深入剖析 Win32 可移植可执行文件格式
第二部分

作者：Matt Pietrek

上个月在本文的第一部分中，我首先对可移植可执行文件进行了全面的介绍。我讲了 PE 文件的历史和组成 PE 文件头的数据结构，还讲了节表。PE 文件头和节表告诉你在可执行文件中都包含什么类型的代码和数据，以及在哪里能找到它们。

本月我要讲一下常见的节。最后讲一下我的最新的经过彻底改进的PEDUMP程序，它可以在 2002 年 2 月的专栏中下载。如果你不熟悉PE文件的基本概念，应该首先读一下本文的第一部分。

上个月我讲了节是怎样的一个逻辑上属于一起的代码或数据块。例如可执行文件的所有导入信息都在一个节中。现在让我们来看一下在可执行文件和 OBJ 文件中经常遇到的一些节。除非特别说明，否则下表中的节名都来自 Microsoft 的工具。

名称	描述
.text	默认的代码节。
.data	默认的可读/可写数据节。全局变量通常在这个节中。
.rdata	默认的只读数据节。字符串常量和 C++/COM 虚表就放在这个节中。
.idata	导入表。实际上，链接器经常把.idata 节合并到其它节中（或者是明确指定的，或者是通过链接器的默认行为）。默认情况下，链接器仅在创建发行版的程序时才把.idata 节合并到其它节中。
.edata	导出表。当创建要导出函数或数据的可执行文件时，链接器会创建一个.EXP 文件。这个.EXP 文件包含一个.edata 节，这个节被添加到最后的可执行文件中。与.idata 节一样，.edata 节也经常被合并到.text 节或.rdata 节中。
.rsrc	资源节。这个节是只读的。它不应该被命名为其它名称，也不应该被合并到其它节中。
.bss	未初始化的数据节。在最新的链接器创建的可执行文件中很少见到。链接器扩展可执行文件的.data 节的 VirtualSize 域以便容纳未初始化的数据。
.crt	添加到可执行文件中的数据，用来支持C++运行时库（CRT）。一个比较好的例子就是用于调用静态C++对象的构造函数和析构函数的指针。要获取更详细的信息，可以参考2001 年 1 月的Under The Hood专栏。
.tls	这个节中的数据用来支持使用__declspec(thread)语法创建的线程局部存储变量。它包括数据的初始值，以及运行时需要的附加变量。
.reloc	可执行文件中的基址重定位节。通常 DLL 需要基址重定位信息而 EXE 并不需要。在创建发行版的程序时，链接器并不为 EXE 文件生成基址重定位信息。可以使用/FIXED 链接器选项移除基址重定位信息。
.sdata	通过全局指针（Global Pointer）相对寻址的“短（Short）”可读/可写数据。用于 IA-64 和其它使用全局指针寄存器的平台上。IA-64 平台上正常大小的全局变量在这个节中。
.srdata	通过全局指针相对寻址的“短（Short）”只读数据。用于 IA-64 和其它使用全局指针寄存器的平台上。
.pdata	异常表。它包含一个 IMAGE_RUNTIME_FUNCTION_ENTRY 结构数组，这个结构与平台体系结构相关。数据目录中索引为 IMAGE_DIRECTORY_ENTRY_EXCEPTION 的项指向它。用于使用基于表的异常处理的平台，例如 IA-64。惟一不使用基于表的异常处理的平台是 x86（它使用的是

名称	描述
	基于堆栈的异常处理）。
.debug$S	OBJ 文件中的 Codeview 格式的调试符号（Symbol）信息。这是一列可变长度的 CodeView 格式的调试符号记录。
.debug$T	OBJ 文件中的 Codeview 格式的调试类型（Type）记录。这是一列可变长度的 CodeView 格式的调试类型记录。
.debug$P	可以在使用预编译头（Precompiled Headers）生成的 OBJ 文件中找到这个节。
.drectve	这个节包含链接器指令，并且只存在于 OBJ 文件中。这些指令是传递到链接器命令行的 ASCII 码字符串，例如：-defaultlib:LIBC。指令之间用空格分开。
.didat	延迟加载导入数据。可以在非发行版本的可执行文件中找到。在发行版本中，延迟加载数据被合并到其它节中。

导出表

当一个 EXE 或 DLL 导出函数或变量时，其它 EXE 或 DLL 就可以使用这些导出的函数或变量。为了简单起见，我把导出的函数和导出的变量统称为“符号”。当导出一些符号时，最起码导出符号的地址需要能够以一种已定义好的方式被获取。每个导出的符号都有一个与之关联的序数，它可以用来查找这个符号。同时，几乎总有一个 ASCII 码格式的字符串名称与这个导出的符号关联。一般来说，导出的符号名与源文件中的符号名是一样的，尽管它们可以被修改的不一样。

通常，当可执行文件导入符号时，它使用的是符号的名称而不是它的序号。但是当通过名称导入时，系统仅使用这个名称去查找所需符号对应的导出序数，然后根据这个序数值去获取相应的地址。如果先使用的是序数值的话查找过程会快一点。通过名称导出和导入只是为了让程序员使用方便罢了。

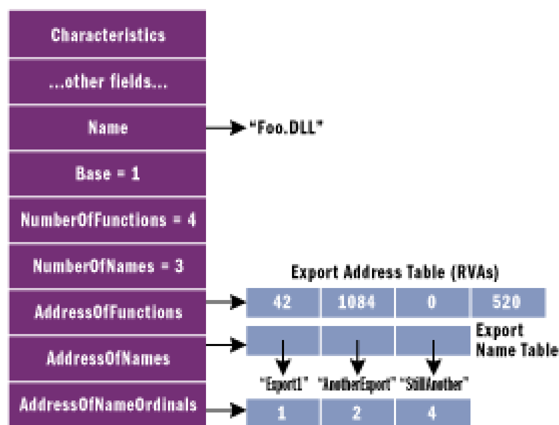
在.DEF 文件中的 Exports 节中使用 ORDINAL 关键字可以告诉链接器创建一个导入库，这个导入库强制函数只能通过序数导入而不能通过名称导入。

我首先介绍 IMAGE_EXPORT_DIRECTORY 结构，如下表所示：

大小	域	描述
DWORD	Characteristics	导出标志。当前未定义任何值。
DWORD	TimeDateStamp	导出数据的创建时间。这个域的定义与 IMAGE_NT_HEADERS.FileHeader.TimeDateStamp 相同(从 GMT 时间 1970 年 1 月 1 日 00:00 以来的总秒数)。
WORD	MajorVersion	导出数据的主版本号。未用，设置为 0。
WORD	MinorVersion	导出数据的次版本号。未用，设置为 0。
DWORD	Name	与导出符号相关的 DLL 的名称 ASCII 字符串的 RVA（例如 KERNEL32.DLL）。
DWORD	Base	这个域包含了这个可执行文件的导出符号所使用的序数值的起始值。通常情况下这个值为 1，但并不总是这样。当通过序数查找导出符号时，将序数值减去这个域的值就得到了这个导出符号在导出地址表（Export Address Table ，EAT）中的索引。

大小	域	描述
DWORD	NumberOfFunctions	EAT 中的元素数。注意 EAT 中的某些元素可能为 0，这表明没有代码/数据使用那个序数值导出。
DWORD	NumberOfNames	导出名称表（Export Names Table，ENT）中的元素数。这个域的值总是小于或等于 NumberOfFunctions 域的值。当某些符号仅使用序数导出时，它就小于那个域的值。如果导出序数之间有间隔，它同样也小于那个域的值。这个域的值也是导出序数表的大小（见下文）。
DWORD	AddressOfFunctions	EAT 的 RVA。EAT 中的每个元素都是一个 RVA。其中每个非 0 的 RVA 都对应一个导出符号。
DWORD	AddressOfNames	ENT 的 RVA。ENT 中的每个元素都是一个 ASCII 码字符串的 RVA。其中的每个 ASCII 码字符串都对应一个由名称导出的符号。这些字符串是按一定顺序排列的。这就使得加载器在查找导出符号时可以进行二进制搜索。名称字符串的排序是按二进制（与 C++运行时库函数 strcmp 类似），而不是与位置相关的字母表顺序。
DWORD	AddressOfNameOrdinals	导出序号表的 RVA。这个表是一个 WORD 类型的数组。它将 ENT 中的索引映射到导出地址表中相应的元素上。

导出目录（Export Directory）指向三个数组和一个 ASCII 码字符串表。其中只有导出地址表是必需的，它是一个由指向导出函数的指针组成的数组。导出序数是这个数组的索引（见下图）。



让我们通过例子来看一下导出表的工作原理。下图显示了 KERNEL32.DLL 导出表的部分内容：

```
exports table:
  Name:            KERNEL32.dll
  Characteristics: 00000000
  TimeDateStamp:   3B7DDFD8 -> Fri Aug 17 23:24:08 2001
  Version:         0.00
  Ordinal base:    00000001
  # of functions:  000003A0
  # of Names:      000003A0

  Entry Pt  Ordn  Name
```

```
  00012ADA      1   ActivateActCtx
  000082C2      2   AddAtomA
•••remainder of exports omitted
```

假设你调用 GetProcAddress 来获取 KERNEL32 中的 AddAtomA 这个 API 的地址。这时系统开始查找 KERNEL32 的 IMAGE_EXPORT_DIRECTORY 结构。它从那里获取了导出名称表的起始地址，知道了在这个数组中有 0x3A0 个元素，它通过二进制搜索来查找字符串“AddAtomA”。

假设加载器发现 AddAtomA 是这个数组中的第二个元素。然后它从导出序数表（Export Ordinal Table）中读取相应的第二个值。这个值就是 AddAtomA 的导出序数。将这个导出序数作为 EAT 的索引（加上 Base 域的值），它最终获取 AddAtomA 的相对虚拟地址（RVA）是 0x82C2。将此值与 KERNEL32 的加载地址相加就得到了 AddAtomA 的实际地址。

导出转发

导出表一个特别聪明的地方是它能将一个导出函数转发（Forwarding）到其它 DLL。例如在 Windows NT®、Windows® 2000 和 Windows XP 中，KERNEL32 中的 HeapAlloc 函数被转发到了 NTDLL 导出的 RtlAllocHeap 函数上。转发是在链接时通过.DEF 文件中的 EXPORTS 节中的一种特殊语法形式来实现的。对于 HeapAlloc 这个例子，KERNEL32 的.DEF 文件一定包含下面的内容：

```
EXPORTS
•••
HeapAlloc = NTDLL.RtlAllocHeap
```

怎样才能区别转发的函数与正常导出的函数呢？这需要一些技巧。通常 EAT 中包含的是导出符号的 RVA。但是如果这个 RVA 位于导出表中(通过相应的 DataDirectory 中的 VirtualAddress 域和 Size 域进行判断），那么它就是转发的。

当转发一个符号时，它的 RVA 很明显不能是当前模块中的代码或数据的地址。实际上，它的 RVA 指向一个由 DLL 和转发到的符号名称组成的字符串。在前面的例子中，这个字符串就是 NTDLL.RtlAllocHeap。

导入表

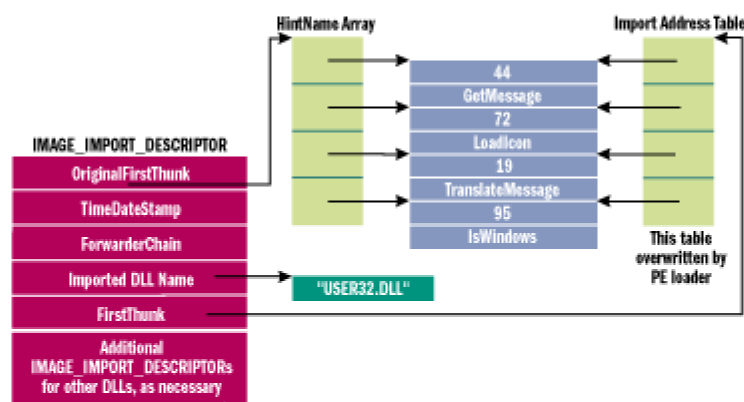
与导出函数或变量相反的就是导入它们。为了与前面保持一致，我仍然使用“符号”这个术语来指代导入的函数和变量。

导入数据被保存在 IMAGE_IMPORT_DESCRIPTOR 结构中。对应着导入表的数据目录项就指向由这个结构组成的数组。每个 IMAGE_IMPORT_DESCRIPTOR 结构都与一个导入的可执行文件对应。这个数组的最后一个元素的所有域都被设置为 0。下表是这个结构的内容：

大小	域	描述
DWORD	OriginalFirstThunk	这个域的命名太不恰当。它包含导入名称表的 RVA。导入名称表是一个 IMAGE_THUNK_DATA 结构数组。这个域被设置为 0 表示 IMAGE_IMPORT_DESCRIPTOR 结构数组的结尾。

大小	域	描述
DWORD	TimeDateStamp	如果可执行文件并未绑定导入的 DLL，这个域的值为 0。当使用老的绑定类型进行绑定（参考“绑定”一节）时，这个域包含日期/时间戳。当使用新的绑定类型进行绑定时，这个域的值为-1。
DWORD	ForwarderChain	这是首个转发的函数的索引。如果没有转发的函数，这个域被设置为-1。它仅用于老的绑定类型，因为那种绑定类型不能很有效地处理转发的函数。
DWORD	Name	导入的 DLL 名称字符串（ASCII 码格式）的 RVA。
DWORD	FirstThunk	导入地址表的 RVA。IAT 是一个 IMAGE_THUNK_DATA 结构数组。

每个 IMAGE_IMPORT_DESCRIPTOR 结构指向两个数组，这两个数组实际上是一样的。它们有好几种叫法，但最常用的名称是导入地址表（Import Address Table，IAT）和导入名称表（Import Name Talbe，INT）。下图显示的是可执行文件从 USER32.DLL 中导入一些 API 时的情况。



这两个数组的元素均为 IMAGE_THUNK_DATA 类型的结构，这个结构是一个与指针大小相同的共用体（或者称为联合）。每个 IMAGE_THUNK_DATA 结构对应着从可执行文件中导入的一个函数。这两个数组最后都以一个值为 0 的 IMAGE_THUNK_DATA 结构作为结尾。这个共用体（实际是一个 DWORD 值）可以有如下几种含义：

```
DWORD ForwarderString;// 转发函数字符串的 RVA（见上文）
DWORD Function;       // 导入函数的内存地址
DWORD Ordinal;        // 导入函数的序数
DWORD AddressOfData;  // IMAGE_IMPORT_BY_NAME 和导入函数名称的 RVA（见下文）
```

IAT中的IMAGE_THUNK_DATA结构的用途可以分为两种。在可执行文件中，它们或者是导入函数的序数，或者是一个IMAGE_IMPORT_BY_NAME结构的RVA。IMAGE_IMPORT_BY_NAME结构只是一个WORD类型的值，它后面跟着导入函数的名称字符串。这个WORD类型的值是一个“提示（hint）”，它提示加载器导入函数的序号可能是什么。当加载器加载可执行文件时，它用导入函数的实际地址来覆盖IAT中的每个元素。这一点是理解下文的关键。我强烈建议你读一读本期杂志中Russell Osterlund的文章——揭开Windows加载器的神秘面纱，这篇文章详细讲述了Windows加载器的行为。

在可执行文件被加载之前，是否存在一种方法能够区分 IMAGE_THUNK_DATA 结构中到底包含的是导入函数的序数呢，还是 IMAGE_IMPORT_BY_NAME 结构的 RVA 呢？答案在 IMAGE_THUNK_DATA

结构的最高位。如果它为 1，那么低 31 位（在 64 位可执行文件中是低 63 位）中是导入函数的序数。如果最高位为 0，那么 IMAGE_THUNK_DATA 结构的值就是 IMAGE_IMPORT_BY_NAME 结构的 RVA。

另一个数组 INT，本质上与 IAT 是一样的。它也是一个 IMAGE_THUNK_DATA 结构数组。关键的区别在于当加载器将可执行文件加载进内存时，它并不覆盖 INT。为什么对于从 DLL 中导入的每组 API 都需要有两个并列的数组呢？答案在于一个称为绑定（binding）的概念。当在绑定过程（后面我会讲到）中覆盖可执行文件的 IAT 时，需要以某种方式保存原来的信息。而作为这个信息的副本的 INT，正是这个用途。

INT 对于可执行文件的加载并不是必需的。但是如果它不存在的话，那么这个可执行文件就不能被绑定。Microsoft 链接器总是生成 INT，但是长期以来，Borland 链接器（TLINK）都不生成它。这样，由 Borland 链接器生成的可执行文件就不能被绑定。

在早期的 Microsoft 链接器中，导入节并不是专门针对于链接器的。组成可执行文件导入节的所有数据都来自导入库。你可以对一个导入库文件运行 DUMPBIN 或 PEDUMP 来看一下。你会发现一些节名类似于.idata$3 和.idata$4 的节。链接器只是简单地遵守它的规则来组合节，所有的结构和数组就神奇般地各就其位了。几年前 Microsoft 引进了一种新的导入库格式，这种导入库特别小，以便让链接器能在创建导入数据时更具主动性。

绑定

当可执行文件被绑定时（例如通过Bind程序），其IAT中的IMAGE_THUNK_DATA结构中是导入函数的实际地址。也就是说，磁盘上的可执行文件的IAT中存储的就是其导入的DLL中的函数在内存中的实际地址。当加载一个被绑定的可执行文件时，Windows加载器可以跳过查找每个导入函数并覆盖IAT这一步。因为IAT中已经是正确的地址了。但是这只有正确对齐时才行。我在2000 年 5 月的Under the Hood专栏中讲了一些测试标准，你可以通过它们来确定绑定可执行文件能够对加载性能有多大提高。

你也许会怀疑将可执行文件绑定是否保险。你可能会想，如果绑定了可执行文件，但它导入的 DLL 发生了变化，这时怎么办呢？当这种情况发生时，IAT 中的地址已经失效了。加载器会检查这种情况并随机应变。如果 IAT 中的地址已经失效，加载器会根据 INT 中的信息重新解析导入函数的地址。

在安装程序时对其进行绑定应该是最可能发生的情况了。Windows Installer 中的 BindImage 这个动作可以替你做这件事。同样，IMAGEHLP.DLL 中也提供了 BindImageEx 这个 API。不管用哪一种方法，绑定都是个比较好的做法。如果加载器确定绑定信息是有效的，那么可执行文件就会被加载的更快。如果绑定信息失效，它也并不会比不绑定效果差。

对加载器来说，使绑定生效的一个关键步骤是确定 IAT 中的绑定信息是否有效。当可执行文件被绑定时，有关它导入的 DLL 的信息也被放在可执行文件中。加载器检查这个信息以快速确定绑定的有效性。在绑定的最初实现中并未添加这个信息，因此可执行文件可能按老的绑定方式进行绑定，或者按新的绑定方式进行绑定。我在这里讲的是新的绑定方式。

确定绑定信息有效性的一个关键数据结构是 IMAGE_BOUND_IMPORT_DESCRIPTOR。被绑定的可执行文件中有一个此结构的列表。每个 IMAGE_BOUND_IMPORT_DESCRIPTOR 结构表示一个绑定到的

DLL的日期/时间戳。这个列表的RVA由数据目录中索引为IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT的元素给出。IMAGE_BOUND_IMPORT_DESCRIPTOR结构中的成员如下：

- TimeDateStamp，这是包含导入的DLL的日期/时间戳的一个DWORD类型的值。

- OffsetModuleName，这是包含导入的DLL的名称字符串偏移地址的一个WORD类型的值。这个域是相对于首个IMAGE_BOUND_IMPORT_DESCRIPTOR结构的偏移(而不是RVA)。

- NumberOfModuleForwarderRefs，这是一个WORD类型的值，它包含紧跟在这个结构后面的IMAGE_BOUND_FORWARDER_REF结构的数目。除了最后一个WORD类型的成员（NumberOfModuleForwarderRefs）是保留的外，IMAGE_BOUND_FORWARDER_REF结构与IMAGE_BOUND_IMPORT_DESCRIPTOR结构一样。

一般情况下，每个导入的DLL对应的IMAGE_BOUND_IMPORT_DESCRIPTOR结构简单地组成一个数组。但是当绑定的API转发到了另一个DLL上时，这个转发到的DLL的有效性也需要检查。在这种情况下，IMAGE_BOUND_FORWARDER_REF结构就与IMAGE_BOUND_IMPORT_DESCRIPTOR结构交叉在了一起。下面举一个例子来说明。

假设你链接到了KERNEL32.DLL中的HeapAlloc这个API上，而它实际上被转发到了NTDLL中的RtlAllocateHeap上，然后你绑定这个可执行文件。那么在这个可执行文件中，对应于KERNEL32.DLL这个导入的DLL就有一个相应的IMAGE_BOUND_IMPORT_DESCRIPTOR结构，同时它后面是一个对应于NTDLL.DLL的IMAGE_BOUND_FORWARDER_REF结构。紧跟在它们后面的可能是与你导入并绑定到的其它DLL对应的IMAGE_BOUND_IMPORT_DESCRIPTOR结构。

延迟加载数据

前面我已经讲过延迟加载（Delayload）一个DLL就是隐含导入与通过LoadLibrary和GetProcAddress显式导入这两种方式的混合。现在让我们来看一下延迟加载所需的数据结构以及它的工作原理。

一定要记住延迟加载并不是操作系统的功能。它完全是由链接器和运行时库添加的附加代码和数据来实现的。正因为如此，WINNT.H中并没有几个地方涉及到延迟加载。但是你会发现延迟加载数据和正常导入数据二者的定义是平行的。

DataDirectory中的IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT元素指向延迟加载数据。这个元素中实际是一个ImgDelayDescr结构数组的RVA，这个结构被定义在Visual C++的DelayImp.H文件中。下表是这个结构的内容。对应于每一个导入的DLL都有一个相应的ImgDelayDescr结构。

大小	域	描述
DWORD	grAttrs	此结构的属性。当前惟一定义的标志是dlattrRva（值为1）。这个标志表明此结构中的地址域是RVA，而不是虚拟地址。设置这个标志表明延迟加载描述符是VC7.0或其后续版本。
RVA	rvaDLLName	导入的DLL的名称字符串的RVA。这个字符串被传递给LoadLibrary函数。

大小	域	描述
RVA	rvaHmod	一块 HMODULE 大小的内存的 RVA。当延迟加载的 DLL 被加载进内存时，它的 HMODULE 被存储在这个位置。
RVA	rvaIAT	此 DLL 的导入地址表的 RVA。它的格式与正常的 IAT 相同。
RVA	rvaINT	此 DLL 的导入名称表的 RVA。它的格式与正常的 INT 相同。
RVA	rvaBoundIAT	可选的绑定 IAT 的 RVA。它是此 DLL 的导入地址表的一个绑定副本的 RVA。它的格式与正常的 IAT 相同。当前这个 IAT 副本并未绑定，但这个功能可能被添加到将来的 BIND 程序中。
RVA	rvaUnloadIAT	原始的 IAT 的可选副本的 RVA。它是此 DLL 的导入地址表的一个未绑定的副本的 RVA。它的格式与正常的 IAT 相同。当前总是设置为 0。
DWORD	dwTimeStamp	延迟加载导入的 DLL 的日期/时间戳。通常设置为 0。

我们从 ImgDelayDescr 结构中可以获取的主要内容就是它包含了 DLL 的 IAT 和 INT 的地址。这些表与正常情况下的表是一样的，只不过它们是由运行时库代码进行读写而不是由操作系统。当你调用延迟加载的 DLL 中的函数时，运行时库代码就调用 LoadLibrary 加载相应的 DLL（如果需要的话），然后调用 GetProcAddress 来获取函数地址，最后将获取的地址存储在延迟加载 IAT 中，以便将来可以直接调用这个函数。

延迟加载所使用的数据结构在设计时有一个失误的地方需要解释一下。在 Visual C++ 6.0 中——这是它最初的形式，ImgDelayDescr 结构中的所有包含地址的域使用的都是虚拟地址，而不是 RVA。也就是说，它们包含了延迟加载数据所在位置的实际地址。这些域都是 DWORD 类型的，也就是 x86 上一个指针的大小。

现在要全面支持 IA-64 了。突然，4 字节已经不够保存一个完整的地址了。哎呀！在这个时候，Microsoft 做了一件正确的事，把包含地址的域都改为包含 RVA 了。如前面所示，我使用的是已经修订过的结构定义和名称。

还有一个问题就是确定 ImgDelayDescr 使用的是 RVA 还是虚拟地址。这个结构中有一个域包含了相关的标志。当 grAttrs 域为 1 时，这个结构中的成员中包含的是 RVA。从 Visual Studio® .NET 和 64 位编译器开始，这是惟一选项。如果 grAttrs 不是 1，ImgDelayDescr 结构中的域包含的都是虚拟地址。

资源节

在 PE 文件的所有节中，在资源节中定位数据是最复杂的。在这里我只讲述一些获取诸如图标、位图以及对话框之类的资源的原始数据所需的一些数据结构。我不涉及它们的实际格式，那已经超出了本文的范围。

资源可以在一个叫做.rsrc 的节中找到。DataDirectory 中索引为 IMAGE_DIRECTORY_ENTRY_RESOURCE 的元素包含了资源的 RVA 和大小。由于多方面的原因，资源被组织得与文件系统类似——有目录和叶结点。

DataDirectory 中的资源指针指向了一个 IMAGE_RESOURCE_DIRECTORY 类型的结构。这个结构中包含了目前尚未使用的 Characteristics 域、TimeDateStamp 域以及版本号域(MajorVersion 和 MinorVersion)。这个结构中真正有用的域是 NumberOfNamedEntries 和 NumberOfIdEntries。

每个 IMAGE_RESOURCE_DIRECTORY 结构后面是一个 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构数组。另外，IMAGE_RESOURCE_DIRECTORY 结构中的 NumberOfNamedEntries 和 NumberOfIdEntries 这两个域保存的就是这个数组中 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构的数目。（如果你感觉这些数据结构的名称让你看得头疼，说句实在话，我将它们写下来也挺难受的！）

每个目录项（即 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构）或者指向另一个资源目录，或者指向具体的资源数据。当它指向另一个资源目录时，这个结构中的第二个 DWORD 的最高位为 1，其余的 31 位是那个资源目录的偏移。这个偏移是相对于资源节开头来说的，而不是 RVA。

当它指向实际的某种资源时，第二个 DWORD 的最高位为 0，其余的 31 位是具体资源（例如对话框）的偏移。同上面一样，这个偏移同样是相对于资源节开头来说的，而不是 RVA。

每个目录项可以通过名称或者 ID 值来标识。它们就是你在.RC 文件中为具体资源指定的名称或 ID 值。当目录项的第一个 DWORD 的最高位为 1 时，其余的 31 位是资源名称（字符串）的偏移；如果最高位为 0，那么其低 16 位是资源标识（ID）的值。

理论已经足够了！现在让我们看一个实际的例子。下面是 PEDUMP 输出的 ADVAPI32.DLL 的资源节的部分内容：

```
Resources (RVA: 6B000)
ResDir (0) Entries:03 (Named:01, ID:02) TimeDate:00000000
    ---------------------------------------------------------------
    ResDir (MOFDATA) Entries:01 (Named:01, ID:00) TimeDate:00000000
        ResDir (MOFRESOURCENAME) Entries:01 (Named:00, ID:01) TimeDate:00000000
            ID: 00000409  DataEntryOffs: 00000128
            DataRVA: 6B6F0  DataSize: 190F5  CodePage: 0
    ---------------------------------------------------------------
    ResDir (STRING) Entries:01 (Named:00, ID:01) TimeDate:00000000
        ResDir (C36) Entries:01 (Named:00, ID:01) TimeDate:00000000
            ID: 00000409  DataEntryOffs: 00000138
            DataRVA: 6B1B0  DataSize: 0053C  CodePage: 0
    ---------------------------------------------------------------
    ResDir (RCDATA) Entries:01 (Named:00, ID:01) TimeDate:00000000
        ResDir (66) Entries:01 (Named:00, ID:01) TimeDate:00000000
            ID: 00000409  DataEntryOffs: 00000148
            DataRVA: 85908  DataSize: 0005C  CodePage: 0
```

其中以“ResDir”开头的每一行对应于一个 IMAGE_RESOURCE_DIRECTORY 结构。“ResDir“后面的括号中是资源目录的名称。在这个例子中，资源目录的名称分别为 0、MOFDATA、MOFRESOURCENAME、STRING、C36、RCDATA 和 66。名称后面是以名称标识的和以 ID 标识的资源目

录的总个数（后面的括号中是它们分别的个数）。在这个例子中，顶级目录一共有 3 个直接的子目录，所有其它目录都只有一个下级子目录。

顶级目录类似于文件系统中的根目录。根目录下的每个子目录项（也就是第二级目录）代表资源的类型（字符串表、对话框、菜单等等）。它们下面还有第三级子目录。

对于某种具体的资源类型来说，一般有三级目录。例如如果有五个对话框，那么第二级的 DIALOG 目录下面将会有五个子目录项。这五个子目录项本身也都是目录。在这五个目录下面都只有一项内容，它就是具体资源的原始数据的偏移地址。很简单，不是吗？

如果你更喜欢通过读源代码来学习的话，你可以仔细看一下 PEDUMP 中转储资源的那部分代码（PEDUMP 的源代码可以从 2002 年 2 月本文的第一部分中下载）。除了显示所有的资源目录以及它们的元素个数外，PEDUMP 还可以显示几种常见的资源类型，例如对话框等。

基址重定位

在可执行文件中的许多地方，你都会发现内存地址的踪迹。当链接器在生成可执行文件时，它假定这个可执行文件会被加载到内存中的某一个地址处（即首选地址）。只有在可执行文件被加载到其首选地址时，所有这些内存地址才是正确的。这个首选地址由 IMAGE_FILE_HEADER 结构中的 ImageBase 域给出。

如果加载器由于某种原因需要把可执行文件加载到其它地址处时，所有这些地址都变成不正确的了。这将会额外增加加载器的工作量。在 2000 年 5 月的 Under The Hood 专栏（前面已经提到）中我已经讲过当几个 DLL 首选加载地址相同时会导致性能损失，以及如何使用 REBASE 工具来解决这个问题。

基址重定位（Base Relocations）信息告诉加载器可执行文件不能被加载到其首选地址时需要进行修改的每一个位置。对于加载器来说，幸运的是它并不需要知道地址使用的细节问题。它只知道有一个地址列表，其中的每一个地址都需要以同样的方式进行修改。

让我们来看一个 x86 平台上的可执行文件的例子。假设有以下指令，它将一个全局变量（地址 0x0040D434）的值加载到 ECX 寄存器中：

```
00401020: 8B 0D 34 D4 40 00  mov ecx,dword ptr [0x0040D434]
```

这条指令在地址 0x00401020 处，长为 6 个字节。前两个字节（0x8B 0x0D）是指令的机器码。剩下的四个字节是一个 DWORD 值的地址（0x0040D434）。在这个例子中，这条指令实际来自一个首选地址为 0x00400000 的可执行文件，因此这个全局变量的 RVA 为 0xD434。

如果这个可执行文件被加载到了 0x00400000 处，这条指令当然可以正确执行。但是现在我们假设它被加载到了 0x00500000 处。如果真是这样，那么这条指令的最后的四个字节需要被改成 0x0050D434。

那么加载器是如何做的呢？它比较首选加载地址与实际加载地址，然后计算出△（delta，音译为德耳塔，数学中的常用符号，表示差值的意思）。在这个例子中，△为 0x00100000。这

个△被加到变量原来的地址值（大小为 DWORD）上，形成新的地址。在前面的例子中，关于地址 0x00401022 处，即指令中的 DWORD 值处，将会有一个相应的重定位信息。

简而言之，基址重定位信息只是可执行文件中的一个地址列表，当加载进内存时，这些地址中的值都要再加上△。为了提高系统性能，可执行文件的页面只有在需要时才会被加载进内存（可执行文件的加载与内存映射文件类似），基址重定位信息的格式就反映了这个特性。基址重定位信息所在的节通常被称为.reloc 节，但是查找它的正确方法是通过数据目录中索引为 IMAGE_DIRECTORY_ENTRY_BASERELOC 的那个元素。

基址重定位信息是一些非常简单的 IMAGE_BASE_RELOCATION 结构。此结构中的 VirtualAddress 域包含了需要进行重定位的内存范围的起始 RVA。SizeOfBlock 域给出了重定位信息的大小，其中包括 IMAGE_BASE_RELOCATION 自身的大小。

紧跟着 IMAGE_BASE_RELOCATION 结构后面是一组可变数目的 WORD 值。这些 WORD 值的数目可以从 IMAGE_BASE_RELOCATION 结构的 SizeOfBlock 域推出。其中每个 WORD 值由两部分组成。高 4 位指明了重定位的类型，由 WINNT.H 中的一系列 IMAGE_REL_BASED_*xxx* 值给出。低 12 位是相对于 IMAGE_BASE_RELOCATION 结构的 VirtualAddress 域的偏移，这是应该进行重定位的地方。

在前面那个关于基址重定位的例子中，我把情况简化了。实际上有多种类型的重定位方式。对于 x86 平台上的可执行文件来说，所有的重定位类型都是 IMAGE_REL_BASED_HIGHLOW。你经常会在一组重定位信息之后看到类型为 IMAGE_REL_BASED_ABSOLUTE 的重定位信息。它们实际上并没有什么作用，只是为了填充空间以便下一个 IMAGE_BASE_RELOCATION 结构能够按 4 字节的边界对齐。

对于 IA-64 平台上的可执行文件来说，重定位类型好像总是 IMAGE_REL_BASED_DIR64。与 x86 平台一样的是，通常也会有作为填充的 IMAGE_REL_BASED_ABSOLUTE 类型的重定位信息。有趣的一点是，尽管 IA-64 平台上每个页面是 8KB，但基址重定位信息仍旧是分成 4KB 的块。

在 Visual C++ 6.0 中，链接器在创建发行版的 EXE 文件时并不生成重定位信息。这是由于 EXE 文件是最先被加载到进程的地址空间中的，因此可以绝对保证它能被加载到其首选地址上。DLL 就没有这么幸运了，因此 DLL 中总是存在基址重定位信息，除非你使用/FIXED 链接器选项明确忽略它们。在 Visual Studio .NET 中，链接器在生成调试版和发行版的 EXE 文件时都不产生基址重定位信息。

调试目录

当创建可执行文件并生成相应的调试信息时，通常文件中会包含这种信息格式的细节以及它的位置。操作系统运行可执行文件时并不需要调试信息，但它对于开发工具非常有用。一个 EXE 文件可以包含多种格式的调试信息，调试目录（Debug Directory）结构指出哪种格式可用。

可以通过数据目录中索引为 IMAGE_DIRECTORY_ENTRY_DEBUG 的元素找到调试目录。它是由 IMAGE_DEBUG_DIRECTORY 结构组成的数组，其中每一个结构对应一种类型的调试信息，如下表所示。调试目录中元素的数目可以使用数据目录中的 Size 域计算得出。

大小	域	描述
DWORD	Characteristics	未用，设置为 0。

大小	域	描述
DWORD	TimeDateStamp	调试信息的日期/时间戳。
WORD	MajorVersion	调试信息的主版本号，未用。
WORD	MinorVersion	调试信息的次版本号，未用。
DWORD	Type	调试信息的类型。以下是经常遇到的类型： IMAGE_DEBUG_TYPE_COFF IMAGE_DEBUG_TYPE_CODEVIEW // 包含 PDB 文件 IMAGE_DEBUG_TYPE_FPO // 帧指针省略 IMAGE_DEBUG_TYPE_MISC // IMAGE_DEBUG_MISC IMAGE_DEBUG_TYPE_OMAP_TO_SRC IMAGE_DEBUG_TYPE_OMAP_FROM_SRC IMAGE_DEBUG_TYPE_BORLAND // Borland 格式
DWORD	SizeOfData	文件中调试数据的大小。不包括外部调试文件（例如.PDB 文件）的大小。
DWORD	AddressOfRawData	当映射进内存时调试数据的 RVA。如果调试信息不被映射，它被设置为 0。
DWORD	PointerToRawData	调试数据的文件偏移（不是 RVA）。

到目前为止，最流行的调试信息格式是 PDB 文件。PDB 文件实质上是 CodeView 格式调试信息的发展。一个类型为 IMAGE_DEBUG_TYPE_CODEVIEW 的调试目录标志着 PDB 信息的存在。如果你检查由这个元素指向的数据，会发现一个短的 CodeView 格式的头部。这个调试数据主要是一个外部 PDB 文件的路径。在 Visual Studio 6.0 中，调试头开始处是一个 NB10 签名。在 Visual Studio .NET 中，这个头开始处是 RSDS。

在 Visual Studio 6.0 中，可以使用/DEBUGTYPE:COFF 链接器选项来生成 COFF 调试信息。Visual Studio .NET 将这项功能移除了。对于经过优化的 x86 代码，由于函数可能没有正常的栈帧，所有使用帧指针省略（Frame Pointer Omission，FPO）调试信息。FPO 数据允许调试器定位局部变量和参数。

有两种 OMAP 调试信息仅用于 Microsoft 的程序。Microsoft 内部使用一种工具对可执行文件中的代码进行重新排列以减少分页。（它所做的不仅仅是 Working Set Tuner 所能做到的。）OMAP 信息让工具可以在调试信息中的原始地址与重排后的代码中的新地址之间进行转换。

顺便说一下，DBG 文件也包含了一个类似于我上面讲的调试目录。DBG 文件流行于 Windows NT 4.0 时代，它们主要包含 COFF 调试信息，但是 Windows XP 偏爱 PDB 文件而将它们淘汰了。

.NET 头部

对于开发工具生成的用于 Microsoft .NET 环境下的可执行文件来说，它们首先是 PE 文件。但是在大多数情况下.NET 文件中正常的代码和数据是微不足道的。.NET 可执行文件的主要目的是将.NET 特定的信息，例如元数据和中间语言（IL），加载进内存。另外.NET 可执行文件链接到了 MSCOREE.DLL 文件上。这个 DLL 是.NET 进程的起点。当加载.NET 可执行文件时，它的入口点通常是一个小的占位程序。这个占位程序只是跳转到 MSCOREE.DLL 的一个导出函数（_CorExeMain 或_CorDllMain）上。从那里开始，MSCOREE 获取控制权，开始使用可执行文件中的元数据和 IL。这类似于（.NET 版之前的）Visual Basic 中的应用程序使用 MSVBVM60.DLL 所采用的方式。.NET 信息的起点是 IMAGE_COR20_HEADER 结构，它当前被定义在.NET Framework SDK 中的 CorHDR.H 文件以及最新的 WINNT.H 文件中。数据目录中索引为 IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 的项指向 IMAGE_COR20_HEADER 结构。下表列出了 IMAGE_COR20_HEADER 结构中的域。关于 IMAGE_COR20_HEADER 指向的元数据、方法 IL 以及其它内容将在后续文件中详细讲述。

类型	域	描述
DWORD	cb	头部的大小（以字节计）。
WORD	MajorRuntimeVersion	运行这个程序所需的运行时组件的最小版本号。对于第一个发行的.NET Framework 而言，此值为 2。
WORD	MinorRuntimeVersion	次版本号，当前为 0。
IMAGE_DATA_DIRECTORY	MetaData	元数据表的 RVA。

DWORD	Flags	包含这个映像属性的标志。当前定义了以下值：
		COMIMAGE_FLAGS_ILONLY
		// 映像仅包含 IL 代码，并不需要运
		// 行于特定 CPU 上
		COMIMAGE_FLAGS_32BITREQUIRED
		// 仅运行于 32 位处理器上
		COMIMAGE_FLAGS_IL_LIBRARY
		STRONGNAMESIGNED
		// 映像已经用散列数据签名
		COMIMAGE_FLAGS_TRACKDEBUGDATA
		// 让 JIT 或运行时组件为方法保持
		// 调试信息

DWORD	EntryPointToken	映像入口点的MethodDef的记号。.NET 运行时调用这个方法开始托管执行。
IMAGE_DATA_DIRECTORY	Resources	.NET 资源的 RVA 和大小。
IMAGE_DATA_DIRECTORY	StrongNameSignature	强名称散列数据的 RVA。
IMAGE_DATA_DIRECTORY	CodeManagerTable	代码管理器表的 RVA。代码管理器包含获取正在运行的程序的状态（例如堆栈跟踪和跟踪 GC 引用）所需的代码。
IMAGE_DATA_DIRECTORY	VTableFixups	需要被修正的函数指针组成的数组。用于支持非托管的 C++虚表。
IMAGE_DATA_DIRECTORY	ExportAddressTableJumps	由对应于导出符号的 JMP 形实转换块被写入的位置（RVA）组成的数组的 RVA。这些形实转换块允许托管方法被导出，这样非托管代码可以调用它们。
IMAGE_DATA_DIRECTORY	ManagedNativeHeader	在内存中供.NET 运行时组件内部使用。在可执行文件中被设置为 0。

TLS 初始化

当使用__declspec(thread)定义线程局部变量时，编译器将它们放入一个名为.tls 的节中。当系统创建新线程时，它从进程堆中分配内存来保存用于新线程的线程局部变量。这部分内存使用.tls 节中的值进行初始化。系统将分配的内存的地址保存在 TLS 数组中，FS:[2Ch]指向这个数组（在 x86 平台上）。

如果数据目录中索引为 IMAGE_DIRECTORY_ENTRY_TLS 的元素不为 0，那就表示可执行文件中存在线程局部存储（TLS）。而这个元素指向一个 IMAGE_TLS_DIRECTORY 结构，如下表所示。

大小	域	描述
DWORD	StartAddressOfRawData	用于在内存中初始化新线程的 TLS 数据的一段内存的起始地址。
DWORD	EndAddressOfRawData	用于在内存中初始化新线程的 TLS 数据的一段内存的结束地址。
DWORD	AddressOfIndex	当可执行文件被加载进内存时，如果它包含.tls 节，加载器调用 TlsAlloc 给它分配一个 TLS 句柄，并将分配的句柄保存在这个域指定的位置处。运行时库使用这个句柄定位线程局部数据。
DWORD	AddressOfCallBacks	由 PIMAGE_TLS_CALLBACK 类型的函数指针组成的数组的地址。当创建或撤销线程时，这个列表中的每个函数都会被调用。最后一个元素的值为 0，它标志着表的结尾。一般由 Visual C++生成的可执行文件中这个表是空的。
DWORD	SizeOfZeroFill	已初始化数据中除了由 StartAddressOfRawData 和 EndAddressOfRawData 域组成的已初始化数据界限之外的大小（以字节计）。所有超出这个范围的用于单个线程的数据都被初始化为 0。
DWORD	Characteristics	保留，当前被设置为 0。

注意到 IMAGE_TLS_DIRECTORY 中的地址都是虚拟地址而不是 RVA 这一点很重要。因此如果可执行文件不能被加载到其首选加载地址时，它们都要进行基址重定位。同时，IMAGE_TLS_DIRECTORY 结构本身并不在.tls 节中，它位于.rdata 节中。

程序异常数据

一些平台（包括 IA-64）并不使用 x86 平台上的基于帧的异常处理，它们使用的是基于表的异常处理。在这种异常处理中有一个表，它包含了可能会被异常展开（unwinding）影响到的每一个函数的信息。这些信息主要包括每个函数的开始地址、结束地址以及在哪里并如何处理异常。当发生异常时，系统搜索整个表来寻找处理它的相应项并处理。异常表是一个由 IMAGE_RUNTIME_FUNCTION_ENTRY 结构组成的数组。数据目录中索引为 IMAGE_DIRECTORY_ENTRY_EXCEPTION 的元素引向此数组。这个结构的格式因平台而异。对于 IA-64 平台，它的结构如下：

```
DWORD BeginAddress;
DWORD EndAddress;
DWORD UnwindInfoAddress;
```

UnwindInfoAddress数据的结构并未在WINNT.H文件中给出。但是它的具体格式可以在Intel的"IA-64 Software Conventions and Runtime Architecture Guide"一书第 11 章中找到。

PEDUMP 程序

现在我的 PEDUMP 程序与 1994 年时的相比已经有了很大改进。它可以显示本文中讲的所有结构，其中包括：

- IMAGE_NT_HEADERS
- 导入表/导出表
- 资源
- 基址重定位
- 调试目录

- 延迟导入表
- 绑定导入描述符
- IA-64 异常处理表
- TLS 初始化数据
- .NET 运行时头

除了可以转储可执行文件外，PEDUMP 还可以转储 COFF 格式的 OBJ 文件、COFF 导入库（新格式以及老格式）、COFF 符号表和 DBG 文件。

PEDUMP 是一个命令行程序。对前面提到的各种文件运行 PEDUMP 时，如果不加任何选项，它默认输出的是最有用的数据结构信息。有好几个命令行选项可以用来添加其它的输出信息：

名称	描述
/A	转储所有内容
/B	显示基址重定位信息
/H	包括每个节中原始数据的十六进制形式
/I	包括导入地址表形实转换块的地址
/L	包括行号信息
/P	包括 PDATA（运行时函数）
/R	包括详细的资源信息（字符串表和对话框）
/S	显示符号表

关于 PEDUMP 的源代码有几个地方值得注意。首先它可以按 32 位或 64 位可执行文件编译和运行。如果你手边有 Itanium 机器可以试一下。另外，无论 PEDUMP 以何种方式编译，它都可以同时转储 32 位和 64 位文件。换句话说，32 位版的 PEDUMP 可以转储 32 位和 64 位文件，64 位版的 PEDUMP 也可以转储 32 位和 64 位文件。

在考虑使 PEDUMP 可以同时处理 32 位和 64 位文件时，我想避免为 32 位结构和 64 位结构分别写一个函数。因此我使用了 C++模板。

在好几个文件（特别是 EXEDUMP.CPP）中，你都会发现各种模板函数。大多数情况下，模板函数的参数最终会被扩展为 IMAGE_NT_HEADERS32 结构或 IMAGE_NT_HEADERS64 结构。当调用这些函数时，由代码自身确定是 32 位还是 64 位文件并用相应参数类型去调用相应的函数，引起相应的模板展开。

伴随 PEDUMP 源代码的还有一个 Visual C++ 6.0 工程文件。工程配置除了传统的 x86 debug 和 release 外，还有相应的 64 位配置。要想使它正常工作，你需要把 64 位工具（当前在 Platform SDK 中）的路径添加到 Tools | Options | Directories 选项卡最上面的 Executable files 路径中，同时还要设置相应的 64 位 Include 目录和 Lib 目录的路径。在我的机器上这个工程文件可以正常工作，但是在你的机器上可能需要进行少量修改才行。

为了使 PEDUMP 可以处理的内容尽可能全面，这就需要使用最新的 Windows 头文件。我在开发这个程序时使用的是 2001 年 6 月的 Platform SDK，需要的这些文件都在.\include\prerelease 和.\Include\Win64\crt 目录中。在 2001 年 8 月的 SDK 中， WINNT.H 文件已经被更新，因此也就不需要 prerelease 目录中的文件了。最终可以成功创建这个程序。你需要做的可能只是尽是安装最新的 Platform SDK 或在创建 64 位版的程序时对工程目录进行一些修改。

结束语

可移植可执行文件格式是一种结构非常好且相对简单的可执行文件格式。特别好的一点是 PE 文件可以被直接映射进内存，这样它在磁盘上的数据结构与运行时 Windows 使用的结构一致。

我同时非常惊奇于 PE 格式是如何经受住 10 多年来的各种变化，甚至包括移植到 64 位 Windows 以及.NET 平台上，对它的影响。（PE 格式的设计者竟然如此深谋远虑！）

尽管我讲了 PE 文件许多方面的内容，但是仍然还有一些主题我没有涉及到，其中包括一些标志、属性以及数据结构。我认为它们并不常用，因此也就没有在这里讲。但是我希望我在这里对 PE 文件的讲解能使你更容易理解 Microsoft 的 PE 规范。

（*MSDN Magazine* 2002 年 3 月Under The Hood专栏）

译者：SmartTech　　电子信箱：zhzhtst@163.com