# PE 格式实验

【实验目的】

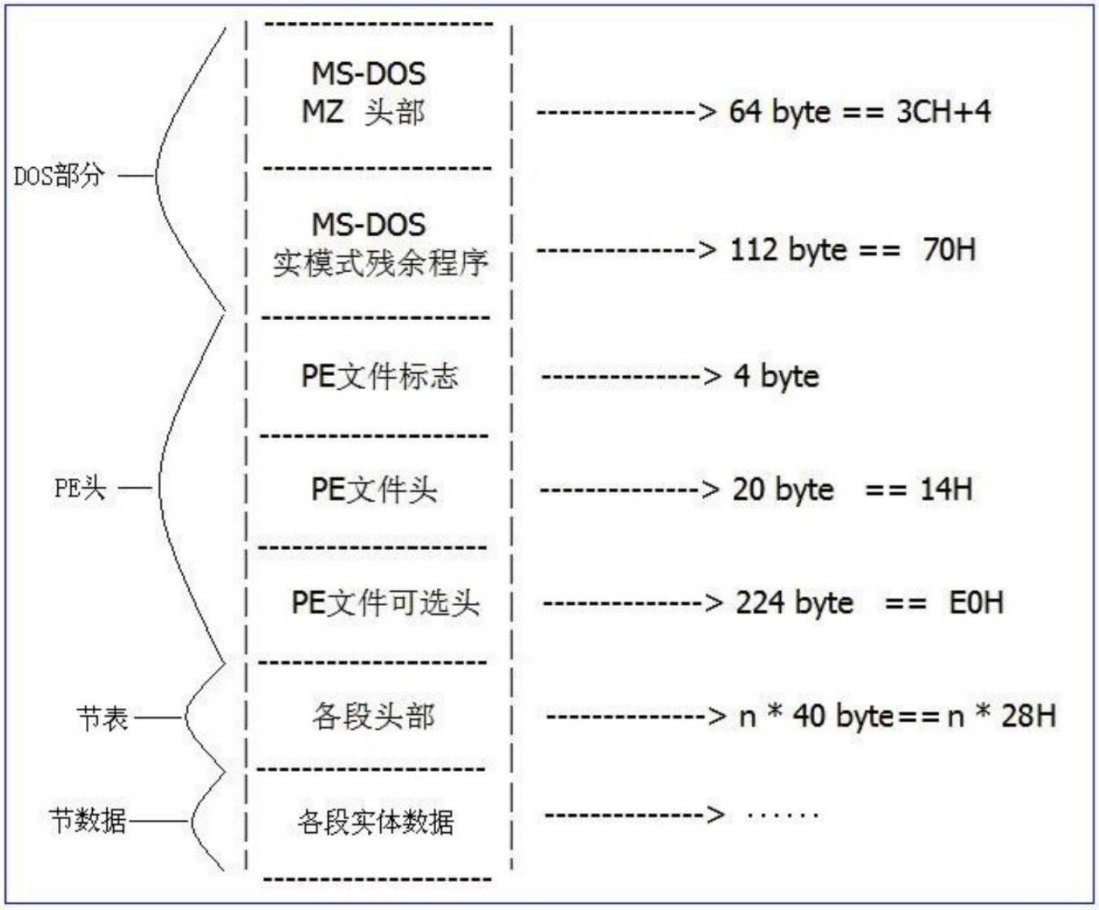
通过实验，了解 PE 文件的基本结构，理解 PE 文件头、节表、导入表等相关知识。

【实验环境】

- Windows 系统
- 任意十六进制编辑器（以 UltraEdit 为例）

【前置知识】

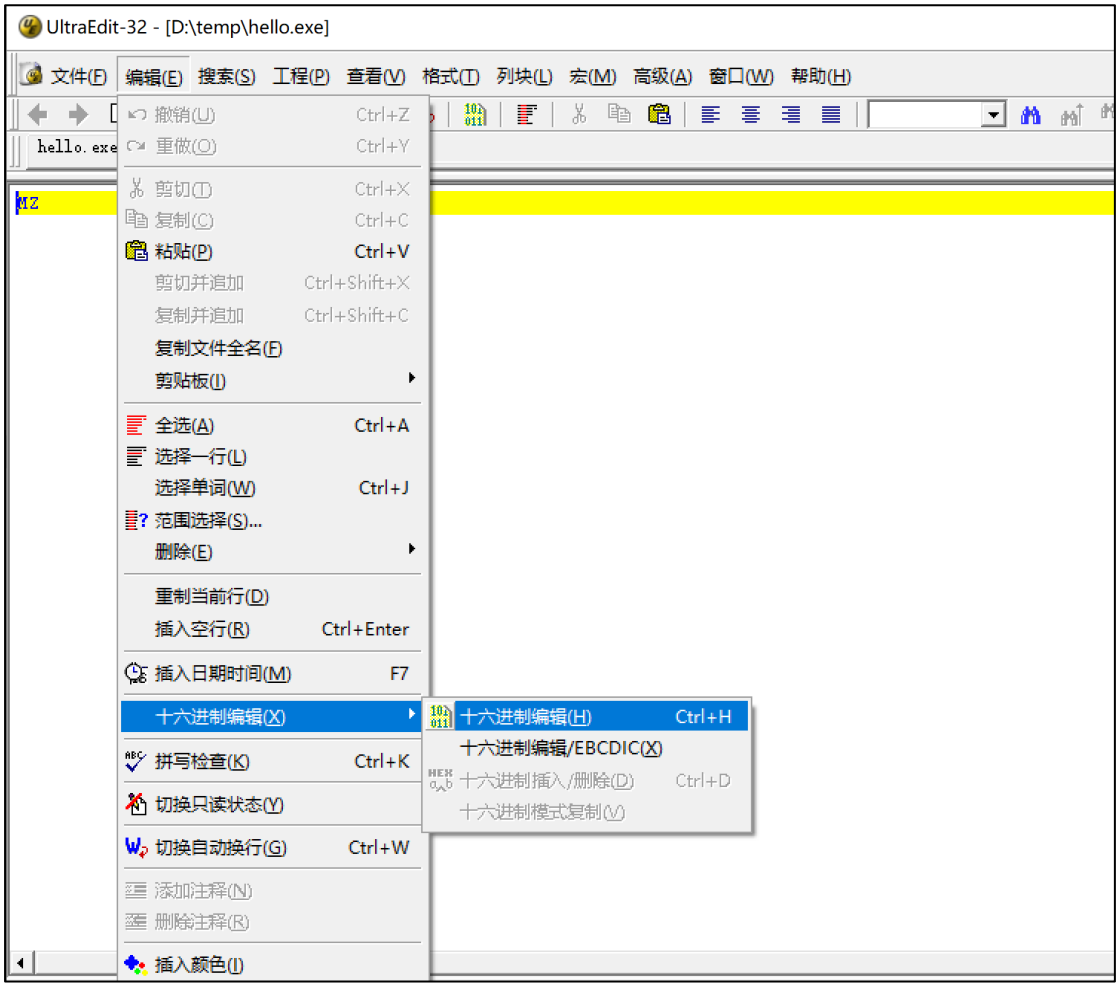
PE 结构示意图




【实验步骤】

(1) 在 UltraEdit 中新建文件，输入字母 MZ，作为 DOS 文件头的第一个成员 e_magic 的值。然后将文件保存为 hello.exe。

(2) 切换到十六进制编辑模式。




(3) 然后填写 58 个字节，作为 DOS 头的第 2-18 个成员，值都设为 00。
再输入 4 个字节 B0 00 00 00，作为第 19 个成员 e_lfanew 的值，即 PE 头的偏移位置。
(4) 从地址 040h 开始，继续填写 112 个字节，值都设为 00，作为 DOS Stub program。

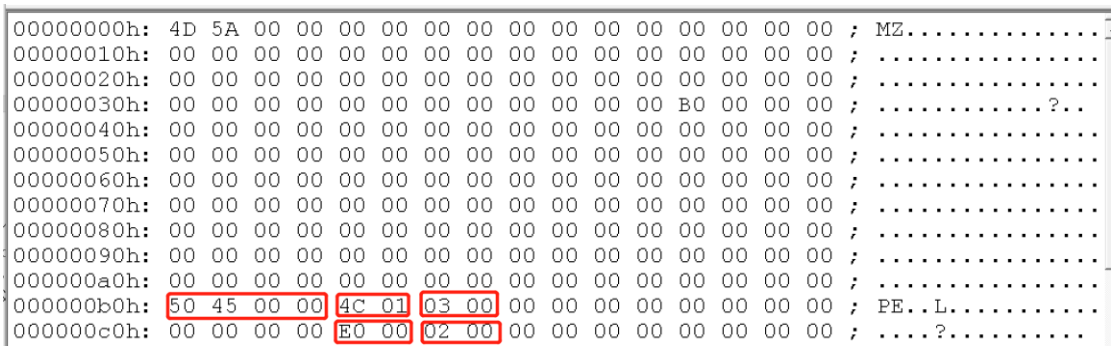
```
00000000h: 4D 5A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; MZ..............
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 B0 00 00 00 ; ............?..
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000070h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000080h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000090h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

至此完成了 DOS 部分的内容设置。

(5) 下面进入 PE 头的内容，在地址 0b0h 处填写 4 个字节 50 45 00 00，作为 PE 文件标识。
(6) 从地址 0b4h 开始，继续填写 20 个字节，作为 PE 文件头的 FileHeader 成员。下面对各成员值分别进行设置。

- 成员 1 Machine 占 2 个字节，值依次修改为 4C 01，表示要运行在 Intel 平台上；
- 成员 2 NumberOfSections，占 2 个字节，值依次修改为 03 00，表示程序内容共有 3 个节；
- 成员 3-5，占 12 个字节，值都设为 00，暂时不用关心；

- 成员 6 SizeOfOptionalHeader 占 2 个字节，值依次修改为 E0 00，表示 PE 文件头的 OptionalHeader 成员的大小是 224（0x00E0）个字节；
- 成员 7 Characteristics 占 2 个字节，值依次修改为 02 00，表示当前程序是可执行程序。（16 个二进制位各代表不同的含义）




(7) 从地址 0c8h 开始，继续填写 224 个字节，作为 PE 文件头的 OptionalHeader 成员。下面对各成员值进行设置。

- 成员 1 Magic 占 2 个字节，值依次修改为 0B 01，表示程序为.exe 文件；
- 成员 2-6，占 14 个字节，值均填充为 00，暂时不用关心；
- 成员 7 AddressOfEntryPoint，占 4 个字节，表示代码入口的 RVA（Relative Virtual Address，相对虚拟地址），即程序的入口地址；这里暂时将值修改为 AA AA AA AA，等后面确定之后再重新修改。
- 成员 8-9，占 8 个字节，值均填充为 00，暂时不用关心；
- 成员 10 ImageBase，占 4 个字节，值依次修改为 00 00 40 00，表示程序被加载进内存时的优先加载地址。PE 加载器默认情况下会优先尝试把文件加载到内存地址 00400000h 处。
- 成员 11 SectionAlignment，占 4 个字节，值依次修改为 00 10 00 00，表示程序节加载到内存后的对齐粒度为 4k（0x00001000）。
- 成员 12 FileAlignment，占 4 个字节，值依次修改为 00 02 00 00，表示程序节在磁盘文件中的对齐粒度为 512（0x00000200）。
- 成员 13-16,占 8 个字节，值均填充为 00，暂时不用关心；
- 成员 17 MajorSubsystemVersion，占 2 个字节，值修改为 04 00，表示子系统主版本号为 win32 子系统 4.0 版本；
- 成员 18 MinorSubsystemVersion，占 2 个字节，值修改为 00 00，表示子系统副版本号；
- 成员 19 Win32VersionValue，占 2 个字节，值依次修改为 00 00，表示 win32 版本值；
- 成员 20 SizeOfImage，占 4 个字节，值依次修改为 00 40 00 00，表示程序载入内存后占用内存的大小；

**计算内存映像大小，**

是指经过节对齐处理后的内存映像大小。因为文件头部分总长小于 1000h，但是内存中的对齐粒度是 1000h，所以文件头部分要占 1000h。另外还有 3 个节。每个节的长度都小于 1000h，被映射后同样要占 1000h，所以共占用内存的大小为 1000h + 3 * 1000h = 4000h。所以这个值是 0x00004000。

- 成员 21 SizeOfHeaders，占 4 个字节，值依次修改为 00 04 00 00，表示文件头在磁盘文件中的大小；

**计算磁盘上文件头大小**

是指节之前的所有文件头加节表的大小，相当于文件大小减去文件中所有节的大小。这个值也可以作为 PE 文件第一节的文件偏移。文件头总大小为：64 + 112 + 4 + 20 + 224 = 424，节表大小 3 * 40 =120。424 + 120 = 544(0x0220)。因为文件中的对齐粒度是 0x0200，所以经过文件对齐后实际占用 0x0400 的空间，所以此值为 0x00000400”。

- 成员 22 CheckSum，占 4 个字节，值依次修改为 00 00 00 00，表示校验和；
- 成员 23 Subsystem，占 2 个字节，值依次修改为 03 00，表示 NT 子系统；
- 成员 24-29，占 22 个字节，值均填充为 00，暂时不用关心；
- 成员 30 NumberOfRvaAndSizes，占 4 个字节，值依次修改为 10 00 00 00，表示成员 31 的元素个数 16（0x00000010）；
- 成员 31 DataDirectory,占 128 个字节,是一个 IMAGE_DATA_DIRECTORY 结构体的数组，共有 16 个元素，分别代表 16 个目录项。IMAGE_DATA_DIRECTORY 结构体有两个成员，各占 4 个字节，所以成员 31 的大小是 128 个字节（2*4*16）。

对于目前的程序，只需要关心第 2 个元素 IMAGE_DIRECTORY_ENTRY_IMPORT 导入表项，其它元素的值均填充为 00，暂时不用关心。

导入表目录项标识了程序从其他模块导入的函数信息。

为了显示一个消息框，需要导入 user32.dll 库中的 MessageBoxA 函数。为了程序正常退出，还要导入 kernel32.dll 库中的 ExitProcess 函数。

这里先把导入表目录项的两个成员值，即地址 130h 处的 8 个字节依次修改为 BB BB BB BB CC CC CC CC。

IMAGE_DATA_DIRECTORY 结构体中第一个成员表示表的起始 RVA 地址，第二个成员表示该表的长度。因为现在还不知道具体的值，等后面计算出来之后再来修改。

```
000000b0h: 50 45 00 00 4C 01 03 00 00 00 00 00 00 00 00 00 ; PE..L...........
000000c0h: 00 00 00 00 E0 00 02 00 0B 01 00 00 00 00 00 00 ; ....?...........
000000d0h: 00 00 00 00 00 00 00 00 AA AA AA AA 00 00 00 00 ; ........    ....
000000e0h: 00 00 00 00 00 00 40 00 00 10 00 00 00 02 00 00 ; ......@.........
000000f0h: 00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 ; ................
00000100h: 00 40 00 00 00 04 00 00 00 00 00 00 03 00 00 00 ; .@..............
00000110h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000120h: 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000130h: BB BB BB BB CC CC CC CC 00 00 00 00 00 00 00 00 ; 换换烫烫........
00000140h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000150h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000160h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000170h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000180h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000190h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000001a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

(8) 接下来构造节表。节表是一个 IMAGE_SECTION_HEADER 结构体数组，数组元素个数就是 file header(IMAGE_FILE_HEADER) 结构中 NumberOfSections 成员的值，前面已经设为 3。表示这个程序共有 3 个节，分别是代码节 (.text)、只读数据节 (.rdata)和全局变量数据节(.data)。

程序中用到的所有代码、资源、全局数据等内容都被存放在不同的节中。节表则包含了每个节的加载位置、节大小、节属性等信息

IMAGE_SECTION_HEADER 结构有 10 个成员，下面依次来完成三个节表项的构造。

首先是代码节。

- 成员 1（Name），8 个字节，表示该节的名称，这里循惯例使用.text，所以在地址 1a8h 处填写相应的 ASCII 码值 2E 74 65 78 74 00 00 00;
- 成员 2（VirtualSize），4 个字节，值修改为 26 00 00 00，表示该节数据映射到

内存后所占的字节数，即有效代码所占的字节数。后面会给出程序需要的 0x0026 字节代码；

- 成员 3（VirtualAddress），4 个字节，值修改为 00 10 00 00，表示该节映射到内存中的起始地址是 00001000h。

**计算内存中代码节起始地址：**

因为.text 紧跟在文件头之后，而整个文件头的大小是 64+112+4+20+224+40*3=544 字节(0x0220)，小于 0x1000，所以映射到内存后占的大小为 0x1000（经过内存对齐)。

这时， OptionalHeader 的成员 7 AddressOfEntryPoint 的值也可以确定了。它表示程序的入口地址，也就是.text 段的起始地址。现在可以把地址 0d8h 处的 AA AA AA AA 修改为 00 10 00 00。

（程序的入口地址并不一定就是代码节的起始位置。它通常是由编译器生成的。只要符合 PE 结构的要求，可以把代码的起始地址随意安排在什么位置，作为程序执行代码的入口地址。本程序为了方便，选择把代码入口放在代码节的起始地址处）

- 成员 4（SizeOfRawData），4 个字节，值修改为 26 00 00 00，表示该节在文件中所占的大小。在成员 2 处已经提到实际代码是 0x0026 个字节。另外，这里也可以填写此值经过文件对齐后的值即 0x0200(00 02 00 00)。
- 成员 5（PointerToRawData），4 个字节，值修改为 00 04 00 00，表示该节在文件中的起始地址。

**计算文件中代码节起始地址：**

因为.text 是紧跟在文件头之后，而整个文件头大小前面已经算过是 0x0220 字节,所以在文件中占的大小为 0x0400（经过文件对齐)。

- 成员 6-9 共 12 个字节，暂时不用关心，都用 00 填充。
- 成员 10（Characteristics），4 个字节，值修改为 60 00 00 20。表示当前程序可执行、含有初始化数据等节属性（32 个二进制位各代表不同的含义）

至此，整个.text 节表完成。

按照同样的的方法，再分别构造.rdata 节表(只读数据节表)和.data 数据节表。

先看只读数据节表。

- 成员 1（Name），8 个字节，表示该节的名称，这里循惯例使用.rdata，所以在地址 1d0h 处填写相应的 ASCII 码值 2E 72 64 61 74 61 00 00;
- 成员 2（VirtualSize），4 个字节，值修改为 92 00 00 00，表示该节数据映射到内存后所占的字节数，即有效内容所占的字节数。后面会给出相应的内容；
- 成员 3（VirtualAddress），4 个字节，值修改为 00 20 00 00，表示该节映射到内存中的起始地址是 00002000h。

**计算内存中只读数据节起始地址：**

因为.rdata 是紧跟在.text 之后，所以映射到内存后占的起始地址为 2000h（经过内存对齐)

- 成员 4（SizeOfRawData），4 个字节，值修改为 92 00 00 00，表示该节在文件中所占的大小。
- 成员 5（PointerToRawData），4 个字节，值修改为 00 06 00 00，表示该节在文件中的起始地址。

**计算文件中只读数据节起始地址：**

因为.rdata 是紧跟在.text 之后，而文件头和.text 已经占用 0x600（经过文件对齐)。

- 成员 6-9 共 12 个字节，暂时不用关心，都用 00 填充。

- 成员 10（Characteristics），4 个字节，值修改为 40 00 00 40。表示当前节可读等节属性（32 个二进制位各代表不同的含义）

至此，整个.rdata 节表完成。

最后是数据节。

- 成员 1（Name），8 个字节，表示该节的名称，这里循惯例使用.data，所以在地址 1f8h 处填写相应的 ASCII 码值 2E 64 61 74 61 00 00 00;
- 成员 2（VirtualSize），4 个字节，值修改为 00 10 00 00，表示该节数据映射到内存后所占的字节数，即有效内容所占的字节数。后面会给出相应的内容；
- 成员 3（VirtualAddress），4 个字节，值修改为 00 30 00 00，表示该节映射到内存中的起始地址是 00003000h。

**计算内存中数据节起始地址：**

因为.data 是紧跟在.rdata 之后，所以映射到内存后占的起始地址为 3000h（经过内存对齐）

- 成员 4（SizeOfRawData），4 个字节，值修改为 16 00 00 00，表示该节在文件中所占的大小。
- 成员 5（PointerToRawData），4 个字节，值修改为 00 08 00 00，表示该节在文件中的起始地址。

**计算文件中数据节起始地址：**

因为.data 是紧跟在.rdata 之后，而文件头和.text 节、.rdata 节已经占用 0x800（经过文件对齐）。

- 成员 6-9 共 12 个字节，暂时不用关心，都用 00 填充。
- 成员 10（Characteristics），4 个字节，值修改为 40 00 00 C0。表示当前节可读等节属性（32 个二进制位各代表不同的含义）

至此，整个.data 节表完成。

```
000000d0h: 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 ; ................
000000e0h: 00 00 00 00 00 00 40 00 00 10 00 00 00 02 00 00 ; ......@.........
000000f0h: 00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 ; ................
00000100h: 00 40 00 00 00 04 00 00 00 00 00 00 03 00 00 00 ; .@..............
00000110h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000120h: 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000130h: BB BB BB BB CC CC CC CC 00 00 00 00 00 00 00 00 ; 换换烫烫........
00000140h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000150h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000160h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000170h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000180h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000190h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000001a0h: 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00 ; .........text...
000001b0h: 26 00 00 00 00 10 00 00 26 00 00 00 00 04 00 00 ; &.......&.......
000001c0h: 00 00 00 00 00 00 00 00 00 00 00 00 60 00 00 20 ; ............`..
000001d0h: 2E 72 64 61 74 61 00 00 92 00 00 00 00 20 00 00 ; .rdata..?... ..
000001e0h: 92 00 00 00 00 06 00 00 00 00 00 00 00 00 00 00 ; ?...............
000001f0h: 00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00 ; ....@..@.data...
00000200h: 00 10 00 00 00 30 00 00 16 00 00 00 00 08 00 00 ; .....0..........
00000210h: 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0 ; ............@..?
```

因为要文件对齐，所以后面用 00 补齐到地址 3ffh 处。

PE 加载器根据节表加载程序的过程：

读取 IMAGE_FILE_HEADER 的 NumberOfSections 成员值，得到文件中节的数目。

读取 IMAGE_OPTIONAL_HEADER 的 SizeOfHeaders 成员值，将其作为节表的文件偏移，以此定位文件中的节表。

然后遍历节表（结构体数组）检查各成员值。

对于每个结构体，读取 PointerToRawData 成员值并定位到该文件偏移量，再读取 SizeOfRawData 成员值来决定映射内存的字节数。将 VirtualAddress 成员值加上 ImageBase 成员值就等于节起始的内存虚拟地址。然后把节映射进内存，并根据 Characteristics 成员值设置属性。

直到所有的节处理完毕。

(9) 下面构造文件的内容，即三个节。先看.text 节。

从地址 400h 开始，继续填写 512 个字节（0x0200）。

.text 节存放所有可执行的指令代码（机器码）。

先编写汇编指令，然后反汇编出机器码。

要实现的程序功能是弹出一个消息框，需要调用 MessageBoxA 函数，单击确定后程序要退出，需要调用 ExitProcess 函数。

实现该功能的汇编代码如下：

```
push 0 ; MessageBoxA 的第四个参数，即消息框的风格，这里传入 0。
push ? ? ? ? ;第三个参数，消息框的标题字符串所在的地址，需要计算。
push ? ? ? ? ;第二个参数，消息框的内容字符串所在的地址，需要计算。
push 0 ;第一个参数，消息框所属窗口句柄，这里填 0。
call ? ? ? ? ; 调用 MessageBoxA，实际是跳转到该函数的跳转指令所在地址，即下
               面的第 1 个 jmp 处
push 0 ; ExitProcess 函数的参数，程序退出码，传入 0.
call ? ? ? ? ; 调用 ExitProcess，实际是跳转到该函数的跳转指令所在地址，即下
               面的第 2 个 jmp 处
jmp ? ? ? ? ; 跳转到 MessageBoxA 的真正地址处。
jmp ? ? ? ? ; 跳转到 ExitProcess 的真正地址处。
```

由于代码中问号处的值需要等其它节完成后才能给出，这里暂时先跳过。

(10) 构造.rdata 节。

.rdata 节非常重要，也非常繁琐。从地址 600h 开始，继续插入 512 个字节（0x0200）。

这里需要手工打造导入表并存入.rdata 节中。通常情况下，导入表都是由编译器生成的。

导入表是一个 IMAGE_IMPORT_DESCRIPTOR 结构体数组。数组中的每个结构体包含了当前 PE 文件从某一个 DLL 库引入函数的相关信息。

先来计算导入表目录项。

前面在文件可选头 OptionalHeader 的成员 31 中，第 2 个表项 IMAGE_DATA_DIRECTORY 结构体中的两个成员的值暂时赋成了 BB BB BB BB 和 CC CC CC CC（第一个成员表示表的起始 RVA 地址，第二个成员表示该表的长度）。

本程序直接把导入表放到.rdata 节的起始地址处，所以导入表的起始地址也就是.rdata 节的起始地址。.rdata 紧随.text 之后，它的起始地址偏移应该为 PE 头的大小 0x1000+.text 的大小 0x1000=0x00002000。

因为程序需要从 2 个 DLL 库中导入函数（为了显示一个消息框，需要导入 user32.dll 库中的 MessageBoxA 函数；为了程序正常退出，需要导入 kernel32.dll 库中的 ExitProcess

函数），所以导入表中有 2 个数组元素分别针对这 2 个 DLL，再加上一个全零的数组元素作为结尾，共 3 个元素。

IMAGE_IMPORT_DESCRIPTOR 结构体有 5 个成员，都是 DOWRD 类型，所以大小为 4*5=20 个字节。因此本程序的导入表大小为（2+1）*20=60 个字节（0x0000003C）。

现在可以把地址 130h 处导入表目录项的第一个成员 BB BB BB BB 改为 00 20 00 00，第二个成员从 CC CC CC CC 改为 3C 00 00 00。

```
00000130h: 00 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00
```

接下来应该填写导入表的两个数组元素，分别对应需要导入的 user32.dll 和 kernel32.dll。但是在开始之前，需要先构造一组名称和相应的地址表。

先要构造**函数名称表**。也就是把要导入的所有函数名称依次保存到某个位置，然后再计算其 RVA，构造函数名称地址表。

为了方便，这里直接把需要用到的各个函数名和动态库名字符串放在导入表之后。

前面已经计算得到导入表的长度为 0x3c，所以字符串的位置应该保存到文件中的地址为：600h（导入表的起始文件偏移地址）+03ch（导入表长度）= 63ch。

在地址 063ch 处开始填写 00 00 4D 65 73 73 61 67 65 42 6F 78 41 00。

前两个字节应当填写函数序号，这里暂时用不到，直接填 0，也可以使用从动态库的导出表中查到的正确值。 接下来是“MessageBoxA” 字符串的 ASCII 码值，注意最后是以一个字节的 0x00 结尾。

如果程序还导入了同一 DLL 库中的其它函数，就按照同样的格式依次输入那些函数名。本程序只导入了这一个函数。

接在在后面填写 75 73 65 72 33 32 2E 64 6C 6C 00，也就是 DLL 库名（user32.dll）的 ASCII 码值。

这样就完成了一个导入库的函数名称表。

接着再以相同方式完成另一个 DLL 库的导入函数名称表，即“ExitProcess”字符串和“kernel32.dll”字符串。依次填写 00 00 45 78 69 74 50 72 6F 63 65 73 73 00 和 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00。

```
00000630h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4D 65 ; ..............Me
00000640h: 73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32 ; ssageBoxA.user32
00000650h: 2E 64 6C 6C 00 00 00 45 78 69 74 50 72 6F 63 65 ; .dll...ExitProce
00000660h: 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 ; ss.kernel32.dll.
```

再构造**函数名称地址表**。

函数名称地址表中存放了从某一个 DLL 库中导入的所有函数的函数名地址的 RVA。导入了几个 DLL 库，就需要有几个函数名称地址表。本程序共导入了两个 DLL 库，所以需要有两个函数名称地址表。

因为函数名称表起始地址为文件偏移 63Ch，长度是 0x34，所以函数名称地址表的起始为 63Ch+034h=670h。

在地址 670h 处填写 3C 20 00 00 00 00 00 00，这是 user32.dll 的函数名地址（63Ch 的 RVA）和表示当前 DLL 函数名地址结束的 4 字节 00；接着再填写 55 20 00 00 00 00 00 00，这是 kernel32.dll 的函数名地址（655h 的 RVA）和表示当前 DLL 函数名地址结束的 4

字节 00。

这样，可以知道 user32.dll 库函数名称地址表的起始文件偏移为 670h,对应的 RVA 是 2070h。Kernel32.dll 库函数名称地址表的起始文件偏移为 678h，对应的 RVA 是 2078h.

计算函数名称地址表

从 user32.dll 库中导入了 MessageBoxA。因为函数名称表已经构造完成，可以知道到它的文件偏移是 63Ch。根据文件偏移可以计算得到对应的 RVA 值。这涉及到内存对齐粒度和文件对齐粒度。

先是文件头经过文件对齐占 0x0400，经过内存对齐加载入内存后占 0x1000；然后是.text 节经文件对齐占 0x0200，经过内存对齐加载入内存后占 0x1000；文件偏移 600h 对应的内存 RVA 是 2000h，所以文件地址 63Ch 对应的 RVA 应该是 203Ch。

所以函数名称地址表中应当填写 3C 20 00 00 00 00 00 00。

因为 user32.dll 库中只导入了一个函数，所以在后面接着填写一个全零的 DWORD 值 0x00000000 表示结束。

```
00000670h: 3C 20 00 00 00 00 00 00 55 20 00 00 00 00 00 00 ; < ......U ......
```

现在再来看对应于 user32.dll 的导入表数组元素。它是一个 IMAGE_IMPORT_DESCRIPTOR 结构体，包含 5 个成员。

- 成员 1 OriginalFirstThunk，4 个字节，是指向一个 IMAGE_THUNK_DATA 结构体数组的 RVA。

这个 IMAGE_THUNK_DATA 结构体数组就是刚才构建的函数名称地址表。它存放了从某一个 DLL 库中导入的所有函数的函数名地址的 RVA，最后由一个全零 DWORD 值 0x00000000 结束。（不是直接记录函数名称，而是通过名称地址中转一下这个调用关系，思考一下这是为什么？）。

前面已经知道，user32.dll 的函数名称地址表的文件偏移地址是 670h,对应的 RVA 是 2070h,所以在地址 600h 处，填写 70 20 00 00。

- 成员 2 和成员 3，各占 4 个字节，暂时不用关心，用 00 填充。
- 成员 4 name，占 4 个字节，是指向 DLL 名字的 RVA。根据刚刚构建的导入函数名称表可以知道“user32.dll”这个 DLL 名称的文件偏移为 64Ah,对应的 RVA 值是 204Ah，所以在地址 60Ch 处填写 4A 20 00 00”。
- 成员 5 FirstThunk，占 4 个字节，也指向一个 IMAGE_THUNK_DATA 结构体数组的 RVA，但这个结构体数组的含义跟成员 1 不同。它将会保存所有导入函数在内存中的真实调用地址，而不是成员 1 指向的函数名称地址表，叫做函数调用地址表，也叫导入地址表（import address table，IAT）。IAT 中的内容是在 PE 文件被装载进内存时，由 PE 加载器把导入函数在内存中的真实地址填写进来的。这个表也是以一个全零的 DWORD 值作为结束标志。

本程序把 **IAT** 安排在函数名称地址表之后，所以它的起始文件偏移是 670h+010h=680h（函数名称地址表起始偏移地址是 670h，大小为 010h 个字节）,转换为 RVA 是 2080h，所以成员 5 填写 80 20 00 00”。

需要注意的是，虽然函数调用地址表的内容最终要由 PE 加载器来填充，只需指定该表的位置。但是由于该表以全零的 DWORD 值作为结束标记。所以需要先随便填入一个非零值，以免导致错误。

在地址 680h 处填入 11 00 00 00，之后是结束标记 00 00 00 00。

紧接着是第二个导入库的 IAT，在地址 688h 处填入 11 00 00 00，之后是结束标记 00 00 00 00。

最终完成的导入函数调用地址表如下：

```
00000680h: 11 00 00 00 00 00 00 00 11 00 00 00 00 00 00 00
```

接着再看对应于 kernel32.dll 的导入表数组元素。它同样也是一个 IMAGE_IMPORT_DESCRIPTOR 结构体，包含 5 个成员。

- 成员 1 OriginalFirstThunk，4 个字节。

前面已经知道,kernel32.dll 的函数名称地址表的文件偏移地址是 678h,对应的 RVA 就是 2078h,所以在地址 614h 处，填写 78 20 00 00。

- 成员 2 和成员 3，各占 4 个字节，暂时不用关心，用 00 填充。
- 成员 4 name，占 4 个字节，是指向 DLL 名字的 RVA。根据前面构建的导入函数名称表可以知道“kernel32.dll”这个 DLL 名称的文件偏移为 663h，对应的 RVA 值是 2063h，所以在地址 620h 处填写 63 20 00 00”。
- 成员 5 FirstThunk，占 4 个字节，填写 88 20 00 00，是 kernel32.dll 的 IAT 地址。

```
00000600h: 70 20 00 00 00 00 00 00 00 00 00 00 4A 20 00 00 ; p ..........J ..
00000610h: 80 20 00 00 78 20 00 00 00 00 00 00 00 00 00 00 ; € ..x ..........
00000620h: 63 20 00 00 88 20 00 00 00 00 00 00 00 00 00 00 ; c ..?..........
00000630h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4D 65 ; ..............Me
00000640h: 73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32 ; ssageBoxA.user32
00000650h: 2E 64 6C 6C 00 00 00 45 78 69 74 50 72 6F 63 65 ; .dll...ExitProce
00000660h: 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 ; ss.kernel32.dll.
00000670h: 3C 20 00 00 00 00 00 00 55 20 00 00 00 00 00 00 ; < ......U ......
00000680h: 11 00 00 00 00 00 00 00 11 00 00 00 00 00 00 00 ; ................
```

因为要文件对齐，所以后面用 00 补齐到地址 7ffh 处。

(11) 构造.data 节。

从地址 800h 开始，继续插入 512 个字节（0x200）。

.data 节中存放 MessageBoxA 需要用到的两个参数，字符串“PETEST”和“Hello World!”的 ASCII 码值。

```
00000800h: 50 45 54 45 53 54 2E 48 65 6C 6C 6F 2C 20 57 6F ; PETEST.Hello, Wo
00000810h: 72 6C 64 21 00 00 00 00 00 00 00 00 00 00 00 00 ; rld!............
```

因为要文件对齐，所以后面用 00 补齐到地址 9ffh 处。

（12）确定代码

首先把汇编代码中的两个字符串地址修正如下：

```
push 0 ; MessageBoxA 的第四个参数，即消息框的风格，这里传入 0。
push 0x403000 ;第三个参数，消息框的标题字符串所在的地址，需要计算。
push 0x403007 ;第二个参数，消息框的内容字符串所在的地址，需要计算。
push 0 ;第一个参数，消息框所属窗口句柄，这里填 0。
call ???? ; 调用 MessageBoxA，实际是跳转到该函数的跳转指令所在地址，即下
            面的第 1 个 jmp 处
push 0 ; ExitProcess 函数的参数，程序退出码，传入 0.
call ???? ; 调用 ExitProcess，实际是跳转到该函数的跳转指令所在地址，即下
```

面的第 2 个 jmp 处
jmp ？？？？ ；跳转到 MessageBoxA 的真正地址处。
jmp ？？？？ ；跳转到 ExitProcess 的真正地址处。

计算 MessageBoxA 两个字符串参数的地址。

这两个字符串“PETEST”和“HelloWorld！”都存放在.data(全局变量数据节)。它位于.rdata 之后，所以算出它的起始内存地址应该是 3000h。

- 文件头占 0x1000 字节，
- .text 节只有 0x0026 字节，内存对齐后为 0x1000，
- .rdata 节也不超过 0x1000，对齐后为 0x1000,

可知.data 节的起始内存地址在偏移为 1000h+1000h+1000h=3000h 处，程序的基址为 00400000h，所以.data 节的绝对内存地址为： 00400000h+3000h=00403000h。“PETEST”字符串放在这里，占 7 个字节，所以后面的“Hello World!”字符串的地址是 00403000h+7h=00403007h。

然后把汇编代码中的两个 jmp 目标地址修正如下：
push 0 ； MessageBoxA 的第四个参数，即消息框的风格，这里传入 0。
push 0x403000 ;第三个参数，消息框的标题字符串所在的地址，需要计算。
push 0x403007 ;第二个参数，消息框的内容字符串所在的地址，需要计算。
push 0 ;第一个参数，消息框所属窗口句柄，这里填 0。
call ？？？？ ；调用 MessageBoxA，实际是跳转到该函数的跳转指令所在地址，即下面的第 1 个 jmp 处
push 0 ; ExitProcess 函数的参数，程序退出码，传入 0.
call ？？？？ ；调用 ExitProcess，实际是跳转到该函数的跳转指令所在地址，即下面的第 2 个 jmp 处
jmp **dword ptr[0x402080]** ；跳转到 MessageBoxA 的真正地址处。
jmp **dword ptr[0x402088]** ；跳转到 ExitProcess 的真正地址处。

获得 jmp 目标地址：

这两个 jmp 跳转，是要跳转到函数的真正地址处，而真正地址是到 PE 加载时才能知道的。但是通过前面函数调用地址表（导入地址表 IAT）的构造，已经知道了各个导入函数真正地址所要填充的位置。PE 加载器会把函数调用的真实地址填写到 IAT 中，所以只需设计为 jmp dword ptr［被填充地址］，也就是跳转到被填充地址所指向的位置即可。

通过 IAT 可以得到 MessageBoxA 的填充地址为 2080h，这是 RVA 值,对应的绝对地址还要再加上基址，即：0x400000+0x2080=0x402080。同理可知，ExitProcess 函数的填充地址为 0x402088。

最后把两个 call 指令的目标地址修正如下：
push 0 ； MessageBoxA 的第四个参数，即消息框的风格，这里传入 0。
push 0x403000 ;第三个参数，消息框的标题字符串所在的地址，需要计算。
push 0x403007 ;第二个参数，消息框的内容字符串所在的地址，需要计算。
push 0 ;第一个参数，消息框所属窗口句柄，这里填 0。

call **40101A** ；调用 MessageBoxA，实际是跳转到该函数的跳转指令所在地址，即下面的第 1 个 jmp 处

push 0 ；ExitProcess 函数的参数，程序退出码，传入 0.

call **401020** ；调用 ExitProcess，实际是跳转到该函数的跳转指令所在地址，即下面的第 2 个 jmp 处

jmp dword ptr[0x402080] ；跳转到 MessageBoxA 的真正地址处。

jmp dword ptr[0x402088] ；跳转到 ExitProcess 的真正地址处。

计算 call 指令目标地址：

执行代码的起始地址即.text 节的起始地址，偏移为 1000h。后面各条指令的长度分别为：

push 0 ；指令长度为 2。

push 0x403000 ；指令长度为 5。

push 0x403007 ；指令长度为 5。

push 0 ；指令长度为 2。

call ？？？？ ；指令长度为 5。

push 0 ；指令长度为 2。

call ？？？？ ；指令长度为 5。

jmp dword ptr 指令长度为 6。

以上代码的总长度为 2+5+5+2+5+2+5=26（1Ah）。所以，紧随其后的两条 jmp 指令的地址偏移分别是 101Ah 和 1020h，分别再加上基址 400000h，得到其绝对地址分别为 40101Ah 和 401020h。

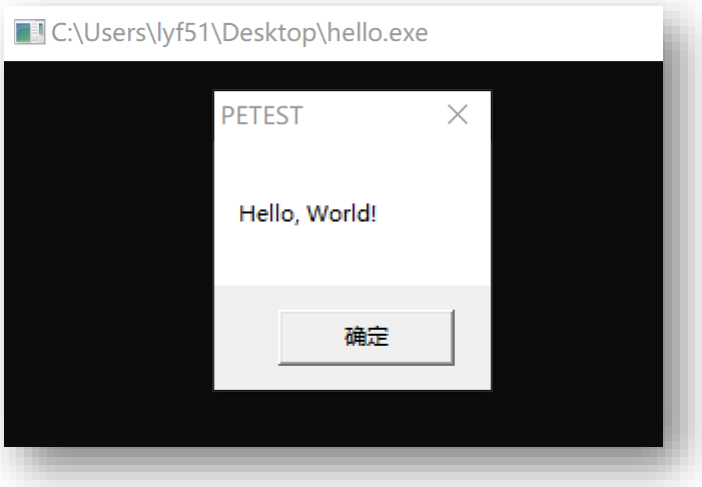
最后把得到的汇编代码翻译成机器码，结果为：

6A 00 68 00 30 40 00 68 07 30 40 00 6A 00 E8 07 00 00 00 6A 00 E8 06 00 00 00 FF 25 80 20 40 00 FF 25 88 20 40 00。

把这些内容填入.text 节（地址 400h 处），其余部分用 00 填充，直到地址 5ffh 处。

```
00000400h: 6A 00 68 00 30 40 00 68 07 30 40 00 6A 00 E8 07 ; j.h.0@.h.0@.j.?
00000410h: 00 00 00 6A 00 E8 06 00 00 00 FF 25 80 20 40 00 ; ...j.?...   %€ @.
00000420h: FF 25 88 20 40 00 00 00 00 00 00 00 00 00 00 00 ;  %?@.........
```

（13）保存文件，就得到了一个纯手工制作的 PE 文件。双击即可运行。



【实验原理】

```
typedef struct _IMAGE_DOS_HEADER {
    WORD e_magic;
    WORD e_cblp;
    WORD e_cp;
    WORD e_crlc;
    WORD e_cparhdr;
    WORD e_minalloc;
    WORD e_maxalloc;
    WORD e_ss;
    WORD e_sp;
    WORD e_csum;
    WORD e_ip;
    WORD e_cs;
    WORD e_lfarlc;
    WORD e_ovno;
    WORD e_res[4];
    WORD e_oemid;
    WORD e_oeminfo;
    WORD e_res2[10];
    LONG e_lfanew;
} IMAGE_DOS_HEADER

typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS

typedef struct _IMAGE_FILE_HEADER {
    WORD Machine;
    WORD NumberOfSections;
    DWORD TimeDateStamp;
    DWORD PointerToSymbolTable;
    DWORD NumberOfSymbols;
    WORD SizeOfOptionalHeader;
    WORD Characteristics;
} IMAGE_FILE_HEADER

typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD Magic;
    BYTE MajorLinkerVersion;
    BYTE MinorLinkerVersion;
```

```
    DWORD SizeOfCode;
    DWORD SizeOfInitializedData;
    DWORD SizeOfUninitializedData;
    DWORD AddressOfEntryPoint;
    DWORD BaseOfCode;
    DWORD BaseOfData;
    DWORD ImageBase;
    DWORD SectionAlignment;
    DWORD FileAlignment;
    WORD MajorOperatingSystemVersion;
    WORD MinorOperatingSystemVersion;
    WORD MajorImageVersion;
    WORD MinorImageVersion;
    WORD MajorSubsystemVersion;
    WORD MinorSubsystemVersion;
    DWORD Reserved1;
    DWORD SizeOfImage;
    DWORD SizeOfHeaders;
    DWORD CheckSum;
    WORD Subsystem;
    WORD DllCharacteristics;
    DWORD SizeOfStackReserve;
    DWORD SizeOfStackCommit;
    DWORD SizeOfHeapReserve;
    DWORD SizeOfHeapCommit;
    DWORD LoaderFlags;
    DWORD NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER
```

IMAGE_DATA_DIRECTORY 结构体

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

16 个目录表：

IMAGE_DIRECTORY_ENTRY_EXPORT (0) 导出目录，用于 DLL
IMAGE_DIRECTORY_ENTRY_IMPORT (1) 导入目录
IMAGE_DIRECTORY_ENTRY_RESOURCE (2) 资源目录
IMAGE_DIRECTORY_ENTRY_EXCEPTION (3) 异常目录
IMAGE_DIRECTORY_ENTRY_SECURITY (4) 安全目录
IMAGE_DIRECTORY_ENTRY_BASERELOC (5) 重定位表

IMAGE_DIRECTORY_ENTRY_DEBUG (6) 调试目录
IMAGE_DIRECTORY_ENTRY_COPYRIGHT (7) 描述版权串
IMAGE_DIRECTORY_ENTRY_GLOBALPTR (8) 机器值
IMAGE_DIRECTORY_ENTRY_TLS (9) 本地线程存储目录
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG (10) 载入配置目录
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (11) 绑定导入表目录
IMAGE_DIRECTORY_ENTRY_IAT (12) 输入地址表目录
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT(13) 延迟加载导入描述目录
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR(14) COM 运行时描述目录

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER
```

bit 5 (IMAGE_SCN_CNT_CODE)，置 1，节内包含可执行代码。
bit 6 (IMAGE_SCN_CNT_INITIALIZED_DATA)置 1，节内包含的数据在执行前是确定的。
bit 7 (IMAGE_SCN_CNT_UNINITIALIZED_DATA) 置 1，本节包含未初始化的数据，执行前即将被初始化为 0。一般是 BSS.
bit 9 (IMAGE_SCN_LNK_INFO) 置 1,节内不包含映象数据除了注释,描述或者其他文档外，是一个目标文件的一部分，可能是针对链接器的信息。比如哪个库被需要。
bit 11 (IMAGE_SCN_LNK_REMOVE) 置 1，在可执行文件链接后，作为文件一部分的数据被清除。
bit 12 (IMAGE_SCN_LNK_COMDAT) 置 1，节包含公共块数据，是某个顺序的打包的函数。
bit 15 (IMAGE_SCN_MEM_FARDATA) 置 1，不确定。
bit 17 (IMAGE_SCN_MEM_PURGEABLE) 置 1，节的数据是可清除的。
bit 18 (IMAGE_SCN_MEM_LOCKED) 置 1，节不可以在内存内移动。
bit 19 (IMAGE_SCN_MEM_PRELOAD)置 1， 节必须在执行开始前调入。
bits 20-23 指定对齐。一般是库文件的对象对齐。
bit 24 (IMAGE_SCN_LNK_NRELOC_OVFL) 置 1， 节包含扩展的重定位。
bit 25 (IMAGE_SCN_MEM_DISCARDABLE) 置 1，进程开始后节的数据不再需要。
bit 26 (IMAGE_SCN_MEM_NOT_CACHED) 置 1，节的 数据不得缓存。
bit 27 (IMAGE_SCN_MEM_NOT_PAGED) 置 1，节的 数据不得交换出去。

bit 28 (IMAGE_SCN_MEM_SHARED) 置 1，节的数据在所有映象例程内共享，如 DLL 的初始化数据。
bit 29 (IMAGE_SCN_MEM_EXECUTE) 置 1，进程得到“执行”访问节内存。
bit 30 (IMAGE_SCN_MEM_READ) 置 1，进程得到“读出”访问节内存。
bit 31 (IMAGE_SCN_MEM_WRITE)置 1，进程得到“写入”访问节内存。

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics;
        PIMAGE_THUNK_DATA OriginalFirstThunk;
    };
    DWORD TimeDateStamp;
    DWORD ForwarderChain;
    DWORD Name;
    PIMAGE_THUNK_DATA FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR,

typedef struct _IMAGE_THUNK_DATA {
    union {
        PBYTE ForwarderString;
        PDWORD Function;
        DWORD Ordinal;
        PIMAGE_IMPORT_BY_NAME AddressOfData;
    } u1;
} IMAGE_THUNK_DATA
```