Windows PE/COFF

5.1 Windows 的二进制文件格式 PE/COFF
5.2 PE 的前身——COFF
5.3 链接指示信息
5.4 调试信息
5.5 大家都有符号表
5.6 Windows 下的 ELF——PE
5.7 本章小结

5.1 Windows 的二进制文件格式 PE/COFF

在 32 位 Windows 平台下，微软引入了一种叫 PE（Protable Executable）的可执行格式。作为 Win32 平台的标准可执行文件格式，PE 有着跟 ELF 一样良好的平台扩展性和灵活性。PE 文件格式事实上与 ELF 同根同源，它们都是由 COFF（Common Object File Format）格式发展而来的，更加具体地讲是来源于当时著名的 DEC（Digital Equipment Corporation）的 VAX/VMS 上的 COFF 文件格式。因为当微软开始开发 Windows NT 的时候，最初的成员都是来自于 DEC 公司的 VAX/VMS 小组，所以他们很自然就将原来系统上熟悉的工具和文件格式都搬了过来，并且在此基础上做重新设计和改动。

微软将它的可执行文件格式命名为“Portable Executable”，从字面意义上讲是希望这个可执行文件格式能够在不同版本的 Windows 平台上使用，并且可以支持各种 CPU。比如从 Windows NT、Windows 95 到 Windows XP 及 Windows Vista，还有 Windows CE 都是使用 PE 可执行文件格式。不过可惜的是 Windows 的 PC 版只支持 x86 的 CPU，所以我们几乎只要关注 PE 在 x86 上的各种性质就行了。

请注意，上面在讲到 PE 文件格式的时候，只是说 Windows 平台下的**可执行文件**采用该格式。事实上，在 Windows 平台，VISUAL C++编译器产生的**目标文件**仍然使用 COFF 格式。由于 PE 是 COFF 的一种扩展，所以它们的结构在很大程度上相同，甚至跟 ELF 文件的基本结构也相同，都是基于段的结构。所以我们下面在讨论 Windows 平台上的文件结构时，目标文件默认为 COFF 格式，而可执行文件为 PE 格式。但很多时候我们可以将它们统称为 PE/COFF 文件，当然我们在下文中也会对比 PE 与 COFF 在结构方面的区别之处。

随着 64 位 Windows 的发布，微软对 64 位 Windows 平台上的 PE 文件结构稍微做了一些修改，这个新的文件格式叫做 PE32+。新的 PE32+并没有添加任何结构，最大的变化就是把那些原来 32 位的字段变成了 64 位，比如文件头中与地址相关的字段。绝大部分情况下，PE32+与 PE 的格式一致，我们可以将它看作是一般的 PE 文件。

与 ELF 文件相同，PE/COFF 格式也是采用了那种基于段的格式。一个段可以包含代码、数据或其他信息，在 PE/COFF 文件中，至少包含一个代码段，这个代码段的名字往往叫做“.code”，数据段叫做“.data”。不同的编译器产生的目标文件的段名不同，VISUAL C++使用“.code”和“.data”，而 Borland 的编译器使用“CODE”，“DATA”。也就是说跟 ELF 一样，段名只有提示性作用，并没有实际意义。当然，如果使用链接脚本来控制链接，段名可能会起到一定的作用。

跟 ELF 一样，PE 中也允许程序员将变量或函数放到自定义的段。在 GCC 中我们使用“__attribute__((section(“name”)))”扩展属性，在 VISUAL C++中可以使用“#pragma”编译器指示。比如下面这个语句：

```
#pragma data_seg("FOO")
int global = 1;
#pragma data_seg(".data")
```

就表示把所有全局变量“global”放到“FOO”段里面去，然后再使用“#pragram”将这个编译器指示换回来，恢复到“.data”，否则，任何全局变量和静态变量都会被放到“FOO”段。

5.2 PE 的前身——COFF

还记得刚开始分析 ELF 文件格式时的那个 SimpleSection.c 吗？我们接下来还是以它为例子，看看在 Windows 下，它被编译成 COFF 目标文件时，所有的变量和函数是怎么存储的。在这个过程中，我们将用到“Microsoft Visual C++”的编译环境。包括编译器“cl”，链接器“link”，可执行文件查看器“dumpbin”等，你可以通过 Microsoft 的官方网站下载免费的 Visual C++ Express 2005 版，这已经足够用了。

要使用这些工具，我们要在 Windows 命令行下面运行它们，Visual C++在安装完成后就会有一个批处理文件用来建立运行这些工具所须要的环境。它位于**开始/程序/Microsoft Visual Studio 2005/Visual Studio Tools/ Visual Studio 2005 Command Prompt**，这样我们就可以通过命令行使用 VC++的编译器了。然后使用“cd”命令进入到源代码所在目录后运行：

```
cl /c /Za SimpleSection.c
```

“cl”是 VISUAL C++的编译器，即“Compiler”的缩写。/c 参数表示只编译，不链接，即将.c 文件编译成.obj 文件，而不调用链接器生成.exe 文件。如果不加这个参数，cl 会在编译“SimpleSection.c”文件以后，再调用 link 链接器将该产生的 SimpleSection.obj 文件与默认的 C 运行库链接，产生可执行文件 SimpleSection.exe。

VISUAL C++有一些 C 和 C++语言的专有扩展，这些扩展并没有定义 ANSI C 标准或 ANSI C++标准，具体可以参阅 MSDN 的 Microsoft Extensions to C and C++这一节。“/Za”参数禁用这些扩展，使得我们的程序跟标准的 C/C++兼容，这样可以尽量地看到问题的本质。另外值得一提的是，使用/Za 参数时，编译器自动定义了__STDC__这个宏，我们可以在程序里通过判断这个宏是否被定义而确定编译器是否禁用了 Microsoft C/C++语法扩展。

编译完成以后我们得到了一个 971 字节的 SimpleSection.obj 目标文件，当然文件大小可

能会因为编译器版本、选项及机器平台不同而不同。跟 GNU 的工具链中的“objdump”一样，Visual C++也提供了一个用于查看目标文件和可执行文件的工具，就是“dumpbin”。下面这个命令可以查看 SimpleSection.obj 的结构：

```
dumpbin /ALL SimpleSection.obj > SimpleSection.txt
```

“/ALL”参数是将打印输出目标文件的所有相关信息，包括文件头、每个段的属性和段的原始数据及符号表。由于输出信息较多，如果直接打印到终端上，可能不太便于查看，所以我们将其导向到一个输出文件“SimpleSection.txt”中。因为在接下来的分析过程中，我们将会经常用到这个“dumpbin”的输出结果，所以将它保存在“SimpleSection.txt”文件中，以便后面分析时逐一对照。我们也可以用“/SUMMARY”选项来查看整个文件的基本信息，它只输出所有段的段名和长度：

```
dumpbin SimpleSection.obj /SUMMARY
Microsoft (R) COFF/PE Dumper Version 8.00.50727.762
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file SimpleSection.obj

File Type: COFF OBJECT

  Summary

           4 .bss
           C .data
          86 .debug$S
          18 .drectve
          4E .text
```

COFF 文件结构

几乎跟 ELF 文件一样，COFF 也是由文件头及后面的若干个段组成，再加上文件末尾的符号表、调试信息的内容，就构成了 COFF 文件的基本结构，我们在 COFF 文件中几乎都可以找到与 ELF 文件结构相对应的地方。COFF 文件的文件头部包括了两部分，一个是描述文件总体结构和属性的**映像头**（Image Header），另外一个是描述该文件中包含的段属性的**段表**（Section Table）。文件头后面紧跟着的就是文件的段，包括代码段、数据段等，最后还有符号表等。整体结构如图 5-1 所示。

映像（Image）：因为 PE 文件在装载时被直接映射到进程的虚拟空间中运行，它是进程的虚拟空间的映像。所以 PE 可执行文件很多时候被叫做**映像文件**（Image File）。

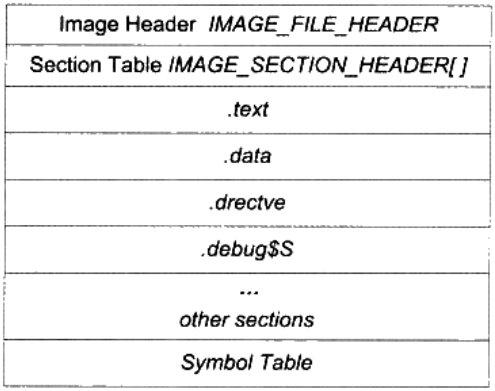

图 5-1 COFF 目标文件格式

文件头里描述 COFF 文件总体属性的映像头是一个“IMAGE_FILE_HEADER”的结构，很明显，它跟 ELF 中的“Elf32_Ehdr”结构的作用相同。这个结构及相关常数被定义在“VC\PlatformSDK\include\WinNT.h”里面：

```
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;
```

再回头对照前面“SimpleSection.txt”中的输出信息，我们可以看到输出的信息里面最开始一段“FILE HEADER VALUES”中的内容跟 COFF 映像头中的成员是一一对应的：

```
File Type: COFF OBJECT

FILE HEADER VALUES
            14C machine (x86)
              5 number of sections
       45C975E6 time date stamp Wed Feb 07 14:47:02 2007
            1E0 file pointer to symbol table
             14 number of symbols
              0 size of optional header
              0 characteristics
```

可以看到这个目标文件的文件类型是“COFF OBJECT”，也就是 COFF 目标文件格式。文件头里面还包含了目标机器类型，例子里的类型是 0x14C，微软定义该类型为 x86 兼容 CPU。按照微软的预想，PE/COFF 结构的可执行文件应该可以在不同类型的硬件平台上使用，所以预留了该字段。如果你安装了 VC 或 Windows SDK（也叫 Platform SDK），就可以

在 WinNT.h 里面找到相应的以“IMAGE_FILE_MACHINE_”开头的目标机器类型的定义。VISUAL C++里面附带的 Platform SDK 定义了 28 种 CPU 类型，从 x86 到 MIPS R 系列、ALPHA、ARM、PowerPC 等。但是由于目前 Windows 只能应用在为数不多的平台上（目前只有 x86 平台），所以我们看到的这个类型值几乎都是 0x14C。文件头里面的“Number of Sections”是指该 PE 所包含的“段”的数量。“Time date stamp”是指 PE 文件的创建时间。“File pointer to symbol table”是符号表在 PE 中的位置。“Size of optional header”是指 Optional Header 的大小，这个结构只存在于 PE 可执行文件，COFF 目标文件中该结构不存在，所以为 0，我们在后面介绍 PE 文件结构时还会提到这个成员。

映像头后面紧跟着的就是 COFF 文件的段表，它是一个类型为“IMAGE_SECTION_HEADER”结构的数组，数组里面每个元素代表一个段，这个结构跟 ELF 文件中的“Elf32_Shdr”很相似。很明显，这个数组元素的个数刚好是该 COFF 文件所包含的段的数量，也就是映像头里面的“NumberOfSections”。这个结构是用来描述每个段的属性的，它也被定义在 WinNT.h 里面：

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[8];
    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;
```

可以看到每个段所拥有的属性包括段名（Section Name）、物理地址（Physical address）、虚拟地址（Virtual address）、原始数据大小（Size of raw data）、段在文件中的位置（File pointer to raw data）、该段的重定位表在文件中的位置（File pointer to relocation table）、该段的行号表在文件中的位置（File pointer to line numbers）、标志位（Characteristics）等。我们挑几个重要的字段来进行分析，主要有 VirtualSize、VirtualAddress、SizeOfRawData 和 Characteristics 这几个字段，如表 5-1 所示。

表 5-1

字段	含义
VirtualSize	该段被加载至内存后的大小
VirtualAddress	该段被加载至内存后的虚拟地址

续表

字段	含义
SizeOfRawData	该段在文件中的大小。注意：这个值有可能跟 VirtualSize 的值不一样，比如.bss 段的 SizeOfRawData 是 0，而 VirtualSize 值是.bss 段的大小。另外涉及一些内存对齐等问题，这个值往往比 VirtualSize 小 关于.bss 的内容请阅读后面的“.bss 段”一节
Characteristics	段的属性，属性里包含的主要是段的类型（代码、数据、bss）、对齐方式及可读可写可执行等权限。段的属性是一些标志位的组合，这些标志位被定义在 WinNT.h 里，比如 IMAGE_SCN_CNT_CODE（0x00000020）表示该段里面包含的是代码；IMAGE_SCN_MEM_READ（0x40000000）表示该段在内存中是可读的；IMAGE_SCN_MEM_EXECUTE（0x20000000）表示该段在内存中是可执行的，等等

段表以后就是一个个的段的实际内容了，我们在分析 ELF 文件的过程中已经分析过代码段、数据段和 BSS 段的内容及它们的存储方式，COFF 中这几个段的内容与 ELF 中几乎一样，我们在这里也不详细介绍了。在这里我们准备介绍两个 ELF 文件中不存在的段，这两个段就是“.drectve”段和“.debug$S”段。

5.3 链接指示信息

我们将“SimpleSection.txt”中关于“.drectve”段相关的内容摘录如下：

```
SECTION HEADER #1
.drectve name
    0 physical address
    0 virtual address
    18 size of raw data
    DC file pointer to raw data (000000DC to 000000F3)
    0 file pointer to relocation table
    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
  100A00 flags
         Info
         Remove
         1 byte align

RAW DATA #1
  00000000: 20 20 20 2F 44 45 46 41 55 4C 54 4C 49 42 3A 22     /DEFAULTLIB:"
  00000010: 4C 49 42 43 4D 54 22 20                             LIBCMT"

   Linker Directives
   -----------------
   /DEFAULTLIB:"LIBCMT"
```

“.drectve 段”实际上是“Directive”的缩写，它的内容是编译器传递给链接器的指令

（Directive），即编译器希望告诉链接器应该怎样链接这个目标文件。段名后面就是段的属性，包括地址、长度、位置等我们这些在分析 ELF 时已经很熟知的属性，最后一个属性是标志位“flags”，即 IMAGE_SECTION_HEADERS 里面的 Characteristics 成员。“.drectve”段的标志位为“0x100A00”，它是表 5-2 中的标志位的组合。

表 5-2

标志位	宏定义	意义
0x00100000	IMAGE_SCN_ALIGN_1BYTES	1 个字节对齐。相当于不对齐
0x00000800	IMAGE_SCN_LNK_REMOVE	最终链接成映像文件时抛弃该段
0x00000200	IMAGE_SCN_LNK_INFO	该段包含的是注释或其他信息

“dumpbin”已经为我们打印出了标志位的三个组合属性：Info、Remove、1 byte align。即该段是信息段，并非程序数据；该段可以在最后链接成可执行文件的时候被抛弃；该段在文件中的对齐方式是 1 个字节对齐。

输出信息中紧随其后的是该段在文件中的原始数据（RAW DATA #1，用十六进制显示的原始数据及相应的 ASCII 字符）。“dumpbin”知道该段是个“.drectve”段，并且对段的内容进行了解析，解析结果为一个“/DEFAULTLIB:‘LIBCMT’”的链接指令（Linker Directives），实际上它就是“cl”编译器希望传给“link”链接器的参数。这个参数表示编译器希望告诉链接器，该目标文件须要 LIBCMT 这个默认库。LIBCMT 的全称是（Library C Multithreaded），它表示 VC 的**静态链接的多线程 C 库**，对应的文件在 VC 安装目录下的 lib/libcmt.lib，我们在前面介绍静态库链接时已经简单介绍过了。所以当我们使用“link”命令链接“SimpleSection.obj”时，链接器看到输入文件中有这个段，就会将“/DEFAULT:‘LIBCMT’”参数添加到链接参数中，即将 libcmt.lib 加入链接输入文件中。

注意 我们可以在 cl 编译器参数里面加入/Zl 来关闭默认 C 库的链接指令。

5.4 调试信息

COFF 文件中所有以“.debug”开始的段都包含着调试信息。比如“.debug$S”表示包含的是**符号**（Symbol）相关的调试信息段；“.debug$P”表示包含**预编译头文件**（Precompiled Header Files）相关的调试信息段；“.debug$T”表示包含**类型**（Type）相关的调试信息段。在“SimpleSection.obj”中，我们只看到了“.debug$S”段，也就是只有调试时的相关信息。我们可以从该段的文本信息中看到目标文件的原始路径，编译器信息等。调试信息段的具体格式被定义在 **PE 格式文件标准**中，我们在这里就不详细展开了。调试段相关信息在“SimpleSection.txt”中的内容如下：

```
SECTION HEADER #2
.debug$S name
    0 physical address
    0 virtual address
    86 size of raw data
    F4 file pointer to raw data (000000F4 to 00000179)
    0 file pointer to relocation table
    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
42100040 flags
        Initialized Data
        Discardable
        1 byte align
        Read Only

RAW DATA #2
  00000000: 02 00 00 00 46 00 09 00 00 00 00 00 3F 43 3A 5C  ....F.......?C:\
  00000010: 57 6F 72 6B 69 6E 67 5C 62 6F 6F 6B 5C 63 6F 64  Working\book\cod
  00000020: 65 5C 43 68 61 70 74 65 72 20 32 5C 53 69 6D 70  e\Chapter 2\Simp
  00000030: 6C 65 53 65 63 74 69 6F 6E 73 5C 53 69 6D 70 6C  leSections\Simpl
  00000040: 65 53 65 63 74 69 6F 6E 2E 6F 62 6A 38 00 13 10  eSection.obj8...
  00000050: 00 22 00 00 07 00 0E 00 00 00 27 C6 0E 00 00 00  ."........'?....
  00000060: 27 C6 21 4D 69 63 72 6F 73 6F 66 74 20 28 52 29  '?!Microsoft (R)
  00000070: 20 4F 70 74 69 6D 69 7A 69 6E 67 20 43 6F 6D 70   Optimizing Comp
```

5.5 大家都有符号表

“SimpleSection.txt”的最后部分是 COFF 符号表（Symbol table），COFF 文件的符号表包含的内容几乎跟 ELF 文件的符号表一样，主要就是符号名、符号的类型、所在的位置。我们把“SimpleSection.txt”关于符号表的输出摘录如下：

```
COFF SYMBOL TABLE
000 006DC627 ABS    notype         Static        | @comp.id
001 00000001 ABS    notype         Static        | @feat.00
002 00000000 SECT1  notype         Static        | .drectve
    Section length   18, #relocs   0, #linenums  0, checksum    0
004 00000000 SECT2  notype         Static        | .debug$S
    Section length   86, #relocs   0, #linenums  0, checksum    0
006 00000004 UNDEF  notype         External      | _global_uninit_var
007 00000000 SECT3  notype         Static        | .data
    Section length    C, #relocs   0, #linenums  0, checksum AC5AB941
009 00000000 SECT3  notype         External      | _global_init_var
00A 00000004 SECT3  notype         Static        | $SG594
00B 00000008 SECT3  notype         Static
                                   | ?static_var@?1??main@@9@9
(`main'::`2'::static_var)
00C 00000000 SECT4  notype         Static        | .text
    Section length   4E, #relocs    5, #linenums 0, checksum CC61DB94
00E 00000000 SECT4  notype ()      External      | _func1
00F 00000000 UNDEF  notype ()      External      | _printf
010 00000020 SECT4  notype ()      External      | _main
```

```
011 00000000 SECT5  notype         Static      | .bss
    Section length    4, #relocs   0, #linenums    0, checksum      0
013 00000000 SECT5  notype         Static      | ?static_var2@?1??main@@9@9
('main'::'2'::static_var2)
```

在输出结果的最左列是符号的编号，也是符号在符号表中的下标。接着是符号的大小，即符号所表示的对象所占用的空间。第三列是符号所在的位置，ABS（Absolute）表示符号是个绝对值，即一个常量，它不存在于任何段中；SECT1（Section #1）表示符号所表示的对象定义在本 COFF 文件的第一个段中，即本例中的“.drectve”段；UNDEF（Undefined）表示符号是未定义的，即这个符号被定义在其他目标文件。第四列是符号类型，可以看到对于 C 语言的符号，COFF 只区分了两种，一种是变量和其他符号，类行为 notype，另外一种是函数，类型为 notype ()，这个符号类型值可以用于其他一些需要强符号类型的语言或系统中，可以给链接器更多的信息来识别符号的类型。第五列是符号的可见范围，Static 表示符号是局部变量，只有目标文件内部是可见的；External 表示符号是全局变量，可以被其他目标文件引用。最后一列是符号名，对于不需要修饰的符号名，“dumpbin”直接输出原始的符号名；对于那些经过修饰的符号名，它会把修饰前和修饰后的名字都打印出来，后面括号里面的就是未修饰的符号名。

从符号表的 dump 输出信息中，我们可以看到“_global_init_varabal”这个符号位于 Section #3，即“.data”段，它的长度是 4 个字节，可见范围是全局。另外还有一个为$SG574 的符号，其实它表示的是程序中的那个“%d\n”字符串常量。因为程序中要引用到这个字符串常量，而该字符串常量又没有名字，所以编译器自动为它生成了一个名字，并且作为符号放在符号表里面，可以看到这个符号对外部是不可见的。可以看到，ELF 文件中并没有为字符串常量自动生成的符号，另外所有的段名都是一个符号，“dumpbin”如果碰到某个符号是一个段的段名，那么它还会解析该符号所表示的段的基本属性，每个段名符号后面紧跟着一行就是段的基本属性，分别是段长度、重定位数、行号数和校验和。

5.6 Windows 下的 ELF——PE

PE 文件是基于 COFF 的扩展，它比 COFF 文件多了几个结构。最主要的变化有两个：第一个是文件最开始的部分不是 COFF 文件头，而是 **DOS MZ 可执行文件格式的文件头和桩代码**（DOS MZ File Header and Stub）；第二个变化是原来的 COFF 文件头中的“IMAGE_FILE_HEADER”部分扩展成了 PE 文件文件头结构“IMAGE_NT_HEADERS”，这个结构包括了原来的“Image Header”及新增的 **PE 扩展头部结构**（PE Optional Header）。PE 文件的结构如图 5-2 所示。

DOS 下的可执行文件的扩展名与 Windows 下的可执行文件扩展名一样，都是“.exe”，但是 DOS 下的可执行文件格式是“MZ”格式（因为这个格式比较古老，我们在这里并不打

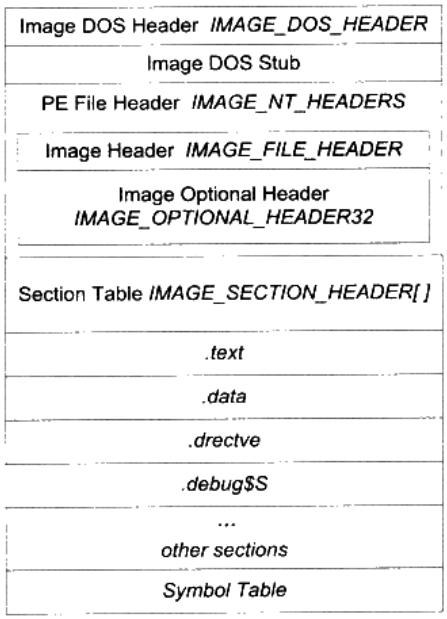

图 5-2 PE 文件格式

算展开介绍这种格式），与 Windows 下的 PE 格式完全不同，虽然它们使用相同的扩展名。在 Windows 发展的早期，那时候 DOS 系统还如日中天，而且早期的 Windows 版本还不能脱离 DOS 环境独立运行，所以为了照顾 DOS 系统，那些为 Windows 编写的程序必须尽量兼容原有的 DOS 系统，所以 PE 文件在设计之初就背负着历史的累赘。PE 文件中“Image DOS Header”和“DOS Stub”这两个结构就是为了兼容 DOS 系统而设计的，其中“IMAGE_DOS_HEADER”结构其实跟 DOS 的“MZ”可执行结构的头部完全一样，所以从某个角度看，PE 文件其实也是一个“MZ”文件。“IMAGE_DOS_HEADER”的结构中有的前两个字节是“e_magic”结构，它是里面包含了“MZ”这两个字母的 ASCII 码；“e_cs”和“e_ip”两个成员指向程序的入口地址。

当 PE 可执行映像在 DOS 下被加载的时候，DOS 系统检测该文件，发现最开始两个字节是“MZ”，于是认为它是一个“MZ”可执行文件。然后 DOS 系统就将 PE 文件当作正常的“MZ”文件开始执行。DOS 系统会读取“e_cs”和“e_ip”这两个成员的值，以跳转到程序的入口地址。然而 PE 文件中，“e_cs”和“e_ip”这两个成员并不指向程序真正的入口地址，而是指向文件中的“DOS Stub”。“DOS Stub”是一段可以在 DOS 下运行的一小段代码，这段代码的唯一作用是向终端输出一行字：“This program cannot be run in DOS”，然后退出程序，表示该程序不能在 DOS 下运行。所以我们如果在 DOS 系统下运行 Windows 的程序就可以看到上面这句话，这是因为 PE 文件结构兼容 DOS“MZ”可执行文件结构的缘故。

“IMAGE_DOS_HEADER”结构也被定义在 WinNT.h 里面，该结构的大多数成员我们都不关心，唯一值得关心的是“e_lfanew”成员，这个成员表明了 PE 文件头（IMAGE_NT_HEADERS）在 PE 文件中的偏移，我们须要使用这个值来定位 PE 文件头。这个成员在 DOS 的“MZ”文件格式中它的值永远为 0，所以当 Windows 开始执行一个后缀名为“.exe”的文件时，它会判断“e_lfanew”成员是否为 0。如果为 0，则该“.exe”文件是一个 DOS“MZ”可执行文件，Windows 会启动 DOS 子系统来执行它；如果不为 0，那么它就是一个 Windows 的 PE 可执行文件，“e_lfanew”的值表示“IMAGE_NT_HEADERS”在文件中的偏移。

“IMAGE_NT_HEADERS”是 PE 真正的文件头，它包含了一个**标记（Signature）**和两个结构体。标记是一个常量，对于一个合法的 PE 文件来说，它的值为 0x00004550，按照小端字节序，它对应的是'P'、'E'、'\0'、'\0'这 4 个字符的 ASCII 码。文件头包含的两个结构分别是**映像头（Image Header）**、**PE 扩展头部结构（Image Optional Header）**。这个结构定义如下：

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

“Image Header”我们在介绍 COFF 目标文件结构时已经和“SectionTable”一起介绍过了。这里新出现的是 PE 扩展头部结构，这个结构的字面意思是“可选”（Optional），也就是说不是必须的，但实际上对于 **PE 可执行文件（包括 DLL）**来说，它是必需的。这里的可选可能是相对于 COFF 目标文件来说的。该结构里面包含了很多重要的信息，同样，我们可以在“WinNT.h”里面找到该结构的定义：

```
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;
```

我们这里所讨论的“Optional Image Header”是 32 位版本的“IMAGE_OPTIONAL_HEADER32”。因为 64 位的 Windows 也采用 PE 结构，所以也就有了 64 位的 PE 可执行文件格式。为了区别这两种格式，Windows 中把 32 位的 PE 文件格式叫做 PE32，把 64 位的 PE 文件格式叫做 PE32+。这两种格式就像 ELF32 和 ELF64 一样，都大同小异，只不过关于地址和长度的一些成员从 32 位扩展成了 64 位，还增加了若干个额外的成员之外，没有其他区别。“WinNT.h”里面定义了 64 位版本的“Optional Image Header”，叫做“IMAGE_OPTIONAL_HEADER64”。

我们平时可以使用“IMAGE_OPTIONAL_HEADER”作为“Optional Image Header”的定义。它是一个宏，在 64 位的 Windows 下，Visual C++在编译时会定义“_WIN64”这个宏，那么“IMAGE_OPTIONAL_HEADER”就被定义成“IMAGE_OPTIONAL_HEADER64”；32 位 Windows 下没有定义“_WIN64”这个宏，那么它就是 IMAGE_OPTIONAL_HEADER32。跟 ELF 文件中一样，我们这里只介绍 32 位版本的格式，64 位的格式与 32 位区别不大。

“Optional Header”里面有很多成员，有些部分跟 PE 文件的装载与运行相关。我们不打算先在这里一一列举所有成员的具体含义，只是挑选一部分跟静态链接有关的加以介绍，其他的成员在本书的其他部分会再次回顾。这些成员很多都是跟 Windows 系统相关联的，很多关于 Windows 系统的编程书籍上也都会有介绍，也可以在 Microsoft 的 MSDN 上找到关于它们的信息。

5.6.1 PE 数据目录

在 Windows 系统装载 PE 可执行文件时，往往须要很快地找到一些装载所须要的数据结构，比如导入表、导出表、资源、重定位表等。这些常用的数据的位置和长度都被保存在了

一个叫**数据目录**（Data Directory）的结构里面，其实它就是前面“IMAGE_OPTIONAL_HEADER”结构里面的“DataDirectory”成员。这个成员是一个“IMAGE_DATA_DIRECTORY”的结构数组，相关的定义如下：

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16
```

可以看到这个数组的大小为16，IMAGE_DATA_DIRECTORY结构有两个成员，分别是虚拟地址以及长度。DataDirectory数组里面每一个元素都对应一个包含一定含义的表。“WinNT.h”里面定义了一些以“IMAGE_DIRECTORY_ENTRY_”开头的宏，数值从0到15，它们实际上就是相关的表的宏定义在数组中的下标。比如“IMAGE_DIRECTORY_ENTRY_EXPORT”被定义为0，所以这个数组的第一个元素所包含的地址和长度就是**导出表**（Export Table）所在的地址和长度。

这个数组中还包含其他的表，比如导入表、资源表、异常表、重定位表、调试信息表、线程私有存储（TLS）等的地址和长度。这些表多数跟装载和DLL动态链接有关，与静态链接没什么关系，所以我们在此不展开分析。在本书的第3部分我们会经常碰到这些表，在这里我们只要通过解析DataDirectory结构了解这些表的位置和长度就可以了。

5.7 本章小结

在这一章中，我们介绍了Windows下的可执行文件和目标文件格式PE/COFF。PE/COFF文件与ELF文件非常相似，它们都是基于段的结构的二进制文件格式。Windows下最常见的目标文件格式就是COFF文件格式，微软的编译器产生的目标文件都是这种格式。COFF文件有一个很有意思的段叫“.drectve段”，这个段中保存的是编译器传递给链接器的命令行参数，可以通过这个段实现指定运行库等功能。

Windows下的可执行文件、动态链接库等都使用PE文件格式，PE文件格式是COFF文件格式的改进版本，增加了PE文件头、数据目录等一些结构，使得能够满足程序执行时的需求。