深入剖析Win32可移植可执行文件格式
第一部分

作者：Matt Pietrek

很久以前，我开始为Microsoft Systems Journal（现在的MSDN® Magazine）写文章，其中有一篇名为"探索PE文件内幕——Win32可移植可执行文件格式之旅"的文章很受欢迎，大大超出了我的意料。直到现在，我还听说有人（甚至在Microsoft）仍然在使用那篇文章，它依旧被收录在MSDN Library中。不幸的是，文章的最大问题是它们是静止的。但是Win32®的世界在这些年已经发生了很大的变化，因此那篇文章已经严重过时了。我要从本月开始用两部分系列的文章来补救这种情况。

你可能想知道为什么要关注可执行文件的格式。答案永远是：操作系统的可执行文件格式和数据结构展现了操作系统内部许多信息。通过理解EXE和DLL的内部情况，你会发现你已经变成你周围一个更优秀的程序员。

当然，通过阅读Microsoft的PECOFF规范你可以获得许多我将要告诉你的内容。但是与大多数规范一样，它更注重完整性而不是可读性。在本文中，我把精力集中于解释整个故事中最重要的部分，同时填补那些并不适合出现在官方规范中的怎么样（How）以及为什么（Why）的问题。另外，在本文中我还会讲到一些非常有用的内容，它们并未出现在任何Microsoft官方文档中。

让我先举一些例子来说明自从1994年我写那篇文章以来有关可执行文件方面都发生了哪些变化。由于16位Windows®已经成为历史，因此没有必要再与Win16的NE（New Executable）格式相比较了。另一个已经脱离人们视野的是Win32s®。在Windows 3.1上运行Win32程序非常不稳定是最令人讨厌的事。

回到当时，Windows 95（当时代号为"Chicago"）甚至还未发行。Windows NT®还是3.5版。Microsoft链接器还未进行非常有效地优化。值得一提的是当时已经在MIPS和DEC Alpha上实现了Windows NT。

自从那篇文章以来都出现了什么新内容呢？64位Windows引进了它自己的变种的PE文件格式。Windows CE添加了许多的新型处理器。诸如DLL延迟加载、节合并以及绑定之类的优化已经铺天盖地。有许多新东西要加入到这个故事中。

让我们不要忘了Microsoft® .NET。该把它放在什么位置呢？对于操作系统来说，.NET可执行文件只不过是普通的Win32可执行文件。但是.NET运行时能够识别出这些可执行文件中的数据并把它作为元数据（metadata）和中间语言（Intermediate Language，IL），它们对.NET来说非常重要。在本文中，我要敲开.NET元数据结构的大门，但把对它全部光彩的彻底挖掘留给下一篇文章。

如果Win32世界中的所有这些加加减减还不足以成为我重新写那篇文章的理由的话，那么我只有列出原来那篇文章中的一些令我害怕的错误了。例如我对线程局部存储（TLS）支持情况的描述是错误的。同样，通篇我对日期/时间戳这个DWORD的描述仅在太平洋时区才是精确的！

另外，有许多内容在当时是正确的，但现在已经不正确了。我说过.rdata 节并没有太大的作用。今天，诚然是这样。我也说过.idata 节是可读/可写的节，但现在却有许多试图拦截 API 的人发现它在很多情况下都是不正确的。

伴随着在这篇文章中完全更新 PE 文件格式的故事，我也对用于显示 PE 文件内容的 PEDUMP 程序进行了彻底修改。PEDUMP 现在可以在 x86 和 IA-64 平台上编译和运行，并且能够转储 32 位和 64 位 PE 文件。最重要的是，PEDUMP 的源代码可以从本文开头的链接处下载。这样，你就有了一个用这里讲的概念和数据结构实际工作的例子。

PE 文件格式概览

Microsoft 引进了 PE 文件格式，更经常被称为 PE 格式，作为最初的 Win32 规范的一部分。然而 PE 文件源自 VAX/VMS 上早期的通用目标文件格式（Common Object File Format，COFF）。这是由于许多最初的 Windows NT 开发团队的成员都来自数字设备公司（Digital Equipment Corporation，DEC）。这些开发者很自然就使用现有的代码以便快速开始新的 Windows NT 平台。

之所以选择术语“可移植可执行”是打算要在所有支持的 CPU 上的所有版本的 Windows 上使用相同的可执行文件格式。从大的方面来说，这个目标已经实现，因为 Windows NT 及其后继操作系统、Windows 95 及其后继操作系统以及 Windows CE 都使用相同的可执行文件格式。

Microsoft 编译器生成的 OBJ 文件也使用 COFF 格式。从 COFF 格式的一些域使用的竟然是八进制编码你就能知道它是多么老。COFF 格式的 OBJ 文件中有许多数据结构和枚举类型与 PE 文件相同，后面我会提到。

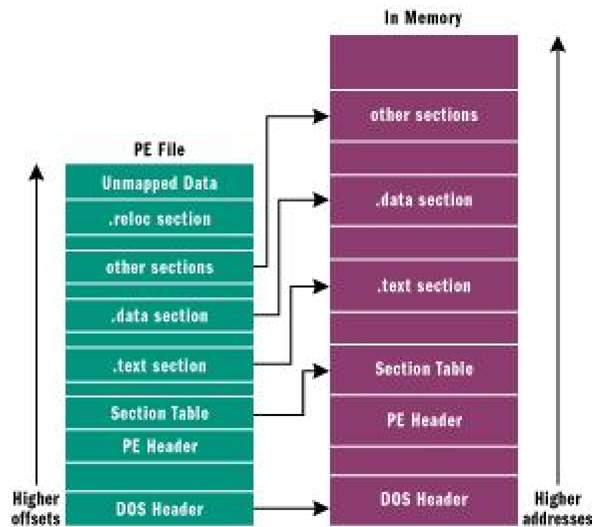
64 位 Windows 需要做的只是修改 PE 格式的少数几个域。这种新的格式被称为 PE32+。它并没有增加任何新域，仅从 PE 格式中删除了一个域。其余的改变就是简单地把某些域从 32 位扩展到 64 位。在大部分情况下，你都能写出同时适用于 32 位和 64 位 PE 文件的代码。Windows 头文件有这种魔力可以使这些区别对于大多数基于 C++的代码都不可见。

EXE 文件与 DLL 文件的区别完全是语义上的。它们使用的是相同的 PE 格式。惟一的不同在于一个位，这个位用来指示文件应该作为 EXE 还是 DLL。甚至 DLL 文件的扩展名也完全也是人为的。你可以给 DLL 一个完全不同的扩展名，例如.OCX 控件和控制面板小程序（.CPL）都是 DLL。

PE 文件一个非常好的地方就是它的数据结构在磁盘上与在内存中一样。加载一个可执行文件到内存（例如通过调用 LoadLibrary 函数）主要就是把 PE 文件中的某个部分映射到地址空间中。因此像 IMAGE_NT_HEADERS（后面我会讲到）这样的数据结构在磁盘上和在内存中是一样的。如果你知道如何在一个 PE 文件中找到某些内容，你几乎可以确定当文件被加载进内存时可以找到同样的信息。

注意到 PE 文件并不是作为单一的内存映射文件被映射进内存的这一点非常重要。相反，Windows 加载器查看 PE 文件并确定文件中的哪些部分需要被映射。当映射进内存时，文件中的高偏移相对于内存中的高地址。某项内容在磁盘文件中的偏移可能与它被加载进内存之后的偏移

不同，但是将磁盘文件中的偏移转换成内存偏移需要的所有信息都存在（见下图）。



当 PE 文件由 Windows 加载器加载进内存时，它在内存中被称为模块（module）。文件被映射到的内存的起始地址被称为 HMODULE。这是需要记住的一点：给你一个 HMODULE，你就知道在那个地址处到底有什么样的数据结构，并且你可以根据 PE 文件的知识找到内存中所有其它的数据结构。这个强大的功能可以被用作其它用途，例如拦截 API。（说得再准确一点，在 Windows CE 上 HMODULE 与加载地址并不相同，但那不是今天要讨论的内容。）

内存中的模块代表一个进程所需的可执行文件中的所有代码、数据和资源。PE 文件中的其它部分可能会被读取，但并不被映射进内存（例如重定位节）。一些部分可能根本就不被映射，例如放在文件末尾的调试信息。PE 文件头中的一个域告诉系统将这个可执行文件映射进内存时需要占用多少内存。不被映射的数据放在文件末尾，位于所有需要被映射的部分之后。

描述 PE 文件（和 COFF 文件）的关键位置是 WINNT.H 文件。在这个头文件中，你能找到几乎所有结构的定义、枚举类型以及使用 PE 文件或它在内存中的等价结构所需的定义。当然在其他地方有这方面的文档，例如 MSDN 中有“Microsoft 可移植可执行文件和通用目标文件格式文件规范”（2001 年十月 MSDN 的 Specifications 下）。但是 WINNT.H 确定了 PE 文件最终的样子。

有许多工具可以用来查看PE文件。Visual Studio附带的Dumpbin和Platform SDK附带的Depends就是其中的两个。我特别喜欢Depends，它以一种非常简洁的方式查看文件的导入表和导出表。另一个很好的PE文件查看工具是由Smidgeonsoft（http://www.smidgeonsoft.com）发行的PEBrowse Professional。本文中包含的PEDUMP程序也是一个非常全面的工具，几乎能做Dumpbin所能做到的一切。

从 API 的观点来看，Microsoft 提供的读取和修改 PE 文件的主要途径是 IMAGEHLP.DLL 文件。

在开始看 PE 文件的细节之前，先复习一下贯穿于整个 PE 文件方面的几个基本概念是非常值得的。在以下的部分中，我将讨论 PE 文件的节（section）、相对虚拟地址（RVA）、数据目录（Data Directory）以及如何导入函数。

PE 文件的节

PE 文件的节代表代码或某些类型的数据。虽然代码只能是代码，但数据却有许多种不同类型。除了可读/可写的程序数据（例如全局变量）外，节中其它类型的数据包括函数导入表和导出表、资源以及重定位信息等。每个节都有它自己的一组内存属性，其中包括节中是否包含代码，是只读的还是可读/可写的以及节中的数据是否在所有使用这个可执行文件的进程中是共享的等等。

一般说来，一个节中的所有代码和数据在逻辑上是相关的。通常一个 PE 文件中至少有两个节，一个是代码节，另一个是数据节。一般在 PE 文件中至少还有一种其它类型的数据节。我将在下个月本文的第二部分中具体描述各种节。

每个节都有一个惟一的名字。它通常用来表示节的用途。例如一个名为.rdata 的节表明它是一个只读数据节。使用节名只是为了方便人们处理文件，它对操作系统来说无关紧要。一个名为 FOOBAR 的节可能实际上是一个代码节，就像.text 节一样。Microsoft 通常在它们的节名前加一个圆点，但这并不是必须的。多少年来，Borland 链接器使用的节名都是 CODE 和 DATA。

虽然编译器生成的节都有标准设置，但那并没有什么神奇的。你可以创建和命名你自己的节，链接器很乐意把它们包含进可执行文件中。在 Visual C++中，你可以告诉编译器把代码或数据插入到你使用#pragma 语句命名的节中。例如以下语句

```
#pragma data_seg( "MY_DATA" )
```

导致所有数据都被 Visual C++放入一个称为 MY_DATA 的节中，而不是默认的.data 节。大多数程序员都使用编译器生成的默认节，但偶尔你也可能需要把代码或数据放进自己定义的节中。

节并不是完全由链接器生成的，在 OBJ 文件中就有它们的身影。这通常是由编译器放在那里的。链接器的工作就是把 OBJ 文件和库文件中所有需要的节组合成 PE 文件中最终相应的节。例如你的工程中的每个 OBJ 文件可能都至少有一个包含代码的.text 节。链接器把各种 OBJ 文件中的所有.text 节组合成单个的.text 节放入 PE 文件。同样，各种 OBJ 文件中的所有.data 节也被组合成 PE 文件中单个的.data 节。.LIB 文件中的代码和数据通常也被包含进可执行文件中，但对它的讨论已经超出了本文的范围。

链接器遵守一组相当完整的规则来确定组合哪个节以及如何组合。我在MSJ杂志 1997 年七月的Under The Hood专栏已经介绍过链接器算法。OBJ文件中有的节是专供链接器使用的，它们并不被放入最后的可执行文件中。这样的节通常用于编译器向链接器传递信息。

节有两种对齐值，一种是在文件中的对齐值，另一种是在内存中的对齐值。PE 文件头中指定了这两种值，它们可以不同。每个节的起始地址都是对齐值的倍数。例如在 PE 文件中，典型的对齐值是 0x200。因此每个节的起始起始地址都是 0x200 的倍数。

一旦映射进内存，节总是从页的边界开始。也就是说，当 PE 文件被映射进内存时，每个节的开头都对应于一个内存页的开始。在 x86 CPU 上，页按 4KB 对齐；在 IA-64 上，页按 8KB 对齐。以下是 PEDUMP 输出的 Windows XP 中的 KERNEL32.DLL 的.text 节和.data 节的情况：

```
Section Table
  01 .text      VirtSize: 00074658  VirtAddr:   00001000
    raw data offs:    00000400  raw data size: 00074800
```

……

```
  02 .data      VirtSize: 000028CA  VirtAddr:  00076000
    raw data offs:   00074C00  raw data size: 00002400
```

上面的输出表明，.text 节的文件偏移是 0x400，在内存中它比 KERNEL32 的加载地址高 0x1000 字节。同样，.data 节的文件偏移是 0x74C00，它在内存中比 KERNEL32 的加载地址高 0x76000 字节。

可以创建一个 PE 文件，使它的节在文件中的偏移与在内存中的偏移相同。但这会使可执行文件相当大，不过可以提高它在 Windows 9x 或 Windows Me 中的加载速度。使用默认的/OPT:WIN98 链接器选项（由 Visual Studio 6.0 引进）就可以创建这样的 PE 文件。在 Visual Studio® .NET 中，链接器可能使用也可能不使用这个选项，这取决于文件是不是足够小。

链接器一个比较有趣的功能就是合并节。如果两个节属性相似、相互兼容时，它们通常会在链接时被合并成一个节。这是通过链接器的/MERGE 选项来完成的。例如以下的链接器选项会把.rdata 节和.text 节组合成单个的.text 节：

/MERGE:.rdata=.text

把节合并起来的好处是可以节省在磁盘上和在内存中的空间。每个节至少要在内存中占用一个页面。如果你能将一个可执行文件中节的数目从四个减少到三个，你很可能就会少占用一页内存。当然，这取决于那两个被合并的节中未使用的空间加起来够不够一页。

当你合并节时就会发生一些有趣的情况，因为这并没有硬性的和快速的规则可以遵守。例如，把.rdata 节合并到.text 节当然可以，但是你不能把.rsrc 节、.reloc 节或者.pdata 节合并到其它节中。在 Visual Studio .NET 之前，你可以把.idata 节合并到其它节中。但是在 Visual Studio .NET 中，这是不允许的。但在创建程序的发行版本时，链接器自己却经常把部分的.idata 节合并到其它节中，例如.rdata 节。

由于导入表要被 Windows 加载器改写，你可能想知道它们怎么能被放在只读节中。这是因为在加载时系统临时将包含导入表的那些页面的属性设置为可读/可写。一旦导入表被初始化，这些页面就会被设置成原来的属性。

相对虚拟地址

在可执行文件中，许多地方都需要被指定一个在内存中的地址。例如在使用全局变量时需要它的地址。PE 文件可以被加载到进程地址空间中的任何地方。虽然它有一个首选地址，但你却不能依赖可执行文件一定会被加载到那个地址。因此就需要按一定方式指定地址，使它们并不依赖于可执行文件的加载地址。

为了避免在 PE 文件中硬编码内存地址，因此就使用了 RVA。RVA 只是一个相对于 PE 文件在内存中的加载位置的偏移。例如假定一个 EXE 文件被加载在 0x400000 处，而它的代码节在 0x401000 处。那么这个代码节的 RVA 就是：

（目标地址）0x401000 - （加载地址）0x400000 = （RVA）0x1000

要把一个 RVA 转换成实际地址，只需要简单地逆着上述过程进行：将 RVA 与实际加载地址相加就能得到实际的内存地址。顺便说一下，按照 PE 格式中的说法，实际的内存地址被称为虚拟地址（Virtual Address，VA）。另外一种考虑 VA 的方式就是把它当成 RVA 加上首选加载地址。不要忘了我前面说过加载地址与 HMODULE 是一回事。

想在内存中探索一些 DLL 内部的数据结构吗？这里就是方法——用 DLL 的名称作为参数调用 GetModuleHandle 函数，它返回的 HMODULE 就是这个 DLL 的加载地址，你可以利用你学的关于 PE 文件结构的知识在这个模块中找到你想找到的一切。

数据目录

在可执行文件中有许多数据结构需要被快速地定位。导入表、导出表、资源以及基址重定位信息等就是一些明显的例子。所有这些广为人知的结构都是以同样的方式被定位的，这些位置被称为数据目录。

数据目录是一个有 16 个（WINNT.H 中定义为 IMAGE_NUMBEROF_DIRECTORY_ENTRIES）元素的结构数组。每个数组元素所指代的内容已经被预先定义好了。WINNT.H 文件中的这些 IMAGE_DIRECTORY_ENTRY_*xxx* 定义就是数据目录的索引（从 0 到 15）。下表描述了每个 IMAGE_DIRECTORY_ENTRY_*xxx* 值所指代的内容。由它们指向的许多数据结构将在本文的第二部分中详细描述。

值	描述
IMAGE_DIRECTORY_ENTRY_EXPORT	指向导出表（IMAGE_EXPORT_DIRECTORY 结构）。
IMAGE_DIRECTORY_ENTRY_IMPORT	指向导入表（IMAGE_IMPORT_DESCRIPTOR 结构数组）。
IMAGE_DIRECTORY_ENTRY_RESOURCE	指向资源（IMAGE_RESOURCE_DIRECTORY 结构）。
IMAGE_DIRECTORY_ENTRY_EXCEPTION	指向异常处理程序表（IMAGE_RUNTIME_FUNCTION_ENTRY 结构数组）。它特定于 CPU，用于基于表的异常处理。适用于除 x86 之外所有类型的 CPU。
IMAGE_DIRECTORY_ENTRY_SECURITY	指向 WIN_CERTIFICATE 结构列表。此结构定义在 WinTrust.H 文件中。它并不作为映像的一部分被映射进内存。因此 VirtualAddress 域是文件偏移，而不是 RVA。
IMAGE_DIRECTORY_ENTRY_BASERELOC	指向基址重定位信息。
IMAGE_DIRECTORY_ENTRY_DEBUG	指向 IMAGE_DEBUG_DIRECTORY 结构数组。其中的每个元素描述了映像中的一些调试信息。要获得 IMAGE_DEBUG_DIRECTORY 结构的数目，用 Size 域除以 IMAGE_DEBUG_DIRECTORY 结构的大小。早期的 Borland 链接器将这个 IMAGE_DATA_DIRECTORY 项的 Size 域设置成 IMAGE_DEBUG_DIRECTORY 结构的数目，而不是数组的大小。
IMAGE_DIRECTORY_ENTRY_ARCHITECTURE	指向与平台相关的数据，这个数据是一个 IMAGE_ARCHITECTURE_HEADER 结构数组。x86 平台和 IA-64 平台并不使用，但好像已经用于 DEC/Compaq Alpha 平台。

导入函数

值	描述
IMAGE_DIRECTORY_ENTRY_GLOBALPTR	在某些平台上，其VirtualAddress域保存的是全局指针（Global Pointer ，GP）的RVA。x86 平台上不使用，但IA-64 平台上使用。Size域并未使用。要获取更多关于IA-64 GP方面的信息，可以参考2000 年 11 月的Under The Hood专栏。
IMAGE_DIRECTORY_ENTRY_TLS	指向线程局部存储（Thread Local Storage）初始化节。
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	指向 IMAGE_LOAD_CONFIG_DIRECTORY 结构。此结构中的信息特定于 Windows NT、Windows 2000 和 Windows XP（例如 GlobalFlag 值）。如果你的可执行文件要使用这个结构，需要定义一个名称为__load_config_used，类型为 IMAGE_LOAD_CONFIG_DIRECTORY 的全局结构体。对于非 x86 平台，这个名称需要被定义成_load_config_used（单下划线）。如果你想使用 IMAGE_LOAD_CONFIG_DIRECTORY 结构，必须使用这个技巧才能在你的 C++代码中得到正确的名字。链接器看到的符号名一定要是__load_config_used（带两个下划线）。C++编译器要在全局符号前加一个下划线。另外，它还使用类型信息来修饰（decorate）全局符号。因此要使一切正常，你应该像下面这个样子使用： extern "C" IMAGE_LOAD_CONFIG_DIRECTORY _load_config_used = {...}
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	指向 IMAGE_BOUND_IMPORT_DESCRIPTOR 结构数组。每个结构对应于这个映像已经绑定的一个 DLL。这个结构中的日期/时间戳（TimeDateStamp 域）可以让加载器快速确定这个绑定是否是最新的。如果不是，加载器将忽略绑定信息，并正常地解析导入的函数。
IMAGE_DIRECTORY_ENTRY_IAT	指向第一个导入地址表（IAT）的开头。对应于每一个导入的 DLL 都有一个相应的 IAT，并且它们在内存中依次排列。Size 域指出了所有 IAT 的总大小。加载器在解析导入符号期间使用这个地址和大小临时将包含 IAT 的页面标记为可读/可写。
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT	指向延迟加载信息，它是 CImgDelayDescr 结构数组，这个结构被定义在 Visual C++的 DELAYIMP.H 文件中。直到首次调用延迟加载的 DLL 中的函数时这个 DLL 才会被加载。特别需要注意的是：Windows 并不知道关于延迟加载 DLL 方面的任何信息。延迟加载特性完全是由链接器与运行时库来实现的。
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR	这个值在最新的系统头文件（CorHdr.h）中被更名为 IMAGE_DIRECTORY_ENTRY_COMHEADER。它指向可执行文件中的.NET 信息中的顶层信息，包括元数据。这个信息保存在 IMAGE_COR20_HEADER 结构中。

导入函数

当你使用其它 DLL 中的代码或数据时，就需要导入它们。当加载 PE 文件时，Windows 加载器的工作之一就是定位所有导入的函数和数据，以便加载的 PE 文件可以使用它们。我把对完成这个任务所需的数据结构的详细讨论留给本文的第二部分，在这里仅给出一个整体概念。

当你直接链接其它 DLL 中的代码和数据时，实际上隐含链接到了相应的 DLL 上。你并不需要做任何工作来让你的程序使用这些导入的函数。加载器把这些全包了。另一种使用 DLL 的方式是显式链接（explicit linking）。这意味着你需要明确地加载这些 DLL，然后查找这些函数的地址。这种方法几乎总是通过 LoadLibrary 和 GetProcAddress 这两个 API 来完成的。

当你隐含链接函数时，类似于使用 LoadLibrary 和 GetProcAddress 这两个 API 的代码仍然存在，但加载器自动为你做这些事。同时加载器确保这个被加载的 PE 文件所需的其它 DLL 也会被加载。例如用 Visual C++®创建的程序一般都会链接到 KERNEL32.DLL，而 KERNEL32.DLL 又从 NTDLL.DLL 中导入了函数。同样，如果你从 GDI32.DLL 导入函数，而实际上这个 DLL 依赖于 USER32、ADVAPI32、NTDLL 以及 KERNEL32 这些 DLL。加载器会确保这些 DLL 都被加载，以便解析这些导入的函数。（Visual Basic 6.0 和 Microsoft .NET 可执行文件并不直接链接到 KERNEL32，而是链接到了其它的 DLL 上，但原理是一样的。）

当隐含链接（也称为隐式链接）时，对主要的 EXE 文件及其依赖的所有 DLL 的解析过程发生在程序启动时。如果这时出现任何问题（例如它引用的一个 DLL 找不到），相应的进程就会被终止。

Visual C++ 6.0 引入了一个延迟加载（delayload）特性，它是隐含链接与显式链接的混合。当你延迟加载 DLL 时，链接器生成了一些非常类似于它为正常导入的 DLL 生成的数据那样的数据，但是操作系统却忽略这些数据。当你的程序在执行过程中首次调用这些延迟加载的函数其中之一时，由链接器为此生成的一部分代码就会执行，由它加载相应的 DLL（如果尚未加载），然后调用 GetProcAddress 来确定要调用的函数的地址。这些额外的工作使得接下来对这个函数的调用就好像这个函数是正常导入的一样。

在 PE 文件中，对应于每一个导入的 DLL 有一个相应的结构数组。其中的每个结构都给出了导入的 DLL 的名称和一个指向函数指针数组的指针。这个函数指针数组被称为导入地址表（Import Address Talbe，IAT）。每个导入的函数都在 IAT 中有一个对应的位置，Windows 加载器就在这个位置上写入这个导入函数的地址。这一点非常重要：一旦一个模块的加载过程结束，那么其 IAT 中就包含了导入函数的地址。

IAT 的美妙之处就在于，在 PE 文件中，只有一个地方保存了导入函数的地址。无论在你的程序中对某个导入的函数调用多少次，所有调用使用的同样都是 IAT 中对应于这个函数的指针。

现在让我们来看一下如何调用导入函数。它分为两种情况：高效率方式与低效率方式。按最好的情况（高效率方式）来说，对一个导入函数的调用应该像下面这个样子：

```
CALL DWORD PTR [0x00405030]
```

如果你不熟悉 x86 汇编语言，我可以告诉你这条指令表示通过函数指针来调用相应的函数。在地址 0x00405030 处的一个 DWORD 类型的值就是 CALL 指令要将控制权转到的地方。在这个例子中，地址 0x00405030 在 IAT 中。

调用导入函数的低效率方式类似下面这个样子：

```
CALL 0x0040100C
...
0x0040100C:
JMP       DWORD PTR [0x00405030]
```

在这种情况下，CALL 指令把控制权转到了一个小占位程序（stub）中。这个占位程序只是一条 JMP 指令，用以跳转到保存在地址 0x405030 处的地址中。同样，记住 0x405030 是 IAT 中的一个元素。一句话，调用 API 的这种低效率方式使用了 5 个字节的附加代码（JMP 指令是 1 字节，地址是 4 个字节），并且由于使用了额外的 JMP 指令，因此执行时要花费更长的时间。

你可能奇怪既然有高效率的调用方式,为什么还要使用低效率的调用方式呢？理由是很充足的。由于自身的限制，编译器并不能区分调用导入的函数与调用同一模块中的函数之间的区别。因此编译器为函数调用生成的指令是这样的：

CALL *XXXXXXXX*

而 *XXXXXXXX* 处是实际代码的地址，这个地址由链接器在后面填充。注意这最后的 CALL 指令并不是通过函数指针（调用函数的）。相反，它使用的是实际代码的地址。为了保持一致的方式，链接器需要用一个代码块来替换 *XXXXXXXX*。最简单的做法就是调用一个 JMP 之类的占位程序，就像上面你所看到的那样。

那 JMP 指令是从哪里来的呢？令人惊讶的是，它来自于相应函数的导入库。如果你仔细查看导入库，并且查看与导入函数名称相连的代码时，就会看到类似上面的 JMP 指令。这意味着，默认情况下，如果没有任何干涉，调用导入函数使用的总是低效率的调用方式。

按照逻辑推理，下一个要问的问题一定是如何才能使用高效率的调用方式。答案是你必须给编译器一个提示。__declspec(dllimport)这个函数修饰符告诉编译器这个函数在其它的 DLL 中，编译器应该生成下面这样的指令：

CALL DWORD PTR [XXXXXXXX]

而不是下面这样的指令：

CALL XXXXXXXX

另外，编译器生成相应的信息来告诉链接器去解析上面那条指令中的函数指针部分时应该去找的符号是__imp_函数名（就是在相应的函数名称前加上__imp_）。例如，如果你调用 MyFunction 这个函数，那么相应的符号名应为__imp_MyFunction。看一下导入库，除了看到正常的符号名外，你还会看到一个以__imp_为前缀的同样的符号名。这个__imp_类型的符号被直接解析成了 IAT 项，而不是 JMP 占位程序。

这对你日常工作有什么影响呢？如果你编写导出函数并且提供了相应的.H 文件，记得使用__declspec(dllimport)函数修饰符。例如：

```
__declspec(dllimport) void Foo(void);
```

如果你看一下 Windows 系统头文件，你会发现所有的 Windows API 都使用了__declspec(dllimport)。要想看到它并不容易，但是如果你搜索定义在 WINNT.H 中，并且用于像 WinBase.h 之类的头文件中的 DECLSPEC_IMPORT 宏，你就会发现__declspec(dllimport)是如何用于系统 API 声明的。

PE 文件结构

现在让我们开始挖掘PE文件的实际格式吧。我要从文件的开头处开始，描述存在于所有PE文件中的数据结构。然后我会描述PE文件的节中具体的数据结构（例如导入表与资源）。下面我要讨论的所有数据结构都被定义在WINNT.H文件中，除非我特别声明。

在许多情况下，32位和64位数据结构都是成对出现的——例如IMAGE_NT_HEADERS32和IMAGE_NT_HEADERS64。这些结构几乎总是一样的，除了相应的64位结构中一些域的数据宽度更宽。如果你想编写可移植的代码，在WINNT.H文件中有相应的宏，这些宏可以选择合适的32位或64位结构，并且把它们用一个不能表明大小的别名来代替（在上面的例子中，它就是IMAGE_NT_HEADERS）。结构的选择依赖于你想在何种模式下编译（具体来说就是是否定义了_WIN64）。只有在你所需编译成的PE文件的大小属性与你正在编译的平台的大小属性不同时才需要使用具体的32位或64位结构。

MS-DOS文件头

每一个PE文件都以一个小的MS-DOS®可执行文件开始。早期的Windows需要这个小占位程序，因为那时很多用户还未使用Windows。当可执行文件在没有安装Windows的机器上运行时，这个程序至少可以输出一条消息，用来指明它需要运行在Windows平台上。

PE文件开头是传统的MS-DOS文件头，其中前面的一部分被称为IMAGE_DOS_HEADER。此结构中最重要的两个域是e_magic和e_lfanew。e_lfanew域保存的是真正的PE文件头的偏移。e_magic域需要被设置成0x5A4D。它被定义为IMAGE_DOS_SIGNATURE。如果用ASCII码表示，0x5A4D就是“MZ”，这是Mark Zbikowski的姓名缩写，他是最初的MS-DOS设计者之一。

IMAGE_NT_HEADERS文件头

IMAGE_NT_HEADERS结构是存储PE文件细节的主要位置。它的偏移由文件开头的IMAGE_DOS_HEADER结构的e_lfanew域给出。实际有两种版本的IMAGE_NT_HEADER结构，一种供32位可执行文件使用，另一种供64位使用。它们之间的差别非常小，因此我在讨论中把它们看作相同的结构。区别这两种结构惟一正确的、由Microsoft官方认可的方法是通过IMAGE_OPTION_HEADER结构（很快就会讲到）的Magic域。

IMAGE_NT_HEADER结构由三个域组成：

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

在一个合法的PE文件中，Signature域被设置成0x00004550。用ASCII码表示为“PE\0\0”。它被定义为IMAGE_NT_SIGNATURE。第二个域是一个类型为IMAGE_FILE_HEADER的结构，这个结构在PE文件出现之前就已经出现了。它包含了关于文件的一些基本信息。最重要的是，其中有一个域指明了跟在这个结构后面的可选文件头的大小。在PE文件中，这个可选文件头是必须的，但它仍然被称为IMAGE_OPTIONAL_HEADER。

下表列出了IMAGE_FILE_HEADER结构的各个域及相应的描述。这个结构也可以在COFF格式的OBJ文件开头找到。

大小	域	描述
WORD	Machine	目标平台 CPU 的类型。常用的值有： IMAGE_FILE_MACHINE_I386 0x014c // Intel 386 IMAGE_FILE_MACHINE_IA64 0x0200 // Intel 64
WORD	NumberOfSections	指示节表中节的数目。节表紧跟着 IMAGE_NT_HEADERS 结构。
DWORD	TimeDateStamp	指示文件创建时间。这个值是从格林尼治时间（GMT）1970 年 1 月 1 日 00:00 以来的总秒数。它比文件系统所指明的日期/时间更精确。使用_ctime 函数可以很容易地把这个值转换成可读性比较好的字符串（这个函数与时区相关）。另一个可用于这个值的函数 gmtime 也比较有用。
DWORD	PointerToSymbolTable	COFF 符号表的文件偏移。Microsoft 的 PECOFF 规范 5.4 节描述了 COFF 符号表。COFF 符号表在 PE 文件中非常少见，因为新的调试符号格式已经取代了它。在 Visual Studio .NET 之前，可以使用/DEBUGTYPE:COFF 这个链接器选项来指定创建 COFF 符号表。它总是存在于 OBJ 文件中。如果不存在符号表的话，将它设置为 0。
DWORD	NumberOfSymbols	符号表中的符号数（如果存在的话）。COFF 符号是一个大小固定的结构，这个域用来定位 COFF 符号表的结尾。紧跟着 COFF 符号表的是一个字符串表，它用来保存长符号名。
WORD	SizeOfOptionalHeader	IMAGE_FILE_HEADER 结构后面的可选数据的大小。在 PE 文件中，这个可选数据就是 IMAGE_OPTIONAL_HEADER。这个大小在 32 位和 64 位文件中是不同的。对于 32 位 PE 文件来说，它通常是 224；对于 64 位 PE32+文件来说，它通常是 240。但是，它们只是最小值，可能有更大的值。
WORD	Characteristics	指示文件属性的一组位标志。这些标志的合法值就是 WINNT.H 文件中定义的 IMAGE_FILE_xxx 值。一些常见的值列于下表。

下表列出了常用的 IMAGE_FILE_*xxx* 值：

标志	描述
IMAGE_FILE_RELOCS_STRIPPED	重定位信息已经从文件中移除。
IMAGE_FILE_EXECUTABLE_IMAGE	文件是可执行映像。
IMAGE_FILE_AGGRESSIVE_WS_TRIM	让操作系统尽量减小工作集（working set）。
IMAGE_FILE_LARGE_ADDRESS_AWARE	此应用程序可以处理大于 2GB 的地址。
IMAGE_FILE_32BIT_MACHINE	需要字长为 32 位的机器。
IMAGE_FILE_DEBUG_STRIPPED	调试信息已经被移到.DBG 文件中。
IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	如果可执行映像在可移动媒体上,把它复制到交换文件中并从交换文件中运行。
IMAGE_FILE_NET_RUN_FROM_SWAP	如果可执行映像在网络上,把它复制到交换文件中并从交换文件中运行。
IMAGE_FILE_DLL	文件是 DLL。
IMAGE_FILE_UP_SYSTEM_ONLY	只能运行于单处理器机器上。

下表列出了 IMAGE_OPTIONAL_HEADER 结构的成员：

大小	域	描述
WORD	Magic	一个特征字，用于表明文件头的类型。两个常用的值为： IMAGE_NT_OPTIONAL_HDR32_MAGIC 0x10b IMAGE_NT_OPTIONAL_HDR64_MAGIC 0x20b
BYTE	MajorLinkerVersion	用于创建这个可执行文件的链接器的主版本号。对于由 Microsoft 链接器生成的可执行文件来说，这个版本号对应于 Visual Studio 的版本号（例如 Visual Studio 6.0 就是版本 6）。
BYTE	MinorLinkerVersion	用于创建这个可执行文件的链接器的次版本号
DWORD	SizeOfCode	带有 IMAGE_SCN_CNT_CODE 属性的所有节的总大小。
DWORD	SizeOfInitializedData	所有由已初始化的数据组成的节的总大小。
DWORD	SizeOfUninitializedData	所有由未初始化的数据组成的节的总大小。它通常是 0，因为链接器经常把未初始化的数据添加到正常的数据节的末尾。
DWORD	AddressOfEntryPoint	文件中首先被执行的代码的第一个字节的 RVA。对于 DLL 来说，入口点在进程初始化和退出期间，以及线程创建和退出期间都会被调用。在大多数可执行文件中，这个地址并不是直接指向 main、WinMain 或者 DllMain，而是指向调用上述函数的运行时库代码。对于 DLL 来说，这个域可以设为 0，这样它就接收不到前面说的四个通知。/NOENTRY 链接器选项可以将这个域设置为 0
DWORD	BaseOfCode	加载进内存之后代码的第一个字节的 RVA。
DWORD	BaseOfData	理论上这是加载进内存之后数据的第一个字节的 RVA。但是这个域的值在不同版本的 Microsoft 链接器间是不一致的。64 位可执行文件中并不存在这个域。
DWORD	ImageBase	这个文件在内存中的首选加载地址。如果有可能的话（也就是说这个内存当前并未被占用，并且它是对齐的，同时是一个合法的地址等等），加载器尽量把 PE 文件加载到这个地址。如果可执行文件被加载到这个地址，加载器就可以跳过基址重定位（将在本文的第二部分中描述）。对于 EXE 来说，默认的 ImageBase 为 0x400000；对于 DLL 来说，它是 0x10000000。可以在链接时使用/BASE 选项或者以后使用 REBASE 工具来设定此值。
DWORD	SectionAlignment	加载进内存之后节的对齐值。这个对齐值必须大于或等于文件对齐值（下面将要讲到）。默认的对齐值是目标平台的页面大小。对于运行于 Windows 9x 或 Windows Me 上的用户模式的可执行文件来说，最小的对齐值是一个页面（4KB）。这个域的值可以使用/ALIGN 链接器选项来设定。

大小	域	描述
DWORD	FileAlignment	节在 PE 文件中的对齐值。对于 x86 可执行文件来说，它或者是 0x200，或者是 0x1000。不同版本的 Microsoft 链接器的默认值不一样。这个值必须是 2 的幂，并且如果 SectionAlignment 域的值小于 CPU 的页面大小，这个值必须与 SectionAlignment 域的值匹配。链接器选项 /OPT:WIN98 将 x86 平台上的可执行文件的对齐值设为 0x1000，而/OPT:NOWIN98 选项将它设为 0x200。
WORD	MajorOperatingSystemVersion	所需的操作系统的主版本号。随着众多版本 Windows 的到来，这个域已失去了它最初的意义。
WORD	MinorOperatingSystemVersion	所需的操作系统的次版本号。
WORD	MajorImageVersion	此文件的主版本号。系统并未使用这个域，可以设置为 0。使用/VERSION 链接器选项可以设定这个域的值。
WORD	MinorImageVersion	此文件的次版本号。
WORD	MajorSubsystemVersion	可执行文件所需的子系统的主版本号。以前相对于旧版本的 Windows NT 界面来说，用它来指明需要新的 Windows 95 或 Windows NT 4.0 用户界面。现在由于 Windows 版本繁多，这个域已经不使用了，通常被设为 4。使用链接器选项 /SUBSYSTEM 可以设置这个域的值。
WORD	MinorSubsystemVersion	可执行文件所需的子系统的次版本号。
DWORD	Win32VersionValue	一个从来不用的域，通常设为 0。
DWORD	SizeOfImage	SizeOfImage 包含了假设存在于最后一个节之后的那个节的 RVA。这等效于把此文件加载进内存时系统需要保留的内存数量。这个域的值必须是节的对齐值的倍数。
DWORD	SizeOfHeaders	MS-DOS 文件头、PE 文件头和节表的总大小。在 PE 文件中，这些内容出现于任何代码或数据节之前。这个域的值被向上舍入到文件对齐值的倍数。
DWORD	CheckSum	映像的校验和。IMAGEHLP.DLL 中的 CheckSumMappedFile API 可以计算这个值。对于内核模式的驱动程序和一些系统 DLL 来说，校验和是必须的。否则这个域被设置为 0。当使用/RELEASE 链接器选项时，校验和会被放在文件中。

WORD	Subsystem	指示可执行文件所需子系统（用户界面类型）的一个枚举值。在 EXE 文件中这个值比较重要。一些重要的值如下：
		IMAGE_SUBSYSTEM_NATIVE
		// 不需要子系统
		IMAGE_SUBSYSTEM_WINDOWS_GUI
		// 使用 Windows GUI
		IMAGE_SUBSYSTEM_WINDOWS_CUI
		// 控制台应用程序。当它运行时，操作系统为其创一
		// 个控制台并提供 stdin、stdout 和 stderr 文件句柄。

大小	域	描述

WORD	DllCharacteristics	指示 DLL 特征的标志。这些值对应于 WINNT.H 文件中的 IMAGE_DLLCHARACTERISTICS_xxx 定义。当前值如下：
		IMAGE_DLLCHARACTERISTICS_NO_BIND
		// 不绑定映像
		IMAGE_DLLCHARACTERISTICS_WDM_DRIVER
		// 使用 WDM 模型的驱动程序
		IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE
		// 当终端服务器加载一个并没有准备运行于终端服务
		// 器上的应用程序时，它同时加载包含兼容代码的 DLL

DWORD	SizeOfStackReserve	在 EXE 文件中，它表示进程中的线程堆栈最初可以增长到的最大值。默认是 1MB。并不是初始化时就提交这里指定的所有内存。
DWORD	SizeOfStackCommit	在 EXE 文件中，它表示初始化时提交的堆栈的大小。默认是 4KB。
DWORD	SizeOfHeapReserve	在 EXE 文件中，它表示最初为默认进程堆保留的内存数量。默认是 1MB。然而对于当前版本的 Windows，在没有用户干预的情况下，堆可以超过这个值。
DWORD	SizeOfHeapCommit	在 EXE 文件中，它表示提交的堆的大小。默认是 4KB。
DWORD	LoaderFlags	此域已经废弃不用。
DWORD	NumberOfRvaAndSizes	在 IMAGE_NT_HEADERS 结构末尾处是一个 IMAGE_DATA_DIRECTORY 结构数组。这个域包含了这个数组的元素数目。由于以前发行的 Windows NT 的原因，它被设置为 16。
IMAGE_ DATA_ DIRECTORY	DataDirectory[16]	IMAGE_DATA_DIRECTORY 结构数组。每个结构包含可执行文件中一些重要部分（例如导入表、导出表以及资源等）的 RVA 和大小。

IMAGE_OPTIONAL_HEADER 结构末尾的 DataDirectory 数组就像是可执行文件中重要位置的地址簿。DataDirectory 的每个元素结构如下：

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;      // 数据的 RVA
    DWORD   Size;                // 数据的大小
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

节表

紧跟着 IMAGE_NT_HEADERS 结构的是节表（section table）。节表是一个 IMAGE_SECTION_HEADER 结构数组。此结构提供了与它相关的节的信息，其中包括位置、长度和属性。下表给出了对此结构的描述。节表中此结构的数目由 IMAGE_NT_HEADERS.FileHeader.NumberOfSections 给出。

大小	域	描述
BYTE	Name[8]	节的名称（ASCII 码）。节名并不保证以 NULL 结尾。如果你指定的节名大于 8 个字节，链接器在生成可执行文件时将其截断为 8 个字符。在 OBJ 文件中存在一种机制可以让节名更长。节名通常以圆点开始，但这并不是必需的。对于带有$字符的节名链接器会特殊对待。如果几个节名中$字符以前的部分相同，那么这些节会被合并。它们按$字符后面的部分在字母表中的顺序出现于最终的节中。关于节名中带有$字符的节和它们如何被合并方面还有很多内容，但对它的详细讨论已经超出了本文的范围。
DWORD	VirtualSize	指示节实际占用的内存大小。这个域的值可能比 SizeOfRawData 域的值大或小。如果大，SizeOfRawData 域表示可执行文件中已初始化的数据的大小，VirtualSize 域比它大的部分用 0 填充。在 OBJ 文件中，此域的值为 0。
DWORD	VirtualAddress	在可执行文件中，它表示在内存中节的起始 RVA。在 OBJ 文件中它被设置为 0。
DWORD	SizeOfRawData	可执行文件或 OBJ 文件中的节中存储的数据的大小（以字节计）。对于可执行文件来说，它必须是 PE 文件头中给出的文件对齐值的倍数。如果它被设置为 0，表示这个节中是未初始化的数据。
DWORD	PointerToRawData	节中数据起始的文件偏移。对于可执行文件来说，它必须是 PE 文件头中给出的文件对齐值的倍数。
DWORD	PointerToRelocations	节的重定位信息的文件偏移。它只用于 OBJ 文件，在可执行文件中它被设置为 0。在 OBJ 文件中，如果它不为 0，那么它指向一个 IMAGE_RELOCATION 结构。
DWORD	PointerToLinenumbers	节中 COFF 行号信息的文件偏移。如果它不为 0，那么它指向一个 IMAGE_LINENUMBER 结构。
WORD	NumberOfRelocations	PointerToRelocations 域指向的重定位信息的数目。在可执行文件中应该为 0。
WORD	NumberOfLinenumbers	PointerToLinenumbers 域指向的行号信息的数目。只有当生成 COFF 行号信息时才使用。
DWORD	Characteristics	指示节属性的标志（可以用“或”连接）。这些标志中的大部分可以使用链接器的/SECTION 选项来设置。常用的值列于下表。

下表列出了常用的节属性标志：

标志	描述
IMAGE_SCN_CNT_CODE	节中包含代码。
IMAGE_SCN_MEM_EXECUTE	节是可执行的。
IMAGE_SCN_CNT_INITIALIZED_DATA	节中包含已初始化的数据。
IMAGE_SCN_CNT_UNINITIALIZED_DATA	节中包含未初始化的数据。
IMAGE_SCN_MEM_DISCARDABLE	这个节在最终的可执行文件中可以被丢弃。用于保存链接器使用的信息，包括.debug$节。

标志	描述
IMAGE_SCN_MEM_NOT_PAGED	这个节不能被交换到页面文件中，因此它应该总是存在于物理内存中。经常用于内核模式驱动程序。
IMAGE_SCN_MEM_SHARED	包含这个节的物理页面将在加载这个可执行文件的所有进程之间共享。因此每个进程看到的这个节中的数据的值完全一样。对于在进程的所有实例之间共享全局变量比较有用。要共享某个节，使用/SECTION:节名,S 链接器选项。
IMAGE_SCN_MEM_READ	节是可读的。几乎总是设置这个值。
IMAGE_SCN_MEM_WRITE	节是可写的。
IMAGE_SCN_LNK_INFO	节中包含链接器使用的信息。仅存在于 OBJ 文件中。
IMAGE_SCN_LNK_REMOVE	这个节中的内容将不成为最终的映像的一部分。仅用于 OBJ 文件。
IMAGE_SCN_LNK_COMDAT	节中的内容是公共数据（comdat）。公共数据（Communal data）是可以被定义在多个 OBJ 文件中的数据（或代码）。链接器只将其中的一份副本包含进最终的可执行文件中。Comdats 对于支持 C++的模板函数和函数级的链接至关重要。它仅存在于 OBJ 文件中。
IMAGE_SCN_ALIGN_xBYTES	这个节中的数据在最终的可执行文件中的对齐值。有各种各样的值（_4BYTES，_8BYTES，_16BYTES 等）。如果不指定，默认为 16 字节。仅在 OBJ 文件中才设置这些标志。

可执行文件中的节在文件中的对齐值对文件的大小有重要影响。在 Visual Studio 6.0 中，链接器默认的节对齐值是 4KB，除非使用/OPT:NOWIN98 选项或/ALIGN 选项。对于 Visual Studio .NET 链接器，虽然仍是默认使用/OPT:NOWIN98 选项，但它要确定可执行文件是否小于某一固定值，如果小于的话，它将使用 0x200 字节的对齐值。

另一个比较有用的对齐值来自.NET 文件规范。这个规范说.NET 可执行文件在内存中的对齐值应该为 8KB，而不是 x86 上的 4KB。这是为了确保用 x86 入口点代码创建的.NET 可执行文件可以运行在 IA-64 中。如果在内存中节的对齐值为 4KB，那么 IA-64 将不能加载这个文件，因为在 64 位 Windows 上页面是按 8KB 对齐的。

总结

这一部分主要讲了 PE 文件头。在本文的第二部分中，我会继续带领读者游历可执行文件中常见的节。然后讲一下这些节中主要的数据结构，其中包括导入表、导出表以及资源。最后我会讲一下最新的经过彻底改进的 PEDUMP 工具的源代码。

（*MSDN Magazine* 2002 年 2 月Under The Hood专栏）

译者：SmartTech　　电子信箱：zhzhtst@163.com