**第 4 题 Windows PE文件解析**

张旭，软件工程1805班，U201817106 组长

蒋熙浩，软件工程1805班，U201817097

赵真，软件工程1805班，U201817105

张惊天，软件工程1805班，U201817108

何旺，软件工程1805班，U201817102

一. 功能需求（或任务描述）………………………………………………………………………………2

二. 程序设计思路（ 或整体结构或整体流程）……………………………………………………2

三. 开发环境和配置 （或主要模块和函数分析）……………………………………………… 3

四. 关键技术和难点分析……………………………………………………………………………………21

五. 运行和测试过程（或结论或总结）…………………………………………………………… 22

六. 参考网址……………………………………………………………………………………………………… 22

**一. 功能需求 （或任务描述）**

1.分析Windows中PE文件的格式和作用

2.结合C程序验证，C程序要求有2个全局变量（1个已初始化，1个未初始化），2个自定义函数（一个带参数，1个不带参数），每个函数都有 2个局部变量。都有返回值

**二. 程序设计思路（ 或整体结构或整体流程）**

1.PE文件的结构一般如下图所示，DOS头，NT头，节表，以及各种具体的节（图中并没有展示所有的节）



PE文件的执行顺序：

1、PE文件被执行，PE装载器为文件在内存分配一个空的位置。创建进程和主线程。

2、PE装载器检查 DOS MZ header 里的 PE header 偏移量。如果找到，则跳转到 PE header。

3、PE装载器检查 PE header 的有效性。如果有效，就跳转到PE header的尾部。

4、紧跟 PE header 的是节表。PE装载器读取其中的节信息，并采用文件映射方法将这些节映射到内存，同时赋上节表里指定的节属性。

5、PE文件映射入内存后，PE装载器将处理PE文件中类似 import table（引入表）逻辑部分。

**三. 开发环境和配置 （或主要模块和函数分析）**

**解析工具：010 Editor，配置：安装EXE.bt模板**

接下来分析PE文件的结构。

PE文件的格式部分在winnt.h头文件中，打开winnt.h，搜索Image Format即可到达

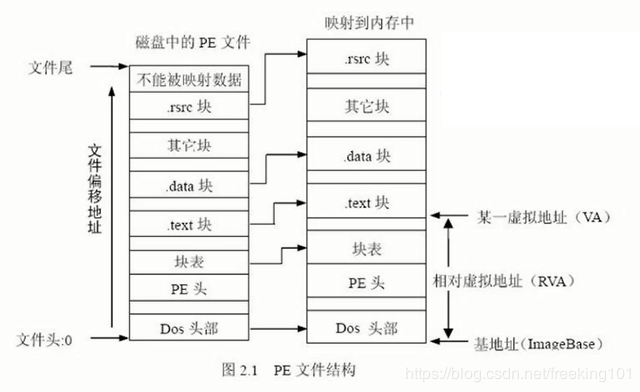
PE文件结构说明：

1、DOS头用来兼容MS-DOS，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，未必是上图中的三个。

 由于进程分页加载，所以存在页面对齐，在内存中中间会有间隙，以0填充

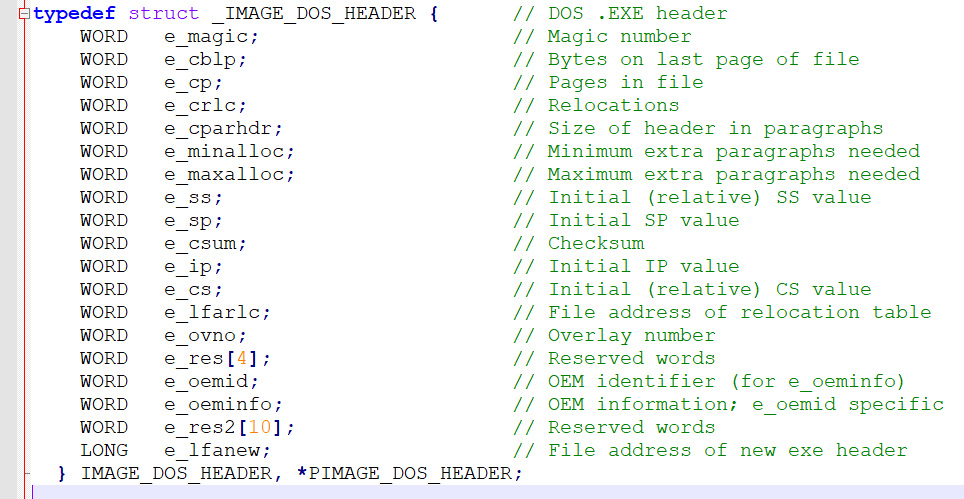
由于存在不同的对齐方式，所以在PE文件内部对地址的描述也采用了两种方式，针对在硬盘上存储文件中的地址，称为原始存储地址或物理地址表示距离文件头的偏移；另外一种是针对加载到内存以后映象中的地址，称为相对虚拟地址（RVA），表示相对内存映象头的偏移。在内存中使用VA， VA 与 RVA 满足下面的换算关系： **RVA + ImageBase = VA**

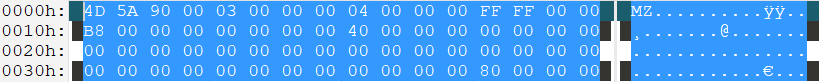
具体分析各个部分格式：

1、DOS头

PE文件DOS头由两部分组成DosHeader和DosStub，都是结构体

（1）DosHeader：





DOS头中主要的两个成员：

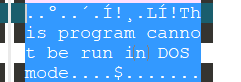
* e\_magic：一个WORD类型，值是一个常数0x4D5A，用文本编辑器查看该值位‘MZ’，可执行文件必须都是'MZ'开头。
* e\_lfanew：为32位可执行文件扩展的域，用来表示DOS头之后的NT头相对文件起始地址的偏移。

在示例程序中可以看到e\_lfanew = 80h

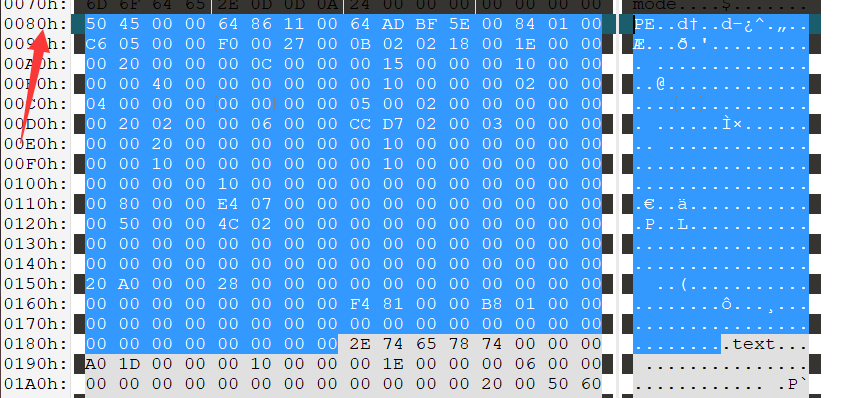
（2）DosStub：



是一个字符串，一串提示信息

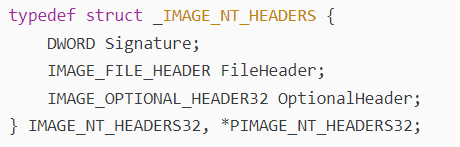


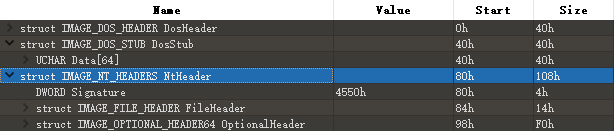
2、NT头（80h）



NT头中包含 Windows PE 文件的主要信息，其中包括PE签名、PE头文件（IMAGE\_FILE\_HEADER）和PE可选头（IMAGE\_OPTIONAL\_HEADER32）三个部分。

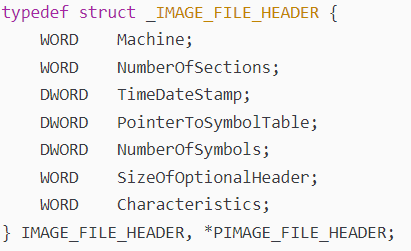
结构体描述为：



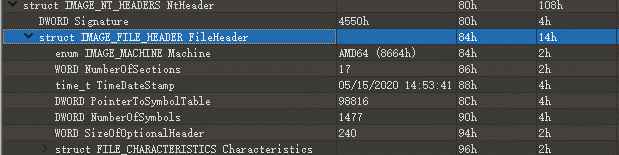


如图，在示例exe程序中，该NT头起始位置为0080h,在其最开头即为NT头的（1）第一部分内容：PE签名，类似于DOS头中的e\_magic，取值为4550h占用4个字节，在右边可看到翻译出的文本为“PE..”四个字符

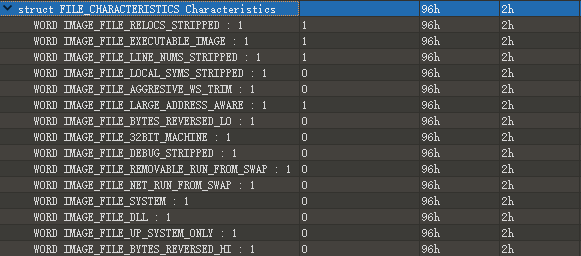
（2）NT头的第二部分是PE头文件, PE文件头中定义了PE文件的基本信息和属性，这些属性可以被加载器利用，加载器在加载时会检查PE文件头中定义的属性，若其不满足当前的运行环境，将会终止加载该PE文件.



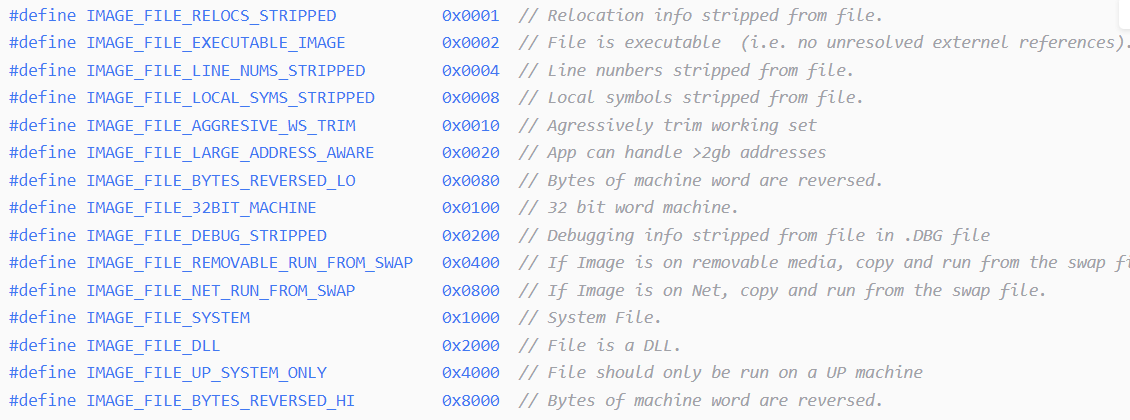
如下图所示：



* Machine描述了该文件的运行平台，如x86、x64 等等，上图示例中为ADM64(8664h)
* NumberOfSections描述了该PE文件中有多少个节，即节表中的项数，17个节
* TimeDateStamp描述了PE文件的创建时间，该文件创建时间为2020/05/15 14:53:41
* PointerToSymbolTable指出了COFF文件符号表在文件中的偏移。
* NumberOfSymbols表明出了符号表的数量。
* SizeOfOptionalHeader标明了紧随其后的PE可选头的总大小。
* Characteristics结构体标出了可执行文件的各种属性，其结构如下图：

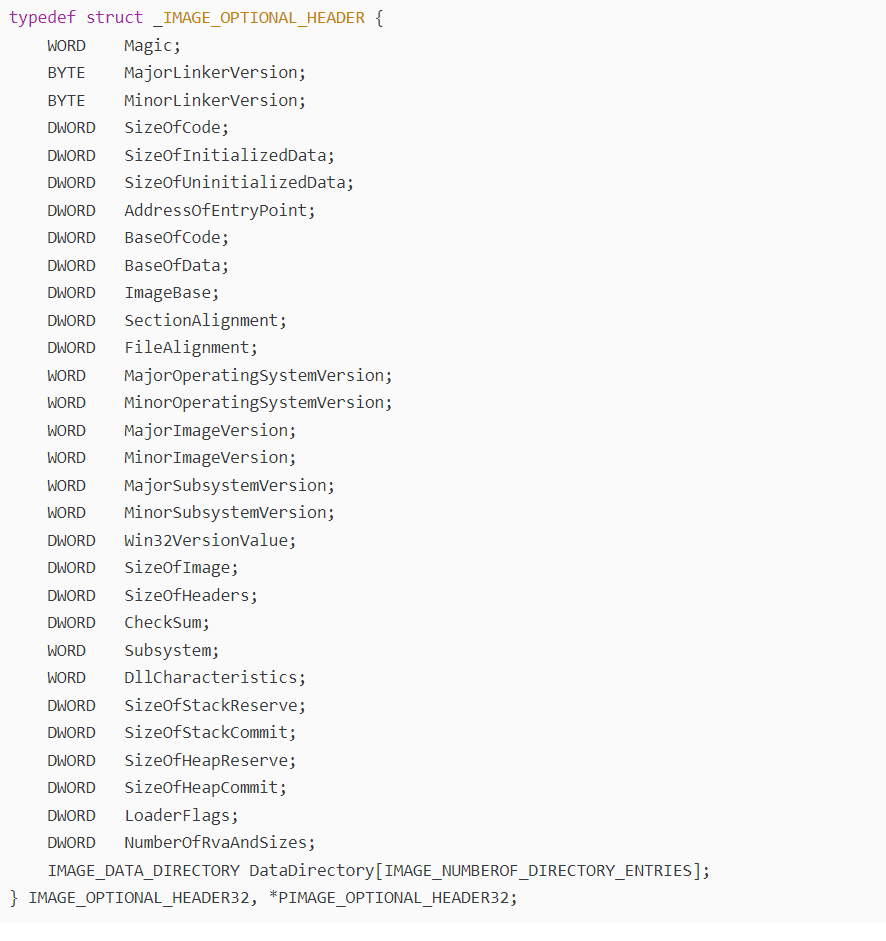
****

可见该EXE满足多个属性，通过异或连接。

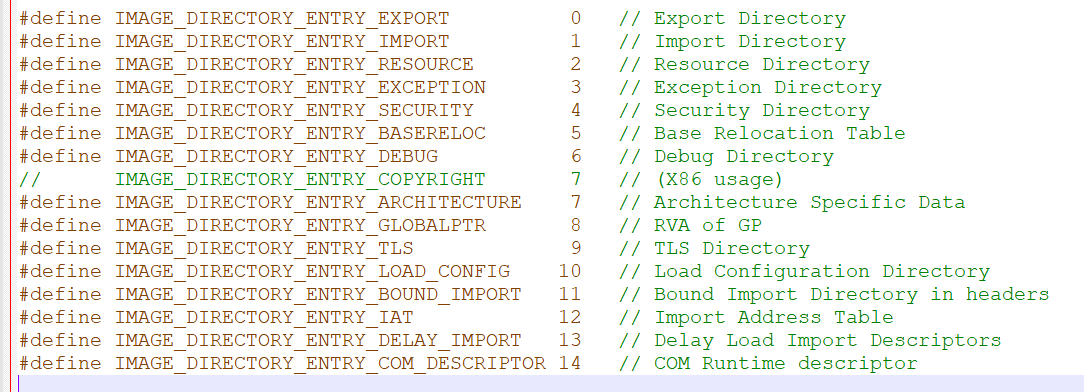


（3）PE可选头：

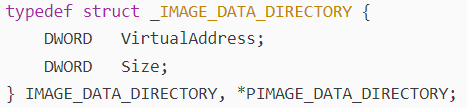
NT头的第三部分是PE可选头，其结构如下：



* Magic表示可选头的类型；上图取值为PE64，表示该可选头为64位可选头
* MajorLinkerVersion 和 MinorLinkerVersion的值指出了链接器的版本号
* SizeOfCode：代码段的长度，如果有多个代码段，则是代码段长度的总和。上图取值表明示例程序中代码段总长度为7680
* SizeOfInitializedData：初始化的数据长度，取值为7680
* SizeOfUninitializedData：未初始化的数据长度，取值为3072
* AddressOfEntryPoint：程序入口的相对虚拟地址RVA，对于该exe文件来说这个地址可以理解为代码中main函数的RVA。
* BaseOfCode：代码段起始地址的RVA。
* BaseOfData：数据段起始地址的RVA。
* ImageBase：PE文件的映像被加载到内存中，而ImageBase则指出了映象建议加载的基地址，如果无法加载到这个地址，系统会为其选择其他地址。
* SectionAlignment：节对齐，PE中的节被加载到内存时会按照这个域指定的值来对齐，图中该值为4096，即4k，1000h
* FileAlignment：取值为512，节在文件中按此值对齐，该值应当小于等于SectionAlignment，即200h
* MajorOperatingSystemVersion、MinorOperatingSystemVersion：指出该文件所需操作系统的版本号
* MajorImageVersion、MinorImageVersion：映象的版本号，该条目的值是由开发者自己指定的，由连接器填写。
* MajorSubsystemVersion、MinorSubsystemVersion：所需子系统版本号。
* Win32VersionValue：保留，取值为0。
* SizeOfImage：映象的大小，PE文件加载到内存中空间是连续的，这个值指定占用虚拟空间的大小。
* SizeOfHeaders：所有文件头（包括节表）的大小，这个值是以FileAlignment对齐的。
* CheckSum：映象文件的校验和。
* Subsystem：运行该PE文件所需的子系统，示例中取值为WINDOWS\_CUI（3），表示映像需要windows字符子系统
* SizeOfStackReserve：运行时为每个线程栈保留内存的大小。
* SizeOfStackCommit：运行时每个线程栈初始占用内存大小。
* SizeOfHeapReserve：运行时为进程堆保留内存大小。
* SizeOfHeapCommit：运行时进程堆初始占用内存大小。
* LoaderFlags：保留，必须为0。
* NumberOfRvaAndSizes：数据目录的项数，
* DataDirectory：数据目录，这是一个数组，数组中的每一项对应一个特定的数据结构，包括导入表、导出表等等，其内容如下：



其中每一项的定义如下：



VirtualAddress定义了该项的相对虚拟地址

Size定义了该项的大小

在这些成员中比较重要的就是前两个，导出表和导入表

（1）PE导出表：

Windows在加载一个程序后会在内存中为该程序开辟一个单独的虚拟地址空间，程序用到的函数会加载到其地址空间运行。而有一些函数会有很多程序都能用到，因此采用将一些函数封装成动态链接库，程序在需要时加载动态链接库的方式可以大大节约内存空间。

导出表是用来记载动态链接库的一些导出信息的结构，其主要成分是一个表格，内含函数名称、输出序数等等。结构体定义如下：



其中主要的一些有意义字段如下：

* TimeDateStamp记录导出表生成的时间戳，由连接器生成。
* Name：模块的名字，指向一个定义模块名称的字符串
* Base：导出函数序号的起始值，按序号导出函数的序号值从Base开始递增。
* NumberOfFunctions：文件中所有导出函数的总数
* NumberOfNames：可以通过函数名的方式导出的函数的总数。
* AddressOfFunctions：一个RVA，指向一个DWORD数组，数组中的每一项是一个导出函数的RVA，顺序与导出序号相同。
* AddressOfNames：一个RVA，依然指向一个DWORD数组，数组中的每一项仍然是一个RVA，指向函数名字的字符串。
* AddressOfNameOrdinals：一个RVA，还是指向一个WORD数组，数组中的每一项与AddressOfNames中的每一项对应，表示该名字的函数在AddressOfFunctions中的序号。

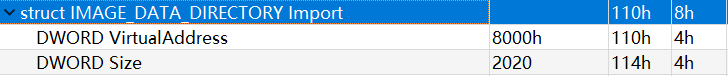
通过上面导出表的结构可以看出，函数导出的方式有按名字导出和按序号导出两种，每种导出方式在导出表中的描述方式也是不同的。

导出表一般存在于.dll文件中，而.exe文件中一般不存在导出表。

但是拥有导入表。

（2）导入表

IMAGE\_DIRECTORY\_ENTRY\_IMPORT，即导入表



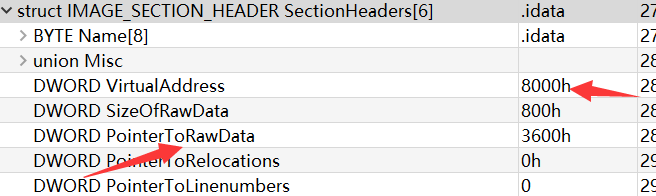
在PE文件加载时，会根据这个表里的内容加载依赖的DLL，并填充所需函数的地址。

* IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT：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：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：导入函数地址表，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函数地址表中的。

这些结构的成员均为VA和size。

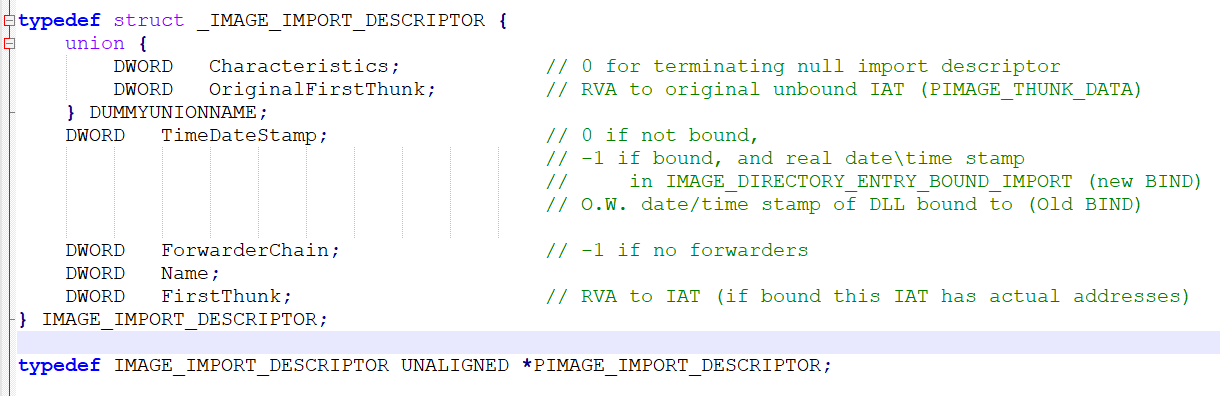
将VA转化为文件偏移地址可以得到相应的导入表中的信息。

VA=8000h，对照节表，

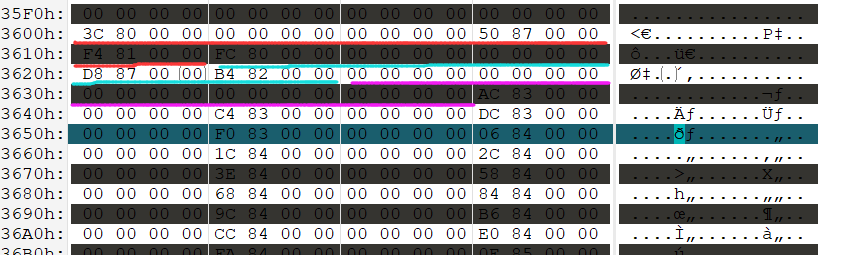


可以得到偏移为0，

文件偏移地址为3600h+0 = 3600h，可以找到导入表对应的数组，简称IID。在这个数组中并没有指出有多少项，但是他最后以一个全为NULL（0）的IID结尾。



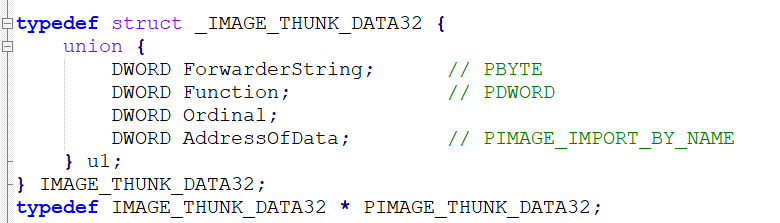
结构的大小为5\*4=20字节。



可以看到第三组为0，所以只有两个\_IMAGE\_IMPORT\_DESCRIPTOR。每个导入的DLL都会对应一个，也就是说示例EXE存在两个导入DLL。

* Characteristics和OriginalFirstThunk：一个联合体，如果是数组的最后一项Characteristics为0，否则OriginalFirstThunk保存一个RVA，指向一个IMAGE\_THUNK\_DATA的数组，这个数组中的每一项表示一个导入函数。
* TimeDateStamp：映象绑定前，这个值是0，绑定后是导入模块的时间戳。
* ForwarderChain：转发链，如果没有转发器，这个值是-1。
* Name：一个RVA，指向导入模块的名字，所以一个IMAGE\_IMPORT\_DESCRIPTOR描述一个导入的DLL。
* FirstThunk：也是一个RVA，也指向一个IMAGE\_THUNK\_DATA数组。

其中OriginalFirstThunk指向一个IMAGE\_THUNK\_DATA的数组INT，FirstThunk指向一个IMAGE\_THUNK\_DATA数组IAT。（都是以0标志结束）



4个字节大小

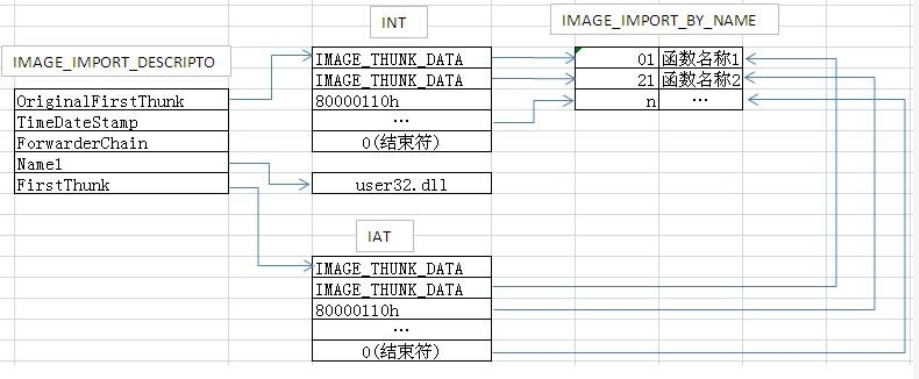
* ForwarderString是转发用的
* Function表示函数地址，如果函数是按序号导入Ordinal就有用了，
* AddressOfData指向名字信息，如果函数按名字导入。

可以看出这个结构体就是一个大的union，但是在不同时刻代表不同的意义那到底应该是名字还是序号，该如何区分呢？可以通过Ordinal判断，如果Ordinal的最高位是1，就是按序号导入的，这时候，低16位就是导入序号，如果最高位是0，则AddressOfData是一个RVA，指向一个IMAGE\_IMPORT\_BY\_NAME结构，用来保存名字信息，由于Ordinal和AddressOfData实际上是同一个内存空间，所以AddressOfData其实只有低31位可以表示RVA，但是一个PE文件不可能超过2G，所以最高位永远为0，这样设计很合理的利用了空间。实际编写代码的时候微软提供两个宏定义处理序号导入：IMAGE\_SNAP\_BY\_ORDINAL判断是否按序号导入，IMAGE\_ORDINAL用来获取导入序号。

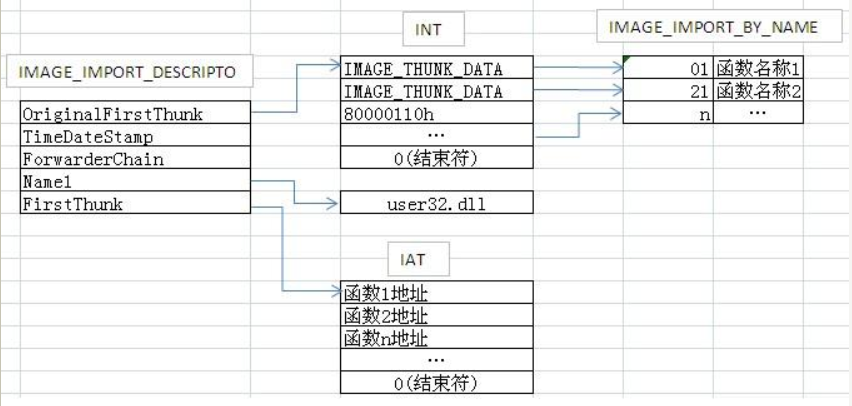
 FirstThunk和OriginalFirstThunk都指向IMAGE\_THUNK\_DATA数组，但是却是不同的。OriginalFirstThunk指向的是单独的一项，而且不能被修改，成为INT。FirstThunk指向的事实上是由PE加载器重写的。

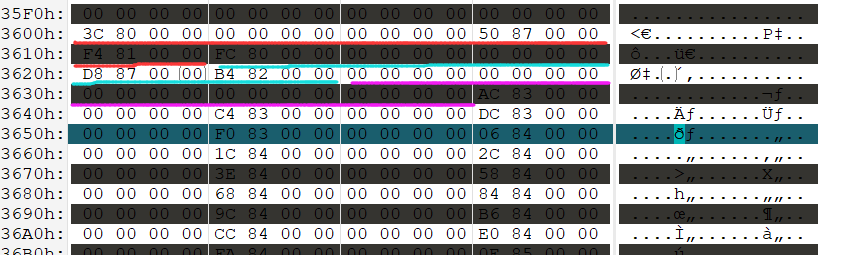
在PE文件加载以前或者说在导入表未处理以前，FirstThunk所指向的数组与OriginalFirstThunk中的数组虽不是同一个，但是内容却是相同的，都包含了导入信息，而在加载之后，FirstThunk中的Function开始生效，他指向实际的函数地址，因为FirstThunk实际上指向IAT中的一个位置，IAT就充当了IMAGE\_THUNK\_DATA数组，加载完成后，这些IAT项就变成了实际的函数地址，即Function的意义。

PE加载前：

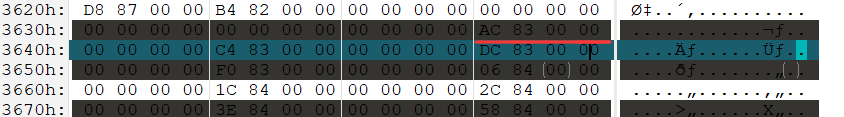


PE加载后：

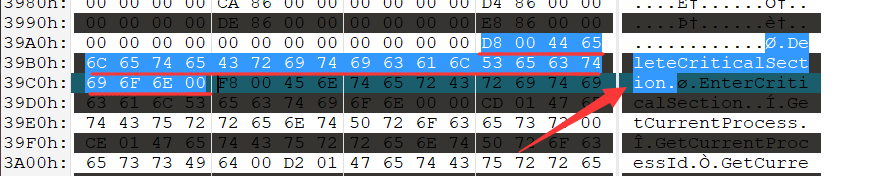
  
PE加载器首先搜索OriginalFirstThunk，找到之后加载程序迭代搜索数组中的每个指针，找到每个IMAGE\_IMPORT\_BY\_NAME结构所指向的输入函数的地址，然后加载器用函数真正入口地址来代替由FirstThunk数组中的一个入口，因此称为输入地址表IAT。  
结合示例程序分析：



首先看第一个OriginalFirstThunk为00 00 80 3C，所以偏移为3C，文件偏移地址为3600h+3ch = 363ch

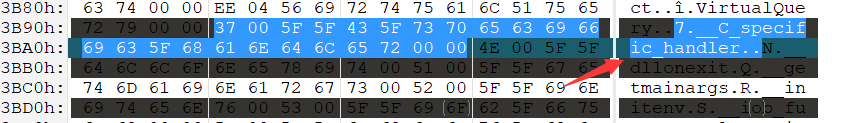


找到了INT，最高位为0，所以RVA=83ACh，文件偏移地址为39Ach



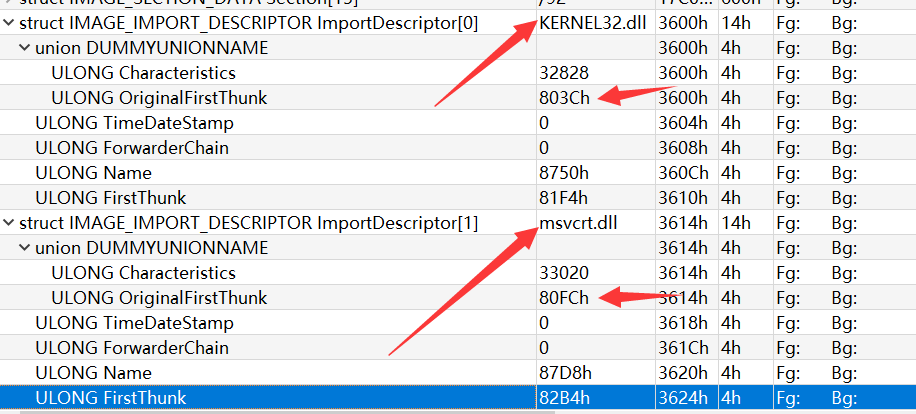
可以得到导入的函数名称

同理可以得到第二个对应文件偏移地址3B94h



得到第二个名称

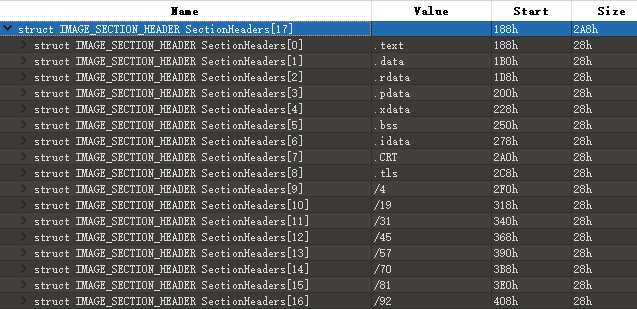
与模板中的对照



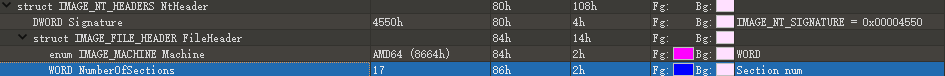
可以看出计算正确，找到了函数名称。

3、节表

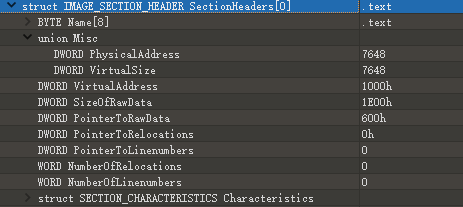
节表由一系列的IMAGE\_SECTION\_HEADER结构排列而成，每个结构用来描述一个节。其排列顺序和节在文件中的排列顺序一致。节表以一个空的IMAGE\_SECTION\_HEADER结构作为结尾，因此节表中IMAGE\_SECTION\_HEADER结构的总数等于节的数量+1。



示例程序有16个节，故节表中有17个IMAGE\_SECTION\_HEADER结构。这个数字保存在Nt头的NumberOfSections中。



所有IMAGE\_SECTION\_HEADER结构具有相同的结构，以示例程序节表中第一个IMAGE\_SECTION\_HEADER结构为例，这个结构描述了程序中的.text节：



**BYTE Name**：8个字节，记录节的名称的ASCⅡ码。



**VirtualSize**：节在没有进行对齐处理前的实际大小。例如该节的实际大小为7648字节。

**VirtualAddress**：该节装载到内存中的RVA地址，这个地址是按照内存页来对齐的。

**SizeOfRawData**：该节在磁盘中所占的大小。数值等于节的实际大小按照Nt头中记录的FileAlignment的值对齐后的大小。例如，该文件的FileAlignment的值为512字节，



该节的实际大小为7648字节，故该节的SizeOfRawData值=7648/512的值向上取整再乘以512，得15\*512=7680，即16进制下的1E00h。

**PointerToRawData**：节在磁盘文件中所处的位置，数值等于从文件头开始算起的偏移量。

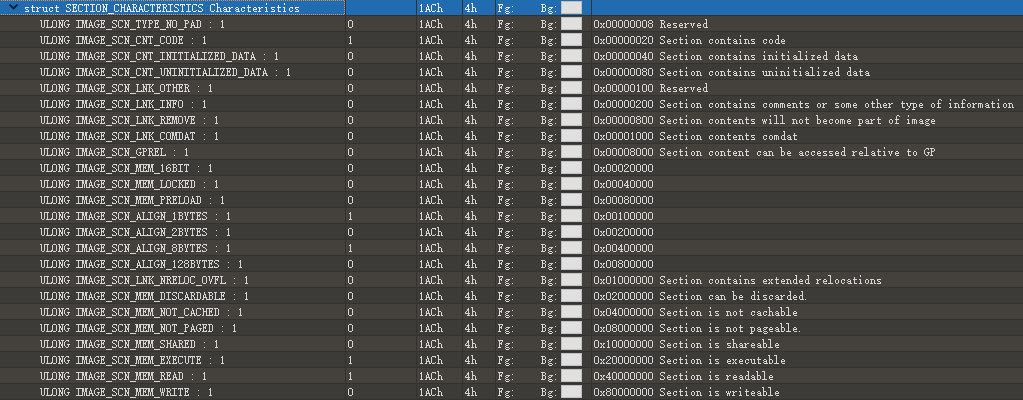
**PointerToRelocations**：在obj文件中使用，重定位的偏移。

**PointerToLinenumbers**：行号表的偏移(供调试使用地)。

**NumberOfRelocations**：在obj文件中使用，重定位项数目。

**NumberOfLinenumbers**：行号表中行号的数目。

**Characteristics**：



节的属性，每一位存储一项，包括该节是否包含代码、是否包含数据、是否可读、是否可写、是否可执行等。

4、具体的节

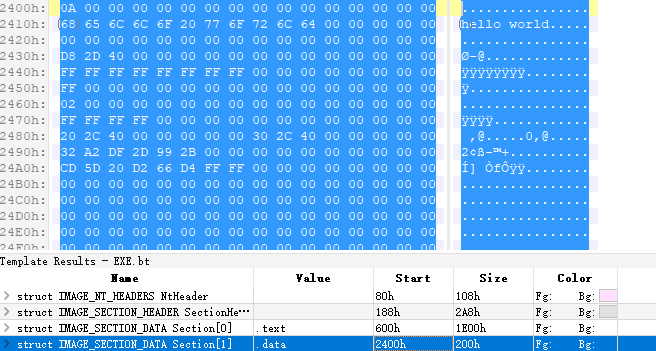
**.idata节**

导入段。包含程序需要的所有DLL文件信息。

**.data节**  
数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

根据示例程序中

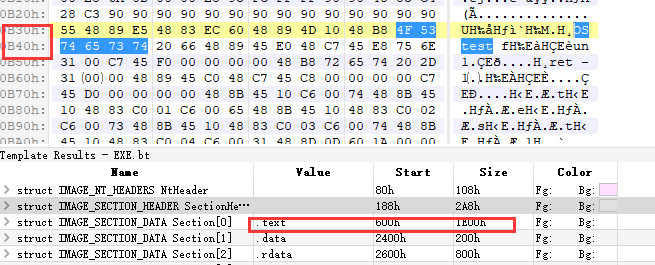
通过010editor，我们可以在.data段找到如下信息



**.text/CODE节**

代码段（text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域属于只读。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。需要注意的是，borland这里叫做code，而不是text 。





**.rdata节**

.rdata节一般最少两种用途。通常.data节的开头是程序的Debug目录，该目录仅存在于.EXE文件当中。Debug目录是一个 IMAGE\_DEBUG\_DIRECTORY 结构体的数组，这些结构体存储了文件各种不同的Debug信息。

同时.rdata节有一部分是用来描述字符串信息。如果程序的DEF文件中存在特别指定的DESCRIPTION entry，那么这个DESCRIPTION entry就会出现在.rdata节中。

**.CRT节**

.CRT节是另一个 Microsoft C/C++ run-time libraries的初始化数据段。

**.tls节**

.tls=thread local storage，当用户使用compiler directive \_ \_declspec线程时，定义的数据会出现在当前段。当处理.tls节时，内存管理器会建立一个页表来保证进程切换线程时，一组新的物理存储页会映射到.tls节的地址空间。

**四. 关键技术和难点分析**

关键点：RVA地址

RVA地址（Relative Virtual Address）是相对虚拟地址的缩写，PE文件中各种数据结构中涉及地址的字段大部分是以RVA地址表示的。RVA表示一个PE文件被装载到内存后，某个数据位置相对于文件头的偏移量。例如，Windows装载器将一个PE文件装入00400000h处的内存中，某个数据的RVA为1000h，则该数据在内存中的实际地址为00400000h+1000h=00401000h。

任何RVA都要经过到文件偏移的换算才能用于定位和访问文件中的数据，换算需要三步：

1. 扫描节表，根据每个IMAGE\_SECTION\_HEADER结构中的VirtualAddress字段（描述了该节的起始地址）和SizeOfRawData字段（描述了该节的实际大小）可以知道每个节的起始和结束地址，依次判断所要查找的目标RVA是否在某个节内；

2. 确定了目标所在的节后，用目标RVA减去目标所在的节的VirtualAddress，得到目标RVA相对于节的起始地址的偏移量；

3. 用该偏移量加上目标所在的节的PointerToRawData（描述了节在文件中的偏移地址）,得到目标RVA在文件中的偏移地址。

**五. 运行和测试过程（或结论或总结）**

PE文件是Windows操作系统上的程序文件，可移植的可执行文件。一个操作系统的可执行文件在很多方面是这个操作系统的一面镜子。通过对PE文件结构，内部原理，作用的研究，让我们了解了Windows操作系统下，可执行文件是如何加载的，同时对操作系统的理解也更加深入了。同时也对软件安全，软件解密有了一定的了解。通过大作业的学习，也加深了对《操作系统原理》这门课程的理解。

**六. 参考网址**

<https://baike.baidu.com/item/pe%E6%96%87%E4%BB%B6/6488140?fr=aladdin>

<https://blog.csdn.net/freeking101/article/details/102752048?utm_medium=distribute.pc_relevant.none-task-blog-baidujs-2>

<https://blog.csdn.net/adam001521/article/details/84658708>

<https://www.cnblogs.com/guanlaiy/archive/2012/04/28/2474504.html>

<https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10)?redirectedfrom=MSDN>