

1 PE文件结构

从本章开始就要学习PE相关的内容。

1.1 可执行文件

在了解PE之前，我们需要知道什么是可执行文件，从字面理解可执行文件就是可以由操作系统进行加载执行的文件。

Windows平台下的可执行文件的格式，我们称之为PE（Portable Executable）文件结构；Linux平台下的可执行文件格式，我们称之为ELF（Executable and Linking Format）文件结构。

仔细的人可能会发现PE的全称是**Portable Executable**，其中文意思就是便携的可执行，而ELF的全称**Executable and Linking Format**就是可执行可链接格式，那么两者之间的差距就出现了，Windows平台下的PE文件结构是便携的，也就表示其在Windows下是通用兼容的，**例如你在Windows7下的可执行文件也可以在Windows8、10系统下运行**，而Linux则不一样，不同内核编译的可执行文件在不同内核的环境下是无法使用的。

在这些领域下会用到PE文件格式：

1. 病毒和反病毒；
2. 外挂和反外挂；
3. 加壳和脱壳（保护与破解）；
4. 无源码的情况下修改功能、汉化软件...

1.2 识别PE文件

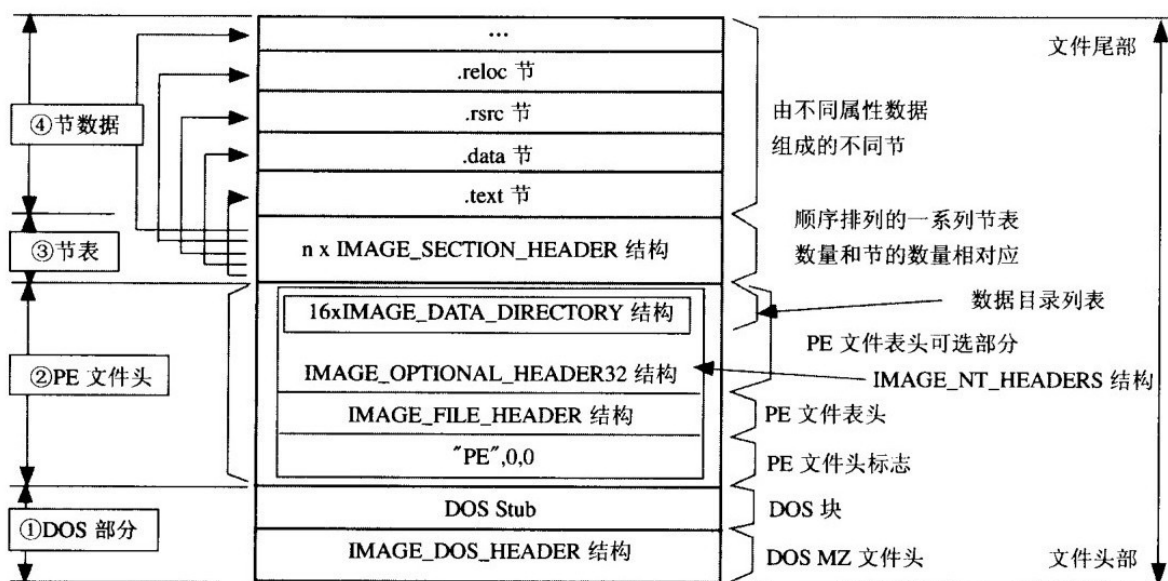
你想要识别一个文件是不是PE文件，或者是不是一个可执行文件，可以根据PE指纹来识别：首先你需要找到一个可以以16进制打开PE文件的工具（010 Editor），然后找到一个PE文件，用该工具打开PE文件，**在文件的开始位置有一个0x5A4D(十进制：MZ)，接着在0x003C位置向后有一个0x100，接着我们再去寻找0x100位置就会出现一个0x4550(十进制：PE)**，那么当你用这个方法可以顺利的走通整个流程找到PE，就表示这是一个PE文件，同样这这也是一个**PE指纹**：

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ.....ÿÿ..
0010h:	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	,.....@.....
0020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00
0040h:	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..°..'.Í! ,.LÍ!Th
0050h:	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program cannot
0060h:	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	be run in DOS
0070h:	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....\$.
0080h:	EF	31	A7	9A	AB	50	C9	C9	AB	50	C9	C9	AB	50	C9	C9	ï1šš«PÉE«PÉE«PÉE
0090h:	BF	3B	CA	C8	A6	50	C9	C9	BF	3B	CC	C8	3F	50	C9	C9	¿;ÈÈ!PÉE¿;ÌÈ?PÉE
00A0h:	BF	3B	CD	C8	B9	50	C9	C9	F9	25	CC	C8	8E	50	C9	C9	¿;ÌÈ'PÉEù%ÌÈŽPÉE
00B0h:	F9	25	CD	C8	BB	50	C9	C9	F9	25	CA	C8	B8	50	C9	C9	ù%ÌÈ»PÉEù%ÈÈ,PÉE
00C0h:	BF	3B	C8	C8	AE	50	C9	C9	AB	50	C8	C9	D5	50	C9	C9	¿;ÈÈ@PÉE«PÉEÔPÉE
00D0h:	1E	25	C0	C8	AF	50	C9	C9	1E	25	36	C9	AA	50	C9	C9	.%ÀÈ˘PÉE.%6È˘aPÉE
00E0h:	AB	50	5E	C9	AA	50	C9	C9	1E	25	CB	C8	AA	50	C9	C9	«P^È˘aPÉE.%ÈÈ˘aPÉE
00F0h:	52	69	63	68	AB	50	C9	C9	00	00	00	00	00	00	00	00	Rich«PÉE.....
0100h:	50	45	00	00	4C	01	05	00	2F	7B	86	61	00	00	00	00	PE..L.../{†a...
0110h:	00	00	00	00	E0	00	02	01	0B	01	0E	1D	00	9A	03	00à.....š..
0120h:	00	A2	33	06	00	00	00	00	60	DB	00	00	00	10	00	00	..ç3.....`0.....

如上示例中我使用的是exe后缀的文件，但即使不是exe后缀的文件，例如.sys、.dll后缀的文件，实际上你通过这种方式会发现它们也是PE文件，所以我们不要只看后缀名来认定是不是PE文件，而要具体去看文件中的指纹。

1.3 PE文件的整体结构

如上所述中我们可以了解到通过PE指纹的方式识别PE文件，但是我又是如何知道这是否是一个PE文件的呢？这是因为PE文件结构有一个规范和定义，如下图所示就是PE文件的整体结构：

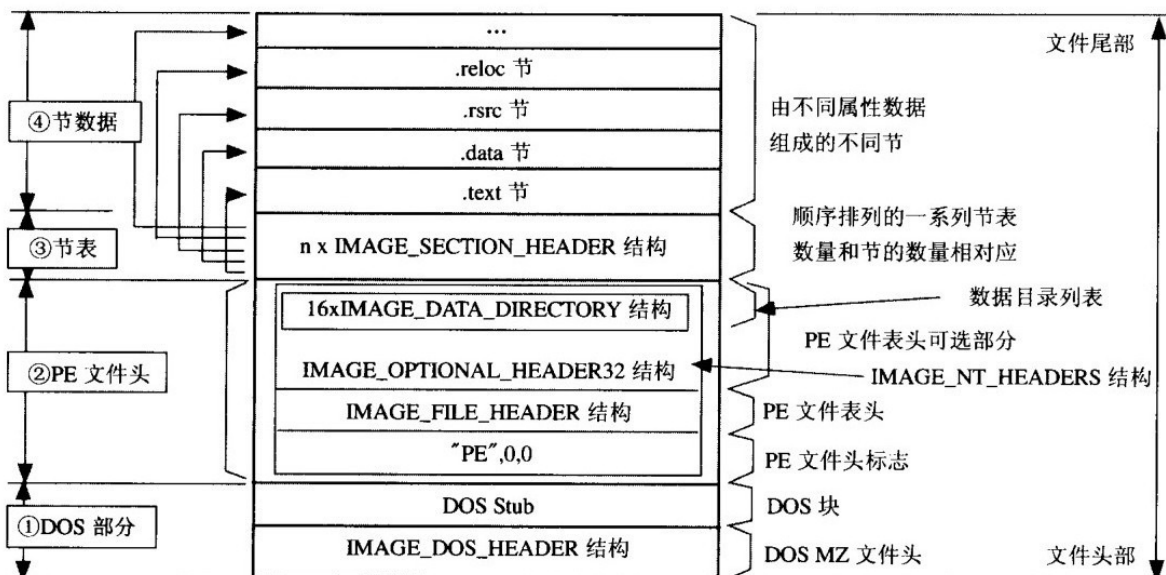


如上图所示众可以发现PE文件有很多结构，其结构格式图可以见附件：PE格式图.pdf（看上去很多，但不需要害怕，一步一步学下去还是非常容易理解的）

2 PE文件的两种状态

2.1 主要结构体

上文中，我们了解了PE文件的整体结构，我们可以看见其中有很多结构体：



这几个主要结构体分别对应的宽度如下所示：

结构体	宽度 (字节)
IMAGE_DOS_HEADER	64
IMAGE_FILE_HEADER	20
IMAGE_OPTIONAL_HEADER32	224
IMAGE_SECTION_HEADER	40



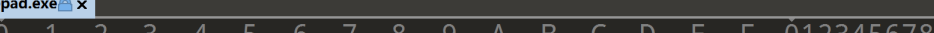
这些结构体你都可以在**Microsoft Visual Studio\VC98\Include\WINNT.H**头文件中看见。

2.2 文件分析

这些结构体的具体细节，在之后的章节会详细了解，现在我们只需要按照PE文件的整体结构来看一个PE文件（使用010 Editor打开文件）。

2.2.1 DOS部分

首先来看一下DOS部分，首先是DOS MZ文件头**IMAGE_DOS_HEADER**结构，这个结构占64字节，文件前四行就是了（类似010 Editor这种编辑器，**单行都是16字节**）：



```

Startup notepad.exe x
0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0000h: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....ÿÿ..
0010h: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ,.....@.....
0020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0030h: 00 00 00 00 00 00 00 00 00 00 00 00 F8 00 00 00 .....ø...
0040h: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..°.'.'!.,.L!Th
0050h: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program cannot

```

接着是DOS块，这个大小是不固定的，但是在上文中，我们了解到可以根据某个值定位到PE文件头，我们可以先找到PE文件头，**这样夹在他们之间的就是DOS块了**，在这里就是**IMAGE_DOS_HEADER**结构体的**e_lfanew**成员，如上图所示这里对应的值是0xF8：

```

typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                      // Magic number
    WORD   e_cblp;                       // Bytes on last page of file
    WORD   e_cp;                          // Pages in file
    WORD   e_crlc;                       // Relocations
    WORD   e_cparhdr;                    // Size of header in paragraphs
    WORD   e_minalloc;                   // Minimum extra paragraphs needed
    WORD   e_maxalloc;                   // Maximum extra paragraphs needed
    WORD   e_ss;                         // Initial (relative) SS value
    WORD   e_sp;                         // Initial SP value
    WORD   e_csum;                       // Checksum
    WORD   e_ip;                         // Initial IP value
    WORD   e_cs;                         // Initial (relative) CS value
    WORD   e_lfarlc;                     // File address of relocation table
    WORD   e_ovno;                       // Overlay number
    WORD   e_res[4];                     // Reserved words
    WORD   e_oemid;                      // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                    // OEM information; e_oemid specific
    WORD   e_res2[10];                   // Reserved words
    LONG   e_lfanew;                     // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;

```

所以如下图所示中，绿色框标记的部分就是DOS块：

Startup x	notepad.exe x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:		4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ.....ÿÿ..
0010h:		B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	,.....@.....
0020h:		00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0030h:		00	00	00	00	00	00	00	00	00	00	00	00	F8	00	00	00ø...
0040h:		0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..°..'Í!..LÍ!Th
0050h:		69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
0060h:		74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
0070h:		6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....\$.....
0080h:		14	D9	54	C8	50	B8	3A	9B	50	B8	3A	9B	50	B8	3A	9B	.ÛTÈP,:>P,:>P,:>
0090h:		59	C0	A9	9B	7E	B8	3A	9B	44	D3	3E	9A	5A	B8	3A	9B	YÀ@>~,:>DÓ>šZ,:>
00A0h:		44	D3	39	9A	53	B8	3A	9B	44	D3	3B	9A	59	B8	3A	9B	DÓ9šS,:>DÓ;šY,:>
00B0h:		50	B8	3B	9B	7C	BD	3A	9B	44	D3	32	9A	4E	B8	3A	9B	P,:> ½:>DÓ2šN,:>
00C0h:		44	D3	3F	9A	71	B8	3A	9B	44	D3	C7	9B	51	B8	3A	9B	DÓ?šq,:>DÓC>Q,:>
00D0h:		44	D3	C5	9B	51	B8	3A	9B	44	D3	38	9A	51	B8	3A	9B	DÓÁ>Q,:>DÓ8šQ,:>
00E0h:		52	69	63	68	50	B8	3A	9B	00	00	00	00	00	00	00	00	RichP,:>.....
00F0h:		00	00	00	00	00	00	00	00	50	45	00	00	64	86	07	00PE..dt..
0100h:		69	BD	FC	86	00	00	00	00	00	00	00	00	F0	00	22	00	i½ü†.....ð."
0110h:		0B	02	0E	14	00	4A	02	00	00	E2	00	00	00	00	00	00J...â.....

2.2.2 PE文件头

接着来看PE文件头，其第一个是PE文件头标志，这里占4字节，也就是上文图中所示的0x4550（**PE标识是不能修改的**），所以在这不赘述了；PE文件头第二部分就是**PE文件表头IMAGE_FILE_HEADER**结构，这个结构占20字节，我们也称之为**标准PE头**：

00F0h:	00 00 00 00	00 00 00 00	50 45 00 00	64 86 07 00PE..d†..
0100h:	69 BD FC 86	00 00 00 00	00 00 00 00	F0 00 22 00	i½ü†.....ě.".
0110h:	0B 02 0E 14	00 4A 02 00	00 E2 00 00	00 00 00 00J...â.....

继续看PE文件头的第三个部分PE文件表头可选部分，我们也称之为扩展PE头，其就是 **IMAGE_OPTIONAL_HEADER32** 结构，默认情况下它在32位下是224字节，在64位下是240字节，你也可以通过 **IMAGE_FILE_HEADER** 结构的成员去获取/修改扩展PE头的宽度：

```
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;
```

在这里也就对应着如下图中的0xF0（因为当前系统和文件都是64位的）：

00F0h:	00 00 00 00	00 00 00 00	50 45 00 00	64 86 07 00PE..d†..
0100h:	69 BD FC 86	00 00 00 00	00 00 00 00	F0 00 22 00	i½ü†.....ě.".
0110h:	0B 02 0E 14	00 4A 02 00	00 E2 00 00	00 00 00 00J...â.....

也就表示在这里扩展PE头的宽度就是240字节：

00F0h:	00	00	00	00	00	00	00	00	00	50	45	00	00	64	86	07	00
0100h:	69	BD	FC	86	00	00	00	00	00	00	00	00	00	F0	00	22	00
0110h:	0B	02	0E	14	00	4A	02	00	00	00	E2	00	00	00	00	00	00
0120h:	B0	3D	02	00	00	10	00	00	00	00	00	00	40	01	00	00	00
0130h:	00	10	00	00	00	02	00	00	00	0A	00	00	00	0A	00	00	00
0140h:	0A	00	00	00	00	00	00	00	00	00	80	03	00	00	04	00	00
0150h:	B0	6B	03	00	02	00	60	C1	00	00	08	00	00	00	00	00	00
0160h:	00	10	01	00	00	00	00	00	00	00	00	10	00	00	00	00	00
0170h:	00	10	00	00	00	00	00	00	00	00	00	00	00	10	00	00	00
0180h:	00	00	00	00	00	00	00	00	00	88	D2	02	00	30	02	00	00
0190h:	00	60	03	00	D8	0B	00	00	00	00	30	03	00	28	11	00	00
01A0h:	00	00	00	00	00	00	00	00	00	00	70	03	00	D4	02	00	00
01B0h:	50	AD	02	00	54	00	00	00	00	00	00	00	00	00	00	00	00
01C0h:	00	00	00	00	00	00	00	00	00	B8	67	02	00	28	00	00	00
01D0h:	A0	66	02	00	18	01	00	00	00	00	00	00	00	00	00	00	00
01E0h:	E0	67	02	00	18	09	00	00	00	A0	CB	02	00	E0	00	00	00
01F0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0200h:	2E	74	65	78	74	00	00	00	00	F2	49	02	00	00	10	00	00

扩展PE头之所以数据宽度较大，是因为其有一个成员是结构体数组：

```

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;
    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;
    DWORD    NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;

```

1	#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES 16
---	---

这个成员的宽度就是16个**IMAGE_DATA_DIRECTORY**结构体的宽度。

2.2.3 节表、节数据

节表很重要，其决定节数据的相关属性，而节数据是我们真正存储数据的地方，其数量和节表是对应的。

节表就是N个IMAGE_SECTION_HEADER结构体组成的，该结构体数据宽度是40字节：

```
#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;
```

我们可以在该PE文件中看一下有多少个IMAGE_SECTION_HEADER结构体，如下图用不同颜色标记的就是每个节，其实通过编辑器右边的内容你就可以大致知道每个节的表示什么类型了：

01F0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0200h:	2E 74 65 78	74 00 00 00	F2 49 02 00	00 10 00 00	.text...òI.....
0210h:	00 4A 02 00	00 04 00 00	00 00 00 00	00 00 00 00	.J.....
0220h:	00 00 00 00	20 00 00 60	2E 72 64 61	74 61 00 00`..rdata..
0230h:	7E 94 00 00	00 60 02 00	00 96 00 00	00 4E 02 00	~"....-...N..
0240h:	00 00 00 00	00 00 00 00	00 00 00 00	40 00 00 40@..@
0250h:	2E 64 61 74	61 00 00 00	A8 27 00 00	00 00 03 00	.data..."'.....
0260h:	00 0E 00 00	00 E4 02 00	00 00 00 00	00 00 00 00ä.....
0270h:	00 00 00 00	40 00 00 C0	2E 70 64 61	74 61 00 00@..À.pdata..
0280h:	28 11 00 00	00 30 03 00	00 12 00 00	00 F2 02 00	(....0.....ò..
0290h:	00 00 00 00	00 00 00 00	00 00 00 00	40 00 00 40@..@
02A0h:	2E 64 69 64	61 74 00 00	78 01 00 00	00 50 03 00	.didat..x....P..
02B0h:	00 02 00 00	00 04 03 00	00 00 00 00	00 00 00 00
02C0h:	00 00 00 00	40 00 00 C0	2E 72 73 72	63 00 00 00@..À.rsrc..
02D0h:	D8 0B 00 00	00 60 03 00	00 0C 00 00	00 06 03 00	Ø.....
02E0h:	00 00 00 00	00 00 00 00	00 00 00 00	40 00 00 40@..@
02F0h:	2E 72 65 6C	6F 63 00 00	D4 02 00 00	00 70 03 00	.reloc..ô....p..
0300h:	00 04 00 00	00 12 03 00	00 00 00 00	00 00 00 00
0310h:	00 00 00 00	40 00 00 42	00 00 00 00	00 00 00 00@..B.....
0320h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

在当前PE文件中我们可以知道有6个节表，那也就表示在文件中存储数据的也就有6个部分，在节表之后的就是编译器的插入的数据，而编译器又是如何知道从哪开始插入数据呢？这实际上取决于一个扩展PE头的一个成员 **SizeOfHeaders**：

```
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;
    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;
    DWORD    NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[ IMAGE_NUMBEROF_DIRECTORY_ENTRIES ];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

该成员用来表示DOS头、PE头与节表加起来按照文件对齐以后的大小。这个真正的大小实际上取决于另外一个成员 **FileAlignment**，**SizeOfHeaders**存储的数值一定是**FileAlignment**的整数倍，默认情况下该成员的值为0x200。

假设当前DOS头、PE头与节表加起来的宽度为302，而成员**FileAlignment**的值为200，这时候成员**SizeOfHeaders**的值按**FileAlignment**的值进行文件对齐就应该是400，而之所以需要文件对齐是为了提高执行效率，这是一个牺牲空间换时间的一种策略，我们可以在当前PE文件中查看这两个成员：

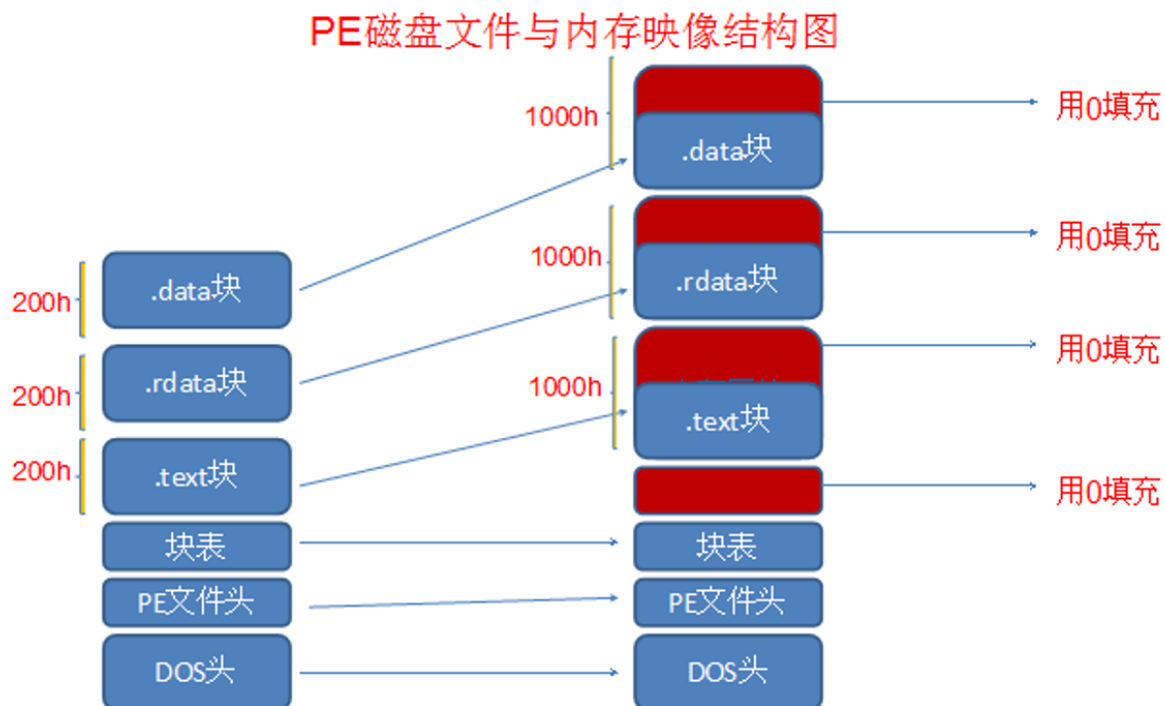
0110h:	0B 02 0E 14	00 4A 02 00	00 E2 00 00	00 00 00 00
0120h:	B0 3D 02 00	00 10 00 00	00 00 00 40	01 00 00 00
0130h:	00 10 00 00	00 02 00 00	0A 00 00 00	0A 00 00 00
0140h:	0A 00 00 00	00 00 00 00	00 80 03 00	00 04 00 00
0150h:	B0 6B 03 00	02 00 60 C1	00 00 08 00	00 00 00 00
0160h:	00 10 01 00	00 00 00 00	00 00 10 00	00 00 00 00
0170h:	00 10 00 00	00 00 00 00	00 00 00 00	10 00 00 00
0180h:	00 00 00 00	00 00 00 00	88 D2 02 00	30 02 00 00
0190h:	00 60 03 00	D8 0B 00 00	00 30 03 00	28 11 00 00
01A0h:	00 00 00 00	00 00 00 00	00 70 03 00	D4 02 00 00
01B0h:	50 AD 02 00	54 00 00 00	00 00 00 00	00 00 00 00
01C0h:	00 00 00 00	00 00 00 00	B8 67 02 00	28 00 00 00
01D0h:	A0 66 02 00	18 01 00 00	00 00 00 00	00 00 00 00
01E0h:	E0 67 02 00	18 09 00 00	A0 CB 02 00	E0 00 00 00
01F0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

这两个成员刚好与我们假设的值是一样的，所以这里**DOS头**、**PE头**与**节表**加起来按照文件对齐以后的大小就是400，但这样确实比实际大小要多出一些空间，这些空间默认会用0x00填充，但也有可能这些空间会被编译器插入一些信息，接着在**400地址**之后的就是**节数据**了。

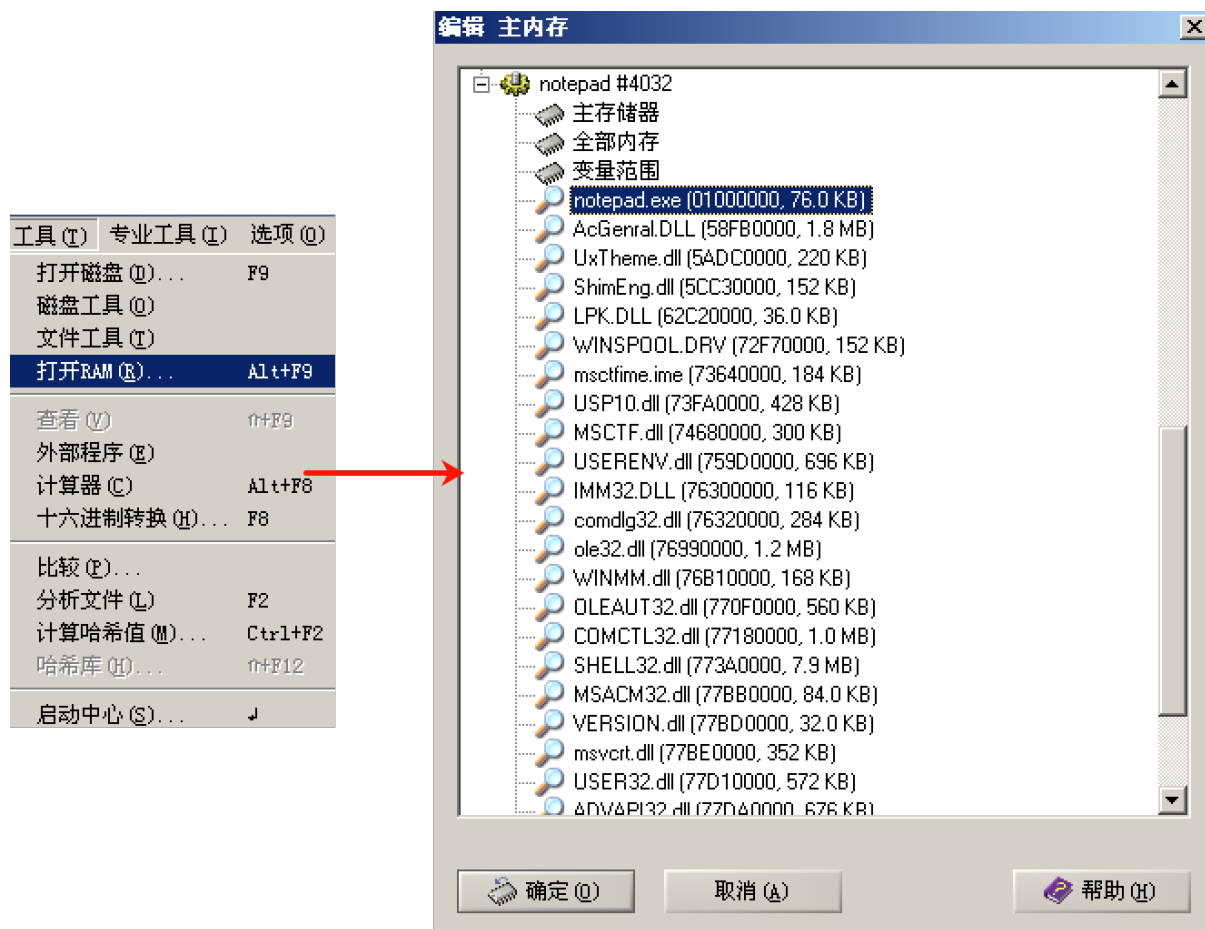
2.3 静动态差异

PE文件在运行前（静态，存储在磁盘上）和运行时（动态，运行在内存中）的格式是有差异的，这种差异对于我们理解PE文件是如何执行的来说很重要。

我们在之前的文件分析过程中实际上所看到的是静态的内容，其大小是要根据**FileAlignment**的值进行文件对齐的，但是在运行时则整体按照扩展PE头的成员**SectionAlignment**的值进行内存对齐，默认情况下该值为0x100：



我们可以实际观察一下在内存中的PE文件，首先打开记事本，然后在Winhex中这样选择：



然后找到对应的扩展PE头的成员**SectionAlignment**的值，这里就是默认的0x1000：

notepad.exe																				
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI	ASCII		
01000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ	ÿÿ		
01000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	,	@		
01000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
01000030	00	00	00	00	00	00	00	00	00	00	00	00	E0	00	00	00		à		
01000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	°	´í! , Lí!Th		
01000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is	program canno		
01000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t	be run in DOS		
01000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode.	\$		
01000080	EC	85	5B	A1	A8	E4	35	F2	A8	E4	35	F2	A8	E4	35	F2	i...	["a50"a50"a50		
01000090	6B	EB	3A	F2	A9	E4	35	F2	6B	EB	55	F2	A9	E4	35	F2	kè:	@a50kèU0@a50		
010000A0	6B	EB	68	F2	BB	E4	35	F2	A8	E4	34	F2	63	E4	35	F2	kèh0»	a50"a40cã50		
010000B0	6B	EB	6B	F2	A9	E4	35	F2	6B	EB	6A	F2	BF	E4	35	F2	kèk0@a50kèj0ç	a50		
010000C0	6B	EB	6F	F2	A9	E4	35	F2	52	69	63	68	A8	E4	35	F2	kèo0@a50Rich	"a50		
010000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
010000E0	50	45	00	00	4C	01	03	00	C3	7C	10	41	00	00	00	00	PE	L	Ã	A
010000F0	00	00	00	00	E0	00	0F	01	0B	01	07	0A	00	78	00	00		à		x
01000100	00	88	00	00	00	00	00	00	9D	73	00	00	00	10	00	00	^		□s	
01000110	00	90	00	00	00	00	00	01	00	10	00	00	00	02	00	00	□			
01000120	05	00	01	00	05	00	01	00	04	00	00	00	00	00	00	00				
01000130	00	30	01	00	00	04	00	00	59	79	01	00	02	00	00	80	0		Yy	€
01000140	00	00	04	00	00	10	01	00	00	00	10	00	00	00	00	00				
01000150	00	00	00	00	10	00	00	00	00	00	00	00	00	00	00	00				
01000160	04	76	00	00	C8	00	00	00	00	B0	00	00	30	7F	00	00	v	È	°	0
01000170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
01000180	00	00	00	00	00	00	00	00	50	13	00	00	1C	00	00	00			P	
01000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
010001A0	00	00	00	00	00	00	00	00	A8	18	00	00	40	00	00	00			@	
010001B0	50	02	00	00	D0	00	00	00	00	10	00	00	48	03	00	00	P	Đ		H
010001C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
010001D0	00	00	00	00	00	00	00	00	2E	74	65	78	74	00	00	00			.text	
010001E0	48	77	00	00	00	10	00	00	00	78	00	00	00	04	00	00	Hw		x	

3 PE文件整体结构解析

之前我们已经按照PE文件的整体结构对实际的PE文件进行了大致上的了解了，现在我们需要来看看每个结构的意义和作用。

3.1 DOS头

在之前，我们已经了解过PE文件的整体结构了，并且我们进行了静态差异的文件分析，其开头部分就是DOS部分，包含了DOS MZ文件头和DOS块，那么我们来了解一些DOS部分的结构和其相关意义。

3.1.1 DOS MZ文件头

DOS MZ文件头就是一个结构体IMAGE_DOS_HEADER，其定义如下所示：

```

1  typedef struct _IMAGE_DOS_HEADER {           // DOS .EXE header
2      WORD    e_magic;                         // Magic number
3      WORD    e_cblp;                         // Bytes on last page of file
4      WORD    e_cp;                           // Pages in file
5      WORD    e_crlc;                         // Relocations
6      WORD    e_cparhdr;                      // Size of header in paragraphs
7      WORD    e_minalloc;                     // Minimum extra paragraphs needed
8      WORD    e_maxalloc;                     // Maximum extra paragraphs needed
9      WORD    e_ss;                           // Initial (relative) SS value
10     WORD    e_sp;                            // Initial SP value
11     WORD    e_csum;                          // Checksum
12     WORD    e_ip;                           // Initial IP value
13     WORD    e_cs;                           // Initial (relative) CS value
14     WORD    e_lfarlc;                       // File address of relocation
15     table
16     WORD    e_ovno;                         // Overlay number
17     WORD    e_res[4];                       // Reserved words
18     WORD    e_oemid;                        // OEM identifier (for e_oeminfo)
19     WORD    e_oeminfo;                      // OEM information; e_oemid
20     specific
21     WORD    e_res2[10];                     // Reserved words
22     LONG    e_lfanew;                       // File address of new exe header
23 } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;

```

它有很多成员，但我们并不需要去深入的理解每个成员的含义和作用，这是因为这个结构体是给16位平台看的，而我们现在的环境大部分都是32位和64位的，所以现在的平台不再需要这个完整的结构体了，只需要其中的两个成员e_magic和e_lfanew。

你可以尝试在16进制的编辑器中去编辑某个EXE文件保留两个成员e_magic和e_lfanew，其他的以0x00填充，然后保存文件，你会发现修改后的文件还是可以正常运行的：

notepad.exe	notepad.exe																	ANSI ASCII	
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F			
00000000	4D	5A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	MZ		
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000030	00	00	00	00	00	00	00	00	00	00	00	00	E0	00	00	00			
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	° · í! , Lí!Th		

保留这两个成员的原因是因为它们代表着我们之前所说的PE指纹，操作系统也是根据这个来识别是否是PE文件的，所以不能够更改、删除（**e_magic**是一种标识，**e_lfanew**则表示PE文件头的位置）。

3.1.2 DOS块

DOS块就是夹在DOS MZ文件头和PE文件头之间的内容，这里面的内容可以根据自己的需要随意的修改和添加，并不会影响文件的正常运行。

notepad.exe	notepad.exe																	ANSI ASCII	
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F			
00000000	4D	5A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	MZ		
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000030	00	00	00	00	00	00	00	00	00	00	00	00	E0	00	00	00			
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	° · í! , Lí!Th		
00000050	69	73	20	70	72	6F	72	61	6D	20	63	61	6E	6E	6F		is program canno		
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS		
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode. \$		
00000080	EC	85	5B	A1	A8	E4	35	F2	A8	E4	35	F2	A8	E4	35	F2	i [, "a50" a50" a50		
00000090	6B	EB	3A	F2	A9	E4	35	F2	6B	EB	55	F2	A9	E4	35	F2	ke:o@a50keU0@a50		
000000A0	6B	EB	68	F2	BB	E4	35	F2	A8	E4	34	F2	63	E4	35	F2	keh0@a50" a40@a50		
000000B0	6B	EB	6B	F2	A9	E4	35	F2	6B	EB	6A	F2	BF	E4	35	F2	keh0@a50kej0@a50		
000000C0	6B	EB	6F	F2	A9	E4	35	F2	52	69	63	68	A8	E4	35	F2	keo0@a50Rich" a50		
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			

notepad.exe	notepad.exe																	ANSI ASCII	
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F			
00000000	4D	5A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	MZ		
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	E0	00	00		à	
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			

3.2 PE头

PE头整体就是如下这个结构体：

```

1  typedef struct _IMAGE_NT_HEADERS {
2      DWORD Signature; // PE标识
3      IMAGE_FILE_HEADER FileHeader; // 标准PE头
4      IMAGE_OPTIONAL_HEADER32 OptionalHeader; // 扩展PE头
5  } IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

第一个成员就是PE标识，该标识不能破坏，因为操作系统在启动一个程序的时候会检测这个标识。

3.2.1 标准PE头

标准PE头是PE头的第二个成员，它是如下所示的结构体：

```

1  typedef struct _IMAGE_FILE_HEADER {
2      WORD Machine; // 可以运行在什么样的CPU上
3      WORD NumberOfSections; // 表示节的数量
4      DWORD TimeDateStamp; // 编译器填写的时间戳
5      DWORD PointerToSymbolTable; // 调试相关
6      DWORD NumberOfSymbols; // 调试相关
7      WORD SizeOfOptionalHeader; // 扩展PE头的大小
```



```
8     WORD    Characteristics; // 文件属性
9 } IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

其第一个成员Machine表示可以运行在什么样的CPU上，如果它的值为0x0则表示可以运行在任意的CPU上，支持在Intel 386以及后续的类型CPU运行则值为0x14c，支持64位的CPU型号则值为0x8664。

我们可以分别在32位、64位系统上提取notepad.exe进行对比来看看这个成员（010 Editor → Tools → Compare Files...）：

notepad_x32.exe x																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000h:	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00
0010h:	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00
0020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0030h:	00	00	00	00	00	00	00	00	00	00	00	00	E0	00	00	00
0040h:	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68
0050h:	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F
0060h:	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20
0070h:	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00
0080h:	EC	85	5B	A1	A8	E4	35	F2	A8	E4	35	F2	A8	E4	35	F2
0090h:	6B	EB	3A	F2	A9	E4	35	F2	6B	EB	55	F2	A9	E4	35	F2
00A0h:	6B	EB	68	F2	BB	E4	35	F2	A8	E4	34	F2	63	E4	35	F2
00B0h:	6B	EB	6B	F2	A9	E4	35	F2	6B	EB	6A	F2	BF	E4	35	F2
00C0h:	6B	EB	6F	F2	A9	E4	35	F2	52	69	63	68	A8	E4	35	F2
00D0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00E0h:	50	45	00	00	4C	01	03	00	C3	7C	10	41	00	00	00	00
00F0h:	00	00	00	00	E0	00	0F	01	0B	01	07	0A	00	78	00	00
0100h:	00	88	00	00	00	00	00	00	9D	73	00	00	00	10	00	00
0110h:	00	90	00	00	00	00	00	01	00	10	00	00	00	02	00	00
0120h:	05	00	01	00	05	00	01	00	04	00	00	00	00	00	00	00
0130h:	00	30	01	00	00	04	00	00	59	79	01	00	02	00	00	80
0140h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
notepad_x64.exe x																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000h:	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00
0010h:	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00
0020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0030h:	00	00	00	00	00	00	00	00	00	00	00	00	E8	00	00	00
0040h:	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68
0050h:	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F
0060h:	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20
0070h:	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00
0080h:	83	C2	32	29	C7	A3	5C	7A	C7	A3	5C	7A	C7	A3	5C	7A
0090h:	CE	DB	D8	7A	C6	A3	5C	7A	CE	DB	C9	7A	C5	A3	5C	7A
00A0h:	CE	DB	CF	7A	DA	A3	5C	7A	C7	A3	5D	7A	33	A3	5C	7A
00B0h:	CE	DB	DF	7A	D3	A3	5C	7A	CE	DB	D5	7A	CC	A3	5C	7A
00C0h:	CE	DB	C8	7A	C6	A3	5C	7A	CE	DB	CD	7A	C6	A3	5C	7A
00D0h:	52	69	63	68	C7	A3	5C	7A	00	00	00	00	00	00	00	00
00E0h:	00	00	00	00	00	00	00	00	50	45	00	00	64	86	06	00
00F0h:	B3	C9	5B	4A	00	00	00	00	00	00	00	00	F0	00	22	00
0100h:	0B	02	09	00	00	A8	00	00	00	58	02	00	00	00	00	00
0110h:	70	35	00	00	00	10	00	00	00	00	00	00	01	00	00	00
0120h:	00	10	00	00	00	02	00	00	06	00	01	00	06	00	01	00
0130h:	06	00	01	00	00	00	00	00	00	50	03	00	00	06	00	00
0140h:	40	57	00	00	00	00	00	00	00	00	00	00	00	00	00	00

第二个成员**NumberOfSections**表示当前PE文件中节的数量，也就是节表中有几个结构体；第三个成员**TimeDateStamp**表示编译器编译的时候插入的时间戳，与文件属性里面的创建时间和修改时间是无关的。

第四、第五个成员是调试相关的，我们暂时不用去了解；第六个成员**SizeOfOptionalHeader**表示扩展PE头的大小，默认情况下32位PE文件对应值0xE0，64位PE文件对应值为0xF0。

第七个成员**Characteristics**用来记录当前PE文件的一些属性，该成员是16位（2字节）大小，其每一数据位对应的属性如下所示：

数据位	常量符号	为 1 时的含义
0	IMAGE_FILE_RELOCS_STRIPPED	文件中不存在重定位信息
1	IMAGE_FILE_EXECUTABLE_IMAGE	文件是可执行的
2	IMAGE_FILE_LINE_NUMS_STRIPPED	不存在行信息
3	IMAGE_FILE_LOCAL_SYMS_STRIPPED	不存在符号信息
4	IMAGE_FILE_AGGRESSIVE_WS_TRIM	调整工作集
5	IMAGE_FILE_LARGE_ADDRESS_AWARE	应用程序可处理大于 2GB 的地址
6		此标志保留
7	IMAGE_FILE_BYTES_REVERSED_LO	小尾方式
8	IMAGE_FILE_32BIT_MACHINE	只在 32 位平台上运行
9	IMAGE_FILE_DEBUG_STRIPPED	不包含调试信息
10	IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	不能从可移动盘运行
11	IMAGE_FILE_NET_RUN_FROM_SWAP	不能从网络运行
12	IMAGE_FILE_SYSTEM	系统文件（如驱动程序），不能直接运行
13	IMAGE_FILE_DLL	这是一个 DLL 文件
14	IMAGE_FILE_UP_SYSTEM_ONLY	文件不能在多处理器计算机上运行
15	IMAGE_FILE_BYTES_REVERSED_HI	大尾方式

3.2.2 扩展PE头

扩展PE头在32位和64位环境下是不一样的，在本章节中只介绍32位扩展PE头。如下结构体就是32位的扩展PE头：

```

1  typedef struct _IMAGE_OPTIONAL_HEADER {
2      WORD    Magic; // PE32 : 10B PE32+ : 20B
3      BYTE    MajorLinkerVersion; // 链接器版本号
4      BYTE    MinorLinkerVersion; // 链接器版本号
5      DWORD   SizeOfCode; // 所有代码节的总和（文件对齐后的大小），编译器填的（没用）
6      DWORD   SizeOfInitializedData; // 包含所有已经初始化数据的节的总大小（文件对
7      DWORD   SizeOfUninitializedData; // 包含未初始化数据的节的总大小（文件对齐后
8      DWORD   AddressOfEntryPoint; // 程序入口
9      DWORD   BaseOfCode; // 代码开始的基址，编译器填的（没用）
10     DWORD   BaseOfData; // 数据开始的基址，编译器填的（没用）
11     DWORD   ImageBase; // 内存镜像基址
12     DWORD   SectionAlignment; // 内存对齐
13     DWORD   FileAlignment; // 文件对齐
14     WORD    MajorOperatingSystemVersion; // 标识操作系统版本号，主版本号
15     WORD    MinorOperatingSystemVersion; // 标识操作系统版本号，次版本号
16     WORD    MajorImageVersion; // PE文件自身的版本号
17     WORD    MinorImageVersion; // PE文件自身的版本号
18     WORD    MajorSubsystemVersion; // 运行所需子系统版本号
19     WORD    MinorSubsystemVersion; // 运行所需子系统版本号
20     DWORD   Win32VersionValue; // 子系统版本的值，必须为0

```

```

21     DWORD    SizeOfImage; // 内存中整个PE文件的映射的尺寸
22     DWORD    SizeOfHeaders; // 所有头加节表按照文件对齐后的大小, 否则加载会出错
23     DWORD    CheckSum; // 校验和
24     WORD     Subsystem; // 子系统, 驱动程序(1)、图形界面(2)、控制台/DLL(3)
25     WORD     DllCharacteristics; // 文件特性
26     DWORD    SizeOfStackReserve; // 初始化时保留的栈大小
27     DWORD    SizeOfStackCommit; // 初始化时实际提交的大小
28     DWORD    SizeOfHeapReserve; // 初始化时保留的堆大小
29     DWORD    SizeOfHeapCommit; // 初始化时实际提交的大小
30     DWORD    LoaderFlags; // 调试相关
31     DWORD    NumberOfRvaAndSizes; // 目录项数目
32     IMAGE_DATA_DIRECTORY
    DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; // 表, 结构体数组
33 } IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;

```

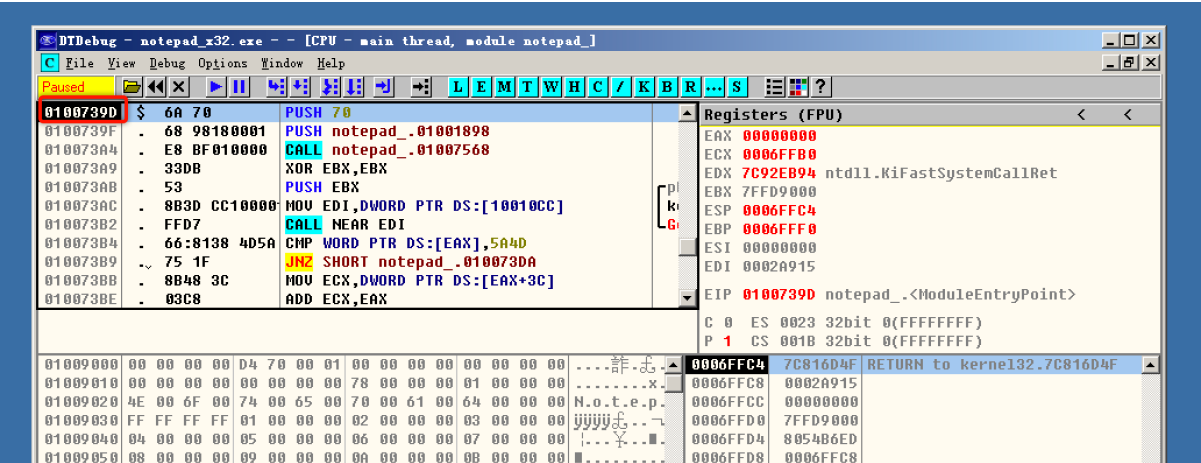
扩展PE头的成员有很多, 但我们不需要每个都记住, 大概的了解一下即可, 重点关注如下这几个成员:

成员**Magic**表示当前PE文件是32位还是64位, 32位时该值对应0x10B, 64位时该值对应0x20B。

成员**AddressOfEntryPoint**表示当前程序入口的地址, 这个成员要与成员**ImageBase**相加才能得出真正的入口地址, 成员**ImageBase**用来表示内存镜像基址, 也就是PE文件在内存中按内存对齐展开后的首地址, 我们可以在实际PE文件中看下, 如下图所示就是PE文件静态状态下的两个成员值, **AddressOfEntryPoint**为0x739D, **ImageBase**为0x1000000, 那么最终的程序在内存中的入口地址就是0x100739D:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI	ASCII	
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ	ÿÿ	
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	.	@	
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00000030	00	00	00	00	00	00	00	00	00	00	00	00	E0	00	00	00		à	
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	°	í	!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	i	s	program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t	b	e run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode.	\$	
00000080	EC	85	5B	A1	A8	E4	35	F2	A8	E4	35	F2	A8	E4	35	F2	i...	ä	ö"ä
00000090	6B	EB	3A	F2	A9	E4	35	F2	6B	EB	55	F2	A9	E4	35	F2	kë:ö	ä	ö
000000A0	6B	EB	68	F2	BB	E4	35	F2	A8	E4	34	F2	63	E4	35	F2	këhò»	ä	ö
000000B0	6B	EB	6B	F2	A9	E4	35	F2	6B	EB	6A	F2	BF	E4	35	F2	këkò@	ä	ö
000000C0	6B	EB	6F	F2	A9	E4	35	F2	52	69	63	68	A8	E4	35	F2	kéoö@	ä	ö
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
000000E0	50	45	00	00	4C	01	03	00	C3	7C	10	41	00	00	00	00	PE	L	ä
000000F0	00	00	00	00	E0	00	0F	01	0B	01	07	0A	00	78	00	00		à	x
00000100	00	88	00	00	00	00	00	00	9D	73	00	00	00	10	00	00	^	Qs	
00000110	00	90	00	00	00	00	00	01	00	10	00	00	00	02	00	00	□		
00000120	05	00	01	00	05	00	01	00	04	00	00	00	00	00	00	00			
00000130	00	30	01	00	00	04	00	00	59	79	01	00	02	00	00	80	0	Yy	€

那么如何证实推断的结果是正确的呢, 我们可以直接使用DTDebug之类的调试器打开这个PE文件, 调试器会自动在程序入口断点, 如下图所示则表示我们的推测是正确的:



成员FileAlignment、SectionAlignment和SizeOfHeader在之前的章节中已经了解过了，这里不再赘述。

成员SizeOfImage表示在内存中整个PE文件映射的大小，可比实际的值大（内存对齐之后的大小，也就表示必须是SectionAlignment的整数倍）。

成员Checksum表示校验和，是用来判断文件是否被修改的，它的计算方法就是文件的两个字节与两个字节相加，最终的值（不考虑溢出情况）就是校验和。

最后一个需要我们了解的成员是DllCharacteristics，它用来表示PE文件的特性，但不要被名字所迷惑，它不是针对DLL文件的；它的数据宽度是16位（4字节），其每一数据位对应的属性如下所示：

数据位	常量符号	为 1 时的含义
0		保留，必须为 0
1		保留，必须为 0
2		保留，必须为 0
3		保留，必须为 0
6	IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE	DLL 可以在加载时被重定位
7	IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY	强制代码实施完整性验证
8	IMAGE_DLLCHARACTERISTICS_NX_COMPAT	该映像兼容 DEP
9	IMAGE_DLLCHARACTERISTICS_NO_ISOLATION	可以隔离，但并不隔离此映像
10	IMAGE_DLLCHARACTERISTICS_NO_SEH	映像不使用 SEH（第 10 章）
11	IMAGE_DLLCHARACTERISTICS_NO_BIND	不绑定映像
12		保留，必须为 0
13	IMAGE_DLLCHARACTERISTICS_WDM_DRIVER	该映像为一个 WDM driver
14		保留，必须为 0
15	IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE	可用于终端服务器

3.3 PE节表

在PE中，节数据有几个，分别对应着什么类型以及其他相关的属性都是由PE节表来决定的，PE节表是一个结构体数组，结构体的定义如下所示：

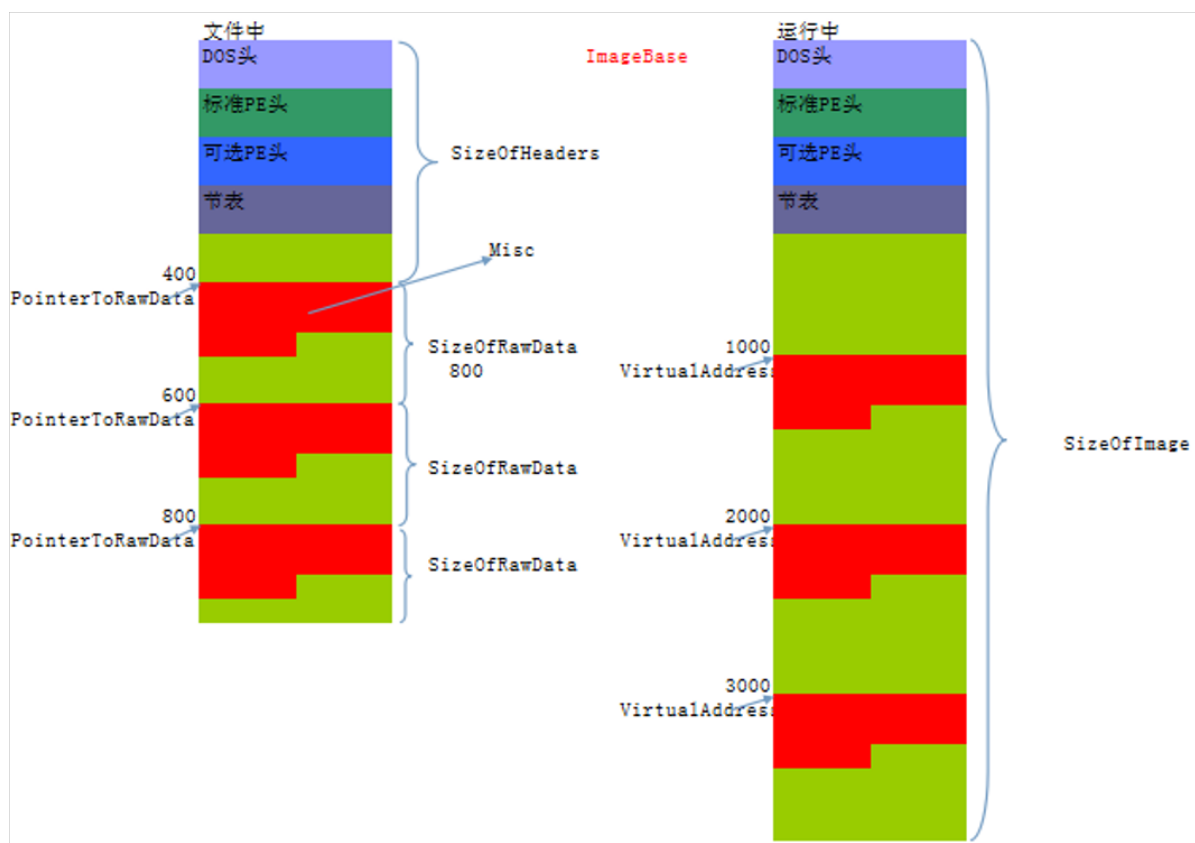
```
1 #define IMAGE_SIZEOF_SHORT_NAME 8
```

```

2  typedef struct _IMAGE_SECTION_HEADER {
3      BYTE    Name[IMAGE_SIZEOF_SHORT_NAME]; // ASCII字符串（节名），可自定义，
        只截取8个字节，可以8个字节都是名字
4      union { // Misc, 双字，是该节在没有对齐前的真实尺寸，该值可以不准确
5          DWORD    PhysicalAddress; // 真实宽度，这两个值是一个联合结构，可以使用其
        中的任何一个
6          DWORD    VirtualSize; // 一般是取后一个
7      } Misc;
8      DWORD    VirtualAddress; // 在内存中的偏移地址，加上ImageBase才是在内存中的真
        正地址
9      DWORD    SizeOfRawData; // 节在文件中对齐后的尺寸
10     DWORD    PointerToRawData; // 节区在文件中的偏移
11     DWORD    PointerToRelocations; // 调试相关
12     DWORD    PointerToLinenumbers; // 调试相关
13     WORD     NumberOfRelocations; // 调试相关
14     WORD     NumberOfLinenumbers; // 调试相关
15     DWORD    Characteristics; // 节的属性
16 } IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;

```

代码中的注释可以大致了解到每个成员的作用，其中有2个成员来描述节的大小，分别是没有对齐前的真实尺寸和对齐后的宽度，这时候会出现一种情况就是对齐前的真实尺寸大于对齐后的宽度，**这就是存在全局变量没有赋予初始值导致的，在文件存储中全局变量没有赋予初始值也就不占空间，但是在内存中是必须要赋予初始值的，这时候宽度就大了一些，所以在内存中节是谁大就按照谁去展开。**



与其他结构体一样，PE节也有属性，这就是成员Characteristics，其数据宽度是16位（4字节），其每一数据位对应的属性如下所示：

数据位	常量符号	位为 1 时的含义
5	IMAGE_SCN_CNT_CODE 或 00000020h	节中包含代码
6	IMAGE_SCN_CNT_INITIALIZED_DATA 或 00000040h	节中包含已初始化数据
7	IMAGE_SCN_CNT_UNINITIALIZED_DATA 或 00000080h	节中包含未初始化数据
8	IMAGE_SCN_LNK_OTHER 或 00000100h	保留供将来使用
25	IMAGE_SCN_MEM_DISCARDABLE 或 02000000h	节中的数据在进程开始以后将被丢弃，如 .reloc
26	IMAGE_SCN_MEM_NOT_CACHED 或 04000000h	节中的数据不会经过缓存
27	IMAGE_SCN_MEM_NOT_PAGED 或 08000000h	节中的数据不会被交换到磁盘
28	IMAGE_SCN_MEM_SHARED 或 10000000h	表示节中的数据将被不同的进程所共享
29	IMAGE_SCN_MEM_EXECUTE 或 20000000h	映射到内存后的页面包含可执行属性
30	IMAGE_SCN_MEM_READ 或 40000000h	映射到内存后的页面包含可读属性
31	IMAGE_SCN_MEM_WRITE 或 80000000h	映射到内存后的页面包含可写属性

更多可以参考如下：

```

1  //
2  // Section characteristics.
3  //
4  //     IMAGE_SCN_TYPE_REG                0x00000000 // Reserved.
5  //     IMAGE_SCN_TYPE_DSECT              0x00000001 // Reserved.
6  //     IMAGE_SCN_TYPE_NOLOAD              0x00000002 // Reserved.
7  //     IMAGE_SCN_TYPE_GROUP               0x00000004 // Reserved.
8  #define IMAGE_SCN_TYPE_NO_PAD            0x00000008 // Reserved.
9  //     IMAGE_SCN_TYPE_COPY                0x00000010 // Reserved.
10
11 #define IMAGE_SCN_CNT_CODE                0x00000020 // Section
12 // contains code.
13 #define IMAGE_SCN_CNT_INITIALIZED_DATA    0x00000040 // Section
14 // contains initialized data.
15 #define IMAGE_SCN_CNT_UNINITIALIZED_DATA  0x00000080 // Section
16 // contains uninitialized data.
17
18 #define IMAGE_SCN_LNK_OTHER               0x00000100 // Reserved.
19 #define IMAGE_SCN_LNK_INFO                0x00000200 // Section
20 // contains comments or some other type of information.
21 //     IMAGE_SCN_TYPE_OVER                0x00000400 // Reserved.
22 #define IMAGE_SCN_LNK_REMOVE              0x00000800 // Section
23 // contents will not become part of image.
24 #define IMAGE_SCN_LNK_COMDAT              0x00001000 // Section
25 // contents comdat.
26 //     IMAGE_SCN_MEM_PROTECTED - Obsolete 0x00002000 // Reserved.
27 //     IMAGE_SCN_MEM_NO_DEFER_SPEC_EXC    0x00004000 // Reset
28 // speculative exceptions handling bits in the TLB entries for this section.
29 #define IMAGE_SCN_GPREL                   0x00008000 // Section
30 // content can be accessed relative to GP
31 #define IMAGE_SCN_MEM_FARDATA              0x00008000
32 //     IMAGE_SCN_MEM_SYSHEAP - Obsolete   0x00010000

```

```

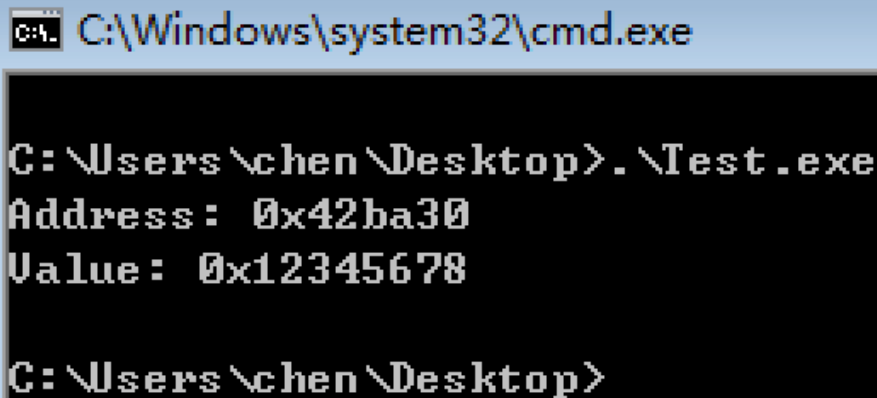
26 #define IMAGE_SCN_MEM_PURGEABLE 0x00020000
27 #define IMAGE_SCN_MEM_16BIT 0x00020000
28 #define IMAGE_SCN_MEM_LOCKED 0x00040000
29 #define IMAGE_SCN_MEM_PRELOAD 0x00080000
30
31 #define IMAGE_SCN_ALIGN_1BYTES 0x00100000 //
32 #define IMAGE_SCN_ALIGN_2BYTES 0x00200000 //
33 #define IMAGE_SCN_ALIGN_4BYTES 0x00300000 //
34 #define IMAGE_SCN_ALIGN_8BYTES 0x00400000 //
35 #define IMAGE_SCN_ALIGN_16BYTES 0x00500000 // Default
alignment if no others are specified.
36 #define IMAGE_SCN_ALIGN_32BYTES 0x00600000 //
37 #define IMAGE_SCN_ALIGN_64BYTES 0x00700000 //
38 #define IMAGE_SCN_ALIGN_128BYTES 0x00800000 //
39 #define IMAGE_SCN_ALIGN_256BYTES 0x00900000 //
40 #define IMAGE_SCN_ALIGN_512BYTES 0x00A00000 //
41 #define IMAGE_SCN_ALIGN_1024BYTES 0x00B00000 //
42 #define IMAGE_SCN_ALIGN_2048BYTES 0x00C00000 //
43 #define IMAGE_SCN_ALIGN_4096BYTES 0x00D00000 //
44 #define IMAGE_SCN_ALIGN_8192BYTES 0x00E00000 //
45 // Unused 0x00F00000
46
47 #define IMAGE_SCN_LNK_NRELOC_OVFL 0x01000000 // Section
contains extended relocations.
48 #define IMAGE_SCN_MEM_DISCARDABLE 0x02000000 // Section can be
discarded.
49 #define IMAGE_SCN_MEM_NOT_CACHED 0x04000000 // Section is not
cachable.
50 #define IMAGE_SCN_MEM_NOT_PAGED 0x08000000 // Section is not
pageable.
51 #define IMAGE_SCN_MEM_SHARED 0x10000000 // Section is
shareable.
52 #define IMAGE_SCN_MEM_EXECUTE 0x20000000 // Section is
executable.
53 #define IMAGE_SCN_MEM_READ 0x40000000 // Section is
readable.
54 #define IMAGE_SCN_MEM_WRITE 0x80000000 // Section is
writeable.

```


4 RVA与FOA的转换

想象一下，如果你想通过逆向的方式改变一个全局变量的初始值，该怎么做？首先我们可以写一个程序，输出一个全局变量的地址和值：

```
1  int a = 0x12345678;
2
3  int main() {
4      printf("Address: 0x%x \n", &a);
5      printf("Value: 0x%x \n", a);
6      getchar();
7      return 0;
8  }
```



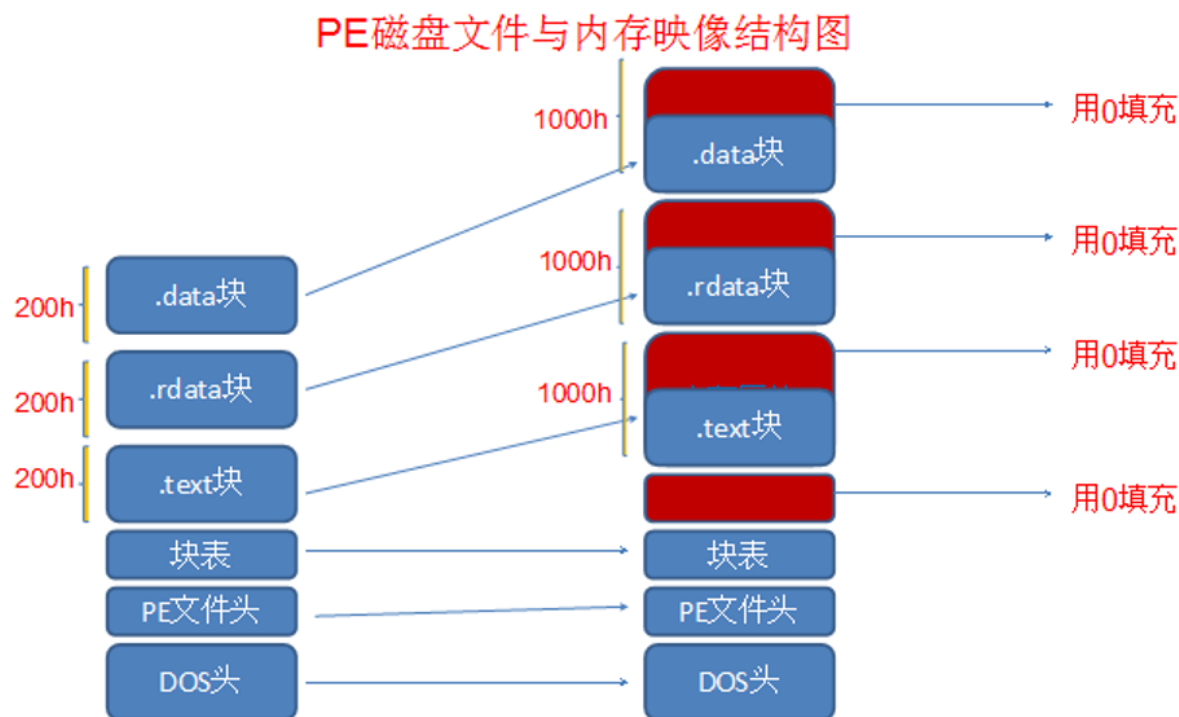
```
C:\Windows\system32\cmd.exe

C:\Users\chen\Desktop>.\Test.exe
Address: 0x42ba30
Value: 0x12345678

C:\Users\chen\Desktop>
```

我们运行程序可以看见相应的值，那么我们可以是否可以在文件中直接搜索对应的值然后修改呢？这种方法没有毛病，但是文件中也许会存在很多个0x12345678，你无法准确的知道哪一个才是全局变量；那么，又是否可以通过已经给出的这个地址0x42ba30直接去寻找呢？当然也是不行的，因为在之前章节的学习中我们了解到，PE文件有2种状态（动静态），在这2种状态下，文件的对齐方式会发生变化，所以当前的地址是PE文件运行

时（动态）的地址，你需要转换成在磁盘上（静态）的地址。



这两种状态的地址相互转换，我们可以称之为RVA与FOA的转换，**RVA就是相对虚拟地址，FOA就是文件偏移地址**；从RVA转换到FOA，就是从文件运行时（动态）的地址转换成在磁盘上（静态）的地址，按如下公式可以进行转换：

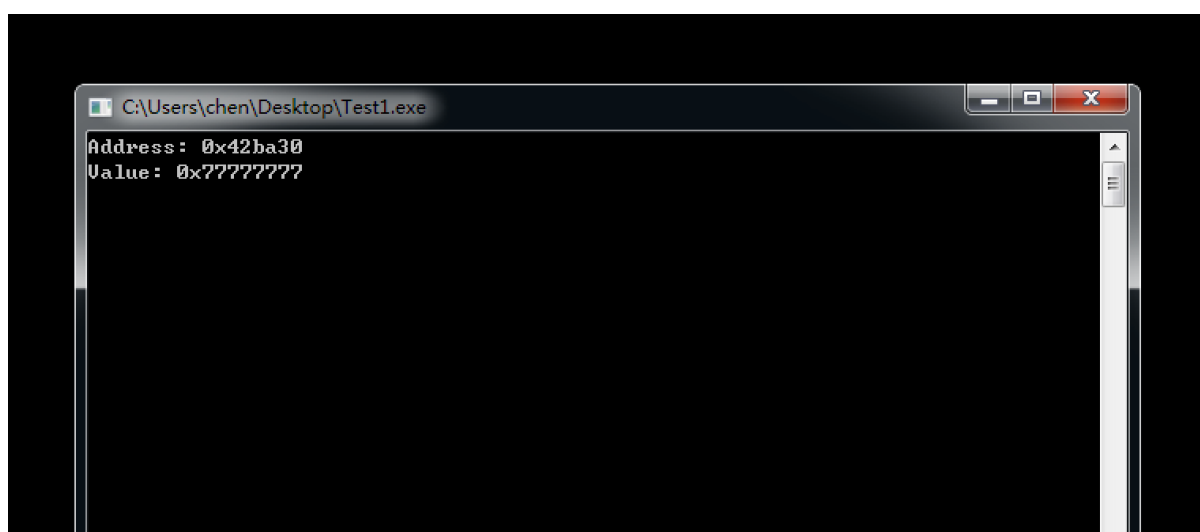
1. RVA地址由内存地址减去ImageBase地址（PE文件在内存中的开始位置是由扩展PE头中的ImageBase决定）；
2. 判断RVA地址是否位于PE头中：
 - a. 如果是，那么RVA等于FOA；
 - b. 如果不是，判断RVA位于哪个节：
 - i. 当满足RVA地址大于等于节.VirtualAddress和RVA地址小于节.VirtualAddress加上当前节内存对齐后的大小时，就表示RVA地址在该节中。
 - ii. RVA地址减去节.VirtualAddress等于差值，FOA地址就是根据节.PointerToRawData加上差值。

在一些较老的编译器中，编译出来的文件会区分文件对齐、内存对齐，但是在现在的编译器编译出来的程序，文件对齐与内存对齐时完全一样的，所以我们不用费这么大的周折，我们只需要算出RVA的值就可以得出FOA的值。

例如，在当前程序中就是这样，根据0x42BA30-0x400000（ImageBase）得出0x2BA30，其是RVA，也是FOA，直接使用Winhex打开找到：

Test.exe	Test.exe																	ANSI ASCII	
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F			
0002B900	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B910	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B920	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B930	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B940	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B950	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B960	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B970	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B980	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B990	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B9A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B9B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B9C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B9D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B9E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002B9F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BA00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BA10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BA20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BA30	78	56	34	12	F0	2D	40	00	01	00	00	00	00	00	00	00	xV4 8-0		
0002BA40	60	DF	42	00	00	00	00	00	60	DF	42	00	01	01	00	00	`BB`BB		
0002BA50	00	00	00	00	00	00	00	00	00	10	00	00	00	00	00	00			
0002BA60	00	00	00	00	00	00	00	00	00	00	00	00	02	00	00	00			
0002BA70	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BA80	00	00	00	00	00	00	00	00	00	00	00	00	02	00	00	00			
0002BA90	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BAA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BAB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BAC0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BAD0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BAE0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BAF0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BB00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BB10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BB20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BB30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BB40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BB50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BB60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BB70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
0002BB80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			

可以直接修改它然后保存运行，这时候你就会发现全局变量的值已经发生了改变：



5 PE空白区添加代码

现在我们有一个任务，需要在空白区添加一段代码（你也可以称之为Shellcode），并且在程序运行之前执行这段代码；首先我们要知道PE空白区是什么，PE空白区表示PE文件按照对齐方式之后多出来的部分，可以是节与节之间的空白区也可以是节表与节之间的空白区。

基本概念了解之后，我们来看下插入的代码，就是如下这个弹窗功能：

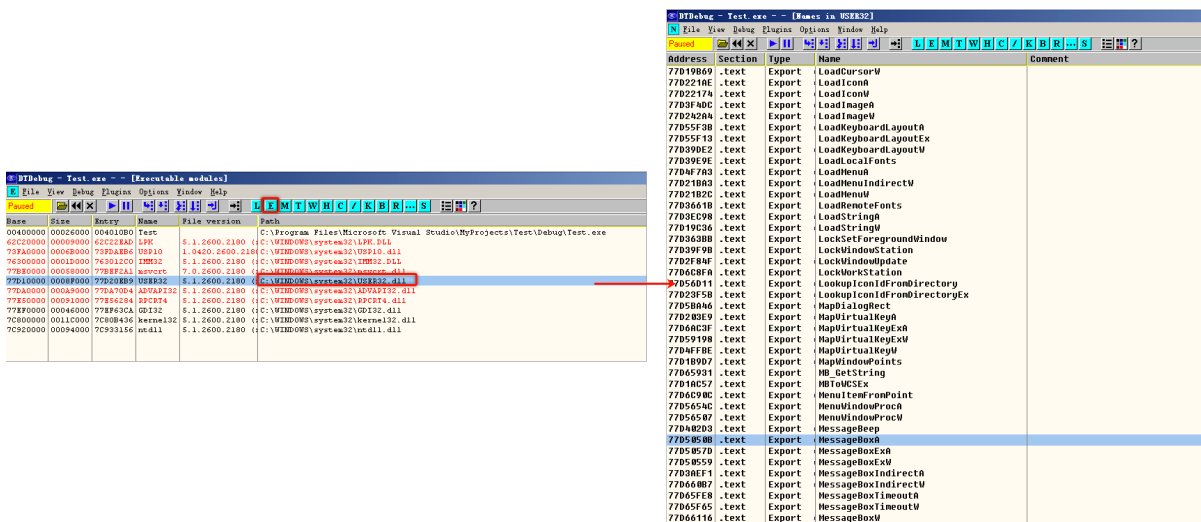
```
1  ::MessageBoxA(0,0,0,0);
```

那么如何让这段代码插入到空白区呢？直接插入代码很明显不可取，因为我们没有源文件，所以我们要插入的是之前所学习的硬编码，我们可以在VC6中下断点反编译查看这段代码对应的硬编码：

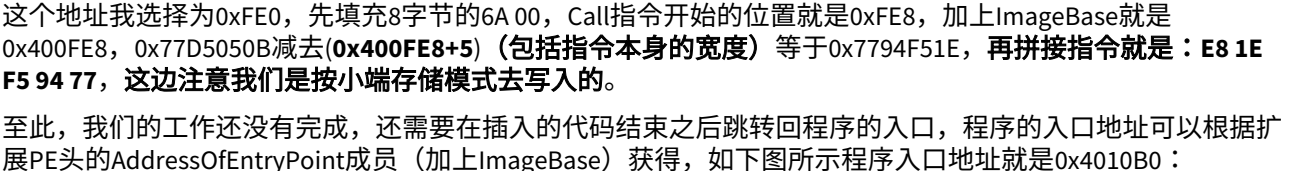
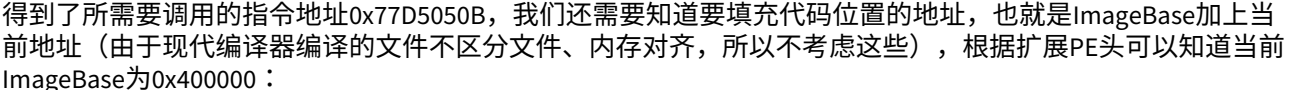
```
5:      ::MessageBoxA(0,0,0,0);
00401028 8B F4      mov     esi,esp
0040102A 6A 00      push    0
0040102C 6A 00      push    0
0040102E 6A 00      push    0
00401030 6A 00      push    0
00401032 FF 15 8C 42 00 call    dword ptr [__imp__MessageBoxA@16 (0042428c)]
00401038 3B F4      cmp     esi,esp
0040103A E8 31 00 00 00 call    __chkesp (00401070)
```

首先我们可以看见有4个传参：6A 00，接着就是调用函数MessageBox了，但是在这里的调用是需要遵循导入表的（后面章节会了解导入表内容），是一个间接调用，我们插入的时候需要修改成直接调用的形式，也就是：E8 00 00 00 00，E8后面的4字节表示一个偏移值，表示当前指令地址与需要调用指令地址之间的偏移，我们可以使用需要调用指令地址减去当前指令地址（包含本身的宽度）就得出这个偏移值。

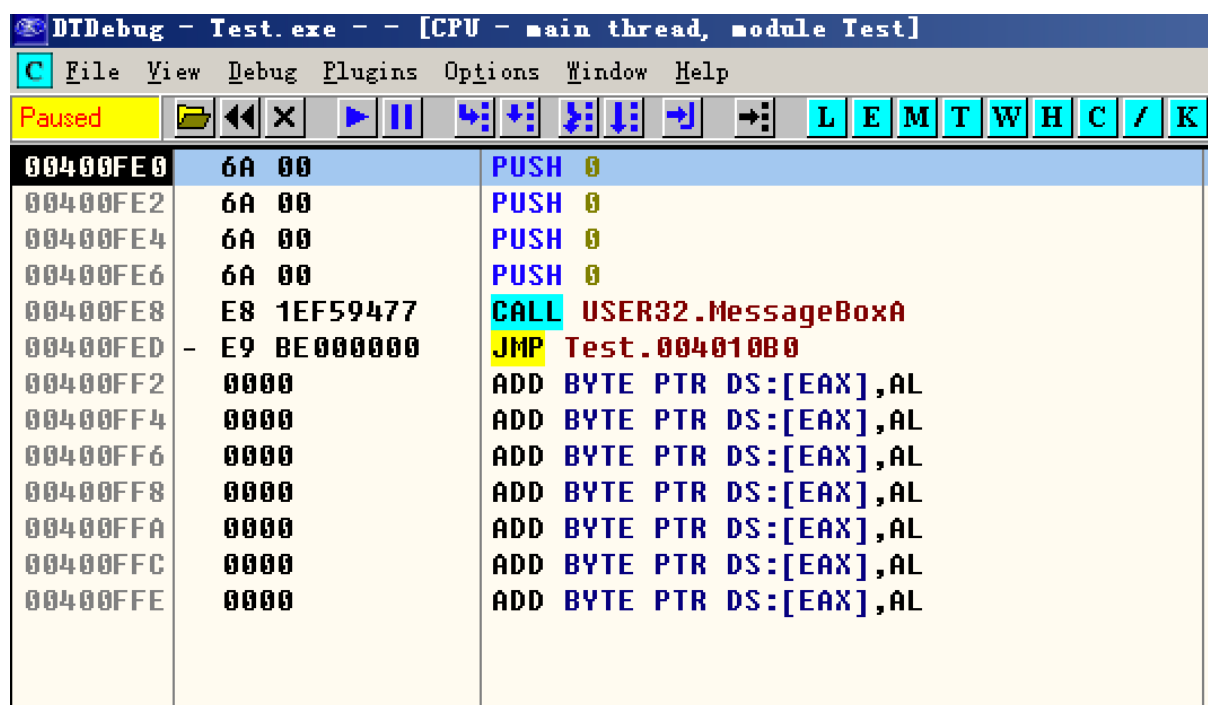
我们可以先找到MessageBoxA这个函数地址，在Win32的时候也了解过这个函数是技术是Windows操作系统的USER32.DLL提供的，我们可以在DTDebug中打开一个文件按如下图所示，找到对应的DLL文件，双击进去按Ctrl+N快捷键找到MessageBoxA函数的地址：



如果你觉得这样麻烦也可以考虑在VC6下面下断点跟进这个函数地址即可：



修改完成之后保存运行就会发现其先运行了我们插入的代码而后再进入程序真正的入口函数代码：



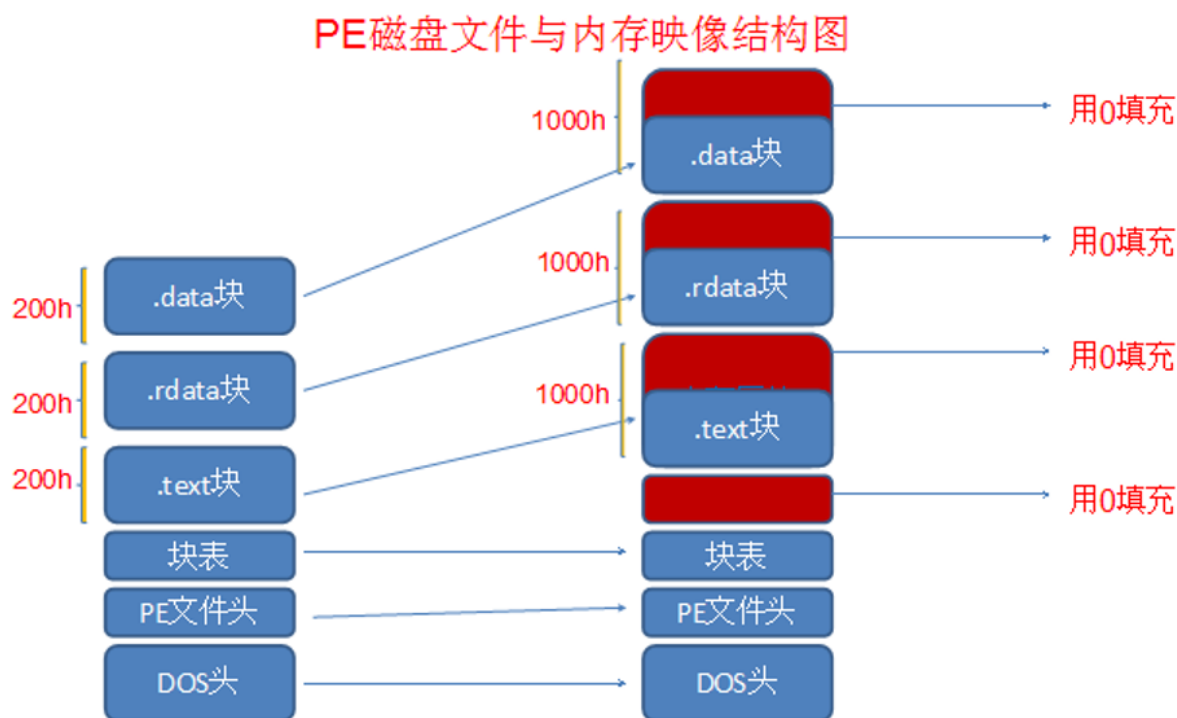
至此，我们就完成了在空白区添加代码并执行的任务了，最后需要注意的是我们这里所插入的代码只能在本机去运行，不能在其他机器上使用，因为这不是一个标准的Shellcode（后续章节会了解Shellcode）。

6 节操作

以下操作中用的PE文件建议自行寻找一个再去实验。

6.1 扩大节

在上一章节中我们可以在任意空白区添加自己的代码，但如果添加的代码比较多，空白区不够怎么办？这时候就需要扩大节，节有很多个，我们应该扩大哪一个节呢？想象一下如果你现在扩大第一个节，那么其他节的偏移量之类的属性都需要修改，这样很麻烦，所以我们可以选择扩大最后一个节，这样就不用修改其他节的属性了。



如下是节表成员的数据结构：

```

1  #define IMAGE_SIZEOF_SHORT_NAME 8
2  typedef struct _IMAGE_SECTION_HEADER {
3      BYTE    Name[IMAGE_SIZEOF_SHORT_NAME]; // ASCII字符串（节名），可自定义，
        只截取8个字节，可以8个字节都是名字
4      union { // Misc, 双字，是该节在没有对齐前的真实尺寸，该值可以不准确
5          DWORD    PhysicalAddress; // 真实宽度，这两个值是一个联合结构，可以使用其
        中的任何一个
6          DWORD    VirtualSize; // 一般是取后一个
7      } Misc;
8      DWORD    VirtualAddress; // 在内存中的偏移地址，加上ImageBase才是在内存中的真
        正地址
9      DWORD    SizeOfRawData; // 节在文件中对齐后的尺寸
10     DWORD    PointerToRawData; // 节区在文件中的偏移

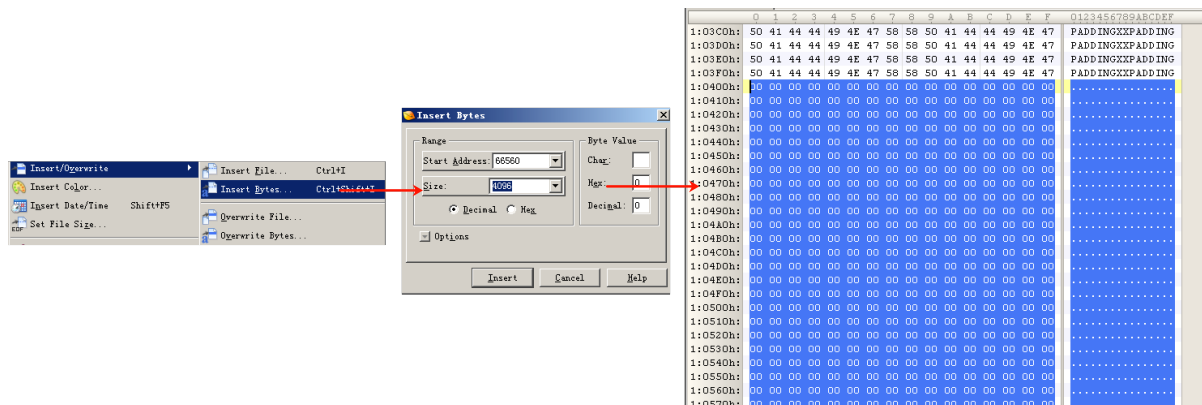
```

```

11     DWORD   PointerToRelocations; // 调试相关
12     DWORD   PointerToLinenumbers; // 调试相关
13     WORD    NumberOfRelocations; // 调试相关
14     WORD    NumberOfLinenumbers; // 调试相关
15     DWORD   Characteristics; // 节的属性
16 } IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;

```

我们想要扩大节就需要修改这几个部分：SizeOfRawData、VirtualSize。接下来我们就来扩大节，首先需要分配一块新的空间，这块空间的大小取决于你需要的代码大小，在这里就定位0x1000，也就是4096字节，如下图所示使用010 Editor在文件末尾插入字节（这样也就不会打乱其他结构的偏移）：



然后找到最后一个节表成员，修改SizeOfRawData和VirtualSize成员的值，这个值是要取SizeOfRawData和VirtualSize成员当前值的最大值进行内存对齐之后的值加上我们插入宽度0x1000：

2E 64 61 74	61 00 00 00	A8 1B 00 00	00 90 00 00	.data..."
VirtualSize 00 7C 00	SizeOfRawData 00 00 00	00 00 00	00 00 00
00 00 00 00	40 00 00 00	00 2E 72 73	72 63 00 00 00@..À.rsrc...
30 7F 00 00	00 B0 00 00	00 80 00 00	00 84 00 00	0....°...€.....
00 00 00 00	00 00 00 00	00 00 00 00	40 00 00 40@..@
5B 45 21 41	58 00 00 00	73 45 21 41	65 00 00 00	[E!AX...sE!Ae...

如上图所示两个成员中值最大的为SizeOfRawData，SizeOfRawData是按照文件对齐的，在这里我们还需要将SizeOfRawData的值按内存对齐，首先来看下内存对齐（SectionAlignment）和文件对齐（FileAlignment）的值：

00E0h:	50 45 00 00	4C 01 03 00	C3 7C 10 41	00 00 00 00	PE..L...Ã .A....
00F0h:	00 00 00 00	E0	SectionAlignment 00 00 00 00	FileAlignment 00 00 00 00	..à.....x..
0100h:	00 88 00 00	00 00 00 00	10 30 01 00	00 10 00 000.....
0110h:	00 90 00 00	00 00 00 01	00 10 00 00	00 02 00 00
0120h:	05 00 01 00	05 00 01 00	04 00 00 00	00 00 00 00

可以看见，在这里两个值不一样，所以当前SizeOfRawData是按文件对齐的值直接添加是不可取的，我们还需要按内存对齐，可以使用如下公式进行计算：

- 1
$$\lceil \text{SizeOfRawData}(0x8000) / \text{SectionAlignment}(0x1000) \rceil = \text{ResultA}(0x8) //$$

0x8000除以0x1000向上取整(符号⌈)
- 2
$$\text{ResultA}(0x8) * \text{SectionAlignment}(0x1000) = \text{ResultB}(0x8000)$$

最终我们得出按内存对齐的值为0x8000，然后我们要将SizeOfRawData和VirtualSize成员的值修改为0x9000（0x8000+0x1000）。

00 00 00 00	40 00 00 C0	2E 72 73 72	63 00 00 00@...â.rsrc...
00 90 00 00	00 B0 00 00	00 90 00 00	00 84 00 00°.....
00 00 00 00	00 00 00 00	00 00 00 00	40 00 00 40@...@

接着，如果你要让这个节中存放代码并且需要执行的话，就需要修改节的属性；然后我们需要去修改扩展PE头中的SizeOfImage成员，该成员表示在内存中整个PE文件映射的大小，如下图是当前的值：

00E0h:	50 45 00 00	4C 01 03 00	C3 7C 10 41	00 00 00 00	PE..L...Ï .A....
00F0h:	00 00 00 00	E0 00 0F 01	0B 01 07 0A	00 78 00 00à.....x..
0100h:	00 88 00 00	00 00 00 00	9D 73 00 00	00 10 00 00	..^.....s.....
0110h:	00 90 00 00	00 00 00 01	00 10 00 00	00 02 00 00
0120h:	05 00 01 00	05 00 01 00	04 00 00 00	00 00 00 00
0130h:	00 30 01 00	00 04 00 00	59 79 01 00	02 00 00 80	.0.....ÿy.....€
0140h:	00 00 04 00	00 10 01 00	00 00 10 00	00 10 00 00
0150h:	00 00 00 00	10 00 00 00	00 00 00 00	00 00 00 00

在之前的章节里一家了解过了SizeOfImage可以比实际值要大并且这个值默认情况下本身就是与内存对齐之后的结果，所以你在原基础上加0x1000即可，这样我们就完成了扩大节的操作。

6.1.1 插入代码执行

那么，在扩大节之后又该如何去插入自己的代码调用呢？如果按照之前插入空白区的方法，根据当前的指令地址再加上ImageBase去调用，很明显这是不可取的，因为当前文件与内存对齐是不一致的，我们想要去掉用就要知道在内存中当前指令的地址。

而这个，我们可以使用原SizeOfImage的值获得，因为原SizeOfImage的值表示PE文件在内存中展开的大小，我们单独添加的空间就在PE文件的末尾，所以可以根据SizeOfImage来知道在添加空间中的指令地址；例如如下图所示众标红部分在内存中的地址范围应为ImageBase加原SizeOfImage与指令的宽度，最终的地址范围就是：0x1013000 - 0x1013003

[illegible]

接着，我们只需要添加自己的指令进去即可，这边还是套用在空白区时所使用的指令MessageBoxA(0x77D5050B)，并计算出偏移量：

1 0x77D5050B (MessageBoxA) - 0x1013000 (指令开始地址) - 13 (指令本身长度) = 0x76D3D4F8

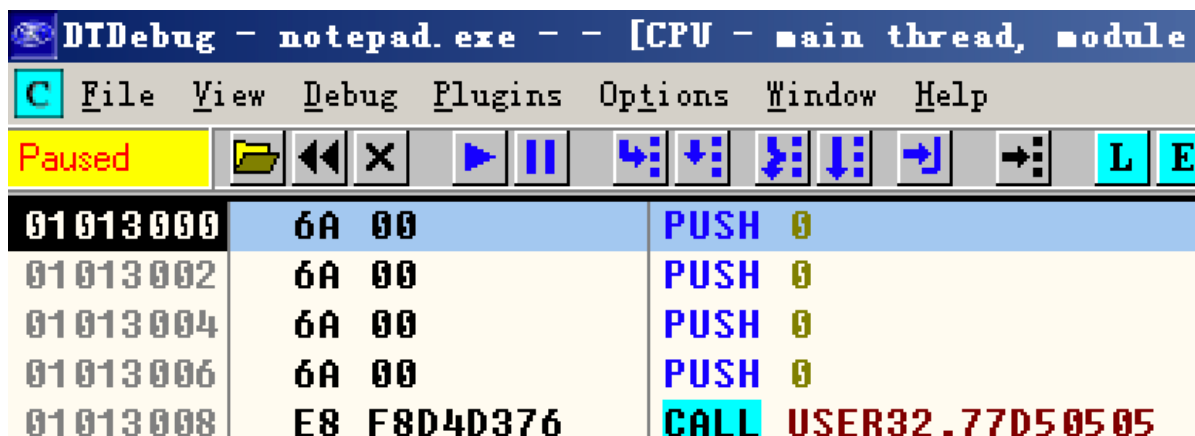
再拼接上指令（别忘了小端存储模式）插入即可（这里的指令进行了简化，省略了JMP跳回原AddressOfEntryPoint地址部分）：

```
1 6A 00 6A 00 6A 00 6A 00 E8 F8 D4 D3 76
```

最后我们需要修改一下OEP（这里的OEP是Original EntryPoint，也就是原始入口点，没有加壳以及其他修改的情况下，AddressOfEntryPoint就是OEP，但是如果存在修改/加壳的情况，AddressOfEntryPoint只能称为是EP，因为不是原始入口点，你需要自己去寻找了）这里修改也按照之前得出的指令地址去除ImageBase的值进行修改：

50 45 00 00	4C 01 03 00	C3 7C 10 41	00 00 00 00	PE..L...ä .A....
00 00 00 00	E0 00 0F 01	0B 01 07 0A	00 78 00 00à.....x..
00 88 00 00	00 00 00 00	00 30 01 00	00 10 00 00	.^......O.....
00 90 00 00	00 00 00 01	00 10 00 00	00 02 00 00
05 00 01 00	05 00 01 00	04 00 00 00	00 00 00 00
00 40 01 00	00 04 00 00	59 79 01 00	02 00 00 80	.@.....Yy.....€

接着保存运行即可：



6.2 新增节

上一章节中我们知道当想插入的代码过多的时候，空白区不够用的情况下，我们可以使用扩大节的方法扩大最后一个节，然后在里面插入自己的代码；这样的方法虽然有效，但还是有一些弊端，比如最后一个节的属性会被修改，插入的代码会与原节的数据混合在一块。所以，我们可以新增节，在新增的节里添加自己的数据。

新增节，首先判断是否有足够的空间可以添加一个节表成员（40字节），我们可以找到一个PE文件来看一下它的节表部分是否有足够多的空白区，首先我们根据标准PE头的第二个成员（NumberOfSections）知道有4个节：

00E0h:	00 00 00 00	00 00 00 00	50 45 00 00	4C 01 04 00PE..L..
00F0h:	50 C8 98 41	00 00 00 00	00 00 00 00	E0 00 0F 01	PE~A.....à..
0100h:	0B 01 06 00	00 A0 01 00	00 C0 00 00	00 00 00 00Ä.....

接着我们在扩展PE头之后找到节表，我们可以看见在这个节表之后有40字节的空白区让我们添加新的节表成员，我们可以选择复制一份".text"节表成员作为新增的成员，这是因为我们要在节数据中添加代码，而".text"就是存放代码的，所以我们直接复制过来就不需要修改节属性了：

01E0h:	2E 74 65 78	74 00 00 00	22 97 01 00	00 10 00 00	.text..."-.....
01F0h:	00 A0 01 00	00 10 00 00	00 00 00 00	00 00 00 00
0200h:	00 00 00 00	20 00 00 60	2E 72 64 61	74 61 00 00`.rdata..
0210h:	9E 30 00 00	00 B0 01 00	00 40 00 00	00 B0 01 00	ž0...°...@...°..
0220h:	00 00 00 00	00 00 00 00	00 00 00 00	40 00 00 40@..@
0230h:	2E 64 61 74	61 00 00 00	D8 37 00 00	00 F0 01 00	.data...07...đ..
0240h:	00 40 00 00	00 F0 01 00	00 00 00 00	00 00 00 00	.@...đ.....
0250h:	00 00 00 00	40 00 00 C0	2E 72 73 72	63 00 00 00@..Ä.rsrc...
0260h:	D8 3F 00 00	00 30 02 00	00 40 00 00	00 30 02 00	0?...0...@...0..
0270h:	00 00 00 00	00 00 00 00	00 00 00 00	40 00 00 40@..@
0280h:	2E 74 65 78	74 00 00 00	22 97 01 00	00 10 00 00	.text..."-.....
0290h:	00 A0 01 00	00 10 00 00	00 00 00 00	00 00 00 00
02A0h:	00 00 00 00	20 00 00 60	00 00 00 00	00 00 00 00

因为我们增加了一个节，所以需要在标准PE头的第二个成员（NumberOfSections）中加1：

00E0h:	00 00 00 00	00 00 00 00	50 45 00 00	4C 01 05 D0PE..L...
00F0h:	50 C8 98 41	00 00 00 00	00 00 00 00	E0 00 0F 01	PË~A.....à...
0100h:	0B 01 06 00	00 A0 01 00	00 C0 00 00	00 00 00 00Ä.....

为了方便添加节我们还需要修改下最后一个节表成员的属性，将其真实大小（VirtualSize）修改成文件对齐之后的大小（SizeOfRawData）：

0250h:	00 00 00 00	40 00 00 C0	2E 72 73 72	63 00 00 00@..Ä.rsrc...
0260h:	00 40 00 00	00 30 02 00	00 40 00 00	00 30 02 00	.@...0...@...0..
0270h:	00 00 00 00	00 00 00 00	00 00 00 00	40 00 00 40@..@

接着修改添加的节表成员的属性：名字、真实大小（0x1000）、文件对齐之后的大小（0x1000）、内存中的偏移（第4个节的偏移地址0x23000+其数据大小0x4000）、文件中的偏移：

0280h:	2E 74 74 74	74 00 00 00	00 10 00 00	00 70 02 00	.tttt.....p..
0290h:	00 10 00 00	00 70 02 00	00 00 00 00	00 00 00 00p.....
02A0h:	00 00 00 00	20 00 00 60	00 00 00 00	00 00 00 00

然后需要修改一下SizeOfImage的大小，加上0x1000即可：

0110h:	D7 83 01 00	00 10 00 00	00 B0 01 00	00 00 40 00	*f.....°...@..
0120h:	00 10 00 00	00 10 00 00	04 00 00 00	00 00 00 00
0130h:	04 00 00 00	00 00 00 00	00 80 02 00	00 10 00 00€.....
0140h:	00 00 00 00	02 00 00 00	00 00 10 00	00 10 00 00

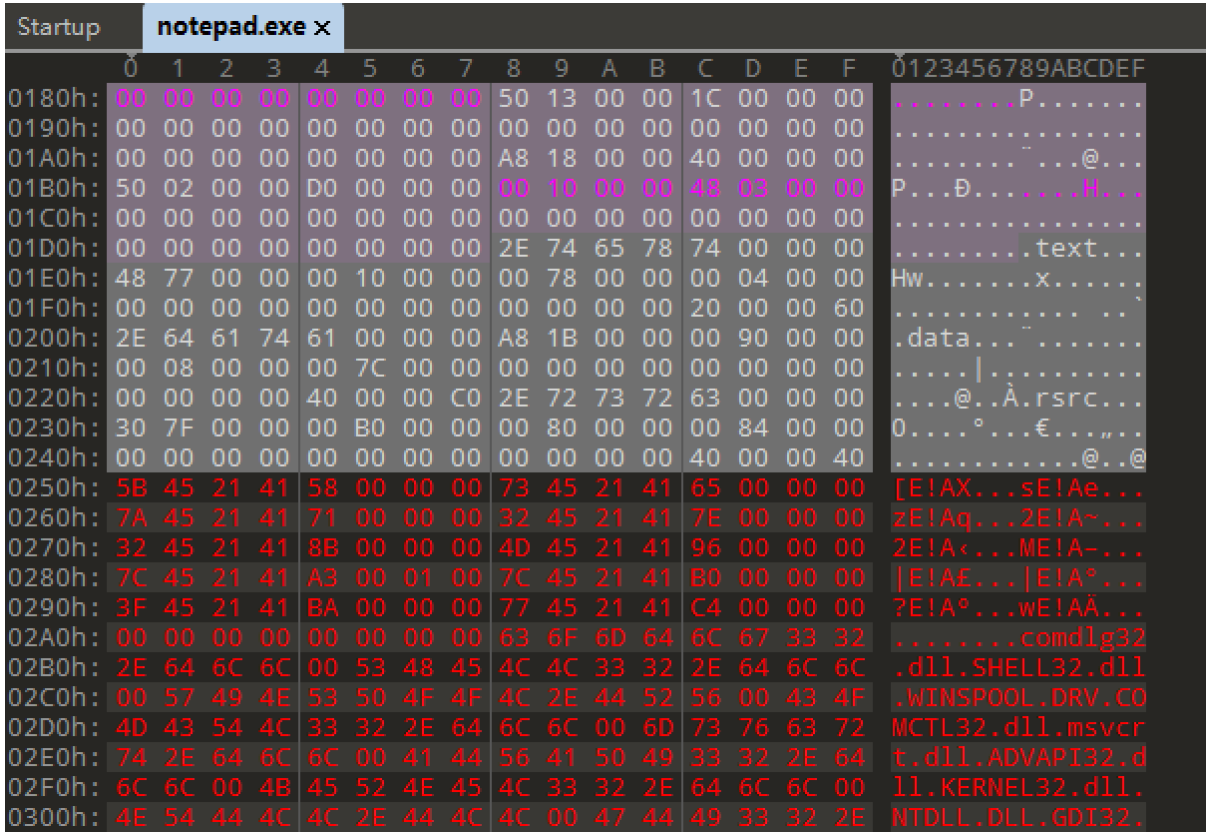
最后在文件末尾添加对应节的数据（0x1000 → 4096个字节）：

Startup ×	IPMSG(2).exe* ×															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2:6FF0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2:7000h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2:7010h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2:7020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2:7030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2:7040h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2:7050h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2:7060h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2:7070h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

这样我们就完成了新增节的所有步骤，你可以基于这个基础再去插入代码执行。

6.3 合并节

上一章中了解到新增节需要在节表之后至少有40个字节的空白区给我们去新增，但并不是所有的程序都可以满足这个条件，如下图所示的程序在节表之后的数据是编译器填充的，这些数据我们并不能覆盖：



6.3.1 按内存对齐展开

那么这样我们如何新增节呢？如果PE的DOS块没有被占用的情况下，我们完全可以将PE头向上提升，替换DOS块的部分，这样就可以多出一块空间出来，但是如果当前PE文件的DOS块被占用，这种方法显然就不可取了。所以，我们想要实现新增节可以采用合并其他节的方法，给我们新增的节表成员留出空间。

我们想要合并节，首先要考虑到当前PE文件的文件对齐和内存对齐是否一致，在当前PE文件来看是不一致的，所以直接合并肯定是不行的，我们需要先将节进行内存对齐展开。首先，需要知道当前PE文件的内存对齐的值：

DWORD SectionAlignment	1000h	118h	4h	Fg:	Bg:
DWORD FileAlignment	200h	11Ch	4h	Fg:	Bg:

如上图所示内存对齐就是0x1000，接着找到节表成员，将其对应成员属性按如下公式代入计算：

▼ struct IMAGE_SECTION_HEADER Section...		1D8h	78h	Fg:	Bg:	
▼ struct IMAGE_SECTION_HEADER Secti...	.text	1D8h	28h	Fg:	Bg:	
> BYTE Name[8]	.text	1D8h	8h	Fg:	Bg:	can end without zero
> union Misc		1E0h	4h	Fg:	Bg:	
DWORD VirtualAddress	1000h	1E4h	4h	Fg:	Bg:	
DWORD SizeOfRawData	7800h	1E8h	4h	Fg:	Bg:	
DWORD PointerToRawData	400h	1ECh	4h	Fg:	Bg:	
DWORD PointerToRelocations	0h	1F0h	4h	Fg:	Bg:	
DWORD PointerToLinenumbers	0	1F4h	4h	Fg:	Bg:	
WORD NumberOfRelocations	0	1F8h	2h	Fg:	Bg:	
WORD NumberOfLinenumbers	0	1FAh	2h	Fg:	Bg:	
> struct SECTION_CHARACTERISTICS...		1FCh	4h	Fg:	Bg:	

```

1  ⌈(max(SizeOfRawData, Misc) / SectionAlignment) = ResultA // 取
2  SizeOfRawData和Misc之间的最大值除以内存对齐的值，最后的结果向上取整(符号⌈)
3
4  // 代入公式计算
5  max(0x7800, 0x7748) = 0x7800
6  ⌈(0x7800 / 0x1000) = ⌈(0x7 余 0.8) = 0x8
7  0x8 x 0x1000 = 0x8000

```

按公式得出的结果替换原先的SizeOfRawData、Misc：

01D0h:	00 00 00 00	00 00 00 00	2E 74 65 78	74 00 00 00text...
01E0h:	00 80 00 00	00 10 00 00	00 80 00 00	00 04 00 00	.€.€.€.....
01F0h:	00 00 00 00	00 00 00 00	00 00 00 00	20 00 00 60`

接着我们需要将内存对齐后的值减去原SizeOfRawData的值，得出一个差值：

```

1  0x8000 - 0x7800 = 0x800

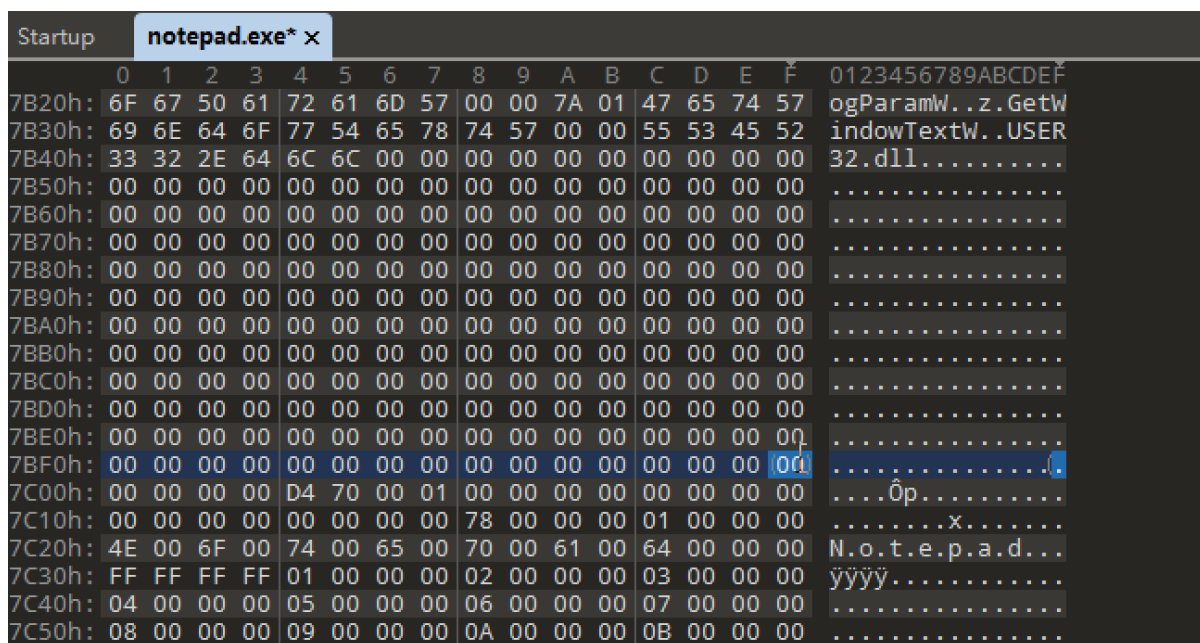
```

这个值则用于增加节的空间，找到该节的末尾，也就是PointerToRawData的值加上原SizeOfRawData的值：

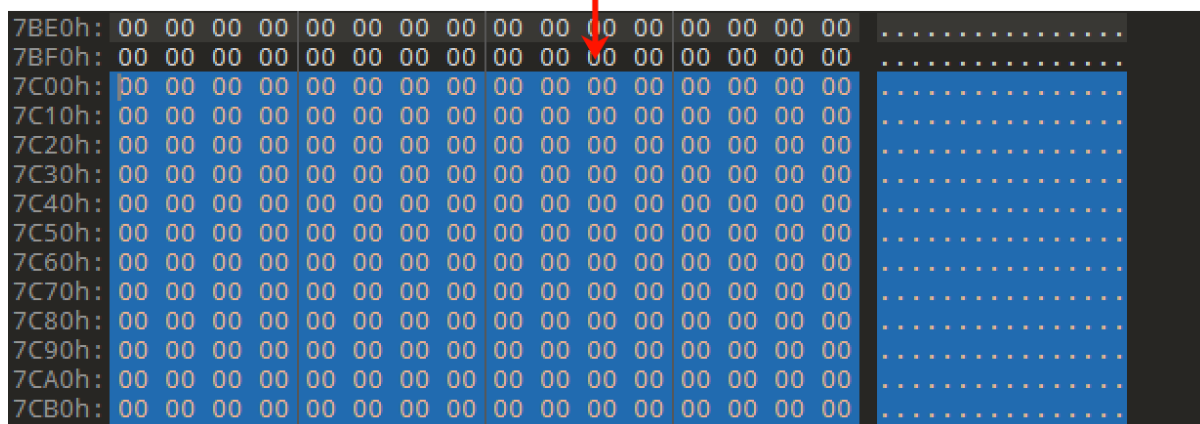
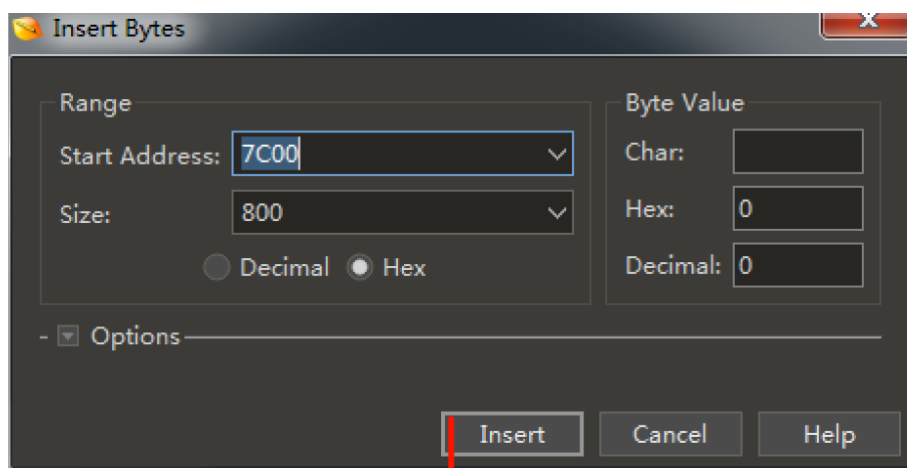
```

1  0x400 + 0x7800 = 0x7C00

```



在这开始添加0x800字节的空间：



由于我们这里修改了第一个节的大小并添加了空间，所以之后的节的文件偏移（PointerToRawData）要对应添加上增加的差值：

[illegible]

按照这样的步骤以此类推将所有的节全部按照内存对齐的方式进行修正（由于SizeOfImage本身就是按内存对齐之后的大小，所以其的值会比实际值大，我们这里添加的空间从理论上来说也不会超出这个值，甚至我们添加之后的大小是等于这个值的，也就无需更改）。

6.3.2 手动合并节

接下来我们需要计算出所有节的大小，这里我们可以使用SizeOfImage减去SizeOfHeaders内存对齐之后的大小，得出的结果0x12000就是所有节的大小（你也可以选择计算所有节展开后的大小和，这时候你就会发现这里的大小是一模一样的）：

DWORD SizeOfImage	13000h	130h	4h	Fg:	Bg:
DWORD SizeOfHeaders	400h	134h	4h	Fg:	Bg:

那么这个值我们就对应给到第一个节表成员中的SizeOfRawData、Misc：

01D0h:	00 00 00 00	00 00 00 00	2E 74 65 78	74 00 00 00text...
01E0h:	00 20 01 00	00 10 00 00	00 20 01 00	00 04 00 00
01F0h:	00 00 00 00	00 00 00 00	00 00 00 00	20 00 00 60

然后由于我们合并了其他的节，但是其他节的Characteristics（属性）是不一样的，我们合并了也要将其他节的属性添加进来，这里可以选择使用或运算，如下图所示就是进行或运算之后的结果0xE0000060：

		20 00 00 60	
		40 00 00 C0	
		40 00 00 40	
0010	0000	0000	0000 0000 0000 0110 0000
0100	0000	0000	0000 0000 0000 1100 0000
0110	0000	0000	0000 0000 0000 1100 0000 (HEX: 60 00 00 E0)
0110	0000	0000	0000 0000 0000 1100 0000
0100	0000	0000	0000 0000 0000 0100 0000
0110	0000	0000	0000 0000 0000 1100 0000 (HEX: 60 00 00 E0)

将这个结果给到第一个节表成员的Characteristics即可。最后因为所有节合并为一了，所以我们要修改NumberOfSections的值为1：

00E0h:	50 45 00 00	4C 01 01 00	3 7C 10 41	00 00 00 00	PE..L...A .A...
00F0h:	00 00 00 00	E0 00 0F	01 00 01 07	00 70 00 00	...à.....x..
0100h:	00 88 00 00	00 00 00		S.....
0110h:	00 90 00 00	00 00 00		
0120h:	05 00 01 00	05 00 01		

其他的节表成员可以用00填充，这一块空间就可以给我们添加自己的节表成员：

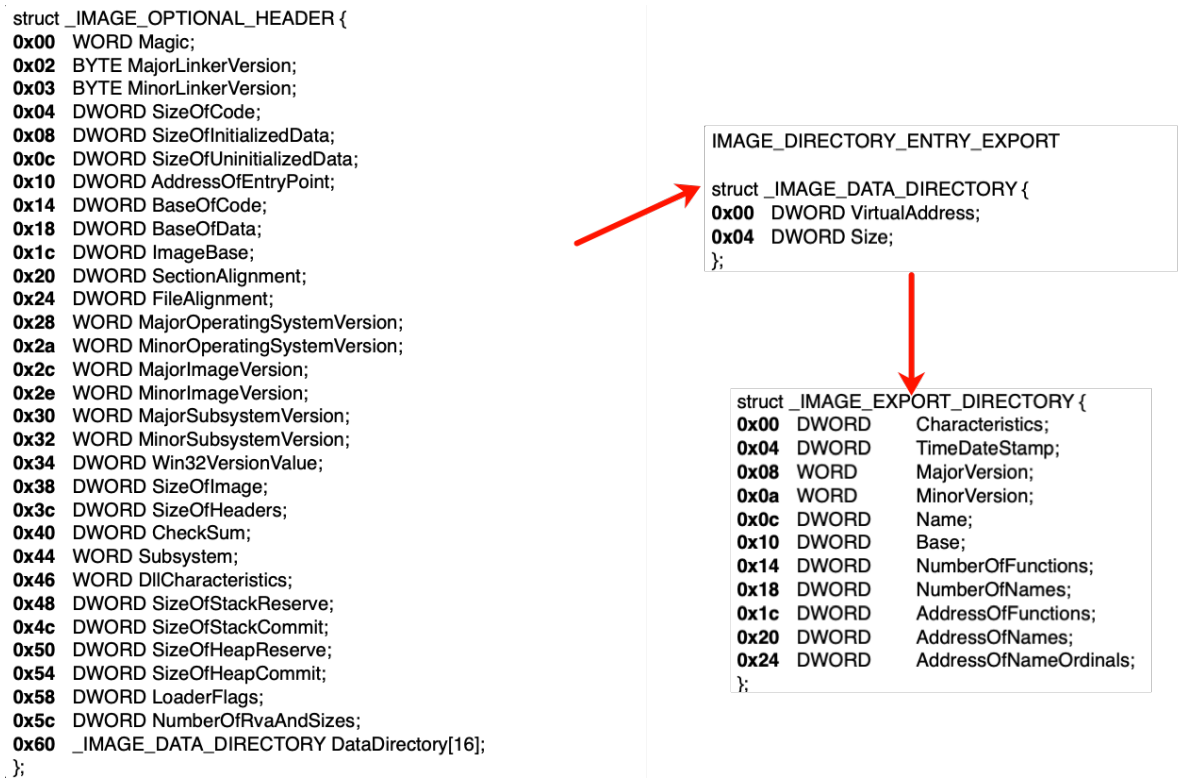
01D0h:	00 00 00 00	00 00 00 00	2E 74 65 78	74 00 00 00text...
01E0h:	00 20 01 00	00 10 00 00	00 20 01 00	00 04 00 00
01F0h:	00 00 00 00	00 00 00 00	00 00 00 00	60 00 00 E0à
0200h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0210h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0220h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0230h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0240h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0250h:	5B 45 21 41	58 00 00 00	73 45 21 41	65 00 00 00	[E!AX...sE!Ae...
0260h:	7A 45 21 41	71 00 00 00	32 45 21 41	7E 00 00 00	zE!Aq...2E!A~...

7 导出表

一个可执行程序是由多个PE文件组成，这些PE文件依靠导入表、导出表进行联系，导出表存储着PE文件提供给别人使用的函数列表，导入表则存储着PE文件所需要用到的PE文件列表。从PE文件的角度去看，任何PE文件都可以有导入、导出表，从一般情况来看，EXE文件不会提供导出表，也就是不会提供给别人使用的函数，但这并不代表不可以提供。

7.1 定位导出表

在PE格式图中，扩展PE头最后一个成员是结构体数组，在这个结构体数组里面有16个结构体，第一个结构体就是导出表相关的信息，它有2个成员，一个表示导出表的地址，一个表示导出表的大小。如下图所示中的 **_IMAGE_EXPORT_DIRECTORY**，就是PE导出表的结构：



我们可以自己编写、发布一个DLL，导出表这样写：

1	EXPORTS
2	
3	Add @10
4	Sub @12
5	Div @13 NONAME
6	Mul @15

接着我们可以手动去定位一下导出表的位置，先找到扩展PE头的最后一个成员结构体数组：

0140h:	00 00 00 00	10 00 00 00	B0 AD 02 00	7F 01 00 00€-.....
0150h:	00 F0 02 00	28 00 00 00	00 00 00 00	00 00 00 00	.ä..(.....
0160h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0170h:	00 00 03 00	08 12 00 00	00 90 02 00	1C 00 00 00
0180h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0190h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
01A0h:	00 00 00 00	00 00 00 00	AC F1 02 00	84 01 00 00~ñ.....
01B0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
01C0h:	00 00 00 00	00 00 00 00	2E 74 65 78	74 00 00 00text...
01D0h:	60 72 02 00	00 10 00 00	00 80 02 00	00 10 00 00	^r.....€.....
01E0h:	00 00 00 00	00 00 00 00	00 00 00 00	20 00 00 60

然后找到该结构体数组的第一个结构体，里面就包含了导出表的地址和大小：

1	VirtualAddress: 0x0002AD80
2	Size: 0x0000017F

这个地址是RVA，它实际上表示的是相对虚拟地址，我们需要将其转为FOA，也就在文件中的偏移地址，由于在当前PE文件中文件对齐和内存对齐是一样的，即RVA等于FOA，所以我们也不需要转换：

DWORD SectionAlignment	1000h	108h	4h	Fg:	Bg:
DWORD FileAlignment	1000h	10Ch	4h	Fg:	Bg:

接着跟进这个地址就可以找到导出表了：

2:AD80h:	00 00 00 00	AB 08 D4 61	00 00 00 00	D2 AD 02 00«.ôa....ô-
2:AD90h:	0A 00 00 00	06 00 00 00	03 00 00 00	A8 AD 02 00-...
2:ADA0h:	C0 AD 02 00	CC AD 02 00	0A 10 00 00	00 00 00 00	À-...î-.....
2:ADB0h:	0F 10 00 00	19 10 00 00	00 00 00 00	14 10 00 00
2:ADC0h:	DE AD 02 00	E2 AD 02 00	E6 AD 02 00	00 00 05 00	p-...â-...æ-.....
2:ADD0h:	02 00 44 4C	4C 44 65 6D	6F 2E 64 6C	6C 00 41 64	..DLLDemo.dll.Ad
2:ADE0h:	64 00 4D 75	6C 00 53 75	62 00 00 00	00 00 00 00	d.Mul.Sub.....
2:ADF0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AE00h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AE10h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AE20h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AE30h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AE40h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AE50h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AE60h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AE70h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AE80h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AE90h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AEA0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AEB0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AEC0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AED0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AEE0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
2:AEF0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

接着我们来看下导出表的结构，你会发现我们实际上找到的导出表，其整体大小是大于如下结构（40字节），这是因为在这张表中还包含了3个子表，也就是如下结构体的最后三个成员：

1	typedef struct _IMAGE_EXPORT_DIRECTORY {
---	--

```

2      DWORD   Characteristics; // 未使用
3      DWORD   TimeDateStamp; // 时间戳, 表示当前PE文件 (DLL) 编译时的时间
4      WORD    MajorVersion; // 未使用
5      WORD    MinorVersion; // 未使用
6      DWORD   Name; // 当前导出表文件名字符串的地址
7      DWORD   Base; // 导出函数起始序号
8      DWORD   NumberOfFunctions; // 所有导出函数的个数
9      DWORD   NumberOfNames; // 以函数名字导出的函数个数
10     DWORD   AddressOfFunctions; // RVA, 导出函数地址表
11     DWORD   AddressOfNames; // 导出函数名称表RVA
12     DWORD   AddressOfNameOrdinals; // 导出函数序号表RVA
13 } IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;

```

所以排除子表, 我们的结构体就只有40字节, 接着我们从第4个成员, 逐步解析。

7.2 解析导出表成员

7.2.1 Name

用于表示当前导出表文件名, 这是一个字符串, 所以我们只需要找到00即停止寻找:

```

1      0x0002ADD2 -> DLLDemo.dll

```

2:ADC0h:	(DE) AD 02 00	E2 AD 02 00	E6 AD 02 00	00 00 05 00	p}...â-...æ-.....
2:ADD0h:	02 00 44 4C	4C 44 65 6D	6F 2E 64 6C	6C 00 41 64	..DLLDemo.dll.Ad
2:ADE0h:	64 00 4D 75	6C 00 53 75	62 00 00 00	00 00 00 00	d.Mul.Sub.....

7.2.2 Base

用于表示当前导出函数的起始序号, 也就是你定义的.DEF中最小的那个序号:

```

1      0x0000000A -> 10

```

7.2.3 NumberOfFunctions

用于表示所有导出函数的个数, 这个数是按照你定义的.DEF中的序号算的, **如果在导出时不按序号顺序导出, 则空余位也计入总和**, 例如如下所示, 我们实际上只有4个导出函数, 但是因为导出函数的序号没有按照顺序来进行导出, 导致计算的时候将空缺的也算进去了:

```

1      0x00000006 -> 6

```

7.2.4 NumberOfNames

用于表示以函数名字导出的函数个数，如果你在定义的.DEF中设置了导出函数的属性为NONAME则就是以序号的方式导出：

1	0x00000003 -> 3
---	-----------------

7.2.5 AddressOfFunctions

用于表示导出函数地址表的地址，这里的地址是RVA，如果当前PE文件的文件对齐与内存对齐不一致，需要转换为FOA，我们这边是一样的，所以不需要进行转换；通过NumberOfFunctions知道有6个导出函数，所以从该地址开始依次寻找6个4字节的地址即可，我们可以很清楚的看见这里的地址有些就是00填充的：

1	0x0002ADA8 ->
2	0x0000100A
3	0x0000100F
4	0x00000000
5	0x00001019
6	0x00000000
7	0x00001014

2:ADA0h:

C0 AD 02 00 CC AD 02 00 0A 10 00 00 00 00 00 00

Ä-...Ï-...

2:ADB0h:

0F 10 00 00 19 10 00 00 00 00 00 00 14 10 00 00

.....

2:ADC0h:

DE AD 02 00 E2 AD 02 00 E6 AD 02 00 00 00 05 00

b-...â-...æ-.....

如果你想看见具体的这些函数对应的代码，可以根据偏移地址找到对应的字节码，而后将其放入DTDebug之类的调试工具即可看见完整的代码，但是在这里编译器做了一下优化，加了一层JMP跳转，具体的就要跟下去了：

函数名	地址	字节码
Add	0x0000100A	E9 61 00 00 00
Sub	0x0000100F	E9 8C 00 00 00
Div	0x00001019	E9 E7 00 00 00
Mul	0x00001014	E9 B2 00 00 00

7.2.6 AddressOfNames

用于表示导出函数名称表的地址，这张表是按照首字母A-Za-z排序的，同样这里的地址是RVA，需要转换为FOA；通过NumberOfNames知道有3个是以函数名字导出的函数，所以从该地址开始依次寻找3个4字节的地址即可：

1	0x0002ADC0 ->
2	0x0002ADDE
3	0x0002ADE2
4	0x0002ADE6

根据地址就可以找到对应的函数名（字符串见0x00即止）：

```
2:ADD0h: 02 00 44 4C 4C 44 65 6D 6F 2E 64 6C 6C 00 41 64 ..DLLDemo.dll.Ad
2:ADE0h: 64 00 4D 75 6C 00 53 75 62 00 00 00 00 00 00 00 d.Mul.Sub.....
```

7.2.7 AddressOfNameOrdinals

用于表示导出函数序号表的地址，同样这里的地址是RVA，需要转换为FOA；这里表中的成员数与AddressOfNames是一致的，但需要注意，序号表中的每个成员为2字节，那也就是从该地址开始依次寻找3个2字节数据即可：

1	0x0002ADCC ->
2	0x0000
3	0x0005
4	0x0002

```
2:ADC0h: DE AD 02 00 E2 AD 02 00 E6 AD 02 00 00 00 05 00 p-...â-...æ-....
2:ADD0h: 02 00 44 4C 4C 44 65 6D 6F 2E 64 6C 6C 00 41 64 ..DLLDemo.dll.Ad
```

至此，我们就解析完所有的导出表成员了。

7.3 子表之间的关系

当我们去调用一个DLL文件，使用其中的方法，需要使用到GetProcAddress函数去获取函数的地址然后调用：

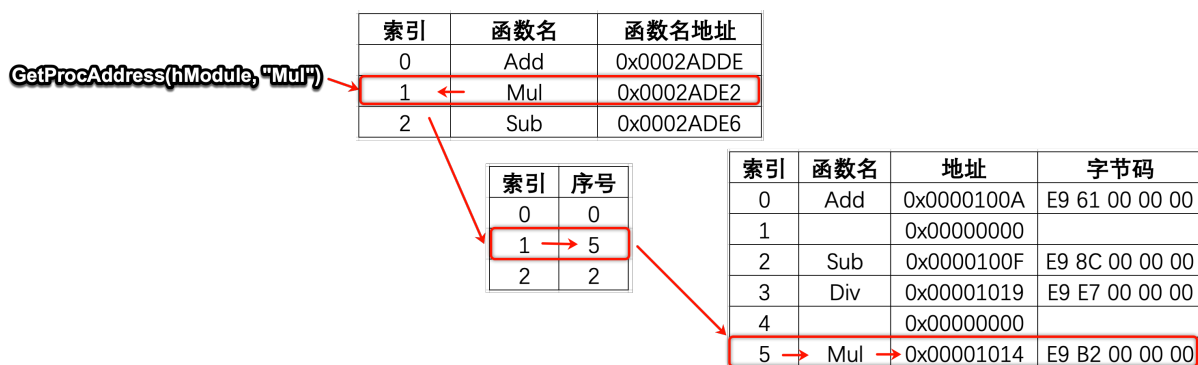
1	FARPROC GetProcAddress(
2	HMODULE hModule, // DLL模块句柄
3	LPCSTR lpProcName // 函数名/序号
4);

它的第二个参数lpProcName可以是函数名也可以是序号，而它的原理也正可以表示导出表中的三张子表之间的关系。

7.3.1 GetProcAddress(hModule, "Mul")

当直接使用函数名去寻找时，它的步骤是这样的：

1. 先用函数名字去查询函数名称表的索引（每张表都有一个从0开始的索引）；
2. 根据函数名称表对应的索引去查函数序号表对应索引的序号；
3. 根据函数序号表对应索引的序号查询函数地址表对应索引的地址。



7.3.2 GetProcAddress(hModule, 13)

当直接使用序号去寻找时，它的步骤是这样的：

1. 使用序号减去Base（起始序号）得到的值；
2. 将该值代入函数地址表对应索引，获得函数地址。



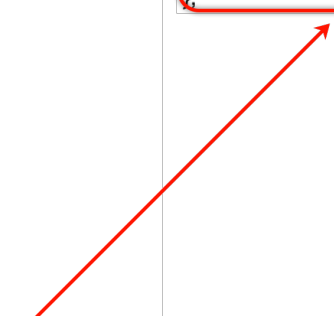
8 导入表

PE文件所依赖的模块以及涉及到依赖模块中的具体函数都存储在导入表中，我们可以在PE格式图的扩展PE头最后一个成员结构体数组中的第二个结构体找到导出表相关的信息，它有2个成员，一个表示导入表的地址，一个表示导入表的大小：

```
struct _IMAGE_OPTIONAL_HEADER {
0x00 WORD Magic;
0x02 BYTE MajorLinkerVersion;
0x03 BYTE MinorLinkerVersion;
0x04 DWORD SizeOfCode;
0x08 DWORD SizeOfInitializedData;
0x0c DWORD SizeOfUninitializedData;
0x10 DWORD AddressOfEntryPoint;
0x14 DWORD BaseOfCode;
0x18 DWORD BaseOfData;
0x1c DWORD ImageBase;
0x20 DWORD SectionAlignment;
0x24 DWORD FileAlignment;
0x28 WORD MajorOperatingSystemVersion;
0x2a WORD MinorOperatingSystemVersion;
0x2c WORD MajorImageVersion;
0x2e WORD MinorImageVersion;
0x30 WORD MajorSubsystemVersion;
0x32 WORD MinorSubsystemVersion;
0x34 DWORD Win32VersionValue;
0x38 DWORD SizeOfImage;
0x3c DWORD SizeOfHeaders;
0x40 DWORD CheckSum;
0x44 WORD Subsystem;
0x46 WORD DllCharacteristics;
0x48 DWORD SizeOfStackReserve;
0x4c DWORD SizeOfStackCommit;
0x50 DWORD SizeOfHeapReserve;
0x54 DWORD SizeOfHeapCommit;
0x58 DWORD LoaderFlags;
0x5c DWORD NumberOfRvaAndSizes;
0x60 _IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

IMAGE_DIRECTORY_ENTRY_IMPORT

```
struct _IMAGE_DATA_DIRECTORY {
0x00 DWORD VirtualAddress;
0x04 DWORD Size;
};
```



导入表与导出表不同，导出表只有一张，而导入表有很多张，每一张表对应着不同的依赖模块，导入表的结构如下，它一共有20字节：

```
1 typedef struct _IMAGE_IMPORT_DESCRIPTOR {
2     union {
3         DWORD Characteristics;
4         DWORD OriginalFirstThunk; // RVA, 指向IMAGE_THUNK_DATA结构数组
5     };
6     DWORD TimeDateStamp; // 时间戳
7     DWORD ForwarderChain;
8     DWORD Name; // RVA, 表示依赖模块名字的地址，这是一串字符串
9     DWORD FirstThunk; // RVA, 指向IMAGE_THUNK_DATA结构数组
10 } IMAGE_IMPORT_DESCRIPTOR;
```

我们可以根据扩展PE头先找到导入表的相关信息：

IPMSG.exe x																																	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0160h:	00	00	00	00	00	00	00	00	80	CE	01	00	DC	00	00	00	€	İ	..	Ü	...											
0170h:	00	30	02	00	D8	3F	00	00	00	00	00	00	00	00	00	00	.0..	ø	?													
0180h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																
0190h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																
01A0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																
01B0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																
01C0h:	00	B0	01	00	94	03	00	00	00	00	00	00	00	00	00	00	..°..	°	..	°												
01D0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																
01E0h:	2E	74	65	78	74	00	00	00	22	97	01	00	00	10	00	00	.text...	"	-													
01F0h:	00	A0	01	00	00	10	00	00	00	00	00	00	00	00	00	00																
0200h:	00	00	00	00	20	00	00	60	2E	72	64	61	74	61	00	00	r	data..													
0210h:	9E	30	00	00	00	B0	01	00	00	40	00	00	00	B0	01	00	ž0...	°	...	@	...	°	...										
0220h:	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40			@	...	@	...										
0230h:	2E	64	61	74	61	00	00	00	D8	37	00	00	00	F0	01	00	.data...	ø	7	...	ä	...											
0240h:	00	40	00	00	00	F0	01	00	00	00	00	00	00	00	00	00	..@...	ä														
0250h:	00	00	00	00	40	00	00	C0	2E	72	73	72	63	00	00	00@...	Ä	.	src...													
0260h:	D8	3F	00	00	00	30	02	00	00	40	00	00	00	30	02	00	ø?	...	0	...	@	...	0	...									
0270h:	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40			@	...	@	...										

- 1 VirtualAddress: 0x0001CE80
- 2 Size: 0x000000DC

根据VirtualAddress找到第一张导入表：

1:CE80h:	B0	CF	01	00	00	00	00	00	00	00	00	00	D8	D5	01	00	°İ	ø	Ö	..												
1:CE90h:	54	B0	01	00	04	D1	01	00	00	00	00	00	00	00	00	00	T°	...	Ñ													
1:CEA0h:	D6	DB	01	00	A8	B1	01	00	A0	CF	01	00	00	00	00	00	ÖÜ	...	±	...	İ											

导入表的每张表之后都跟着另外一张表，直到遇见20个0x00结束，如下图所示，一共有10张表：

1:CE80h:	B0	CF	01	00	00	00	00	00	00	00	00	00	D8	D5	01	00	°İ.....øÖ..
1:CE90h:	54	B0	01	00	04	D1	01	00	00	00	00	00	00	00	00	00	T°...Ñ.....
1:CEA0h:	D6	DB	01	00	A8	B1	01	00	A0	CF	01	00	00	00	00	00	ÖÜ...±...İ.....
1:CEB0h:	00	00	00	00	16	DC	01	00	44	B0	01	00	D4	D2	01	00Ü..D°...öÖ..
1:CEC0h:	00	00	00	00	00	00	00	00	56	DC	01	00	78	B3	01	00VÜ...x³..
1:CED0h:	5C	CF	01	00	00	00	00	00	00	00	00	00	1E	DD	01	00	\İ.....Ý..
1:CEE0h:	00	B0	01	00	DC	D0	01	00	00	00	00	00	00	00	00	00	..°..Üø.....
1:CEF0h:	CE	DD	01	00	80	B1	01	00	E4	D2	01	00	00	00	00	00	İÝ...€±...äö.....
1:CF00h:	00	00	00	00	FC	DD	01	00	88	B3	01	00	6C	D2	01	00üÝ...^³..lò..
1:CF10h:	00	00	00	00	00	00	00	00	14	DE	01	00	10	B3	01	00þ.....³.....
1:CF20h:	8C	CF	01	00	00	00	00	00	00	00	00	00	5E	DE	01	00	Æİ.....^þ..
1:CF30h:	30	B0	01	00	74	D2	01	00	00	00	00	00	00	00	00	00	0°...tò.....
1:CF40h:	6C	DE	01	00	18	B3	01	00	00	00	00	00	00	00	00	00	lp...³.....
1:CF50h:	00	00	00	00	00	00	00	00	00	00	00	00	DC	DC	01	00ÜÖ..

注意：下文中出现的PE文件文件对齐与内存对齐一致，即RVA等于FOA，无需转换。

8.1 确定依赖模块

找到导入表之后，我们可以根据它的第四个成员Name知道当前PE文件所依赖的模块名字，先看第一张表：

```

1:CE80h: B0 CF 01 00 00 00 00 00 00 00 00 00 D8 D5 01 00 °İ.....ØÕ..
1:CE90h: 54 B0 01 00 b4 D1 01 00 00 00 00 00 00 00 00 00 T°..Ñ.....
1:CEA0h: D6 DB 01 00 A8 B1 01 00 A0 CF 01 00 00 00 00 00 ÔÛ..±..İ.....

```

它的Name成员就是：0x0001D5D8，接着找到这个地址查看模块名字，这里也是一个ASCII字符串，所以从这个地址开始，遇见0x00停止寻找：

```

1:D5D0h: 4C 69 62 72 61 72 79 00 4B 45 52 4E 45 4C 33 32 Library.KERNEL32
1:D5E0h: 2E 64 6C 6C 00 b0 AC 02 77 73 70 72 69 6E 74 66 .dll..wsprintf
1:D5F0h: 41 00 30 02 53 65 74 46 6F 72 65 67 72 6F 75 6E A.O.SetForegroun

```

那么它的名字就是KERNEL32.dll，可以用这张方式依次寻找。

我们可以尝试使用工具DTDebug去查看PE文件运行时所要依赖的模块：

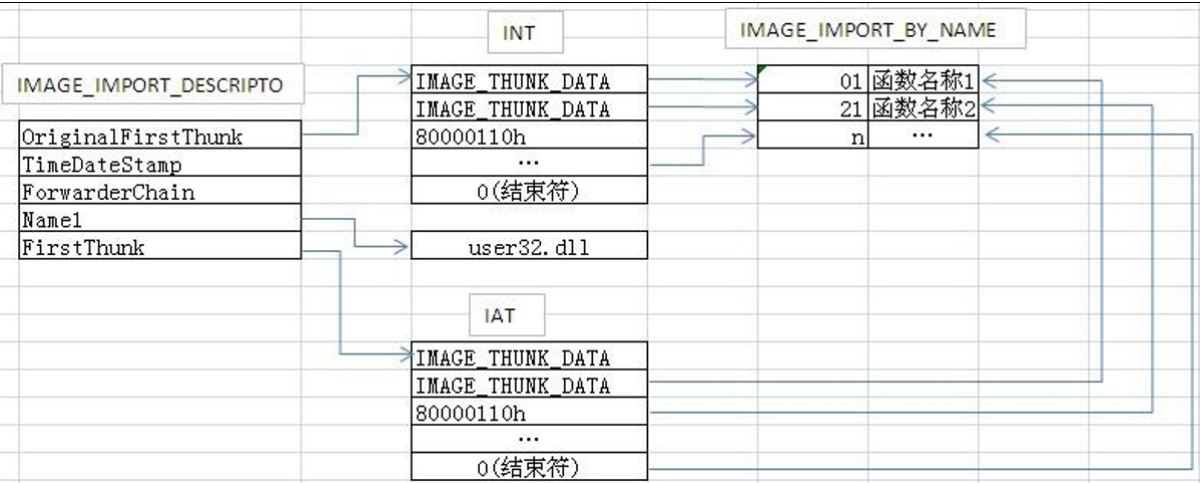
Base	Size	Entry	Name	File version	Path
00400000	00027000	004183D7	IPMSG	2.06	C:\Users\chen\Desktop\IPMSG.exe
73800000	00007000	73801120	WSOCK32	6.1.7600.16385	C:\Windows\system32\WSOCK32.dll
74390000	00002000	74393771	WINRM	6.1.7600.16385	C:\Windows\system32\WINRM.dll
745C0000	00084000	745C19A9	COMCTL32	5.82 (win7_rtm)	C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_5.82.7601.17514_none_ec89dffa859149af\COMCTL32.dll
74840000	0000C000	748410E1	CRYPTBASE	6.1.7601.24384	C:\Windows\syswow64\CRYPTBASE.dll
74850000	00060000	7486A3B0	SapiCL1	6.1.7601.24384	C:\Windows\syswow64\SapiCL1.dll
748B0000	00006000	748B1782	NSI	6.1.7600.16385	C:\Windows\syswow64\NSI.dll
74A60000	0009D000	74A93FD7	USP10	1.0626.7601.1751	C:\Windows\syswow64\USP10.dll
74B10000	000F0000	74B20569	RPCRT4	6.1.7600.16385	C:\Windows\syswow64\RPCRT4.dll
74C00000	00060000	74C1158F	IMM32	6.1.7601.17514	C:\Windows\system32\IMM32.DLL
74C60000	000A1000	74C74919	ADVAPI32	6.1.7601.24384	C:\Windows\syswow64\ADVAPI32.dll
74D10000	00047000	74D17541	KERNELBA	6.1.7600.16385	C:\Windows\syswow64\KERNELBASE.dll
74D60000	00C4A000	74DE1601	SHELL32	6.1.7601.17514	C:\Windows\syswow64\SHELL32.dll
75B00000	00110000	75BF3356	kernel32	6.1.7600.16385	C:\Windows\syswow64\kernel32.dll
75D80000	00090000	75D96343	GDI32	6.1.7601.17514	C:\Windows\syswow64\GDI32.dll
75E60000	0015C000	75EABA3D	ole32	6.1.7600.16385	C:\Windows\syswow64\ole32.dll
75FD0000	00035000	75FD145D	WS2_32	6.1.7600.16385	C:\Windows\syswow64\WS2_32.dll
76010000	00100000	7602B6ED	USER32	6.1.7601.17514	C:\Windows\syswow64\USER32.dll
76750000	0007B000	76751A5E	cmdlg32	6.1.7600.16385	C:\Windows\syswow64\cmdlg32.dll
767D0000	000AC000	767DA472	advccrt	7.0.7600.16385	C:\Windows\syswow64\advccrt.dll
76900000	00057000	76919BA6	SHLWAPI	6.1.7600.16385	C:\Windows\syswow64\SHLWAPI.dll
769E0000	0000A000	769E36A0	LPK	6.1.7600.16385	C:\Windows\syswow64\LPK.dll
769F0000	0000C000	769F16B8	MSCTF	6.1.7600.16385	C:\Windows\syswow64\MSCTF.dll
76AD0000	00019000	76AD4975	sechost	6.1.7600.16385	C:\Windows\SysWow64\sechost.dll
76ED0000	00180000		ntdll	6.1.7600.16385	C:\Windows\SysWow64\ntdll.dll

这时候你会发现这里依赖的模块数量远远大于10，这是因为除了当前PE文件本身需要依赖的模块以外，其所依赖的模块也依赖了其他模块，这里就一并展示进来了。

8.2 确定依赖函数

在上文中我们可以通过导入表确定依赖的模块名（导入表结构体的第四个成员Name），接着我们还需要确定具体依赖的函数是什么。

我们需要关注一下导入表结构体的其他成员，如下图所示，第四个成员很好理解就是指向的依赖的模块名，而第一个成员与第二成员都分别指向了不同的表，分别是INT（Import Name Table，导入名称表）和IAT（Import Address Table，导入地址表）。



仔细观察上图，你会发现，虽然这两个成员指向是不同的表（地址不一样）但是内容却是一样的，我们通过两个成员都可以找到当前所依赖模块中的函数。因此，我们就选择一个成员去寻找依赖函数即可，我在这里选择了第一个成员OriginalFirstThunk。

第一个成员指向的就是一个INT，这张表中的成员就是如下结构体，这个结构体中就只有一个联合体，它的宽度就是4字节：

```
1 typedef struct _IMAGE_THUNK_DATA32 {
2     union {
3         PBYTE ForwarderString;
4         PDWORD Function;
5         DWORD Ordinal; // 序号
6         PIMAGE_IMPORT_BY_NAME AddressOfData; // 指向IMAGE_IMPORT_BY_NAME
7     } u1;
8 } IMAGE_THUNK_DATA32;
```

INT里有很多个这种结构体，它与倒入表一样，当你遇见与该结构体相同宽度的一段0x00填充的数据则该表就**结束了**。同样，你在这张表中发现结构体的数量，就表示你依赖该模块的函数数量。

根据上一章中找到的第一张导入表信息：

```
1 OriginalFirstThunk: B0 CF 01 00
2 Name: D8 D5 01 00 -> KERNEL32.dll
```

跟进地址0x0001CFB0，找到INT，我们可以看见这里有很多个成员：

1:CFB0h:	5E D4 01 00	6C D4 01 00	78 D4 01 00	8A D4 01 00	^Ô..lÔ..xÔ..ŠÔ..
1:CFC0h:	9A D4 01 00	A2 D4 01 00	B4 D4 01 00	C4 D4 01 00	šÔ..cÔ..ôÔ..ĂÔ..
1:CFD0h:	D0 D4 01 00	E0 D4 01 00	F2 D4 01 00	0A D5 01 00	ĐÔ..àÔ..òÔ...Ô..
1:CFE0h:	1A D5 01 00	2E D5 01 00	3E D5 01 00	4E D5 01 00	.Ô...Ô...>Ô..NÔ..
1:CFF0h:	5C D5 01 00	72 D5 01 00	86 D5 01 00	9C D5 01 00	\Ô..rÔ..tÔ..æÔ..
1:D000h:	AC D5 01 00	CA D5 01 00	88 E0 01 00	7C E0 01 00	¬Ô..ÊÔ..^à.. à..
1:D010h:	6A E0 01 00	4C D4 01 00	4C E0 01 00	3E E0 01 00	jà..LÔ..Là..>à..
1:D020h:	2E E0 01 00	1C E0 01 00	02 E0 01 00	EA DF 01 00	.à...à...à..êß..
1:D030h:	D0 DF 01 00	B6 DF 01 00	A0 DF 01 00	84 DF 01 00	Đß..ŕß..ß...ß..
1:D040h:	74 DF 01 00	64 DF 01 00	4E DF 01 00	38 DF 01 00	tß..dß..Nß..8ß..
1:D050h:	24 DF 01 00	10 DF 01 00	00 DF 01 00	F2 DE 01 00	\$ß...ß...ß...òß..
1:D060h:	E4 DE 01 00	D6 DE 01 00	C4 DE 01 00	B2 DE 01 00	äß...Öß...Ăß...²ß..
1:D070h:	9E DE 01 00	90 DE 01 00	84 DE 01 00	78 DE 01 00	žß...ß...ß...xß..
1:D080h:	3E D4 01 00	28 D4 01 00	18 D4 01 00	08 D4 01 00	>Ô..(Ô...Ô...Ô..
1:D090h:	F8 D3 01 00	E8 D3 01 00	D4 D3 01 00	C6 D3 01 00	øÔ..èÔ...ÔÔ...ÆÔ..
1:D0A0h:	B4 D3 01 00	A8 D3 01 00	90 D3 01 00	78 D3 01 00	´Ô...´Ô...´Ô...xÔ..
1:D0B0h:	60 D3 01 00	50 D3 01 00	40 D3 01 00	2A D3 01 00	`Ô...PÔ...@Ô...*Ô..
1:D0C0h:	1A D3 01 00	0C D3 01 00	FE D2 01 00	58 E0 01 00	.Ô...Ô...pÔ...Xà..
1:D0D0h:	F0 D2 01 00	92 E0 01 00	00 00 00 00	AE DD 01 00	ǎÔ...à.....@Ý..

这些成员也有一些精妙之处，你需要判断每一个成员的最高位是否是1，如果是，则去除最高位的值，得出的就不是一个地址了，而是函数的导出序号，反之就是一个RVA，指向了如下结构体，这个结构体一共只有三个字节，第一个成员Hint可能为空，如果不为空，那么它就是函数在导出表中的索引，我们可以使用这个索引直接去导出表找到函数的地址；第二个成员Name，也就是函数的名字，只有一个字节，这是因为函数名字的长度无法确定，所以只取名字的第一个，如果你要完整寻找的话，名字是一个ASCII字符串，从第一个名字开始找，直至遇见0x00结束：

```

1  typedef struct _IMAGE_IMPORT_BY_NAME {
2      WORD      Hint;
3      BYTE      Name[1];
4  } IMAGE_IMPORT_BY_NAME

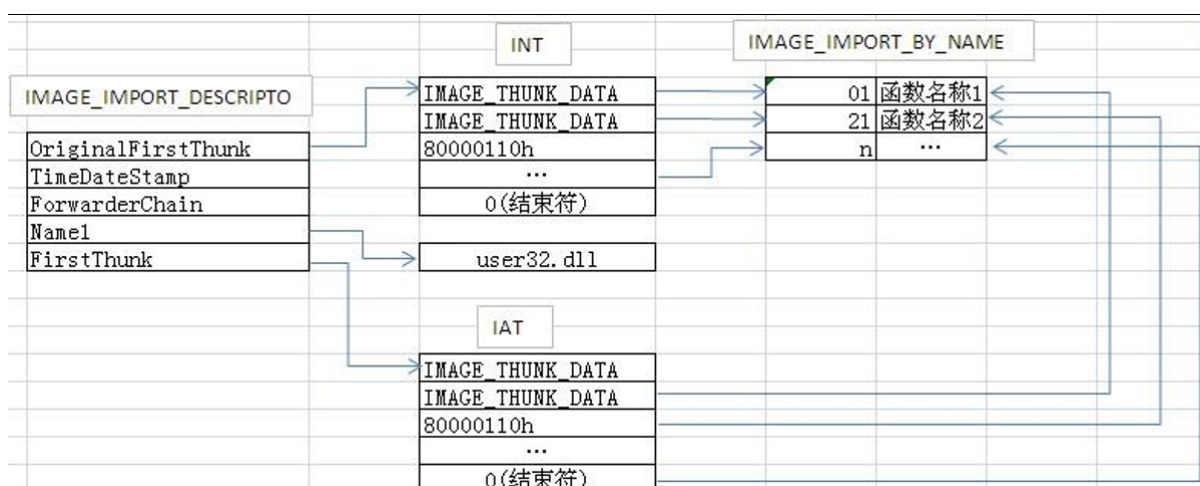
```

那么我们来看一下INT中的第一个成员（地址：0x0001D45E）指向的结构体，如下所示，这里的第一个成员Hint不为空所以我们可以通过它去导入表中找函数地址，第二个成员0x45，从它开始往后直到0x00就是完整的函数名字ExitThread：

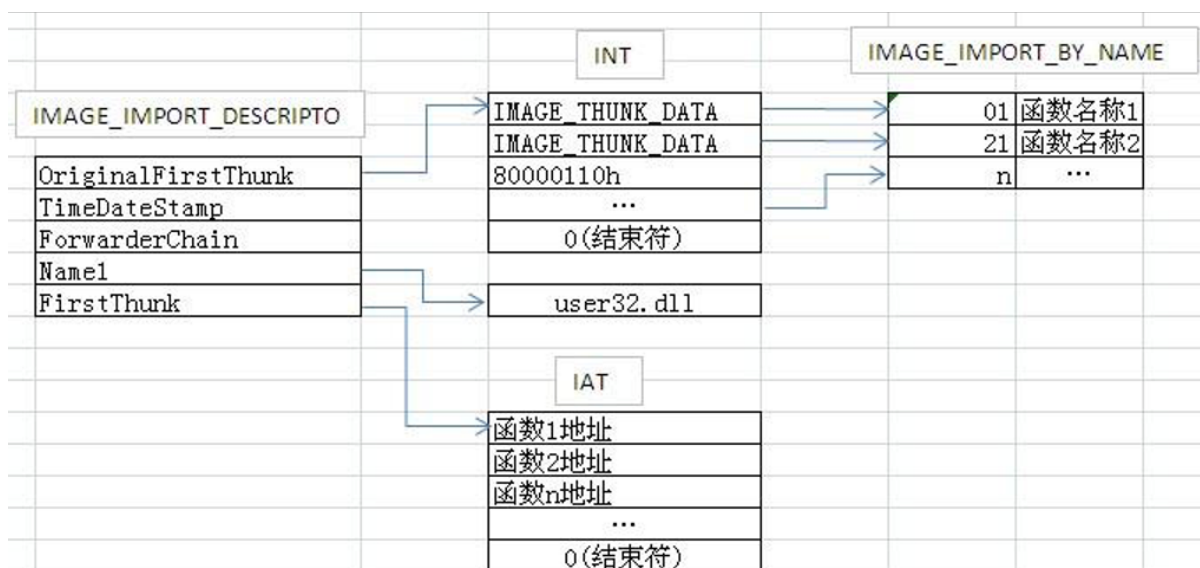
1:D450h:	6D 61 70 56	69 65 77 4F	66 46 69 6C	65 00 7E 00	mapViewOfFile..
1:D460h:	45 78 69 74	54 68 72 65	61 64 00 00	90 00 46 69	ExitThread....Fi

8.3 确定函数地址

之前我们所了解的两张表INT、IAT，在PE文件加载前其结构内容都是一样的：



但是，当PE文件加载之后，IAT就发生了变化，它里面就直接存储了函数的地址：



我们在使用依赖模块的函数时，实际上是间接调用，如下图所示，调用MessageBox，汇编指令去调用并不是直接call地址，而是间接的，从内存中找到地址再去call：

```

11:      ::MessageBoxA(NULL, "Test", "Title", 0);
00401028 8B F4      mov     esi,esp
0040102A 6A 00      push    0
0040102C 68 24 90 42 00 push    offset string "Title" (00429024)
00401031 68 1C 90 42 00 push    offset string "Test" (0042901C)
00401036 6A 00      push    0
00401038 FF 15 74 F3 42 00 call     dword ptr [__imp__MessageBoxA@16 (0042F374)]
0040103E 3B F4      cmp     esi,esp
00401040 E8 2B 00 00 00 call     __chkesp (00401070)

```

而这一块内存就是IAT中存储的函数地址：

Address: 0042f374							
0042F374	0B 05 05 77	00 00	.. 水 ..				
0042F37A	00 00 00 00	00 00				
0042F380	00 00 00 00	00 00				
0042F386	00 00 00 00	00 00				
0042F38C	00 00 00 00	00 00				
0042F392	00 00 00 00	00 00				
0042F398	00 00 00 00	00 00				
0042F39E	00 00 00 00	00 00				
0042F3A4	BE 01 4D 65 73 73	..Mess					
0042F3AA	61 67 65 42 6F 78	ageBox					
0042F3B0	41 00 55 53 45 52	A.USER					
0042F3B6	33 32 2E 64 6C 6C	32.dll					
0042F3BC	00 00 CA 00 47 65Ge					
0042F3C2	74 43 6F 6D 6D 61	tComma					
0042F3C8	6E 64 4C 69 6E 65	ndLine					
0042F3CE	54 00 7E 04 57 65	A t o -					

那么为什么不只留一张表呢？这是因为IAT的函数地址很有可能会被修改掉，导致地址不正确，如果你只有一张表，就没有一个正确的参照物了。

9 重定位表

在PE中最重要的三张表就是导出、导入和重定位表，本章来了解一下重定位表。

重定位表是什么？假设某一PE文件使用了一些模块，这些模块都有自己的ImageBase（在内存中开始的地址），但是实际上在内存中模块的ImageBase被别人占用了，这时候模块就需要偏移，换一个新的内存地址去展开，假设在这模块中有很多已经写好了地址的硬编码（但凡涉及到直接寻址的指令都需要进行重定位处理），当换了地址之后就找不到了，甚至会出现安全隐患，所以硬编码的地址是需要随着偏移而去修改的，这时候就需要一张表去记录需要修正的位置，便于去根据对应偏移修正，这张表我们就称之为重定位表；一般来说，EXE文件是没有重定位表的，因为它不会提供函数给其他人用（导出表），所以运行时它理应是第一个载入内存中的，而DLL之类的PE文件则一定需要重定位表的，因为它并不一定会加载在ImageBase上。

重定位表的位置信息与导入、导出表一样，在扩展PE头的最后一个成员中的第6个结构体，结构体的成员与导入、导出表一样，分别表示重定位表的起始位置和大小：

```
struct _IMAGE_OPTIONAL_HEADER {
0x00 WORD Magic;
0x02 BYTE MajorLinkerVersion;
0x03 BYTE MinorLinkerVersion;
0x04 DWORD SizeOfCode;
0x08 DWORD SizeOfInitializedData;
0x0c DWORD SizeOfUninitializedData;
0x10 DWORD AddressOfEntryPoint;
0x14 DWORD BaseOfCode;
0x18 DWORD BaseOfData;
0x1c DWORD ImageBase;
0x20 DWORD SectionAlignment;
0x24 DWORD FileAlignment;
0x28 WORD MajorOperatingSystemVersion;
0x2a WORD MinorOperatingSystemVersion;
0x2c WORD MajorImageVersion;
0x2e WORD MinorImageVersion;
0x30 WORD MajorSubsystemVersion;
0x32 WORD MinorSubsystemVersion;
0x34 DWORD Win32VersionValue;
0x38 DWORD SizeOfImage;
0x3c DWORD SizeOfHeaders;
0x40 DWORD CheckSum;
0x44 WORD Subsystem;
0x46 WORD DllCharacteristics;
0x48 DWORD SizeOfStackReserve;
0x4c DWORD SizeOfStackCommit;
0x50 DWORD SizeOfHeapReserve;
0x54 DWORD SizeOfHeapCommit;
0x58 DWORD LoaderFlags;
0x5c DWORD NumberOfRvaAndSizes;
0x60 _IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

IMAGE_DIRECTORY_ENTRY_BASERELOC

```
struct _IMAGE_DATA_DIRECTORY {
0x00 DWORD VirtualAddress;
0x04 DWORD Size;
};
```

重定位表中有一个结构体，它一共有8字节：

```
1 typedef struct _IMAGE_BASE_RELOCATION {
```

```
2     DWORD   VirtualAddress; // RVA
3     DWORD   SizeOfBlock;
4 } IMAGE_BASE_RELOCATION;
5 typedef IMAGE_BASE_RELOCATION , * PIMAGE_BASE_RELOCATION;
```

但是完整的重定位表还包含了很多重定位项，所以整体大小是没有一个统一标准的，需要根据实际情况。该结构体的第一个成员VirtualAddress表示基地址，**第二个成员SizeOfBlock也就是减去当前结构体的大小之后其他的所有重定位项加起来的大小。**

这也就表示每一个重定位表是IMAGE_BASE_RELOCATION结构体开始，跟着的就是重定位项。

从理论上来说，重定位表中存储的项应该都是4字节大小的地址，但是这样一旦需要修改的地址多了，就会占用大量的空间，所以重定位表就做了一些优化，假设你现在有这几个地址需要修正：

```
1     0x800123
2     0x800456
3     0x800789
```

那么优化之后，IMAGE_BASE_RELOCATION结构体的第一个成员存储的就是0x800000，而这个结构体之后的每2字节存储**就包含**0x123、0x456、0x789，这样就大大的节省了空间。同时，这也就说明重定位表的实际大小为IMAGE_BASE_RELOCATION结构体（8字节）+N*2字节。

重定位表是按照一个物理页（4KB）进行存储的，也就表示一个4KB内存有需要修正的位置，就会有一个重定位块，一个重定位表只管自己当前的物理页的重定位。

但需要注意的是由于内存对齐的缘故，在重定位表中还是有很多的无用项的，**所以你需要判断当前重定位项（2字节）的高四位是否为3，如果是那么低12位就是偏移量，最后的地址也就是VirtualAddress+低12位，如果不是就表示这是无所谓的值。**