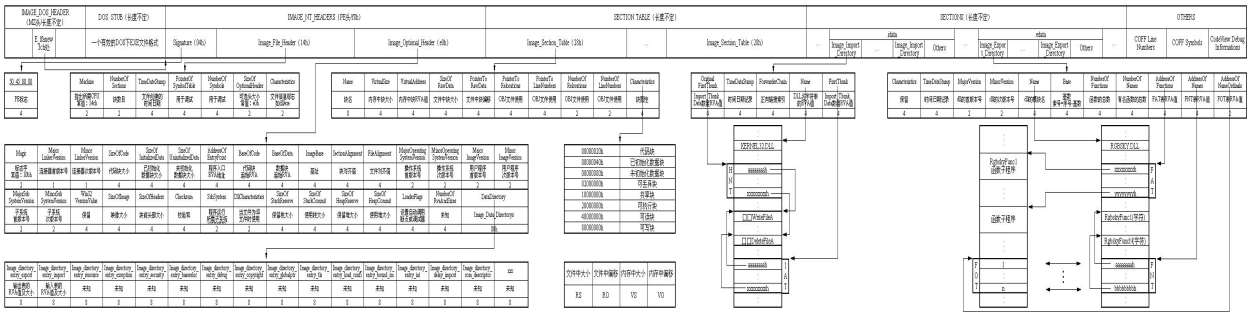
PE文件格式示意图



PE文件格式示意图

PE 文件格式启发式学习

问：1.1
我知道程序中最重要的段是text段，请告诉我text 段在哪？
答：1.1
text 段在文件偏移0x400处，大小0x200字节，该区可运行，可读取，包含代码。
该区在内存中RVA 0x1000处，大小0x1000.
问：1.2
我用urtraedit 打开hello.exe 看了，在0x400处-0x600处，大部分都是0，为什么这样呢。
答：1.2
pe 格式大部分文件都是这样，这是对齐所要求的，文件对齐为0x200, 内存对齐为0x1000
你可以在NT_Option_Header 的Section_Alignment, File_Alignment 域中看到这两个数据。

```
//
// Optional header format.
//
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    //
    // NT additional fields.
    //
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
```

注：以上PE示意图以及文章均来自“看雪”，转载请注明来源（Inking26@gmail.com）

```
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;z
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

问：1.3
慢点，别一下子贴那么多东西，我还没有找到 _IMAGE_OPTIONAL_HEADER 的位置呢，
告诉我怎样找
答：1.3
贴上那个_IMAGE_OPTIONAL_HEADER结构好说话，它的位置紧跟在 MAGE_FILE_HEADER 之后
告诉你个小技巧，那个Magic对NT x86来讲总是010B,在头文件找到那个010b,就是
IMAGE_OPTIONAL_HEADER32 结构的地址。
问：1.4
问题越来越多了。
_IMAGE_OPTIONAL_HEADER 还没有说清呢，又出来一个IMAGE_FILE_HEADER。先不管
IMAGE_FILE_HEADER
先按你的小技巧，在头部找到010b, 因为是little endial, 在ultraedit 中要找0b 01.
好，找到了，离那个 50 45 00 00 (ascii PE)相距不远，在偏移D8处，按你所说SectionAlignment和
FileAlignment 应该在结构第9个，第10个DWORD 处。
好，找到了，在f8处有00001000， FC处为00 00 02 00 （我已经考虑了endian,以后不用提醒了）。
答：1.4
呀，进步不小吗？这样一下子你就把IMAGE_OPTIONAL_HEADER32 中所有的东西都找出来了。
问：1.5
是的，我可以把Optional header中所有东西都找出来，但我现在除了刚才介绍的第9个DWORD为内存对齐
大小，第10个DWORD为文件对齐大小，其它我都不知道是干什么的？
答：1.5
别着急，其实还是很容易理解的，从字面意义就能猜大概。不过我们现在还不是通读Optional header的时
候，还是拣我们最关心的问题插手吧。
问：1.6
还是回到text 段上来吧，刚才你对text段大小，位置，属性分析的头头是到
你是从那看出来的？
答：1.6
是从section header 中看出来的，每一个section, 都有一个section header 描述其位置，大小，属性。
section header 的结构是这样定义的

```
#define IMAGE_SIZEOF_SHORT_NAME              8
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
```

```
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

问：1.7
呦，慢点，怎么又往外甩结构，我很菜！ 哦，不太多，还行吧。
不过你还是告诉我具体位置在哪吧，我好拿结构和数据对对号。
答：1.7
好，正是这种学习方法。你一定能学会的。
节表头是一个数组，它把所有节的位置，长度，属性放在了一起
紧跟在option header 之后，所以你从文件头部往下找就可以了。
看到IMAGE_SECTION_HEADER结构的第一个成员了吗，它是
BYTE   Name【8】
这是节名称，你要找的text 段名字就是 .text, 你看ultraedit
ascii 码区离文件开始不远的地方，有一个.text, 对应的二进制
数据是2E 74 65 78 74, 这就是text 端IMAGE_SECTION_HEADER处

问：1.8
原来玄机在这里呀。我试试看。哦，看见了，在1B8处。 前8个
字节是节名称。后面的00 00 00 28 到底是物理地址还是虚拟大小，
（偷偷的，虚拟大小，表示内存中只有0x28个字节有效，其它全是0），在后面00 00 10 00 是虚拟相对地址 俗称RVA, 就是在内存中相对与起始地址的偏移。再后面00 00 02 00 为SizeOfRawData， 就是文件中大小，再后面 00 00 04 00 是
PointerToRawData，是文件的偏移 后面有三个DWORD 全是0，他们
是重定位信息和行号，很好，EXE文件可以不用管这些。最后一个
60 00 00 20 代表属性可读，可写，是代码。好，我终于理解你的第一句话了。
不解释一下，我怎么能一下子听的懂呢！ 谢谢你。
那么我又有问题了。那程序针真是搜索这个.text字符串找到Text 节表头吗？
答：1.8
不是。前面说过，节表头紧随Optional header 之后。

问：1.9
Optional header 结构变量太多，我数了一下都没数清，到底占多少个字节呢？
答：1.9
正等着你这一问呢？是啊，数都数不清，纵是现在记住了将来也容易忘。
估计微软也想到了这一点，他把OPTION header 的大小放到了 _IMAGE_FILE_HEADER 的一个变量中，下面是_IMAGE_FILE_HEADER 的定义

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

SizeOfOptionalHeader 一般总是0xE0

问：1.10
我今天已经学了不少东西了，看样子后面还很多的样子。再问最后一个问题。
FILE_HEADER 在文件什么位置呢。
答：1.10
这个简单，就在PE标识符后面。看到了吗，在C0处，ascii 是PE. 二进制是50 45 00 00
代学生：
哦，看到了，今天10个问题已经满了，我还想学，可是有点累了。。。
代老师：
今天就到这里吧，好好休息一下。

接问题1.10后的第11个问题
问：2.11
干脆就把位置问题先问到底吧。PE标识符 54 45 00 00 总是在文件偏移00c0 处吗。
答：2.11
基本上可以这么说，主要是因为前面的部分是dos 头部和dos 体
dos 头部 IMAGE_DOS_HEADER 的结构我就不贴了，因为dos 已经离我们远去，它
已经失掉了意义，dos 体也几乎是固定不变的了。这部分的作用是当你拿这个
PE程序到dos 系统上运行时，dos 执行会在控制台上打印一行提示信息，
"this program cannot be run in DOS mode" 然后停在哪。总比你一运行，DOS
就hang 机强多了。如果拿PE代码在DOS 下直接执行，不用说那肯定hang 机。
微软就是怕这个事情发生才用了这么一个措施。
现在你只需要记住一件事，文件头两个字母是MZ标记，在3c偏移地址，00 00 00 c0
指的是NT header 的文件偏移，如果这个偏移处的标识正好是PE. 可以肯定，这个文件
就是PE 文件了。 如果在3c偏移地址处存其它DWORD 地址，那就到所指定的地址去找
如果该处正好ASCII "PE",此处就是NT的header 。
问：2.12
怎么有这么多header, 能否概要总结一下：
答：2.12
好的。在文件开头部分是 _IMAGE_DOS_HEADER ， 小名MZ header, 我们已经不用关心它了。
只要关心地址偏移0x3c 处，该处存有 _IMAGE_NT_HEADER 的偏移。在dos header 和
NT header 之间是dos 体，我们也不用关心它了。
_IMAGE_NT_HEADER 到底是什么样呢？它实际是PE00标识+
NT_FILE_HEADER+NT_OPTION_HEADER
以下是它的结构声明。

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;        //这里的标记是 PE00
    IMAGE_FILE_HEADER FileHeader;    //NT header 包含FILE header 和Option header
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

问： 2.13
这样对header有了一个总体认识，它占据着文件开始部分。反正它是死的，而且每个文件只有
一个，有上面各个header 的结构定义，无非是存储这一些数据，指针。估计详细分析一下，
它也跑不了了。我们还是抓主要的，主要的分析清了，可能顺便就把头中的相关结构变量分析了。
还是回到节表上来。上次已经找到了节表头，前面说是在OptionalHeader下面，现在也可以
说是在NT header下面。其中以.text 居首，根据节表头结构，从.text 偏移一个节表结构，
我们看到了第二个ascii 字符 “.rdata", 不远的地方还要一个”.data", 正好也偏移一个节表头结构“，
还有一个".rsrc",再往后就是全0了，那么这是否是说，这个结构数组含有4个结构元素呢？
答： 2.13
正是如此，在_IMAGE_FILE_HEADER 中有一项定义了该数值

```
    WORD    Machine;      //x86 的machine代码是 01 4c (hello.exe 中00c4处）
```

注：以上PE示意图以及文章均来自“看雪”，转载请注明来源（lnking26@gmail.com）

WORD NumberOfSections; // hello.exe 中 是 00 04
与你数的完全一致。对照一下问题1.9的_IMAGE_FILE_HEADER 和 hello.exe 的，你会很容易辨别的。
问： 2.14
我对照过了，知道了它在文件中的位置。干脆把NT FILE Header结构中的其他数据也分析一下吧。反正也不多。
答： 2.14
好，我再把该结构抄过来：

```
typedef struct _IMAGE_FILE_HEADER {
  WORD    Machine;     //答2.12已经说了，x86 总是01 4c
  WORD    NumberOfSections;   //hello.exe 是00 04
  DWORD   TimeDateStamp;   //时戳。表示你的文件是何时生成的。不过这个DWORD是用秒数表示的。
  DWORD   PointerToSymbolTable;  //调试信息，hello 中为全0
  DWORD   NumberOfSymbols;   //调试信息，hello 中为全0
  WORD    SizeOfOptionalHeader;  //答1.9已经说了，OptionalHeader 大小总是0xe0
  WORD    Characteristics;    // hello.exe 是010F, 看标志有5个bit 是1，那就是说5个属性为真了。
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

NT FileHeader其它项都好理解，没有什么关键的东西，只有这个属性稍微麻烦一点，最多也不超过16个属性。
顺便问一下，你在ultraedit 中看着这个FILE_HEADER 吗？
代学生：哦哦，看着呢，我的光标就停在这个010F 标志处呢。
代老师：
好，继续。16个属性一下都说出来也太多，先学习hello.exe 的这5个吧。
bit0: 文件不包含重定位信息。
bit1: 文件可以运行。
bit2: 文件不包含行号信息
bit3: 文件不包含符号信息。
bit8: 32bit 机器上运行。
怎样，这你个属性很好理解吧，可执行文件不需要重定位，行号及符号信息。在32bit机器上运行。
代学生：
说了半天原来没有一个关键性东西，我还以为有多神秘呢！
代老师： 是的，搞懂了它有时也觉得失望，其实，懂了也就这么简单。
问：2.15
再问一个关键性问题，看起来有点菜。我以为，有text段，有data段就可以了，
那么.rdata, .rsrc 是干什么用的呢
答：2.15
这个问题确实很关键，这正是PE 文件与以往文件的差别所在。
其实只有text段，data段，pe文件是不可以运行的，因为PE文件的运行总是要调用
系统文件，而系统文件都是以DLL 文件格式存在的，所以你必须要在文件中有动态链接
信息。
问：2.16
太复杂了，什么是动态连接信息，什么是动态连接库，听说过但没有真正理解。
答：2.16
动态连接是多进程操作系统引进的一个概念。在DOS时代，单任务是没有动态连接的。
在DOS 时代，连接器总是把库文件直接连接到可执行文件中。叫静态连接。这种做法
在单任务时是可以接受的，无非是每个连接的文件都包含一个库文件，造成磁盘空间
的一点浪费。 但静态连接在多任务时代不可以接受。例如每个进程都会调用 kernel32.Dll
如果采用静态连接，造成磁盘空间浪费不说，假若系统有20个进程，系统中就将会
有20份kernel32.dll, 内存的浪费将是不可容忍的，动态连接的概念就是保证系统中
只有一份DLL，同时各个进程又都能够很好的运行。好在动态连接是加载器的功能，
我们程序不用刻意去做什么，所以用起来也不是太复杂。
问：2.17
哦！是这样，我原来以为链接程序都把事情处理好了，原来还没有，多进程中还要由

加载器进行动态连接。那我们怎样使用动态连接呢？
答：2.17
当我们用汇编语言或C C++或者其它语言开发是，生成的PE文件对系统库的调用都是动态连接。我们并没有做什么。
当你想调用自己生成的DLL(第三方DLL),可以采用隐含动态连接或者显示动态连接来加载DLL. 听起来很炫用起来很简单，隐含链接就跟使用系统dll 一样，你只要在文件中包含第三方头文件（好引用它的函数啊。）在连接选项里设置第三方的lib,dll位置链接程序就帮你搞定了。
显示动态连接是在你的程序里用loadlibrary 加载DLL, 用getprocess 获取DLL中函数地址，然后用函数指针调用第三方函数。
问：2.18
哦. 听你的意思看来使用DLL 也是很简单的。系统DLL使用我们不用管，怎样使用第三方DLL以后再说吧，我现在的重点是想搞明白PE 的文件格式。 那么既然它调用了kernel32.Dll 中的函数。而这个函数的地址连接程序不知道，只能由加载器在运行时动态加载。那么，加载器是怎么知道要加载那个DLL, 要执行DLL中的那个程序呢？
答： 2.18
这个问题问到点子上去了。搞清了这个问题，PE 格式就可以说入门了。
我们还是结合hello.exe 实例说吧。
有HIEW 软件吗，准备一下。
好：
1. 用hiew 打开hello.exe
2. 按F4, 选Decode.
3. 按F5, 敲入偏移400（我们上面分析过，text 段在400)
干脆我把代码贴过来吧， 双// 是我注释的。

```
00000400: 6A00                  push       000
00000402: 6800304000              push       000403000 ;" @0 "
00000407: 680C304000              push       00040300C ;" @0♀"
0000040C: 6A00                  push       000
0000040E: E809000000              call       00000041C  //hiew 分析出，这是MessageBoxA
00000413: 90                    nop
00000414: 90                    nop
00000415: 6A00                  push       000
00000417: E806000000              call       000000422  //hiew 分析出，这是ExitProcess
0000041C: FF2508204000            jmp        d,[00402008]
00000422: FF2500204000            jmp        d,[00402000]
; ------------------------------------------------------------------------
```

我们分析 call Messagebox 吧。
40e 处： call 41c,
41c 处： jmp ds:[00402008]
00402008.
这是虚拟内存地址。我们用hiew 找到它。具体操作如下：
1. 按F4, 选hex. 要看数据，选hex 合适
2. 按F5, 敲入偏移600.
在hiew 中看到了下一行。
.00402000: 76 20 00 00-00 00 00 00-5C 20 00 00-00 00 00 00
问：2.19
喂，慢点，打扰一下，我这个人就喜欢刨根问底。你怎么知道要敲入600呢？
答：2.19
是这样，调用DLL采用动态连接，动态连接信息是放在.rdata 段，叫只读数据段。
你刚才不是问.rdata, .rsrc 是干什么的吗？ 我现在才讲到了.rdata 段
要想知道.rdata 在哪，你的问.rdata 的节表头。我们已经讲过所有节表头组成一个数组，紧跟在NT Header（总header)之后。.rdata 是第二项节表头，其名字是ascii码很明显的。 我把它copy 到这啦。

00001e0: 2e72 6461 7461 0000 9200 0000 0020 0000 .rdata..... ..
00001f0: 0002 0000 0006 0000 0000 0000 0000 0000 ....
0000200: 0000 0000 4000 0040 2e64 6174 6100 0000 ....@..@.data...
这里根据 答1.6中 section header 的结构，可以知道.rdata 的如下信息
名字：.rdata, 虚拟大小:0x92, RVA:0x2000, 文件中大小0x200, 文件偏移0x600
属性，40000040，看来有两个属性有效。bit6 -- 包含初始化数据。bit30 -- 可读属性。
问：2.20
再回到刚才的问题吧。本来想call Messagebox, 由于Messagebox 是user32.dll 中的函数。
汇编器，链接器不知道Messagebox 地址在哪里，PE 文件里翻译的是代码要call 41c,
而41c 处向 402008 地址单元处存储的地址跳跃。由于402008 从文件看来存的是0x205c
看样子要跳到 0x205c 处执行了。
答：2.20
你前面的分析都是对的，但最后一句结论却错了，如果程序真要跳到0x205c 处，肯定会被系统
判你一个操作地址非法，强制你关闭程序。其实这个地方的205c,要被替换成另外的一个数据
这就是Messagebox 在内存的真正地址。谁替换的？ 当然就是loader 了，加载器知道你想要
call MessageBox, 就帮你偷偷的把这个链条给拧上了。
代学生： 呀！讲的正精彩呢，怎么又下课了...
代老师： 好了，我们下一课再接着讲。

接问题2.20后的第21个问题
问：3.21
上回说到loader 为了使程序正常运行，偷偷的把.rdata 节中的某些数据给改了，能再讲清楚一些吗？
答：3.21
加载器未改之前，我们用hiew 看文件有如下数据
.00402000: 76 20 00 00-00 00 00 00-5C 20 00 00-00 00 00 00
加载器改动之后，我们可以用ollydbg 看一下，这是加载器完成修改后的结果
具体操作是用ollydbg 加载hello.exe,点击数据窗口，按ctrl-G,输入地址402000
然后看到如下数据
00402000 >DA CD 81 7C 00 00 00 00 8A 05 D5 77 00 00 00 00 谕|....?諳....
这就说明，加载器把2076 改成了7c81cdda, 把205c 改成了 77d5d58a
问：3.22
呦，还藏着这等玄机呢！ 不过，加载器也是一段程序，它总不能乱改吧？咱就以后面
的205c 为例，它凭什么要把205c 改成77D5058A呢？这个可是我们要call 的MessageBox的地址呢。
答：3.22
连接器知道你要CALL MessageBox, 但MessageBox 是USer32.dll 的函数，连接器不知道
MessageBox 在哪里，只有操作系统才知道USer32.dll 在哪里。loader 也是通过调用系统
函数才知道的。由于link 的时候文件还没有运行，所以它不知道MessagBox 的具体地址。
好，这个问题讲清了，连接器不知道DLL及其函数的具体地址。
但连接器也会尽其所能，告诉加载器一些信息，他对加载器说，运行前你要帮我把USer32.dll
的MessageBox 地址给填好！拜托了！
用计算机来描述是这样的。
他往402008处填了一个205c, 这个205c 是什么，是一个RVA, 它指向一个数据结构，该结构
实现linker 向 loader 的信息传递， loader 来完成linker 未完成的使命。
如果你要想看看linker 向 loader 说了什么，咱还得先看看这个数据结构的地址。
问：3.23
就已hello.exe 为例吧，看看205c 怎么找到那个数据结构地址的。
答：3.23
loader 拿到了数据205c, 知道这是一个RVA, 加上影像基地址0x400000,或者说叫module 地址吧。
得到了一个虚地址0x40205c,你从ollydbg 的数据窗口中看0x40205c 地址。是如下内容：
0040205C 9D 01 4D 65 73 73 61 67 65 42 6F 78 41 00 75 73 ?MessageBoxA.us

```
0040206C  65 72 33 32 2E 64 6C 6C 00 00 80 00 45 78 69 74  er32.dll..€.Exit
0040207C  50 72 6F 63 65 73 73 00 6B 65 72 6E 65 6C 33 32  Process.kernel32
0040208C  2E 64 6C 6C 00 00 00 00 00 00 00 00 00 00 00 00  .dll............
```

它指向一个WORD 数据019D, 后跟MessageBoxA 0字终结字符串，其后再跟USER32.dll。
这个结构有一个学名，叫 IMPORT_BY_NAME,如下定义：

```
//
// Import Format
//
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;
    BYTE    Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Hint, 就是那个019d 了，BYTE Name[1], 这个变量定义看起来很奇怪，是吗？
代学生： 是的，我很少见到这种定义。
通常我们会定义 char buffer[256], BYTE data[8]; 等类型。
BYTE Name[1], 只包含一个元素的数组，它也装不下后面的"MessageBoxA"字符串啊。
代老师：这种定义是一种指针的变通用法。
如果你真要定义成数组来包含后面的字符串，你定义成多大呢？定义成100，短字符串浪费，
长字符串可能就真能碰到一个101个字符的名字，你定义的还是占不下。
所以说，这个Name[1], 不是要你往里面装东西的，C 语言里，你可以借助这个Name 变量访问到它对应的地址。
这种用法通常是很少用的。因为它毛病很多，例如结构后面不能再定义其它变量了，必须是最后一个，定义了
数组又不用它装东西，也不符合数组的初衷. 所以你只有明白这个道理就可以了。

代学生：既然它那么不好用，为什么还那样定义呢。
代老师：还是那句话，是变通。
你看，它简洁，它完成了使命。否则你就要把结构变一变，例如按常规估计应该是这样子。
WORD Hint; BYTE *pName; 然后你要求微软说，Hint 后面不要跟字符串，要跟一个地址。这样C语言好写。
好比说大部分人沿着盘山路往山上走，也有人愿意盘着荆棘往山上爬，后者绕了近路，但风险也大。

代学生：讲了这么多，其实我看一个word 后面跟着一个0字终结符字符串，还是很好理解的吗。
代老师：C 语言以其简洁，高效，使我们受益良多。但在某些特殊的情况下，它也会力不从心。有时刻当你看着一堆堆
结构套结构，一堆堆宏套宏令你头晕时，而看看它最终的list 表或二进制输出反而能令你豁然开朗。
哦，有点扯远了。 我还是最喜欢C的。

问: 3.24
找到了这个 IMPORT BY Name 结构， loader 是怎样根据这些信息修改地址的呢？
答：3.24
loader 在加载时，是要先收集信息的，不过这部分还没有讲，收集好后，以MessageBoxA为例
loader询问系统，USER32.dll 加载了吗？没有我要先加载它啦。哦，加载了，告诉我MessageBoxA 函数地址是多少？系统返回一个地址 77d5d58a， loader 就把这个地址覆盖了原来存储的 0x205c 。
这样就把MessageBoxA 的内存地址给拧上了。
注意了，这个 77d5d58a 是我机器上的MessageBoxA 内存地址， 到了你的机器上，它就变成别的啦。
这正是DLL 存在的妙处。
意思表达很明确，不是吗？
问：3.25
对，它肯定能表达明确。不过目前我还有很多问题要问，别嫌麻烦呦。我问得可是很细致的。
答：3.25
难得你精神可嘉，咱们也是互相促进的。能走到这里，也可以说是渐入佳境了。

注：以上PE示意图以及文章均来自“看雪”，转载请注明来源（lnking26@gmail.com）

问：3.26
弱弱得问一下，3.23 提到的那个module 地址0x400000, 是固定死的吗？
答：3.26
module 加载地址，是loader 将应用程序加载到内存时的起始地址，loader 是从Option header 结构中的Image Base 项得到这个地址的，由于exe 文件不存在地址冲突问题，所以loader 总能把程序加载到Option header 中Image Base 指定的位置，这个位置通常都是0x400000.
问：3.27
刚才没顾上问，MessageBoxA 前面那个WORD 019d， 就是hint, 是干什么的。
答：3.27
那个东东是loader 向系统询问函数地址的另外一种方式，它可以向系统询问user32.dll 第019d 个导出函数是多少？ 系统回答 77d5d58a 。 这种导入方式叫ordinal. DLL 中有名称的导出函数都可以用名称访问，也可以用ordinal 访问，而有的导出函数没有名，只能用ordinal方式导入。
问：3.28
刚才你是用ollydbg 来讲解的，我们不是一直用ultraedit 打开这个文件，直接分析它的二进制结构的吗。能用ultraedit 再讲一讲linker 向 loader 说了什么吗？
答：3.28
这主要是因为动态链接的数据是一个RVA, 所以用ollydbg 讲的方便。同时由于在ollydbg中已经完成了动态连接，跟utraedit 静态分析正好有个对照。现在我们再来看从ultraedit 中怎样找到 ？ ？ ？ 结构loader 拿到了数据205c, 哦， 不对，是我们拿到了205A,知道这是一个RVA.需要把它转成文件偏移好看看它到底对我们说了什么。
从RVA 到 OFFSET, 我们好像还没有讲呢，就在这里补上吧。
从RVA 到 OFFSET 没有一个简单的公式，唯一的办法就是查表。
查什么表，查section header 表。
已hello.exe 为例，我们查表，看到205c 落入.rdata 节，该节的虚拟地址（就是内存地址）0x2000 对应文件偏移0600，那么205c, 则对应文件偏移的065c. 用ultraedit 观看，如下图示。跟ollydbg看到的内容一样。

```
0000650: 0000 0000 5c20 0000 0000 0000 9d01 4d65  ....\ ........Me
0000660: 7373 6167 6542 6f78 4100 7573 6572 3332  ssageBoxA.user32
0000670: 2e64 6c6c 0000 8000 4578 6974 5072 6f63  .dll....ExitProc
0000680: 6573 7300 6b65 726e 656c 3332 2e64 6c6c  ess.kernel32.dll
```

代学生：顺便问一下，那些查虚拟地址到文件偏移转换的工具是这样查的吗？
代老师：是的。
问3.29
.rdata 节中大部分数据都明白了，但从610-65d 那一段数据是干什么的？
答3.29
这段数据，当然也是动态加载用的。
前面讲的link 与 loader 对话，确实如上所说，但那只是问题的一半，还有一半就是，当loader 把程序加载到内存，它要修改数据，它怎样找到修改数据的地址，也就是说，那个存储着RVA 205c 的地址00402008 loader 是怎样得到的？还有，它怎么知道MessageBoxA在user32.dll 里面？
问3.30
平时都是我问，现在忽然被反问。翻翻前面讲的 。。。
啊！ 是2.18 时提出来的，程序要call 41c, 41c 要向402008 地址所装的内容处跳。
哦，这是我们的读法。loader 是不会这么读的，loader 是死的，loader 是一段程序，它只会干机械的事。它怎么找到402008的，它怎么知道MessageBoxA在user32.dll 里面？ 还是听你讲吧。
答3.30
讲清这个问题，我们还要再看option header. 坚持住，动态加载导入部分也就差这一点点了。
在1.2中提到option header 的数据结构，在他的底部有一个成员。

```
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
```

看着眼晕，还不如写简单点,它的数组就是16个元素

```
IMAGE_DATA_DIRECTORY DataDirectory[16];
```

前面那个结构叫数据目录，如下定义
//

```
// Directory format.
//
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   RelativeVirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

挺简单的，其实就是8个字节。 16个目录吗，就是16*8 = 108 字节。占8整行
别担心，大部分没有用，微软在这里搞捉迷藏。我们只关心几个就行了。
这里首先介绍的一个叫导入表(import table)。

```
0000130:        0000 0000 0000 0000  ................
0000140: 1020 0000 3c00 0000 0040 0000 a003 0000  . ..<....@......
0000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000160: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000170: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000180: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000190: 0000 0000 0000 0000 0020 0000 1000 0000  ......... ......
00001a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001b0: 0000 0000 0000 0000 2e74 6578 7400 0000  .........text...
```

为什么还留下那个.text, 一看就知道，.text 代表的是section header table 的开始地址
其上的8个整行就是16 个目录了。真要让你像计算机一样从上到下数偏移，我们还真不行，找ascii 字，我们还在行。
第一个目录叫导出表，这里为空
第二个目录叫导入表，这里虚拟地址是0x2010, 大小是3c.
2010 RVA 对应的文件偏移是610 （算法我想你已经掌握了,看3.28）
第三个目录叫资源表。RVA=0x4000, size=0x3a0 (暂时先不讨论）
第十三个目录叫IAT 表，英文全称为：Import Address Table. RVA=0x2000,size=0x10
其它的目录都是空的，没用，不理它们。
代老师：哦，是不是讲的有点多了？
代学生： 还行，反正讲得挺清楚的。想不到这个小小的hello.exe 还有这么多事情。
不过再看看hello.exe文件，大部分内容都讲了，我想也不会有太多东西了。
代老师：正是，坚持一下就是胜利。

接问题3.30后的第31个问题
问4.31：
上回谈到目录项，有两个目录项要关心。
1. 导入表：RVA=0x2010, size=0x3c
2. 导入地址表（俗称IAT)： RVA=0x2000,size=0x10
具体什么用途呢？
答4.31：
先看IAT表，loader要做的工作是把RVA=0x2000,size=0x10这么大范围的DWORD都要重新定址。如果是00 00 00 00 就不用了。
对hello.exe 而言文件偏移是：（今后RVA 和 OFFSET 的转换我就不再提了）

```
0000600: 7620 0000 0000 0000 5c20 0000 0000 0000  v ......\ ......
```

再看导入表RVA=0x2010, offset=610. size=0x3c,内容如下：

```
0000610: 5420 0000 0000 0000 0000 0000 6a20 0000  T .........j ..
0000620: 0820 0000 4c20 0000 0000 0000 0000 0000  . ..L ..........
```

```
0000630: 8420 0000 0020 0000 0000 0000 0000 0000  . ... ..........
0000640: 0000 0000 0000 0000 0000 0000
```

这里是一个结构数组。该结构叫导入表的描述符

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;          // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;       // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    };
    DWORD   TimeDateStamp;                // 0 if not bound,
    DWORD   ForwarderChain;
    DWORD   Name;
    DWORD   FirstThunk;                   // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
```

每个结构包含5个DWORD 成员,最后一个结构，成员全是0
结构中第二个TimeDateStamp，第三个成员ForwarderChain不用关心，它们总是0
name 变量：
第一个结构中为206A, 对应文件偏移66A, "user32.dll"
第二个结构中为2084, 对应文件偏移684 "kernel32.dll"
FirstThunk 地址：
第一个结构中为2008, 对应文件偏移608, //我们看到这个就是IAT 表的内容了
第二个结构中为2000, 对应文件偏移600，//我们看到这个就是IAT 表的内容了
OriginalFirstThunk 地址：
第一个结构中为2054, 对应文件偏移654, //654与608 存储的内容是一样的
第二个结构中为204c, 对应文件偏移64c，//64c与600 存储的内容是一样的
怪不得一个叫first thunk, 一个叫OriginalFirstThunk，原来所指地址处包含的内容是一样的.

```
0000640: 0000 0000 0000 0000 0000 0000 7620 0000  ............v ..
0000650: 0000 0000 5c20 0000 0000 0000 9d01 4d65  ....\ ........Me
0000660: 7373 6167 6542 6f78 4100 7573 6572 3332  ssageBoxA.user32
0000670: 2e64 6c6c 0000 8000 4578 6974 5072 6f63  .dll....ExitProc
0000680: 6573 7300 6b65 726e 656c 3332 2e64 6c6c  ess.kernel32.dll
```

导入描述符有2个，说明它有两个DLL 要导入，从Name 能看出来，
第一个叫"user32.dll", 第二个叫"Kernel32.dll"
描述符的 FirstThunk, 指向一个导入函数地址数组，就叫thunk data数组吧，该数组最后一项为00000000
描述符的 OriginalFirstThunk ，虽然与FirstThunk所指地址不同，但该地址所包含的内容却一样。
以第一个结构为例：
这个DWORD 数组为5c20 0000 0000 0000 ，这个数组只有两项，去掉末尾标志项，只有一项，说明只有一个函数要导入。怎么导入，在这个地方，我们以0x205c 为例， 把3.23中叙述过的导入过程补充完整。

1. loader 从目录第二项得到 Import Table(导入表）
2. 导入表是一个导入描述符结构数组，以第一个结构为例
   从名字项中它知道，这个动态库的名字叫 "user32.dll"
   。
3. first thunk 指向thunnk data数组，是一个以全0结尾的DWORD 数组，
   非全0 的元素或者是一个RVA(最高位为0），指向一个Import_BY_NAME结构，

或者是一个Ordinal(序号，（其最高位为1，使用时把最高位去掉就成序号啦）。
first thunk 指针属于IAT 表的地址范围。
hello.exe 中import第一个描述符 first thunk表偏移是608，608处存205c（RVA)
对应Import_BY_NAME MessageBoxA, 下一个DWORD=0数组就结束了
说明user32.dll 只导入了一个函数。

从上面分析可以看出。
目录项中的导入地址表：指向一个大的IAT。
目录项中的导入表： 指向一个导入描述符数组。每一个描述符，描述一个DLL。 而FirstThunk 指向那个大的IAT中的一个部分。我们称它为thunk data数组，数组最后一个元素DWORD为全0.
这样的一个设计结构，导入表和导入地址表的信息是有冗余的。我们不管它，知道就行了。

问：4.32
既然有了 FirstThunk， 还要 OriginalFirstThunk 干什么？ 它们指向的IAT 内容都是一样的。
答：4.32
FirstThunk 所指向的IAT, 会在加载时被loader 修改为真实的函数地址。而OriginalFirstThunk 所指向的IAT 是不会被修改的 。其实我也认为，留着这个 OriginalFirstThunk 没有用途，可能是微软认为那个IAT已经被修改了， 万一你要再用你到拿找哇。 其实我看这是多虑了， 第一 ,IAT用完了我不会再用它。 第二,万一被我们改了回头又要用，我们可以先把没改之前的备份一下呀。
不过，这都是个人意见，人家这么定义了，我们知道就行了。

问：4.33
从运行的角度来看，好像没有问题了，.text 段依靠.rdata段的帮助，由loader 修改为可执行代码。代码执行时读取或存入数据段数据。hello 就可以运行了。
那.rsrc 段是什么呢 ？ 是资源段吗？
答：4.33
是的。 本来我是没想加这个资源段的。可是不小心给加上了。这个其实你可以不用关心它的。
解开附件hello.rar, 图标是一个笑脸，这就是那个资源段的功能。如果没有那个资源段。默认的图标是一个WINDOWS ICON.

问：4.34
要想显示需要的图标，需要我们做什么呢？
答：4.34
你只要把ICON资源文件连进去就行了，代码并不需要做任何事情。在创建MessageBox 窗口时，系统发现你文件里有一个ICON
资源项，就替你把它画出来，如果你没有包含ICON 资源，它就把自己手边的那个叫默认ICON,画到MessagBox 左上角了。

问：4.35
hello.exe 文件都分析完了，只是option header 中还有一些项没有提到，它们重要吗？
大：4.35
有些项还是很重要的，例如程序入口点，但它们都已经很好理解了。
这里，我就把option 做一个标注，此时再浏览option header 的各个项，已经是时候了。
//
// Optional header format.
//

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;      // NT HDR32 定义为010b
    BYTE    MajorLinkerVersion;    //link version 不重要
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;      //代码段大小
    DWORD   SizeOfInitializedData;  //初始化数据大小
    DWORD   SizeOfUninitializedData;  //未初始化数据大小
    DWORD   AddressOfEntryPoint;  //程序入口点：  重要
    DWORD   BaseOfCode;      //代码基址（RVA)
    DWORD   BaseOfData;      //数据基址(RVA)

    //
    // NT additional fields.
    //

    DWORD   ImageBase;      //模块基址
    DWORD   SectionAlignment;    //内存对齐调整
    DWORD   FileAlignment;    //文件对齐调整
    WORD    MajorOperatingSystemVersion;  //版本信息，不重要
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;    //模块的大小
    DWORD   SizeOfHeaders;    //header的大小
    DWORD   CheckSum;      //未使用
    WORD    Subsystem;      //02 为gui, 03 是console
    WORD    DllCharacteristics;    //dll 用
    DWORD   SizeOfStackReserve;    //系统加载堆和栈初始化信息
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;z    // 不重要
    DWORD   NumberOfRvaAndSizes;  // 总是16
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];  //16个目
录，重要
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

optionheader 重要的项是程序入口点，目录项。

optionheader 中标注不重要的就不用关心了。
下面的项重要程度中等，了解一下就可以了。
1.SizeOfHeaders 是文件头和节表大小之和。所以其后面紧跟段地址，该数值应与节表中第一节数值匹配。
2.BaseOfCode，SizeOfCode， 该数值与节表中代码节的文件大小也是一致的,有多余之嫌。

3.SizeOfInitializedData 已初始化的数据组成的块的大小.但我不知道它有什么用。

4. ".bss" 段：给loader 参考的。
加载器在虚拟内存中申请空间，但在磁盘上的文件中并不占用空间的块的尺寸。
这些块在程序启动时不需要指定初值，因此术语名就是"未初始化的数据"。未初始化的数据通常在一个名叫 .bss 的块中。
BaseOfData，已载入映像的未初始化数据（".bss"段）的相对偏移量
SizeOfUninitializedData。 申请的".bss" 空间的大小。当为0时，表示没有使用".bss"段

问：4.36
能总结一下PE 文件的重要项吗？
答：4.36
从运行的角度看，PE文件中重要的是程序入口点，节表，目录项。
代学生：感谢你为我们写了这么多东西。
代老师：大家共同提高。由于时间仓促，水平有限，有些观点未必正确，错误之处在所难免，希望海涵并欢迎指正。

全文完。