

[博客专区](#) > [j\\_m的博客](#) > [j\\_m的博客详情](#)[免费上云](#)

## 转 PE文件格式”1.9版 完整译文

j\_m 发表于 5年前 阅读 99 收藏 2 点赞 0 评论 0

[收藏](#)

PE文件格式系列译文之一-----

【翻译】“PE文件格式”1.9版 完整译文（附注释）

=====

原著：Bernd. Luevelsmeyer

翻译：ah007

[注意：本译文的所有大小标题序号都是译者添加，以方便大家阅读。圆圈内的数字是注释的编号，其中注释②译自微软的《PECOFF规范》，其它译自网络。-----译者]

### 一、前言（Preface）

-----

PE（“portable executable”，可移植的可执行文件）文件格式，是微软WindwosNT,Windows95和Win32子集①中的可执行的二进制文件的格式；在WindowsNT中，驱动程序也是这种格式。它还能被应用于各种目标文件②和库文件中。

这种文件格式是由微软设计的，并于1993年被TIS（tool interface standard,工具接口标准）委员会（由

Microsoft, Intel, Borland, Watcom, IBM, 等等组成) 所批准, 它明显的基于COFF文件格式的许多知识。COFF ( “common object file format”, 通用目标文件格式 ) 是应用于好几种UNIX系统③和VMS④系统中的目标文件和可执行文件的格式。

Win32 SDK⑤中包含一个名叫<winnt.h>的头文件, 其中含有很多用于PE格式的#define和typedef定义。我将逐步地提到其中的很多结构成员名字和#define定义。

你也可能发现DLL文件“imagehelp.dll”很有用途, 它是WindowNT的一部分, 但其书面文件却很缺乏。它的一些功用在“Developer Network”( 开发者网络 ) 中有所描述。

## 二、总览 ( General Layout )

-----

在一个PE文件的开始处, 我们会看到一个MS-DOS可执行体 ( 英语叫“stub”, 意为“根, 存根” ); 它使任何PE文件都是一个有效的MS-DOS可执行文件。

在DOS-根之后是一个32位的签名以及魔数0x00004550 ( IMAGE\_NT\_SIGNATURE ) ( 意为“NT签名”, 也就是PE签名; 十六进制数45和50分别代表ASCII码字母E和P----译者注 ) 。

之后是文件头 ( 按COFF格式 ), 用来说明该二进制文件将运行在何种机器之上、分几个区段、链接的时间、是可执行文件还是DLL、等等。( 本文中可执行文件和DLL文件的区别在于: DLL文件不能被启动, 但能被别的二进制文件使用, 而一个二进制文件则不能链接到另一个可执行文件。 )

那些之后, 是可选头 ( 尽管它一直都存在, 却仍被称作“可选”----因为COFF文件格式仅为库文件使用一个“可选头”, 却不为目标文件使用一个“可选头”, 这就是为什么它被称为“可选”的原因 )。它会告诉我们该二进制文件怎样被载入的更多信息: 开始的地址呀、保留的堆栈数呀、数据段的大小呀、等等。

可选头的一个有趣的部分是尾部的“数据目录”数组；这些目录包含许多指向各“节”数据的指针。例如：如果一个二进制文件拥有一个输出目录，那么你就会在数组成员“IMAGE\_DIRECTORY\_ENTRY\_EXPORT”（输出目录项）中找到一个指向那个目录的指针，而该指针指向文件中的某节。

跟在各种头后面我们就发现各个“节”了，它们都由“节头”引导。本质上讲，各节中的内容才是你执行一个程序真正需要的东西，所有头和目录这些东西只是为了帮助你找到它们。

每节都含有和对齐、包含什么样的数据（如“已初始化数据”等等）、是否能共享等有关的一些标记，还有就是数据本身。大多数（并非所有）节都含有一个或多个可通过可选头的“数据目录”数组中的项来参见的目录，如输出函数目录和基址重定位目录等。无目录形式的内容有：例如“可执行代码”或“已初始化数据”等。

```
+-----+
| DOS-stub | --DOS-头
+-----+
| file-header | --文件头
+-----+
| optional header | --可选头
| - - - - - |
| |
| data directories | --数据目录
| |
+-----+
| |
| section headers | --节头
| |
+-----+
| |
| section 1 | --节1
| |
+-----+
```

```

| |
| section 2 | --节2
| |
+-----+
| |
| ... |
| |
+-----+
| |
| section n | --节n
| |
+-----+

```

### 三、DOS-根和签名 ( DOS-stub and Signature )

-----

DOS-根的概念很早从16位windows的可执行文件（当时是“NE”格式⑥）时就广为人知了。根原来是用于OS/2⑦系统的可执行文件的，也用于自解压档案文件和其它的应用程序。对于PE文件来说，它是一个总是由大约100个字节所组成的和MS-DOS 2.0兼容的可执行体，用来输出象“this program needs windows NT”之类的错误信息。

你可以通过确认DOS-头部分是否为一个IMAGE\_DOS\_HEADER（DOS头）结构来认出DOS-根，它的前两个字节必须为连续的两个字母“MZ”（有一个#define IMAGE\_DOS\_SIGNATURE的定义是针对这个WORD单元的）。

你可以通过跟在后面的签名来将一个PE二进制文件和其它含有根的二进制文件区分开来，跟在后面的签名可由头成员'e\_lfanew'（它是从字节偏移地址60处开始的，有32字节长）所设定的偏移地址找到。对于OS/2系统和

Windows系统的二进制文件来说，签名是一个16位的word单元；对于PE文件来说，它是一个按照8位字节边界对齐的32位的longword单元，并且IMAGE\_NT\_SIGNATURE（NT签名）的值已由#define定义为0x00004550（即字母“PE/0/0”-----译者）。

#### 四、文件头（File Header）

-----

要到达IMAGE\_FILE\_HEADER（文件头）结构，请先确认DOS-头“MZ”（起始的2个字节），然后找出DOS-根的头部的成员“e\_lfanew”，并从文件开始处跳过那么多的字节。在核实你在那里找到的签名后，IMAGE\_FILE\_HEADER（文件头）结构的文件头就紧跟其后开始了，下面我们将从头至尾的介绍其成员。

1）第一个成员是“Machine（机器）”，一个16位的值，用来指出该二进制文件预定运行于什么样的系统。已知的合法的值有：

IMAGE\_FILE\_MACHINE\_I386 (0x14c)

Intel 80386 处理器或更高

0x014d

Intel 80386 处理器或更高

0x014e

Intel 80386 处理器或更高

0x0160

R3000 (MIPS®)处理器，大尾⑨

IMAGE\_FILE\_MACHINE\_R3000 (0x162)

R3000 (MIPS)处理器, 小尾

IMAGE\_FILE\_MACHINE\_R4000 (0x166)

R4000 (MIPS)处理器, 小尾

IMAGE\_FILE\_MACHINE\_R10000 (0x168)

R10000 (MIPS)处理器, 小尾

IMAGE\_FILE\_MACHINE\_ALPHA (0x184)

DEC Alpha AXP@处理器

IMAGE\_FILE\_MACHINE\_POWERPC (0x1F0)

IBM Power PC,小尾

2) 然后是"NumberOfSections (节数)"成员, 16位的值。它是紧跟在头后面的节的数目。我们以后将讨论节的问题。

3) 下一个成员是时间戳"TimeStamp" (32位), 用来给出文件建立的时间。即使它的"官方"版本号没有改变, 你也可通过这个值来区分同一个文件的不同版本。(除了同一个文件的不同版本之间必须唯一, 时间戳的格式没有明文规定, 但似乎是按照UTC?时间"从1970年1月1日00:00:00算起的秒数值"----也就是大多数C语言编译器给time\_t标志使用的格式。)

这个时间戳是用来绑定各个输入目录的, 我们稍后再讨论它。

警告: 有一些链接器往往将时间戳设为荒唐的值, 而不是如前所述的time\_t格式的链接时间。

4-5) 成员"PointerToSymbolTable (符号表指针)"和成员"NumberOfSymbols (符号数)" (都是32位) 都用于调试信息的。我不知道该怎样去解读它, 并且我发现该指针的值总为0。

6) 成员"SizeOfOptionalHeader (可选头大小)" (16位) 只是"IMAGE\_OPTIONAL\_HEADER (可选头)"项的大小, 你能用它去验证PE文件结构的正确性。

7) 成员“Characteristics (特性)”是一个16位的, 由许多标志位形成的集合组成, 但大多数标志位只对目标文件和库文件有效。具体如下:

位0 IMAGE\_FILE\_RELOCS\_STRIPPED (重定位被剥离文件) 表示如果文件中没有重定位信息, 该位置1, 这就表明各节的重定位信息都在它们各自的节中; 可执行文件不使用该位, 它们的重定位信息放在下面将要描述的“base relocation”(基址重定位) 目录中。

位1 IMAGE\_FILE\_EXECUTABLE\_IMAGE (可执行映像文件) 表示如果文件是一个可执行文件, 也即不是目标文件或者库文件时, 置1。如果链接器尝试创建一个可执行文件, 却因为一些原因失败了, 并保存映像以便下次例如增量链接时使用, 此时此标志位也可能置1。

位2 IMAGE\_FILE\_LINE\_NUMS\_STRIPPED (行数被剥离文件) 表示如果行数信息被剥除, 此位置1; 此位也不用于可执行文件。

位3 IMAGE\_FILE\_LOCAL\_SYMS\_STRIPPED (本地符号被剥离文件) 表示如果文件中没有关于本地符号的信息时, 此位置1 (此位也不用于可执行文件)。

位4 IMAGE\_FILE\_AGGRESSIVE\_WS\_TRIM (强行工作集修剪文件) 表示如果操作系统被假定为: 通过将正在运行的进程 (它所使用的内存数量) 强行的页清除来修剪它的工作集时, 此位置1。如果一进程是大部分时间处于等待, 且一天中仅被唤醒一次的演示性的应用程序之类时, 此位也应该被置1。

位7 IMAGE\_FILE\_BYTES\_REVERSED\_LO (低字节变换文件) 和 位15 IMAGE\_FILE\_BYTES\_REVERSED\_HI (高字节变换文件) 表示如果一文件的字节序不是机器所预期的形式, 因此它在读入前必须调换字节时, 此位置1。这样做对可执行文件是不可靠的 (操作系统期望可执行文件都已经被正确地按字节排整齐了)。

位8 IMAGE\_FILE\_32BIT\_MACHINE (32位机器文件) 表示如果使用的机器被期望为32位的机器时, 此位置1。现在的应用程序总将此位置1; NT5系统可能工作不同。

位9 IMAGE\_FILE\_DEBUG\_STRIPPED ( 调试信息被剥离文件) 表示如果文件中没有调试信息, 此位置1。此位可执行文件不用。按照其它信息([6]) ( 这里指的是参考书目中的第[6]种----译者注 ), 此位被称作“恒定”, 并且当一个映象文件只有在被装入优先的装入地址才能运行 ( 亦即: 此文件不可重定位 ) 时, 此位置1。

位10 IMAGE\_FILE\_REMOVABLE\_RUN\_FROM\_SWAP ( 移动介质文件从交换文件运行) 表示如果一个应用程序不可以从可移动的介质, 如软盘或CD-ROM上运行时, 此位置1。在这种情况下, 建议操作系统将文件复制到交换文件并从那里执行。

位11 IMAGE\_FILE\_NET\_RUN\_FROM\_SWAP ( 网络文件从交换文件运行) 表示如果一个应用程序不可以从网络上运行时, 此位置1。在这种情况下, 建议操作系统将文件复制到交换文件并从那里执行。

位12 IMAGE\_FILE\_SYSTEM ( 系统文件) 表示如果文件是一个象驱动程序那样的系统文件, 此位置1。此位可执行文件不用; 我所见过的所有NT系统的驱动程序也不用。

位13 IMAGE\_FILE\_DLL ( DLL文件) 表示如果文件是一个DLL文件时, 此位置1。

位14 IMAGE\_FILE\_UP\_SYSTEM\_ONLY ( 仅但处理器系统的文件) 表示如果文件不设计运行在多处理器系统上 ( 也就是说, 因为此文件严格地依赖单一处理器的一些方式工作, 所以它会发生冲突 ) 时, 此位置1。

## 五、相对虚拟地址 ( Relative Virtual Addresses )

-----

PE格式大量地使用所谓的RVA ( 相对虚拟地址 )。一个RVA, 亦即一个“Relative Virtual Addresses ( 相对虚拟地址 )”, 是在你不知道基地址时, 被用来描述一个内存地址的。它是需要加上基地址才能获得线性地址的数值。基地址就是PE映象文件被装入内存的地址, 并且可能会随着一次又一次的调用而变化。



例如：假若一个可执行文件被装入的地址是0x400000，并且从RVA 0x1560处开始执行，那么有效的执行开始处将位于0x401560地址处。假若它被装入的地址为0x100000，那么执行开始处就位于0x101560地址处。

因为PE-文件中的各部分（各节）不需要像已载入的映象文件那样对齐，事情变得复杂起来。例如，文件中的各节常按照512（十六进制的0x200----译者注）字节边界对齐，而已载入的映象文件则可能按照4096（十六进制的0x1000----译者注）字节边界对齐。参见下面的“SectionAlignment（节对齐）”和“FileAlignment（文件对齐）”。

因此，为了在PE文件中找到一个特定RVA地址的信息，你得按照文件已被载入时的那样来计算偏移量，但要按照文件的偏移量来跳过。

试举一例，假若你已知道执行开始处位于RVA 0x1560地址处，并且想从那里开始的代码处反汇编。为了从文件中找到这个地址，你得先查明在RAM（内存）中各节是按照4096字节对齐的，并且“.code”节是从RVA 0x1000地址处开始，有16384字节长；然后你才知道RVA 0x1560地址位于此节的偏移量0x560处。你还要查明在文件中那节是按512字节边界对齐，且“.code”节在文件中从偏移量0x800处开始，然后你就知道在文件中代码的执行开始处就在 $0x800 + 0x560 = 0xd60$ 字节处。

然后你反汇编它并发现访问一个变量的线性地址位于0x1051d0处。二进制文件的线性地址在装入时将被重定位，并常被假定使用的是优先载入地址。因为你已查明优先载入地址为0x100000，因此我们可开始处理RVA 0x51d0了。因数据节开始于RVA 0x5000处，且有2048字节长，所以它处于数据节中。又因数据节在文件中开始于偏移量0x4800处，所以该变量就可以在文件中的 $0x4800 + 0x51d0 - 0x5000 = 0x49d0$ 处找到。

## 六、可选头（Optional Header）

-----

紧跟在文件头后面的就是IMAGE\_OPTIONAL\_HEADER（尽管它名叫“可选头”，它却一直都在那里）。它包含有怎样去准确处理PE文件的信息。我们也将从头至尾的介绍其成员。

1）第一个16位的word单元叫“Magic（魔数）”，就我目前所观察过的PE文件而言，它的值总是0x010b。

2-3）下面2个字节是创建此文件的链接器的版本（‘MajorLinkerVersion’，“链接器主版本号”和‘MinorLinkerVersion’，“链接器小版本号”）。这两个值又是不可靠的，并不能总是正确地反映链接器的版本号。（有好几个链接器根本就不设置这个值。）况且，你可想象一下，你连使用的是“什么”链接器都不知道，知道它的版本号又有什么作用呢？

4-6）下面3个longword（每个32位）分别用来设定可执行代码的大小（“SizeOfCode”）、已初始化数据的大小（“SizeOfInitializedData”，所谓的“数据段”）、以及未初始化数据的大小（“SizeOfUninitializedData”，所谓的“bss?段”）。这些值也是不可靠的（例如：数据段实际上可能会被编译器或者链接器分成好几段），并且你可以通过查看可选头后面的各个“节”来获得更准确的大小。

7）下一个32位值是RVA。这个RVA是代码入口点的偏移量（‘AddressOfEntryPoint’，“入口点地址”）。执行将从这里开始，它可以是：例如DLL文件的LibMain()的地址，或者一个程序的开始代码（这里相应的叫main()）的地址，或者驱动程序的DriverEntry()的地址。如果你敢于“手工”装载映象文件，那么在你完成所有的修正和重定位后，你可以从这个地址开始执行你的进程。

8-9）下两个32位值分别是可执行代码的偏移值（‘BaseOfCode’，“代码基址”）和已初始化数据的偏移值（‘BaseOfData’，“数据基址”），两个都是RVA，并且两个对我们来说都没有多少意义，因为你可以通过查看可选头后面的各个“节”来获得更可靠的信息。

未初始化的数据没有偏移量，正因为它没有初始化，所以在映象文件中提供这些数据是没有用处的。

10）下一项是个32位值，提供整个二进制文件包括所有头的优先（线性）载入地址（‘ImageBase’，“映象文件基址”）。这是一个文件已被链接器重定位后的地址（总是64 KB的倍数）。如果二进制文件事实上能被载入这个地址，那么加载器就不用再重定位文件了，也就节省了一些载入时间。

优先载入地址在另一个映象文件已被先载入那个地址（“地址冲突”，在当你载入好几个全部按照链接器的缺省值重定位的DLL文件时经常发生）时，或者该内存已被用于其它目的（堆栈、malloc()、未初始化数据、或不管是什么）时，就不能用了。在这些情况下，映象文件必须被载入其它的地址，并且需要重定位（参见下面的“重定位目录”）。如果是一个DLL文件，这么做还会产生其它问题，因为此时的“绑定输入”已不再有效，所以使用DLL的二进制文件必须被修正----参见下面的“输入目录”一节。

11-12 ) 下两个32位值分别是RAM中的“SectionAlignment”（当映象文件已被载入后，意为“节对齐”）和文件中的“FileAlignment”（文件对齐），它们都是PE文件的各节的对齐值。这两个值通常都是32，或者是：FileAlignment为512，SectionAlignment为4096。节会在以后讨论。

13-14 ) 下2个16位word单元都是预期的操作系统版本信息（MajorOperatingSystemVersion，“操作系统主版本号”）和（MinorOperatingSystemVersion，“操作系统小版本号”）[它们都使用微软自己书面确定的名字]。这个版本信息应该为操作系统的版本号（如NT 或 Win95），而不是子系统的版本信息（如Win32）。版本信息常常被不提供或者错误提供。很明显的，加载器并不使用它们。

15-16 ) 下2个16位word单元都是本二进制文件的版本信息('MajorImageVersion'“映象文件主版本号”和'MinorImageVersion'“映象文件小版本号”)。很多链接器不正确地设定这个信息，许多程序员也懒得提供这些，因此即便存在这样的信息，你最好也不要信赖它。

17-18 ) 下2个16位word单元都是预期的子系统版本信息('MajorSubsystemVersion'“子系统主版本号”和'MinorSubsystemVersion'“子系统小版本号”)。此信息应该为Win32或POSIX的版本信息，因为很明显的，16位程序或OS/2程序都不是PE格式的。子系统版本应该被正确的提供，因为它“会”被检验和使用：如果一个应用程序是一个Win32-GUI应用程序并运行于NT4系统之上，而且子系统版本“不是”4.0的话，那么对话框就不会是以3D形式显示，并且一些其它的特征也只会按“老式”的方式工作，因为此应用程序预期是在NT 3.51系统上运行的，而NT 3.51系统上只有程序管理器而没有浏览器、等等，于是NT 4.0系统就尽可能地仿照那个系统的行为来运行程序。

19 ) 然后，我们便碰到32位的“Win32VersionValue”（Win32版本值）。我不清楚它有什么作用。在我所观察过的PE文件中，它全部都为0。

20) 下一个是32位值，给出映象文件将要使用的内存数量，单位为字节（'SizeOfImage'，“映象文件大小”）。如果是按照“SectionAlignment”对齐的，它就是所有头和节的长度的总和。它提示加载器，为了载入映象文件需要多少页。

21) 下一个是32位值，给出所有头的总长度，包括数据目录和节头（'SizeOfHeaders'，“头的大小”）。同时，它也是从文件的开头到第一节的数据的偏移量。

22) 然后，我们发现一个32位的校验和（“Checksum”）。这个校验和，对于当前的NT版本，只在映象文件是NT驱动程序时才校验（如果校验和不正确，驱动就将装载失败）。对于其他的二进制文件形式，校验和不需要并且可能为0。计算校验和的算法是微软的私产，他们不会告诉你的。但是，Win32 SDK的好几个工具都会计算和/或补正一个有效的校验和，而且imagehelp.dll中的ChecksumMappedFile()函数也会做同样的工作。使用校验和的目的是为了防止载入无论如何都会冲突的、已损坏的二进制文件——况且一个冲突的驱动程序会导致一个BSOD?错误，因此最好根本就不载入这样的坏文件。

23) 然后，就到了一个16位的word单元“Subsystem”（子系统），用来说明映象文件应运行于什么样的NT子系统之上：

IMAGE\_SUBSYSTEM\_NATIVE (1)

二进制文件不需要子系统。用于驱动程序。

IMAGE\_SUBSYSTEM\_WINDOWS\_GUI (2)

映象文件是一个Win32二进制图象文件。（它还是能用AllocConsole()打开一个控制台界面，但在开始时却不能自动地打开。）

IMAGE\_SUBSYSTEM\_WINDOWS\_CUI (3)

二进制文件是一个Win32控制台界面二进制文件。（它将在开始时按照缺省值打开一个控制台，或者继承其父程序的控制台。）

IMAGE\_SUBSYSTEM\_OS2\_CUI (5)

二进制文件是一个OS/2控制台界面二进制文件。（OS/2控制台界面二进制文件是OS/2格式，因此此值在PE文件中很少使用。）

IMAGE\_SUBSYSTEM\_POSIX\_CUI (7)

二进制文件使用POSIX?控制台子系统。

Windows 95的二进制文件总是使用Win32子系统，因此它的二进制文件的合法值只有2和3；我不知道windows 95的“原”二进制文件是否可能（会有其它值----译者添加，仅供参考）。

24) 下一个是16位的值，指明，如果是DLL文件，何时调用DLL文件的入口点（‘DllCharacteristics’，“DLL特性”）。此值似乎不用；很明显地，DLL文件总是被通报所有的情况。

如果位0被置1，DLL文件被通知进程附加（亦即DLL载入）。

如果位1被置1，DLL文件被通知线程附加（亦即线程终止）。

如果位2被置1，DLL文件被通知线程附加（亦即线程创建）。

如果位3被置1，DLL文件被通知进程附加（亦即DLL卸载）。

25-28) 下4个32位值分别是：保留栈的大小（SizeOfStackReserve）、初始时指定栈大小（SizeOfStackCommit）、保留堆的大小（SizeOfHeapReserve）和指定堆大小（SizeOfHeapCommit）。 “保留的”数量是保留给特定目的的地址空间（不是真正的RAM）；在程序开始时，“指定的”数量是指在RAM中实际分配的大小。如果需要的话，“指定的”值也是指定的堆或栈用来增加的数量。（有资料说，不管“SizeOfStackCommit”的值是多少，栈都是按页增加的。我没有验证过。）

因此，举例来说，如一个程序的保留堆有1 MB，指定堆为64 KB,那么启动时堆的大小为64 KB,并且保证可以扩大到1 MB。堆将按64 KB一块来增加。

“堆”在本文中是指主要（缺省）堆。如果它愿意的话，一个进程可创建很多堆。

栈是指第一个线程的栈（启动main()的那个）。进程可以创建很多线程，每个线程都有自己的栈。

DLL文件没有自己的堆或栈，所以它们的映象文件忽略这些值。我不知道驱动程序是否有它们自己的堆或栈，但我认为它们没有。

29) 堆和栈的这些描述之后，我们就发现一个32位的“LoaderFlags ( 加载器标志 )”，我没有找到它的任何有用的描述。我只发现一篇时新的关于设置此标志位的短文，说设置此标志位会在映象文件载入后自动地调用一个断点或者调试器；可似乎不正确。

30) 接着我们会发现32位的“NumberOfRvaAndSizes ( Rva数和大小 )”，它是紧随其后的目录的有效项的数目。我已发现此值不可靠；你也许希望用常量IMAGE\_NUMBEROF\_DIRECTORY\_ENTRIES ( 映象文件目录项数目 ) 来代替它，或者用它们中的较小者。

NumberOfRvaAndSizes之后是一个IMAGE\_NUMBEROF\_DIRECTORY\_ENTRIES (16) ( 映象文件目录项数目 ) 个IMAGE\_DATA\_DIRECTORY ( 映象文件数据目录 ) 数组。这些目录中的每一个目录都描述了一个特定的、位于目录项后面的某一节中的信息的位置 ( 32位的RVA，叫“VirtualAddress ( 虚拟地址 )”) 和大小 ( 也是32位，叫“Size ( 大小 )”)。

例如，安全目录能在索引4中给定的RVA处发现并具有索引4中给定的大小。

稍后我将讨论我知道其结构的目录。

已定义的目录及索引有：

IMAGE\_DIRECTORY\_ENTRY\_EXPORT (0)

输出符号目录；大多用于DLL文件。

后面介绍。

IMAGE\_DIRECTORY\_ENTRY\_IMPORT (1)

输入符号目录；参见后面。

IMAGE\_DIRECTORY\_ENTRY\_RESOURCE (2)

资源目录。后面介绍。

IMAGE\_DIRECTORY\_ENTRY\_EXCEPTION (3)

异常目录 - 结构和用途不详。

#### IMAGE\_DIRECTORY\_ENTRY\_SECURITY (4)

安全目录 - 结构和用途不详。

#### IMAGE\_DIRECTORY\_ENTRY\_BASERELOC (5)

基址重定位表 - 参见后面。

#### IMAGE\_DIRECTORY\_ENTRY\_DEBUG (6)

调试目录 - 内容编译器相关。此外, 许多编译器将编译信息填入代码节, 并不为此创建一个单独的节。

#### IMAGE\_DIRECTORY\_ENTRY\_COPYRIGHT (7)

描述字符串 - 一些随意的版权信息之类。

#### IMAGE\_DIRECTORY\_ENTRY\_GLOBALPTR (8)

机器值 (MIPS GP) - 结构和用途不详。

#### IMAGE\_DIRECTORY\_ENTRY\_TLS (9)

线程级局部存储目录 - 结构不详; 包含声明为"\_\_declspec(thread)"的变量, 也就是每线程的全局变量。

#### IMAGE\_DIRECTORY\_ENTRY\_LOAD\_CONFIG (10)

载入配置目录 - 结构和用途不详。

#### IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT (11)

绑定输入目录 - 参见输入目录的描述。

#### IMAGE\_DIRECTORY\_ENTRY\_IAT (12)

输入地址表 - 参见输入目录的描述。

试举一例, 如果我们在索引7中发现2个longword: 0x12000 和 33, 并且载入地址为0x10000, 那么我们就知道版权信息数据位于地址0x10000+0x12000 (在哪个节都有可能) 处, 并且版权信息有33字节长。



如果二进制文件中没有使用特殊类型的目录，Size（大小）和VirtualAddress（虚拟地址）的值就都为0。

## 七、节目录（Section directories）

-----

节由两个主要部分组成：首先，是一个节描述(IMAGE\_SECTION\_HEADER[意为“节头”]类型的)，然后是原始的节数据。因此，我们会在数据目录后发现一“NumberOfSections”个节头组成的数组，它们按照各节的RVA排序。

节头包括：

1) 一个IMAGE\_SIZEOF\_SHORT\_NAME (8) (意为“短名的大小”) 个字节的数组，形成节的名称 (ASCII形式)。如果所有的8位都被用光，该字符串就没有0结束符！典型的名称象“.data”或“.text”或“.bss”形式。开头的“.”不是必须的，名称也可能为“CODE”或“IAT”或类似的形式。

请注意：并不是所有的名称都和节中的内容相关。一个名叫“.code”的节可能包含也可能不包含可执行代码；它还可能只包含输入地址表；它也可能包含代码“和”地址表”和“未初始化数据。要找到节中的信息，你必须通过可选头的数据目录来查询它。既不要过分相信它的名称，也不要以为节的原始数据会从节的开头就开始。

2) IMAGE\_SECTION\_HEADER (“节头”) 的下一个成员是一个32位的、“PhysicalAddress (物理地址)”和“VirtualSize (虚拟大小)”组成的共用体。在目标文件中，它是内容重定位到的地址；在可执行文件中，它是内容的大小。事实上，此域似乎没被使用；因为有的链接器输入大小，有的链接器输入地址，我还发现有一个链接器输入0，而所有的可执行文件都运行如风。

3) 下一个成员是“VirtualAddress (虚拟地址)”，是一个32位的值，用来保存载入RAM (内存) 后，节中数据的RVA。

4) 然后，我们到了32位的“SizeOfRawData” (意味“原始数据大小”)，它表示节中数据被大约到下一个“FileAlignment”的整数倍时节的大小。



5) 下一个是“PointerToRawData”(意味“原始数据指针”，32位)，它特别有用，因为它是从文件的开头到节中数据的偏移量。如果它为0，那么节的数据就不包含在文件中，并且要在载入时才定。

6-9) 然后，我们得到“PointerToRelocations”(意味“重定位指针”，32位)和“PointerToLinenumbers”(意味“行数指针”，也是32位)，以及“NumberOfRelocations”(意味“重定位数”，16位)和“NumberOfLinenumbers”(意味“行数数”，也是16位)。所以这些都是只用于目标文件的信息。可执行文件拥有一个特殊的基址重定位目录，并且行数信息(如果真的存在的话)通常包含在有一个特殊目的的调试段中或者别的什么地方。

10) 节头的最后一个成员是32位的“Characteristics”(意味“特性”)，它是一串描述节的内存如何被处理的标志：

如果位5 IMAGE\_SCN\_CNT\_CODE (含有代码的节)被置1，表示节中包含可执行代码。

如果位6 IMAGE\_SCN\_CNT\_INITIALIZED\_DATA (含有初始化数据的节)被置1，表示节中包含执行开始前即取得已定义值的数据。换言之：文件中节的数据就是有意义的。

如果位7 IMAGE\_SCN\_CNT\_UNINITIALIZED\_DATA (含有未初始化数据的节)被置1，表示节中包含未初始化数据，并需于执行开始前被初始化为全0。这通常是BSS节。

如果位9 IMAGE\_SCN\_LNK\_INFO (链接器信息节)被置1，表示节中不包含映象数据，只有一些注释、描述或者其他的文档。这些信息是目标文件的一部分，并有可能是提供给链接器的信息，比如需要哪些库文件。

如果位11 IMAGE\_SCN\_LNK\_REMOVE (链接可删除节)被置1，表示数据是目标文件的、被预定于可执行文件被链接后丢弃掉的节的一部分。常和位9连用。

如果位12 IMAGE\_SCN\_LNK\_COMDAT (链接通用块节)被置1，表示节中包含“common block data”(通用块数据)，也即某种形式的打包函数。

如果位15 IMAGE\_SCN\_MEM\_FARDATA (内存远程数据节)被置1, 表示我们拥有远程数据----意味着什么。此位的含义不明。

如果位17 IMAGE\_SCN\_MEM\_PURGEABLE (内存可清除节)被置1, 表示节中的数据可清除----但我认为它和“可丢弃”不是一回事, 可丢弃拥有自己的标志位, 参见后面。同样, 它也明显的不是用来指示16位信息的, 因为它也有一个IMAGE\_SCN\_MEM\_16BIT定义。此位的含义不明。

如果位18 IMAGE\_SCN\_MEM\_LOCKED (内存被锁节)被置1, 表示节不应该被从内存中移除? 抑或表明没有重定位信息? 此位的含义不明。

如果位19 IMAGE\_SCN\_MEM\_PRELOAD (内存预载入节)被置1, 表示节在执行开始前应该被页载入? 此位的含义不明。

位20至23 指定我没有找到信息的对齐。诸如#define IMAGE\_SCN\_ALIGN\_16BYTES之类。我曾经见过的唯一值为0, 是16位的缺省对齐。我怀疑它们是库之类文件的目标对齐。

如果位24 IMAGE\_SCN\_LNK\_NRELOC\_OVFL (链接扩展重定位节)被置1, 表示节中包含一些我不知道的扩展重定位。

如果位25 IMAGE\_SCN\_MEM\_DISCARDABLE (内存可丢弃节)被置1, 表示节中的数据在进程启动后就不需要了。它是, 举例来说, 含有重定位信息的情况。我曾经见过它也用于只执行一次的驱动和服务程序的启动例程, 还用于输入目录。

如果位26 IMAGE\_SCN\_MEM\_NOT\_CACHED (内存不缓存节)被置1, 表示节中的数据不应该被缓存。不要问我为什么不。这是不是意味着关掉2级缓存?

如果位27 IMAGE\_SCN\_MEM\_NOT\_PAGED (内存不可页换出节)被置1, 表示节中的数据不应该页换出。它对驱动程序有意义。

如果位28 IMAGE\_SCN\_MEM\_SHARED ( 内存共享节)被置1, 表示节中的数据在映象文件的所有正在运行的实例中共享。如果它是, 例如DLL文件的未初始化数据, 那么DLL的所有正在运行的实例程序在任何时候都将拥有相同的变量内容。

注意: 只有第一个实例的节被初始化。

含有代码的节总是被共享写时拷贝 ( copy-on-write ) ( 亦即: 如果重定位必不可少, 那么共享就不工作 )。( 译注: “写时拷贝”的译法也许根本就是错误的, 但我一时找不到更准确的翻译, 也不清楚其具体含义, 只能以此充数了。希望知情着指点。 )

如果位29 IMAGE\_SCN\_MEM\_EXECUTE ( 内存可执行节)被置1, 表示进程对节的内存有“执行”的存取权限。

如果位30 IMAGE\_SCN\_MEM\_READ ( 内存可读节)被置1, 表示进程对节的内存有“读”的存取权限。

如果位31 IMAGE\_SCN\_MEM\_WRITE ( 内存可写节)被置1, 表示进程对节的内存有“写”的存取权限。

在节头之后, 我们就会发现节本身。在文件中, 它们按照“FileAlignment”( 文件对齐 ) 的字节数对齐 ( 也就是说, 在可选头之后和每个节的数据之后将要填充一些字节 ) 并按照它们的RVA排序。在载入后 ( 内存中 ), 它们按照“SectionAlignment”( 节对齐 ) 的字节数对齐。

试举一例, 如果可选头在文件的偏移量981处结束, “FileAlignment”( 文件对齐 ) 的值为512, 那么第一个节将于1024字节处开始。注意: 你可通过“PointerToRawData”( 原始数据指针 ) 或者“VirtualAddress”( 虚拟地址 ) 的值来找到各节, 因此实际上根本没必要在对齐上小题大做。

试画映象文件的全图如下:

```
+-----+
| DOS-根 |
+-----+
| 文件头 |
+-----+
```

```

| 可选头 |
|-----|
| |-----+
| 数据目录 | | |
| | |
| (指向节中 |-----+ |
| 目录的RVA) | | |
| |-----+ | |
| | | |
+-----+ | | |
| |-----+ | | |
| 节头 | | | |
| (指向节 |--+ | | |
| 边界的RVA) | | | |
+-----+<--+ | | |
| | | <--+ | |
| 节数据 1 | | | |
| | | <-----+ |
+-----+<-----+ |
| | |
| 节数据 2 | |
| | <-----+
+-----+

```

每个节都有一个节头，并且每个数据目录都会指向其中的一个节（几个数据目录有可能指向同一个节，而且也可能有的节没有数据目录指向它们）。

## 八、节的原始数据 ( Sections' raw data )

-----

### 1.概述 ( general )

-----

所有的节在载入内存后都按“SectionAlignment”（节对齐）对齐，在文件中则以“FileAlignment”（文件对齐）对齐。节由节头中的相关项来描述：在文件中你可通过“PointerToRawData”（原始数据指针）来找到，在内存中你可通过“VirtualAddress”（虚拟地址）来找到；长度由“SizeOfRawData”（原始数据长度）决定。

根据节中包含的内容，可分为好几种节。大多数（并非所有）情况下，节中至少由一个数据目录，并在可选头的数据目录数组中有一个指针指向它。

### 2.代码节 ( code section )

-----

首先，我将提到代码节。此节，至少，要将“IMAGE\_SCN\_CNT\_CODE”（含有代码节）、“IMAGE\_SCN\_MEM\_EXECUTE”（内存可执行节）和“IMAGE\_SCN\_MEM\_READ”（内存可读节）等标志位设为1，并且“AddressOfEntryPoint”（入口点地址）将指向节中的某个地方，指向开发者希望首先执行的那个函数的开始处。

“BaseOfCode”（代码基址）通常指向这一节的开始处，但是，如果一些非代码字节被放在代码之前的话，它也可能指向节中靠后的某个地方。

通常，除了可执行代码外，本节没有别的东东，并且通常只有一个代码节，但是不要太迷信这一点。

典型的节名有“.text”、“.code”、“AUTO”之类。

### 3.数据节 ( data section )

-----

我们要讨论的下一件事情就是已初始化变量；本节包含的是已初始化的静态变量（象“static int i = 5;”）。它

将，至少，使“IMAGE\_SCN\_CNT\_INITIALIZED\_DATA”（含有已初始化数据节）、“IMAGE\_SCN\_MEM\_READ”（内存可读节）和“IMAGE\_SCN\_MEM\_WRITE”（内存可写节）等标志位被置为1。

一些链接器可能会将常量放在没有可写标志位的它们自己的节中。如果有一部分数据可共享，或者有其它的特定情况，那么可能会有更多的节，且它们的合适的标志位会被设置。

不管是一节，还是多节，它们都将处于从“BaseOfData”（数据基址）

到“BaseOfData”+“SizeOfInitializedData”（数据基址+已初始化数据的大小）的范围之内。

典型的名称有“.data”、“.idata”、“DATA”、等等。

#### 4.BSS节（bss section）

-----

其后就是未初始化的数据（一些象“static int k;”之类的静态变量）；本节十分象已初始化的数据，但它的“PointerToRawData”（文件偏移量）却为0，表明它的内容不存储在文件中；并

且“IMAGE\_SCN\_CNT\_UNINITIALIZED\_DATA”（含有未初始化数据节）而不是

“IMAGE\_SCN\_CNT\_INITIALIZED\_DATA”（含有已初始化数据节）标志位被置为1，表明在载入时它的内容应该被置为0。这就意味着，在文件中只有节头，没有节身；节身将由加载器创建，并全部为0字节。

它的长度由“SizeOfUninitializedData”（未初始化数据大小）确定。

典型的名称有“.bss”、“BSS”之类。

有些节数据“没有”被数据目录指向。它们的内容和结构是由编译器而不是链接器提供。

（栈段和堆段不是二进制文件中的节，它们是由加载器根据可选头中的栈大小和堆大小项来创建的。）

#### 5.版权（copyright）

-----

为了从一个简单的目录节开始讲解，让我们来看一看数据目

录“IMAGE\_DIRECTORY\_ENTRY\_COPYRIGHT”（版权目录项）项。它的内容是一个版权信息或ASCII形式的描述字符串（不是以0结尾的），象“Gonkulator control application, copyright (c) 1848 Hugendubel &

Cie"这样。这个字符串，通常，是用命令行或者描述文件提供给链接器的。

这个字符串在运行时并不需要，并可能被丢弃。它是不可写的；事实上，应用程序根本不需要存取它。因此，如果已有一个可丢弃的、非可写的节存在，链接器就会找到它；如果没有，就创建一个（命名为".descr"之类）。然后它就将那个字符串填入该节中并让版权目录项指针指向这个字符串。

串。"IMAGE\_SCN\_CNT\_INITIALIZED\_DATA"（含有已初始化数据节）标志位应置为1。

## 6.输出符号（exported symbols）

-----

（注意：本文的1993年03月12日之前的各个版本中，输出目录的描述有误。文中没有描述中转、只以序数输出、或者使用好几个名称输出等内容。）

下一件最简单的事情是输出目录，是由"IMAGE\_DIRECTORY\_ENTRY\_EXPORT"（输出目录项）指向的。它是一个典型的在DLL中常见到的目录；包含一些输出函数的入口点（以及输出对象等的地址）。当然可执行文件也可能拥有输出符号但一般没有。

包含它们的节应该有"已初始化数据的"和"可读的"特性。这样的节应该是不可丢弃的，因为在运行时，进程有可能调用"GetProcAddress()"来寻找一个函数的入口点。如果单独成节的话，本节通常被称作".edata"；更常见的是，它被并入象"已初始化数据"之类的节中。

输出表（"IMAGE\_EXPORT\_DIRECTORY"）的结构由一个头和输出数据，也就是：符号名称、它们的序号和它们的入口点偏移量等构成。

1）首先，我们有一个没被使用并通常为0的、32位的"Characteristics"（特性）。

2）然后是一个32位的"TimeStamp"（时间戳），大概是提供此表被创建的时间；天呀，它的值并不总是有效（有些链接器将它设置为0）。

3-4）往后我们看到2个16位的、有关版本信息的word单元（"MajorVersion"和"MinorVersion"，含义分别为'主版本号'和'小版本号'），同样，它们很多地被设为0。

5) 下一个东东是32位的“Name”(名称);它是一个指向以0结尾的ASCII字符串为DLL名称的RVA。(为防DLL被改名时的错误,名称是必须的----参见输入目录中的“绑定”部分。)

6) 然后是32位的“Base”(基址)。稍后我们再讨论。

7) 下一个32位值是输出条目的总数(“NumberOfFunctions”,意为‘函数数’)。除了它们的序数外,各条目还可能使用一个或多个名称来输出。接下来的一个32位数字是输出名称的总数(“NumberOfNames”,意为‘名字数’)。

在大多数情况下,每一个输出条目都准确的有一个相应的名称,并且将用这个名称来使用它;但是一个条目可能拥有好几个相关联的名称(那样它们的每一个名称都可访问);或者它也可能没有名称,此时它只能以它的序数来访问。无名输出项(只有序数)的使用是不鼓励的,因为此时输出DLL的所有版本都必须使用相同的序数法,而这会造成维护的问题。

8) 下一个32位值“AddressOfFunctions”(函数地址)是指向输出条目列表的RVA。它指向一个32位值的“NumberOfFunctions”(函数数)数组,数组的每一项都是一个指向输出函数或变量的RVA。

关于此列表有两个怪事:第一,这样一个输出的RVA竟可能会为0,在此情况下,此值没被使用。第二,如果一个RVA指向含有输出目录的节,那么它就是一个中转输出。一个中转输出就是指指向另一个二进制文件中的输出项的指针;如果使用了它,就可用另一个二进制文件中的被指向的输出项来代替使用。此时的RVA指向,正如已提到的,输出目录的节中,指向一个以以零结尾的字符串组成的、被指向的DLL的名称和一个用点分开的输出项的名称,象“otherdll.exportname”这样,或者是DLL的名称和输出序数,象“otherdll.#19”这样。

现在到了解释输出序数的时候了。一个输出项的序数就是函数地址数组中的索引值加上上面提到的“Base”(基址)的值的和。在大多数情况下,“Base”(基址)的值为1,这就意味着第一个输出项的序数为1,第二个输出项的序数为2,以此类推。

9-10) “AddressOfFunctions”(函数地址) RVA之后,我们发现二个RVA,一个指向符号名称的32位RVA的数组“AddressOfNames”(名字的地址),另一个指向16位序数“AddressOfNameOrdinals”(名字序数的地



址)的数组。两个数组都有“NumberOfNames”(名字数)个元素。

符号名称可能会全部丢失,此时“AddressOfNames”(名字的地址)为0;否则,被指向的数组并行运行,这意味着它们的每个索引中的元素共同拥有。“AddressOfNames”(名字的地址)数组由以0结尾的输出名称的RVA组成;这些名称以一个分类的列表排列(即:数组的第一个成员是按照字母顺序排列的最小的名称的RVA;这使当按名称查找一个输出符号时,搜索的效率更高。)

根据PE规范,“AddressOfNameOrdinals”(名字序数的地址)数组每个名称拥有一个相应的序数,然而,我发现这个数组却将实际的索引包含到“AddressOfFunctions”(函数地址)数组中去。

我将画一个有关这三个表的图:

函数地址

|  
|  
|

v

带序数'基址'的输出RVA

带序数'基址+1'的输出RVA

...

带序数'基址+函数数-1'的输出RVA

名字地址 名字序数地址

||  
||  
||

v v

第一个名字的RVA <-> 第一个名字的输出索引

第二个名字的RVA <-> 第二个名字的输出索引

... ..

第`名字数`个名字的RVA <-> 第`名字数`个名字的输出索引

举一些例子是适宜的。

为按序数找到一个输出符号，先减去“Base”（基址）值以得到索引值，再根据“AddressOfFunctions”（函数地址）的RVA得到输出项数组，并用索引值去找到数组中的输出RVA。如果结果没有指向输出节中，你就完了。否则，它就指向那里的一个描述输出DLL和（输出项）名称或序数的字符串，之后你就得在那里查找中转输出。

为按名称找到一个输出符号，先跟随“AddressOfNames”（名字的地址）的RVA（如果是0就没有名称）找到输出名称的RVA数组。在列表中搜寻你要找的名称。用该名称在“AddressOfNameOrdinals”（名字序数的地址）数组中的索引，得到和找到的名称相应的16位数字。根据PE规范，这是一个序数，你需先减去“Base”（基址）值以得到输出索引值；但依据我的经验，这就是输出索引值，你不需要再减了。使用输出索引值，你就能在“AddressOfFunctions”（函数地址）数组中找到输出RVA了，要么是输出RVA本身，要么是一个描述中转输出的字符串的RVA。

## 7.输入符号 ( imported symbols )

-----

当编译器发现一个对别的可执行文件（大多数是DLL文件）中的函数调用时，在最简单化的情况下，它会对此情况一无所知，只是简单地输出一个对那个符号的正常调用指令。链接器不得不修正那个符号的地址，就象它为任何其它的外部符号所做的那样。

链接器使用一个输入库来查找从哪个DLL文件输入了哪个符号，并为所有的输入符号都建立存根，每个存根包含一个跳转指令；存根就是实际的调用目标。这些跳转指令实际上将跳往从所谓的输入地址表中提取的一个地址。在更复杂的应用程序（使用“\_\_declspec(dllimport)”时）中，编译器会知道函数是输入的，并直接输出一个位于输入地址表中的地址的调用，绕过跳转。

不管怎样，DLL文件中的函数地址总是必要的，并将于应用程序载入时，由加载器从输出DLL文件的输出目录中提

供。加载器知道哪个库中的哪些符号需要被查找以及哪些地址需要通过搜索输入目录来修正。

我最好给你一个例子。有或无\_\_declspec(dllimport)的调用如下所示：

源文件：

```
int symbol(char *);
__declspec(dllimport) int symbol2(char*);
void foo(void)
{
    int i=symbol("bar");
    int j=symbol2("baz");
}
```

汇编：

```
...
call _symbol ; 没有declspec(dllimport)的
...
call [__imp__symbol2] ; 含有declspec(dllimport)的
...
```

在第一种（没有\_\_declspec(dllimport)）情况下，编译器不知道“\_symbol”位于一个DLL文件中，因此链接器必须要提供“\_symbol”函数。因为此函数不存在，它就为输入符号提供一个存根函数，即一个间接跳转。所有输入存根的集合被称为“转移区”（有时也叫做“跳板”，因为你跳到那里的目的是为了跳到别的地方）。

典型地，此转移区位于代码节中（它不是输入目录的一部分）。每一个函数存根都是一个跳往DLL文件中的实际函数的跳转。转移区的形式象这样：

```
_symbol: jmp [__imp__symbol]
_other_symbol: jmp [__imp__other__symbol]
```

...

这意味着：如果你不指定“\_\_declspec(dllimport)”来使用输入符号，那么链接器将会为它们产生一个由间接跳转所组成的转移区。如果你真指定了“\_\_declspec(dllimport)”，那么编译器就会自己做间接（跳转），转移区也就不需要了。（这也意味着：如果你输入的是变量或其它东西，你就必须指定“\_\_declspec(dllimport)”，因为一个具有jmp指令的存根只合适于函数。）

不管怎样，符号“x”的地址都被存在“\_\_imp\_x”的存储单元。所有这样的存储单元一起形成所谓的“输入地址表”，此表是由被用到的各DLL文件中的输入库提供给链接器的。输入地址表就是由下面这种形式的一组地址组成的：

```
__imp__symbol: 0xdeadbeef
__imp__symbol2: 0x40100
__imp__symbol3: 0x300100
...
```

这个输入地址表是输入目录的一部分，并且被IMAGE\_DIRECTORY\_ENTRY\_IAT（输入地址表目录项）目录指针所指向（尽管有些链接器不设置此目录项，程序也能运行；很明显地，这是因为加载器不使用IMAGE\_DIRECTORY\_ENTRY\_IAT（输入地址表目录项）目录也能解决输入问题）。这些地址并不被链接器所知；链接器只插入一些伪地址（函数名称的RVA；参见后面的更多信息），这些伪地址会在载入时被加载器用输出DLL文件中的输出目录来修正。输入地址表，以及它是怎样被加载器找到的，将会在本章的后面被详细讲述。

注意:这个介绍是针对C语言规范的；有些别的应用程序构建环境是不使用输入库的，尽管它们都需要建立一个输入地址表，用来让它们的程序访问输入对象和函数。C语言编译器往往使用输入库，因为无论如何讲，这都有利于它们----它们的链接器使用好库。别的环境使用的是例如：一个列出需要的DLL文件名称和函数名称的描述文件（比如“模块定义文件”），或者一个源文件中的声明形式的列表等。

这就是程序的代码如何使用输入函数的；现在我们再来看看输入目录是如何建立以便加载器使用的。

输入目录应该存在于是“已初始化数据”并且“可读”的节中。

输入目录是一个多IMAGE\_IMPORT\_DESCRIPTOR（输入描述结构）的数组，每个被使用的DLL文件都有一个。（它们的）列表由一个全部用0填充的IMAGE\_IMPORT\_DESCRIPTOR（输入地址表目录项）结构作为结束。一个IMAGE\_IMPORT\_DESCRIPTOR（输入地址表目录项）是一个拥有下列成员的结构体：

OriginalFirstThunk（原始第一个换长）（汉译的说明见注释？）

它是一个RVA（32位），指向一个以0结尾的、由IMAGE\_THUNK\_DATA（换长数据）的RVA构成的数组，其每个IMAGE\_THUNK\_DATA（换长数据）元素都描述一个函数。此数组永不改变。

TimeStamp（时间日期戳）

它是一个具有好几个目的的32位的时间戳。让我们先假设时间戳为0，一些高级的情况将在以后处理。

ForwarderChain（中转链）

它是输入函数列表中第一个中转的、32位的索引。中转也是高级的东东。对初学者先将所有位设为-1。

Name（名称）

它是一个DLL文件的名称（0结尾的ASCII码字符串）的、32位的RVA。

FirstThunk（第一换长）

它也是一个RVA（32位），指向一个0结尾的、由IMAGE\_THUNK\_DATA（换长数据）的RVA构成的数组，其每个IMAGE\_THUNK\_DATA（换长数据）元素都描述一个函数。此数组是输入地址表的一部分，并且可以改变。

因此，数组中的每个IMAGE\_IMPORT\_DESCRIPTOR（输入描述结构）结构体都给出输出DLL文件的名称，并且，除了中转和时间日期戳，它还给出2个指向IMAGE\_THUNK\_DATA（换长数据）的数组的RVA，都是32位。（每个数组的最后一个成员都全部填充为0字节，以标志结尾。）

目前看来，每个IMAGE\_THUNK\_DATA（换长数据）都是一个RVA，指向一个描述输入函数的IMAGE\_IMPORT\_BY\_NAME（输入名字）项。

现在，有趣的是两个数组并行运行，也就是说：它们指向同一组IMAGE\_IMPORT\_BY\_NAME（输入名字）。

没有必要失望，我将再画一图。这里是IMAGE\_IMPORT\_DESCRIPTOR（输入描述结构）的关键内容：

原始第一个换长 第一个换长

||  
||  
||  
VV

```
0--> 函数1 <--0  
1--> 函数2 <--1  
2--> 函数3 <--2  
3--> foo <--3  
4--> mumpitz <--4  
5--> knuff <--5  
6--> 0 0<--6 /* 最后的RVA是0! */
```

图当中的名字就是尚未讨论的IMAGE\_IMPORT\_BY\_NAME（输入名字）。每一个都是一个16位的数字（一个提示）跟着一些数量未定的字节，它们都是以0结尾的、输入符号的ASCII码名字。

提示就是指向输出DLL文件名字表的索引（参见上面的输出目录）。那个索引中的名字将被一一尝试，如果没有相符的，再使用二进制搜索来寻找名字。

（有些链接器不愿意查找正确的提示，总是只简单的将其指定为1，或者其它的随意数字。这并无大害，只是使解决名字的第一次尝试总是失败，并迫使每个名字都使用二进制搜索来进行。）

总结一下：如果你想从“knurr”DLL中查找输入函数“foo”的信息，第一步你先找到数据目录中的IMAGE\_DIRECTORY\_ENTRY\_IMPORT（输入目录项）项，得到一个RVA，再在原始节数据中找到那个地址，现在你就得到一个IMAGE\_IMPORT\_DESCRIPTOR（输入描述结构）数组了。通过查看根据它们的“名称”被指向的字符串，得到和“knurr”DLL有关的这个数组的成员（即一个输入描述结构）。在你找到正确的IMAGE\_IMPORT\_DESCRIPTOR（输入描述结构）后，顺着它的“OriginalFirstThunk”（原始第一个换长）得到被指向的IMAGE\_THUNK\_DATA（换长数据）数组；再通过查询RVA找到“foo”函数。

好了，为什么我们有“两”列指向IMAGE\_IMPORT\_BY\_NAME（输入名字）的指针呢？这是因为在运行时，应用程序不需要输入函数的名字，只需要地址。在这里输入地址表又出现了。加载器将从相关的DLL文件的输出目录中查找每一个输入符号，并用DLL文件入口点的线性地址替换“FirstThunk”（第一个换长）列表中的IMAGE\_THUNK\_DATA（换长数据）元素（到现在之前它还是指向IMAGE\_IMPORT\_BY\_NAME（输入名字）的）。

请记住带有象“\_\_imp\_\_symbol”标签的地址列表；被数据目录IMAGE\_DIRECTORY\_ENTRY\_IAT（输入地址表目录项）所指向的输入地址表，就是被“FirstThunk”（第一个换长）所指向的列表。[在从好几个DLL文件输入的情况下，输入地址表是包含所有DLL文件的“FirstThunk”（第一个换长）数组。目录项IMAGE\_DIRECTORY\_ENTRY\_IAT（输入地址表目录项）可能会丢失，但输入（函数）仍能工作良好。]“OriginalFirstThunk”（原始第一个换长）数组保持不变，因此你总能通过“OriginalFirstThunk”（原始第一个换长）列表查找原始的输入名字列表。

现在输入已经被用正确的线性地址修正，如下所示：

原始第一个换长 第一个换长

||  
||  
||  
V V

0--> 函数1 0--> 输出函数1  
1--> 函数2 1--> 输出函数2  
2--> 函数3 2--> 输出函数3  
3--> foo 3--> 输出函数foo  
4--> mumpitz 4--> 输出函数mumpitz  
5--> knuff 5--> 输出函数knuff  
6--> 0 0<--6

这是简单情况下的基本结构。现在我们将要学习输入目录中的需细讲的东西。

第一，当数组中IMAGE\_THUNK\_DATA元（换长数据）素的IMAGE\_ORDINAL\_FLAG（序数标志）位（也是：MSB，参见注释？）被置1时，表示列表中没有符号的名字信息，符号只以序数输入。你可通过查看IMAGE\_THUNK\_DATA（换长数据）中的低地址word来得到序数。通过序数输入是不鼓励的，通过名字输入会更安全，因为如果输出DLL文件不是预期的版本时输出序数可能会改变。

第二，有所谓的“绑定输入”。

请思考一下加载器的工作：当它想执行的一个二进制文件需要一个DLL中的函数时，加载器会载入该DLL，找到它的输出目录，查找函数的RVA并计算函数的入口点。然后用这样找到的地址修正“FirstThunk”（第一个换长）列表。

假设程序员很聪明，给DLL文件提供的唯一优先载入地址不会发生冲突，那么我们就认为函数的入口点将总是相同的。它们在链接时能被算出并被补进“FirstThunk”（第一个换长）列表中，这就是“绑定输入”所发生的一切。（“绑定”工具就是干这个的，它是Win32 SDK的一部分。）

当然，你得慎重：用户的DLL可能是不同的版本，或者DLL必须重定位，这些都会使先前修正的“FirstThunk”（第一个换长）列表不再有效；此时，加载器仍能查寻“OriginalFirstThunk”（原始第一个换长）列表，找出输入符号并重新补正“FirstThunk”（第一个换长）列表。加载器知道这是必须的，当：1）输出DLL文件的版本不符，或2）输出DLL文件需要重定位时。

确定有没有重定位表对加载器来说不是问题，但该怎样找出版本的不同呢？这时IMAGE\_IMPORT\_DESCRIPTOR（输入描述结构）的“时间戳”就派上用场了。如果它是0，表明输入列表没有被绑定，加载器总是要修复入口点。否则的话，输入被绑定，“时间戳”必须要和“文件头”中的输出DLL文件的“时间戳”相符；如果不符的话，加载器就认为该二进制文件被绑到一个“错误”的DLL文件上并重新补正输入列表。

这里有另外一个有关输入列表中的“中转”的怪事。一个DLL文件能输出一个不定义在本DLL文件中却需从另一个DLL文件中输入的符号；这样的符号据说就是被中转的（参见上面的输出目录描述）。



现在，很明显的，你 cannot 通过查看那个实际上并不包含入口点信息的DLL文件的时间戳来确定一个符号的入口点是否有效。因此，出于安全的原因，中转符号的入口点必须总是被修正。在二进制文件的输入列表中，中转符号的输入必须被找出，以便加载器能补正它们。

这一点可通过“ForwarderChain”（中转链）来做到。它是一个指向换长列表中的索引值；被索引位置的输入就是一个中转输出，并且此位置的“FirstThunk”（第一个换长）列表中的内容就是“下一个”中转输入的索引值，以此类推，直到索引值为-1，就表明已没有其他的中转了。如果根本就没有中转，那么“ForwarderChain”（中转链）的值本身就为-1。

这就是所谓的“老式”绑定。

至此，我们应该总结一下我们目前已掌握的情况：-)

OK，我将认为你已找到了IMAGE\_DIRECTORY\_ENTRY\_IMPORT（输入目录项）并且已根据它找到了它的输入目录，位于某个节中。现在你已处于IMAGE\_IMPORT\_DESCRIPTOR（输入描述结构）数组的开头了，此类数组的最后一个将以全0字节填充。

要读懂一个IMAGE\_IMPORT\_DESCRIPTOR（输入描述结构），你得先查看它的“名字”项，根据它的RVA，你就能找到输出DLL文件的名字。下一步你得确定输入是否是绑定的；如果输入是绑定的，“时间戳”就会是非“0”的。如果它们是绑定的，现在就是你通过比较“时间戳”来检查DLL文件的版本是否相符的好机会了。

现在你根据“OriginalFirstThunk”（原始第一个换长）的RVA来到了IMAGE\_THUNK\_DATA（换长数据）数组；过完这些数组（它是0结尾的），它的每个成员都将是一个IMAGE\_IMPORT\_BY\_NAME（输入名字）的RVA（除非它的高位被置1，此时你找不到名字只有序数）。根据那个RVA，并跳过2字节（即‘提示’），现在你就得到一个以0结尾的字符串，这就是输入函数的名字。

在绑定输入时要找到提供的入口点，先根据“FirstThunk”（第一个换长）平行的来到“OriginalFirstThunk”（原始第一个换长）数组；数组成员就是入口点的线性地址（暂时不考虑中转的话题）。

还有一件我到现在都没有提及的事情：明显地有些链接器在构建输入目录时会产生bug（我就发现一个还在被一个Borland C链接器使用的bug）。这些链接器把IMAGE\_IMPORT\_DESCRIPTOR（输入描述结构）中的“OriginalFirstThunk”（原始第一个换长）设为0，并只建立“FirstThunk”（第一个换长）。很明显的，这样

的输入目录不能被绑定（否则重修输入的必须信息就会丢失----你根本找不到函数名字）。在这种情况下，你得根据“FirstThunk”（第一个换长）数组来取得输入符号名字，你将永远得不到预先补正的入口地址。我已发现一个TIS文件(参考书目[6])，讲述一个在某种程度上和此bug兼容的输入目录，因此那个文件可能就是该bug的起源。

TIS文件规定：

IMPORT FLAGS（输入标志）

TIME/DATE STAMP（时间/日期戳）

MAJOR VERSION - MINOR VERSION（主版本号 - 小版本号）

NAME RVA（名字的RVA）

IMPORT LOOKUP TABLE RVA（输入查询表的RVA）

IMPORT ADDRESS TABLE RVA（输入地址表的RVA）

而别处使用的对应结构是：

OriginalFirstThunk（原始第一个换长）

TimeDateStamp（时间日期戳）

ForwarderChain（中转链）

Name（名字）

FirstThunk（第一个换长）

最后一个关于输入目录的需要细讲的就是所谓的“新式”绑定（在参考书目[3]中讲述），它也可以由“绑定”工具来处理。当使用这种方式时，“时间日期戳”的所有位被置为1，并且没有中转链；此时所有输入符号的地址都将被补正，而不管它们是不是中转的。尽管如此，你还是需要知道DLL的版本，并且你还是需要将序数符号从中转符号中区分开来。为了达到这个目的，IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT（绑定输入目录项）目录被创建了。就我所见，它将不被放在节中，而是被放在头中，处于节头之后第一节之前。（咳，这不是我的发明，我只是讲述它而已！）

这个目录告诉你，每一个已使用的DLL文件的中转输出是从哪些别的DLL文件中来的。

结构是IMAGE\_BOUND\_IMPORT\_DESCRIPTOR（绑定输入描述结构）形式的，包括（按这个顺序）：

一个32位数字，“时间戳”。

一个16位数字，“OffsetModuleName（模块名字偏移量）”，是从目录开头到以0结尾的DLL文件名的偏移量；

一个16位数字，“NumberOfModuleForwarderRefs（模块中转参考的数字）”，给出这个DLL文件为它的中转使用的DLL文件数。

紧随这个结构之后你会发现“NumberOfModuleForwarderRefs（模块中转参考的数字）”结构，告诉你这个DLL文件的中转所来自的DLL文件的名称和版本。这些结构就是“IMAGE\_BOUND\_FORWARDER\_REF（绑定中转参考）”结构的：

一个32位的数字“时间日期戳”（TimeStamp）；

一个16位的数字“模块名称偏移量”（OffsetModuleName），它就是从目录开头到中转来自的那个DLL文件的0结尾的名字处的偏移量；

一个16位的未使用单元。

跟在“IMAGE\_BOUND\_FORWARDER\_REF（绑定中转参考）”后的是下一个“IMAGE\_BOUND\_IMPORT\_DESCRIPTOR（绑定输入描述结构）”，以此类推；列表最终以一个全部为0位的IMAGE\_BOUND\_IMPORT\_DESCRIPTOR（绑定输入描述结构）结束。

我对由此（描述）造成的不便表示歉意，但这就是它看起来的样子:-）

现在，如果你有一个新的绑定输入目录，你得载入所有的DLL文件，并使用目录指针IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT（绑定输入目录项）找到IMAGE\_BOUND\_IMPORT\_DESCRIPTOR（绑定输入描述结构），扫描整个IMAGE\_BOUND\_IMPORT\_DESCRIPTOR（绑定输入描述结构），并检查被载入的DLL文件的“时间日期戳”和这个目录中提供的是否相符。如果不符，就将输入目录中“第一换长”（FirstThunk）中的错误全部修改过来。

## 8.资源（resources）

-----

资源，比如对话框、菜单、图标等等，都存储在IMAGE\_DIRECTORY\_ENTRY\_RESOURCE（“资源目录项”）指向的数据目录中。它们处于一个至少“IMAGE\_SCN\_CNT\_INITIALIZED\_DATA（已初始化数据内容节）”和“IMAGE\_SCN\_MEM\_READ（内存可读节）”标志位都被置为1的节中。

资源的基础是“资源目录”（IMAGE\_RESOURCE\_DIRECTORY）；它包含好几个“资源目录项”（IMAGE\_RESOURCE\_DIRECTORY\_ENTRY），其中的每一项反过来又可能指向一个“资源目录”。按照这种方式，你就得到一个以“资源目录项”为树叶的“资源目录”树；它们的树叶指向实际的资源数据。

在实际使用中，情况会稍微简单些。一般你不会遇到不可能理清的特别复杂的树的。通常，它的层次结构是这样的：一个目录作为根。它指向很多目录，每种资源类型都有一个。这些目录又指向子目录，每个子目录都有一个名字或者ID号并指向这个资源所提供的各种语言的目录；每种语言你都能找到一个资源项，资源项最终指向（具体的）数据。（注意：多语言资源不能在Win95上运行。即使程序有好几种语言，Win95也总是使用相同的资源----我没有查出是哪一种，但我猜测肯定是它最先碰到的那种。多语言资源在NT系统上可以运行。）

没有指针的树大致象这样：

```
( 根 )
|
+-----+-----+
| | |
菜单 对话框 图标
| | |
+-----+-----+ +-+-----+ +-+-----+-----+
| | | | | |
"main" "popup" 0x10 "maindlg" 0x100 0x110 0x120
| | | | | |
+---+--+ | | | | |
| | default english default def. def. def.
```

german english

一个“资源目录项” ( IMAGE\_RESOURCE\_DIRECTORY ) 包含：

32位未使用标志，叫做“特征” ( Characteristics ) ；

32位“时间日期戳” ( 同样按常用的time\_t表示法 ) ，告诉你资源被创建的时间 ( 如果此项被设置的话 ) ；

16位“主版本号” ( MajorVersion ) 和16位“小版本号” ( MinorVersion ) ，以允许你据此维护资源的几个版本；

16位“已命名项目数” ( NumberOfNamedEntries ) 和另一个16位的“ID项目数” ( NumberOfIdEntries ) 。

紧随此结构后的是“已命名项目数”+“ID项目数”两结构体，它们都是“资源目录项”格式，都以名字开头。它们可能指向下一个“资源目录”或者指向实际的资源数据。

一个“资源目录项”由下面组成：

32位单元提供你它所描述的资源的ID或者是目录；

32位的到数据的偏移量或者是到下一个子目录的偏移量。

ID的含义取决于树中的层次；ID可能是一个数字 ( 如果最高位为0 ) 也可能是一个名字 ( 如果最高位为1 ) 。如果是一个名字，它的低31位就是从资源节原始数据的开始到这个名字 ( 名字有16位长并由unicode的宽字符而不是0结尾符作为结束 ) 的偏移量。

如果你位于根目录之中，且如果ID是一个数字的话，那么它指的就是下面的一种资源类型：

- 1: 光标
- 2: 位图
- 3: 图标
- 4: 菜单
- 5: 对话框
- 6: 字串表
- 7: 字体目录
- 8: 字体
- 9: 快捷键

10: 未格式化资源数据

11: 信息表

12: 组光标

14: 组图标

16: 版本信息

任何其它数字都是用户自定义的。任何有类型名的资源类型也是用户自定义的。

如果你处于（树的）下一层当中，此时ID一定是一个数字，且就是资源的一个特例的语言ID号；例如，你可以（同时）拥有澳大利亚英语、加拿大法语和瑞士德语等本地化形式的对话框，并且它们分享同一个资源ID。系统会根据线程的地点来选择要使用的对话框，反过来地点又反映了用户的“区域设置”。（如果资源找不到线程地点，系统将先使用一个中性的子语言资源作为地点，比如它将寻找标准法语而不是用户所拥有的加拿大法语；如果它还是找不到，就使用最小语言ID号的那个实例。必须注意，所有这些只工作于NT系统之上的。）

为便于辨认语言ID，使用宏PRIMARYLANGID()（意为“主语言ID”）和SUBLANGID()（意为“子语言ID”）将它分开为主语言ID和子语言ID，分别使用它的0-9位和10-15位。这些值定义在“winresrc.h”文件中。

语言资源只支持快捷键、对话框、菜单、资源数据或字符串等；其它资源类型必须为

LANG\_NEUTRAL/SUBLANG\_NEUTRAL（中性语言/中性子语言）。

要确定资源目录的下一层是不是另一个目录，你可查看它的偏移量的最高位。如果它是1，剩下的31位就是从资源节原始数据的开始到下一层目录的偏移量，还是按“资源目录”后接“资源目录项”的格式。如果高位为0，它就是从资源节原始数据的开始到资源的原始数据描述，即一个资源数据项的偏移量。资源的原始数据描述包含32位的“OffsetToData”（到数据的偏移量）（指的是到原始数据的偏移量，从资源节原始数据的开头算起），32位的数据的“Size”（大小），32位的“CodePage”（代码页）和一个未使用的32位单元。

（不鼓励使用代码页，你应该使用“语言”的特性来支持多地域。）

原始数据格式依赖于资源类型；详细的介绍可在微软的SDK文档中找到。注意：除了用户自定义资源，资源中的任何字符串总是按UNICODE格式，明显的，用户自定义的资源按的是开发者选定的格式。

## 9.重定位 ( relocations )

-----

我将要描述的最后一个是基址重定位目录。它是由可选头数据目录中的  
IMAGE\_DIRECTORY\_ENTRY\_BASERELOC ( 基址重定位目录项 ) 项来指向的。典型的，它包含在自己的节  
中，名字象".reloc"这样，并且IMAGE\_SCN\_CNT\_INITIALIZED\_DATA ( 已初始化数据内容节 )、  
IMAGE\_SCN\_MEM\_DISCARDABLE ( 内存可丢弃节 ) 和IMAGE\_SCN\_MEM\_READ ( 内存可读节 ) 等标志位被  
置1。

如果映象文件不能被加载到可选头中提到的优先载入地址"ImageBase" ( 映象基址 ) 时，重定位数据对加载器来说就是必须的。此时，链接器所提供的固定地址就不再有效，并且加载器将不得不对静态变量、字符串文字等使用的绝对地址进行修正。

所谓重定位目录就是一些连续的块，每一块都包含4K映象文件的重定位信息。块由一个  
"IMAGE\_BASE\_RELOCATION ( 基址重定位 )" 结构体开始，这个结构体包含一个32位  
的"VirtualAddress ( 虚拟地址 )" 项和一个32位的"SizeOfBlock ( 块大小 )" 项。跟在它们后面的就是块的实际  
重定位数据，每一条都是16位的。

"VirtualAddress ( 虚拟地址 )" 就是重定位所在块需要应用的基本的RVA；"SizeOfBlock ( 块大小 )" 就是整个  
块的字节大小；跟在后面的重定位的数目是：('SizeOfBlock'-sizeof(IMAGE\_BASE\_RELOCATION))/2个。当  
你碰到一个"VirtualAddress ( 虚拟地址 )" 值为0的"IMAGE\_BASE\_RELOCATION ( 基址重定位 )" 结构体时，  
重定位信息就结束了。

每一个16位的重定位信息由低12位的重定位位置和高4位的重定位类型组成。要得到重定位的RVA，你需要用这  
个12位的位置加上"IMAGE\_BASE\_RELOCATION ( 基址重定位 )" 中的"VirtualAddress ( 虚拟地址 )"。类型  
是下面之一：

IMAGE\_REL\_BASED\_ABSOLUTE (0)

这种不需操作；用于将块按32位边界对齐。位置应该为0。

IMAGE\_REL\_BASED\_HIGH (1)



重定位的高16位必须被用于被偏移量所指向的那个16位的WORD单元，此WORD是一个32位的DWORD的高位WORD。

IMAGE\_REL\_BASED\_LOW (2)

重定位的低16位必须被用于被偏移量所指向的那个16位的WORD单元，此WORD是一个32位的DWORD的低位WORD。

IMAGE\_REL\_BASED\_HIGHLOW (3)

重定位的全部32位必须应用于上面所说的全部32位。这种（和无需操作的第“0”种）是我在二进制文件种实际发现的仅有的重定位类型。

IMAGE\_REL\_BASED\_HIGHADJ (4)

这是一种复杂的。请自己参阅（参考文献[6]），并努力弄懂它的意思：“高调整。这种修正要求一个全32位值。高16位定位于偏移量处，低16位定位在下一个数组元素（此数组元素包括在大小的域中）的偏移量处。它们两个需要被连成一个有符号的变量。加上32位的增量。然后加上0x8000并将有符号变量的高16位存储在偏移量处的16位域中。”

IMAGE\_REL\_BASED\_MIPS\_JMPADDR (5)

不清楚

IMAGE\_REL\_BASED\_SECTION (6)

不清楚

IMAGE\_REL\_BASED\_REL32 (7)

不清楚

举一个例子，如果你发现重定位信息是

0x00004000 (32位, 开始的RVA)

0x00000010 (32位, 块的大小)

0x3012 (16位的重定位数据)

0x3080 (16位的重定位数据)

0x30f6 (16位的重定位数据)

0x0000 (16位的重定位数据)

0x00000000 (下一块的RVA)

0xff341234



你知道第一块描述的重定位开始于RVA 0x4000处，有16字节长。因为头用掉了8字节，并且一个重定位要用2字节，所以块中计有 $(16-8)/2=4$ 个重定位。第一个重定位被应用于0x4012处的DWORD，第二个于0x4080处的DWORD，第三个于0x40f6处的DWORD。最后一个不需操作。

下一块的RVA是0，列表结束。

好，你怎么处理一个重定位呢？

你能知道映象文件“被”重定位到可选头“ImageBase（映象基址）”的优先载入地址；你也能知道你真正载入的地址。如果它们相同，你什么也不用做。如果它们不同，你需计算出实际基址-优先基址的差并加上重定位位置的值（有符号，可能为负值），此值你可通过上面讲述的方法找到。

## 九、致谢（Acknowledgments）

-----

感谢David Binette的调试和校读。（剩下的错误全部都是我的。）

也感谢wotsit.org网站让我将此文放到他们的网站上。

## 十、版权（Copyright）

-----

本文的版权属于B. Luevelsmeyer，1999年。它是免费的，你可以任意的使用，但后果自负。它含有错误并不完整，特此警告。

## 十一、Bug报告（Bug reports）

-----

Bug报告（或其他建议）请发送至：bernd.luevelsmeyer@iplan.heitec.net

## 十二、版本 ( Versions )

-----

你可在文件的顶部找到当前的版本号。

1998-04-06

第一次公开发表

1998-07-29

将映像文件版本和子系统版本中错误的"byte"改为"word"

更正"栈只限于1 MB"的错误 ( 实际上没有上限 )

更正一些输入错误

1999-03-15

更正输出目录的描述, 原来非常不全

调整输入目录的描述, 原来讲的不清

更正输入错误并为其它节改了一些词句

## 十三、参考文献 ( Literature )

-----

[1]

"Peering Inside the PE: A Tour of the Win32 Portable Executable File Format" (M. Pietrek), in: Microsoft Systems Journal 3/1994

[2]

"Why to Use \_declspec(dllimport) & \_declspec(dllexport) In Code", MS Knowledge Base Q132044

[3] 《Windows 问与答》

"Windows Q&A" (M. Pietrek), in: Microsoft Systems Journal 8/1995

[4] 《编写多语言资源》

"Writing Multiple-Language Resources", MS Knowledge Base Q89866

[5]

"The Portable Executable File Format from Top to Bottom" (Randy Kath),  
in: Microsoft Developer Network

[6] 《Windows下TIS格式规范1.0版》

Tool Interface Standard (TIS) Formats Specification for Windows Version  
1.0 (Intel Order Number 241597, Intel Corporation 1993)

附录 ( Appendix: hello world ) :

-----

在这个附录中我将给大家展示一下怎样手工建立一个程序。因为我不懂DEC Alpha的，本例将使用Intel汇编语言。

本程序相当于

```
#include <stdio.h>
int main(void)
```

```
{  
puts(hello,world);  
return 0;  
}
```

首先，我使用Win32函数来翻译它以取代C运行时库：

```
#define STD_OUTPUT_HANDLE -11UL  
#define hello "hello, world\n"
```

```
__declspec(dllimport) unsigned long __stdcall  
GetStdHandle(unsigned long hdl);
```

```
__declspec(dllimport) unsigned long __stdcall  
WriteConsoleA(unsigned long hConsoleOutput,  
const void *buffer,  
unsigned long chrs,  
unsigned long *written,  
unsigned long unused  
);
```

```
static unsigned long written;
```

```
void startup(void)  
{  
WriteConsoleA(GetStdHandle(STD_OUTPUT_HANDLE),hello,sizeof(hello)-1,&written,0);  
return;  
}
```

现在我将笨拙的将它汇编出来：

startup:

; WriteConsole()的参数, 反向的

6A 00 push 0x00000000

68 ?? ?? ?? ?? push offset \_written

6A 0D push 0x0000000d

68 ?? ?? ?? ?? push offset hello

; GetStdHandle()的参数

6A F5 push 0xffffffff5

2E FF 15 ?? ?? ?? ?? call dword ptr cs:\_\_imp\_\_GetStdHandle@4

; 结果是WriteConsole()的参数

50 push eax

2E FF 15 ?? ?? ?? ?? call dword ptr cs:\_\_imp\_\_WriteConsoleA@20

C3 ret

hello:

68 65 6C 6C 6F 2C 20 77 6F 72 6C 64 0A "hello, world\n"

\_written:

00 00 00 00

以上就是编译的部分。任何人都能做到这点。从现在起让我们扮演起链接器的角色，这会非常有趣 :-)

我需要先找出函数WriteConsoleA()和GetStdHandle()。碰巧它们都在“kernel32.dll”中。(这是“输入库”部分。)

现在我开始做可执行文件。问号代表待定的值；它们将在以后被修正。

首先是DOS-根，开始于0x0，有0x40字节长：

00 | 4d 5a 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```
10 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 | 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00
```

正如你所见到的，这不是真正的MS-DOS程序。它只是一个开始部分有“MZ”签名的头和紧跟在头后面的e\_lfanew指针，没有任何代码。这是因为它并非打算运行于MS-DOS之上；它之所以在这里只是因为规范的需要。

然后是PE签名，开始于0x40，有0x4字节长：

```
50 45 00 00
```

现在到了文件头，开始于0x44，有0x14字节长：

Machine 4c 01 ; i386

NumberOfSections 02 00 ; 代码段和数据段

TimeStamp 00 00 00 00 ; 谁管它？

PointerToSymbolTable 00 00 00 00 ; 未用

NumberOfSymbols 00 00 00 00 ; 未用

SizeOfOptionalHeader e0 00 ; 常量

Characteristics 02 01 ; 32位机器上的可执行文件

接着是可选头，开始于0x58，有0x60字节长：

Magic 0b 01 ; 常量

MajorLinkerVersion 00 ; 我是 0.0 版:-)

MinorLinkerVersion 00 ;

SizeOfCode 20 00 00 00 ; 32字节代码

SizeOfInitializedData ?? ?? ?? ?? ; 待找出

SizeOfUninitializedData 00 00 00 00 ; 我们没有BSS节

AddressOfEntryPoint ?? ?? ?? ?? ; 待定

BaseOfCode ?? ?? ?? ?? ; 待定

BaseOfData ?? ?? ?? ?? ; 待定

ImageBase 00 00 10 00 ; 1 MB, 随意选  
SectionAlignment 20 00 00 00 ; 32字节对齐  
FileAlignment 20 00 00 00 ; 32字节对齐  
MajorOperatingSystemVersion 04 00 ; NT 4.0  
MinorOperatingSystemVersion 00 00 ;  
MajorImageVersion 00 00 ;0.0版  
MinorImageVersion 00 00 ;  
MajorSubsystemVersion 04 00 ; Win32 4.0  
MinorSubsystemVersion 00 00 ;  
Win32VersionValue 00 00 00 00 ; 未使用?  
SizeOfImage ?? ?? ?? ?? ; 待定  
SizeOfHeaders ?? ?? ?? ?? ; 待定  
Checksum 00 00 00 00 ; 非驱动不用  
Subsystem 03 00 ; Win32控制台  
DllCharacteristics 00 00 ; 未用 (不是一个DLL)  
SizeOfStackReserve 00 00 10 00 ; 1 MB栈  
SizeOfStackCommit 00 10 00 00 ; 开始时4 KB  
SizeOfHeapReserve 00 00 10 00 ; 1 MB堆  
SizeOfHeapCommit 00 10 00 00 ; 开始时4 KB  
LoaderFlags 00 00 00 00 ; 未知  
NumberOfRvaAndSizes 10 00 00 00 ; 常量

正如你所见，我计划只用2个节，一个用于代码，一个用于所有剩余的东西（数据、常量和输入目录等）。没有重定位和象资源之类其它东西。我也不用BSS节并将变量“written”放入已初始化数据。文件和RAM中的节对齐都是一样的（32字节）；这将有助于使任务简单，否则我就得来回地计算RVA很多次。

现在我们设置数据目录，开始于0xb8字节，有 0x80字节长：

地址 大小

00 00 00 00 00 00 00 00 ; IMAGE\_DIRECTORY\_ENTRY\_EXPORT (0)

```
?? ?? ?? ?? ?? ?? ?? ?? ; IMAGE_DIRECTORY_ENTRY_IMPORT (1)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_RESOURCE (2)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_EXCEPTION (3)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_SECURITY (4)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_BASERELOC (5)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_DEBUG (6)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_COPYRIGHT (7)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_GLOBALPTR (8)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_TLS (9)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG (10)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (11)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_IAT (12)
00 00 00 00 00 00 00 00 ; 13
00 00 00 00 00 00 00 00 ; 14
00 00 00 00 00 00 00 00 ; 15
```

仅使用输入目录。

下一个使节头。首先我们做代码节的，代码节将包含前面所编的汇编语句。它有32字节长，所以代码节也就是这么长。节头从0x138处开始，有0x28字节长：

```
Name 2e 63 6f 64 65 00 00 00 ; ".code"的ASCII码值
VirtualSize 00 00 00 00 ; 未用
VirtualAddress ?? ?? ?? ?? ; 待定
SizeOfRawData 20 00 00 00 ; 代码的大小
PointerToRawData ?? ?? ?? ?? ; 待定
PointerToRelocations 00 00 00 00 ; 未用
PointerToLinenumbers 00 00 00 00 ; 未用
NumberOfRelocations 00 00 ; 未用
NumberOfLinenumbers 00 00 ; 未用
```



Characteristics 20 00 00 60 ; 代码节, 可执行, 可读

第二节将包含数据。节头开始于0x160处, 有0x28字节长:

Name 2e 64 61 74 61 00 00 00 ; ".data"的ASCII码值

VirtualSize 00 00 00 00 ; 未用

VirtualAddress ?? ?? ?? ?? ; 待定

SizeOfRawData ?? ?? ?? ?? ; 待定

PointerToRawData ?? ?? ?? ?? ; 待定

PointerToRelocations 00 00 00 00 ; 未用

PointerToLinenumbers 00 00 00 00 ; 未用

NumberOfRelocations 00 00 ; 未用

NumberOfLinenumbers 00 00 ; 未用

Characteristics 40 00 00 c0 ; 已初始化的, 可读, 可写

下一个字节位于0x188处, 但节需要按32字节(的倍数)对齐(因为我是这样选择的), 所以我们需要添一些(0)字节直到0x1a0处:

00 00 00 00 00 00 ; 填充的

00 00 00 00 00 00

00 00 00 00 00 00

00 00 00 00 00 00

现在第一节, 就是上面所汇编的代码节, “到”了。它开始于0x1a0处, 有0x20字节长:

6A 00 ; push 0x00000000

68 ?? ?? ?? ?? ; push offset \_written

6A 0D ; push 0x0000000d

68 ?? ?? ?? ?? ; push offset hello\_string

6A F5 ; push 0xffffffff

```
2E FF 15 ?? ?? ?? ?? ; call dword ptr cs:__imp__GetStdHandle@4
50 ; push eax
2E FF 15 ?? ?? ?? ?? ; call dword ptr cs:__imp__WriteConsoleA@20
C3 ; ret
```

因为这一节的长度（刚好32字节），在下一节（数据节）前我们不需要填充任何字节。下一节到了，从0x1c0处开始：

```
68 65 6C 6C 6F 2C 20 77 6F 72 6C 64 0A ; "hello, world\n"的ASCII码值
00 00 00 ; 填充几个0以和_written对齐
00 00 00 00 ; _written
```

现在剩下的只有输入目录了。本文件将从"kernel32.dll"库中输入2个函数，输入目录将从本节的变量后面立即开始。首先我们先将上面的数据按32字节对齐：

```
00 00 00 00 00 00 00 00 00 00 00 00 ; 填充的
```

在0x1e0处开始输入描述（IMAGE\_IMPORT\_DESCRIPTOR）：

```
OriginalFirstThunk ?? ?? ?? ?? ; 待定
TimeStamp 00 00 00 00 ; 未绑定
ForwarderChain ff ff ff ff ; 无中转
Name ?? ?? ?? ?? ; 待定
FirstThunk ?? ?? ?? ?? ; 待定
```

我们需要用一个0字节项来结束输入目录（我们现在位于0x1f4）：

```
OriginalFirstThunk 00 00 00 00 ; 结束符号
TimeStamp 00 00 00 00 ;
ForwarderChain 00 00 00 00 ;
Name 00 00 00 00 ;
```

FirstThunk 00 00 00 00 ;

现在只剩下DLL名字，还有2个换长，以及换长数据和函数名字了。但现在我们真的很快就要完成了。

DLL名字，以0结尾，开始于0x208处：

6b 65 72 6e 65 6c 33 32 2e 64 6c 6c 00 ; "kernel32.dll"的ASCII码值

00 00 00 ; 填充到32位边界

原始第一个换长，开始于0x218处：

AddressOfData ?? ?? ?? ?? ; "WriteConsoleA"函数名的RVA

AddressOfData ?? ?? ?? ?? ; "GetStdHandle"函数名的RVA

00 00 00 00 ; 结束符号

第一个换长就是同样的列表，开始于0x224处：

(\_\_imp\_\_WriteConsoleA@20, at 0x224)

AddressOfData ?? ?? ?? ?? ; "WriteConsoleA"函数名的RVA

(\_\_imp\_\_GetStdHandle@4, at 0x228)

AddressOfData ?? ?? ?? ?? ; "GetStdHandle"函数名的RVA

00 00 00 00 ; 结束符号

现在剩下的只有输入名字 ( IMAGE\_IMPORT\_BY\_NAME ) 形式的两个函数名了。我们现处于0x230字节。

01 00 ; 序数，不需要正确

57 72 69 74 65 43 6f 6e 73 6f 6c 65 41 00 ; "WriteConsoleA"的ASCII码值

02 00 ; 序数，不需要正确

47 65 74 53 74 64 48 61 6e 64 6c 65 00 ; "GetStdHandle"的ASCII码值

Ok, 这就全部结束了。下一个字节，我们并不真正需要，是0x24f。我们必须将节填充到0x260处：

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; 填充的

00

-----

我们已经完成了。因为我们已经知道了所有的字节偏移量，我们可以应用我们的修正到所有原先被用“？”符号标为“未知”的地址和大小了。

我将不强迫你一步一步地去读它（很好懂的），只直接给出结果来：

-----

DOS-头, 始于0x0:

```
00 | 4d 5a 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 | 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00
```

签名, 始于0x40:

50 45 00 00

文件头, 始于0x44:

Machine 4c 01 ; i386

NumberOfSections 02 00 ; 代码和数据

TimeStamp 00 00 00 00 ; 谁管它?

PointerToSymbolTable 00 00 00 00 ; 未用

NumberOfSymbols 00 00 00 00 ; 未用

SizeOfOptionalHeader e0 00 ; 常量

Characteristics 02 01 ; 可执行于32位机器上

可选头, 始于0x58:

Magic 0b 01 ; 常量

MajorLinkerVersion 00 ; 我是 0.0版 :-)  
MinorLinkerVersion 00 ;  
SizeOfCode 20 00 00 00 ; 32字节代码  
SizeOfInitializedData a0 00 00 00 ; 数据节大小  
SizeOfUninitializedData 00 00 00 00 ; 我们没有 BSS节  
AddressOfEntryPoint a0 01 00 00 ; 代码节的开始处  
BaseOfCode a0 01 00 00 ; 代码节的RVA  
BaseOfData c0 01 00 00 ; 数据节的RVA  
ImageBase 00 00 10 00 ; 1 MB, 任意选择  
SectionAlignment 20 00 00 00 ; 32字节对齐  
FileAlignment 20 00 00 00 ; 32字节对齐  
MajorOperatingSystemVersion 04 00 ; NT 4.0  
MinorOperatingSystemVersion 00 00 ;  
MajorImageVersion 00 00 ; 0.0版本  
MinorImageVersion 00 00 ;  
MajorSubsystemVersion 04 00 ; Win32 4.0  
MinorSubsystemVersion 00 00 ;  
Win32VersionValue 00 00 00 00 ; 未用?  
SizeOfImage c0 00 00 00 ; 所有节大小的总数  
SizeOfHeaders a0 01 00 00 ; 第一节的偏移量  
Checksum 00 00 00 00 ; 非驱动程序不须用  
Subsystem 03 00 ; Win32控制台程序  
DllCharacteristics 00 00 ; 未用(不是一个DLL)  
SizeOfStackReserve 00 00 10 00 ; 1 MB 栈  
SizeOfStackCommit 00 10 00 00 ; 开始时4 KB  
SizeOfHeapReserve 00 00 10 00 ; 1 MB 堆  
SizeOfHeapCommit 00 10 00 00 ; 开始时4 KB  
LoaderFlags 00 00 00 00 ; 未知  
NumberOfRvaAndSizes 10 00 00 00 ; 常量

数据目录, 开始于 0xb8:

地址 大小

```
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_EXPORT (0)
e0 01 00 00 6f 00 00 00 ; IMAGE_DIRECTORY_ENTRY_IMPORT (1)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_RESOURCE (2)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_EXCEPTION (3)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_SECURITY (4)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_BASERELOC (5)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_DEBUG (6)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_COPYRIGHT (7)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_GLOBALPTR (8)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_TLS (9)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG (10)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (11)
00 00 00 00 00 00 00 00 ; IMAGE_DIRECTORY_ENTRY_IAT (12)
00 00 00 00 00 00 00 00 ; 13
00 00 00 00 00 00 00 00 ; 14
00 00 00 00 00 00 00 00 ; 15
```

节头(代码节), 开始于0x138:

Name 2e 63 6f 64 65 00 00 00 ; ".code"

VirtualSize 00 00 00 00 ; 未用

VirtualAddress a0 01 00 00 ; 代码节的RVA

SizeOfRawData 20 00 00 00 ; 代码的大小

PointerToRawData a0 01 00 00 ; 代码节的文件偏移量

PointerToRelocations 00 00 00 00 ; 未用

PointerToLinenumbers 00 00 00 00 ; 未用

NumberOfRelocations 00 00 ; 未用

NumberOfLinenumbers 00 00 ; 未用

Characteristics 20 00 00 60 ; 代码节, 可执行, 可读

节头(数据节), 开始于0x160:

Name 2e 64 61 74 61 00 00 00 ; ".data"

VirtualSize 00 00 00 00 ; 未用

VirtualAddress c0 01 00 00 ; 数据节的RVA

SizeOfRawData a0 00 00 00 ; 数据节的大小

PointerToRawData c0 01 00 00 ; 数据节的文件偏移量

PointerToRelocations 00 00 00 00 ; 未用

PointerToLinenumbers 00 00 00 00 ; 未用

NumberOfRelocations 00 00 ; 未用

NumberOfLinenumbers 00 00 ; 未用

Characteristics 40 00 00 c0 ; 已初始化, 可读, 可写

(填充)

00 00 00 00 00 00 ; 填充的

00 00 00 00 00 00

00 00 00 00 00 00

00 00 00 00 00 00

代码节, 开始于0x1a0:

6A 00 ; push 0x00000000

68 d0 01 10 00 ; push offset \_written

6A 0D ; push 0x0000000d

68 c0 01 10 00 ; push offset hello\_string

6A F5 ; push 0xffffffff5

2E FF 15 28 02 10 00 ; call dword ptr cs: \_\_imp\_\_GetStdHandle@4

50 ; push eax

```
2E FF 15 24 02 10 00 ; call dword ptr cs:___imp___WriteConsoleA@20
C3 ; ret
```

数据节, 始于0x1c0:

```
68 65 6C 6C 6F 2C 20 77 6F 72 6C 64 0A ; "hello, world\n"
```

```
00 00 00 ; 填充到和_written对齐
```

```
00 00 00 00 ; _written
```

填充:

```
00 00 00 00 00 00 00 00 00 00 00 00 ; 填充的
```

输入描述 ( IMAGE\_IMPORT\_DESCRIPTOR ), 始于0x1e0:

```
OriginalFirstThunk 18 02 00 00 ; 原始第一个换长的RVA
```

```
TimeDateStamp 00 00 00 00 ; 未绑定
```

```
ForwarderChain ff ff ff ff ; -1, 无中转
```

```
Name 08 02 00 00 ; DLL名字的RVA
```

```
FirstThunk 24 02 00 00 ; 第一个换长的RVA
```

结束标志(0x1f4):

```
OriginalFirstThunk 00 00 00 00 ; 结束标志
```

```
TimeDateStamp 00 00 00 00 ;
```

```
ForwarderChain 00 00 00 00 ;
```

```
Name 00 00 00 00 ;
```

```
FirstThunk 00 00 00 00 ;
```

DLL名字, 始于0x208:

```
6b 65 72 6e 65 6c 33 32 2e 64 6c 6c 00 ; "kernel32.dll"
```

```
00 00 00 ; 填充到32位边界
```

原始第一个换长, 始于0x218:

```
AddressOfData 30 02 00 00 ; 函数名"WriteConsoleA"的RVA
```



AddressOfData 40 02 00 00 ; 函数名"GetStdHandle"的RVA  
00 00 00 00 ; 结束标志

第一个换长,开始于0x224:

AddressOfData 30 02 00 00 ; 函数名"WriteConsoleA"的RVA  
AddressOfData 40 02 00 00 ; 函数名"GetStdHandle"的RVA  
00 00 00 00 ; 结束标志

输入函数名称 ( IMAGE\_IMPORT\_BY\_NAME ) , 开始于0x230:

01 00 ; 序数, 不需要正确  
57 72 69 74 65 43 6f 6e 73 6f 6c 65 41 00 ; "WriteConsoleA"的ASCII码值

IMAGE\_IMPORT\_BY\_NAME,开始于0x240:

02 00 ; 序数, 不需要正确  
47 65 74 53 74 64 48 61 6e 64 6c 65 00 ; "GetStdHandle"的ASCII码值  
(填充)  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; 填充的  
00

第一个未使用字节开始于: 0x260

-----

噢, 这个文件能在NT上却不能在windows 95上运行。windows 95不能运行按32字节节对齐的应用程序, 它要求节对齐为4 KB; 并且很明显的, 文件对齐也应为512字节。因此要想在windows 95上运行, 你得插入很多的0字节 ( 为了对齐 ) 并调整RVA。感谢D. Binette在windows 95上的 ( 运行 ) 试验。

-- 全文结束 --

[译后记]：

由于时间等因素，遗漏、重复、不准确甚至错误等情况在所难免，敬请各位批评、指正！另外，由于我保留了所有的英文术语（译文就在后面），所以译文看起来有点乱，请大家见谅！

本文的原文写于1999年，由于时间关系，文中所说的有关公司、某某项目应用的操作系统范围等等介绍可能已经不对或不准确了，请大家自己分析、鉴别。

最后再谈点个人感想：

1) 个人觉得本文的难点在于输入符号（表）部分，而其精华乃在附录之中。在学习前面的各种项目成员名称、说明等的同时，如能对照后面的附录来学习，将会起到事半功倍的效果。另外，文中所说的什么结构体、共用体之类术语都是针对编程而言，如果你并不想或不会编程的话，可以将其理解为一个将其它东西集合在一起的一个容器就行了。

2) 原文发表于1998-1999年之间，而相应的中文译文至今也难在网上搜寻得到，这对中国的破界来说不能说不是一个很大的遗憾！本文仅起抛砖引玉之用，希望能有更多、更好、更及时的国外类似资料出现在我们的网络之上，以造福于我们这些广大的菜鸟。

沈忠平 2006.02 于和州

=====

|“PE文件格式”1.9版注释：|

=====

#### ①Win32s和Win32

Win32s是“WIN32 subset”的缩写，它是一个可被加入到Windows 3.1和Windows for Workgroups系统中以使它们能够运行32位应用程序的软件包。正如它的名字所暗示的那样，Win32s只是Windows 95和Windows NT系统中使用的Win32 API的一个子集。Win32s的主要功能就是在32位和16位内存地址间相互转换，也就是一种被称为换长的操作。

Win32是32位Windows ( 包括Windows NT,95, 98 和2000等 ) 操作系统的编程接口(API)。当应用程序是按Win32 API编写时, 它们就具有16位API(Win16)所不具备的一些高级性能。一个按Win32编写的程序能运行在所有的操作系统之上, 除非这个程序要求特定的操作系统特性, 而这些特性别的操作系统又没有。例如, Windows NT提供的安全特性Windows 95/98就没有。一个为NT系统的这些特性编写的程序就不能运行在其它的Windows系统之上。

使用此API的程序 能运行在...上

Win32 95, 98, NT, 2000, XP

Win32s 3.1, 95, 98, NT, 2000, XP

Win32c 95

Win16 3.0, 3.1, 95, 98, NT, 2000, XP

## ②目标文件 ( Object file ) 和映象文件 ( Image file )

目标文件 ( Object file ) 指的是链接程序 ( 链接器 ) 的输入文件。链接器输出的是映象文件, 映象文件反过来又是加载器的输入文件。“object file”一词未必含有任何和面向对象的编程有关的联系。

映象文件 ( Image file ) 指的就是可执行文件: 或者是.EXE, 或者是.DLL。一个映象文件可被想象为“内存映象”。“映象文件”一词常被用来代替“可执行文件”, 因为后者有时被用来专指.EXE文件。

## ③UNIX

是一个很流行的多用户、多任务的操作系统, 由贝尔实验室于上世纪70年代早期开发出来的。只有很少的程序员建立的UNIX系统本来是设计给他们这些程序员专用的、小巧的、灵活的系统。UNIX是用高级编程语言, 就是C语言, 编写的第一批操作系统之一。这就意味着只要电脑上有C语言编译器, UNIX就可以被虚拟地安装到任何电脑上。天生的可移植性加上低廉的价格使得UNIX成为各大学的流行选择。( 因为反信用条款禁止贝尔实验室将UNIX作为它的全权产品推向市场, 所以UNIX的价格不贵。 )

贝尔实验室只发布它自己源语言形式的UNIX操作系统, 所以任何获得一份拷贝的人都可以按照自己的意愿来修改和定制它。到上世纪70年代末时, 有好几十种不同版本的UNIX运行在世界各地。( 更多信息请参阅别的资料。 )

#### ④VMS

“Open Virtual Memory System”或仅VMS，是运行于VAX和Alpha系列电脑之上的高端电脑服务器操作系统的名字，现在用于使用英特尔Itanium CPU的Hewlett-Packard（惠普）系统之上。VAX和Alpha系列电脑由美国马萨诸塞州Maynard市的数据设备（DEC）公司（现在由HP拥有）生产的。OpenVMS 是一个基于多用户、多处理虚拟存储的操作系统，设计用于时间共享、批处理和事项处理等。

#### ⑤SDK

是“software development kit”（软件开发工具箱）的缩写，它是一个供程序员为特定平台开发应用程序的编程包。典型的，一个SDK包含一个或多个API库、各种编程工具和相关文档等。

#### ⑥Ne Format（New-style EXE Format的缩写）

是一个早期Windows操作系统的可执行文件(.EXE)，包含一个代码和数据的集合或者一个代码、数据和资源的集合。这种可执行文件也包括两个头：一个MS-DOS头和一个Windows头，和一些节。（具体参看其他资料）

#### ⑦OS/2（IBM Operating System/2，IBM 操作系统/2）

操作系统/2（OS/2）最初是由 Microsoft 和 IBM 共同合作开发的一种应用于 PC 机的操作系统。现在只由 IBM 销售、支持和管理。其设计目标是替换传统的 DOS 操作系统。OS/2 与 DOS、Windows 都相兼容。换句话说，OS/2 操作系统可运行所有的 DOS 和 Windows 程序，但在 OS/2 下运行的某些特殊写程序却不能在 DOS 或 Windows 下运行。

OS/2 是一个32位的、为个人计算机而设计的、支持保护模式和多任务的操作系统。OS/2 系统中的图形表示管理器（Presentation Manager）作为其图形系统，主要负责管理窗口、字体及控件等。OS/2 系统顶部是 Workplace 命令解释程序（WPS - 该内容在 OS/2 2.0中有具体介绍），WPS 以文档为中心，允许用户访问文件和打印机，并可以启动程序。WPS 遵循 IBM 的用户界面标准，即“通用用户访问”。

OS/2 操作系统中包含一种系统对象模型（SOM），包括磁盘、文件夹、文件、程序对象及打印机等对象。SOM

允许应用程序间代码共享，但这与编程语言无关。一种称为 DSOM 的分布式版本支持不同计算机上对象间的相互通信。DSOM 建立在 CORBA 基础上。SOM 类似于微软的组件对象模型 (Component Object Model)，同时两者相互竞争。目前人们对 SOM 和 DSOM 已停止深度开发。

OS/2 操作系统也包括一种叫做 OpenDoc 的混合文档技术，它由 Apple 开发而成。但目前人们对 OpenDoc 也已停止深度开发。

由于 OS/2 存在市场局限性，IBM 公司已于2003年3月12日按照电子商务计划停止了 OS/2 的发展市场。

## ⑧MIPS

MIPS是世界上很流行的一种RISC处理器。MIPS的意思是“无内部互锁流水级的微处理器”(Microprocessor without interlocked piped stages)，其机制是尽量利用软件办法避免流水线中的数据相关问题。它最早是在80年代初期由斯坦福(Stanford)大学Hennessy教授领导的研究小组研制出来的。MIPS公司的R系列就是在此基础上开发的RISC工业产品的微处理器。这些系列产品为很多计算机公司采用构成各种工作站和计算机系统。如 R3000、R4000、R10000等都是其生产的处理器。

MIPS技术公司是美国著名的芯片设计公司，它采用精简指令系统计算结构(RISC)来设计芯片。和英特尔采用的复杂指令系统计算结构(CISC)相比，RISC具有设计更简单、设计周期更短等优点，并可以应用更多先进的技术，开发更快的下一代处理器。MIPS是出现最早的商业RISC架构芯片之一，新的架构集成了所有原来MIPS指令集，并增加了许多更强大的功能。

## ⑨big-endian、Little-endian和endian

Big-endian和Little-endian是用来表述一组有序的字节数存放在计算机内存中时的顺序的术语。Big-endian(即“大端结束”或者“大尾”)是将高位字节(序列中最重要值)先存放在低地址处的顺序，而Little-endian(即“小端结束”或者“小尾”)是将低位字节(序列中最不重要值)先存放在低地址处的顺序。举例来说，在使用Big-endian顺序的计算机中，要存储一个十六进制数4F52所需要的字节将会以4F52的形式存储(比如4F存放在内存的1000位置，而52将会被存储在1001位置)。而在使用Little-endian顺序的系统中，存储的形式将会是524F(52在地址1000处，4F在地址1001处)。IBM的370种大型机、大多数基于RISC的计算机以及Motorola的微处理器使用的是Big-endian顺序，TCP/IP协议也是。而Intel的处理器和DEC公司的一些程序则使

用的Little-endian方式。

“endian”这个词出自《格列佛游记》。小人国的内战就源于吃鸡蛋时是究竟从大头(Big-Endian)敲开还是从小头(Little-Endian)敲开，由此曾发生过六次叛乱，其中一个皇帝送了命，另一个丢了王位。

我们一般将endian翻译成“字节序”，将big endian和little endian称作“大尾”和“小尾”。

#### ⑩Alpha AXP

“DEC Alpha”，也被称作“Alpha AXP”，是一个原来由美国数据设备公司(DEC)开发和制造的64位RISC微处理器（例如：DEC Alpha AXP 21064 微处理器），他们将它用在自己的工作站和服务器的系列上。被设计作为VAX系列计算机的继承者，Alpha AXP不但支持VMS操作系统，同时也支持Digital UNIX操作系统。后来的一些开放源码操作系统也能运行于Alpha之上，著名的Linux和BSD UNIX操作系统特别支持。微软直到Windows NT 4.0 SP6才支持这种处理器，但Windows 2000第2版之后就又不支持了。

#### ?UTC

是“Coordinated Universal Time”的缩写，意为“协调通用时间”，它是综合了只以地球的不停旋转速率为基准的格林威治标准时间（Greenwich Mean Time）和高度精确的原子时间的一种时标。当原子时间和地球时间达到一秒的时差时，一个闰秒就被算进UTC时间中。UTC设计于1972年1月1日，并被国际度量衡局（International Bureau of Weights and Measures）于巴黎协调通过。跟格林威治标准时间一样，UTC也被设定于0经度的本初子午线。

#### ?BSS

是“Block Started by Symbol”的缩写，意为“以符号开始的块”。BSS是Unix链接器产生的未初始化数据段。其他的段分别是包含程序代码的“text”段和包含已初始化数据的“data”段。BSS段的变量只有名称和大小却没有值。此名后来被许多文件格式使用，包括PE。

“以符号开始的块”指的是编译器处理未初始化数据的地方。BSS节不包含任何数据，只是简单的维护开始和结束的地址，以便内存区能在运行时被有效地清零。BSS节在应用程序的二进制映像文件中并不存在，例如：

```
unsigned char var; // 分配到.bss节的8位未初始化变量
```

`unsigned char var2 = 25; // 分配到.data节的8位已初始化变量`

?BSOD ( blue screen of death , 蓝屏死机 )

是运行在Windows环境下的计算机上出现的一个错误，甚至包括最早版本的Windows，比如Windows 3.0和3.1，在后来的Windows版本比如Microsoft Windows 95, Windows 98, Windows NT,和Windows 2000上仍能出现。它被开玩笑地称为蓝屏之死是因为错误发生时，屏幕变成蓝色，电脑总是不能正常运转并需要重新启动。

?POSIX

是“Portable Operating System Interface for UNIX”( UNIX可移植操作系统接口 ) 的首字母缩写，它是定义程序和操作系统之间的接口的一套IEEE和ISO标准。通过将他们的程序设计为符合POSIX标准，开发者就能获得一些让他们的程序可以容易被移植到其他POSIX兼容的操作系统上的保证，主要包括大多数UNIX操作系统。POSIX标准目前由IEEE下叫做“Portable Applications Standards Committee”(PASC) ( 可移植的应用程序标准委员会 ) 维护。

?thunk

( 动词 ) 换长，变长；已经想到的，预先想到的

( 指在个人电脑中，将一个16位内存地址转换为一个32位的地址，或者相反。换长是必须的，因为英特尔的老16位微处理器使用一种叫分段内存的定址方式，而它的32位微处理器使用的却是一个统一的地址空间。Window 95支持一种允许32位程序调用16位DLL的换长机制，叫统一换长。而另一方面，运行在Windows 3.x和Windows for Workgroup下的16位应用程序不能使用32位DLL，除非32位地址被转换为16位地址。这就是Win32的功能，并被称为通用换长。

根据民间传说，thunk一词是由一位Algol-60编程语言的开发者编出的，他在一天深夜意识到参数的数据类型是可以被编译器稍先一点知道的。也就是说，到了编译器处理参数的时候，它就已经想到了(thunked)数据类型了。该词的含义近年来已变化很大了。)

( 名词 ) 换长，变长 ( 在一个分段内存地址空间和一个统一地址空间之间互相转换的操作 )



( 我查遍书店中所有的大大小小的英汉和英英词典，都没有找到thunk这个词的含义。后在网上找到了它的英语解释，却找不到它对应的汉语译法，现根据它的意思，姑且译之。各位勿笑，还请高手指点。 )

( 英文参见：[url]<http://www.webopedia.com/TERM/T/thunk.html> ) [/url]

?MSB

"Most Significant Bit"的首字母缩写，意为"最重要的位"。在一个二进制的数字中，它就是最左边的那一位，也是最重要的那一位。

原文地址：<http://bbs.pediy.com/printthread.php?p=162585>

分类：Compiler 字数：26635

标签：PE

打赏

点赞

收藏

分享



j m

程序员 徐汇


+ 关注

粉丝 23 | 博文 145 | 码字总数 18


相关博客





- PE格式

 y\_x

60 0
- PE文件格式和ELF文件格式  
(上) ---PE文件

 j\_m

158 0
-  U盘装机助理-PE环境

 mvpbang

5 0

评论 (0)

Ctrl+Enter 发表评论