

# PE文件结构

## 可执行文件 (executable file)

- 可以由操作系统进行加载、执行的文件
- 在不同的操作系统环境下，可执行文件的格式不一样
- 二进制文件，不同于txt、word、excel等文本文件

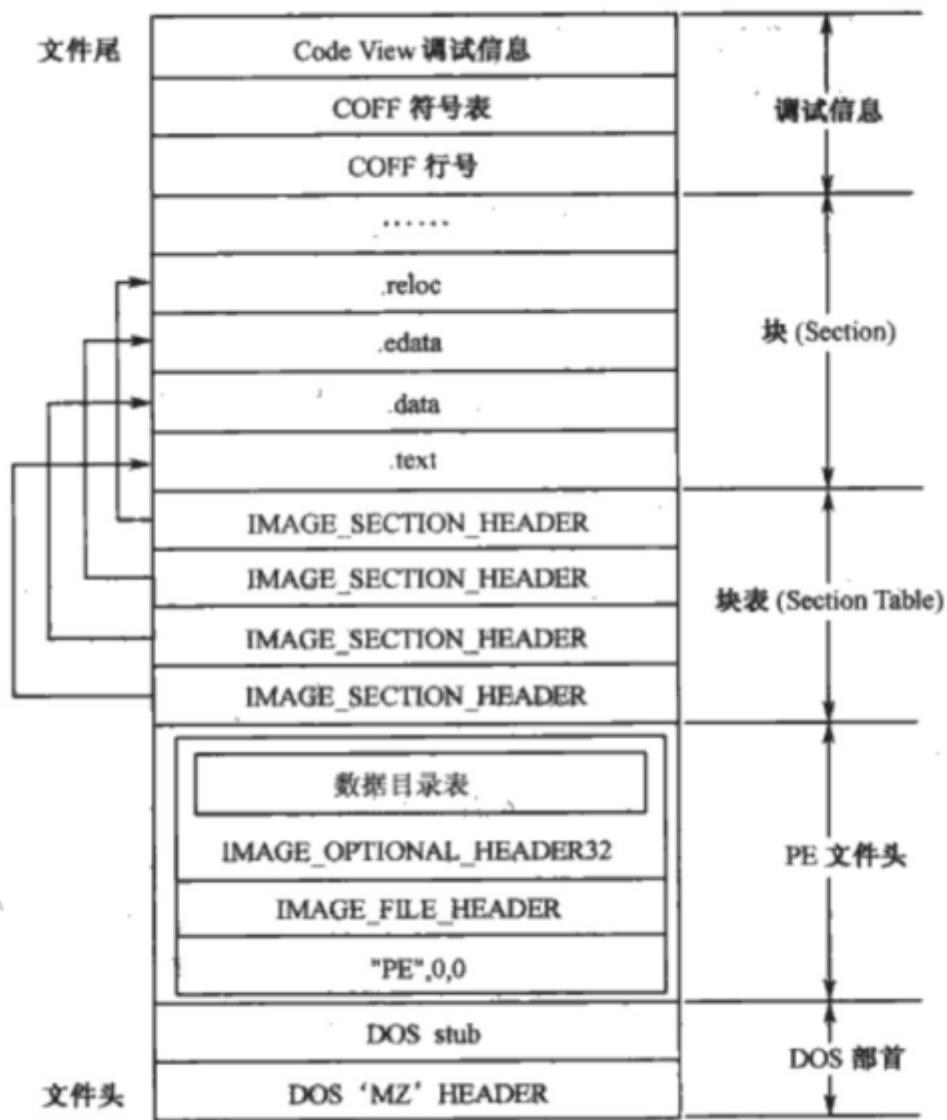
Windows系统可执行文件使用PE文件格式，Linux系统可执行文件使用ELF文件格式。

PE (Portable Executable File Format) 可移植可执行文件结构。

在Windows操作系统下，可执行程序可以是.com文件、.exe文件、.sys文件、.dll文件、.scr文件等类型文件。

- com文件在IBM PC早期出现，格式主要用于**命令行应用程序**、最大65,280字节，与MS-DOS操作系统的可执行文件兼容。
- .exe,.dll,.sys文件使用的是PE文件结构

## PE文件结构



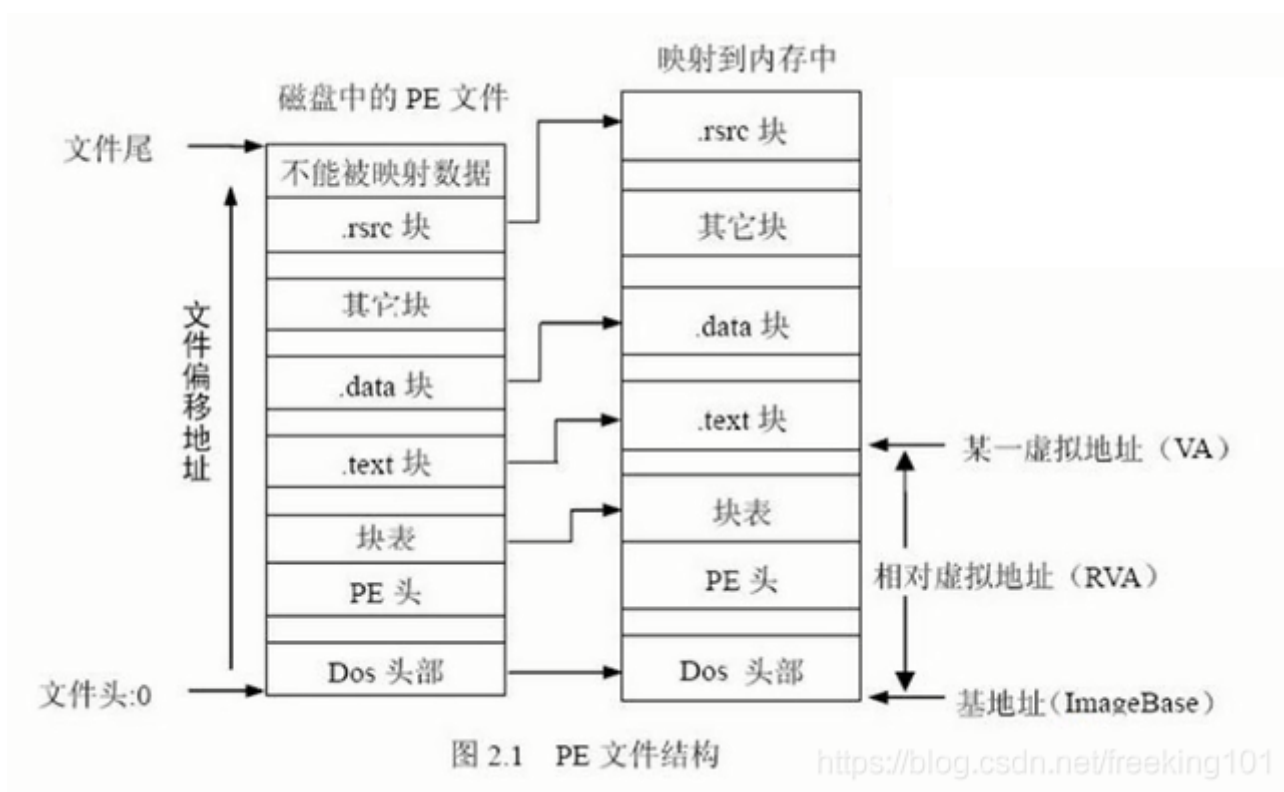
PE 文件框架结构

<https://blog.csdn.net/freeking101>

PE文件使用的是一个平面地址空间，所有代码和数据都合并在一起，组成了一个很大的结构。文件的内容被分割为不同的节(Section，也叫做块、区块等)。区块中包含代码或数据，各个区块按页边界对齐。

节：

- 代码节、数据节
- 各个节按**页边界对齐**
- 节是一个**连续结构**，没有大小限制
- 每个节都有自己的内存属性



扇区 - 硬盘的最小存储存储单位 (Sector)。一般每个扇区储存512字节 (相当于0.5KB)。

## 模块

当PE文件通过Windows加载器载入内存后，内存中的版本称为**模块(Module)**。映射文件的起始地址称为**模块句柄(hModule)**，这个初始内存地址也称为**基地址(ImageBase)**。

内存中的**模块**代表进程将这个可执行文件所需要的代码、数据、资源、输入表、输出表及其它有用的数据结构都放在一个**连续的内存节**中。

00400000	00001000	proc		PE 文件头
00401000	00001000	proc	.text	代码
00402000	00001000	proc	.rdata	□入表
00403000	00001000	proc	.data	数据
00410000	00005000			
004E0000	00006000			
006E0000	00005000			
75580000	00001000	KERNEL32		PE 文件头
75590000	00064000	KERNEL32	.text	代码
75600000	0002F000	KERNEL32	.rdata	□入表, 输出表
75630000	00001000	KERNEL32	.data	数据
75640000	00001000	KERNEL32	.rsrc	资源
75650000	00005000	KERNEL32	.reloc	定位
76830000	00001000	KERNELBASE		PE 文件头
76831000	001C3000	KERNELBASE	.text	代码, 输出表
769F4000	00004000	KERNELBASE	.data	数据
769F8000	00006000	KERNELBASE	.idata	□入表
769FE000	00001000	KERNELBASE	.didat	
769FF000	00001000	KERNELBASE	.rsrc	资源
76A00000	0002A000	KERNELBASE	.reloc	定位

模块句柄：Windows将Module的**基地址**作为Module的实例句柄。

- GetModuleHandle获得DLL句柄，通过句柄访问DLL Module的内容。

## 虚拟内存地址空间

每个程序都有自己的虚拟内存地址空间，虚拟空间的内存地址称为虚拟地址(Virtual Address, VA)。

- 在一台32位系统上，**虚拟地址空间为4GB**。
- 不同进程的虚拟地址空间是相互隔离的。
- 一个进程中的虚拟地址空间通常有多个PE文件结构，包括dll文件、操作系统内核文件等。

## 虚拟地址

PE文件被系统加载器映射到内存中，每个程序都有自己的虚拟空间，这个虚拟空间的内存地址成为虚拟地址。

## 相对虚拟地址

在可执行文件中，有许多地方需要内存地址。例如，引用全局变量时需要指定它的地址。

尽管PE文件有一个首选的载入地址(基地址)，但PE文件可以载入到进程空间任何地方相对虚拟地址，所以不能依赖PE的载入点，必须有一个方法指定地址。

为了避免绝对内存地址，引入了相对虚拟地址(RVA)的概念，RVA是**相对于PE文件载入地址**的偏移位置。假设一个EXE文件从400000h处**载入内存**，**代码节起始地址**为401000h，代码节起始地址的RVA计算方法如下：目标地址401000h - 载入地址400000h = RVA 1000h

将RVA转换成虚拟地址VA的过程，用实际的载入地址ImageBase加相对虚拟地址RVA。虚拟地址(VA)=基地址(ImageBase)+相对虚拟地址(RVA)。

## 文件偏移地址

PE文件储存在磁盘中，某个数据的位置相对于文件头的偏移量称为文件偏移地址(File Offset)或物理地址(RAW Offset)。

- 文件偏移地址从PE文件的第1个字节开始计数，起始值为0。

## PE文件结构

### PE 文件结构说明：

1. **DOS头** 是用来兼容 MS-DOS 操作系统的，目的是当这个文件在 MS-DOS 上运行时提示一段文字，大部分情况下是：This program cannot be run in DOS mode. 还有一个目的，就是指明 NT 头在文件中的位置。
2. **NT头** 包含 windows PE 文件的主要信息，其中包括一个 'PE' 字样的签名，**PE文件头 (IMAGE\_FILE\_HEADER)** 和 **PE可选头 (IMAGE\_OPTIONAL\_HEADER32)** 。
3. **节表**：是 PE 文件后续节的描述，windows 根据节表的描述加载每个节。
4. **节**：每个节实际上是一个容器，可以包含 代码、数据 等等，每个节可以有独立的内存权限，比如代码节默认有读/执行权限，节的名字和数量可以自己定义。

# DOS文件头

每个PE文件都是以一个16位的DOS程序开始的，有了它，程序一旦在DOS下执行，DOS就能识别出这是一个有效的执行体，然后运行紧随MZ header的DOS stub  
DOS MZ头与DOS stub合称为DOS文件头

PE文件的第一个字节位于一个传统的MS-DOS头部，称作IMAGE\_DOS\_HEADER，其结构如下：

```
IMAGE_DOS_HEADER_STRUCT{
+0h   e_magic      WORD  ?           ;DOS 可执行文件标记 "MZ"
+2h   e_cblp       WORD  ?
+4h   e_cp         WORD  ?
+6h   e_crlc       WORD  ?
+8h   e_cparhdr    WORD  ?
+0ah  e_minalloc   WORD  ?
+0ch  e_maxalloc   WORD  ?
+0eh  e_ss         WORD  ?
+10h  e_sp         WORD  ?
+12h  e_csum       WORD  ?
+14h  e_ip         WORD  ?           ;DOS 代码入口 IP
+16h  e_cs         WORD  ?           ;DOS 代码入口 CS
+18h  e_lfarlc     WORD  ?
+1ah  e_ovno       WORD  ?
+1ch  e_res        WORD  4 dup(?)
+24h  e_oemid      WORD  ?
+26h  e_oeminfo    WORD  ?
+28h  e_res2       WORD  10 dup(?)
+3ch  e_lfanew     DWORD ?           ;指向 PE 文件头"PE",0,0
} IMAGE_DOS_HEADER_ENDS
```

有两个字段比较重要，分别是e\_magic和e\_lfanew。e\_magic的值需要被设置为5A4Dh。  
e\_lfanew字段是真正的PE文件头的相对偏移，指出真正PE头的文件偏移位置，占用4字节，位于从文件开始偏移3Ch字节处。

用十六进制编辑器（WinHex、Hex Workshop 等带偏移量显示功能的尤佳）打开随书文件中的示例程序 PE.exe，定位在文件起始位置，此处就是 MS-DOS 头部，如图 11.3 所示。文件的第 1 个字符“MZ”就是 e\_magic 字段；偏移量 3Ch 就是 e\_lfanew 的值，在这里显示为“B0 00 00 00”。因为 Intel CPU 属于 Little-Endian 类，字符储存时低位在前，高位在后，所以，将次序恢复后，e\_lfanew 的值为 000000B0h，这个值就是真正的 PE 文件头偏移量。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ!.....yy...
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	?.....@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	B0	00	00	00	.....?...
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	81	4C	CD	21	54	68	..?.???L?Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0B	0D	0A	24	00	00	00	00	00	00	00	mode....\$.....
00000080	5D	17	1D	0B	19	76	73	88	19	76	73	88	19	76	73	88	]..?vs?vs?vs?
00000090	19	76	73	88	0A	76	73	88	E5	56	61	88	18	76	73	88	.vs?vs端Va?vs?
000000A0	57	69	63	68	19	76	73	88	00	00	00	00	00	00	00	00	Rich.vs?.....
000000B0	50	45	00	00	4C	01	03	00	2C	97	B8	3D	00	00	00	00	PE..L....楼=....

图 11.3 查看 PE 文件 MS-DOS 头部

# PE文件头

PE文件头(PE Header)紧跟在DOS stub的后面。PE header是PE相关结构NT映像头 (IMAGE\_NT\_HEADERS) 的简称，里面包含很多PE装载器能用到的重要字段。

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

0:000> dt /r1 ntdll! IMAGE\_NT\_HEADERS notepad+e0

+0x000 Signature	: 0x4550	PE 签名
+0x004 FileHeader	: IMAGE_FILE_HEADER	PE 文件头
+0x000 Machine	: 0x14c	
+0x002 NumberOfSections	: 3	
+0x004 TimeDateStamp	: 0x48025287	
+0x008 PointerToSymbolTable	: 0	
+0x00c NumberOfSymbols	: 0	
+0x010 SizeOfOptionalHeader	: 0xe0	
+0x012 Characteristics	: 0x10f	
+0x018 OptionalHeader	: IMAGE_OPTIONAL_HEADER	PE 可选头
+0x000 Magic	: 0x10b	
+0x002 MajorLinkerVersion	: 0x7	
+0x003 MinorLinkerVersion	: 0xa	
+0x004 SizeOfCode	: 0x7800	
+0x008 SizeOfInitializedData	: 0x8800	
+0x00c SizeOfUninitializedData	: 0	
+0x010 AddressOfEntryPoint	: 0x739d	
+0x014 BaseOfCode	: 0x1000	
+0x018 BaseOfData	: 0x9000	
+0x01c ImageBase	: 0x1000000	
+0x020 SectionAlignment	: 0x1000	
+0x024 FileAlignment	: 0x200	
+0x028 MajorOperatingSystemVersion	: 5	
+0x02a MinorOperatingSystemVersion	: 1	
+0x02c MajorImageVersion	: 5	
+0x02e MinorImageVersion	: 1	
+0x030 MajorSubsystemVersion	: 4	
+0x032 MinorSubsystemVersion	: 0	
+0x034 Win32VersionValue	: 0	
+0x038 SizeOfImage	: 0x13000	
+0x03c SizeOfHeaders	: 0x400	
+0x040 CheckSum	: 0x18ada	
+0x044 Subsystem	: 2	
+0x046 DllCharacteristics	: 0x8000	
+0x048 SizeOfStackReserve	: 0x40000	
+0x04c SizeOfStackCommit	: 0x11000	
+0x050 SizeOfHeapReserve	: 0x100000	
+0x054 SizeOfHeapCommit	: 0x1000	
+0x058 LoaderFlags	: 0	
+0x05c NumberOfRvaAndSizes	: 0x10	
+0x060 DataDirectory	: [16] _IMAGE_DATA_DIRECTORY	

每个字段相对于所属结构体的偏移，加上所属结构体的偏移位置，就是字段的 RVA

#### RVA 示例：

**Machine** 字段：相对于 **\_IMAGE\_FILE\_HEADER** 结构体的偏移为 0x000，结构体 **\_IMAGE\_FILE\_HEADER** 相对于 PE 头的偏移为 0x004，所以 **Machine** 的 **RVA = 0x000 + 0x004 = 0x004**，即位于 PE 头从 0 开始数的第 4 个字节，长度是 1 个字(2 个字节)，即 0x014C，Intel CPU 是小端序(低地址对应低位，高地址对应高位)，所以内存里面表示为 4C 01

#### RVA 示例：

**Magic** 字段：相对于 **\_IMAGE\_OPTIONAL\_HEADER** 结构体的偏移为 0x000，结构体 **\_IMAGE\_FILE\_HEADER** 相对于 PE 头的偏移为 0x018，所以 **Magic** 的 **RVA = 0x000 + 0x018 = 0x018**，即位于 PE 头从 0 开始数的第 24 个字节，长度是 1 个字(2 个字节)，即 0x010B，Intel CPU 是小端序(低地址对应低位，高地址对应高位)，所以内存里面表示为 0B 01

<https://blog.csdn.net/freeking101>

- 当执行体在支持 PE 文件结构的操作系统中执行时，PE 装载器就从 **IMAGE\_DOS\_HEADER** 结构的 **e\_lfanew** 字段找到 PE Header 的起始偏移量，加上基址，得到 PE 文件头的指针
- $PNTHeader = ImageBase + DosHeader \rightarrow e\_lfanew$



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ.....
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	.....@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	30	00	00	00	.....
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	.....!...L...Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....\$. ....
00000080	C5	7E	C6	DE	81	1F	A8	8D	81	1F	A8	8D	81	1F	A8	8D	.~.....
00000090	0F	00	BB	8D	87	1F	A8	8D	7D	3F	BA	8D	82	1F	A8	8D	.....}?. ....
000000A0	52	69	63	68	81	1F	A8	8D	00	00	00	00	00	00	00	00	Rich.....
000000B0	50	45	00	00	4C	01	03	00	15	9C	82	5E	00	00	00	00	PE...L....^....
000000C0	00	00	00	00	E0	00	0F	01	0B	01	05	0C	00	02	00	00	.....
000000D0	00	04	00	00	00	00	00	00	00	10	00	00	00	10	00	00	.....

## Signature字段

PE文件中的Signature字段被设置为0x00004550，ASCII码为PE00，是文件头的开始，e\_lfanew正是指向它。

## IMAGE\_FILE\_HEADER字段

映像文件头结构包含PE文件的一些基本信息，并指出了IMAGE\_OPTIONAL\_HEADER的大小

```

IMAGE_FILE_HEADER STRUCT
+04h Machine          WORD      ?      ;运行平台
+06h NumberOfSections WORD      ?      ;文件的区块数
+08h TimeDateStamp    DWORD     ?      ;文件创建日期和时间
+0Ch PointerToSymbolTable  DWORD  ?      ;指向符号表（用于调试）
+10h NumberOfSymbols  DWORD     ?      ;符号表中符号的个数（用于调试）
+14h SizeOfOptionalHeader WORD    ?      ;IMAGE_OPTIONAL_HEADER32 结构的大小
+16h Characteristics  WORD      ?      ;文件属性
IMAGE_FILE_HEADER ENDS

```

用十六进制工具查看 IMAGE\_FILE\_HEADER 结构的情况，如图 11.4 所示，图中的标号对应于以下字段。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000000B0	50	45	00	00	4C	01	03	00	2C	97	B8	3D	00	00	00	00	PE..L....核=.....
000000C0	00	00	00	00	E0	00	0F	01	0B	01	05	0C	00	02	00	00	....?.....

## Machine：可执行文件的目标CPU类型

PE文件可以在多种机器上使用，不同平台上指令的机器码不同。

## NumberOfSections：节的数目

节表紧跟在IMAGE\_NT\_HEADER后定义

## TimeDateStamp：文件的创建时间

## Characteristics：文件属性

特征值	含 义
0001h	文件中不存在重定位信息
0002h	文件可执行。如果为 0，一般是链接时出问题了
0004h	行号信息被移去
0008h	符号信息被移去
0020h	应用程序可以处理超过 2GB 的地址。该功能是从 NT SP3 开始被支持的。因为大部分数据库服务器需要很大的内存，而 NT 仅提供 2GB 给应用程序，所以从 NT SP3 开始，通过加载 /3GB 参数，可以使应用程序被分配 2~3GB 区域的地址，而该处原来属于系统内存区
0080h	处理机的低位字节是相反的
0100h	目标平台为 32 位机器
0200h	.DBG 文件的调试信息被移去
0400h	如果映像文件在可移动介质中，则先复制到交换文件中再运行
0800h	如果映像文件在网络中，则复制到交换文件后才运行
1000h	系统文件
2000h	文件是 DLL 文件
4000h	文件只能运行在单处理器上
8000h	处理机的高位字节是相反的

判断都有哪些文件属性：按比特位进行与操作

## IMAGE\_OPTIONAL\_HEADER结构

可选映像头(IMAGE\_OPTIONAL\_HEADER)结构中定义了更多的PE文件数据，IMAGE\_FILE\_HEADER和IMAGE\_OPTIONAL\_HEADER合在一起构成PE文件头。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000000C0	00	00	00	00	E0	00	0F	01	0B	01	05	0C	00	02	00	00	.....?
000000D0	00	04	00	00	00	00	00	00	00	10	00	00	00	10	00	00	.....
000000E0	00	20	00	00	00	00	40	00	00	10	00	00	00	02	00	00	.....@.....
000000F0	04	00	00	00	04	00	00	00	04	00	00	00	00	00	00	00	.....
00000100	3B	30	00	00	00	04	00	00	A7	20	00	00	02	00	00	00	80.....?
00000110	00	00	10	00	00	10	00	00	00	00	10	00	00	10	00	00	.....
00000120	00	00	00	00	10	00	00	00	00	00	00	00	00	00	00	00	.....

图 11.5 IMAGE\_OPTIONAL\_HEADER32 结构

1. Magic (WORD)：这是一个标记字，说明文件是ROM映像(0107h)还是普通可执行的映像(010Bh)
2. MajorLinkerVersion (BYTE)：链接程序的主版本号
3. MinorLinkerVersion(BYTE)：链接程序的次版本号
4. SizeOfCode(DWORD)：代码段的大小
5. SizeOfInitializedData(DWORD)：已初始化数据块的大小：
6. SizeOfUninitializedData(DWORD)：未初始化数据块的大小：
7. **AddressOfEntryPoint** (DWORD) 程序执行的入口RVA



8. BaseOfCode (DWORD) 代码段的起始RVA
9. BaseOfData (DWORD) 数据段的起始RVA
10. ImageBase (DWORD) 文件在内存中的载入地址, 除了imagebase, 其他都是RVA
11. SectionAlignment(DWORD): 内存上区块间的对齐大小, 默认的对齐尺寸是CPU的页尺寸 00001000h 4096字节
12. FileAlignment(DWORD): 硬盘上区块间的对齐大小 00000200h 512字节
13. MajorOperatingSystemVersion (WORD) PE文件执行所需要的Windows系统最低版本号的主版本号
14. MinorOperatingSystemVersion (WORD) PE文件执行所需要的Windows系统最低版本号的次版本号
15. MajorImageVersion(WORD)程序的主版本号, 由程序员自己定义
16. MinorImageVersion(WORD)程序的次版本号, 由程序员自己定义
17. MajorSubsystemVersion(WORD)PE文件所要求最低的子系统的版本号的主版本号
18. MinorSubsystemVersion(WORD)PE文件所要求最低的子系统的版本号的次版本号
19. Win32VersionValue(DWORD): 通常被设置为0IMAGE\_OPTIONAL\_HEADER
20. SizeOfImage (DWORD) : 映像载入内存后的总大小, 是指载入文件从ImageBase到最后一个块的大小
21. SizeOfHeaders (DWORD) : MS-DOS头部、PE文件头、区块表的总尺寸
22. SizeOfStackReserve (DWORD)
23. SizeOfStackCommit (DWORD)
24. SizeOfHeapReserve (DWORD)
25. SizeOfHeapCommit (DWORD) 栈、堆的设置信息
26. LoaderFlags (DWORD) :与调试有关
27. 数据目录表: 定位PE文件的导入表、导出表、资源的时候, 需要用到数据目录
  - VirtualAddress: 该目录的相对虚拟地址
  - Size: 该目录的大小