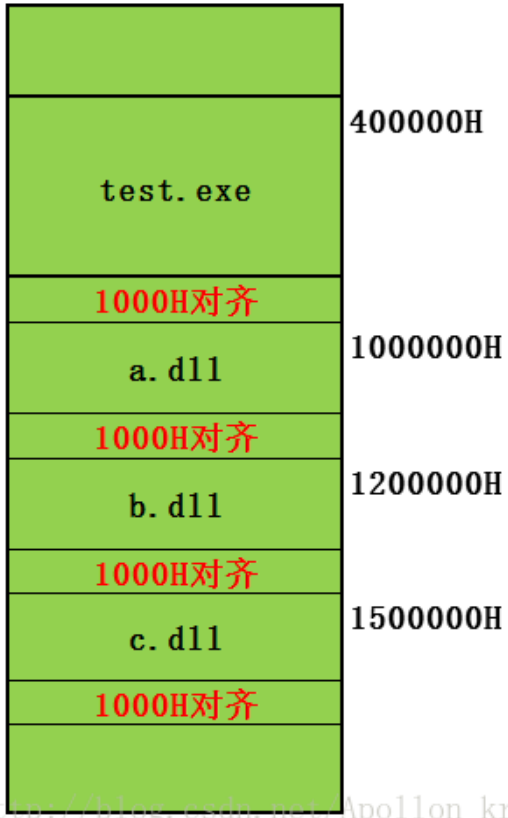
PE文件学习笔记（四）：重定位表（Relocation Table）解析

原创 Apollon_krj 于 2017-08-18 17:19:02 发布

分类专栏： COFF PE/ELF 文章标签： 操作系统 内存 库

1、重定位表的作用

重定位表（Relocation Table）用于在程序加载到内存中时，进行内存地址的修正。为什么要进行内存地址的修正？我们举个例子来说：test.exe可执行程序需要三个动态链接库dll（a.dll，b.dll，c.dll），**假设test.exe的ImageBase为400000H，而a.dll、b.dll、c.dll的基址ImageBase均为1000000H。**
那么操作系统的加载程序在将test.exe加载进内存时，直接复制其程序到400000H开始的虚拟内存中，接着一一加载a.dll、b.dll、c.dll：假设先加载a.dll，如果test.exe的ImageBase + SizeOfImage + 1000H不大于1000000H，则a.dll直接复制到1000000H开始的内存中；当b.dll加载时，虽然其基址也为1000000H，但是由于1000000H已经被a.dll占用，则b.dll需要重新分配基址，比如加载程序经过计算将其分配到1200000H的地址，c.dll同样经过计算将其加载到150000H的地址。
如下图所示：



但是b.dll和c.dll中有些地址是根据ImageBase固定的，被写死了的，而且是绝对地址不是相对偏移地址。比如b.dll中存在一个call 0X01034560，这是一个绝对地址，其相对于ImageBase的地址为δ = 0X01034560 - 0X01000000 = 0X34560H；而此时的内存中b.dll存在的地址是1200000H开始的内存，加载器分配的ImageBase和b.dll中原来默认的ImageBase（1000000H）相差了200000H，因此该call的值也应该加上这个差值，被修正为0X01234560H，那么δ = 0X01234560H - 0X01200000H = 0X34560H则相对不变。否则call的地址不修正会导致call指令跳转的地址不是实际要跳转的地址，获取不到正确的函数指令，程序则不能正常运行。

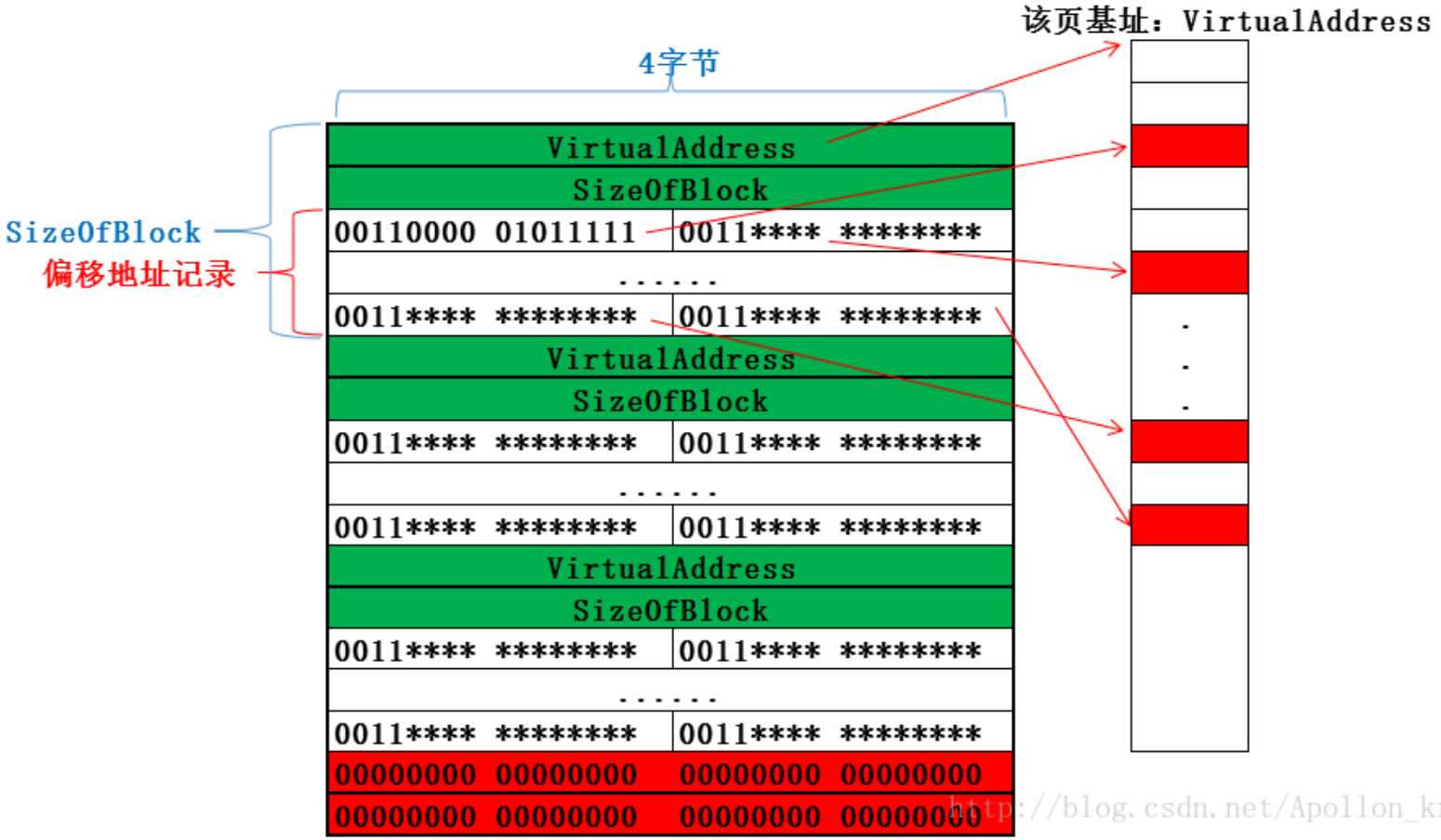
由于一个dll中的需要修正的地址不止一两个，可能有很多，所以用一张表记录那些“写死”的地址，将来加载进内存时，可能需要一一修正，这张表称作为重定位表，一般每一个PE文件都有一个重定位表。当加载器加载程序时，如果加载器为某PE（.exe、.dll）分配的基址与其自身默认记录的ImageBase不相同，那么该程序文件加载完毕后就需要修正重定位表中的所有需要修正的地址。如果加载器分配的基址和该程序文件中记录默认的ImageBase相同，则不需要修正，重定位表对于该dll也是没有效用的。比如test.exe和a.dll的重定位表都是不起作用的（**由于一般情况.exe运行时被第一个加载，所以exe文件一般没有重定位表，但是不代表所有exe都没有重定位表**）。同理如果先加载b.dll后加载a.dll、c.dll，那么b.dll的重定位表就不起作用了。

2、重定位表的结构与解析：

知道了重定位表的作用，现在我们来分析一个重定位表在PE文件中是如何存在的，首先来看看描述重定位表的结构：

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress;//RVA
    DWORD   SizeOfBlock;
} IMAGE_BASE_RELOCATION,* PIMAGE_BASE_RELOCATION;
#define IMAGE_SIZEOF_BASE_RELOCATION         8
```

该结构体有两个成员：一个是地址，一个是大小。如下图所示：一个重定位表由多个大小SizeOfBlock的Block组成，（不同块的SizeOfBlock大小不一）。每一个块记录了1000H即4KB大小的内存中需要重定位信息的地址（一页大小），这些地址以VirtualAdress为该页的基址，偏移地址占两个字节（1000H最多需要12bit即可：0~FFFH）。所以两个字节的低12位为偏移地址，而高4位就是一个标记，当此标记为0011（3）时低12为才有效，否则该2个字节可能是为了对齐而产生的，并且为对齐而产生的字节其值全为0。



由于重定位表的SizeOfBlock大小不确定，新的Block的重定位信息的结构体接着上一个Block4字节对齐后开始，而当出现一个_IMAGE_BASE_RELOCATION结构体的值全为0时，表明重定位表结束。

3、代码解析PE的Relocation Table：

```
//将解析出来的重定位表信息写入文件
void PETool::print_BaseRelocation()
{
    fprintf(fp_peMess, "重定位表(Relocation):\n");
    if(dataDir[5].VirtualAddress == 0){
        fprintf(fp_peMess, "\t不存在重定位表!\n");
        return;
```

```
    }
    BYTE secName[9] = {0};
    //rec指向重定位表第一个IMAGE_BASE_RELOCATION结构体
    IMAGE_BASE_RELOCATION * rec = (IMAGE_BASE_RELOCATION *)(pFileBuffer + RVAToFOA(dataDir[5].VirtualAddress));
    for(int i = 1; rec->SizeOfBlock && rec->VirtualAddress; i++){
        DWORD foa = RVAToFOA(rec->VirtualAddress);
        DWORD size = (rec->SizeOfBlock - 8) / 2;
        //确定该结构体所处节，并获取节名称
        IMAGE_SECTION_HEADER * section = section_header;
        for(int t = 0; t < sectionNum; t++){
            DWORD lower = RVAToFOA(section[t].VirtualAddress);
            DWORD upper = RVAToFOA(section[t].VirtualAddress) + section[t].Misc.VirtualSize;
            if(foa >= lower && foa <= upper){
                memcpy(secName, section[t].Name, 8);
                break;
            }
        }
        //打印该页主要信息
        fprintf(fp_peMess, "\tIndex[%d]\tsection[%s]\tOffset[%08X]\tItems[%dD:%XH]\t【Block】\n", i, secName, foa, size, size);

        //打印一个页中所有重定位地址与信息
        WORD * recAddr = (WORD *)((BYTE *)rec + 8);//recAddr指向重定位表结构体后的首字节
        fprintf(fp_peMess, "\t\tindex\toffset\t\ttype\t【Block Items】\n");
        for(DWORD j = 0; j < size; j++){
            DWORD offset = (recAddr[j] & 0X0FFF) + foa;//低四位是偏移地址
            WORD type = recAddr[j] >> 12;//高四位是有效判断位
            if(type == 0){
                fprintf(fp_peMess, "\t\t[%d] \t[--------]\tABSOLUTE[%d]\n", j+1, type);
                continue;
            }
            fprintf(fp_peMess, "\t\t[%d] \t[%08X]\tHIGHLOW[%d]\n", j+1, offset, type);
        }
        memset(secName, 0, 9);
        rec = (IMAGE_BASE_RELOCATION *)((BYTE *)rec + rec->SizeOfBlock);//进行下一页的判断

```

```
    }
}
```

由于打印的信息过长（一个只包含加减乘除四个简单函数的库，其重定位信息的地址就有2500多条），这里只提出一部分比较短的Block信息，可以很明显地看到当需要重定位信息的记录长度是4Byte的倍数时，不存在高四位为0000的情况，当其不为4的倍数时，就有一个因对齐而产生的数据：

复制

```
//...
    Index[21]   section[.text]  Offset[00015000]    Items[16D:10H]   【Block】
        index   offset      type    【Block Items】
        [1]     [00015002]  HIGHLOW[3]
        [2]     [00015008]  HIGHLOW[3]
        [3]     [0001500E]  HIGHLOW[3]
        [4]     [00015014]  HIGHLOW[3]
        [5]     [0001501A]  HIGHLOW[3]
        [6]     [00015020]  HIGHLOW[3]
        [7]     [00015026]  HIGHLOW[3]
        [8]     [0001502C]  HIGHLOW[3]
        [9]     [00015032]  HIGHLOW[3]
        [10]    [00015038]  HIGHLOW[3]
        [11]    [0001503E]  HIGHLOW[3]
        [12]    [00015044]  HIGHLOW[3]
        [13]    [0001504A]  HIGHLOW[3]
        [14]    [00015050]  HIGHLOW[3]
        [15]    [00015056]  HIGHLOW[3]
        [16]    [--------]  ABSOLUTE[0]

//...
    Index[23]   section[.rdata] Offset[0002D000]    Items[12D:CH]    【Block】
        index   offset      type    【Block Items】
        [1]     [0002D804]  HIGHLOW[3]
        [2]     [0002D808]  HIGHLOW[3]
        [3]     [0002D810]  HIGHLOW[3]
        [4]     [0002D814]  HIGHLOW[3]
        [5]     [0002D874]  HIGHLOW[3]
        [6]     [0002D878]  HIGHLOW[3]
        [7]     [0002D884]  HIGHLOW[3]
```

```
        [8]     [0002D888]  HIGHLOW[3]
        [9]     [0002D8E4]  HIGHLOW[3]
        [10]    [0002D8E8]  HIGHLOW[3]
        [11]    [0002D8F0]  HIGHLOW[3]
        [12]    [0002D8F4]  HIGHLOW[3]
//...
    Index[25]   section[.data]  Offset[00031000]    Items[4D:4H]    【Block】
        index   offset      type    【Block Items】
        [1]     [00031CC0]  HIGHLOW[3]
        [2]     [00031CC8]  HIGHLOW[3]
        [3]     [00031CCC]  HIGHLOW[3]
        [4]     [--------]  ABSOLUTE[0]
//...
```