PE文件学习笔记（二）：Section Table解析

1、Section Table结构解析：

Section Table（节表）是记录PE文件中各个节的详细信息的集合，其每个成员是struct _IMAGE_SECTION_HEADER结构体，即节表是一个结构体数组来维护，属于线性结构。而节表的相对起始位置为：紧接着可选PE表。即：**DOS头 + 中间空闲及垃圾数据 + NT头（三部分：4字节签名+标准PE头20字节+可选PE头）。**

```
#define IMAGE_SIZEOF_SIGNATURE 4
#define IMAGE_SIZEOF_FILE_HEADER 20
```

用变量描述节标配偏移地址为（这是相对于文件首/ImageBase的偏移量）：

```
SectionTableStart =
_IMAGE_DOS_HEADER.e_lfanew +
IMAGE_SIZEOF_SIGNATURE +
IMAGE_SIZEOF_FILE_HEADER +
_IMAGE_FILE_HEADER.SizeOfOptionalHeader
```

一个结构体成员如下：

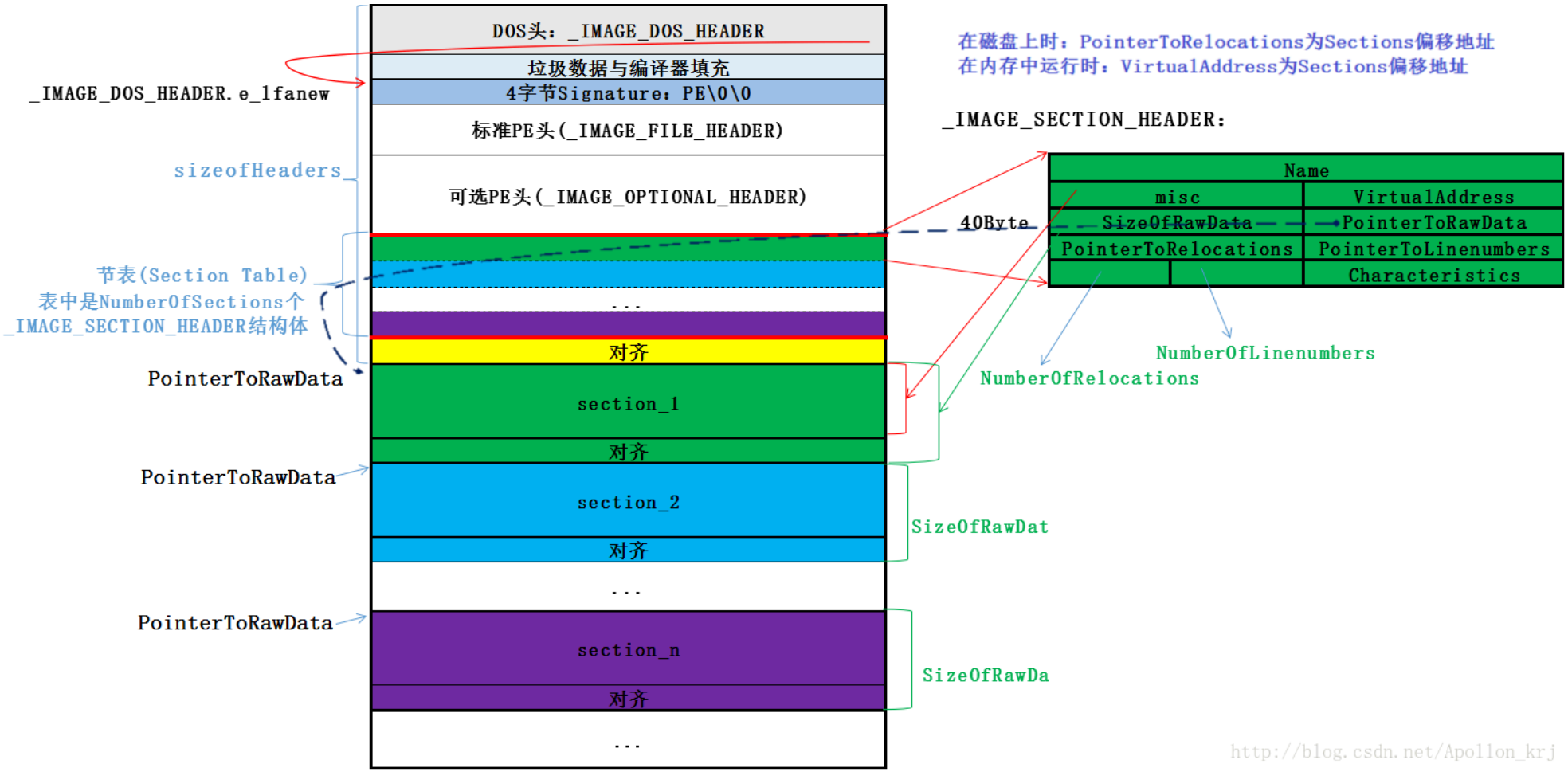
```
#define IMAGE_SIZEOF_SHORT_NAME              8
typedef struct _IMAGE_SECTION_HEADER{
    0X00 BYTE    Name[IMAGE_SIZEOF_SHORT_NAME]; //节（段）的名字.text/.data/.rdata/.cmd等。
                                                //由于长度固定8字节，所以可以没有\0结束符，因此不能用char *直接打印
    0X08 union{
            DWORD   PhysicalAddress;        //物理地址
            DWORD   VirtualSize;            //虚拟大小
    }Misc;//存储的是该节在没有对齐前的真实尺寸,可改,不一定准确(可干掉)
    0X0C DWORD   VirtualAddress;            //块的RVA，相对虚拟地址
    0X10 DWORD   SizeOfRawData;             //该节在文件对齐后的尺寸大小(FileAlignment的整数倍)
    0X14 DWORD   PointerToRawData;          //节区在文件中的偏移量
    //0X18 DWORD   PointerToRelocations;    //重定位偏移(obj中使用)
    //0X1C DWORD   PointerToLinenumbers;    //行号表偏移(调试用)
    //0X20 WORD    NumberOfRelocations;     //重定位项目数(obj中使用)
    //0X22 WORD    NumberOfLinenumbers;     //行号表中行号的数目

```

```
    0X24 DWORD   Characteristics;          //节属性(按bit位设置属性)
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
#define IMAGE_SIZEOF_SECTION_HEADER          40
```

一个结构体大小为40字节（IMAGE_SIZEOF_SECTION_HEADER），因此节表的大小为节表元素个数乘以结构体大小为：**_IMAGE_FILE_HEADER.NumberOfSections * IMAGE_SIZEOF_SECTION_HEADER。**而整个头大小（包含节表）为：

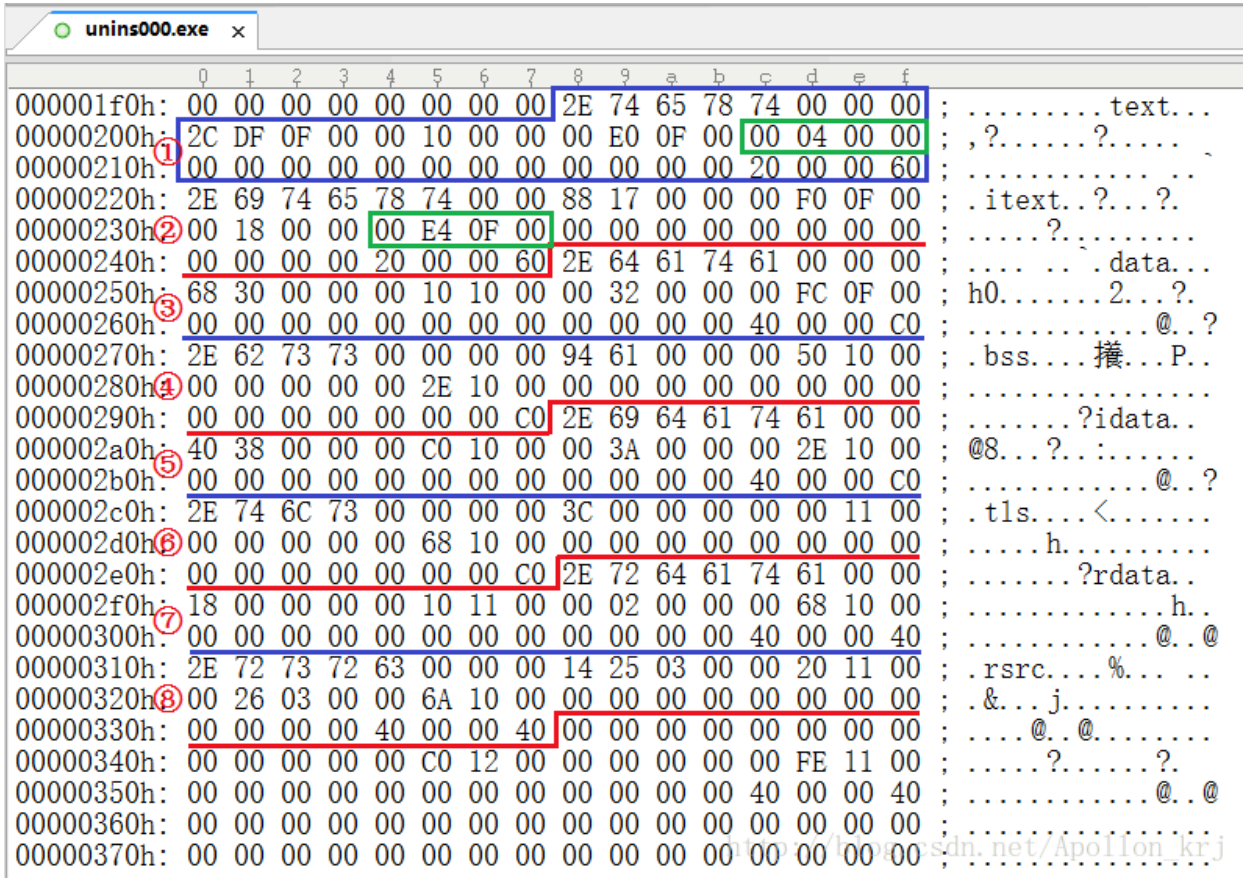
> SizeOfHeaders = SectionTableStart + SizeOfSectionTable =
> **_IMAGE_DOS_HEADER.e_lfanew +**
> **IMAGE_SIZEOF_SIGNATURE +**
> **IMAGE_SIZEOF_FILE_HEADER +**
> **_IMAGE_FILE_HEADER.SizeOfOptionalHeader +**
> **(_IMAGE_FILE_HEADER.NumberOfSections * IMAGE_SIZEOF_SECTION_HEADER)**

综合来说： PE文件的DOS Header、PE Header、Section Table、Sections的具体分布情况以及Section Table的一个**struct _IMAGE_SECTION_HEADER** 成员的细节描述如下图所示：



下来我们就具体的成员进行测试与分析（采用和DOS头与PE头解析 相同的exe文件进行测试）：

我们知道了NT的偏移地址为：e_lfanew = 00000100H，而可选PE头的大小为E0H。所以SectionTableStart = 100H + 4H + 14H + E0H = 0000 01F8H。即节表起始偏移地址为：0000 01F8H，如下所示：



我们在标准PE头**_IMAGE_FILE_HEADER**中获取的**NumberOfSections** 大小为08H，所以总共有八个节（Sections），节表中也有8个结构体元素，每个占用40字节。SizeOfHeaders = SectionTableStart + SizeOfSectionTable = 0000 01F8H + （40*8）（D） = 0000 01F8H + 140H = 338H。由于SizeOfHeaders是对齐后的大小，而FileAlignment = 200H，所以SizeOfHeaders补齐后为400H。如下图所示：

```
000003e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000003f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000400h: 04 10 40 00 03 07 42 6F 6F 6C 65 61 6E 01 00 00 ; ..@...Boolean...
00000410h: 00 00 01 00 00 00 00 10 40 00 05 46 61 6C 73 65 ; ........@..False
00000420h: 04 54 72 75 65 8D 40 00 2C 10 40 00 02 08 41 6E ; .True弟.,.@...An
00000430h: 73 69 43 68 61 72 01 00 00 00 00 FF 00 00 00 90 ; siChar.....  ...?
00000440h: 12 06 73 74 72 69 6E 67 BC 10 40 00 0B 0A 57 69 ; ..string?@...Wi
00000450h: 64 65 53 74 72 69 6E 67 CC 10 40 00 0A 0A 41 6E ; deString?@...An
00000460h: 73 69 53 74 72 69 6E 67 00 00 8B C0 E0 10 40 00 ; siString..嫩?@.
00000470h: 0C 07 56 61 72 69 61 6E 74 8D 40 00 F0 10 40 00 ; ..Variant弟.?@.
00000480h: 0C 0A 4F 6C 65 56 61 72 69 61 6E 74 54 11 40 00 ; ..OleVariantT.@.
00000490h: 44 10 40 00 09 04 43 68 61 72 03 00 00 00 00 FF ; D.@...Char.....
000004a0h: FF 00 00 90 58 10 40 00 01 07 49 6E 74 65 67 65 ;   ..恋.@...Intege
000004b0h: 72 04 00 00 00 80 FF FF FF 7F 8B C0 70 10 40 00 ; r....€       嫩p.@.
000004c0h: 01 04 42 79 74 65 01 00 00 00 00 FF 00 00 00 90 ; ..Byte.....  ...?
000004d0h: 84 10 40 00 01 04 57 6F 72 64 03 00 00 00 00 FF ; ?@...Word.....
000004e0h: FF 00 00 90 98 10 40 00 01 08 43 61 72 64 69 6E ;   ..悩.@...Cardin
000004f0h: 61 6C 05 00 00 00 00 FF FF FF FF 90 B0 10 40 00 ; al.....        琳.@.
00000500h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000510h: 00 00 00 00 00 00 00 00 00 00 00 00 54 11 40 00 ; ............T.@.
00000520h: 08 00 00 00 00 00 00 00 54 41 40 00 5C 41 40 00 ; ........TA@.\A@.
00000530h: E4 42 40 00 DC 42 40 00 FC 42 40 00 00 43 40 00 ; 銷@.屢@.匏@..C@.
00000540h: 04 43 40 00 F8 42 40 00 30 40 40 00 4C 40 40 00 ; .C@.鳥@.0@@.L@@.
```

0000 0400H前的0均是用来补齐的。我们第一个节的名字为8字节：2E 74 65 78 74 00 00 00，即“.text…”。VirtualSize大小为4个字节：000FDF2CH，不是200H的倍数，所以补齐需要到000FE000H。而其节首地址为：PointerToRawData（第5个元素）= 00000400H，即紧接着节表对齐处开始。则第一节结束地址为：0000 0400H + 000F E000H = 000FE400H（即第四个元素SizeOfRawData：文件对齐后的尺寸）。如下所示：

```
000fe3e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000fe3f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000fe400h: 83 2D E0 58 50 00 01 0F 83 A0 00 00 00 E8 E2 3D ; ?鄖P...儭...桱=
000fe410h: F0 FF C6 05 0C 10 50 00 02 C7 05 14 50 50 00 74 ; ??..P..?.PP.t
000fe420h: 12 40 00 C7 05 18 50 50 00 7C 12 40 00 C6 05 56 ; .@.?.PP.|.@.?V
000fe430h: 50 50 00 02 E8 97 9D F0 FF A3 58 50 50 00 C7 05 ; PP..钏瀰    PP.?
000fe440h: 00 50 50 00 CC 77 40 00 E8 EB 4D F0 FF 84 C0 74 ; .PP.夔@.桦M?钫t
```

FE400H前面的0是用来对齐的内存填充的。而查看节表第二个元素的第5个成员，得知第二节开始地址为000FE400H，是紧接着对齐后的第一节的，依次类推。由于查看时看的是硬盘上的文件，所以VirtualAddress成员没有意义，但是在内存中的分析方式和硬盘上PE文件是一样的，需要注意的是ImageBase（硬盘上基址是0）和SectionAlignment（硬盘上是FileAlignment）两个关键点。

除了exe中不太会用到几个成员外还有最后一个属性成员Characteristics，其常见的属性情况如下所示：

```
00000020h [IMAGE_SCN_CNT_CODE]               // Section contains code.(包含可执行代码)
00000040h [IMAGE_SCN_CNT_INITIALIZED_DATA]   // Section contains initialized data.(该块包含已初始化的数据)
00000080h [IMAGE_SCN_CNT_UNINITIALIZED_DATA] // Section contains uninitialized data.(该块包含未初始化的数据)
00000200h [IMAGE_SCN_LNK_INFO]               // Section contains comments or some other type of information.
00000800h [IMAGE_SCN_LNK_REMOVE]             // Section contents will not become part of image.
00001000h [IMAGE_SCN_LNK_COMDAT]             // Section contents comdat.
00004000h [IMAGE_SCN_NO_DEFER_SPEC_EXC]      // Reset speculative exceptions handling bits in the TLB entries for this section.
00008000h [IMAGE_SCN_GPREL]                  // Section content can be accessed relative to GP.
00500000h [IMAGE_SCN_ALIGN_16BYTES]          // Default alignment if no others are specified.
01000000h [IMAGE_SCN_LNK_NRELOC_OVFL]        // Section contains extended relocations.
02000000h [IMAGE_SCN_MEM_DISCARDABLE]        // Section can be discarded.
04000000h [IMAGE_SCN_MEM_NOT_CACHED]         // Section is not cachable.
08000000h [IMAGE_SCN_MEM_NOT_PAGED]          // Section is not pageable.
10000000h [IMAGE_SCN_MEM_SHARED]             // Section is shareable(该块为共享块).
20000000h [IMAGE_SCN_MEM_EXECUTE]            // Section is executable.(该块可执行)
40000000h [IMAGE_SCN_MEM_READ]               // Section is readable.(该块可读)
80000000h [IMAGE_SCN_MEM_WRITE]              // Section is writeable.(该块可写)
```


而第一节的属性为60 00 00 20 = 40 00 00 00H | 20 00 00 00H | 00 00 00 20H = **IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_CNT_CODE**，即.text包含可执行代码，可读可执行。第二节与第一节相同，第三节为C0000040即**40000000H | 80000000H | 00000040H**，“.data”为可读可写已初始化；第四节“.bss”为C0000000，即可读可写。之后的节均是相同的分析方式。

注意：SizeOfRawData的大小在硬盘上一定不小于Misc，但在加载进内存中运行时则不一定大于Misc。
比如：在该节有一个未初始化的数组char a[1000],在编译连接完成生成.exe文件后并没有分配内存(因为是未初始化的)，对应在硬盘上就是没有预留的一块地址空间初始化为0。SizeOfRawData是文件对齐的倍数，文件对齐时参考的大小是不包含未初始化的a[1000]，可能大小是170则对齐是200，SizeOfRawData(200)在硬盘上大于Misc(170)。
而在真正加载进内存运行时，a[1000]就要分配1000字节，Misc就变成了(1170)超过了SizeOfRawData的大小(200)。产生这一特点的原因就是：**SizeOfRawData是在编译连接完成后就确定的，是不可变的。而Misc是在硬盘上内存里随时可能发生变化的。**更本质一点的原因是：未初始化的变量在装载时才分配空间，如果不存在这种动态变化的且无人为修改Misc，则SizeOfRawData定不小于Misc。

2、代码解析节表：

```
1  void PETool::print_SECTIONS_TABEL()
2  {
3      fprintf(fp_peMess, "节表(Section Table):\n");
4      IMAGE_SECTION_HEADER * section = section_header;
5      char name[IMAGE_SIZEOF_SHORT_NAME + 1] = {0};
6      for(int i = 0; i < sectionNum; i++){
7          memset(name, 0, IMAGE_SIZEOF_SHORT_NAME + 1);
8
```

```
        memcpy(name, section+i, IMAGE_SIZEOF_SHORT_NAME);
        fprintf(fp_peMess,"\tsection[%d]:\n",i);

        fprintf(fp_peMess, "\t\tName                :[%s]\n",name);
        fprintf(fp_peMess, "\t\tVirtualSize:        :[%08X]\n",section[i].Misc.VirtualSize);
        fprintf(fp_peMess, "\t\tVirtualAddress      :[%08X]\n",section[i].VirtualAddress);
        fprintf(fp_peMess, "\t\tSizeOfRawData       :[%08X]\n",section[i].SizeOfRawData);
        fprintf(fp_peMess, "\t\tPointerToRawData    :[%08X]\n",section[i].PointerToRawData);
        fprintf(fp_peMess, "\t\tPointerToRelocations:[%08X]\n",section[i].PointerToRelocations);
        fprintf(fp_peMess, "\t\tPointerToLinenumbers:[%08X]\n",section[i].PointerToLinenumbers);
        fprintf(fp_peMess, "\t\tNumberOfRelocations :[%04X]\n",section[i].NumberOfRelocations);
        fprintf(fp_peMess, "\t\tNumberOfLinenumbers :[%04X]\n",section[i].NumberOfLinenumbers);
        fprintf(fp_peMess, "\t\tCharacteristics     :[%08X]\n",section[i].Characteristics);
    }
}
```