIRECTORY_ENTRY_TLS,
                                            &size );

    __try   // 用 try/catch 块保护所有代码
    {
        if ( pTlsDir->AddressOfCallbacks )
        {
            if ( _ShowSnaps )   // 输出诊断信息
            {
                _DbgPrint(  "LDR: Tls Callbacks Found. "
                            "Imagebase %lx Tls %lx CallBacks %lx\n",
                            hModule, TlsDir, pTlsDir->AddressOfCallbacks );
            }

            // 获取指向包含 TLS 回调函数地址的数组的起始位置的指针
            PVOID * pCallbacks = pTlsDir->AddressOfCallbacks;

            while ( *pCallbacks )   // 遍历数组中的每一个元素
            {
                PIMAGE_TLS_CALLBACK pTlsCallback = *pCallbacks;
                pCallbacks++;

                if ( _ShowSnaps )   // 输出更多诊断信息
                {
                    _DbgPrint(  "LDR: Calling Tls Callback "
                                "Imagebase %lx Function %lx\n",
                                hModule, pTlsCallback );
                }

                // 实际调用回调函数
                pTlsCallback( hModule, fdwReason, 0 );
            }
        }
    }
    __except( EXCEPTION_EXECUTE_HANDLER )
    {
    }
}
```

返回

（*Microsoft System Journal* 1999 年 9 月Under The Hood专栏）

译者：SmartTech　　电子信箱：zhzhtst@163.com