COFF格式

　　 COFF – 通用对象文件格式（Common Object File Format），是一种很流行的对象文件格式（注意：这里不说它是“目标”文件，是为了和编译器产生的目标文件（\*.o/\*.obj）相区别，因为这种格式不只用于目标文件，库文件、可执行文件也经常是这种格式）。大家可能会经常使用VC吧？它所产生的目标文件（\*.obj）就是这种格式。其它的编译器，如GCC（GNU Compiler Collection）、ICL（Intel C/C++ Compiler）、VectorC，也使用这种格式的目标文件。不仅仅是C/C++，很多其它语言也使用这种格式的对象文件。统一格式的目标文件为混合语言编程带来了极大的方便。

　　 当然，并不是只有这一种对象文件格式。常用格式的还有OMF-对象模型文件（Object Module File）以及ELF-可执行及连接文件格式（Executable and Linking Format）。OMF是一大群IT巨头在n年制定的一种格式，在Windows平台上很常见。大家喜欢的Borland公司现在使用的目标文件就是这种格式。MS和Intel在n年前用的也是这种格式，现在都改投异侧，用COFF格式了。ELF格式在非Windows平台上使用得比较多，在Windows平台基本上没见过。做为程序员，很有必要认识一下这些你经常打交道的家伙！不过这次让我介绍COFF先！

　　 COFF的文件结构

　　 让我们先来看一下COFF文件的整体结构，看看它到底长得什么样！

　　File Header

　　Optional Header

　　Section Header 1

　　......

　　Section Header n

　　Section Data

　　Relocation Directives

　　Line Numbers

　　Symbol Table

　　String Table

　　如左图：

　　COFF文件一共有8种数据，自上而下分别为：

　　1. 文件头（File Header）

　　2. 可选头（Optional Header）

　　3. 段落头（Section Header）

　　4. 段落数据（Section Data）

　　5. 重定位表（Relocation Directives）

　　6. 行号表（Line Numbers）

　　7. 符号表（Symbol Table）

　　8. 字符串表（String Table）

　　 其中，除了段落头可以有多个节（因为可以有多个段落）以外，其它的所有类型的节最多只能有一个。

　　 文件头：顾名思义，它就是COFF文件的头，它用来保存COFF文件的基本信息，如文件标识，各个表的位置等等。

　　 可选头：再顾名思义，它也是一个头，还是可选的，而且可有可无。在目标文件中，基本上都没有这个头；但在其它的文件中（如：可执行文件）这个段用来保存在文件头中没有描述到的信息。

　　 段落头：又顾……（不顾了，再顾有人要打我了J），这个头（怎么这么多的头啊？！）是用来描述段落信息的，每个段落都有一个段落头来描述。段落的数目在文件头中会指出。

　　 段落数据：这通常是COFF文件中最大的数据段，每个段落真正的数据就保存在这个位置。至于怎么区分这些数据是哪个段落的，不要问我，去问段落头。

　　 重定位表：这个表通常只存在于目标文件中，它用来描述COFF文件中符号的重定位信息。至于为什么要重定位，请回家看看你的操作系统的书籍。

　　 符号表：这个表用来保存COFF文件中所用到的所有符号的信息，连接多个COFF文件时，这个表帮助我们重定位符号。调试程序时也要用到它。

　　 字符串表：不用我说，大家也知道它用来保存字符串的。可是字符串保存给谁看呢？不知道了吧！？问我啊！J符号表是以记录的形式来描述符号信息的，但它只为符号名称留置了8个字符的空间，早期的小程序还将就能行，可在现在的程序中，一个符号名动不动就数十个字符，8个字符怎么能够？没办法，只好把这些名称存在字符串表中。而符号表中只记录这些字符串的位置。

　　 文件的结构大体上就是这样了。长得是丑了点，不过还算它的设计者有点远见。可扩充性设计得不错，以致于沿用至今。了解了文件的整体结构，现在让我们来逐个段落分析它。

　　 文件头

　　 文件头，自然是从文件的0偏移处开始，它的结构很简单。用C的结构描述如下：

　　typedef struct {

　　 unsigned short usMagic; // 魔法数字

　　 unsigned short usNumSec; // 段落（Section）数

　　 unsigned long ulTime; // 时间戳

　　 unsigned long ulSymbolOffset; // 符号表偏移

　　 unsigned long ulNumSymbol; // 符号数

　　 unsigned short usOptHdrSZ; // 可选头长度

　　 unsigned short usFlags; // 文件标记

　　} FILEHDR;

　　 结构中usMagic成员是一个魔法数字（Magic Number），在I386平台上的COFF文件中它的值为0x014c。如果COFF文件头中魔法数字不为0x014c，那就不用看了，这不是一个I386平台的COFF文件。其实这就是一个平台标识。

　　 第二个成员usNumSec是一个无符号短整型，它用来描述段落的数量。段落头（Section Header）的数目就是它。

　　 ulTime成员是一个时间戳，它用来描述COFF文件的建立时间。当COFF文件为一个可执行文件时，这个时间戳经常用来当做一个加密用的比对标识。

　　 ulSymbolOffset是符号表在文件中的偏移量，这是一个绝对偏移量，要从文件头开始计数。在COFF文件的其它节中，也存在这种偏移量，它们都是绝对偏移量。

　　 ulNumSymbol成员给出了符号表中符号记录的数量。

　　 usOptHdrSZ是可选头的长度，通常它为0。而可选头的类型也是从这个长度得知的，针对不同的长度，我们就要选择不同的处理方式。

　　 usFlag是COFF文件的属性标记，它标识了COFF文件的类型，COFF文件中所保存的数据等等信息。

　　 其值如下：

　　值

　　名称

　　说明

　　0x0001

　　F\_RELFLG

　　无重定位信息标记。这个标记指出COFF文件中没有重定位信息。通常在目标文件中这个标记们为0，在可执行文件中为1。

　　0x0002

　　F\_EXEC

　　可执行标记。这个标记指出 COFF 文件中所有符号已经解析， COFF 文件应该被认为是可执行文件。

　　0x0004

　　F\_LNNO

　　文件中所有行号已经被去掉。

　　0x0008

　　F\_LSYMS

　　文件中的符号信息已经被去掉。

　　0x0100

　　F\_AR32WR

　　些标记指出文件是 32 位的 Little-Endian COFF 文件。

　　 注：Little-Endian，记不得它的中文名称了。它是指数据的排列方式。比如：十六进制的0x1234以Little-Endian方式在内存中的顺序为0x34 0x12。与之相反的是Big-Endian，这种方式下，在内存中的顺序是0x12 0x34。

　　这个表的内容并不全面，但在目标文件中，常用的也就只有这些。其它的标记我将在以后介绍PE格式时给出。

　　可选头

　　 可选头接在文件头的后面，也就是从COFF文件的0x0014偏移处开始。长度可以为0。不同长度的可选头，其结构也不同。标准的可选头长度为24或28字节，通常是28啦。这里我就只介绍长度为28的可选头。（因为这种头的长度是自定义的，不同的人定义的结果就不一样，我只能选一种最常用的头来介绍，别的我也不知道）

　　这种头的结构如下：

　　typedef struct {

　　 unsigned short usMagic; // 魔法数字

　　 unsigned short usVersion; // 版本标识

　　 unsigned long ulTextSize; // 正文（text）段大小

　　 unsigned long ulInitDataSZ; // 已初始化数据段大小

　　 unsigned long ulUninitDataSZ; // 未初始化数据段大小

　　 unsigned long ulEntry; // 入口点

　　 unsigned long ulTextBase; // 正文段基址

　　 unsigned long ulDataBase; // 数据段基址（在PE32中才有）

　　} OPTHDR;

　　 第一个成员usMagic还是魔法数字，不过这回它的值应该为0x010b或0x0107。当值为0x010b时，说明COFF文件是一个一般的可执行文件；当值为，0x0107时，COFF则为一个ROM镜像文件。

　　 usVersion是COFF文件的版本，ulTextSize是这个可执行COFF的正文段长度，ulInitDataSZ和ulUninitDataSZ分别为已初始化数据段和未初始化数据段的长度。

　　 ulEntry是程序的入口点，也就是COFF载入内存时正文段的位置（EIP寄存器的值），当COFF文件是一个动态库时，入口点也就是动态库的入口函数。

　　 ulTextBase是正文段的基址。

　　 ulDataBase是数据段基址。

　　 其实在这些成员中，只要注意usMagic和ulEntry就可以了。

　　 段落头

　　 段落头紧跟在可选头的后面（如果可选头的长度为0，那么它就是紧跟在文件头后）。它的长度为36个字节，如下：

　　typedef struct {

　　 char cName[8]; // 段名

　　 unsigned long ulVSize; // 虚拟大小

　　 unsigned long ulVAddr; // 虚拟地址

　　 unsigned long ulSize; // 段长度

　　 unsigned long ulSecOffset; // 段数据偏移

　　 unsigned long ulRelOffset; // 段重定位表偏移

　　 unsigned long ulLNOffset; // 行号表偏移

　　 unsigned short ulNumRel; // 重定位表长度

　　 unsigned short ulNumLN; // 行号表长度

　　 unsigned long ulFlags; // 段标识

　　} SECHDR;

　　 这个头可是个重要的头头，我们要用到的最终信息就由它来描述。一个COFF文件可以不要其它的节，但文件头和段落头这两节是必不可少的。

　　 cName用来保存段名，常用的段名有.text，.data，.comment，.bss等。.text段是正文段，通常也就是代码段；.data是数据段，在这个数据段中所保存的数据是初始化过的数据；.bss段也可以用来保存数据，不过这里的数据是未初始化的，这个段也是一个空段；.comment段，看名字也知道，它是注释段，用来保存一些编译信息，算是对COFF文件的注释。

　　 ulVSize是段数据载入内存时的大小。只在可执行文件中有效，在目标文件中总为0。如果它的长度大于段的实际长度，则多的部分将用0来填充。

　　 ulVAddr是段数据载入或连接时的虚拟地址。对于可执行文件来说，这个地址是相对于它的地址空间而言。当可执行文件被载入内存时，这个地址就是段中数据的第一个字节的位置。而对于目标文件而言，这只是重定位时，段数据当前位置的一个偏移量。为了计算方便，便定位的计算简化，它通常设为0。

　　 ulSize这才是段中数据的实际长度，也就是段数据的长度，在读取段数据时就由它来确定要读多少字节。

　　 ulSecOffset是段数据在COFF文件中的偏移量。

　　 ulRelOffset是该段的重定位信息的偏移量。它指向了重定位表的一个记录。

　　 ulLNOffset是该段的行号表的偏移量。它指向的是行号表中的一个记录。

　　 ulNumRel是重定位信息的记录数。从ulRelOffset指向的记录开始，到第ulNumRel个记录为止，都是该段的重定位信息。

　　 ulNumLN和ulNumRel相似。不过它是行号信息的记录数。

　　 ulFlags是该段的属性标识。其值如下表：

　　值

　　名称

　　说明

　　0x0020

　　STYP\_TEXT

　　正文段标识，说明该段是代码。

　　0x0040

　　STYP\_DATA

　　数据段标识，有些标识的段将用来保存已初始化数据。

　　0x0080

　　STYP\_BSS

　　有这个标识段也是用来保存数据，不过这里的数据是未初始化数据。

　　 注意，在BSS段中，ulVSize、ulVAddr、ulSize、ulSecOffset、ulRelOffset、ulLNOffset、ulNumRel、ulNumLN的值都为0。（上表只是部分值，其它值在PE格式中介绍，后同）

　　 段数据

　　 “人”如其名，这里是保存各个段的数据的位置。不同类型的段，数据的内容、结构也不尽相同。但在目标文件中，这些数据都是原始数据（Raw Data）。不存在什么特别的格式。

　　 重定位表

　　 这个表所保存的是各个段的重定位信息。这是一张很大的表，因为所有段的重定位信息都在这个表里。各个段落头记录了自己的重定位信息的偏移和数量。要用到重定位信息时就到这个表里来读。当然，你也可以把整个重定位表看成多个重定位表，每个段落都有一个自己的重定位表。这个表只在目标文件中有，可执行文件中是不存在这个表的。

　　 既然有表，那么就会有记录。重定位表中的每一条记录就是一条重定位信息。这个记录的结构很简单，如下：

　　typedef struct {

　　 unsigned long ulAddr; // 定位偏移

　　 unsigned long ulSymbol; // 符号

　　 unsigned short usType; // 定位类型

　　} RELOC;

　　 有够简单吧，一共只三个成员！ulAddr是要定位的内容在段内偏移。比如：一个正文段，起始位置为0x010，ulAddr的值为0x05，那你的定位信息就要写在0x15处。而且信息的长度要看你的代码的类型，32位的代码要写4个字节，16位的就只要字2个字节。

　　 ulSymbol是符号索引，它指向符号表中的一个记录。注意，这里是索引，不是偏移！它只是符号表中的一个记录的记录号。这个成员指明了重定位信息所对映的符号。

　　usType是重定位类型的标识。32位代码中，通常只用两种定位方式。一是绝对定位，二是相对定位。其代码如下：

　　值

　　名称

　　说明

　　6

　　RELOC\_ADDR32

　　32位绝对定位。

　　20

　　RELOC\_REL32

　　32位相对定位。

　　 对于不同的处理器，这些值也不尽相同。这里给出的是i386平台上最常用的两个种定位方式的标识。

　　 其定位方式如下：

　　 绝对定位

　　 在绝对定位方式下，你要给出符号的绝对地址（注意，有时候这里可能不是地址，而是值，对于常量来说，你不用给出它的地值，只用给出它的值）。当然，这个地址也不是现成的，你要用符号的相对地址＋它所在段的相对地址来得到它的绝对地址。

　　 公式：符号绝对地址＝段偏移＋符号偏移

　　 这些偏移量你要分别从段落头和符号表中得到。当段落要重定位时，当然还要先重定位段落，才能定位其中的符号。

　　 相对定位

　　 相对定位要复杂一些。它所要的地址信息是相对于当前位置的偏移，这个当前位置就是ulAddr所指向的这个偏移的绝对地址后四个字节（32位代码是四个字节，16位是两个字节）的位置。也就是用定位偏移＋当前段偏移＋机器字长÷8

　　 公式：当前地址＝定位偏移＋当前段偏移＋机器字长÷8

　　 有了当前地址，相对地址就好计算了。只要用符号的绝对地址减去当前地址就可以了。

　　 公式：相对地址＝符号绝对地址-当前地址

　　 计算好了地址，把它写到ulAddr所指向的位置，就一切OK！你已经完成了重定位的工作了。

　　 行号表

　　 行号表在调试时很有用。它把可执行的二进制代码与源代码的行号之间建立了对映关系。这样，当程序执行不正确时（其实正确的也可以J），我们就可以根据当前执行代码的位置得知出错源代码的行号，再加以修改。如果没有它的话，鬼才知道是哪一行出了毛病！

　　 它的格式也很简单。只有两个成员，如下：

　　typedef struct {

　　 unsigned long ulAddrORSymbol; // 代码地址或符号索引

　　 unsigned short usLineNo; // 行号

　　} LINENO;

　　 让我们先看第二个成员，usLineNo。这是一个从1开始计数的计数器，它代表源代码的行号。第一个成员ulAddrORSymbol在行号大于0时，代表源代码的地址；而当行号为0时，它就成了行号所对映的符号在符号表中的索引。下面让我们来看看符号表吧！

　　 符号表

　　 符号表是对象文件中用来保存符号信息的一张表，也是COFF文件中最为复杂的一张表。所有段落使用到的符号都在这个表里。它也是由很多条记录组成，每条记录都以如下结构保存：

　　typedef struct {

　　 union {

　　 char cName[8]; // 符号名称

　　 struct {

　　 unsigned long ulZero; // 字符串表标识

　　 unsigned long ulOffset; // 字符串偏移

　　 } e;

　　 } e;

　　 unsigned long ulValue; // 符号值

　　 short iSection; // 符号所在段

　　 unsigned short usType; // 符号类型

　　 unsigned char usClass; // 符号存储类型

　　 unsigned char usNumAux; // 符号附加记录数

　　} SYMENT;

　　 cName符号名称，和前面所有的名称一样，它也是8个字节，但不同的是它在一个联合体中。和它占相同的存储空间的还有ulZero和ulOffset这两个成员。如果符号的名称只有8个字符，那很好，可以直接放到这个cName中；可是，如果名称的长度大于8个字节，这里就放不下了，只好放到字符串表中。这时候，ulZero的值就会为0，而在ulOffset中会给出我们所用的符号的名称在字符串表中的偏移。

　　 一个符号有了名称不够，它还要有值！ulValue就是这个符号所代表的值。

　　 iSection成员指出了这个符号所在的段落。如果它的值为0，那么这个符号就是一个外部符号，要从其它的COFF文件中解析（连接多个目标文件就是要解析这种符号）。当它的值为-1时，说明这个符号的值是一个常量，不是它在段落中的偏移。而当它的值为-2时，这个符号只是一个调试符号，只有在调试时才会用到它。当它大于0时，才是符号所在段的索引值。

　　 usType是符号的类型标识。它用来说明这个符号的类型，是函数？整型？还是其它什么。这个标识是两个字节。

　　 低字节的低四位是基本标识，它指出了符号的基本类型，如整型，字符，结构，联合等。高四位指出了符号的高级类型，如指针（0001b），函数（0010b），数组（0011b），无类型（0000b）等。现在的编译器，通常不使用基本类型，只使用高级类型。所以，符号的基本类型通常被设为0。

　　高字节通常未用。

　　 usClass是符号的存储类型标识。它指明了符号的存储方式。

　　 其值与意义见下表：

　　值

　　名称

　　说明

　　NULL

　　0

　　无存储类型。

　　AUTOMATIC

　　1

　　自动类型。通常是在栈中分配的变量。

　　EXTERNAL

　　2

　　外部符号。当为外部符号时，iSection的值应该为0，如果不为0，则ulValue为符号在段中的偏移。

　　STATIC

　　3

　　静态类型。ulValue为符号在段中的偏移。如果偏移为0，那么这个符号代表段名。

　　REGISTER

　　4

　　寄存器变量。

　　MEMBER\_OF\_STRUCT

　　8

　　结构成员。ulValue值为该符号在结构中的顺序。

　　STRUCT\_TAG

　　10

　　结构标识符。

　　MEMBER\_OF\_UNION

　　11

　　联合成员。ulValue值为该符号在联合中的顺序。

　　UNION\_TAG

　　12

　　联合标识符。

　　TYPE\_DEFINITION

　　13

　　类型定义。

　　FUNCTION

　　101

　　函数名。

　　FILE

　　102

　　文件名。

　　 最后一个成员usNumAux是附加记录的数量。附加记录是用来描述符号的一些附加信息，为了便于保存，这些附加记录通常选择成为一条符号信息记录的整数倍（多数为1）。所以，如果这个成员的值为1，那么就表示在当前符号信息记录后附加了一条记录，用来保存附加信息。

　　 附加信息的结构是与符号的类型以及存储类型相关的。不同的类型的符号，其附加信息（如果有的话）的结构也不同。如果你不在意这些内容，也可以把它们乎略。

　　 当段的类型为FILE时，附加信息就是一个字符串，它是目标文件对应源文件的名称。其它类型在介绍PE时再进行详细讨论。

　　 字符串表

　　 不用多说，瞎子也能看出这个表是用来保存字符串的。它紧接在符号表后。至于为什么要保存字符串，前面已经说过了。这里就不再多说了，只说说字符串的保存格式。

　　 字符串表是所有节中最简单一节。如下图：

　　0 4

　　字符串表长度

　　字符串1\0

　　....

　　字符串n\0

　　 字符串表的前四个字节是字符串表的长度，以字节为单位。其后就是以0结尾的字符串（C风格字符串）。要注意的是，字符串表的长度不仅仅是字符串的长度（这个长度要包括每个字符串后的‘\0’）的总合，它还包括这个长度域的四个字节。符号表中ulOffset成员所指出的偏移就是从字符串表起始处的偏移。比如：指像每一个字符串的符号，ulOffset的值总为4。

　　 下面给出的代码，是从字符串表中读取字符串的典型C代码。

　　int iStrlen,iCur=4; // iStrLen是字符串表的长度，iCur是当前字符串偏移

　　char \*str; // 字符串表

　　read(fn, &iStrlen, 4); // 得到字符串表长度

　　str = (char \*)malloc(iStrlen); // 为字符串表分配空间

　　while (iCur<iStrlen ) // 读字符串表，直到全部读入内存

　　 iCur+=read(fn, str+iCur, iStrlen- iCur);

　　iCur=4; // 把当前字符串偏移指到每一个字符串

　　while (iCur<iStrlen ) { // 显示每一个字符串

　　 printf("String offset 0x%04X : %s\n", iCur, str + iCur);

　　 iCur+=(strlen(str+iCur)+1); // 计算偏移时不要忘了计算‘\0’字符所占的1个字节！

　　}

　　free(str); // 释放字符串表空间

　　 直到这里，整个COFF的结构已经全部介绍完了。很多了解PE格式的朋友一定会奇怪，好像少了很多内容！？是的，标准的COFF文件只有这么多的东西。但MS为了和DOS的可执行文件兼容，以及对可执行文件功能的扩展，在COFF格式中加了很多它自己的标准。让我差点就认不出COFF了。但了解了COFF文件以后，再来学习PE文件的格式，那就很简单了。

　　 想了解PE文件的格式？网上有很多它的资料，我将在本文的基础上再写几篇文章，分别介绍PE，OMF以及ELF的格式。

　　 现在大家可以自己动手，写一个COFF文件解析器或是一个简单的连接程序了！