



可执行文件格式

本次课程支撑的毕业要求指标点

◆ 毕业要求3-3:

充分理解信息安全领域软硬件系统的基础上，能够设计或开发满足特定需求和约束条件的信息安全系统、模块或算法流程，并能够进行系统级优化。

EXE-PE文件的格式

简介

- ◆ 在DOS环境下有四种基本的可执行文件格式
 - ▶ 批处理文件，以.BAT结尾的文件
 - ▶ 设备驱动文件，是以.SYS结尾的文件，如CONFIG.SYS
 - ▶ COM文件，是以.COM结尾的纯代码文件
 - 没有文件头部分，缺省情况下总是从0x100H处开始执行，没有重定位项，所有代码和数据必须控制在64K以内
 - ▶ EXE文件，是以.EXE结尾的文件
 - 文件以英文字母“MZ”开头，通常称之为MZ文件
 - MZ文件有一个文件头，用来指出每个段的定义，以及重定位表。.EXE文件摆脱了代码大小最多不能超过64K的限制，是DOS下最主要的文件格式
- ◆ 在Windows 3.0/3.1的可执行文件，在MZ文件头之后又有一个以“NE”开始的文件头，称之为NE文件
- ◆ 在Win32位平台可执行文件格式：可移植的可执行文件(Portable Executable File)格式，即PE格式。MZ文件头之后是一个以“PE”开始的文件头

EXE文件的格式

MZ文件格式-Mark Zbikowski

- ◆ .EXE文件由三部分构成：文件头、重定位表和二进制代码
- ◆ 允许代码、数据、堆栈分别处于不同的段，每一段都可以是64KB.

		偏移	大小(字节)	描述
确定MZ文件的大小 以大小为512B的页为存储单位	→	00	2	EXE文件类型标记: 4D5Ah(ASCII字符MZ)
		02	2	文件最后一个扇区的字节数
		04	2	文件的总扇区(页)数 文件的大小=(总扇区数-1)×512+最后一个扇区的字节数
确定代码的开始处	→	06	2	重定位项的个数
		08	2	EXE文件头的大小(16字节的倍数)
		0A	2	最小分配数(16字节的倍数)
		0C	2	最大分配数(16字节的倍数)
		0E	2	初始化堆栈段(SS初值)
执行代码的入口地址 重定位表的指针链表 (比如调用C的库函数)	→	10	2	初始化堆栈指针(SP初值)
		12	2	补码校验和
	→	14	2	初始代码段指针(IP初值)
	→	16	2	初始代码段段地址(CS初值)
	→	18	2	定位表的偏移地址(第一个重定位项的偏移量)
		1A	2	连接程序产生的覆盖号

加载EXE文件

调用C的库函数

程序编译后：

```
0000:0000    9A78563412    call far 1234:5678
```

程序加载器的重定位工作，就是将程序中需要重定位的地方，都加上程序的加载地址。

这个程序被加载到了内存中的1111段处。那么完成重定位后，代码应该是这样：

```
1111:0000    9A78564523    call far 2345:5678
```

NE文件格式

- ◆ **NE**是**New Executable**的缩写，是16位Windows可执行文件的标准格式，这种格式基本上没用了
- ◆ **NE**在**MZ**文件头之后添加了一个以“**NE**”开始的文件头

PE文件格式

- ◆ Win32可执行文件，如*.EXE、*.DLL、*.OCX等，都是PE格式
- ◆ PE的意思就是Portable Executable(可移植、可执行)，它是Win32可执行文件的标准格式
- ◆ 由于大量的EXE文件被执行，且传播的可能性最大，因此，Win32病毒感染文件时，基本上都会将EXE文件作为目标

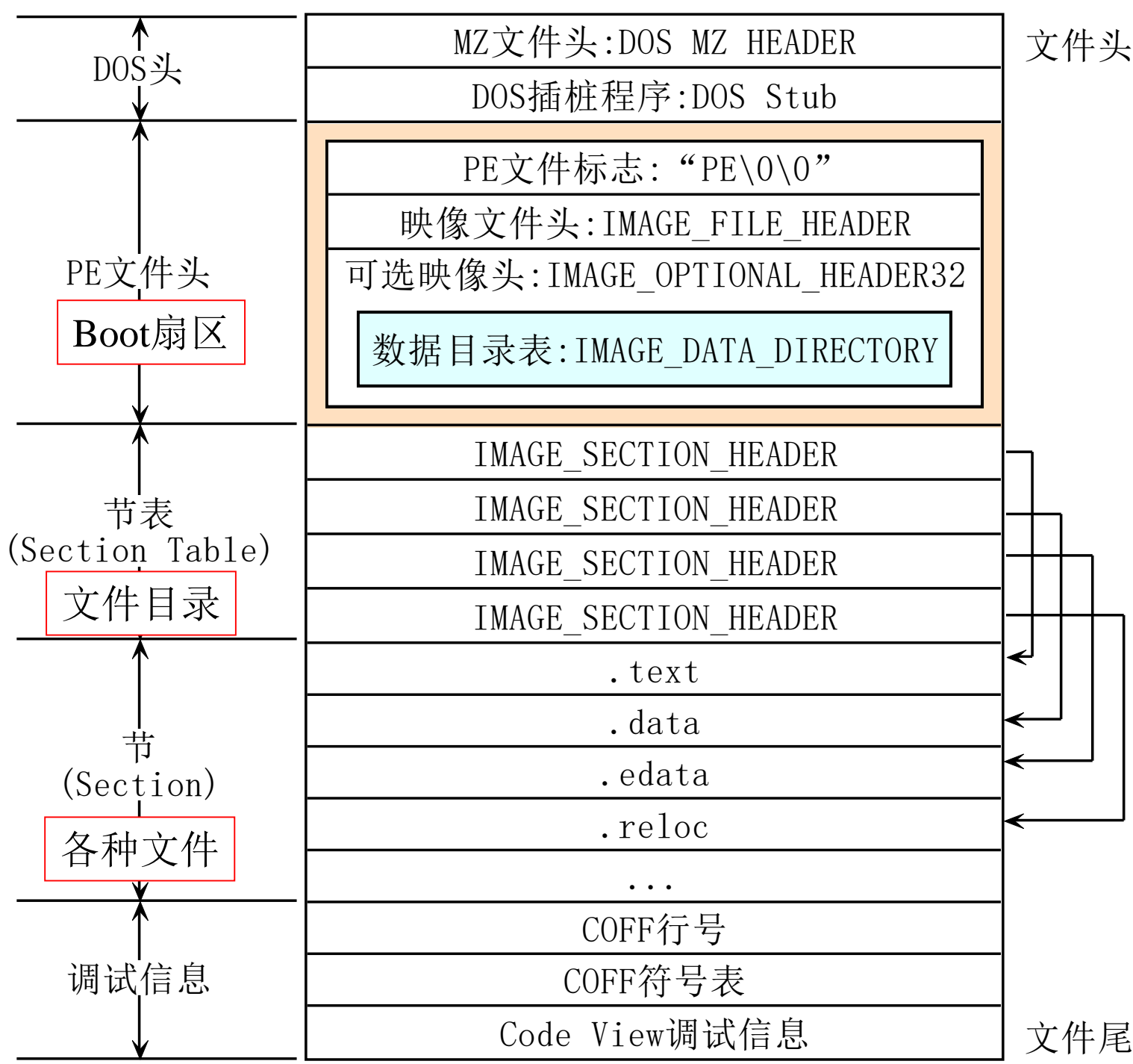
计算机病毒也是程序或者程序代码，而且也是可执行的，否则无法感染、破坏、隐藏等，其病毒文件也是遵循PE的格式结构。

病毒

- ◆ 一般来说，病毒往往先于**HOST**程序获得控制权。运行**Win32**病毒的一般流程示意如下：
- ◆ ①用户点击或系统自动运行**HOST**程序；
- ◆ ②装载**HOST**程序到内存；
- ◆ ③通过PE文件中的**AddressOfEntryPoint+ImageBase**，定位第一条语句的位置(程序入口)；
- ◆ ④从第一条语句开始执行(这时执行的其实是病毒代码)；
- ◆ ⑤病毒主体代码执行完毕，将控制权交给**HOST**程序原来的入口代码；
- ◆ ⑥**HOST**程序继续执行。

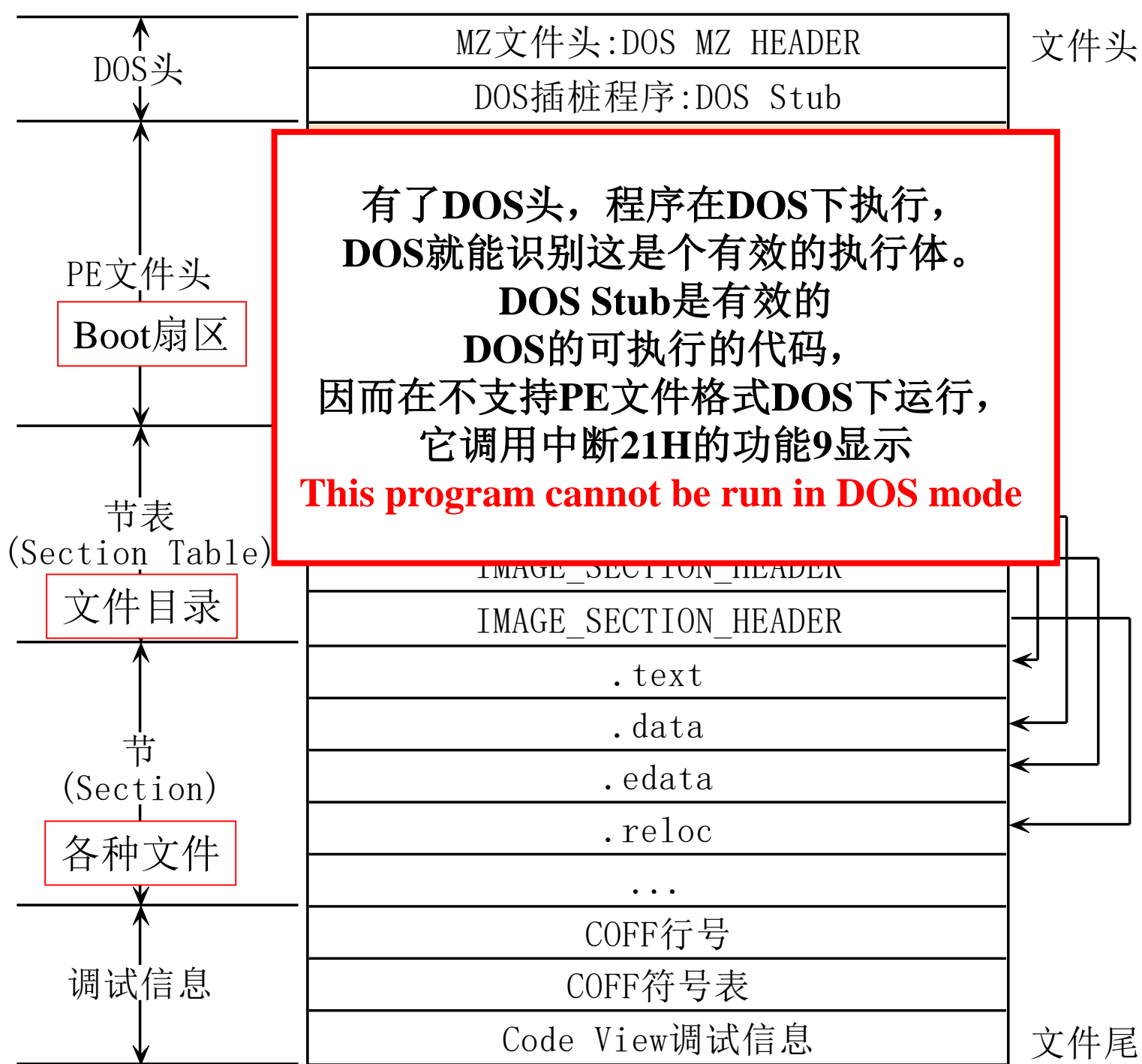
EXE文件的
PE文件

PE文件格式
可看作为逻辑磁盘



EXE文件的 PE文件

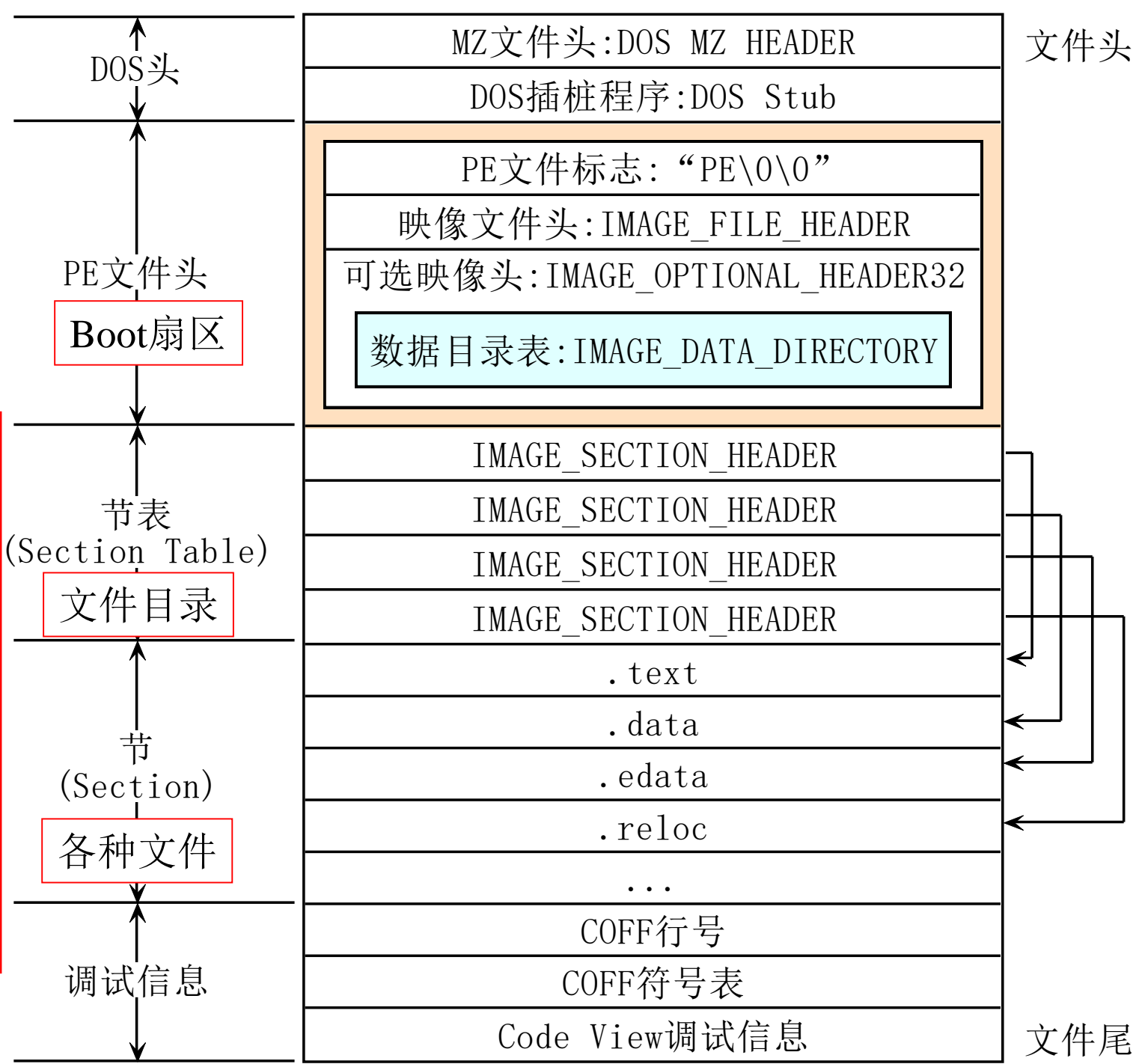
PE文件格式
可看作为逻辑磁盘



EXE文件的
PE文件

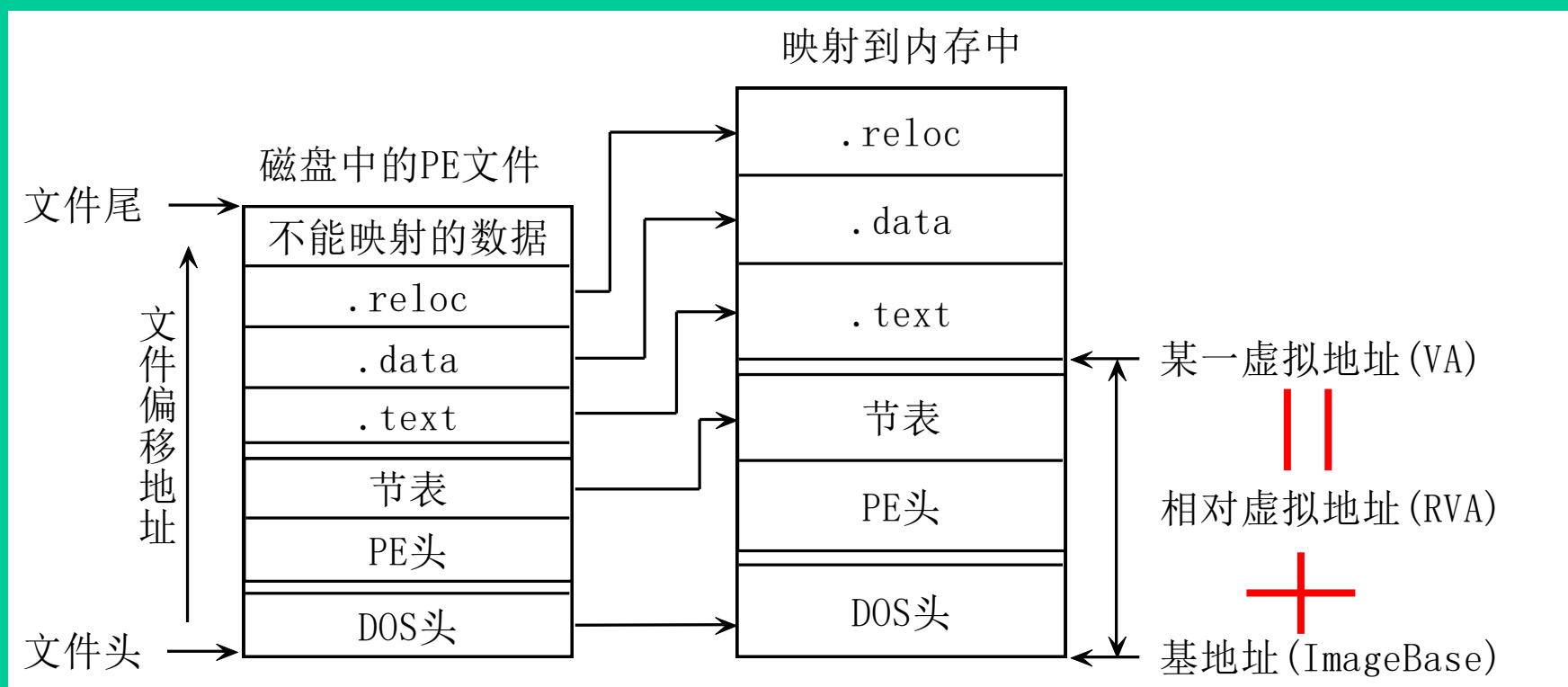
PE文件格式
可看作为逻辑磁盘

每种文件有不同属性，如只读、系统、隐藏、文档等。节的划分是基于各种数据的共同属性，而不是逻辑概念。PE文件中的数据/代码拥有相同的属性，就能被列入同一节。因而节名仅仅是个名称而已，为了识别。真正理解节，要靠节的属性设置

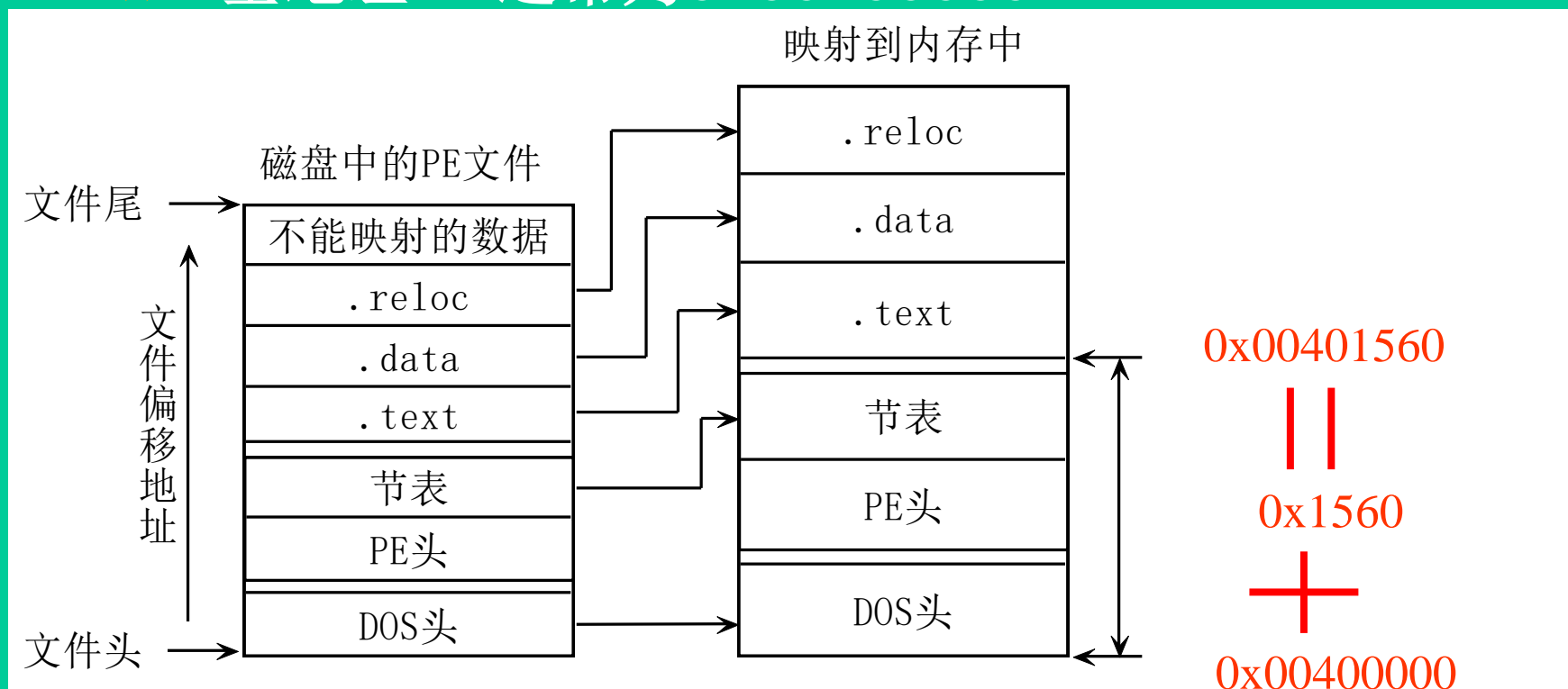


◆ 相对虚拟地址 (Relative Virtual Address, RVA)

- ▶▶ 相对虚拟地址是一个相对于PE文件映射到内存的基地址的偏移量



- ◆ 相对虚拟地址 (Relative Virtual Addresses, RVA)
 - ▶▶ 相对虚拟地址是一个相对于PE文件映射到内存的基地址的偏移量
 - ▶▶ 基地址VA通常为0x00400000



EXE文件的格式

PE文件格式 MS-DOS头 (64字节) USHORT(双字节无符号数)

```
typedef struct _IMAGE_DOS_HEADER { // DOS的.EXE头部
    USHORT e_magic;                // 魔术数字
    USHORT e_cblp;                  // 文件最后页的字节数
    USHORT e_cp;                    // 文件页数
    USHORT e_crlc;                  // 重定向元素个数
    USHORT e_cparhdr;               // 头部尺寸, 以段落为单位
    USHORT e_minalloc;              // 所需的最小附加段
    USHORT e_maxalloc;              // 所需的最大附加段
    USHORT e_ss;                    // 初始的SS值(相对偏移量)
    USHORT e_sp;                    // 初始的SP值
    USHORT e_csum;                  // 校验和
    USHORT e_ip;                    // 初始的IP值
    USHORT e_cs;                    // 初始的CS值(相对偏移量)
    USHORT e_lfarlc;                // 重分配表文件地址
    USHORT e_ovno;                  // 覆盖号
    USHORT e_res[4];                // 保留字
    USHORT e_oemid;                 // OEM标识符(相对e_oeminfo)
    USHORT e_oeminfo;               // OEM信息
    USHORT e_res2[10];              // 保留字
    LONG e_lfanew;                  // 新EXE头部的文件地址
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

‘MZ’

**PE头位置
PE装载器
跳过DOS Stub
定位到PE文件头**

PE文件格式

◆ DOS头与DOS插桩程序

- ▶▶ PE结构中紧随MZ文件头之后的DOS插桩程序(DOS Stub)
- ▶▶ 可以通过IMAGE_DOS_HEADER结构来识别一个合法的DOS头
- ▶▶ 可以通过该结构的e_lfanew(偏移60, 32bits)成员来找到PE开始的标志0x00004550(“PE\0\0”)
- ▶▶ 病毒通过“MZ”、“PE”这两个标志,初步判断当前程序是否是目标文件——PE文件。如果要精确校验指定文件是否为一有效PE文件,则可以检验PE文件格式里的各个数据结构,或者仅校验一些关键数据结构。大多数情况下,没有必要校验文件里的每一个数据结构,只要一些关键数据结构有效,就可以认为是有效的PE文件

PE文件格式

◆ PE文件头

- ▶▶ 紧接着DOS Stub的是PE header
- ▶▶ PE header是IMAGE_NT_HEADERS的简称，即NT映像头(PE文件头)，存放PE整个文件信息分布的重要字段，包含了许多PE装载器用到的重要域。
- ▶▶ 在支持PE文件结构的操作系统中执行时,PE装载器将从DOS MZ header中找到PE header的起始偏移量，从而跳过DOS Stub直接定位到真正的文件头PE header

PE文件格式

◆ PE文件头的结构

```
IMAGE_NT_HEADERS STRUCT  
Signature dd ?  
FileHeader IMAGE_FILE_HEADER <>  
OptionalHeader IMAGE_OPTIONAL_HEADER32<>  
IMAGE_NT_HEADERS ENDS
```

▶▶ 字符串“PE\0\0”(Signature)(4H字节)

- 首先检验文件头部第一个字的值是否等于 **IMAGE_DOS_SIGNATURE**，是则 **DOS MZ header** 有效。
- 一旦证明文件的 **DOS header** 有效后，就可用 **e_lfanew** 来定位 **PE header** 了。
- 比较 **PE header** 的第一个字的值是否等于 **IMAGE_NT_IMAGE_NT_SIGNATURE**。如果前后两个值都匹配，那我们就认为该文件是一个有效的 **PE** 文件。

检验PE文件的
有效性

EXE文件的格式

PE文件头的结构

关于PE文件物理分布的基本信息

-映像文件头-
NT映像头的
主要部分，
包含有PE文
件的基本信息

```
IMAGE_NT_HEADERS STRUCT  
Signature dd ?  
FileHeader IMAGE_FILE_HEADER <>  
OptionalHeader IMAGE_OPTIONAL_HEADER32<>  
IMAGE_NT_HEADERS ENDS
```

PE文件逻辑分布的信息

每个节表28H字节

病毒感兴趣
的地方，
添加一个新节

```
typedef struct _IMAGE_FILE_HEADER {  
    WORD    Machine;                // 0x04, 该程序要执行的环境及平台  
    WORD    NumberOfSections;       // 0x06, 文件中节的个数  
    DWORD   TimeDateStamp;          // 0x08, 文件建立的时间  
    DWORD   PointerToSymbolTable;   // 0x0c, COFF符号表的偏移  
    DWORD   NumberOfSymbols;        // 0x10, 符号数目  
    WORD    SizeOfOptionalHeader;   // 0x14, 可选头的长度  
    WORD    Characteristics;        // 0x16, 标志集合  
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

关于文件信息的标记，比如文件是 exe 还是 dll

EXE文件的格式

PE文件格式

PE文件逻辑分布的信息

▶▶ 紧跟映像文件头后面的是可选映像头-是必须的！

```
typedef struct _IMAGE_OPTIONAL_HEADER {  
    // 标准域:  
    //  
    WORD    Magic;                // 0x18, 一般是0x010B  
    BYTE    MajorLinkerVersion;   // 0x1a, 链接器的主/次版本号,  
    BYTE    MinorLinkerVersion;   // 0x1b, 这两个值都不可靠  
    DWORD    SizeOfCode;           // 0x1c, 可执行代码的长度  
    DWORD    SizeOfInitializedData; // 0x20, 初始化数据的长度(数据节)  
    DWORD    SizeOfUninitializedData; // 0x24, 未初始化数据的长度(bss节)  
    DWORD    AddressOfEntryPoint; // 0x28, 代码的入口RVA地址, 程序从这开始执行  
    DWORD    BaseOfCode;           // 0x2c, 可执行代码起始位置, 意义不大  
    DWORD    BaseOfData;           // 0x30, 初始化数据起始位置, 意义不大  
    // NT 附加域:  
    //  
    DWORD    ImageBase;            // 0x34, 载入程序首选的VA地址  
    DWORD    SectionAlignment;     // 0x38, 加载后节在内存中的对齐方式-节的大小  
    DWORD    FileAlignment;        // 0x3c, 节在文件中的对齐方式-节的大小  
};
```

(待续)

运行PE文件的第一条指令的RVA。假设进程从虚址VA 401000H开始执行, 那么该值为多少?

病毒感兴趣! --指向病毒体代码

EXE文件的格式

PE文件格式

PE文件逻辑分布的信息

▶▶ 紧跟映像文件头后面的是可选映像头-是必须的！

```
typedef struct _IMAGE_OPTIONAL_HEADER {  
    // 标准域:  
    //  
    WORD    Magic;                // 0x18, 一般是0x010B  
    BYTE    MajorLinkerVersion;   // 0x1a, 链接器的主/次版本号,  
    BYTE    MinorLinkerVersion;   // 0x1b, 这两个值都不可靠  
    DWORD    SizeOfCode;           // 0x1c, 可执行代码的长度  
    DWORD    SizeOfInitializedData; // 0x20, 初始化数据的长度(数据节)  
    DWORD    SizeOfUninitializedData; // 0x24, 未初始化数据的长度(bss节)  
    DWORD    AddressOfEntryPoint; // 0x28, 代码的入口RVA地址, 程序从这开始执行  
    DWORD    BaseOfCode;           // 0x2c, 可执行代码起始位置, 意义不大  
    DWORD    BaseOfData;           // 0x30, 初始化数据起始位置, 意义不大  
    // NT 附加域:  
    //  
    DWORD    ImageBase;            // 0x34, 载入程序首选的VA地址  
    DWORD    SectionAlignment;     // 0x38, 加载后节在内存中的对齐方式-节的大小  
    DWORD    FileAlignment;        // 0x3c, 节在文件中的对齐方式-节的大小  
};
```

(待续)

首选不是必须，如果该值为400000H，但是被其他模块占用，PE装载器会选择其他空闲地址。

EXE文件的格式

PE文件格式

PE文件逻辑分布的信息

▶▶ 紧跟映像文件头后面的是可选映像头-是必须的！

```
typedef struct _IMAGE_OPTIONAL_HEADER {
```

```
    // 标准域:
```

```
    //
```

```
    WORD    Magic;
```

```
    BYTE    MajorLinkerVersion;
```

```
    BYTE    MinorLinkerVersion;
```

```
    DWORD    SizeOfCode;
```

```
    DWORD    SizeOfInitializedData;
```

```
    DWORD    SizeOfUninitializedData;
```

```
    DWORD    AddressOfEntryPoint;
```

```
    DWORD    BaseOfCode;
```

```
    DWORD    BaseOfData;
```

```
    // NT 附加域:
```

```
    //
```

```
    DWORD    ImageBase;
```

```
    DWORD    SectionAlignment;
```

```
    DWORD    FileAlignment;
```

```
    // 0x10, 节在文件中的对齐方式-节的大小
```

内存中节对齐的粒度。该值为1000H（4096）。

那么每节的起始地址必须是4096的倍数。

若第一节从401000H开始，大小为10字节，那么下一节从什么地方开始？

文件中节对齐的粒度。若该值为200H（512）。那么每节的起始地址必须是512倍数。

```
    // 0x34, 载入程序首选的VA地址
```

```
    // 0x38, 加载后节在内存中的对齐方式-节的大小
```

```
    // 0x3c, 节在文件中的对齐方式-节的大小
```

(待续)

EXE文件的格式

PE文件格式

(续前)

```
WORD    MajorOperatingSystemVersion; // 0x3e, 操作系统主/次版本,
WORD    MinorOperatingSystemVersion; // 0x40, Loader并没有用这两个值
WORD    MajorImageVersion;           // 0x42, 可执行文件主/次版本
WORD    MinorImageVersion;           // 0x44
WORD    MajorSubsystemVersion;       // 0x46, 子系统版本号
WORD    MinorSubsystemVersion;       // 0x48
DWORD    Win32VersionValue;          // 0x4c, Win32版本, 一般是0
DWORD    SizeOfImage;               // 0x50, 程序调入后占用内存大小(字节)
DWORD    SizeOfHeaders;            // 0x54, 文件头的长度之和
DWORD    Checksum;                 // 0x58, 校验和
WORD     Subsystem;               // 0x5c, 可执行文件的子系统GUI或CUI
WORD     DllCharacteristics;         // 0x5e, 何时DllMain被调用, 一般为0
DWORD    SizeOfStackReserve;         // 0x60, 初始化线程时保留的堆栈大小
DWORD    SizeOfStackCommit;         // 0x64, 初始化线程时提交的堆栈大小
DWORD    SizeOfHeapReserve;          // 0x68, 进程初始化时保留的堆大小
DWORD    SizeOfHeapCommit;          // 0x6c, 进程初始化时提交的堆大小
DWORD    LoaderFlags;               // 0x70, 装载标志, 与调试相关
DWORD    NumberOfRvaAndSizes;        // 0x74, 数据目录的项数, 一般是16
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```


EXE文件的格式

DataDirectory: 数据目录表

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

- ◆ 是一个 **IMAGE_DATA_DIRECTORY** 结构数组，有16个这样的元素。
- ◆ 数据目录表-每个结构给出一个重要数据结构的起始RVA和大小信息。
- ◆ 节表可以看作是PE文件各节的根目录的话，也可以认为 **data directory** 是存储在这些节里的逻辑元素的根目录。

目录表查看器

	RVA	大小
输出表:	00000000	00000000
输入表:	000073A4	000000B4
资源:	0002D000	00004348
例外:	00000000	00000000
安全性:	00089BB8	00001558
基址重定位:	00000000	00000000
调试:	00000000	00000000
版权:	00000000	00000000
全局指针:	00000000	00000000
TLS 表:	00000000	00000000
载入配置:	00000000	00000000
输入表范围:	00000000	00000000
输入地址表:	00007000	0000028C

EXE文件的格式

重要数据结构

◆ 什么重要数据结构？

如：导入目录-导入函数(引入函数 **import**)

一个引入函数是被某模块调用但又不在调用模块中的函数，位于一个或者更多的**DLL**里，因而要保留一些函数信息，包括函数名及其驻留的**DLL**名。

◆ 怎么样获得**PE**文件中重要数据结构？

目录表查看器

	RVA	大小
输出表: Export Table:		00000000
输入表: Import Table:		000000B4
资源: Recourse:		00004348
例外: Exception:		00000000
安全性: Security:		00001558
基址重定位: Baserelease:		00000000
调试: Debug:		00000000
版权: Copyright:		00000000
全局指针: Globalptr:		00000000
TLS 表: Tls Table:		00000000
载入配置: Load Config:		00000000
输入表范围: Bound Import:		00000000
输入地址表: Import Address Table:		0000028C

EXE文件的格式

怎么样获得PE文件中重要数据结构？

从 DOS header 定位到 PE header

从 optional header 读取 data directory 的地址。

IMAGE_DATA_DIRECTORY 结构尺寸乘上找寻结构的索引号。例如，欲获取 import symbols 的位置信息，必须用 **IMAGE_DATA_DIRECTORY** 结构尺寸(8 bytes)乘上1（import symbols 在 data directory 中的索引号）。

将上面的结果加上 data directory 地址，就得到包含所查询数据结构信息的 **IMAGE_DATA_DIRECTORY** 结构项

目录表查看器

	RVA	大小
输出表:	Export Table:	00000000
输入表:	Import Table:	000000B4
资源:	Resource:	00043448
例外:	Exception:	00000000
安全性:	Security:	00015558
基址重定位:	BaseReloc:	00000000
调试:	Debug:	00000000
版权:	Copyright:	00000000
全局指针:	Globalptr:	00000000
TLS 表:	Tls Table:	00000000
载入配置:	Load Config:	00000000
输入表范围:	Bound Import:	00000000
输入地址表:	Import Address Table:	000028C

EXE文件的格式

PE文件格式

- ◆ 节表是紧挨着NT映像头的一结构数组，其成员的数目由映像文件头中NumberOfSections决定

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    UCHAR Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        ULONG PhysicalAddress;
        ULONG VirtualSize;
    } Misc;
    ULONG VirtualAddress;
    ULONG SizeOfRawData;
    ULONG PointerToRawData;
    ULONG PointerToRelocations;

    ULONG PointerToLinenumbers;
    USHORT NumberOfRelocations;
    USHORT NumberOfLinenumbers;
    ULONG Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

本节的实际字节数 如388H字节

本节的相对虚拟地址 如为1000H，
而PE文件装载地址400000H，？

// OBJ文件中表示本节物理地址

// EXE文件中表示节的实际字节数

// 本节的RVA

// 本节经过文件对齐后的尺寸

// 本节原始数据在文件中的位置

// OBJ文件中表示本节重定位信

// 息的偏移，EXE文件中无意义

// 行号偏移

// 本节需重定位的数目

// 本节在行号表中的行号数目

// 节属性

EXE文件的格式

PE文件格式

- ◆ 节表是紧挨着NT映像头的一结构数组，其成员的数目由映像文件头中NumberOfSections决定

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    UCHAR Name[IMAGE_SIZEOF_SHORT_NAME]; // 节名
    union {
        ULONG PhysicalAddress;
        ULONG VirtualSize;
    } Misc;
    ULONG VirtualAddress; // 本节的RVA
    ULONG SizeOfRawData; // 本节经过文件对齐后的尺寸
    ULONG PointerToRawData; // 本节原始数据在文件中的位置
    ULONG PointerToRelocations; // OBJ文件中表示本节重定位信息的偏移，EXE文件中无意义
    ULONG PointerToLinenumbers; // 行号偏移
    USHORT NumberOfRelocations; // 本节需重定位的数目
    USHORT NumberOfLinenumbers; // 本节在行号表中的行号数目
    ULONG Characteristics; // 节属性
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

PE装载器通过本域找到节的位置

EXE文件的格式

PE文件格式

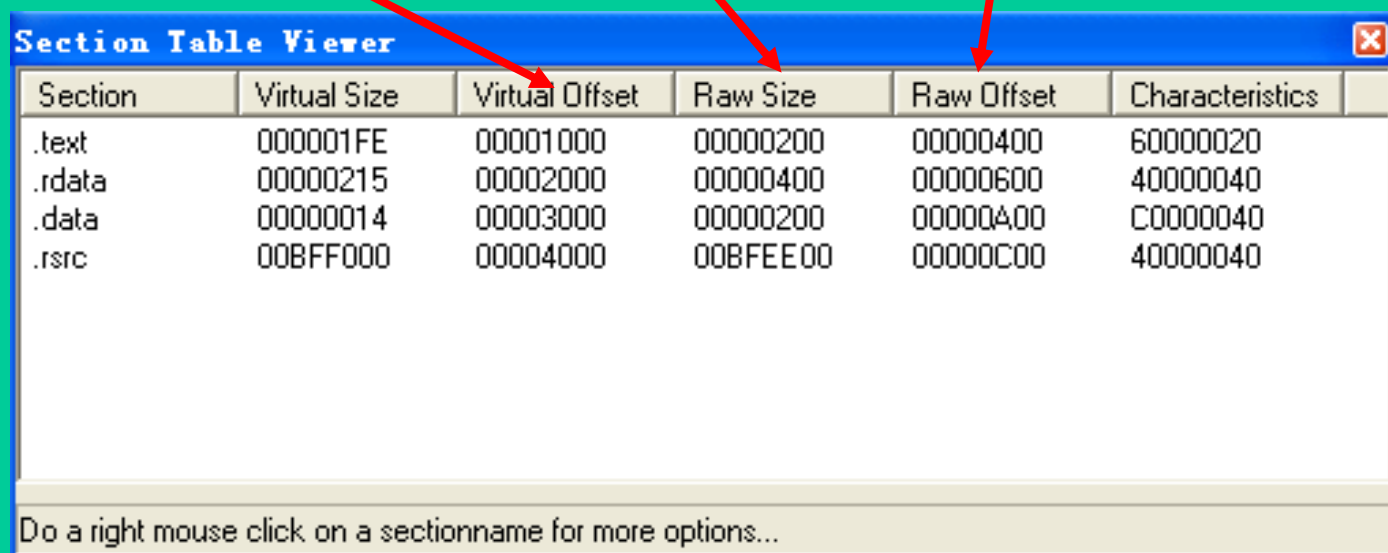
NumberOfSections知道有几个节

SizeOfHeaders知道节表在什么地方开始

遍历节表，**PointerToRawData**知道节在文件中偏移量

SizeOfRawData来决定映射内存的字节数

VirtualAddress加上**ImageBase**知道节的起始虚拟地址



Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.text	000001FE	00001000	00000200	00000400	60000020
.rdata	00000215	00002000	00000400	00000600	40000040
.data	00000014	00003000	00000200	00000A00	C0000040
.rsrc	00BFF000	00004000	00BFEE00	00000C00	40000040

Do a right mouse click on a sectionname for more options...

PE文件格式

◆ 节

- ▶▶ PE文件的真正内容划分成块，称之为**Section(节)**，紧跟在节表之后
- ▶▶ 每个节是一块拥有共同属性的数据，比如代码/数据、读/写等
- ▶▶ 可以把PE文件想象成一逻辑磁盘，**PE header**是磁盘的**Boot**扇区，节表就是根目录，而**Section**就是各种文件，每种文件自然就有不同属性如只读、系统、隐藏、文档等等
- ▶▶ 节的划分是基于各组数据的共同属性而不是逻辑概念——如果PE文件中的数据/代码拥有相同属性，它们就能被归入同一节中
- ▶▶ 节名称仅仅是个区别不同节的符号而已，类似“**data**”、“**code**”的命名只为了便于识别，惟有节的属性设置决定了节的特性和功能

感谢大家！

