**PE文件结构分析工具的研究**

**尚佳佳**

**摘要:** 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。本项目通过对PE文件的结构分析，实现资源文件、资源数据、证书信息、函数列表等内容的解构，了解可执行文件的执行机制，能够有效的指导软件设计和软件代码编写，并且可以为测试工作带来更宽广的视野。

**关键词:** PE文件；节表；数据目录；PE头文件

**Research of PE File Formats Analyzing Tool**

**Abstract:** The format of an operating system's executable file is in many ways a mirror of the operating system. Although studying an executable file format isn't usually high on most programmers' list of things to do, a great deal of knowledge can be gleaned this way. This project mainly deconstructs resource file, resource data, certificate information and lists of function in PE file formats via research of the structure of PE file formats. It’ll help to lead the construction of software design and coding, and open mind for testing work.

**Key words:** PE File; Section Table; Data Directory, PE File Header

**1. PE文件概述**

PE文件的全称是Portable Executable，即可移植的可执行文件，我们平时常常看到的EXE，DLL，COM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。因为我们采用C#编写此项目，所以涉及到的PE文件就是以DLL文件。

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（注意：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，后续部分我们将会详细解析PE文件结构。

**1.1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。

32位CPU的寻址能力为4GB，一旦决定采用分页机制，就应当把这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。

PE文件被加载到内存中采用的就是分页机制。

**1.2 PE文件与内存**

PE文件保存在磁盘中，然后被Windows加载器加载到内存中，加载完后我们将称之为映像。了解PE文件第一件非常重要的事情就是保存在磁盘上的PE文件和被加载到内存中的映像非常相似，都是通过十六进制流方式存储，所以加载过程就相当于简单地将PE文件搬运到内存中，是非常容易的一个过程。但是值得注意的是，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(所谓的节我们可以视其为一个容器，它包含了代码和数据等信息) ，而这些节在磁盘上连续存储，但是被加载到内存中之后会按照分页机制存储，这样节与节之间可能存在一些空白区域，如果不是很明白这个特点，可以将其想象成word文档，在某一部分描述的内容完结后，直接开始下一页内容，这样上一页可能存在空白区域。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，它表示距离文件头的偏移；另外一种是针对加载到内存以后在映像中的地址，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。

了解了PE文件在内存中的存储方式之后，我们接下来详细了解PE文件的层次结构。

**1.3 PE文件层次结构**

**1.3.1 DOS 头**

PE文件格式以DOS头这一结构开始，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  } |

DOS头中值得注意的字段有两个，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

所以有时候也会称呼DOS头为DOS MZ Header。至于e\_lfanew，它是一个双字数据，为PE头离文件头部的偏移量，Windows加载器可以直接根据这个值跳过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，具体不做阐述，图一显示了真实情况：

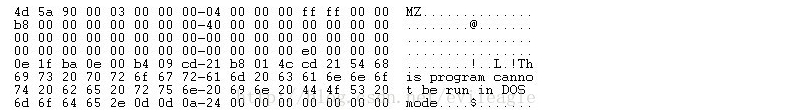


图1-1

**1.3.2 PE（NT） 头**

上面我们了解到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即熟称PE头。根据下面的代码，我们发现PE头中定义了PE文件头以及它的可选头，对于PE文件来说这两个部分是非常重要的，这就导致PE头是最有用的头:

Struct IMAGE\_NT\_HEADERS

{

Public UInt32 Signature;

Public IMAGE\_FILE\_HEADER FileHeader;

Public IMAGE\_OPTIONAL\_HEADER OptionalHeader;

}

其中Signature类似于DOS头中e\_magic，表示成字符就是PE；IMAGE\_FILE\_HEADER和IMAGE\_OPTIONAL\_H-EADER即PE文件头和可选头，后续内容会深入探究，图二是一张真实的NT头结构及其各个域的取值。

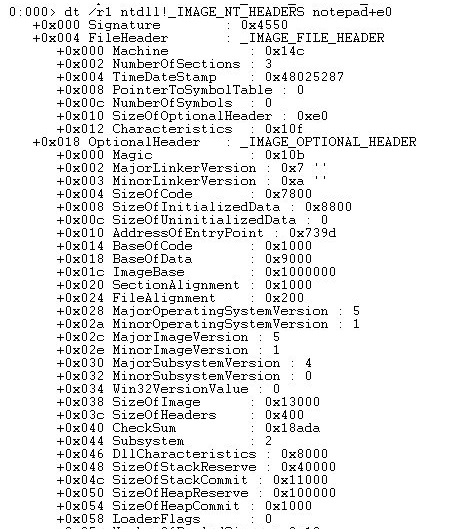


图1-2

**1.3.3 PE文件头**

文件头的结构如以下代码，我们称之为IMAGE\_FILE\_HEADER，它主要描述了文件大部分的基本信息。它从C-OFF文件直接继承而来，基本上结构没有发生变化。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；NumberOfSections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；Characteristics对应可执行文件的属性，可参考项目中IMAGE\_Characteristics。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。

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

}

**1.3.4 PE可选头**

PE可选头的结构定义如下，一共31个字段成员，其中最重要的有7个。AddressOfEntry指明了程序执行入口RVA；通过ImageBase可以获知映像默认加载的地址；SectionAlignment表示内存中节的对齐大小；FileAlign-ment表示在磁盘文件中区块的对齐大小；SizeOfImage指明文件加载到内存中的大小；SizeOfHeaders指明所有头部加载到内存中的大小；NumberOfRvaAndSizes表明数据目录的个数（数据目录将会在后续讲解）；当然最重要的一个成员就是DataDirectory，它表示数据目录表，每一项都代表不同的资源，后续将会探究。

struct IMAGE\_OPTIONAL\_HEADER\_X64

{

public UInt16 Magic;

byte MajorLinkerVersion;

byte MinorLinkerVersion;

public UInt32 SizeOfCode;

public UInt32 SizeOfInitializedData;

public UInt32 SizeOfUninitializedData;

public UInt32 AddressOfEntryPoint;

public UInt32 BaseOfCode;

public UInt32 BaseOfData;

public UInt32 ImageBase;

public UInt32 SectionAlignment;

public UInt32 FileAlignment;

public UInt16 MajorOperatingSystemVersion;

public UInt16 MinorOperatingSystemVersion;

public UInt16 MajorImageVersion;

public UInt16 MinorImageVersion;

public UInt16 MajorSubsystemVersion;

public UInt16 MinorSubsystemVersion;

public UInt32 Win32VersionValue;

public UInt32 SizeOfImage;

public UInt32 SizeOfHeaders;

public UInt32 CheckSum;

public UInt16 Subsystem;

public UInt16 DllCharacteristics;

public UInt64 SizeOfStackReserve;

public UInt64 SizeOfStackCommit;

public UInt64 SizeOfHeapReserve;

public UInt64 SizeOfHeapCommit;

public UInt32 LoaderFlags;

public UInt32 NumberOfRvaAndSizes;

IMAGE\_DATA\_DIRECTORY[] DataDirectory;

}

**1.3.5 数据目录**

我们在讲述可选头的时候，发现它有个IMAGE\_DATA\_DIRECTORY数组，即数据目录表，该数组索引的不同它代表的结构也不同，索引为0代表导出表的数据目录，索引为1代表导入表的数据目录。也就是说我们可以通过这个数组快速地找到对应数据结构的数据目录。通过分析以下数据目录结构我们发现它里面存储的是虚拟地址和大小，这样设计的目的是为了方便系统直接快速地址定位到对应的数据结构。

struct IMAGE\_DATA\_DIRECTORY

{

public UInt32 VirtualAddress;

public UInt32 Size;

}

数据目录是一共有16个（WINNT.H中定义为IMAGE\_NUMBEROF\_DIRECTORY\_ENTRIES）元素的结构数组。每个数组元素所指代的内容已经被预先定义好了。WINNT.H文件中的这些IMAGE\_DIRECTORY\_ENTRY\_*xxx*定义就是数据目录的索引（从0到15）。下表描述了每个IMAGE\_DIRECTORY\_ENTRY\_*xxx*值所指代的内容。由它们指向的许多数据结构将在本文的第二部分中详细描述。

| 值 | 描述 |
| --- | --- |
| IMAGE\_DIRECTORY\_ENTRY\_EXPORT | 指向导出表（IMAGE\_EXPORT\_DIRECTORY结构）。 |
| IMAGE\_DIRECTORY\_ENTRY\_IMPORT | 指向导入表（IMAGE\_IMPORT\_DESCRIPTOR结构数组）。 |
| IMAGE\_DIRECTORY\_ENTRY\_RESOURCE | 指向资源（IMAGE\_RESOURCE\_DIRECTORY结构）。 |
| IMAGE\_DIRECTORY\_ENTRY\_EXCEPTION | 指向异常处理程序表（IMAGE\_RUNTIME\_FUNCTION\_ENTRY结构数组）。它特定于CPU，用于基于表的异常处理。适用于除x86之外所有类型的CPU。 |
| IMAGE\_DIRECTORY\_ENTRY\_SECURITY | 指向WIN\_CERTIFICATE结构列表（文件签名）。此结构定义在WinTrust.H文件中。它并不作为映像的一部分被映射进内存。因此VirtualAddress域是文件偏移，而不是RVA。 |
| IMAGE\_DIRECTORY\_ENTRY\_BASERELOC | 指向基址重定位信息。 |
| IMAGE\_DIRECTORY\_ENTRY\_DEBUG | 指向IMAGE\_DEBUG\_DIRECTORY结构数组。其中的每个元素描述了映像中的一些调试信息。要获得IMAGE\_DEBUG\_DIRECTORY结构的数目，用Size域除以IMAGE\_DEBUG\_DIRECTORY结构的大小。早期的Borland链接器将这个IMAGE\_DATA\_DIRECTORY项的Size域设置成IMAGE\_DEBUG\_DIRECTORY结构的数目，而不是数组的大小。 |
| IMAGE\_DIRECTORY\_ENTRY\_ARCHITECTURE | 指向与平台相关的数据，这个数据是一个IMAGE\_ARCHITECTURE\_HEADER结构数组。x86平台和IA-64平台并不使用，但好像已经用于DEC/Compaq Alpha平台。 |
| IMAGE\_DIRECTORY\_ENTRY\_GLOBALPTR | 在某些平台上，其VirtualAddress域保存的是全局指针（Global Pointer ，GP）的RVA。x86平台上不使用，但IA-64平台上使用。Size域并未使用。要获取更多关于IA-64 GP方面的信息，可以参考[2000年11月的Under The Hood专栏。](http://msdn2.microsoft.com/en-us/magazine/bb985017.aspx) |
| IMAGE\_DIRECTORY\_ENTRY\_TLS | 指向线程局部存储（Thread Local Storage）初始化节。 |
| IMAGE\_DIRECTORY\_ENTRY\_LOAD\_CONFIG | 指向IMAGE\_LOAD\_CONFIG\_DIRECTORY结构。此结构中的信息特定于Windows NT、Windows 2000和Windows XP（例如GlobalFlag值）。如果你的可执行文件要使用这个结构，需要定义一个名称为\_\_load\_config\_used，类型为IMAGE\_LOAD\_CONFIG\_DIRECTORY的全局结构体。对于非x86平台，这个名称需要被定义成\_load\_config\_used（单下划线）。如果你想使用IMAGE\_LOAD\_CONFIG\_DIRECTORY结构，必须使用这个技巧才能在你的C++代码中得到正确的名字。链接器看到的符号名一定要是\_\_load\_config\_used（带两个下划线）。C++编译器要在全局符号前加一个下划线。另外，它还使用类型信息来修饰（decorate）全局符号。因此要使一切正常，你应该像下面这个样子使用：  extern "C"  IMAGE\_LOAD\_CONFIG\_DIRECTORY \_load\_config\_used = {...} |
| IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT | 指向IMAGE\_BOUND\_IMPORT\_DESCRIPTOR结构数组。每个结构对应于这个映像已经绑定的一个DLL。这个结构中的日期/时间戳（TimeDateStamp域）可以让加载器快速确定这个绑定是否是最新的。如果不是，加载器将忽略绑定信息，并正常地解析导入的函数。 |
| IMAGE\_DIRECTORY\_ENTRY\_IAT | 指向第一个导入地址表（IAT）的开头。对应于每一个导入的DLL都有一个相应的IAT，并且它们在内存中依次排列。Size域指出了所有IAT的总大小。加载器在解析导入符号期间使用这个地址和大小临时将包含IAT的页面标记为可读/可写。 |
| IMAGE\_DIRECTORY\_ENTRY\_DELAY\_IMPORT | 指向延迟加载信息，它是CImgDelayDescr结构数组，这个结构被定义在Visual C++的DELAYIMP.H文件中。直到首次调用延迟加载的DLL中的函数时这个DLL才会被加载。特别需要注意的是：Windows并不知道关于延迟加载DLL方面的任何信息。延迟加载特性完全是由链接器与运行时库来实现的。 |
| IMAGE\_DIRECTORY\_ENTRY\_COM\_DESCRIPTOR | 这个值在最新的系统头文件（CorHdr.h）中被更名为IMAGE\_DIRECTORY\_ENTRY\_COMHEADER。它指向可执行文件中的.NET信息中的顶层信息，包括元数据。这个信息保存在IMAGE\_COR20\_HEADER结构中。 |

**1.3.6 节表**

节表本质上就像包含映像中其他每个节相关信息的电话本，在这个电话本中这些节以他们开始时候相对虚拟地址排序，而不是按照首字母的方式。

在一个NE文件中，你的程序代码和数据是保存在明确的段里面，部分NE头是一些数据结构的数组，数组中的每个结构包含了段的相关信息，这些信息包括段的类型（代码还是数据），它的大小以及它在文件中的具体位置。在PE文件中，节表就类似于NE文件中的段，不同的是PE文件的节表不会存储每个节对应的数据和代码，相反只会存储每个节映射到内存中所对应的地址，因此系统可以根据节表中的地址直接找到相对应的节。下面是个典型的exe文件对应的节表，其中.xxx代表节的名称。

01 .text VirtSize: 00005AFA VirtAddr: 00001000

raw data offs: 00000400 raw data size: 00005C00

relocation offs: 00000000 relocations: 00000000

line # offs: 00009220 line #'s: 0000020C

characteristics: 60000020

CODE MEM\_EXECUTE MEM\_READ

02 .bss VirtSize: 00001438 VirtAddr: 00007000

raw data offs: 00000000 raw data size: 00001600

relocation offs: 00000000 relocations: 00000000

line # offs: 00000000 line #'s: 00000000

characteristics: C0000080

UNINITIALIZED\_DATA MEM\_READ MEM\_WRITE

03 .rdata VirtSize: 0000015C VirtAddr: 00009000

raw data offs: 00006000 raw data size: 00000200

relocation offs: 00000000 relocations: 00000000

line # offs: 00000000 line #'s: 00000000

characteristics: 40000040

INITIALIZED\_DATA MEM\_READ

04 .data VirtSize: 0000239C VirtAddr: 0000A000

raw data offs: 00006200 raw data size: 00002400

relocation offs: 00000000 relocations: 00000000

line # offs: 00000000 line #'s: 00000000

characteristics: C0000040

INITIALIZED\_DATA MEM\_READ MEM\_WRITE

05 .idata VirtSize: 0000033E VirtAddr: 0000D000

raw data offs: 00008600 raw data size: 00000400

relocation offs: 00000000 relocations: 00000000

line # offs: 00000000 line #'s: 00000000

characteristics: C0000040

INITIALIZED\_DATA MEM\_READ MEM\_WRITE

06 .reloc VirtSize: 000006CE VirtAddr: 0000E000

raw data offs: 00008A00 raw data size: 00000800

relocation offs: 00000000 relocations: 00000000

line # offs: 00000000 line #'s: 00000000

characteristics: 42000040

INITIALIZED\_DATA MEM\_DISCARDABLE MEM\_READ

**1.3.7 节**

PE文件中节对应结构是IMAGE\_SECTION\_HEADER，它的内部定义如下代码：

struct IMAGE\_SECTION\_HEADER

{

public byte[] Name1;

public UInt32 Misc;

public UInt32 M\_PhysicalAddress;

public UInt32 M\_VirtualSize;

public UInt32 VirtualAddress;

public UInt32 SizeOfRawData;

public UInt32 PointerToRawData;

public UInt32 PointerToRelocations;。

public UInt32 PointerToLineNumbers;

public UInt16 NumberOfRelocations;

public UInt16 NumberOfLinenumbers;

public UInt32 Characteristics;

}

其中Name1代表节名称，通常以‘.’开头，比如 .text；VirtualAddress表示该节装载到内存中对应的RVA；

SizeOfRawData指明该节在内存中对应的大小；PointerToRawData表示该节在磁盘文件中的偏移；Characteristics表示节的属性，具体有哪些可以查看项目中Section\_Characteristics。

**1.3.8 PE导入表**

接下来将揭示数据目录的第二项IMAGE\_DIRECTORY\_ENTRY\_IMPORT，即导入表。大家可能注意到在数据目录中有几项名字和导入表有关系，其中有: IMAGE\_DIRECTORY\_ENTRY\_IMPORT，IMAGE\_DIRECTORY\_ENTRY\_IAT和IMAGE\_DIRECTORY\_ENTRY\_DELAY\_IMPORT这几个导入都是用来干什么的，他们之间又是什么关系呢？听我慢慢道来。

* IMAGE\_DIRECTORY\_ENTRY\_IMPORT就是我们通常所知道的导入表，在PE文件加载时，会根据这个表里的内容加载依赖的DLL，并填充所需函数的地址。
* IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT叫做绑定导入表，在第一种导入表导入地址的修正是在PE加载时完成，如果一个PE文件导入的DLL或者函数多那么加载起来就会略显的慢一些，所以出现了绑定导入，在加载以前就修正了导入表，这样就会快一些。
* IMAGE\_DIRECTORY\_ENTRY\_DELAY\_IMPORT叫做延迟导入表，一个PE文件也许提供了很多功能，也导入了很多其他DLL，但是并非每次加载都会用到它提供的所有功能，也不一定会用到它需要导入的所有DLL，因此延迟导入就出现了，只有在一个PE文件真正用到需要的DLL，这个DLL才会被加载，甚至于只有真正使用某个导入函数，这个函数地址才会被修正。
* IMAGE\_DIRECTORY\_ENTRY\_IAT是导入地址表，前面的三个表其实是导入函数的描述，真正的函数地址是被填充在导入地址表中的。

举个实际的例子，看一下下面这张图：

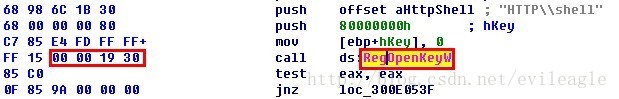


图1-3

这个代码调用了一个RegOpenKeyW的导入函数，我们看到其opcode是FF 15 00 00 19 30气质FF 15表示这是一个间接调用，即call dword ptr [30190000] ;这表示要调用的地址存放在30190000这个地址中，而30190000这个地址在导入地址表的范围内，当模块加载时，PE 加载器会根据导入表中描述的信息修正30190000这个内存中的内容。

PE文件中导入表其实是一个IMAGE\_IMPORT\_DESCRIPTOR的数组，每个导入的DLL对应一个IMAGE\_IMPO-RT\_DESCRIPTOR。

**1.3.8 PE导出表**

和导入表相反的表就是导出表，导出表的作用就是列出一系列被外部exe或者dll使用的方法，一个PE文件中将它导出的信息存储在.edata节中，.edata节中主要是些方法名对应的表格，入口地址以及导出的初始化值。.edata的一开始就是IMAGE\_EXPORT\_DIRECTORY结构。

**2. 项目设计与实现**

**2.1 项目开发环境**

2.1.1开发环境简介

操作系统：Microsoft Windows 10

开发工具：Visual Studio 2015

2.1.2 Visual Studio 2015简介

Visual Studio是微软公司推出的目前最流行的的应用程序开发环境，Visual Studio在三个方面为开发人员提供了关键改进：快速的应用程序开发，高效的团队协作和突破性的用户体验。本项目主要使用Visual Studio的Console平台编写。

**2.2 相关技术简介**

2.2.1 C#语言简介

C#语言是一门非常高效，现代，优雅，类型安全，平台独立的面向对象语言。代码经过编译之后会被转化成中间语言（IL），通常会生成 .exe和 .dll文件，再由CLR加载IL语言的元数据，然后再转化成机器语言进行运行。

2.2.2 控制台程序简介

控制台程序是为了兼容DOS程序而设立的，这种程序的执行就好像在一个DOS窗口中执行一样，没有自己的界面。

所谓的控制台应用程序，就是能够运行在MS-DOS环境中的程序。控制台应用程序通常没有可视化的界面，只是通过字符串来显示或者监控程序。控制台程序常常被应用在测试、监控等用途，用户往往只关心数据，不在乎界面。

**2.3项目设计**

2.3.1 实体结构的设计与实现

根据第一章内容我们深入的解析了PE文件的内部结构，针对它每一个部分都详细讲解了。接下来我们考虑如何将各个部分用C#结构体封装起来，在封装之前先了解下C#的结构体。结构体即Struct，它是值类型，轻量级的一种结构，里面可以定义一系列的成员，从第一章可以发现我已经贴了一些代码，都是采用结构体。我们设计的结构体应该准确，干净，易用。图2-1显示了所定义的所有Headers；图2-2显示了所定义Section。详细信息请参见相关代码。

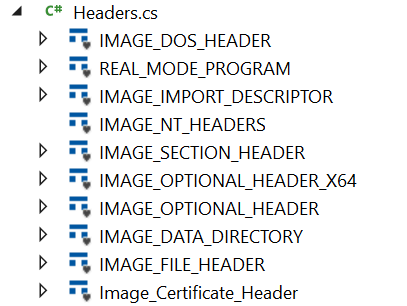


图2-1

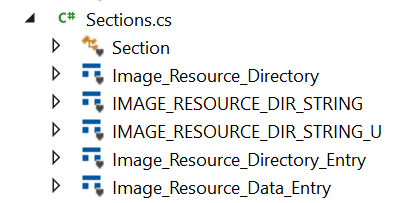


图2-2

2.3.2 代码解析

代码解析一：

class Program

{

static void Main(string[] args)

{

string file = @"d:\\_statvw.dll";

if (File.Exists(file))

{

PEFile peFile = new PEFile(file);

for (int i = 0; i < peFile.Sections.Length; i++)

{

if (peFile.Sections[i].SectionInstance != null &&peFile.Sections[i].SectionInstance.GetType().Equals(typeof(Image\_Resource\_Directory)))

{

((Image\_Resource\_Directory)peFile.Sections[i].

SectionInstance).ExportResource("d:\\Format", 0, 0);

}

}

}

}

}

这段代码是程序的主入口，其中file变量指明了想要解析的文件路径，接下来通过判断该路径下是否存在该文件来确定是否应该进一步操作，如果存在直接以该路径为参数初始化一个PE文件，然后将该文件所有节对应的资源全部输出到D盘的Format文件夹下。