ELF文件详解

ELF初体验

elf 目标文件有三种类型:

- 可重定位文件(Relocatable File) .o)包含适合于与其他目标文件链接来创建可执行文件或者共享目标文件的代码和数据。
- 可执行文件(Executable File) .exe) 包含适合于执行的一个程序，此文件规定了exec() 如何创建一个程序的进程映像。
- 共享目标文件(Shared Object File) .so) 包含可在两种上下文中链接的代码和数据。首先链接编辑器可以将它和其它可重定位文件和共享目标文件一起处理， 生成另外一个目标文件。其次动态链接器(Dynamic Linker)可能将它与某 个可执行文件以及其它共享目标一起组合，创建进程映像。

目标文件全部是程序的二进制表示，目的是直接在某种处理器上直接执行。

ELF文件格式

目标文件格式

目标文件既要参与程序链接又要参与程序执行。出于方便性和效率考虑，目标文件 格式提供了两种并行视图，分别反映了这些活动的不同需求。

<table>
<tr><th>链接视图</th><th>执行视图</th></tr>
<tr><td>ELF 头部</td><td>ELF 头部</td></tr>
<tr><td>程序头部表（可选）</td><td>程序头部表</td></tr>
<tr><td>节区 1</td><td rowspan="2">段 1</td></tr>
<tr><td>...</td></tr>
<tr><td>节区 n</td><td rowspan="2">段 2</td></tr>
<tr><td>...</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>节区头部表</td><td>节区头部表（可选）</td></tr>
</table>

文件开始处是一个 ELF 头部(ELF Header)，用来描述整个文件的组织。节区部 分包含链接视图的大量信息:指令、数据、符号表、重定位信息等等。

程序头部表(Program Header Table)，如果存在的话，告诉系统如何创建进程映像。 用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。

节区头部表(Section Heade Table)包含了描述文件节区的信息，每个节区在表中 都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包 含节区头部表，其他目标文件可以有，也可以没有这个表。

注意： 尽管图中显示的各个组成部分是有顺序的，实际上除了 ELF 头部表以外， 其他节区和段都没有规定的顺序

目标文件中数据表示

目标文件格式支持 8 位字节/32 位体系结构。不过这种格式是可以扩展的，比如现在的64位机器，目标文件因此以某些机器独立的格式表达某些控制数据，使得能够以一种公共的方式来识别和解释其内容。目标文件中的其它数据使用目标处理器的编码结构，而不管文件在何种机器上创建。

名称	大小	对齐	目的
Elf32_Addr	4	4	无符号程序地址
Elf32_Half	2	2	无符号中等整数
Elf32_Off	4	4	无符号文件偏移
Elf32_SWord	4	4	有符号大整数
Elf32_Word	4	4	无符号大整数
unsigned char	1	1	无符号小整数

目标文件中的所有数据结构都遵从“自然”大小和对齐规则。如果必要，数据结构可 以包含显式的补齐，例如为了确保 4 字节对象按 4 字节边界对齐。数据对齐同样适用于文件内部。

ELF Hearder部分

文件的最开始几个字节给出如何解释文件的提示信息。这些信息独立于处理器，也 独立于文件中的其余内容。ELF Header 部分可以用以下的数据结构表示:

```
/* ELF Header */
typedef struct elfhdr {
    unsigned char    e_ident[EI_NIDENT]; /* ELF Identification */
    Elf32_Half    e_type;        /* object file type */
    Elf32_Half    e_machine;    /* machine */
    Elf32_Word    e_version;    /* object file version */
    Elf32_Addr    e_entry;    /* virtual entry point */
    Elf32_Off    e_phoff;    /* program header table offset */
    Elf32_Off    e_shoff;    /* section header table offset */
    Elf32_Word    e_flags;    /* processor-specific flags */
    Elf32_Half    e_ehsize;    /* ELF header size */
    Elf32_Half    e_phentsize;    /* program header entry size */
    Elf32_Half    e_phnum;    /* number of program header entries */
    Elf32_Half    e_shentsize;    /* section header entry size */
    Elf32_Half    e_shnum;    /* number of section header entries */
    Elf32_Half    e_shstrndx;    /* section header table's "section
```

```
                        header string table" entry offset */
} Elf32_Ehdr;

typedef struct {
    unsigned char    e_ident[EI_NIDENT];    /* Id bytes */
    Elf64_Quarter    e_type;            /* file type */
    Elf64_Quarter    e_machine;        /* machine type */
    Elf64_Half    e_version;        /* version number */
    Elf64_Addr    e_entry;        /* entry point */
    Elf64_Off    e_phoff;        /* Program hdr offset */
    Elf64_Off    e_shoff;        /* Section hdr offset */
    Elf64_Half    e_flags;        /* Processor flags */
    Elf64_Quarter    e_ehsize;        /* sizeof ehdr */
    Elf64_Quarter    e_phentsize;        /* Program header entry size */
    Elf64_Quarter    e_phnum;        /* Number of program headers */
    Elf64_Quarter    e_shentsize;        /* Section header entry size */
    Elf64_Quarter    e_shnum;        /* Number of section headers */
    Elf64_Quarter    e_shstrndx;        /* String table index */
} Elf64_Ehdr;
```

其中，e_ident 数组给出了 ELF 的一些标识信息，这个数组中不同下标的含义如表所示:

名称	取值	目的
EI_MAG0	0	文件标识
EI_MAG1	1	文件标识
EI_MAG2	2	文件标识
EI_MAG3	3	文件标识
EI_CLASS	4	文件类
EI_DATA	5	数据编码
EI_VERSION	6	文件版本
EI_PAD	7	补齐字节开始处
EI_NIDENT	16	e_ident[]大小

这些索引访问包含以下数值的字节:

索引	说明
EI_MAG0 到 EI_MAG3	魔数（Magic Number），标志此文件是一个 ELF 目标文件。 名称 取值 位置 EI_MAG0 0x7f e_ident[EI_MAG0] EI_MAG1 'E' e_ident[EI_MAG1] EI_MAG2 'L' e_ident[EI_MAG2] EI_MAG3 'F' e_ident[EI_MAG3]
EI_CLASS	标识文件的类别，或者说，容量。 名称 取值 位置 ELFCLASSNONE 0 非法类别 ELFCLASS32 1 32 位目标 ELFCLASS64 2 64 位目标 ELFCLASS32 支持虚存范围 4 GB。ELFCLASS64 是为 64 位预留的，不过文件中的其他内容都没有针对 64 位定义。
EI_DATA	字节 e_ident[EI_DATA] 给出处理器特定数据的数据编码方式。 名称 取值 位置 ELFDATANONE 0 非法数据编码 ELFDATA2LSB 1 高位在前 ELFDATA2MSB 2 低位在前
EI_VERSION	ELF 头部的版本号码，不前此值必须是 EV_CURRENT。
EI_PAD	标记 e_ident 中未使用字节的开始。初始化为 0。

e_ident[EI_MAG0]~e_ident[EI_MAG3]即e_ident[0]~e_ident[3]被称为魔数（Magic Number）,其值一般为0x7f,'E','L','F'。 e_ident[EI_CLASS]（即e_ident[4]）识别目标文件运行在目标机器的类别，取值可为三种值：ELFCLASSNONE（0）非法类别；ELFCLASS32（1）32位目标；ELFCLASS64（2）64位目标。 e_ident[EI_DATA]（即e_ident[5]）：给出处理器特定数据的数据编码方式。即大端还是小端方式。取值可为3种：ELFDATANONE（0）非法数据编码；ELFDATA2LSB（1）高位在前；ELFDATA2MSB（2）低位在前。

ELF Header 中各个字段的说明如表：

成员	说明
e_ident	目标文件标识
e_type	目标文件类型： 名称 取值 含义 ET_NONE 0 未知目标文件格式 ET_REL 1 可重定位文件 ET_EXEC 2 可执行文件 ET_DYN 3 共享目标文件 ET_CORE 4 Core 文件（转储格式） ET_LOPROC 0xff00 特定处理器文件 ET_HIPROC 0xffff 特定处理器文件 ET_LOPROC 和 ET_HIPROC 之间的取值用来标识与处理器相关的文件格式。
e_machine	给出文件的目标体系结构类型 名称 取值 含义 EM_NONE 0 未指定 EM_M32 1 AT&T WE 32100 EM_SPARC 2 SPARC EM_386 3 Intel 80386 EM_68K 4 Motorola 68000 EM_88K 5 Motorola 88000 EM_860 7 Intel 80860 EM_MIPS 8 MIPS RS3000 其它值都是保留的。特定处理器的 ELF 名称会使用机器名来进行区分。
e_version	目标文件版本 名称 取值 含义 EV_NONE 0 非法版本 EV_CURRENT 1 当前版本
e_entry	程序入口的虚拟地址。如果目标文件没有程序入口，可以为 0。
e_phoff	程序头部表格（Program Header Table）的偏移量（按字节计算）。如果文件没有程序头部表格，可以为 0。
e_shoff	节区头部表格（Section Header Table）的偏移量（按字节计算）。如果文件没有节区头部表格，可以为 0。
e_flags	保存与文件相关的，特定于处理器的标志。标志名称采用 EF_machine_flag 的格式。
e_ehsize	ELF 头部的大小（以字节计算）。
e_phentsize	程序头部表格的表项大小（按字节计算）。
e_phnum	程序头部表格的表项数目。可以为 0。
e_shentsize	节区头部表格的表项大小（按字节计算）。

e_shnum	节区头部表格的表项数目。可以为 0。
e_shstrndx	节区头部表格中与节区名称字符串表相关的表项的索引。如果文件没有节区名称字符串表，此参数可以为 SHN_UNDEF。

一个实际可执行文件的头文件头部形式如下：

```
$readelf -h hello.so
ELF 头:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF32
  数据:                              2 补码，小端序 (little endian)
  版本:                              1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              DYN (共享目标文件)
  系统架构:                          ARM
  版本:                              0x1
  入口点地址:                        0x0
  程序头起点:                        52 (bytes into file)
  Start of section headers:          61816 (bytes into file)
  标志:                              0x5000000, Version5 EABI
  本头的大小:                        52 (字节)
  程序头大小:                        32 (字节)
  Number of program headers:         9
  节头大小:                          40 (字节)
  节头数量:                          24
  字符串表索引节头:                  23
```

注：linux环境下可以利用命令readelf,mac需要安装binutils。用brew安装很方便，命令：`brew update && brew install binutils`

程序头部（Program Header）

可执行文件或者共享目标文件的程序头部是一个结构数组，每个结构描述了一个段 或者系统准备程序执行所必需的其它信息。目标文件的“段”包含一个或者多个“节区”， 也就是“段内容(Segment Contents)”。程序头部仅对于可执行文件和共享目标文件 有意义。 可执行目标文件在 ELF 头部的 e_phentsize 和 e_phnum 成员中给出其自身程序头部 的大小。程序头部的数据结构:

```
/* Program Header */
typedef struct {
    Elf32_Word    p_type;        /* segment type */
    Elf32_Off    p_offset;    /* segment offset */
    Elf32_Addr    p_vaddr;    /* virtual address of segment */
    Elf32_Addr    p_paddr;    /* physical address - ignored? */
    Elf32_Word    p_filesz;    /* number of bytes in file for seg. */
    Elf32_Word    p_memsz;    /* number of bytes in mem. for seg. */
    Elf32_Word    p_flags;    /* flags */
    Elf32_Word    p_align;    /* memory alignment */
} Elf32_Phdr;
```

```
typedef struct {
    Elf64_Half    p_type;        /* entry type */
    Elf64_Half    p_flags;    /* flags */
    Elf64_Off    p_offset;    /* offset */
    Elf64_Addr    p_vaddr;    /* virtual address */
    Elf64_Addr    p_paddr;    /* physical address */
    Elf64_Xword    p_filesz;    /* file size */
    Elf64_Xword    p_memsz;    /* memory size */
    Elf64_Xword    p_align;    /* memory & file alignment */
} Elf64_Phdr;
```

其中各个字段说明：

- p_type 此数组元素描述的段的类型，或者如何解释此数组元素的信息。具体如下图。
- p_offset 此成员给出从文件头到该段第一个字节的偏移。
- p_vaddr 此成员给出段的第一个字节将被放到内存中的虚拟地址。
- p_paddr 此成员仅用于与物理地址相关的系统中。因为 System V 忽略所有应用程序的物理地址信息，此字段对与可执行文件和共享目标文件而言具体内容是指定的。
- p_filesz 此成员给出段在文件映像中所占的字节数。可以为 0。
- p_memsz 此成员给出段在内存映像中占用的字节数。可以为 0。
- p_flags 此成员给出与段相关的标志。
- p_align 可加载的进程段的 p_vaddr 和 p_offset 取值必须合适，相对于对页面大小的取模而言。此成员给出段在文件中和内存中如何 对齐。数值 0 和 1 表示不需要对齐。否则 p_align 应该是个正整数，并且是 2 的幂次数，p_vaddr 和 p_offset 对 p_align 取模后应该相等。

可执行 ELF 目标文件中的段类型：

名字	取值	说明
PT_NULL	0	此数组元素未用。结构中其他成员都是未定义的。
PT_LOAD	1	此数组元素给出一个可加载的段，段的大小由 p_filesz 和 p_memsz 描述。文件中的字节被映射到内存段开始处。如果 p_memsz 大于 p_filesz，“剩余”的字节要清零。p_filesz 不能大于 p_memsz。可加载的段在程序头部表格中根据 p_vaddr 成员按升序排列。
PT_DYNAMIC	2	数组元素给出动态链接信息。
PT_INTERP	3	数组元素给出一个 NULL 结尾的字符串的位置和长度，该字符串将被当作解释器调用。这种段类型仅对与可执行文件有意义（尽管也可能在共享目标文件上发生）。在一个文件中不能出现一次以上。如果存在这种类型的段，它必须在所有可加载段项目的前面。
PT_NOTE	4	此数组元素给出附加信息的位置和大小。
PT_SHLIB	5	此段类型被保留，不过语义未指定。包含这种类型的段的程序与 ABI 不符。
PT_PHDR	6	此类型的数组元素如果存在，则给出了程序头部表自身的大小和位置，既包括在文件中也包括在内存中的信息。此类型的段在文件中不能出现一次以上。并且只有程序头部表是程序的内存映像的一部分时才起作用。如果存在此类型段，则必须在所有可加载段项目的前面。
PT_LOPROC	0x70000000	此范围的类型保留给处理器专用语义。
PT_HIPROC	0x7fffffff	

节区（Sections）

节区中包含目标文件中的所有信息，除了:ELF 头部、程序头部表格、节区头部 表格。节区满足以下条件:

1. 目标文件中的每个节区都有对应的节区头部描述它，反过来，有节区头部不意 味着有节区。
2. 每个节区占用文件中一个连续字节区域(这个区域可能长度为 0)。
3. 文件中的节区不能重叠，不允许一个字节存在于两个节区中的情况发生。
4. 目标文件中可能包含非活动空间(INACTIVE SPACE)。这些区域不属于任何头部和节区，其内容指定。

节区头部表格

ELF 头部中，e_shoff 成员给出从文件头到节区头部表格的偏移字节数;e_shnum 给出表格中条目数目;e_shentsize 给出每个项目的字节数。从这些信息中可以确切地定 位节区的具体位置、长度。

每个节区头部数据结构描述:

```
/* Section Header */
typedef struct {
    Elf32_Word    sh_name;    /* name - index into section header
                       string table section */
    Elf32_Word    sh_type;    /* type */
    Elf32_Word    sh_flags;    /* flags */
    Elf32_Addr    sh_addr;    /* address */
    Elf32_Off    sh_offset;    /* file offset */
    Elf32_Word    sh_size;    /* section size */
    Elf32_Word    sh_link;    /* section header table index link */
    Elf32_Word    sh_info;    /* extra information */
    Elf32_Word    sh_addralign;    /* address alignment */
    Elf32_Word    sh_entsize;    /* section entry size */
} Elf32_Shdr;

typedef struct {
    Elf64_Half    sh_name;    /* section name */
    Elf64_Half    sh_type;    /* section type */
    Elf64_Xword    sh_flags;    /* section flags */
    Elf64_Addr    sh_addr;    /* virtual address */
    Elf64_Off    sh_offset;    /* file offset */
    Elf64_Xword    sh_size;    /* section size */
    Elf64_Half    sh_link;    /* link to another */
    Elf64_Half    sh_info;    /* misc info */
    Elf64_Xword    sh_addralign;    /* memory alignment */
    Elf64_Xword    sh_entsize;    /* table entry size */
} Elf64_Shdr;
```

对其中各个字段的解释如下:

成员	说明
sh_name	给出节区名称。是节区头部字符串表节区（Section Header String Table Section）的索引。名字是一个 NULL 结尾的字符串。
sh_type	为节区的内容和语义进行分类。参见节区类型。
sh_flags	节区支持 1 位形式的标志，这些标志描述了多种属性。
sh_addr	如果节区将出现在进程的内存映像中，此成员给出节区的第一个字节应处的位置。否则，此字段为 0。
sh_offset	此成员的取值给出节区的第一个字节与文件头之间的偏移。不过，SHT_NOBITS 类型的节区不占用文件的空间，因此其 sh_offset 成员给出的是其概念性的偏移。
sh_size	此成员给出节区的长度（字节数）。除非节区的类型是 SHT_NOBITS，否则节区占用文件中的 sh_size 字节。类型为 SHT_NOBITS 的节区长度可能非零，不过却不占用文件中的空间。
sh_link	此成员给出节区头部表索引链接。其具体的解释依赖于节区类型。
sh_info	此成员给出附加信息，其解释依赖于节区类型。
sh_addralign	某些节区带有地址对齐约束。例如，如果一个节区保存一个 doubleword，那么系统必须保证整个节区能够按双字对齐。sh_addr 对 sh_addralign 取模，结果必须为 0。目前仅允许取值为 0 和 2 的幂次数。数值 0 和 1 表示节区没有对齐约束。
sh_entsize	某些节区中包含固定大小的项目，如符号表。对于这类节区，此成员给出每个表项的长度字节数。 如果节区中并不包含固定长度表项的表格，此成员取值为 0。

索引为零（SHN_UNDEF）的节区头部是存在的，尽管此索引标记的是未定义的节区应用。这个节区固定为：

字段名称	取值	说明
sh_name	0	无名称
sh_type	SHT_NULL	非活动
sh_flags	0	无标志
sh_addr	0	无地址
sh_offset	0	无文件偏移
sh_size	0	无尺寸大小
sh_link	SHN_UNDEF	无链接信息
sh_info	0	无辅助信息
sh_addralign	0	无对齐要求
sh_entsize	0	无表项

sh_type 字段

其他的节区类型是保留的。

sh_flags 字段

sh_flags 字段定义了一个节区中包含的内容是否可以修改、是否可以执行等信息。 如果一个标志位被设置，则该位取值为 1。 定义的各位都设置为 0。

名称	取值
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MASKPROC	0xF0000000

其中已经定义了的各位含义如下:

- SHF_WRITE: 节区包含进程执行过程中将可写的数据。
- SHF_ALLOC: 此节区在进程执行过程中占用内存。某些控制节区并不出现于目标文件的内存映像中，对于那些节区，此位应设置为 0。
- SHF_EXECINSTR: 节区包含可执行的机器指令。
- SHF_MASKPROC: 所有包含于此掩码中的四位都用于处理器专用的语义。

sh_link 和 sh_info 字段

根据节区类型的不同，sh_link 和 sh_info 的具体含义也有所不同:

sh_type	sh_link	sh_info
SHT_DYNAMIC	此节区中条目所用到的字符串表格的节区头部索引	0
SHT_HASH	此哈希表所适用的符号表的节区头部索引	0
SHT_REL SHT_RELA	相关符号表的节区头部索引	重定位所适用的节区的节区头部索引
SHT_SYMTAB SHT_DYNSYM	相关联的字符串表的节区头部索引	最后一个局部符号（绑定 STB_LOCAL）的符号表索引值加一
其它	SHN_UNDEF	0

特殊节区

很多节区中包含了程序和控制信息。下面的表中给出了系统使用的节区，以及它们 的类型和属性。

在分析这些节区的时候，需要注意如下事项:

- 以“.”开头的节区名称是系统保留的。应用程序可以使用没有前缀的节区名称，以避 免与系统节区冲突。
- 目标文件格式允许人们定义不在上述列表中的节区。
- 目标文件中也可以包含多个名字相同的节区。
- 保留给处理器体系结构的节区名称一般构成为:处理器体系结构名称简写 + 节区名称。
- 处理器名称应该与 e_machine 中使用的名称相同。例如 .FOO.psect 街区是由 FOO 体系结构定义的 psect 节区。

另外，有些编译器对如上节区进行了扩展，这些已存在的扩展都使用约定俗成的名 称，如:

- .sdata
- .tdesc
- .sbss
- .lit4
- .lit8
- .reginfo
- .gptab
- .liblist
- .conflict
- ...

字符串表（String Table）

字符串表节区包含以 NULL(ASCII 码 0)结尾的字符序列，通常称为字符串。ELF 目标文件通常使用字符串来表示符号和节区名称。对字符串的引用通常以字符串在字符 串表中的下标给出。

一般，第一个字节(索引为 0)定义为一个空字符串。类似的，字符串表的最后一 个字节也定义为 NULL，以确保所有的字符串都以 NULL 结尾。索引为 0 的字符串在 不同的上下文中可以表示无名或者名字为 NULL 的字符串。

允许存在空的字符串表节区，其节区头部的 sh_size 成员应该为 0。对空的字符串 表而言，非 0 的索引值是非法的。

例如:对于各个节区而言，节区头部的 sh_name 成员包含其对应的节区头部字符串 表节区的索引，此节区由 ELF 头的 e_shstrndx 成员给出。下图给出了包含 25 个字节 的一个字符串表，以及与不同索引相关的字符串。

索引	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

上图中包含的字符串如下:

索引	字符串
0	(无)
1	name.
7	Variable
11	able
16	able
24	(空字符串)

在使用、分析字符串表时，要注意以下几点:

- 字符串表索引可以引用节区中任意字节。
- 字符串可以出现多次
- 可以存在对子字符串的引用
- 同一个字符串可以被引用多次。
- 字符串表中也可以存在未引用的字符串。

符号表（Symbol Table）

目标文件的符号表中包含用来定位、重定位程序中符号定义和引用的信息。符号表 索引是对此数组的索引。索引 0 表示表中的第一表项，同时也作为 定义符号的索引。

符号表项的格式如下:

```
/* Symbol Table Entry */
typedef struct elf32_sym {
    Elf32_Word    st_name;    /* name - index into string table */
    Elf32_Addr    st_value;    /* symbol value */
```

```
    Elf32_Word    st_size;    /* symbol size */
    unsigned char    st_info;    /* type and binding */
    unsigned char    st_other;    /* 0 - no defined meaning */
    Elf32_Half    st_shndx;    /* section header index */
} Elf32_Sym;

typedef struct {
    Elf64_Half    st_name;    /* Symbol name index in str table */
    Elf_Byte    st_info;    /* type / binding attrs */
    Elf_Byte    st_other;    /* unused */
    Elf64_Quarter    st_shndx;    /* section index of symbol */
    Elf64_Xword    st_value;    /* value of symbol */
    Elf64_Xword    st_size;    /* size of symbol */
} Elf64_Sym;
```

其中各个字段的含义说明:

字段	说明
st_name	包含目标文件符号字符串表的索引，其中包含符号名的字符串表示。如果该值非 0，则它表示了给出符号名的字符串表索引，否则符号表项没有名称。 注：外部 C 符号在 C 语言和目标文件的符号表中具有相同的名称。
st_value	此成员给出相关联的符号的取值。依赖于具体的上下文，它可能是一个绝对值、一个地址等等。
st_size	很多符号具有相关的尺寸大小。例如一个数据对象的大小是对象中包含的字节数。如果符号没有大小或者大小未知，则此成员为 0。
st_info	此成员给出符号的类型和绑定属性。下面给出若干取值和含义的绑定关系。
st_other	该成员当前包含 0，其含义没有定义。
st_shndx	每个符号表项都以和其他节区间的关系的方式给出定义。此成员给出相关的节区头部表索引。某些索引具有特殊含义。

st_info 说明

st_info 中包含符号类型和绑定信息，操纵方式如:

```
#define ELF32_ST_BIND(i)    ((i)>>4)
#define ELF32_ST_TYPE(i)    ((i)&0xf)
#define ELF32_ST_INFO(b, t) (((b)<<4) + ((t)&0xf))
```

从中可以看出，st_info 的低四位表示符号绑定，用于确定链接可见性和行为。具体的绑定类型如:

名称	取值	说明
STB_LOCAL	0	局部符号在包含该符号定义的目标文件以外不可见。相同名称的局部符号可以存在于多个文件中，互不影响。
STB_GLOBAL	1	全局符号对所有将组合的目标文件都是可见的。一个文件中对某个全局符号的定义将满足另一个文件对相同全局符号的未定义引用。
STB_WEAK	2	弱符号与全局符号类似，不过他们的定义优先级比较低。
STB_LOPROC	13	处于这个范围的取值是保留给处理器专用语义的。
STB_HIPROC	15	

全局符号与弱符号之间的区别主要有两点:

(1). 当链接编辑器组合若干可重定位的目标文件时，不允许对同名的 STB_GLOBAL 符号给出多个定义。 另一方面如果一个已定义的全局符号已经存在，出现一个同名的弱符号并 不会产生错误。链接编辑器尽关心全局符号，忽略弱符号。 类似地，如果一个公共符号(符号的 st_shndx 中包含 SHN_COMMON)，那 么具有相同名称的弱符号出现也不会导致错误。链接编辑器会采纳公共定 义，而忽略弱定义。

(2). 当链接编辑器搜索归档库(archive libraries)时，会提取那些包含 定 义全局符号的档案成员。成员的定义可以是全局符号，也可以是弱符号。 连接编辑器不会提取档案成员来满足 定义的弱符号。 能解析的弱符号取值为 0。

在每个符号表中，所有具有 STB_LOCAL 绑定的符号都优先于弱符号和全局符 号。符号表节区中的 sh_info 头部成员包含第一个非局部符号的符号表索引。

st_type 字段

在共享目标文件中的函数符号(类型为 STT_FUNC)具有特别的重要性。当其他 目标文件引用了来自某个共享目标中的函数时，链接编辑器自动为所引用的符号创建过 程链接表项。类型不是 STT_FUNC 的共享目标符号不会自动通过过程链接表进行引用。

如果一个符号的取值引用了某个节区中的特定位置，那么它的节区索引成员 (st_shndx)包含了其在节区头部表中的索引。当节区在重定位过程中被移动时，符号 的取值也会随之变化，对符号的引用始终会“指向”程序中的相同位置。

特殊的节区索引

某些特殊的节区索引具有不同的语义:

- SHN_ABS: 符号具有绝对取值，不会因为重定位而发 生变化。
- SHN_COMMON: 符号标注了一个尚 分配的公共 块。符号的取值给出了对齐约束，与节区的 sh_addralign 成员类似。就是说，链接编辑器将为符号分配存储空间， 地址位于 st_value 的倍数处。符号的大小给出了所需要
  的字节数。
- SHN_UNDEF: 此节区表索引值意味着符号没有定义。
  当链接编辑器将此目标文件与其他定义了该符号的目标 文件进行组合时，此文件中对该符号的引用将被链接到实 际定义的位置。

STN_UNDEF 符号

如上所述，符号表中下标为 0(STN_UNDEF)的表项被保留。其中包含如下数值:

名称	取值	说明
st_name	0	无名称
st_value	0	0 值
st_size	0	无大小
st_info	0	无类型，局部绑定
st_other	0	无附加信息
st_shndx	0	无节区

st_value 字段

不同的目标文件类型中符号表项对 st_value 成员具有不同的解释:

(1). 在可重定位文件中，st_value 中遵从了节区索引为 SHN_COMMON 的符号的对齐约束。

(2). 在可重定位的文件中，st_value 中包含已定义符号的节区偏移。就是说，st_value 是从 st_shndx 所标识的节区头部开始计算，到符号位置的偏移。

(3). 在可执行和共享目标文件中，st_value 包含一个虚地址。为了使得这些 文件的符号对动态链接器更有用，节区偏移(针对文 件的解释)让位于 虚拟地址(针对内存的解释)，因为这时与节区号无关。

尽管符号表取值在不同的目标文件中具有相似的含义，适当的程序可以采取高效的数据访问方式。

重定位信息

重定位是将符号引用与符号定义进行连接的过程。例如，当程序调用了一个函数时， 相关的调用指令必须把控制传输到适当的目标执行地址。

重定位表项

可重定位文件必须包含如何修改其节区内容的信息，从而允许可执行文件和共享目 标文件保存进程的程序映像的正确信息。重定位表项就是这样一些数据。

重定位表项的格式:

```
/* Relocation entry with implicit addend */
typedef struct {
    Elf32_Addr    r_offset;    /* offset of relocation */
    Elf32_Word    r_info;        /* symbol table index and type */
} Elf32_Rel;

/* Relocation entry with explicit addend */
typedef struct {
    Elf32_Addr    r_offset;    /* offset of relocation */
    Elf32_Word    r_info;        /* symbol table index and type */

    Elf32_Sword    r_addend;
```

```
} Elf32_Rela;

typedef struct {
    Elf64_Xword    r_offset;    /* where to do it */
    Elf64_Xword    r_info;        /* index & type of relocation */
} Elf64_Rel;

typedef struct {
    Elf64_Xword    r_offset;    /* where to do it */
    Elf64_Xword    r_info;        /* index & type of relocation */
    Elf64_Sxword    r_addend;    /* adjustment value */
} Elf64_Rela;
```

如上所述，只有 Elf32_Rela 项目可以明确包含补齐信息。类型为 Elf32_Rel 的表 项在将被修改的位置保存隐式的补齐信息。 依赖于处理器体系结构，各种形式都可能 存在，甚至是必需的。因此，对特定机器的实现可以仅使用一种形式，也可以根据上下 文使用不同的形式。

重定位节区会引用两个其它节区:符号表、要修改的节区。节区头部的 sh_info 和 sh_link 成员给出这些关系。不同目标文件的重定位表项对 r_offset 成员具有略微不 同的解释。

(1). 在可重定位文件中，r_offset 中包含节区偏移。就是说重定位节区自身 描述了如何修改文件中的其他节区;重定位偏移 指定了被修改节区中的 一个存储单元。

(2). 在可执行文件和共享的目标文件中，r_offset 中包含一个虚拟地址。为 了使得这些文件的重定位表项对动态链接器更为有用，节区偏移(针对文 件的解释)让位于虚地址(针对内存的解释)。

尽管对 r_offset 的解释会有少许不同，重定位类型的含义始终不变。

重定位类型

重定位表项描述如何修改后面的指令和数据字段。一般，共享目标文件在创建时， 其基 虚拟地址是 0，不过执行地址将随着动态加载而发生变化。

重定位的过程，按照如下标记:

- A 用来计算可重定位字段的取值的补齐。
- B 共享目标在执行过程中被加载到内存中的位置(基地址)。
- G 在执行过程中，重定位项的符号的地址所处的位置 —— 全局偏移表的索引。
- GOT 全局偏移表(GOT)的地址。
- L 某个符号的过程链接表项的位置(节区偏移/地址)。过程链接表项把函数调用重定位到正确的目标位置。链接编辑器构造初始的过程链接表，动态链接器在执行过程中修改这些项目。
- P 存储单位被重定位(用 r_offset 计算)到的位置(节区偏移或者地址)。
- S 其索引位于重定位项中的符号的取值。

重定位项的 r_offset 取值给定受影响的存储单位的第一个字节的偏移或者虚拟地 址。重定位类型给出那些位需要修改以及如何计算它们的取值。

SYSTEM V仅使用Elf32_Rel重定位表项，在被重定位的字段中包含补齐量。补 齐量和计算结果始终采用相同的字节顺序。