一个操作系统的可执行文件很大程度上可以被视为此操作系统的一面镜子。虽然学习可执行文件对于程序员来说不是非常必要的一件事情，但是通过挖掘此类文件的内部结构，我们可以对编程有一个更好的了解。接下来这篇论文中，我将详细讨论可执行文件。

**PE文件概述**

PE文件的全称是Portable Executable，也就是可移植的可执行文件，我们平时常常看到的诸如EXE，DLL，COM，对象代码 以及一些其他被用在32位和64位Windows操作系统中的文件都是PE文件，换句话说，PE文件是微软Windows操作系统上的程序文件，它其实是一种数据结构， 里面包含了Windows 操作系统加载器管理可执行代码所需要的必要信息。PE文件使用的是一个平面地址空间，所有代码和数据都被合并在一起，组成一个很大的结构。

PE文件是Unix COFF文件的变更版本，它继承了很多COFF文件的特点，所以想了解PE文件就得稍微了解COFF文件。COFF文件-通用对象文件格式（Common Object File Format），是一种很流行的对象文件格式（主意：这里不说它是“目标”文件是为了和编译器产生的文件相区别，因为这种格式不只用于目标文件，库文件、可执行文件也经常是这种格式）。COFF文件一共有8中数据结构，自上而下分别为文件头（File Header）、可选头（Optional Header）、段落头（Section Header）、段落数据（Section Data）、重定位表（Relocation Directives）、行号表（Line Numbers）、符号表（Symbol Table）、字符串表（String Table）。这些数据结构被PE文件很好地继承，并且同时进行了适当的扩展，在具体了解PE文件的层次结构之前我们需要先了解下PE文件与内存的交互。

**分页机制**

首先我们先来了解下分页机制。当需要将一定内容段存储在内存中时，因为段的长短不定，在分配内存时，可能会发生内存中的空闲区域小于要加载的段，或者空闲区域远远大于要加载的段。在前一种情况下，我们就需要额外的需求空闲区域；在最后一种情况下，即使我们分配成功，也会浪费一大片内存空间。为了解决这一问题，目前很多处理器已经引入了分页机制。分页机制从整体上来说，是用长度固定的页来代替长度不一定的段，借此来解决因为段长度不同而带来的内存空间管理问题。

一旦决定采用页式内存管理，就应当把4GB内存分成大小相同的页。页的最小单位是4KB，也就是4096字节，用十六进制表示的话就是0x1000.因此第一个页的物理地址就是0x00000000，第二个页的物理地址是0x00001000，第三个就是0x00000000……最后一个页就是0xfffff000。这样4GB内存就被划分为1048576（0x100000）个页。

由于32为CPU的寻址能力为4GB，但是有些用户的物理内存达不到这个值。于是操作系统和CPU的内存机制共同作用为用户提供虚拟的内存管理机制，采用的就是分页机制实现。这样用户就会感觉自己好像在使用4GB的内存。

**PE文件与内存**

系统考虑加载一个PE文件到内存中时，因为它在磁盘和内存中基本一样，都是通过二进制流方式存储，所以加载过程就相当于简单地将PE文件”搬运“到内存中。PE文件是通过加载器去加载，加载完后我们将称之为“映像”，一般说来，PE文件在磁盘上和内存上的表现是不完全一样的，被加载到内存后以后PE文件所占用的虚拟空间要比磁盘上占用的空间大一些，这是因为PE文件中存在节表，节表中存在一系列的节，所谓的节其实就相当于一个容器，里面包含了代码和数据等信息，而这些节在磁盘上是连续存储的，但是被加载到内存中之后因为会按照分页机制存储，这样节与节之间可能存在一些“空洞”。因为这个特点，所以在PE结构内部表示某个位置的地址采用了两种方式，针对在磁盘上存储文件中的地址，称之为原始内存地址或物理地址表示距离文件头的偏移；另外一种是针对加载到内存以后映像中的地址，称为相对虚拟地址（RVA），表示相对内存映像头的偏移，关于头的概念（Header）将会在后续讲解PE文件层次结构时进行讨论。

但是CPU某些指令是需要使用绝对地址的，比如取全局变量地址，传递函数的地址，传递函数的地址编译以后的汇编指令中肯定用到绝对地址而不是相对映像头的偏移量，因此PE文件会建议操作系统将其加载到某个内存地址，即基地址，然后编译器可以根据这个基地址求出相对应的全局变量以及函数的地址，最后将这些地址用到对应的指令中，这种表示方式我们称为虚拟地址（VA）。

那么问题来了，既然有VA存在我们为什么还需要RVA呢？因为虽然PE文件为自己指定加载的的基地址，但是Windows中有非常多的可执行文件，每个软件都有自己对应的可执行文件，如果指定的基地址都被其他可执行文件占了怎么办？如果PE文件无法加载到预期的地址，那么系统会帮他重新选择一个合适的基地址将他加载到此处，这时原有的VA就全部失效了，NT头保存了PE文件加载所需的信息，在不知道PE会加载到哪个基地址之前，VA是无效的，所以在PE文件头中大部分是使用RVA来表示地址的，而在代码中是用VA表示全局变量和函数地址的。那又有人要问了，既然加载基址变了以后VA都失效了，那存在于代码中的那些VA怎么办呢？答案是：重定位。系统有自己的办法修正这些值，到后续重定位表的文章中会详细描述。既然有重定位，为什么NT头不能依靠重定位采用VA表示地址呢（十万个为什么）？因为不是所有的PE都有重定位，早期的EXE就是没有重定位的。

我们都知道PE文件可以导出函数让其他的PE文件使用，也可以从其他PE文件导入函数，这些是如何做到的？PE文件通过导出表指明自己导出那些函数，通过导入表指明需要从哪些模块导入哪些函数。

**PE文件层次结构**

**DOS 头**

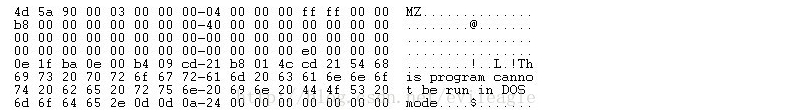
PE文件格式以DOS头这一结构开始，它既是头文件也是一段可执行代码，它存在的主要目的只是为了兼容早期的DOS操作系统，DOS头是由IMAGE\_DOS\_HEADER结构体来定义，以下为其结构定义代码：

|  |
| --- |
| struct IMAGE\_DOS\_HEADER  {  public UInt16 e\_magic; // 魔术数字  public UInt16 e\_cblp; // 文件最后页的字节数  public UInt16 e\_cp; // 文件页数  public UInt16 e\_crlc; // 重定义元素个数  public UInt16 e\_cparhdr; // 头部尺寸，以段落为单位  public UInt16 e\_minalloc; // 所需的最小附加段  public UInt16 e\_maxalloc; // 所需的最大附加段  public UInt16 e\_ss; // 初始的SS值（相对偏移量）  public UInt16 e\_sp; // 初始的SP值  public UInt16 e\_csum; // 校验和  public UInt16 e\_ip; // 初始的IP值  public UInt16 e\_cs; // 初始的CS值（相对偏移量）  public UInt16 e\_lfarlc; // 重分配表文件地址  public UInt16 e\_ovno; // 覆盖号  public UInt16[] e\_res; // 保留字，8字节  public UInt16 e\_oemid; // OEM标识符（相对e\_oeminfo）  public UInt16 e\_oeminfo; // OEM信息  public UInt16[] e\_res2; // 保留字，20字节  public UInt32 e\_lfanew; // 新exe头部的文件地址  } |

此结构体中有两个字段需要特别注意，一个是e\_magic，一个是e\_lfanew字段。e\_magic字段是一个DOS可执行文件的标识符，分别为4Dh和5Ah，它占用两个字节，保存着的字符是”MZ“，并且它在Winnt.h头文件中有一个宏定义，定义内容如下：

#define IMAGE\_DOS\_SIGNATURE 0x5A4D //MZ

正因为这样，有时候也会称呼DOS头为DOS MZ Header。至于e\_lfanew，它是一个双字数据，为PE头的离文件头部的偏移量。Windows加载器可以通过它跳过DOS Stub部分直接找到PE头。DOS头后面还会跟着一个DOS Stub的数据，是链接器链接执行文件的时候加入的部分数据，一般情况下，如果在DOS系统中执行PE文件，会根据这部分内容显示：”This program cannot be run in DOS mode“。这个可以通过修改链接器的设置来修改成自定义的数据。图一显示了真实情况：



**图一**

**NT 头**

根据上面内容我们了解到DOS头有个e\_lfanew字段，通过这个字段我们可以直接找到对应的NT头。NT头中定义了PE文件头以及可选头，对于PE文件来说这两个部分至关重要，这就导致NT头是最有用的头，它的定义如下：

Struct IMAGE\_NT\_HEADERS

{

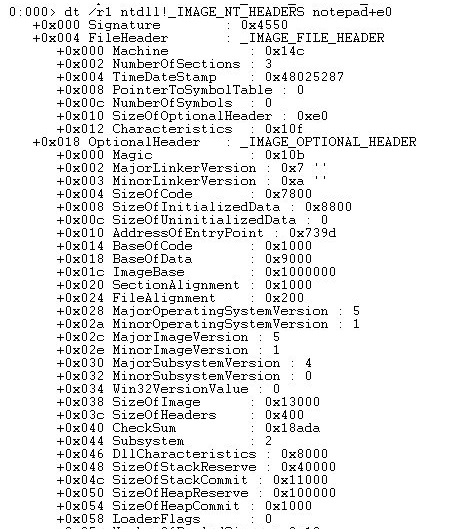
Public UInt32 Signature; // 4 bytes

Public IMAGE\_FILE\_HEADER FileHeader; // 20 bytes

Public IMAGE\_OPTIONAL\_HEADER OptionalHeader; // 224 bytes

}

Signature类似于DOS头中e\_magic，它的高16位是0，低的16位是0x4550，也就是”PE“；IMAGE\_FILE\_HEADER和IMAGE\_OPTIONAL\_HEADER即PE文件头和可选头，后续将给出解析，图二是一张真实的NT头结构及其各个域的取值。



**图二**

**PE文件头和可选头**

文件头的结构如下，其中Machine字段指定该文件对应的运行平台，是x86、x64还是I64等等，可参考项目中的IMAGE\_MACHINE\_TYPE，考虑论文篇幅此处不做详细讨论；Number-OfSections指定了该PE文件中有多少个节，即节表中的项数；TimeDateStamp对应该PE文件的创建时间；PointerToSymbolTable对应COFF文件符号表在文件中的地址偏移量；NumberOfSymbols对应符号表的数量；SizeOfOptionalHeader指定了接下来的可选头的大小；Characteri-stics对应可执行文件的属性，可参考项目中IMAGE\_Characteristics，此处不做详细讨论。

Struct IMAGE\_FILE\_HEADER

{

Public UInt16 Machine; // 目标机器类型

Public UInt16 NumberOfSections; // 节数目

Public UInt32 TimeDateStamp; // 文件被创建时间

Public UInt32 PointerToSymbolTable; // COFF符号表的文件地址偏移量

Public UInt32 NumberOfSymbols; // 符号表中的符号数量

Public UInt16 SizeOfOptionalHeader; // 可选文件头的大小

Public UInt16 Characteristics; // 文件特征标志

}

从中我们发现PE文件头定义了部分基本信息和内容，当加载器加载PE文件时会用到这些内容，如果信息不符合当前运行环境，那么会立刻终止加载PE文件。

接下来就是可选头，虽然可选头被可选两个字修饰，但其实这个头一点都不能少，在不同的平台下可选头的定义是不一样的，例如32位操作系统中可选头就是IMAGE\_OPTIONAL\_HEARDER32，而在64为操作系统中就是IMAGE\_OP-TIONAL\_HEARDER64，我们目前定义的可选头是在32位操作系统下面的：