### PE文件整体结构

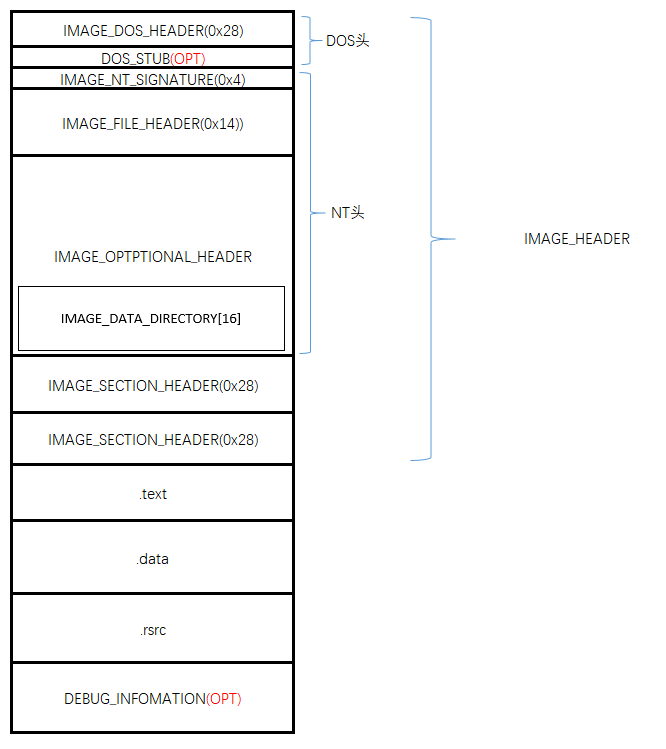


图 1 - 1

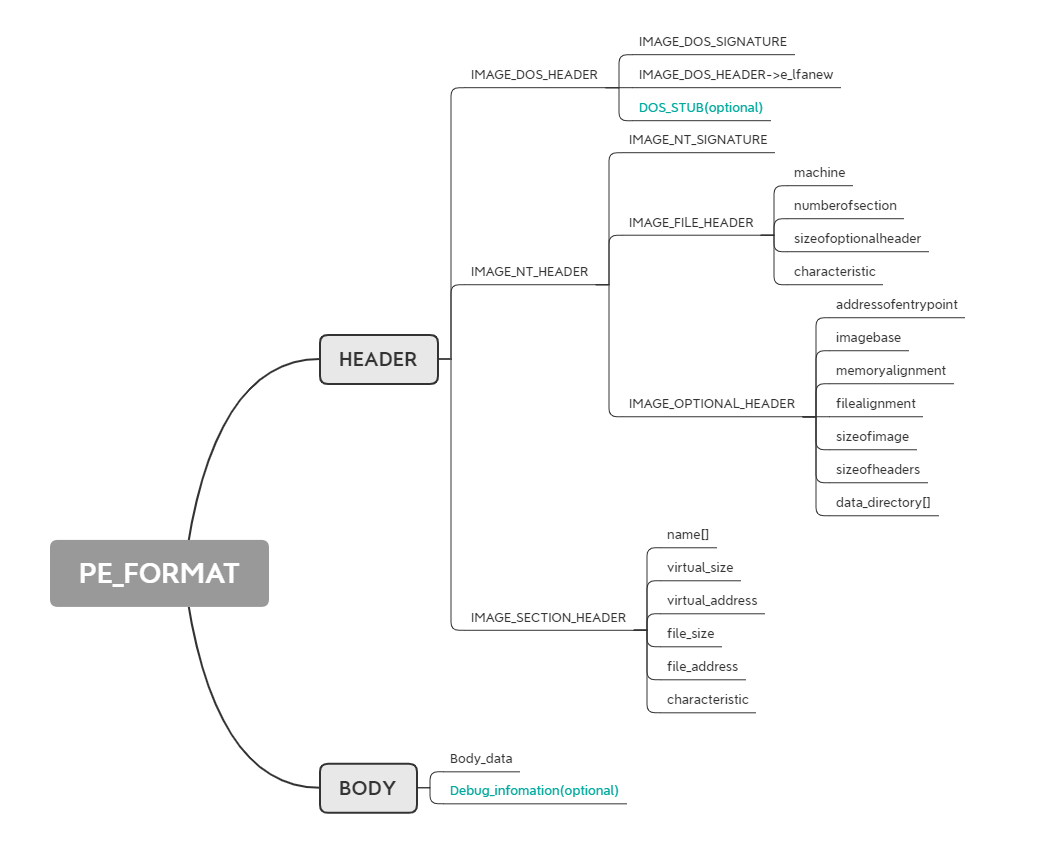


图 1 - 2

以上图 1 - 1和图 1 - 2描述了PE文件格式的整体轮廓，其中图 1 - 1中从文件布局的角度反应了各个结构的相对位置，在PE文件结构的组织上，头部结构大多有固定大小，在图中括号注解标注出。图 1 - 2描述了PE文件头部中关键的结构以及重要的字段，后续解析过程中会详细解析各个结构及其字段的含义。

### PE文件加载映射过程

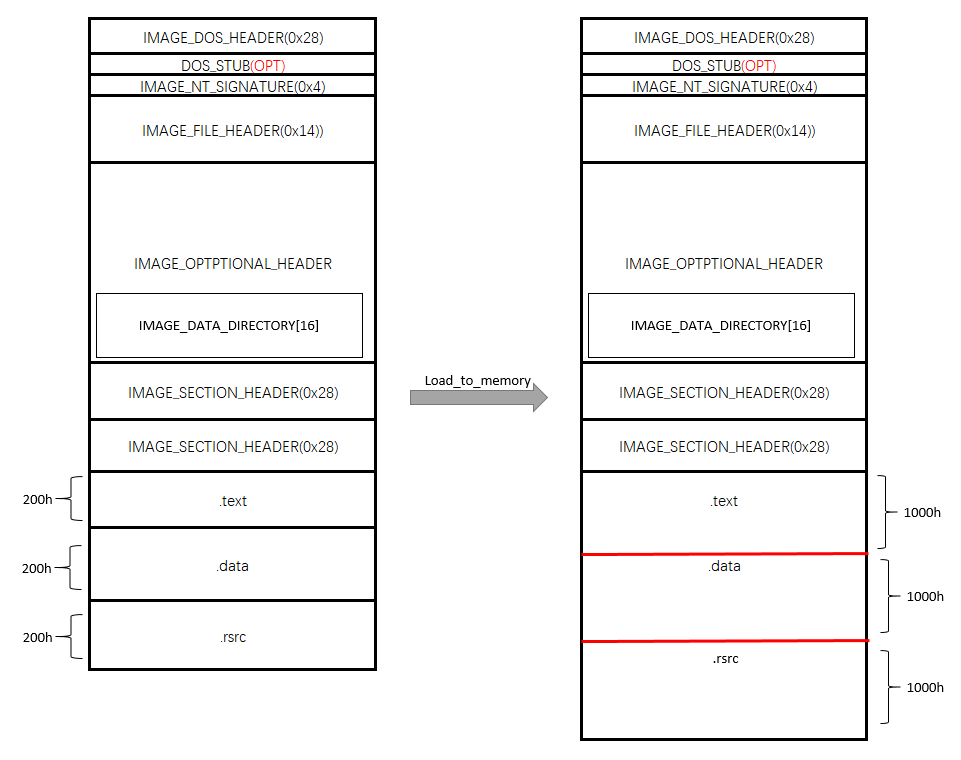


图 2 - 1

加载器在将PE文件从磁盘加载到内存中执行时，直接将IMAGE\_DOS\_HEADER、IMAGE\_NT\_HEADER、IMAGE\_SECTION\_HEADERS等文件头部信息拷贝到ImageBase对应的地址处，然后根据IMAGE\_OPTIONAL\_HEADER中定义的内存和文件对其粒度对节区对其到粒度边界。

### 解析术语

在解析过程会用到以下术语描述PE文件的不同状态。

**VA**：Virtual Address，经过加载器加载到内存后PE文件所处的地址位置。例如，PE文件直接使用VA地址寻址字符串。

**RVA**：Relative Virtual Address，相对虚拟地址

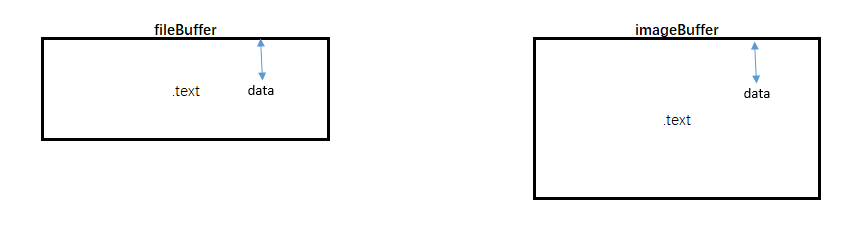


图 3 - 1

由于节区在加载后经过对其，要跟根据imageBuffer中的data地址找到fileBuffer中对应的data，就需要根据图中蓝色距离相等即(data–sec\_base\_address)->fileBuffer==(data–sec\_base\_address)->imageBuffer来定位。而imageBuffer中data的地址即RVA，相对于其节区基址的地址。

**FOA：**File Offset Address上述根据imageBuffer中data定位的描述即可得到fileBuffer中data的FOA地址，相对于fileBuffer的偏移。

### 文件头部解析(IMAGE\_HEADER)

#### 4.1. DOS文件头部

如图 1 - 2描述，已经将PE文件头部重要的字段描述出来，将文件读入fileBuffer之后，即可定位到IMAGE\_DOS\_HEADER,此结构固定大小为0x28字节，重要的字段只有头尾两个e\_magic一个两字节的魔术字，纪念PE文件格式的创建者名字的首字母”MZ”->0x4d5a,加载器会校验此特征，篡改之后PE文件无法正常加载，如图 4 - 3所示，最后一个四字节e\_lfanew指向IMAGE\_NT\_HEADER的偏移。

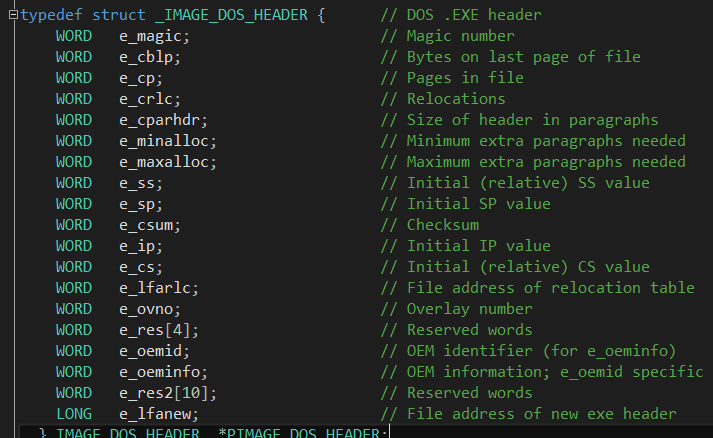


图 4 - 1

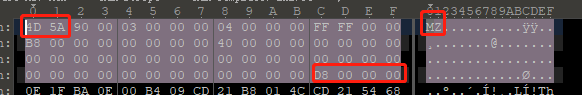


图 4 - 2

篡改文件的DOS\_SIGNATURE之后无法正常加载。

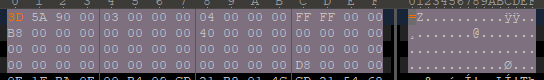




图 4 - 3

#### 4.2 NT文件头部

根据DOS头部最后一个字段偏移寻址到IMAGE\_NT\_HEADER,此结构由三部分组成，首先是一个四字节PE标记IMAGE\_NT\_SIGNATURE，然后是一个20字节的IMAGE\_FILE\_HEADER和一个IMAGE\_OPTIONAL\_HEADER.

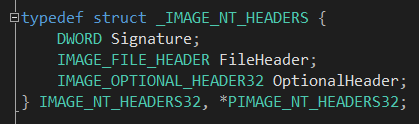


图 4 - 4

同样，Signature字段也是固定标记字，不能篡改。IMAGE\_FILE\_HEADER中定义了PE文件编译目录平台信息Machine字段，在winnt.h中定义了完整的平台特征码。查用如下：

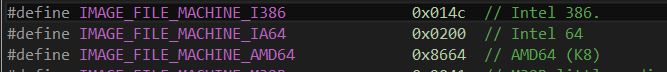


图 4 - 5

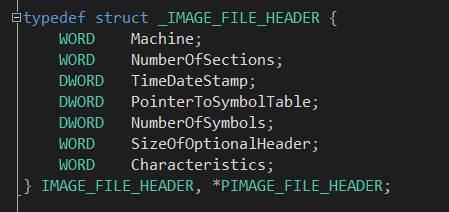


图 4 - 6

NumberOfSections定义了节区的数量，也即IMAGE\_SECTION\_HEADER的数量，节区表和节区一一对应。SizeOfOptionalHeader表明紧挨着的IMAGE\_OPTIONAL\_HEADER的大小，在i386平台上为0xE0, x64平台为0xF0字节大小。最后Characteristic定义了PE文件格式的属性信息，如下是对应的属性码：

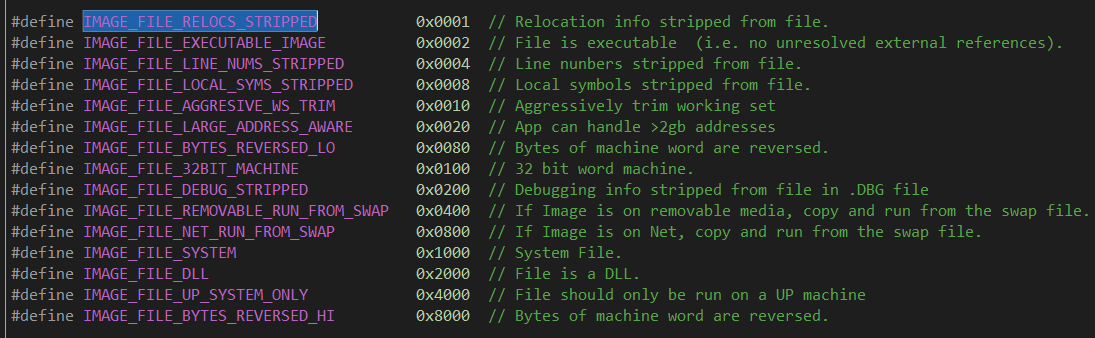


图 4 - 7

之后便是IMAGE\_OPTIONAL\_HEADER，此结构较大，重要的字段有相对偏移0x10的AddressOfEntryPoint，其定义了PE文件加载到内存中执行的第一条指令的偏移。然后是偏移0x1c的ImageBase，加载器会将PE文件加载此虚拟地址处，紧接着两个DWORD分别代表内存映像对齐粒度SectionAlignment和磁盘文件对齐粒度FileAlignment，加载过程中会根据内存对齐粒度”拉伸”对应的节区。后面是0x38偏移处的SizeOfImage,文件加载到内存中的大小，和SizeOfHeader，文件头部的总大小(经过文件对齐)，包含DOS、NT、节表。最后最重要的就是0x60偏移处的数据目录表了，包含十六项，每一个8字节，分别代表对应数据目录表对应的RVA和大小。后续解析对应数据目录的时候会用到此目录。

以上偏移地址对应于32位PE文件，在64位PE文件格式中，头部仅仅增大了IMAGE\_OPTIONAL\_HEADER，由0xE0->0xF0,具体有：

1. 将原本偏移0x18和0x1c的SizeOfData和ImageBase合并为一个8字节的ImageBase。
2. 将原本偏移0x48处的四个4字节字段扩展为8字节。

所以整个结构就增大了16字节，数据目录表就位于0x70处了。

#### 4.3 节表

解析完IMAGE\_OPTIONAL\_HEADER后紧接着就是IMAGE\_SECTION\_HEADER，每一个节表对应描述一个节区的属性，如Name指向节区的名称，以’\0’为结束符。VirtualSize,该节区在内存中的大小(未对齐),VirtualAddress, 节区加载到内存中相对于基址的偏移。同理，SizeOfRawData和PointerToRawData代表文件中对应的大小和偏移，Characteristics指示对应节区的属性，有如下预定义的特征码：

如是否为代码节、是否可读、可写、可执行等等。

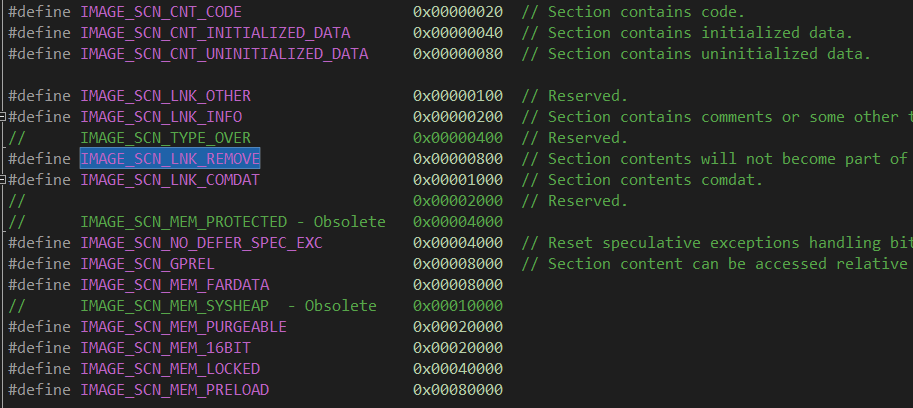


图 4 - 8

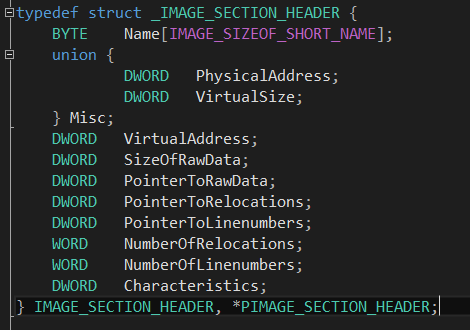


图 4 - 9

### 数据目录表解析(IMAGE\_DATA\_DIRECTORY)

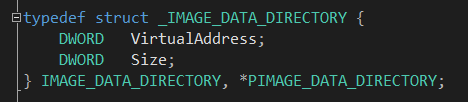


图 5 - 1

数据目录结构中只有两个成员，VirtualAddress指定了数据目录的RVA，Size则表示该块的大小。数据目录表根据编译的目标平台不同可能位于相对IMAGE\_OPTIONAL\_HEADER头偏移0x60或0x70处，其大小为0x80字节。包含16个数据项，每一项8字节，分别指向对应数据目录项的RVA和其大小。其中比较重要的有第0项导出表、第1项导入表、第2项资源表、第4项证书表(主要用于PE校验和检测)、第5项重定位表.

### 导出表解析(IMAGE\_EXPORT\_DIRECTORY)

首先根据数据目录表中对应的项RVA转换FOA后定位到IMAGE\_EXPORT\_DIRECTORY，

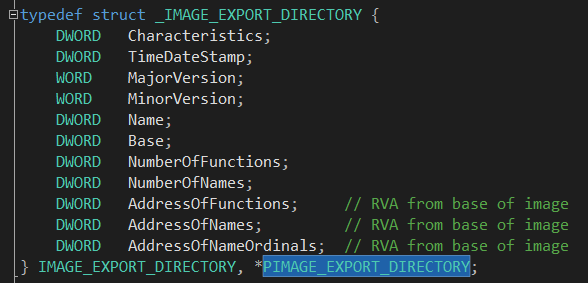


图 6 - 1

导出表用于导出函数，导出函数供外部使用，结构中前四个字段基本没什么用，除此之外的字段都是重要字段。可以看到一个函数的导出可以以名称或者直接以ID进行导出，于是便可看到最后三个字段分别指向导出函数地址表，导出函数名称表，导出函数序号表。

AddressOfFunction, RVA地址，用于索引到存储导出函数地址的表。NumberOfFunctions限定表长。

AddressOfNames，RVA地址，用于索引到所有以名称导出的函数的名称表所在，表长为NumberOfNames，索引到的结构也是一个RVA地址组成的指针表。

AddressOfOrdinals，RVA地址，用于索引到序号表，该表元素为2字节的序号组成的数组，长度为NumberOfNames.

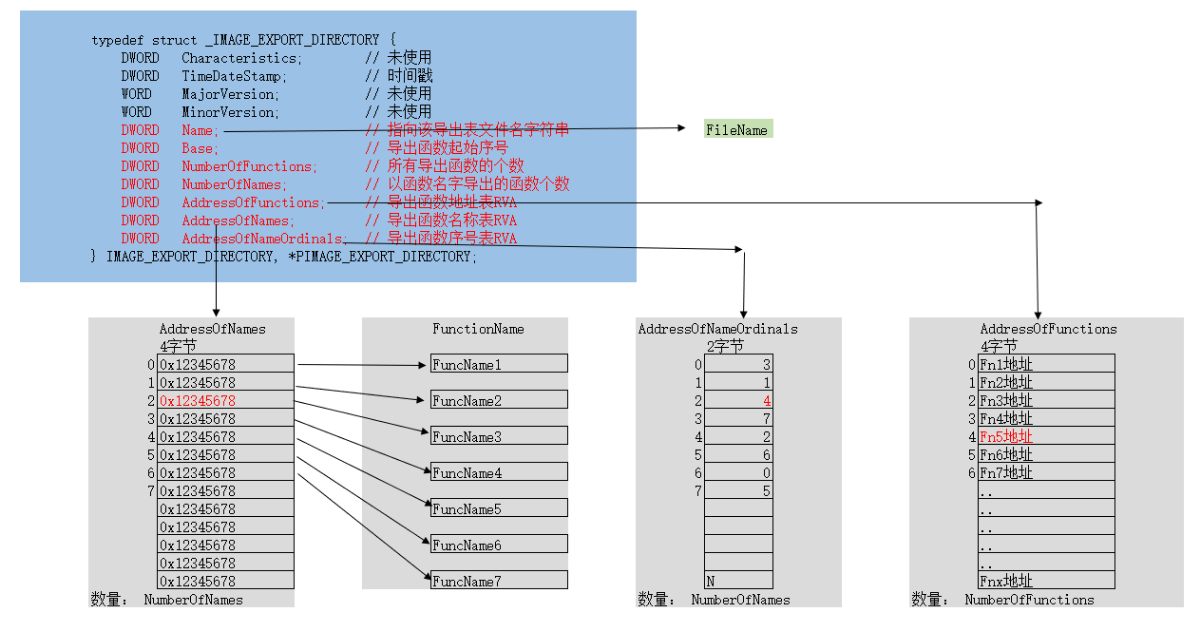


图 6 - 2

通过导出表查找函数地址的方式：

1. 通过函数名查找、

(1). 首先定位函数名称表，遍历函数名称与所查找名称比较，取得函数名称在表中的索引；

(2). 使用索引取得序号表中对于的序号；

(3). 使用取得的序号作为索引在函数地址表中取得函数地址。

1. 通过序号查找

(1). 直接计算函数地址表的索引，使用待查序号-导出表base值即可

(2). 使用序号作为索引在函数地址表中取得函数地址。

### 七．导入表解析(IMAGE\_IMPORT\_DESCRIPTOR)

### 八．资源表解析(IMAGE\_RESOURCE\_DIRECTORY)

### 九．重定位表解析(IMAGE\_BASE\_RELOCATION)

程序加载执行时会导入多个dll文件，每个dll文件也是PE结构，也有着对于的加载基址，当多个dll加载的基址冲突时，就需要将后加载的dll文件加载到内存的其他位置，而PE文件中有一些地址是以VA地址的形式出现在文件中的，一旦该表其加载基址，这些VA地址都将失效，为了修复这些地址引入重定位表。修复的方式为：需要重定位的地址 – PE文件中定义的基址 + 当前加载的真实基址。

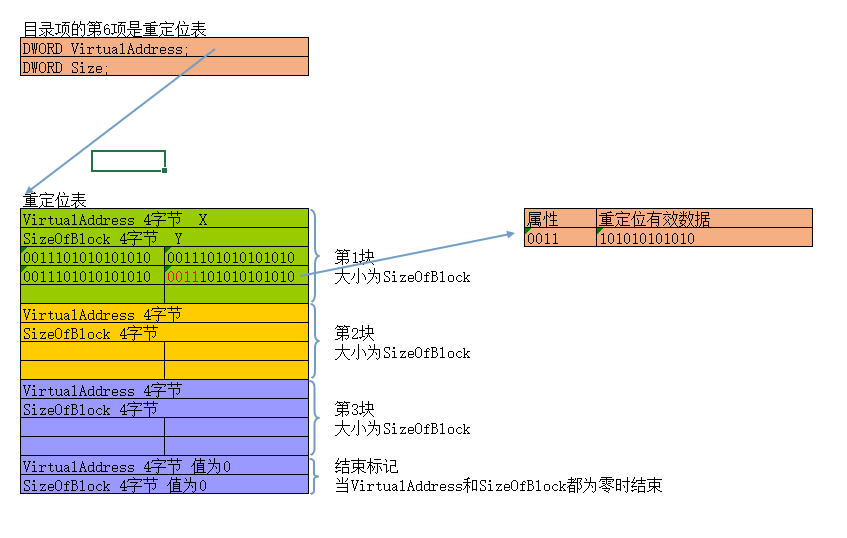


图 9 - 1