PE文件 — [DOS头](#DOS头) — e\_magic，DOS签名，必须为MZ

PE头 — e\_lfanew，NT头偏移

— ……

— [DOS存根](#DOS存根)

— NT头 — [签名结构体](#NT头：签名结构体（Signature）)，PE文件格式核心标识“PE\0\0”

— [文件头](#NT头：文件头（FileHeader）) — Machine编码

— NumberOfSections，文件节区数量

— SizeOfOptionalHeader，特定架构下的可选头长度

— Characteristics，标识文件属性

— TimeDateStamp，纪录编译创建时间

— ……

— [可选头](#NT头：可选头（OptionalHeader）) — magic

— AdressOfEntryPoint，指出程序最先执行的代码起始地址

— ImageBase，指出文件优先装载地址

— SectionAlignment，节区内存中的最小单位

— FileAlignment，节区在磁盘中的最小单位

— SizeOfImage，指定PEImage在虚拟内存中所占的空间大小

— SubSystem，区分系统驱动文件和普通可执行文件

— NumberOfRvaAndSize，指定DataDirectory数组个数

— DataDirectory数组

— ……

— [节区头](#节区头) — VirtualSize，内存中节区大小

— VirtualAddress，内存节区起始地址，由可选头SectionAlignment确定

— SizeOfRawData，磁盘中节区所占大小

— PointerToRawData，磁盘节区起始地址，由FileAlignment确定

— Characteristics，节区属性

— ……

（code、data、resource存在不同节区，节区头就由上面的结构体组成数组，每个结构体对应一个区）

DOS头

共64字节。

typedef struct \_IMAGE\_DOS\_HEADER {

WORD e\_magic; *// DOS签名 "MZ" (0x5A4D)，重要*

WORD e\_cblp; *// 字节数（最后页）*

WORD e\_cp; *// 页数*

WORD e\_crlc; *// 重定位项数*

WORD e\_cparhdr; *// 头部段数*

WORD e\_minalloc; *// 最小内存分配*

WORD e\_maxalloc; *// 最大内存分配*

WORD e\_ss; *// 初始SS值*

WORD e\_sp; *// 初始SP值*

WORD e\_csum; *// 校验和*

WORD e\_ip; *// 初始IP值*

WORD e\_cs; *// 初始CS值*

WORD e\_lfarlc; *// 重定位表偏移*

WORD e\_ovno; *// 覆盖号*

WORD e\_res[4]; *// 保留字段*

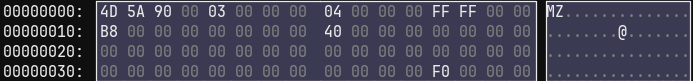
WORD e\_oemid; *// OEM标识符*

WORD e\_oeminfo; *// OEM信息*

WORD e\_res2[10]; *// 保留字段*

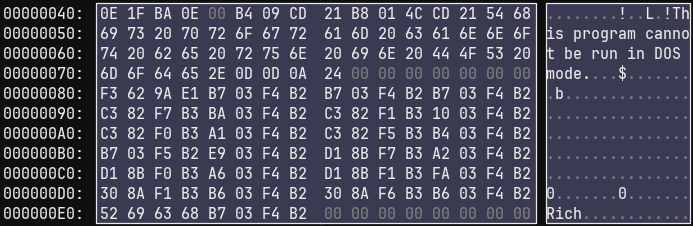
LONG e\_lfanew; *// NT头偏移（PE文件起始位置），重要*

} IMAGE\_DOS\_HEADER, \*PIMAGE\_DOS\_HEADER;



DOS存根

在DOS系统下提示一句话，包含数据和代码，可以修改，但是修改不可覆盖DOS头和NT头，长度不能随意修改，如果缩短或增长需要修改地址，非常麻烦。



NT头

没什么说的。

typedef struct \_IMAGE\_NT\_HEADERS {

DWORD Signature; *// PE签名 "PE\0\0" (0x00004550)*

IMAGE\_FILE\_HEADER FileHeader; *// 文件头*

IMAGE\_OPTIONAL\_HEADER OptionalHeader; *// 可选头（32/64位）*

} IMAGE\_NT\_HEADERS32, \*PIMAGE\_NT\_HEADERS32;

*// 64位版本（IMAGE\_OPTIONAL\_HEADER64）*

typedef struct \_IMAGE\_NT\_HEADERS64 {

DWORD Signature;

IMAGE\_FILE\_HEADER FileHeader;

IMAGE\_OPTIONAL\_HEADER64 OptionalHeader;

} IMAGE\_NT\_HEADERS64, \*PIMAGE\_NT\_HEADERS64;

NT头：签名结构体（Signature）

PE文件核心标识，内容“PE\0\0”。



NT头：文件头（FileHeader）

固定20字节。

typedef struct \_IMAGE\_FILE\_HEADER {

WORD Machine; *// 目标CPU架构（如0x014C=Intel 386）*

WORD NumberOfSections; *// 节区数量*

DWORD TimeDateStamp; *// 编译时间戳*

DWORD PointerToSymbolTable; *// 符号表偏移（调试用）*

DWORD NumberOfSymbols; *// 符号数量*

WORD SizeOfOptionalHeader; *// 可选头大小*

WORD Characteristics; *// 文件属性（如可执行/DLL）*

} IMAGE\_FILE\_HEADER, \*PIMAGE\_FILE\_HEADER;

其中Characteristics按bit位定义，每一位含义如下：

|  |  |  |  |
| --- | --- | --- | --- |
| Bit0 | IMAGE\_FILE\_RELOCS\_STRIPPED | 0x0001 | 重定位信息已移除（通常是 EXE） |
| Bit 1 | IMAGE\_FILE\_EXECUTABLE\_IMAGE | 0x0002 | 文件是可执行的 |
| Bit 2 | IMAGE\_FILE\_LINE\_NUMS\_STRIPPED | 0x0004 | 行号信息已移除（已废弃） |
| Bit 3 | IMAGE\_FILE\_LOCAL\_SYMS\_STRIPPED | 0x0008 | 符号表已移除（已废弃） |
| Bit 4 | IMAGE\_FILE\_AGGRESIVE\_WS\_TRIM | 0x0010 | 优化工作集（已废弃） |
| Bit 5 | IMAGE\_FILE\_LARGE\_ADDRESS\_AWARE | 0x0020 | 支持 >2GB 地址空间 |
| Bit 7 | IMAGE\_FILE\_BYTES\_REVERSED\_LO | 0x0080 | 小端字节序（已废弃） |
| Bit 8 | IMAGE\_FILE\_32BIT\_MACHINE | 0x0100 | 32 位架构（x86） |
| Bit 9 | IMAGE\_FILE\_DEBUG\_STRIPPED | 0x0200 | 调试信息已移除 |
| Bit 10 | IMAGE\_FILE\_REMOVABLE\_RUN\_FROM\_SWAP | 0x0400 | 可从交换设备运行（已废弃） |
| Bit 11 | IMAGE\_FILE\_NET\_RUN\_FROM\_SWAP | 0x0800 | 可从网络运行（已废弃） |
| Bit 12 | IMAGE\_FILE\_SYSTEM | 0x1000 | 系统文件（如内核驱动） |
| Bit 13 | IMAGE\_FILE\_DLL | 0x2000 | 这是一个 DLL 文件 |
| Bit 14 | IMAGE\_FILE\_UP\_SYSTEM\_ONLY | 0x4000 | 仅单处理器运行（已废弃） |
| Bit 15 | IMAGE\_FILE\_BYTES\_REVERSED\_HI | 0x8000 | 大端字节序（已废弃） |



NT头：可选头（OptionalHeader）

长度由文件头里的SizeOfOptionalHeader确定，32位PE文件通常为0xE0（224字节），64位PE文件通常为0xF0（240字节）。

typedef struct \_IMAGE\_OPTIONAL\_HEADER32 {

*// 标准字段（所有PE文件）*

WORD Magic; *// 标识：0x10B=32位，0x20B=64位*

BYTE MajorLinkerVersion; *// 链接器主版本号*

BYTE MinorLinkerVersion; *// 链接器次版本号*

DWORD SizeOfCode; *// 所有代码段的总大小*

DWORD SizeOfInitializedData; *// 已初始化数据的总大小*

DWORD SizeOfUninitializedData; *// 未初始化数据（BSS）的总大小*

DWORD AddressOfEntryPoint; *// 入口点RVA（相对于ImageBase）*

DWORD BaseOfCode; *// 代码段的起始RVA*

DWORD BaseOfData; *// 数据段的起始RVA（仅32位存在）*

*// NT扩展字段（Windows专用）*

DWORD ImageBase; *// 进程内存中的优先加载地址*

DWORD SectionAlignment; *// 内存中的节区对齐粒度（通常0x1000）*

DWORD FileAlignment; *// 文件中的节区对齐粒度（通常0x200）*

WORD MajorOperatingSystemVersion; *// 要求的最低OS主版本*

WORD MinorOperatingSystemVersion; *// 要求的最低OS次版本*

WORD MajorImageVersion; *// 映像主版本号（用户定义）*

WORD MinorImageVersion; *// 映像次版本号（用户定义）*

WORD MajorSubsystemVersion; *// 子系统主版本（通常4=Win95）*

WORD MinorSubsystemVersion; *// 子系统次版本*

DWORD Win32VersionValue; *// 保留（必须为0）*

DWORD SizeOfImage; *// 映像在内存中的总大小*

DWORD SizeOfHeaders; *// 所有头部的总大小（对齐后）*

DWORD CheckSum; *// 校验和（驱动/DLL常用）*

WORD Subsystem; *// 子系统类型（1=Native，2=GUI，3=CUI）*

WORD DllCharacteristics; *// DLL属性（如ASLR/DEP）*

DWORD SizeOfStackReserve; *// 初始保留的栈大小*

DWORD SizeOfStackCommit; *// 初始提交的栈大小*

DWORD SizeOfHeapReserve; *// 初始保留的堆大小*

DWORD SizeOfHeapCommit; *// 初始提交的堆大小*

DWORD LoaderFlags; *// 保留（已废弃）*

DWORD NumberOfRvaAndSizes; *// 数据目录项数（通常16）*

IMAGE\_DATA\_DIRECTORY DataDirectory[IMAGE\_NUMBEROF\_DIRECTORY\_ENTRIES]; *// 数据目录表*

} IMAGE\_OPTIONAL\_HEADER32, \*PIMAGE\_OPTIONAL\_HEADER32;

其中的IMAGE\_DATA\_DIRECTORY结构如下：

typedef struct \_IMAGE\_DATA\_DIRECTORY {

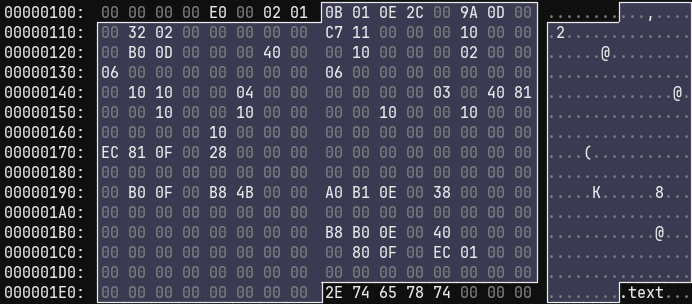
DWORD VirtualAddress; *// 数据的 RVA（相对虚拟地址）*

DWORD Size; *// 数据的大小（字节数）*

} IMAGE\_DATA\_DIRECTORY, \*PIMAGE\_DATA\_DIRECTORY;

DataDirectory是数组，储存RVA，下面是数组每一项的含义：

1. = EXPORT Directory 导出表（DLL 导出的函数列表），重要
2. = IMPORT Directory 导入表（依赖的外部 DLL 函数），重要
3. = RESOURCE Directory 资源表
4. = EXCEPTION Directory 异常处理表
5. = SECURITY Directory 数字签名
6. = BASERELOC Directory 重定位表
7. = DEBUG Directory 调试信息
8. = COPYRIGHT Directory 架构特定数据
9. = GLOBALPTR Directory 全局指针寄存器
10. = TLS Directory TLS表，重要
11. = LOAD\_CONFIG Directory 加载配置表
12. = BOUND\_IMPORT Directory 绑定导入表
13. = IAT Directory 导入地址表（IAT）
14. = DELAY\_IMPORT Directory 延迟加载导入表
15. = COM\_DESCRIPTOR Directory .NET元数据
16. = Reserved Directory 保留，未使用



typedef struct \_IMAGE\_OPTIONAL\_HEADER64 {

*// 标准字段（与32位类似）*

WORD Magic; *// 标识：0x20B=64位*

BYTE MajorLinkerVersion;

BYTE MinorLinkerVersion;

DWORD SizeOfCode;

DWORD SizeOfInitializedData;

DWORD SizeOfUninitializedData;

DWORD AddressOfEntryPoint; *// 入口点RVA*

DWORD BaseOfCode; *// 代码段起始RVA*

*// BaseOfData 字段在64位中不存在！*

*// NT扩展字段*

ULONGLONG ImageBase; *// 64位优先加载地址*

DWORD SectionAlignment;

DWORD FileAlignment;

WORD MajorOperatingSystemVersion;

WORD MinorOperatingSystemVersion;

WORD MajorImageVersion;

WORD MinorImageVersion;

WORD MajorSubsystemVersion;

WORD MinorSubsystemVersion;

DWORD Win32VersionValue;

DWORD SizeOfImage;

DWORD SizeOfHeaders;

DWORD CheckSum;

WORD Subsystem;

WORD DllCharacteristics;

ULONGLONG SizeOfStackReserve; *// 64位栈/堆大小*

ULONGLONG SizeOfStackCommit;

ULONGLONG SizeOfHeapReserve;

ULONGLONG SizeOfHeapCommit;

DWORD LoaderFlags;

DWORD NumberOfRvaAndSizes; *// 数据目录项数（通常16）*

IMAGE\_DATA\_DIRECTORY DataDirectory[IMAGE\_NUMBEROF\_DIRECTORY\_ENTRIES]; *// 数据目录表*

} IMAGE\_OPTIONAL\_HEADER64, \*PIMAGE\_OPTIONAL\_HEADER64;

节区头

每个节区头固定40字节。下面的实际例子里有7个节区头，对应的从上面NT头中文件头处第二个字可以看到0x0007（小端序，读的时候倒一下），就是说这里有7个节区头。

typedef struct \_IMAGE\_SECTION\_HEADER {

BYTE Name[8]; *// 节区名（如".text"），可以填入任何值，只是做参考*

union {

DWORD PhysicalAddress;

DWORD VirtualSize; *// 内存中的实际大小，重要*

} Misc;

DWORD VirtualAddress; *// 内存中的RVA，重要*

DWORD SizeOfRawData; *// 文件中的大小，重要*

DWORD PointerToRawData; *// 文件中的偏移，重要*

DWORD PointerToRelocations; *// 重定位表偏移*

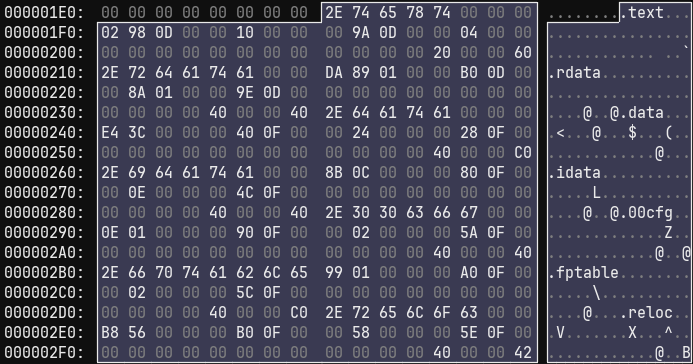
DWORD PointerToLinenumbers; *// 行号表偏移*

WORD NumberOfRelocations; *// 重定位项数*

WORD NumberOfLinenumbers; *// 行号项数*

DWORD Characteristics; *// 节区属性（可读/可写/可执行等），重要*

} IMAGE\_SECTION\_HEADER, \*PIMAGE\_SECTION\_HEADER;



计算文件偏移：

RAW（文件中的物理偏移） = RVA（内存中的相对地址） - VirtualAddress + PointorToRawData

在这个文件截图中以text节区头为例，RVA = 0x1000，VirtualAddress = 0x1000，PointorToRawData = 0xD9A00，算得RAW = 0xD9A00。这里VirtualAddress与RVA相同，因为VirtualAddress字段直接存储的是RVA值。但实际上他们的关系是所有VirtualAddress都是 RVA，但并非所有 RVA都是VirtualAddress，VirtualAddress是节区头的字段，表示该节区在内存中的起始 RVA，目标 RVA是你要转换的具体数据（如函数）的 RVA。换句话说，只是节区头里的VirtualAddress字段名表示的是RVA，但是其他PE结构中的内存偏移会明确用RVA表示，VirtualAddress和RVA概念还是不同的。

程序运行时CPU和操作系统访问的都是VA（虚拟地址）。RAW、RVA、VA三者转换关系为RAW（磁盘中的物理偏移，与内存无关）-> RVA（加载到内存，是PE文件内部的相对偏移，用于静态分析） -> VA（运行时访问）

磁盘文件：文件地址 = 文件起始位置（0x00) + 偏移量（RAW）

内存映射：VA = ImageBase + RVA，这里ImageBase是NT头IMAGE\_OPTIONAL\_HEADER里的。

实际运行：VA = 实际加载基质（随机化）+ RVA