漫谈兼容内核之十一：
Windows DLL 的装入和连接

毛德操

在 PE 映像的装入和启动过程中，DLL 的装入和连接是一个重要的环节。读者在上一篇漫谈中看到，Windows 的 DLL 装入(除 ntdll.dll 外)和连接是通过 ntdll.dll 中的一个函数 LdrInitializeThunk()实现的。在 Wine 中，这个环节也是通过一个同名的函数实现的，只不过这个函数不在 ntdll.dll 中，而是 wine-kthread 里面的一个函数。在 ReactOS 中则同样也是 LdrInitializeThunk()，同样也在 ntdll.dll 中。DLL 的装入和连接当然离不开 PE 格式，但是本文的目的不在于完整、系统地介绍 PE 格式，而在于从 LdrInitializeThunk()这个函数入手讲述 DLL 动态连接的过程，这个过程涉及 PE 格式中的什么成分，就讲解什么成分。这样，整个过程下来，PE 格式中关键性的部件也就差不多都见到了。

先对 LdrInitializeThunk()这个函数名作些解释。“Ldr”显然是“Loader”的缩写。而“Thunk”意为“翻译”、“转换”、或者某种起着“桥梁”作用的东西。这个词在一般的字典中是查不到的，但却是个常见于微软的资料、文档中的术语。这个术语起源于编译技术，表示一小片旨在获取某个地址的代码，最初用于函数调用时“形参”和“实参”的结合。后来这个术语有了不少新的特殊含义和使用，但是 DLL 的动态连接与函数调用时的“形实结合”确实有着本质的相似。其实 Unix/Linux 也常常用到类似的技术，只是很少使用这个术语。例如，在 Linux 内核(i386 版本)中，current 是作为指针使用的，但实际上却是个宏操作：

```
#define current get_current()
```

而 get_current()是一个函数：

```
static inline struct task_struct * get_current(void)
{
    struct task_struct *current;
    __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
    return current;
}
```

这显然就是一个 Thunk。还有，ELF 映像(与.so 模块)的连接所用的“过程连接表”PLT 和“全局位移表”GOT 本质上也是 Thunk。所以，Thunk 就是“动态连接”的同义词。

下面我们就来看 LdrInitializeThunk()的代码。由于 Windows 不公开它的代码，而 Wine 需要处理 PE 和 ELF 两种格式的动态连接库，相比之下略为复杂一些，所以我们先看 ReactOS 代码中的这个函数，其代码在 reactos/ib/ntdll\ldr/entry.S 中：

```
.globl _LdrInitializeThunk@16
_LdrInitializeThunk@16:
#if defined(_M_IX86)
  nop               /* breakin overwrites this with "int 3" */
```

```
 jmp ___true_LdrInitializeThunk@16
 #elif defined(_M_ALPHA)
. . . . . .
```

由于是汇编代码，所以在函数名前面加上了‘_’，而后缀@16 则表示调用参数一共是 16 字节、即 4 个参数(不过下面的代码并未用到这 4 个参数)。其实这个函数的本身就是个 Thunk，因为它起的只是桥梁、跳板的作用，真正起作用的是__true_LdrInitializeThunk()。

在进入这个函数之前，目标 EXE 映像已经被映射到当前进程的用户空间，系统 DLL ntdll.dll 的映像也已经被映射，但是并没有在 EXE 映像与 ntdll.dll 映像之间建立连接(实际上 EXE 映像未必就直接调用 ntdll.dll 中的函数)。LdrInitializeThunk()是 ntdll.dll 中不经连接就可进入的函数，实质上就是 ntdll.dll 的入口。除 ntdll.dll 以外，别的 DLL 都还没有被装入(映射)。此外，当前进程(除内核中的“进程控制块”EPROCESS 等数据结构外)在用户空间已经有了一个“进程环境块”PEB，以及该进程的第一个“线程环境块”TEB。这就是进入__true_LdrInitializeThunk()前的“当前形势”。

```
VOID STDCALL
__true_LdrInitializeThunk (ULONG Unknown1, ULONG Unknown2,
                           ULONG Unknown3, ULONG Unknown4)
{
    . . . . . .

    DPRINT("LdrInitializeThunk()\n");
    if (NtCurrentPeb()->Ldr == NULL || NtCurrentPeb()->Ldr->Initialized == FALSE)
    {
        Peb = (PPEB)(PEB_BASE);
        DPRINT("Peb %x\n", Peb);
        ImageBase = Peb->ImageBaseAddress;
        . . . . . .
        /* Initialize NLS data */
        RtlInitNlsTables (Peb->AnsiCodePageData, Peb->OemCodePageData,
                          Peb->UnicodeCaseTableData, &NlsTable);
        RtlResetRtlTranslations (&NlsTable);

        NTHeaders = (PIMAGE_NT_HEADERS)(ImageBase + PEDosHeader->e_lfanew);
        . . . . . .
        /* create process heap */
        RtlInitializeHeapManager();
        Peb->ProcessHeap = RtlCreateHeap(HEAP_GROWABLE, NULL,
                                         NTHeaders->OptionalHeader.SizeOfHeapReserve,
                                         NTHeaders->OptionalHeader.SizeOfHeapCommit,
                                         NULL, NULL);
        . . . . . .

        /* create loader information */
```

```
        Peb->Ldr = (PPEB_LDR_DATA)RtlAllocateHeap (Peb->ProcessHeap,
                                                   0,
                                                   sizeof(PEB_LDR_DATA));
        . . . . . .
        Peb->Ldr->Length = sizeof(PEB_LDR_DATA);
        Peb->Ldr->Initialized = FALSE;
        Peb->Ldr->SsHandle = NULL;
        InitializeListHead(&Peb->Ldr->InLoadOrderModuleList);
        InitializeListHead(&Peb->Ldr->InMemoryOrderModuleList);
        InitializeListHead(&Peb->Ldr->InInitializationOrderModuleList);

        . . . . . .
        /* add entry for ntdll */
        NtModule = (PLDR_MODULE)RtlAllocateHeap (Peb->ProcessHeap,
                                                 0,
                                                 sizeof(LDR_MODULE));
        . . . . . .
        InsertTailList(&Peb->Ldr->InLoadOrderModuleList,
                       &NtModule->InLoadOrderModuleList);
        InsertTailList(&Peb->Ldr->InInitializationOrderModuleList,
                       &NtModule->InInitializationOrderModuleList);
        . . . . . .
        /* add entry for executable (becomes first list entry) */
        ExeModule = (PLDR_MODULE)RtlAllocateHeap (Peb->ProcessHeap,
                                                  0,
                                                  sizeof(LDR_MODULE));
        . . . . . .
        InsertHeadList(&Peb->Ldr->InLoadOrderModuleList,
                       &ExeModule->InLoadOrderModuleList);
        . . . . . .
        EntryPoint = LdrPEStartup((PVOID)ImageBase, NULL, NULL, NULL);
        . . . . . .
    }
    /* attach the thread */
    RtlEnterCriticalSection(NtCurrentPeb()->LoaderLock);
    LdrpAttachThread();
    RtlLeaveCriticalSection(NtCurrentPeb()->LoaderLock);
}
```

这个函数的开头一段让我们看到了“进程环境块”PEB 的一部分作用。比方说，PEB 中的 ImageBaseAddress 字段是个指针，指向 EXE 映像在用户空间的起点，这是由内核为其设置好的；PEB 中的 AnsiCodePageData、OemCodePageData、和 UnicodeCaseTableData 等字段则与人机界面的语言本地化有关。还有，PEB 中的 ProcessHeap 字段指向本进程用户空间可动态分配的内存区块“堆”、即 Heap，这是一个可供动态分配的内存区块队列；而函数

RtlAllocateHeap(Peb->ProcessHeap, …)则从这个堆分配空间。不过此前首先要通过RtlCreateHeap()创建一个这样的堆、及其第一个区块，其初始的大小来自映像头部的建议值，其中 SizeOfHeapReserve 是估计的最大值，SizeOfHeapCommit 是初始值，这是在编译/连接时确定的。PEB 中的另一个字段 Ldr 是个 PEB_LDR_DATA 结构指针，所指向的数据结构用来为本进程维持三个“模块”队列、即 InLoadOrderModuleList、InMemoryOrderModuleList、和 InInitializationOrderModuleList。这里所谓“模块”就是 PE 格式的可执行映像，包括 EXE 映像和 DLL 映像。前两个队列都是模块队列，第三个是初始化队列。两个模块队列的不同之处在于排列的次序，一个是按装入的先后，一个是按装入的位置(实际上目前 ReactOS 的代码中并未使用这个队列)。每当为本进程装入一个模块、即.exe 映像或 DLL 映像时，就要为其分配/创建一个 LDR_MODULE 数据结构，并将其挂入 InLoadOrderModuleList。然后，完成对这个模块的动态连接以后，就把它挂入 InInitializationOrderModuleList 队列，以便依次调用它们的初始化函数。相应地，LDR_MODULE 数据结构中有三个队列头，因而可以同时挂在三个队列中。

这样，有了 InLoadOrderModuleList 队列以后，就可以避免重复装入(映射)同一个模块。要是模块 A 引入(import)模块 B 和 C(即需要调用模块 B 和 C 中的函数)，而 B 又要求引入 C；那么在装入 A、B、C 三个模块、并连接 A 与 B、A 与 C 以后，当处理 B 的引入时，由于 C 已经在这个队列中，就不需要再次装入，而只要建立 B 与 C 之间的连接就可以了。

到此刻为止，已经装入用户空间的模块只有两个，即目标 EXE 映像和 ntdll.dll 的映像，但是尚未连接。所以这里为这两个模块分配 LDR_MODULE 数据结构，并将它们挂入 InLoadOrderModuleList 队列。不过 ntdll.dll 是最底层的模块，它不再引入别的 DLL，也无需连接，所以这里也将其挂入了初始化队列。

PEB 是用户空间的程序中常常要用到的数据结构，这里分别使用了两种手段来获取当前 PEB 的地址。一是使用 NtCurrentPeb()，这一般是一个宏定义：

```
#define NtCurrentPeb() (NtCurrentTeb()->Peb)
```

就是说，要获取 PEB 的地址，先获取 TEB 的地址，而 TEB 中有个指针指向 PEB。那么怎样获取 TEB 的地址呢？ntdll.dll 为此提供了一个库函数 NtCurrentTeb()。

如果所有的应用程序和别的 DLL 都用由 ntdll.dll 提供的 NtCurrentTeb()来获取 TEB 的地址和 PEB 的地址，那么 TEB 和 PEB 实际所在的位置就无关紧要，只要 ntdll.dll 知道 TEB 在哪里就行了。可是事情却并不那么理想，应用程序和别的 DLL 事实上还可能用别的办法来获取 TEB 的地址，例如，有些应用程序(或 DLL)可能自己搞一个 NtCurrentTeb()：

```
/* on the x86, the TEB is contained in the FS segment */
static inline struct _TEB * NtCurrentTeb(void)
{
  struct _TEB * pTeb;

  __asm mov eax, fs:0x18
  __asm mov pTeb, eax

  return pTeb;
}
```

“Secrets”一书中(234 页和 428-430 页)对此有说明：当 CPU 运行于用户模式时，段寄存器 fs:0(通过 LDT 中的相应的段描述符)指向当前线程的 TEB。而 TEB 结构中的第一个成分是“线程信息块”TIB，TIB 中位移为 0x18 处是个指针“Self”，指向这个数据结构本身的起点。所以 fs:0x18 就是 TIB 的起点，也就是 TEB 的起点。可是为什么不直接获取寄存器 fs 的内容呢？因为段寄存器 fs 是一个 16 位的寄存器，在“保护模式”中它的内容只是一个“段选择符”，只是被用来选取 LDT 中的一个“段描述符”，段描述符中所含的地址才是真正的地址。这个地址被装入 CPU 中 fs 的“隐藏”部分，而隐藏部分仅供 CPU 自己使用，对于程序员是不可见的(详见“Intel Architecture Software Developer’s Manual”第三卷)。可想而知，不同线程的 fs 所指向的段描述符有着不同的内容。

这一来，TEB 和 PEB 实际所在的位置就不是无关紧要、而是至关重要了。而且，不光是 TEB 和 PEB 所在的位置，连 fs 的内容、以及相应段描述符的内容也至关重要了；因为即使 TEB 和 PEB 确实在它们应该在的地方，可是 fs 和相应段描述符的内容不正确，那也还是有问题。段描述符的内容只能在内核中(系统模式下)加以设置，Linux 为此提供了一个系统调用 modify_ldt()，让应用程序可以在用户空间通过这个系统调用设置自己的 LDT。Wine 就是在 wine-kthread 或 wine-pthread 的 thread_init()中用 modify_ldt()建立起自己的 TEB 和 PEB 的。

再看第二种手段，那就是上面程序中所用的“Peb = (PPEB)(PEB_BASE)”，这里 PEB_BASE 定义为 0x7FFDF000。这也意味着 PEB 的位置是固定的，而且必须在 0x7FFDF000 这个地方。显然，要对 Windows 软件提供较好的兼容性，就必须遵守这些规矩。

下面就是这里的主题、即对 LdrPEStartup()的调用了，除 ntdll.dll 以外的所有 DLL 的装入和(向下)连接都是由它完成的。当 CPU 从 LdrPEStartup()返回时，EXE 对象需要直接或间接引入的所有 DLL 均已映射到用户空间并已完成连接，对 EXE 模块的“启动”、即初始化也已完成。再往下就是对 LdrpAttachThread()的调用，目的是调用各个 DLL 的初始化过程，以及对 TLS、即“线程本地存储(Thread Local Storage)”的初始化。如果具体的 TLS 有需要在初始化时加以回调的函数，则要加以调用。同一进程中的所有线程都共享同一用户空间，一般而言只有堆栈是只属于具体线程的，所以除局部变量以外就不再有线程自己的“私房”。但是局部变量是暂时的，一旦从其所在的函数返回就不复存在。可是，有时候又确实需要让每个线程都对于同一个全局变量有一份自己的拷贝，TLS 就是为此而设的。

注意在调用 LdrPEStartup()时的参数 ImageBase 是目标 EXE 映像在用户空间的位置，而不是 ntdll.dll 在用户空间的位置。后者的位置并没有作为参数传下去，但是在模块队列中已经有了它的 LDR_MODULE 数据结构。

[__true_LdrInitializeThunk > LdrPEStartup()]

```
PEPFUNC LdrPEStartup (PVOID    ImageBase, HANDLE SectionHandle,
                      PLDR_MODULE* Module, PWSTR FullDosName)
{
    . . . . . .
    DosHeader = (PIMAGE_DOS_HEADER) ImageBase;
    NTHeaders = (PIMAGE_NT_HEADERS) (ImageBase + DosHeader->e_lfanew);

    /*
```

```
  * If the base address is different from the
  * one the DLL is actually loaded, perform any
  * relocation.
  */
if (ImageBase != (PVOID) NTHeaders->OptionalHeader.ImageBase)
   {
      DPRINT("LDR: Performing relocations\n");
      Status = LdrPerformRelocations(NTHeaders, ImageBase);
      . . . . . .
   }

if (Module != NULL)
   {
      *Module = LdrAddModuleEntry(ImageBase, NTHeaders, FullDosName);
      (*Module)->SectionHandle = SectionHandle;
   }
else
   {
      Module = &tmpModule;
      Status = LdrFindEntryForAddress(ImageBase, Module);
      . . . . . .
   }

. . . . . .

/*
  * If the DLL's imports symbols from other
  * modules, fixup the imported calls entry points.
  */
DPRINT("About to fixup imports\n");
Status = LdrFixupImports(NULL, *Module);
if (!NT_SUCCESS(Status))
   {
      DPRINT1("LdrFixupImports() failed for %wZ\n", &(*Module)->BaseDllName);
      return NULL;
   }
DPRINT("Fixup done\n");

. . . . . .
Status = LdrpInitializeTlsForProccess();
. . . . . .

/*
  * Compute the DLL's entry point's address.
```

```
    */
    ......
    if (NTHeaders->OptionalHeader.AddressOfEntryPoint != 0)
      {
          EntryPoint = (PEPFUNC) (ImageBase
                                  + NTHeaders->OptionalHeader.AddressOfEntryPoint);
      }
    DPRINT("LdrPEStartup() = %x\n",EntryPoint);
    return EntryPoint;
}
```

注意调用 LdrFixupImports()的参数之一 Module 是 LDR_MODULE 数据结构的间接指针(PLDR_MODULE 本身就已经是指针)，它最终指向目标 EXE 映像的 LDR_MODULE 数据结构，因此 LdrFixupImports()是以 EXE 映像为起点的。下面读者就会看到，LdrFixupImports()是个递归的过程，处理的是一棵动态连接的调用树(实际上不是树，因为多个父节点可以通向同一个子节点，但为叙述方便我们暂且称之为树)，而 EXE 映像就是这棵树的根。这棵树中除根节点以外的所有节点都是 DLL，但是“叶节点”通常只有一个(不过并无限制)，那就是 ntdll.dll。

PE 映像的 NtHeader 中有个指针，指向一个 OptionalHeader。说是“Optional”，实际上却是关键性的。在 OptionalHeader 中有个字段 ImageBase，是具体映像建议、或者说希望被装入的地址，我们不妨称之为“愿望地址”。在装入一个映像时，只要相应的区间(取决于它的愿望地址和大小)，就总是会遂其所愿。但是如果与已经被占用的区间相冲突，就只好易地安置。下面是一些常见.exe 映像和 DLL 映像的愿望地址：

notepad.exe	0x01000000	00013000
cmd.exe	0x4ad00000	0005e000
csrss.exe	0x4a680000	00004000
winword.exe	0x30000000	00836000
powerpnt.exe	0x30000000	00420000
excel.exe	0x30000000	006d6000
outlook.exe	0x30000000	0000e000
iexplore.exe	0x00400000	00019000
WinDVD.exe	0x00400000	00031000
ntdll.dll	0x77f50000	000a7000
kernel32.dll	0x77e60000	000e7ed3
gdi32.dll	0x77c70000	00040000
user32.dll	0x77d40000	0008c000
advapi32.dll	0x77dd0000	0008d000
csrsrv.dll	0x75b40000	0000a000
crtdll.dll	0x73d90000	00027000
mfc40.dll	0x61a90000	000e7000
ole32.dll	0x771b0000	00121000

可见，无论是 EXE 映像还是 DLL 映像，都没有一个统一的愿望地址。在 Windows 的

空间结构中，用户空间与系统空间的分界线是 0x80000000、即 2GB 处，分界线以下为用户空间。如前所述，PEB 的位置是 0x7FFDF000(即分界线以下 68KB 处)，PEB 以下是 TEB，每个 TEB 占 4KB、即一个页面。系统 DLL ntdll.dll 的愿望地址是 0x77f50000，离分界线的距离约 128MB，而映像大小是 0xa7000，小于 1MB。Winword.exe 的愿望地址是 0x30000000，离分界线的距离大于 1GB，映像大小是 0x836000，稍大于 8MB。注意 EXE 映像的大小未必反映应用软件的实际大小，因为许多应用软件都是由 EXE 和 DLL 共同实现的。例如，csrss.exe 的映像大小是 0x4000，即 16KB，但是它的许多函数都在 csrsrv.dll 中，其大小倒反而是 0xa000、即 40KB。

那么映像的愿望地址到底是怎么一回事，有着什么物理的或者逻辑的意义呢？我们知道，软件在编译以后有个连接的过程，即为函数的调用者落实被调用函数的入口地址、为全局变量(按绝对地址)的引用(读/写)者落实变量地址的过程。连接有静态和动态两种，静态连接是在“制造”软件时进行的，而动态连接则是在使用软件时进行的。尽管 EXE 模块和 DLL 模块之间的连接是动态连接，但是 EXE 或 DLL 模块内部的连接却是静态连接。既是静态连接，就必须为模块的映像提供一个假定的起点地址。如果以此假定地址为基础进行连接以后就不可变更，使用时必须装入到这个地址上，那么这个地址就是固定的“指定地址”了。早期的静态连接往往都是使用指定地址的。但是，如果允许按假定地址连接的映像在实际使用时进行“重定位”，那么这假定地址就是可浮动的“愿望地址”了。可“重定位”的静态连接当然比固定的静态连接灵活。事实上，要是没有可“重定位”的静态连接技术，DLL 的使用根本就不现实，因为根本就不可能事先为所有可能的 DLL 划定它们的装入位置和大小。至于可“重定位”静态连接的实现，则一般都是采用间接寻址，通过指针来实现。

一般而言，目标 EXE 映像的装入(映射)地址就是其愿望地址，因为一般而言这是最早映射的，没有理由要改变其位置。但是万一改变了就要通过 LdrPerformRelocations()实行重定位，这是由 LdrPerformRelocations()实现的：

[__true_LdrInitializeThunk > LdrPEStartup() > LdrPerformRelocations()]

```
static NTSTATUS
LdrPerformRelocations(PIMAGE_NT_HEADERS NTHeaders, PVOID ImageBase)
{
  . . . . . .

  RelocationDDir =
&NTHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
  . . . . . .

  ProtectSize = PAGE_SIZE;
  Delta = (ULONG_PTR)ImageBase - NTHeaders->OptionalHeader.ImageBase;
  RelocationDir = (PIMAGE_BASE_RELOCATION)((ULONG_PTR)ImageBase +
                    RelocationDDir->VirtualAddress);
  RelocationEnd = (PIMAGE_BASE_RELOCATION)((ULONG_PTR)ImageBase +
                    RelocationDDir->VirtualAddress + RelocationDDir->Size);

  while (RelocationDir < RelocationEnd && RelocationDir->SizeOfBlock > 0)
    {
```

```
        Count = (RelocationDir->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) /
                sizeof(USHORT);
        Page = ImageBase + RelocationDir->VirtualAddress;
        TypeOffset = (PUSHORT)(RelocationDir + 1);

        /* Unprotect the page(s) we're about to relocate. */
        ProtectPage = Page;
        Status = NtProtectVirtualMemory(NtCurrentProcess(), &ProtectPage,
                                        &ProtectSize, PAGE_READWRITE, &OldProtect);
        . . . . . .

        if (RelocationDir->VirtualAddress + PAGE_SIZE <
            NTHeaders->OptionalHeader.SizeOfImage)
          {
            ProtectPage2 = ProtectPage + PAGE_SIZE;
            Status = NtProtectVirtualMemory(NtCurrentProcess(), &ProtectPage2,
                                        &ProtectSize, PAGE_READWRITE, &OldProtect2);
            . . . . . .
          }
        else
          {
            ProtectPage2 = NULL;
          }

        RelocationDir = LdrProcessRelocationBlock(Page, Count, TypeOffset, Delta);
        . . . . . .

        /* Restore old page protection. */
        NtProtectVirtualMemory(NtCurrentProcess(),&ProtectPage,
                                        &ProtectSize, OldProtect, &OldProtect);

        if (ProtectPage2 != NULL)
          {
            NtProtectVirtualMemory(NtCurrentProcess(), &ProtectPage2,
                                           &ProtectSize, OldProtect2, &OldProtect2);
          }
      }

  return STATUS_SUCCESS;
}
```

PE 映像的 OptionalHeader 中有个大小为 16 的数组 DataDirectory[]，其元素都是“数据目录”、即 IMAGE_DATA_DIRECTORY 数据结构：

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
} IMAGE_DATA_DIRECTORY,*PIMAGE_DATA_DIRECTORY;
```

显然，数组中的每一个元素都说明了一个数据目录(在映像中)的位置及其大小(以 32 位长字为单位)，用于各种不同的目的。其中之一(下标为 5)就是“重定位目录”，这是一个 IMAGE_BASE_RELOCATION 结构数组。

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD VirtualAddress;
    DWORD SizeOfBlock;
} IMAGE_BASE_RELOCATION,*PIMAGE_BASE_RELOCATION;
```

每个 IMAGE_BASE_RELOCATION 数据结构代表着一个“重定位块”，每个重定位块的(容器)大小是两个页面(8KB)，而 SizeOfBlock 则说明具体重定位块的实际大小。这实际的大小中包括了这 IMAGE_BASE_RELOCATION 数据结构本身。

于是，所谓重定位，就是计算出实际装入地址与建议装入地址间的位移 Delta，然后调整每个重定位块中的每一个重定位项、即指针，具体就是在指针上加 Delta。而映像中使用的所有绝对地址(包括函数入口、全局量数据的位置)实际上用的都是间接寻址，每个这样的地址都有个指针存在于某个重定位块中。可见，这与 ELF 格式中的 PLT 实质上是一样的。具体的指针调整是由 LdrProcessRelocationBlock() 完成的，此前和此后的 NtProtectVirtualMemory()只是为了先去除这些指针所在页面的写保护，而事后加以恢复。

完成了可能需要的 EXE 映像重定位以后，下一个主要的操作就是 LdrFixupImports()了。实际上这才是关键所在，它所处理的就是当前模块所需 DLL 模块的装入(如果尚未装入的话)和连接。如前所述，这个函数递归地施行于所有的模块，直至最底层的“叶节点”ntdll.dll 为止。

最后，LdrPEStartup()返回的是根模块即 EXE 映像的程序入口。相比之下，LdrFixupImports()则并不返回各个模块的程序入口，各 DLL 的程序入口纪录在它们的 LDR_MODULE 数据结构中(但是 ntdll.dll 的入口已经不再需要，因为现在已经在这个模块里面了)，借助 InInitializationOrderModuleList 队列就可依次调用所有 DLL 的初始化函数。

下面我们看 LdrFixupImports()的代码：

```
[__true_LdrInitializeThunk > LdrPEStartup() > LdrFixupImports()]

static NTSTATUS
LdrFixupImports(IN PWSTR SearchPath OPTIONAL, IN PLDR_MODULE Module)
{
  ......
  /*
   * Process each import module.
   */
  ImportModuleDirectory = (PIMAGE_IMPORT_MODULE_DIRECTORY)
                          RtlImageDirectoryEntryToData(Module->BaseAddress, TRUE,
```

```
                        IMAGE_DIRECTORY_ENTRY_IMPORT, NULL);

BoundImportDescriptor = (PIMAGE_BOUND_IMPORT_DESCRIPTOR)
                        RtlImageDirectoryEntryToData(Module->BaseAddress, TRUE,
                        IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT, NULL);

if (BoundImportDescriptor != NULL && ImportModuleDirectory == NULL)
{
    DPRINT1("%wZ has only a bound import directory\n", &Module->BaseDllName);
    return STATUS_UNSUCCESSFUL;
}

if (BoundImportDescriptor)      /*  绑定引入  */
{
  DPRINT("BoundImportDescriptor %x\n", BoundImportDescriptor);

  BoundImportDescriptorCurrent = BoundImportDescriptor;
  while (BoundImportDescriptorCurrent->OffsetModuleName)
  {
    ImportedName = (PCHAR)BoundImportDescriptor +
                                    BoundImportDescriptorCurrent->OffsetModuleName;
    TRACE_LDR("%wZ bound to %s\n", &Module->BaseDllName, ImportedName);
    Status = LdrpGetOrLoadModule(SearchPath, ImportedName,
                                            &ImportedModule, TRUE);
    . . . . . .
    if (ImportedModule->TimeDateStamp !=
                    BoundImportDescriptorCurrent->TimeDateStamp)
    {
      TRACE_LDR("%wZ has stale binding to %wZ\n",
                        &Module->BaseDllName, &ImportedModule->BaseDllName);
      Status = LdrpProcessImportDirectory(Module, ImportedModule, ImportedName);
      . . . . . .
    }
    else
    {
      BOOLEAN WrongForwarder;
      WrongForwarder = FALSE;
      . . . . . .
      if (BoundImportDescriptorCurrent->NumberOfModuleForwarderRefs)
      {
        PIMAGE_BOUND_FORWARDER_REF BoundForwarderRef;
        ULONG i;
        PLDR_MODULE ForwarderModule;
        PCHAR ForwarderName;
```

```
 BoundForwarderRef = (PIMAGE_BOUND_FORWARDER_REF)
                  (BoundImportDescriptorCurrent + 1);
 for (i = 0;
       i < BoundImportDescriptorCurrent->NumberOfModuleForwarderRefs;
       i++, BoundForwarderRef++)
 {
    ForwarderName = (PCHAR)BoundImportDescriptor +
                              BoundForwarderRef->OffsetModuleName;
    TRACE_LDR("%wZ bound to %s via forwardes from %s\n",
                     &Module->BaseDllName, ForwarderName, ImportedName);
    Status = LdrpGetOrLoadModule(SearchPath,
                    ForwarderName, &ForwarderModule, TRUE);
    . . . . . .
    if (ForwarderModule->TimeDateStamp !=
                         BoundForwarderRef->TimeDateStamp    ||
                     ForwarderModule->Flags & IMAGE_NOT_AT_BASE)
    {
                     TRACE_LDR("%wZ has stale binding to %s\n",
                              &Module->BaseDllName, ForwarderName);
                     WrongForwarder = TRUE;
    }
     else
     {
                     TRACE_LDR("%wZ has correct binding to %s\n",
                              &Module->BaseDllName, ForwarderName);
                  }
   } //end for
}
if (WrongForwarder ||
           ImportedModule->Flags & IMAGE_NOT_AT_BASE)
{
   Status = LdrpProcessImportDirectory(Module, ImportedModule,
                                                    ImportedName);
   . . . . . .
}
else if (ImportedModule->Flags & IMAGE_NOT_AT_BASE)
{
           TRACE_LDR("Adjust imports for %s from %wZ\n",
                        ImportedName, &Module->BaseDllName);
   Status = LdrpAdjustImportDirectory(Module,
                                  ImportedModule, ImportedName);
    . . . . . .
}
```

```
            else if (WrongForwarder)
            {
               . . . . . .
               Status = LdrpProcessImportDirectory(Module,
                                                   ImportedModule, ImportedName);
               . . . . . .
             }
             else
             {
                /* nothing to do */
             }
          }
          BoundImportDescriptorCurrent +=
                        BoundImportDescriptorCurrent->NumberOfModuleForwarderRefs + 1;
   } //end while (BoundImportDescriptorCurrent->OffsetModuleName)
}
else if (ImportModuleDirectory)       /*   普通引入   */
{
       DPRINT("ImportModuleDirectory %x\n", ImportModuleDirectory);

       ImportModuleDirectoryCurrent = ImportModuleDirectory;
       while (ImportModuleDirectoryCurrent->dwRVAModuleName)
       {
          ImportedName = (PCHAR)Module->BaseAddress +
                                ImportModuleDirectoryCurrent->dwRVAModuleName;
          TRACE_LDR("%wZ imports functions from %s\n",
                                &Module->BaseDllName, ImportedName);

          Status = LdrpGetOrLoadModule(SearchPath, ImportedName,
                                               &ImportedModule, TRUE);
          . . . . . .
          Status = LdrpProcessImportDirectoryEntry(Module,
                          ImportedModule, ImportModuleDirectoryCurrent);
          . . . . . .
          ImportModuleDirectoryCurrent++;
   }   //end while (ImportModuleDirectoryCurrent->dwRVAModuleName)
} //end if (ImportModuleDirectory)

if (TlsDirectory && TlsSize > 0)
{
   LdrpAcquireTlsSlot(Module, TlsSize, FALSE);
}

return STATUS_SUCCESS;
```

}

前面讲过，映像头部的 OptionalHeader 中有个数组 DataDirectory[]，其中之一是重定位目录。除此之外，数组中还有“(普通)引入(import)”、“绑定引入(bound import)”、“引出(export)”、以及其它多种目录，但是我们在这里只关心“引入”和“绑定引入”。这两个目录都是用于库函数的引入，但是作用不同，目录项的数据结构也不同。

先看普通的“引入”，其数据结构为：

```
typedef struct _IMAGE_IMPORT_MODULE_DIRECTORY
{
   DWORD       dwRVAFunctionNameList;
   DWORD       dwUseless1;
   DWORD       dwUseless2;
   DWORD       dwRVAModuleName;
   DWORD       dwRVAFunctionAddressList;
}
IMAGE_IMPORT_MODULE_DIRECTORY,*PIMAGE_IMPORT_MODULE_DIRECTORY;
```

这里的 RVA 是“相对虚拟地址(Relative Virtual Address)”的缩写，实际上就是在映像内部的位移。每个引入目录项代表着一个被引入模块，其模块名、即文件名在 dwRVAModuleName 所指的地方。需要从同一个被引入模块引入的函数通常有很多个，dwRVAFunctionNameList 指向一个字符串数组，数组中的每一个字符串都是一个函数名；与此相对应，dwRVAFunctionAddressList 则指向一个指针数组。这两个数组是平行的，同一个函数在两个数组中具有相同的下标。可想而知，从一个被引入模块中引入一个函数的过程大体上就是：根据函数名在被引入模块的引出目录中搜索，找到目标函数以后就把它实际装入后的入口地址填写到指针数组中的相应位置上。但是，这个过程可能是个开销相当大、速度比较慢的过程。为此，又发展起一种称为“绑定”的优化。

所谓绑定，就是在软件的“制造”过程中先对使用时的动态连接来一次预演，预演时假定所有的 DLL 都被装入到它们的愿望地址上，然后把预演中得到的被引入函数的地址直接记录在引入者模块中相应引入目录下的指针数组中。这样，使用软件时的动态连接就变得很简单快捷，因为实际上已经事先连接好了。其实“绑定引入”和静态连接并无实质的不同，只不过是软件的“静态连接，分块发行”，而不是“静态连接，整块发行”。但是，既然是“静态连接，分块提供”，实际使用时不再比对函数名，各模块的版本配套就成为一个问题，因为万一使用的某个 DLL 不是当初绑定时的版本，而且其引出目录又发生了变化，就有可能引起混乱。为此，PE 格式增加了一种“绑定引入”目录，其目录项中加上了时戳字段。这样，只要引入者的“绑定引入”目录项和被引入者具有相同的时戳，就可以认定它们是当初绑定的“原配”。“绑定引入”目录项的数据结构如下：

```
typedef struct _IMAGE_BOUND_IMPORT_DESCRIPTOR {
    DWORD TimeDateStamp;
    WORD OffsetModuleName;
    WORD NumberOfModuleForwarderRefs;
}
IMAGE_BOUND_IMPORT_DESCRIPTOR,*PIMAGE_BOUND_IMPORT_DESCRIPTOR;
```

这里的 TimeDateStamp 就是时戳字段，OffsetModuleName 指向被引入模块的模块名；另一个字段 NumberOfModuleForwarderRefs 用于“转引”，下面还会讲到。显然，一个“绑定引入”目录项、即一个 IMAGE_BOUND_IMPORT_DESCRIPTOR 数据结构，只是针对着一个具体的被引入模块。

但是，“绑定引入”毕竟不是很可靠的，万一发现版本不符就不能使用原先的绑定了。所以“绑定引入”不能单独存在，而必须有普通引入作为后备，这样在发生问题时就可以退到普通引入。

代码中首先通过 RtlImageDirectoryEntryToData()分别从映像头部获取指向“引入”目录和“绑定引入”目录的指针。后者是前者的优化。一个映像可以没有“引入”目录，但不能没有“引入”目录却有“绑定引入”目录。每一个 IMAGE_BOUND_IMPORT_DESCRIPTOR 数据结构或 IMAGE_IMPORT_MODULE_DIRECTORY 数据结构都代表着一个需要引入的 DLL，而一个“目录”就是一个这样的结构数组。然后，如果有“绑定引入”目录就优先按它来处理引入，否则就按普通的“引入”目录处理引入。

先看有“绑定引入”目录存在时的操作。

绑定引入是普通引入的优化，但是绑定引入有个引入者和被引入者之间的时戳 TimeDateStamp 是否相符的问题。如果不符就不能按“绑定引入”目录处理引入，而只好退而求其次，改成按普通“引入”目录处理引入。另一方面，所谓“绑定”是指当被引入模块装入在预定位置上时的地址绑定，如果被引入模块的装入位置变了，就得对原先所绑定的地址作相应的调整、即“重定位”。

还有个问题就是对于“转引(forward)”的处理。所谓“转引”，是指这样的情况：假定 A 引入 B，而 B 又引入 C；表面上 A 要引入 B 中的某个函数 f，但是这个函数实际上是由 C 提供的，B 只是转了一下手。如果 A 针对 B 的 IMAGE_BOUND_IMPORT_DESCRIPTOR 数据结构中的字段 NumberOfModuleForwarderRefs 为非 0，就说明 B 存在着转引，此时紧随在 IMAGE_BOUND_IMPORT_DESCRIPTOR 数据结构的后面有着相应数量的 IMAGE_BOUND_FORWARDER_REF 数据结构。

```
typedef struct _IMAGE_BOUND_FORWARDER_REF {
    DWORD TimeDateStamp;
    WORD OffsetModuleName;
    WORD Reserved;
} IMAGE_BOUND_FORWARDER_REF,*PIMAGE_BOUND_FORWARDER_REF;
```

这里的 OffsetModuleName 指向被转引模块的模块名(在映像中的位移)。当然，被转引的模块都需要被装入。

明白了这些，前面 if (BoundImportDescriptor){}里面的代码就不难理解了，那就是对于“绑定引入”目录中的每一个目录项实施下列的操作：

- 先通过 LdrpGetOrLoadModule()找到或装入(映射)被引入模块的映像。首先当然是在模块队列中寻找，找不到就从)被引入模块的磁盘文件装入。
- 检查双方的时戳 TimeDateStamp 是否相符，如果不符就退而求其次，通过 LdrpProcessImportDirectory()处理引入(见下)。当然，那样一来效率就要降低了。
- 假定时戳相符，如果目录项中的 NumberOfModuleForwarderRefs 非 0，就要对每个需要被转引的模块执行 LdrpGetOrLoadModule()，保证它们的映像已被装入。

- 如果某个被转引模块的时戳与转引目录项中的纪录不符，或者其映像装入地址与预定的不符，那么就又得退而求其次了。代码中先把 WrongForwarder 设成 TRUE，然后据此调用 LdrpProcessImportDirectory()。
- 如果转引没有问题，而只是某个直接的被引入模块的装入地址与绑定的不符，那就只要对引入者模块中相应引入目录下的函数指针作出调整即可，这是由 LdrpAdjustImportDirectory()完成的。

下面是 LdrpAdjustImportDirectory()的代码，读者可以自行阅读。

```
[__true_LdrInitializeThunk > LdrPEStartup() > LdrFixupImports()
  > LdrpAdjustImportDirectory()]

static NTSTATUS
LdrpAdjustImportDirectory(PLDR_MODULE Module,
                          PLDR_MODULE ImportedModule, PCHAR ImportedName)
{
  . . . . . .
  ImportModuleDirectory = (PIMAGE_IMPORT_MODULE_DIRECTORY)
                    RtlImageDirectoryEntryToData(Module->BaseAddress,
                          TRUE, IMAGE_DIRECTORY_ENTRY_IMPORT, NULL);
  . . . . . .

  while (ImportModuleDirectory->dwRVAModuleName)
  {
    Name = (PCHAR)Module->BaseAddress + ImportModuleDirectory->dwRVAModuleName;
    if (0 == _stricmp(Name, (PCHAR)ImportedName))
    {
      /* Get the import address list. */
      ImportAddressList = (PVOID *)
          (Module->BaseAddress + ImportModuleDirectory->dwRVAFunctionAddressList);
      /* Get the list of functions to import. */
      if (ImportModuleDirectory->dwRVAFunctionNameList != 0)
      {
        FunctionNameList = (PULONG)
            (Module->BaseAddress + ImportModuleDirectory->dwRVAFunctionNameList);
      }
      else
      {
        FunctionNameList = (PULONG)
            (Module->BaseAddress + ImportModuleDirectory->dwRVAFunctionAddressList);
      }
      /* Get the size of IAT. */
      IATSize = 0;
      while (FunctionNameList[IATSize] != 0L)
      {
```

```
            IATSize++;
        }

        /* Unprotect the region we are about to write into. */
        IATBase = (PVOID)ImportAddressList;
        IATSize *= sizeof(PVOID*);
        Status = NtProtectVirtualMemory(NtCurrentProcess(),
                                        &IATBase,
                                        &IATSize,
                                        PAGE_READWRITE,
                                        &OldProtect);
        . . . . . .

        NTHeaders = RtlImageNtHeader (ImportedModule->BaseAddress);
        Start = (PVOID)NTHeaders->OptionalHeader.ImageBase;
        End = Start + ImportedModule->ResidentSize;
        Offset = ImportedModule->BaseAddress - Start;

        /* Walk through function list and fixup addresses. */
        while (*FunctionNameList != 0L)
        {
          if (*ImportAddressList >= Start && *ImportAddressList < End)
          {
            (*ImportAddressList) += Offset;
          }
          ImportAddressList++;
          FunctionNameList++;
        }

        /* Protect the region we are about to write into. */
        Status = NtProtectVirtualMemory(NtCurrentProcess(),
                                        &IATBase,
                                        &IATSize,
                                        OldProtect,
                                        &OldProtect);
        . . . . . .
      }
      ImportModuleDirectory++;
    }   //end while
    return STATUS_SUCCESS;
}
```

由此可见，对于“绑定引入”的处理基本上只是验证一下，只要确认被引入模块已被装入、并且装入在预定的位置上就行了。即使有个别模块(DLL)没能装入在预定的位置上，毕

竟用 LdrpAdjustImportDirectory()调整一下也不是代价很大。所以“绑定引入”的效率确实是比较高的，但是，如果时戳不符，或者被转引的模块未能装入在预定的位置上，那就只好退下来改用 LdrpProcessImportDirectory()了。

[__true_LdrInitializeThunk > LdrPEStartup() > LdrFixupImports()
> LdrpProcessImportDirectory()]

```
static NTSTATUS
LdrpProcessImportDirectory(
    PLDR_MODULE Module,
    PLDR_MODULE ImportedModule,
    PCHAR ImportedName)
{
  . . . . . .
  ImportModuleDirectory = (PIMAGE_IMPORT_MODULE_DIRECTORY)
                          RtlImageDirectoryEntryToData(Module->BaseAddress,
                              TRUE, IMAGE_DIRECTORY_ENTRY_IMPORT,
                              NULL);
  . . . . . .
  while (ImportModuleDirectory->dwRVAModuleName)
  {
    Name = (PCHAR)Module->BaseAddress +
                        ImportModuleDirectory->dwRVAModuleName;
    if (0 == _stricmp(Name, ImportedName))
    {
      Status = LdrpProcessImportDirectoryEntry(Module,
                                               ImportedModule,
                                               ImportModuleDirectory);
      . . . . . .
    }
    ImportModuleDirectory++;
  }
  return STATUS_SUCCESS;
}
```

先看调用参数。第一个参数 Module 当然是指向当前模块的 LDR_MODULE 数据结构的指针；第二个参数 ImportedModule 同样是指针，但是指向被引入模块的 LDR_MODULE 数据结构；第三个参数 ImportedName 则是字符串指针，指向被引入模块的文件名。这段程序很简单，就是搜索引入者模块的普通引入目录，找到了要求引入给定被引入模块的那个目录项，就调用 LdrpProcessImportDirectoryEntry()。注意前面在 LdrFixupImports()中是按绑定引入目录项、而不是普通引入目录项在处理的，而 LdrpProcessImportDirectoryEntry()要求使用普通引入目录项，所以才需要经由 LdrpProcessImportDirectory()先找到相应的普通引入目录项。

事实上，LdrpProcessImportDirectoryEntry()所实现的才是本来意义上的动态连接。

[__true_LdrInitializeThunk > LdrPEStartup() > LdrFixupImports()
> LdrpProcessImportDirectory() > LdrpProcessImportDirectoryEntry()]

```
static NTSTATUS
LdrpProcessImportDirectoryEntry(PLDR_MODULE Module,
                                PLDR_MODULE ImportedModule,
                                PIMAGE_IMPORT_MODULE_DIRECTORY ImportModuleDirectory)
{
   . . . . . .

   /* Get the import address list. */
   ImportAddressList = (PVOID *)(Module->BaseAddress +
                                 ImportModuleDirectory->dwRVAFunctionAddressList);

   /* Get the list of functions to import. */
   if (ImportModuleDirectory->dwRVAFunctionNameList != 0)
   {
      FunctionNameList = (PULONG) (Module->BaseAddress +
                                   ImportModuleDirectory->dwRVAFunctionNameList);
   }
   else
   {
      FunctionNameList = (PULONG)(Module->BaseAddress +
                                   ImportModuleDirectory->dwRVAFunctionAddressList);
   }

   /* Get the size of IAT. */
   IATSize = 0;
   while (FunctionNameList[IATSize] != 0L)
   {
      IATSize++;
   }

   /* Unprotect the region we are about to write into. */
   IATBase = (PVOID)ImportAddressList;
   IATSize *= sizeof(PVOID*);
   Status = NtProtectVirtualMemory(NtCurrentProcess(),
                                   &IATBase,
                                   &IATSize,
                                   PAGE_READWRITE,
                                   &OldProtect);
   . . . . . .
```

```
  /* Walk through function list and fixup addresses. */
  while (*FunctionNameList != 0L)
  {
    if ((*FunctionNameList) & 0x80000000)
    {
      Ordinal = (*FunctionNameList) & 0x7fffffff;
      *ImportAddressList =
                    LdrGetExportByOrdinal(ImportedModule->BaseAddress, Ordinal);
      . . . . . .
    }
    else
    {
      IMAGE_IMPORT_BY_NAME *pe_name;
      pe_name = RVA(Module->BaseAddress, *FunctionNameList);
      *ImportAddressList = LdrGetExportByName(ImportedModule->BaseAddress,
                                              pe_name->Name, pe_name->Hint);
      . . . . . .
    }
    ImportAddressList++;
    FunctionNameList++;
  }

  /* Protect the region we are about to write into. */
  Status = NtProtectVirtualMemory(NtCurrentProcess(),
                                  &IATBase,
                                  &IATSize,
                                  OldProtect,
                                  &OldProtect);
    . . . . . .
  return STATUS_SUCCESS;
}
```

首先根据目录项中的两个位移量取得分别指向函数名字符串数组和函数指针数组的指针。这两个数组是平行的，一个函数的的函数名在字符串数组中的下标是什么，它的函数入口在指针数组中的下标也就是什么。然后对字符串数组中的元素计数，得到该数组的大小IATSize。显然，函数指针数组的大小也是 IATSize。这里 IAT 是“引入地址表(Imported Address Table)”的缩写，其实就是函数指针数组。这个数组在映像内部，其所在的页面在装入映像时已被加上写保护，而下面要做的事正是要改变这些指针的值，所以先要通过NtProtectVirtualMemory()把这些页面的访问模式改成可读可写。做完这些准备之后，下面就是连接的过程了，那就是根据需要把被引入模块所引出的函数入口(地址)填写到引入者模块的 IAT 中。

与当前模块中的两个数组相对应，在被引入模块的“引出”目录中也有两个数组，说明本模块引出函数的名称和入口地址(在映像中的位移)。当然，这两个数组也是平行的。

要获取被引入模块中的函数入口有两种方法，即按序号(Ordinal)引入和按函数名引入。

虽然名曰函数名数组，实际上数组中的元素既可以是个函数名指针(实际上是结构指针，见后)，也可以是个非 0 的序号。如果数组元素的内容是(32 位)序号就把最高位设成 1，使序号(在形式上)都大于 0x80000000。因为任何模块都不可能引出这么多的函数，所以这样做是安全的。另一方面，当数组元素的内容是函数名指针时又必须使最高位为 0，这也不会有问题，因为 Windows 的用户空间本来就在 0x80000000、即 2GB 边界以下。这里，按函数名获取是很好理解的，需要引入的模块以函数名的方式说明要引入那一些函数，而被引入模块也以函数名的方式说明本模块提供(引出)了哪一些函数，通过字符串比对就可以实现配对，并从而取得被引入模块提供的相应函数指针。那么“序号”又是什么呢？其实也很简单，序号本质上就是目标函数在引出数组中的下标，只不过下标是从 0 开始的，而序号是从一个“基序号”开始的(因为必须是非 0)，所以序号是下标加基序号(一般是 1)。引出目录项中有个字段 Base，那就是基序号。显然，按序号获取函数指针的效率更高。

回到上面的代码。根据当前模块引入目录中字符串数组各元素的内容，就可以确定所要求的是按序号引入还是按函数名引入，从而分别调用 LdrGetExportByOrdinal()和 LdrGetExportByName()。这两个函数都返回目标函数在本进程用户空间中的入口地址，把它填写入当前模块引入目录函数指针数组中的相应元素，就完成了一个函数的连接。当然，同样的操作要循环实施于当前模块需要从给定模块引入的所有函数，并且(在上一层)循环实施于所有的被引入模块。

完成了对一个被引入模块的连接之后，又调用 NtProtectVirtualMemory()恢复当前模块中给定目录项内函数指针数组所在页面的保护。

按序号引入和按函数名引入所使用的上述两个函数基本上是一样的，只是 LdrGetExportByOrdinal()略为简单一些(无需字符串比对)，我们就来看这个函数的代码。

```
[__true_LdrInitializeThunk > LdrPEStartup() > LdrFixupImports()
  > LdrpProcessImportDirectory() > LdrpProcessImportDirectoryEntry()
  > LdrGetExportByOrdinal()]

static PVOID
LdrGetExportByOrdinal (PVOID BaseAddress, ULONG Ordinal)
{
  . . . . . .
  ExportDir = (PIMAGE_EXPORT_DIRECTORY)
                    RtlImageDirectoryEntryToData (BaseAddress,
                                                  TRUE,
                                                  IMAGE_DIRECTORY_ENTRY_EXPORT,
                                                  &ExportDirSize);
  ExFunctions = (PDWORD *)RVA(BaseAddress, ExportDir->AddressOfFunctions);
  Function = (0 != ExFunctions[Ordinal - ExportDir->Base]
                    ? RVA(BaseAddress, ExFunctions[Ordinal - ExportDir->Base] )
                    : NULL);
  if (((ULONG)Function >= (ULONG)ExportDir) &&
            ((ULONG)Function < (ULONG)ExportDir + (ULONG)ExportDirSize))
  {
    Function = LdrFixupForward((PCHAR)Function);
  }
```

```
    return Function;
}
```

这里的 RVA()是个宏操作，用来根据映像内位移和映像起点计算装入后的虚拟地址，定义为：

```
#define RVA(m, b) ((ULONG)b + m
```

代码的逻辑很简单，先通过 RtlImageDirectoryEntryToData()从被引入模块的映像获取它的引出目录 ExportDir。再根据 ExportDir->AddressOfFunctions、即函数指针数组在映像中的位移、和映像的起点算出它的地址 ExFunctions。然后，从序号中减去被引入模块使用的基序号，就得到了实际的下标。至于用这下标取得目标函数的入口位移，再换算成虚拟地址、即函数指针，那就是直截了当的事了。

由被引入模块实现并引出的函数当然不会落在引出目录内部，如果目标函数指针落在引出目录内部，那就是特殊的情况了，实际上说明这是个“过路”的转引函数，其实现还在另一个模块中，而此时的“函数指针”其实是个字符串指针，指向被转引模块的模块名。转引函数的连接则需要通过 LdrFixupForward()作进一步的处理。

```
[__true_LdrInitializeThunk > LdrPEStartup() > LdrFixupImports()
   > LdrpProcessImportDirectory() > LdrpProcessImportDirectoryEntry()
   > LdrGetExportByOrdinal() > LdrFixupForward()]

static PVOID
LdrFixupForward(PCHAR ForwardName)
{
   . . . . . .
   strcpy(NameBuffer, ForwardName);
   p = strchr(NameBuffer, '.');
   if (p != NULL)
   {
      *p = 0;
      DPRINT("Dll: %s    Function: %s\n", NameBuffer, p+1);
      RtlCreateUnicodeStringFromAsciiz (&DllName, NameBuffer);
      Status = LdrFindEntryForName (&DllName, &Module, FALSE);
      /* FIXME:
      * The caller (or the image) is responsible for loading of the dll, where
        * the function is forwarded.
        */
      if (!NT_SUCCESS(Status))
      {
         Status = LdrLoadDll(NULL, LDRP_PROCESS_CREATION_TIME,
                             &DllName, &BaseAddress);
         if (NT_SUCCESS(Status))
         {
```

```
            Status = LdrFindEntryForName (&DllName, &Module, FALSE);
          }
        }
        RtlFreeUnicodeString (&DllName);
        if (!NT_SUCCESS(Status))
        {
          DPRINT1("LdrFixupForward: failed to load %s\n", NameBuffer);
          return NULL;
        }
        DPRINT("BaseAddress: %p\n", Module->BaseAddress);
        return LdrGetExportByName(Module->BaseAddress, (PUCHAR)(p+1), -1);
    }
    return NULL;
}
```

先通过 LdrFindEntryForName()在已装入模块的队列中寻找，取得指向被转引模块的 LDR_MODULE 数据结构指针，如果找不到就通过 LdrLoadDll()装入。然后再通过 LdrGetExportByName()获取目标函数的入口。

函数 LdrLoadDll()是 ntdll.dll 的一个引出函数。它不仅供 ntdll.dll 内部的 LdrFixupForward()调用，也供别的 DLL 或.exe 应用程序调用。例如 kernel32.dll 中的 LoadLibraryExA()和 LoadLibraryExW()就都要调用这个函数。

```
[__true_LdrInitializeThunk > LdrPEStartup() > LdrFixupImports()
   > LdrpProcessImportDirectory() > LdrpProcessImportDirectoryEntry()
   > LdrGetExportByOrdinal() > LdrFixupForward() > LdrLoadDll()]

NTSTATUS STDCALL
LdrLoadDll (IN PWSTR SearchPath OPTIONAL,    IN ULONG LoadFlags,
                IN PUNICODE_STRING Name,    OUT PVOID *BaseAddress OPTIONAL)
{
   . . . . . .
   *BaseAddress = NULL;
   Status = LdrpLoadModule(SearchPath, LoadFlags, Name, &Module, BaseAddress);
   if (NT_SUCCESS(Status) && 0 == (LoadFlags & LOAD_LIBRARY_AS_DATAFILE))
      {
        RtlEnterCriticalSection(NtCurrentPeb()->LoaderLock);
        Status = LdrpAttachProcess();
        RtlLeaveCriticalSection(NtCurrentPeb()->LoaderLock);
        if (NT_SUCCESS(Status))
        {
          *BaseAddress = Module->BaseAddress;
        }
     }
   return Status;
```

```
}
```

显然，这个函数是 LdrpLoadModule()和 LdrpAttachProcess()的组合。首先通过 LdrpLoadModule()装入目标 DLL 和以此为根的子树，然后通过 LdrpAttachProcess()调用这些 DLL 的初始化函数。这里的重点是 LdrpLoadModule()。

```
[__true_LdrInitializeThunk > LdrPEStartup() > LdrFixupImports()
  > LdrpProcessImportDirectory() > LdrpProcessImportDirectoryEntry()
  > LdrGetExportByOrdinal() > LdrFixupForward() > LdrLoadDll() > LdrpLoadModule()]

static NTSTATUS
LdrpLoadModule(IN PWSTR SearchPath OPTIONAL,    IN ULONG LoadFlags,
                    IN PUNICODE_STRING Name,    PLDR_MODULE *Module,
                    PVOID *BaseAddress OPTIONAL)
{
  . . . . . .

  if (Module == NULL)
  {
    Module = &tmpModule;
  }
  /* adjust the full dll name */
  LdrAdjustDllName(&AdjustedName, Name, FALSE);

  MappedAsDataFile = FALSE;
  /* Test if dll is already loaded */
  Status = LdrFindEntryForName(&AdjustedName, Module, TRUE);
  if (NT_SUCCESS(Status))
  {
    RtlFreeUnicodeString(&AdjustedName);
    if (NULL != BaseAddress)
    {
      *BaseAddress = (*Module)->BaseAddress;
    }
  }
  else
  {
    /* Open or create dll image section */
    Status = LdrpMapKnownDll(&AdjustedName, &FullDosName, &SectionHandle);
    if (!NT_SUCCESS(Status))
    {
      MappedAsDataFile = (0 != (LoadFlags & LOAD_LIBRARY_AS_DATAFILE));
      Status = LdrpMapDllImageFile(SearchPath, &AdjustedName, &FullDosName,
                                   MappedAsDataFile, &SectionHandle);
```

```
}
. . . . . .
RtlFreeUnicodeString(&AdjustedName);
/* Map the dll into the process */
ViewSize = 0;
ImageBase = 0;
Status = NtMapViewOfSection(SectionHandle, NtCurrentProcess(),&ImageBase,
                                    0, 0, NULL, &ViewSize, 0, MEM_COMMIT,
                                                        PAGE_READWRITE);
. . . . . .
if (NULL != BaseAddress)
{
   *BaseAddress = ImageBase;
}
/* Get and check the NT headers */
NtHeaders = RtlImageNtHeader(ImageBase);
. . . . . .
if (MappedAsDataFile)
{
   assert(NULL != BaseAddress);
   if (NULL != BaseAddress)
   {
      *BaseAddress = (PVOID) ((char *) *BaseAddress + 1);
   }
   *Module = NULL;
   RtlFreeUnicodeString(&FullDosName);
   NtClose(SectionHandle);
   return STATUS_SUCCESS;
}
/* If the base address is different from the one the DLL is actually loaded, perform any
   relocation. */
if (ImageBase != (PVOID) NtHeaders->OptionalHeader.ImageBase)
{
   Status = LdrPerformRelocations(NtHeaders, ImageBase);
   . . . . . .
}
*Module = LdrAddModuleEntry(ImageBase, NtHeaders, FullDosName.Buffer);
(*Module)->SectionHandle = SectionHandle;
if (ImageBase != (PVOID) NtHeaders->OptionalHeader.ImageBase)
{
   (*Module)->Flags |= IMAGE_NOT_AT_BASE;
}
if (NtHeaders->FileHeader.Characteristics & IMAGE_FILE_DLL)
{
```

```
        (*Module)->Flags |= IMAGE_DLL;
    }
    /* fixup the imported calls entry points */
    Status = LdrFixupImports(SearchPath, *Module);
    . . . . . .
    RtlEnterCriticalSection(NtCurrentPeb()->LoaderLock);
    InsertTailList(&NtCurrentPeb()->Ldr->InInitializationOrderModuleList,
                   &(*Module)->InInitializationOrderModuleList);
    RtlLeaveCriticalSection (NtCurrentPeb()->LoaderLock);
  }
  return STATUS_SUCCESS;
}
```

调用参数 Name 可以是文件名，也可能是模块名(例如 ntdll)，所以先通过 LdrAdjustDllName()加以必要的调整，如果是模块名就把它变成文件名。然后就通过 LdrFindEntryForName()在已装入模块队列中寻找，如果找到就万事大吉，只要通过调用参数 BaseAddress 返回该模块映像装入后的起点地址就行了。否则就得要劳累一点了：

- 先通过 LdrpMapKnownDll()在一个目录“\KnownDlls”下面寻找目标 DLL。这个目录完全是为避免四处查找、加快装入速度而设的符号连接。如果找到就为目标 DLL 映像建立一个共享内存区、即 Section。
- 如果不成功，就调用 LdrpMapDllImageFile()找到目标文件，将其打开并验证其确系 PE 格式映像文件，然后为其建立一个 Section。
- 至此，目标模块的映像文件已经打开并建立了一个 Section，只要将它映射到本进程的用户空间就可以了。于是通过系统调用 NtMapViewOfSection()将目标映像影射到用户空间。
- 在调用 LdrpLoadModule()时，可以通过参数 LoadFlags 中的标志位 LOAD_LIBRARY_AS_DATAFILE 说明目标模块是作为数据文件映射的，因而就不存在库函数的动态连接问题。如果是那样的话，那么到这里就完成了。否则就还得再接再厉。
- 如果映像的实际装入地址不同于其“愿望地址”、即 OptionalHeader.ImageBase，就通过 LdrPerformRelocations()进行“重定位”，即对映像中的绝对地址加以调整。我们在前面已经看过有关的代码。
- 通过 LdrAddModuleEntry()创建目标映像的 LDR_MODULE 数据结构，并把它挂入本进程的模块队列。
- 所装入的模块(DLL)本身又可能有引入要求，所以通过 LdrFixupImports()处理其引入。在我们这个情景中，这是对 LdrFixupImports()的递归调用(我们现在所处的位置正是从 LdrFixupImports()逐层调用下来的)，这样的递归调用一直要到被引入的模块本身不再要求引入(例如 ntdll.dll)时为止。
- 通过 InsertTailList()把所装入映像的 LDR_MODULE 数据结构挂入初始化队列。

回到前面 LdrFixupForward()的代码中。既然被转引模块已经装入，接着就可以从中获取目标函数的入口了。

不过目标函数在被转引模块中有可能又是一个转引函数。那也不要紧，再调用 LdrFixupForward()就是了。此时对 LdrFixupForward()的调用是递归调用，我们就不需要再往

下看了。这样，逐层转引就体现为对 LdrFixupForward()的递归调用，一直到目标函数的真正的实现/引出者为止。

回到 LdrFixupImports()的代码。理解了按“绑定引入”目录引入的代码，再看因为“绑定引入”目录不存在而只好按普通“引入”目录处理引入时的操作，就很简单了，这里只是直接调用 LdrpProcessImportDirectoryEntry()。

前面我们看的是按序号引入，这里再提一下按函数名引入。从原理上说按函数名引入毫无新奇之处，比之按序号引入只是多了字符串比对。然而，要是对于引入的每一个函数都要在被引入模块的引出函数表中顺序扫描比对，那个开销也真是太大了。所以，为了提高效率，可以让要求引入的模块提供一个提示、即“Hint”。意思是“您先找一下这儿，看对不对，要是不对您再挨个儿找”。所以，“引入目录”中的函数名数组其实并不真是字符串数组，而是一个 IMAGE_IMPORT_BY_NAME 结构数组，这种数据结构的定义是：

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD Hint;
    BYTE Name[1];
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

结构中的第一个成分就是 Hint，其数值就是目标函数在被引入模块的引出函数表中最可能的下标，这是在引入者模块在编译/连接时设置好的。然后是一个变长的字符数组，这就是函数名字符串。调用 LdrGetExportByName()时，可以把 Hint 也作为参数传下去，例如前面 LdrpProcessImportDirectoryEntry()中的那段代码就是这样：

```
    {
      IMAGE_IMPORT_BY_NAME *pe_name;
      pe_name = RVA(Module->BaseAddress, *FunctionNameList);
      *ImportAddressList = LdrGetExportByName(ImportedModule->BaseAddress,
                                              pe_name->Name, pe_name->Hint);
      ......
    }
```

而 LdrGetExportByName()，则首先以 Hint 作为下标在被引入模块的引出函数表中进行有针对的比对，比对不符才想别的办法，先是对分搜索，最后一手才是顺序搜索。一般使用 Hint 进行比对的命中率很高，所以效率也就大大提高了。

最后，回到前面__true_LdrInitializeThunk()的代码中，在完成了对所有模块的装入和连接以后，还调用了一个函数 LdrpAttachThread()，这一方面是为了 TLS 的初始化，更重要的是要以 DLL_THREAD_ATTACH 为参数调用这些 DLL 的入口函数 DllMain()。

```
NTSTATUS
LdrpAttachThread (VOID)
{
  ......

  Status = LdrpInitializeTlsForThread();
```

```
  if (NT_SUCCESS(Status))
  {
      ModuleListHead = &NtCurrentPeb()->Ldr->InInitializationOrderModuleList;
      Entry = ModuleListHead->Flink;

      while (Entry != ModuleListHead)
      {
          Module = CONTAINING_RECORD(Entry, LDR_MODULE,
                                     InInitializationOrderModuleList);
          if (Module->Flags & PROCESS_ATTACH_CALLED &&
              !(Module->Flags & DONT_CALL_FOR_THREAD) &&
              !(Module->Flags & UNLOAD_IN_PROGRESS))
            {
              TRACE_LDR("%wZ - Calling entry point at %x for thread attaching\n",
                        &Module->BaseDllName, Module->EntryPoint);
              LdrpCallDllEntry(Module, DLL_THREAD_ATTACH, NULL);
            }
          Entry = Entry->Flink;
      }

      Entry = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink;
      Module = CONTAINING_RECORD(Entry, LDR_MODULE, InLoadOrderModuleList);
      LdrpTlsCallback(Module, DLL_THREAD_ATTACH);
  }

  ......

  return Status;
}
```

这里的宏操作 CONTAINING_RECORD 定义为：

```
#define CONTAINING_RECORD(address, type, field) \
  ((type*)((PCHAR)(address) - (PCHAR)(&((type *)0)->field)))
```

因为 LDR_MODULE 数据结构是通过其不同的队列头挂入不同队列的，所以从队列中获取的只是指向相应队列头的指针，要通过这样换算才能得到指向其 LDR_MODULE 数据结构的指针。

还有个问题，当 CPU 从__true_LdrInitializeThunk()返回、也就是从 LdrInitializeThunk()返回时去了哪里。事实上，LdrInitializeThunk()是作为 APC 函数执行的，目的是为 EXE 映像的运行“打前站”，所以返回时又(间接地)回到了内核中，这正是下一片漫谈要讨论的话题。