链接程序和库指南

Previous: 第 6 章 支持接口 **Next**: 动态链接

文件格式

目标文件既可用于程序链接，也可用于程序执行。为了方便和提高效率，目标文件格式提供了文件内容的平行视图，以便反映这些活动的不同需要。下图显示了目标文件的结构。

图 7–1 目标文件格式

ELF 头位于目标文件的起始位置，其中包含用于说明文件结构的**指南**。

注 –

仅有 ELF 头在文件中具有固定位置。由于ELF 格式具有灵活性，因此不要求头表、节或段具有指定的顺序。但是，此图是 Solaris 中使用的典型布局。

节表示 ELF 文件中可以处理的最小不可分割单位。**段**是节的集合。段表示可由 exec(2) 或运行时链接程序映射到内存映像的最小独立单位。

节包含链接视图的批量目标文件信息。此数据包括指令、数据、符号表和重定位信息。本章的第一部分提供了各节的说明。本章的第二部分讨论了各段以及文件的程序执行视图。

程序头表（如果存在）指示系统如何创建进程映像。用于生成进程映像、可执行文件和共享库的文件必须具有程序头表。可重定位目标文件无需程序头表。

节头表包含说明文件各节的信息。每节在表中有一个与之对应的项。每一项都指定了节名和节大小之类的信息。链接编辑过程中使用的文件必须具有节头表。

数据表示形式

目标文件格式支持 8 位字节、32 位体系结构和 64 位体系结构的各种处理器。不过，数据表示形式最好可扩展为更大或更小的体系结构。表 7–1 和 表 7–2 列出了 32 位数据类型和 64 位数据类型。

目标文件表示格式与计算机无关的一些控制数据。此格式可提供目标文件的通用标识和解释。目标文件中的其余数据使用目标处理器的编码，无论在什么计算机上创建该文件都是如此。

表 7–1 ELF 32 位数据类型

名称	大小	对齐	用途
Elf32_Addr	4	4	无符号程序地址
Elf32_Half	2	2	无符号中整数
Elf32_Off	4	4	无符号文件偏移
Elf32_Sword	4	4	带符号整数
Elf32_Word	4	4	无符号整数
unsigned char	1	1	无符号小整数

表 7–2 ELF 64 位数据类型

名称	大小	对齐	目的
Elf64_Addr	8	8	无符号程序地址
Elf64_Half	2	2	无符号中整数
Elf64_Off	8	8	无符号文件偏移
Elf64_Sword	4	4	带符号整数
Elf64_Word	4	4	无符号整数
Elf64_Xword	8	8	无符号长整数
Elf64_Sxword	8	8	带符号长整数
unsigned char	1	1	无符号小整数

目标文件格式定义的所有数据结构都遵循相关类别的自然大小和对齐规则。数据结构可以包含显式填充，以确保 4 字节目标文件的 4 字节对齐，从而强制结构大小为 4 的倍数，依此类推。数据在文件的开头也会适当对齐。例如，包含 `Elf32_Addr` 成员的结构在文件中与 4 字节边界对齐。同样，包含 `Elf64_Addr` 成员的结构与 8 字节边界对齐。

注 –

为便于移植，ELF 不使用位字段。

ELF 头

目标文件中的一些控制结构可以增大，因为 ELF 头包含这些控制结构的实际大小。如果目标文件格式发生变化，则程序可能会遇到大于或小于所需大小的控制结构。因此，程序可能会忽略额外信息。这些忽略的信息的处理方式取决于上下文，如果定义了扩展内容，则会指定处理方式。

ELF 头具有以下结构。 请参见 `sys/elf.h` 。

```
#define EI_NIDENT       16


typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;


typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf64_Half      e_type;
        Elf64_Half      e_machine;
        Elf64_Word      e_version;
        Elf64_Addr      e_entry;
        Elf64_Off       e_phoff;
        Elf64_Off       e_shoff;
        Elf64_Word      e_flags;
        Elf64_Half      e_ehsize;
        Elf64_Half      e_phentsize;
        Elf64_Half      e_phnum;
        Elf64_Half      e_shentsize;
        Elf64_Half      e_shnum;
        Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

e_ident

将文件标记为目标文件的初始字节。这些字节可提供与计算机无关的数据，用于解码和解释文件的内容。ELF 标识中提供了完整说明。

e_type

标识目标文件类型，如下表中所列。

名称	值	含义
ET_NONE	0	无文件类型
ET_REL	1	可重定位文件
ET_EXEC	2	可执行文件
ET_DYN	3	共享库文件
ET_CORE	4	核心转储文件
ET_LOPROC	0xff00	特定于处理器
ET_HIPROC	0xffff	特定于处理器

虽然未指定核心转储文件内容，但类型 ET_CORE 保留用于标记文件。从 ET_LOPROC 到 ET_HIPROC 之间的值（包括这两个值）保留用于特定于处理器的语义。其他值保留供将来使用。

e_machine

指定独立文件所需的体系结构。下表中列出了相关体系结构。

名称	值	含义
EM_NONE	0	无计算机
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_SPARC32PLUS	18	Sun SPARC 32+
EM_SPARCV9	43	SPARC V9
名称 EM_AMD64	**值** 62	**含义** AMD 64

其他值保留供将来使用。特定于处理器的 ELF 名称通过使用计算机名来进行区分。例如，为 e_flags 定义的标志会使用前缀 EF_。 EM_XYZ 计算机的名为 WIDGET 的标志可称为 EF_XYZ_WIDGET 。

e_version

标识目标文件版本，如下表中所列。

名称	值	含义
EV_NONE	0	无效版本
EV_CURRENT	>=1	当前版本

值 1 表示原始文件格式。 EV_CURRENT 的值可根据需要进行更改，以反映当前版本号。

e_entry

系统首先将控制权转移到的虚拟地址，从而启动进程。如果文件没有关联的入口点，则此成员值为零。

e_phoff

程序头表的文件偏移（以字节为单位）。如果文件没有程序头表，则此成员值为零。

e_shoff

节头表的文件偏移（以字节为单位）。如果文件没有节头表，则此成员值为零。

e_flags

与文件关联的特定于处理器的标志。标志名称采用 EF_machine _**flag** 形式。对于 x86，此成员目前为零。下表中列出了 SPARC 标志。

名称	值	含义
EF_SPARC_EXT_MASK	0xffff00	供应商扩展掩码
EF_SPARC_32PLUS	0x000100	通用 V8+ 功能
EF_SPARC_SUN_US1	0x000200	Sun UltraSPARC™ 1 扩展
EF_SPARC_HAL_R1	0x000400	HAL R1 扩展
EF_SPARC_SUN_US3	0x000800	Sun UltraSPARC 3 扩展
EF_SPARCV9_MM	0x3	内存型号掩码
EF_SPARCV9_TSO	0x0	总体存储排序
EF_SPARCV9_PSO	0x1	部分存储排序
EF_SPARCV9_RMO	0x2	非严格存储排序

名称 值 含义

e_ehsize

ELF 头的大小（以字节为单位）。

e_phentsize

文件的程序头表中某一项的大小（以字节为单位）。所有项的大小都相同。

e_phnum

程序头表中的项数。生成的 e_phentsize 和 e_phnum 指定了表的大小（以字节为单位）。如果文件没有程序头表，则 e_phnum 值为零。

e_shentsize

节头的大小（以字节为单位）。节头是节头表中的一项。所有项的大小都相同。

e_shnum

节头表中的项数。生成的 e_shentsize 和 e_shnum 指定了节头表的大小（以字节为单位）。如果文件没有节头表，则 e_shnum 值为零。

如果节数大于或等于 SHN_LORESERVE （0xff00），则 e_shnum 值为零。节头表的实际项数包含在节头表中索引为 0 的 sh_size 字段中。否则，初始节头项的 sh_size 成员值为零。请参见表 7–6 和表 7–7。

e_shstrndx

与节名字符串表关联的项的节头表索引。如果文件没有节名字符串表，则此成员值为 SHN_UNDEF 。

如果节名字符串表的节索引大于或等于 SHN_LORESERVE （0xff00），则此成员值为 SHN_XINDEX （0xffff），节名字符串表的实际节索引包含在节头中索引为 0 的 sh_link 字段中。否则，初始节头项的 sh_link 成员值为零。请参见表 7–6 和表 7–7。

ELF 标识

ELF 提供了一个目标文件框架，用于支持多个处理器、多种数据编码和多类计算机。要支持此目标文件系列，文件的初始字节应指定解释文件的方式。这些字节与发出查询的处理器以及文件的其余内容无关。

ELF 头和目标文件的初始字节对应于 e_ident 成员。

表 7–3 ELF 标识索引

名称	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_OSABI	7	操作系统/ABI 标识
EI_ABIVERSION	8	ABI 版本

名称	值	目的
EI_PAD	9	填充字节的开头
EI_NIDENT	16	e_ident[] 的大小

这些索引可访问值为以下各项的字节。

EI_MAG0 - EI_MAG3

4 字节**魔数**，用于将文件标识为 ELF 目标文件，如下表中所列。

名称	值	位置
ELFMAG0	0x7f	e_ident[EI_MAG0]
ELFMAG1	'E'	e_ident[EI_MAG1]
ELFMAG2	'L'	e_ident[EI_MAG2]
ELFMAG3	'F'	e_ident[EI_MAG3]

EI_CLASS

字节 e_ident[EI_CLASS] 用于标识文件的类或容量，如下表中所列。

名称	值	含义
ELFCLASSNONE	0	无效类
ELFCLASS32	1	32 位目标文件
ELFCLASS64	2	64 位目标文件

文件格式设计用于在各种大小的计算机之间进行移植，而不会将最大计算机的大小强加给最小的计算机。文件类可定义目标文件容器的数据结构所使用的基本类型。包含在目标文件各节中的数据可以遵循其他编程模型。

类 ELFCLASS32 支持文件和虚拟地址空间最高为 4 GB 的计算机。该类使用表 7–1 中定义的基本类型。

类 ELFCLASS64 保留用于 64 位体系结构，如 64 位 SPARC 和 x64。该类使用表 7–2 中定义的基本类型。

EI_DATA

字节 e_ident[EI_DATA] 用于指定目标文件中特定于处理器的数据的数据编码，如下表中所列。

名称	值	含义

ELFDATANONE	0	无效数据编码
ELFDATA2LSB	1	请参见图 7–2。
ELFDATA2MSB	2	请参见图 7–3。

数据编码一节中提供了有关这些编码的更多信息。其他值保留供将来使用。

EI_VERSION

字节 e_ident[EI_VERSION] 用于指定 ELF 头版本号。当前，该值必须为 EV_CURRENT 。

EI_OSABI

字节 e_ident[EI_OSABI] 用于标识操作系统以及目标文件所面向的 ABI。其他 ELF 结构中的一些字段包含的标志和值具有特定于操作系统或 ABI 的含义。这些字段的解释由此字节的值确定。

EI_ABIVERSION

字节 e_ident[EI_ABIVERSION] 用于标识目标文件所面向的 ABI 的版本。此字段用于区分 ABI 的各个不兼容版本。此版本号的解释依赖于 EI_OSABI 字段标识的 ABI。如果没有为对应于处理器的 EI_OSABI 字段指定值，或者没有为 EI_OSABI 字节的特定值所确定的 ABI 指定版本值，则会使用值零来表示未指定的值。

EI_PAD

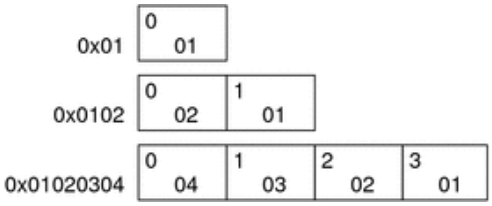
该值用于标记 e_ident 中未使用字节的起始位置。这些字节会保留并设置为零。读取目标文件的程序应忽略这些值。

数据编码

文件的数据编码指定解释文件中的基本目标文件的方式。类 ELFCLASS32 文件使用将占用 1、2 和 4 个字节的目标文件。类 ELFCLASS64 文件使用将占用 1、2、4 和 8 个字节的目标文件。按照定义的编码，目标文件使用如下描述的数字表示。字节编号显示在左上角。

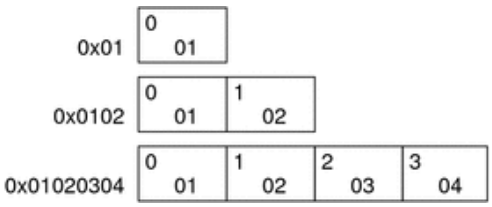
ELFDATA2LSB 编码用于指定 2 的补码值，其中最低有效字节占用最低地址。

图 7–2 数据编码 ELFDATA2LSB



ELFDATA2MSB 编码用于指定 2 的补码值，其中最高有效字节占用最低地址。

图 7–3 数据编码 ELFDATA2MSB



节

使用目标文件的节头表，可以定位文件的所有节。节头表是 Elf32_Shdr 或 Elf64_Shdr 结构的数组。节头表索引是此数组的下标。ELF 头的 e_shoff 成员表示从文件的起始位置到节头表的字节偏移。 e_shnum 成员表示节头表包含的项数。 e_shentsize 成员表示每一项的大小（以字节为单位）。

如果节数大于或等于 SHN_LORESERVE （0xff00），则 e_shnum 值为 SHN_UNDEF （0）。节头表的实际项数包含在节头表中索引为 0 的 sh_size 字段中。否则，初始项的 sh_size 成员值为零。

如果上下文中限制了索引大小，则会保留部分节头表索引。例如，符号表项的 st_shndx 成员和 ELF 头的 e_shnum 和 e_shstrndx 成员。在这类上下文中，保留的值不表示目标文件中的实际各节。同样在这类上下文中，转义值表示会在其他位置（较大字段中）找到实际节索引。

表 7–4 ELF 特殊节索引

名称	值
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_BEFORE	0xff00
SHN_AFTER	0xff01
SHN_AMD64_LCOMMON	0xff02
SHN_HIPROC	0xff1f
SHN_LOOS	0xff20
SHN_LOSUNW	0xff3f
SHN_SUNW_IGNORE	0xff3f
SHN_HISUNW	0xff3f
SHN_HIOS	0xff3f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_XINDEX	0xffff
SHN_HIRESERVE	0xffff

注 –

虽然索引 0 保留作为未定义的值，但节头表包含对应于索引 0 的项。即，如果 ELF 头的 e_shnum 成员表示文件在节头表中具有 6 项，则这些节的索引为 0 到 5 。初始项的内容会在本节的后面指定。

SHN_UNDEF

未定义、缺少、无关或无意义的节引用。例如，**已定义**的与节数 SHN_UNDEF 有关的符号即是未定义符号。

SHN_LORESERVE

所保留索引的范围的下边界。

SHN_LOPROC - SHN_HIPROC

此范围内包含的值保留用于特定于处理器的语义。

SHN_LOOS - SHN_HIOS

此范围内包含的值保留用于特定于操作系统的语义。

SHN_LOSUNW - SHN_HISUNW

此范围内包含的值保留用于特定于 Sun 的语义。

SHN_SUNW_IGNORE

此节索引用于在可重定位目标文件中提供临时符号定义。保留供 dtrace(1M) 内部使用。

SHN_BEFORE, SHN_AFTER

与 SHF_LINK_ORDER 和 SHF_ORDERED 节标志一起用于初始和最终节排序。 请参见表 7–8。

SHN_AMD64_LCOMMON

特定于 x64 的通用块标签。此标签与 SHN_COMMON 类似，但用于标识较大的通用块。

SHN_ABS

对应引用的绝对值。例如，已定义的与节数 SHN_ABS 相关的符号具有绝对值，并且不受重定位影响。

SHN_COMMON

已定义的与此节相关的符号为通用符号，如 FORTRAN COMMON 或未分配的 C 外部变量。这些符号有时称为暂定符号。

SHN_XINDEX

转义值，用于表示实际节头索引过大，以致无法放入包含字段。节头索引可在特定于显示节索引的结构的其他位置中找到。

SHN_HIRESERVE

所保留索引的范围的上边界。系统保留了 SHN_LORESERVE 和 SHN_HIRESERVE 之间的索引（包括这两个值）。这些值不会引用节头表。节头表不包含对应于所保留索引的项。

节包含目标文件中的所有信息，但 ELF 头、程序头表和节头表除外。此外，目标文件中的各节还满足多个条件：

- 目标文件中的每一节仅有一个说明该节的节头。可能会有节头存在但节不存在的情况。
- 每一节在文件中占用可能为空的相邻的一系列字节。
- 文件中的各节不能重叠。文件中的字节不能位于多个节中。
- 目标文件可以包含非活动空间。各种头和节可能不会包括目标文件中的每个字节。非活动数据的内容未指定。

节头具有以下结构。 请参见 sys/elf.h 。

```
typedef struct {
        elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
```

```
        Elf32_Word      sh_entsize;
} Elf32_Shdr;


typedef struct {
        Elf64_Word      sh_name;
        Elf64_Word      sh_type;
        Elf64_Xword     sh_flags;
        Elf64_Addr      sh_addr;
        Elf64_Off       sh_offset;
        Elf64_Xword     sh_size;
        Elf64_Word      sh_link;
        Elf64_Word      sh_info;
        Elf64_Xword     sh_addralign;
        Elf64_Xword     sh_entsize;
} Elf64_Shdr;
```

sh_name

节的名称。此成员值是节头字符串表的节索引，用于指定以空字符结尾的字符串的位置。表 7–10 中列出了节名及其说明。

sh_type

用于将节的内容和语义分类。表 7–5 中列出了节类型及其说明。

sh_flags

节可支持用于说明杂项属性的 1 位标志。表 7–8 中列出了标志定义。

sh_addr

如果节显示在进程的内存映像中，则此成员会指定节的第一个字节所在的地址。否则，此成员值为零。

sh_offset

从文件的起始位置到节中第一个字节的字节偏移。对于 `SHT_NOBITS` 节，此成员表示文件中的概念性偏移，因为该节在文件中不占用任何空间。

sh_size

节的大小（以字节为单位）。除非节类型为 `SHT_NOBITS` ，否则该节将在文件中占用 `sh_size` 个字节。 `SHT_NOBITS` 类型的节大小可以不为零，但该节在文件中不占用任何空间。

sh_link

节头表索引链接，其解释依赖于节类型。表 7–9 说明了相应的值。

sh_info

额外信息，其解释依赖于节类型。表 7–9 说明了相应的值。

sh_addralign

一些节具有地址对齐约束。例如，如果某节包含双字，则系统必须确保整个节双字对齐。在此情况下， `sh_addr` 的值在以 `sh_addralign` 的值为模数进行取模时，同余数必须等于 `0` 。当前，仅允许使用 `0` 和 2 的正整数幂。值 `0` 和 `1` 表示节没有对齐约束。

sh_entsize

一些节包含固定大小的项的表，如符号表。对于这样的节，此成员会指定每一项的大小（以字节为单位）。如果节不包含固定大小的项的表，则此成员值为零。

节头的 `sh_type` 成员用于指定节的语义，如下表中所示。

表 7–5 ELF 节类型 *sh_type*

名称	值
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7
SHT_NOBITS	8
SHT_REL	9
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_INIT_ARRAY	14
SHT_FINI_ARRAY	15
SHT_PREINIT_ARRAY	16
SHT_GROUP	17
SHT_SYMTAB_SHNDX	18
SHT_LOOS	0x60000000

名称	值
SHT_LOSUNW	0x6ffffff4
SHT_SUNW_dof	0x6ffffff4
SHT_SUNW_cap	0x6ffffff5
SHT_SUNW_SIGNATURE	0x6ffffff6
SHT_SUNW_ANNOTATE	0x6ffffff7
SHT_SUNW_DEBUGSTR	0x6ffffff8
SHT_SUNW_DEBUG	0x6ffffff9
SHT_SUNW_move	0x6ffffffa
SHT_SUNW_COMDAT	0x6ffffffb
SHT_SUNW_syminfo	0x6ffffffc
SHT_SUNW_verdef	0x6ffffffd
SHT_SUNW_verneed	0x6ffffffe
SHT_SUNW_versym	0x6fffffff
SHT_HISUNW	0x6fffffff
SHT_HIOS	0x6fffffff
SHT_LOPROC	0x70000000
SHT_SPARC_GOTDATA	0x70000000
SHT_AMD64_UNWIND	0x70000001

名称	值
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

SHT_NULL

将节头标识为非活动。此节头没有关联的节。节头的其他成员具有未定义的值。

SHT_PROGBITS

标识由程序定义的信息，这些信息的格式和含义仅由程序确定。

SHT_SYMTAB 、 SHT_DYNSYM

标识符号表。通常， SHT_SYMTAB 节会提供用于链接编辑的符号。作为完整的符号表，该表可以包含许多对于动态链接不需要的符号。因此，目标文件还可以包含 SHT_DYNSYM 节，其中包含一组尽可能少的动态链接符号，从而可节省空间。 有关详细信息，请参见符号表节。

SHT_STRTAB 、 SHT_DYNSTR

标识字符串表。目标文件可以有多个字符串表节。 有关详细信息，请参见字符串表节。

SHT_RELA

标识包含显式加数的重定位项，如 32 位类的目标文件的 Elf32_Rela 类型。目标文件可以有多个重定位节。 有关详细信息，请参见重定位节。

SHT_HASH

标识符号散列表。动态链接的目标文件必须包含符号散列表。当前，目标文件只能有一个散列表，但此限制在将来可能会放宽。 有关详细信息，请参见散列表节。

SHT_DYNAMIC

标识动态链接的信息。当前，目标文件只能有一个动态节。 有关详细信息，请参见动态节。

SHT_NOTE

标识以某种方法标记文件的信息。 有关详细信息，请参见注释节。

SHT_NOBITS

标识在文件中不占用任何空间，但在其他方面与 SHT_PROGBITS 类似的节。虽然此节不包含任何字节，但 sh_offset 成员包含概念性文件偏移。

SHT_REL

标识不包含显式加数的重定位项，如 32 位类的目标文件的 Elf32_Rel 类型。目标文件可以有多个重定位节。 有关详细信息，请参见重定位节。

SHT_SHLIB

标识具有未指定的语义的保留节。包含此类型的节的程序不符合 ABI。

SHT_INIT_ARRAY

标识包含指针数组的节，这些指针指向初始化函数。数组中的每个指针都视为不返回任何值的无参数过程。 有关详细信息，请参见初始化和终止节。

SHT_FINI_ARRAY

标识包含指针数组的节，这些指针指向终止函数。数组中的每个指针都视为不返回任何值的无参数过程。 有关详细信息，请参见初始化和终止节。

SHT_PREINIT_ARRAY

标识包含指针数组的节，这些指针指向在其他所有初始化函数之前调用的函数。数组中的每个指针都视为不返回任何值的无参数过程。 有关详细信息，请参见初始化和终止节。

SHT_GROUP

标识节组。节组可标识一组相关的节，这些节必须作为一个单位由链接编辑器进行处理。 SHT_GROUP 类型的节只能出现在可重定位目标文件中。 有关详细信息，请参见组节。

SHT_SYMTAB_SHNDX

标识包含扩展节索引的节，扩展节索引与符号表关联。如果符号表引用的任何节头索引包含转义值 SHN_XINDEX ，则需要关联的 SHT_SYMTAB_SHNDX 。

SHT_SYMTAB_SHNDX 节是 Elf32_Word 值的数组。此数组包含一项，可与关联的符号表项中的每一项对应。这些值表示针对其定义符号表各项的节头索引。仅当对应符号表项的 st_shndx 字段包含转义值 SHN_XINDEX 时，匹配的 Elf32_Word 才会包含实际节头索引。否则，该项必须为 SHN_UNDEF (0)。

SHT_LOOS - SHT_HIOS

此范围内包含的值保留用于特定于操作系统的语义。

SHT_LOSUNW - SHT_HISUNW

此范围内包含的值保留用于 Solaris 语义。

SHT_SUNW_cap

指定硬件和软件的功能要求。 有关详细信息，请参见硬件和软件功能节。

SHT_SUNW_SIGNATURE

标识模块验证签名。

SHT_SUNW_ANNOTATE

注释节的处理遵循用于处理节的所有缺省规则。仅当注释节位于不可分配的内存中时，才会发生异常。如果未设置节头标志 SHF_ALLOC ，则链接编辑器将忽略针对此节的所有不满足要求的重定位而无任何提示。

SHT_SUNW_DEBUGSTR 、 SHT_SUNW_DEBUG

标识调试信息。使用链接编辑器的 -s 选项，或者在链接编辑之后使用 strip(1)，可以将此类型的节从目标文件中删除。

SHT_SUNW_move

标识用于处理部分初始化的符号的数据。 有关详细信息，请参见移动节。

SHT_SUNW_COMDAT

标识允许将相同数据的多个副本减少为单个副本的节。 有关详细信息，请参见COMDAT 节。

SHT_SUNW_syminfo

标识其他符号信息。 有关详细信息，请参见Syminfo 表节。

SHT_SUNW_verdef

标识此文件定义的细分版本。 有关详细信息，请参见版本定义节。

SHT_SUNW_verneed

标识此文件所需的细分依赖性。 有关详细信息，请参见版本依赖性节。

SHT_SUNW_versym

标识用于说明符号与文件提供的版本定义之间关系的表。 有关详细信息，请参见版本符号节。

SHT_LOPROC - SHT_HIPROC

此范围内包含的值保留用于特定于处理器的语义。

SHT_SPARC_GOTDATA

标识特定于 SPARC 的数据，使用相对于 GOT 的寻址引用这些数据。即，相对于指定给符号 _GLOBAL_OFFSET_TABLE_ 的地址的偏移。对于 64 位 SPARC，此节中的数据必须在链接编辑时绑定到 {+-} 2^32 字节的 GOT 地址中的位置。

SHT_AMD64_UNWIND

标识特定于 x64 的数据，其中包含对应于栈展开的展开函数表的各项。

SHT_LOUSER

指定保留用于应用程序的索引范围的下边界。

SHT_HIUSER

指定保留用于应用程序的索引范围的上边界。应用程序可以使用 SHT_LOUSER 和 SHT_HIUSER 之间的节类型，而不会与当前或将来系统定义的节类型产生冲突。

其他节类型的值会保留。如前所述，即使索引 0 (SHN_UNDEF) 标记了未定义的节引用，仍会存在对应于该索引的节头。下表显示了这些值。

表 7–6 ELF 节头表项：索引 0

名称	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
sh_entsize	0	无项

如果节或程序头的数目超过 ELF 头数据大小，则节头 0 的各元素可用于定义扩展的 ELF 头属性。下表显示了这些值。

表 7–7 ELF 扩展的节头表项：索引 0

名称	值	说明
sh_name	0	无名称
sh_type	SHT_NULL	非活动
sh_flags	0	无标志

名称	值	说明
sh_addr	0	无地址
sh_offset	0	无文件偏移
sh_size	e_shnum	节头表中的项数
sh_link	e_shstrndx	与节名字符串表关联的项的节头索引
sh_info	0	无辅助信息
sh_addralign	0	无对齐
sh_entsize	0	无项

节头的 sh_flags 成员包含用于说明节属性的 1 位标志：

表 7–8 ELF 节属性标志

名称	值
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MERGE	0x10
SHF_STRINGS	0x20
SHF_INFO_LINK	0x40
SHF_LINK_ORDER	0x80
SHF_OS_NONCONFORMING	0x100
SHF_GROUP	0x200

名称	值
SHF_TLS	0x400
SHF_MASKOS	0x0ff00000
SHF_AMD64_LARGE	0x10000000
SHF_ORDERED	0x40000000
SHF_EXCLUDE	0x80000000
SHF_MASKPROC	0xf0000000

如果在 sh_flags 中设置了标志位，则该节的此属性处于**启用**状态。否则，此属性处于**禁用**状态，或者不适用。未定义的属性会保留并设置为零。

SHF_WRITE

标识在进程执行过程中应可写的节。

SHF_ALLOC

标识在进程执行过程中占用内存的节。一些控制节不位于目标文件的内存映像中。对于这些节，此属性处于禁用状态。

SHF_EXECINSTR

标识包含可执行计算机指令的节。

SHF_MERGE

标识可以将其中包含的数据合并以消除重复的节。除非还设置了 SHF_STRINGS 标志，否则该节中的数据元素大小一致。每个元素的大小在节头的 sh_entsize 字段中指定。如果还设置了 SHF_STRINGS 标志，则数据元素会包含以空字符结尾的字符串。每个字符的大小在节头的 sh_entsize 字段中指定。

SHF_STRINGS

标识包含以空字符结尾的字符串的节。每个字符的大小在节头的 sh_entsize 字段中指定。

SHF_INFO_LINK

此节头的 sh_info 字段包含节头表索引。

SHF_LINK_ORDER

此节可向链接编辑器中添加特殊排序要求。如果此节头的 sh_link 字段引用其他节（链接到的节），则会应用这些要求。如果将此节与输出文件中的其他节合并，则此节将按照与这些其他节相同的相对顺序显示。同样，链接到的节将按照与其合并的节相同的相对顺序显示。

特殊的 sh_link 值 SHN_BEFORE 和 SHN_AFTER （请参见表 7–4）表示，已排序的节将分别位于要排序的集合中其他所有各节之前或之后。如果已排序集合中的多个节包含这些特殊值之一，则会保持输入文件链接行的顺序。

此标志的一个典型用法是生成按地址顺序引用文本或数据节的表。

如果缺少 sh_link 排序信息，则合并到输出文件一个节内的单个输入文件中的各节是相邻的。这些节会与输入文件中的节一样进行相对排序。构成多个输入文件的节按照链接行顺序显示。

SHF_OS_NONCONFORMING

此节除了要求标准链接规则之外，还要求特定于操作系统的特殊处理，以避免不正确的行为。如果此节具有 sh_type 值，或者对于这些字段包含特定于操作系统范围的 sh_flags 位，并且链接编辑器无法识别这些值，则包含此节的目标文件会由于出错而被拒绝。

SHF_GROUP

此节是节组的一个成员，可能是唯一的成员。此节必须由 SHT_GROUP 类型的节引用。只能对可重定位目标文件中包含的节设置 SHF_GROUP 标志。 有关详细信息，请参见组节。

SHF_TLS

此节包含线程局部存储。进程中的每个线程都包含此数据的一个不同实例。 有关详细信息，请参见第 8 章，线程局部存储。

SHF_MASKOS

此掩码中包括的所有位都保留用于特定于操作系统的语义。

SHF_AMD64_LARGE

x64 的缺省编译模型仅用于 32 位位移。此位移限制了节的大小，并最终限制段为 2 GB。特定于处理器的 SHF_AMD64_LARGE 属性标志用于标识可包含超过 2 GB 的节。此标志允许链接使用不同代码模型的目标文件。

不包含 SHF_AMD64_LARGE 属性标志的 x64 目标文件节可以由使用小代码模型的目标文件任意引用。包含此标志的节只能由使用较大代码模型的目标文件引用。例如，x64 中间代码模型目标文件可以引用包含属性标志的节和不包含属性标志的节中的数据。但是，x64 小代码模型目标文件只能引用不包含此标志的节中的数据。

SHF_ORDERED

此节要求相对于相同类型的其他节进行排序。已排序的节会合并到由 sh_link 项指向的节中。已排序节的 sh_link 项可以指向其本身。

如果已排序节的 sh_info 项在相同输入文件中是有效节，则将基于由 sh_info 项所指向的节的输出文件内的相对排序，对已排序的节进行排序。

特殊的 sh_info 值 SHN_BEFORE 和 SHN_AFTER （请参见表 7–4）表示，已排序的节将分别位于要排序的集合中其他所有各节之前或之后。如果已排序集合中的多个节包含这些特殊值之一，则会保持输入文件链接行的顺序。

如果缺少 sh_info 排序信息，则合并到输出文件一个节内的单个输入文件中的节是相邻的。这些节会与输入文件中的节一样进行相对排序。构成多个输入文件的节按照链接行顺序显示。

SHF_EXCLUDE

此节不包括在可执行文件或共享库的链接编辑的输入中。如果还设置了 SHF_ALLOC 标志，或者存在针对此节的重定位，则会忽略此标志。

SHF_MASKPROC

此掩码中包括的所有位都保留用于特定于处理器的语义。

根据节类型，节头中的两个成员 sh_link 和 sh_info 会包含特殊信息。

表 7–9 ELF sh_link 和 sh_info 解释

sh_type	sh_link	sh_info
SHT_DYNAMIC	关联的字符串表的节头索引。	0
SHT_HASH	关联的符号表的节头索引。	0
SHT_REL SHT_RELA	关联的符号表的节头索引。	应用重定位的节的节头索引。 另请参见表 7–10 和重定位节。
SHT_SYMTAB SHT_DYNSYM	关联的字符串表的节头索引。	比上一个局部符号 STB_LOCAL 的符号表索引大一。
SHT_GROUP	关联的符号表的节头索引。	关联的符号表中项的符号表索引。指定的符号表项的名称用于提供节组的签名。
SHT_SYMTAB_SHNDX	关联的符号表的节头索引。	0

sh_type	sh_link	sh_info
SHT_SUNW_move	关联的符号表的节头索引。	0
SHT_SUNW_COMDAT	0	0
SHT_SUNW_syminfo	关联的符号表的节头索引。	关联的 .dynamic 节的节头索引。
SHT_SUNW_verdef	关联的字符串表的节头索引。	节中版本定义的编号。
SHT_SUNW_verneed	关联的字符串表的节头索引。	节中版本依赖性的编号。
SHT_SUNW_versym	关联的符号表的节头索引。	0

特殊节

包含程序和控制信息的各种节。下表中的各节由系统使用，并且具有指明的类型和属性。

表 7–10 ELF 特殊节

名称	类型	属性
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	无
.data 、 .data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.dynamic	SHT_DYNAMIC	SHF_ALLOC + SHF_WRITE
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.eh_frame_hdr	SHT_AMD64_UNWIND	SHF_ALLOC
.eh_frame	SHT_AMD64_UNWIND	SHF_ALLOC + SHF_WRITE
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

名称	类型	属性
.fini_array	SHT_FINI_ARRAY	SHF_ALLOC + SHF_WRITE
.got	SHT_PROGBITS	请参见全局偏移表（特定于处理器）
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.init_array	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE
.interp	SHT_PROGBITS	请参见程序的解释程序
.note	SHT_NOTE	无
.lbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_AMD64_LARGE
.ldata、 .ldata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_AMD64_LARGE
.lrodata、 .lrodata1	SHT_PROGBITS	SHF_ALLOC + SHF_AMD64_LARGE
.plt	SHT_PROGBITS	请参见过程链接表（特定于处理器）
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC + SHF_WRITE
.rela	SHT_RELA	无
.rel *name*	SHT_REL	请参见重定位节
.rela *name*	SHT_RELA	请参见重定位节
.rodata、 .rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	无
.strtab	SHT_STRTAB	请参阅此表后面的说明。

名称	类型	属性
.symtab	SHT_SYMTAB	请参见符号表节
.symtab_shndx	SHT_SYMTAB_SHNDX	请参见符号表节
.tbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata 、 .tdata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.SUNW_bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.SUNW_heap	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.SUNW_cap	SHT_SUNW_cap	SHF_ALLOC
.SUNW_move	SHT_SUNW_move	SHF_ALLOC
.SUNW_reloc	SHT_REL SHT_RELA	SHF_ALLOC
.SUNW_syminfo	SHT_SUNW_syminfo	SHF_ALLOC
.SUNW_version	SHT_SUNW_verdef SHT_SUNW_verneed SHT_SUNW_versym	SHF_ALLOC

.bss

构成程序的内存映像的未初始化数据。根据定义，系统在程序开始运行时会将数据初始化为零。如节类型 SHT_NOBITS 所指明的那样，此节不会占用任何文件空间。

.comment

注释信息，通常由编译系统的组件提供。此节可以由 mcs(1) 进行处理。

.data 、 .data1

构成程序的内存映像的已初始化数据。

.dynamic

动态链接信息。 有关详细信息，请参见动态节。

.dynstr

进行动态链接所需的字符串，通常是表示与符号表各项关联的名称的字符串。

.dynsym

动态链接符号表。 有关详细信息，请参见符号表节。

.eh_frame_hdr 、 **.eh_frame**

用于展开栈的调用帧信息。

.fini

可执行指令，用于构成包含此节的可执行文件或共享库的单个终止函数。 有关详细信息，请参见初始化和终止例程。

.fini_array

函数指针数组，用于构成包含此节的可执行文件或共享库的单个终止数组。 有关详细信息，请参见初始化和终止例程。

.got

全局偏移表。 有关详细信息，请参见全局偏移表（特定于处理器）。

.hash

符号散列表。 有关详细信息，请参见散列表节。

.init

可执行指令，用于构成包含此节的可执行文件或共享库的单个初始化函数。 有关详细信息，请参见初始化和终止例程。

.init_array

函数指针数组，用于构成包含此节的可执行文件或共享库的单个初始化数组。 有关详细信息，请参见初始化和终止例程。

.interp

程序的解释程序的路径名。 有关详细信息，请参见程序的解释程序。

.lbss

特定于 x64 的 未初始化数据。此数据与 `.bss` 类似，但用于大小超过 2 GB 的节。

.ldata 和 .ldata1

特定于 x64 的已初始化数据。此数据与 `.data` 类似，但用于大小超过 2 GB 的节。

.lrodata 和 .lrodata1

特定于 x64 的只读数据。此数据与 `.rodata` 类似，但用于大小超过 2 GB 的节。

.note

注释节中说明了该格式的信息。

.plt

过程链接表。 有关详细信息，请参见过程链接表（特定于处理器）。

.preinit_array

函数指针数组，用于构成包含此节的可执行文件或共享库的单个 预初始化 数组。 有关详细信息，请参见初始化和终止例程。

.rela

不适用于特定节的重定位。此节的用途之一是用于寄存器重定位。 有关详细信息，请参见寄存器符号。

.rel ***name*** 、 **.rela** ***name***

重定位信息，如重定位节中所述。如果文件具有包括重定位的可装入段，则此节的属性将包括 `SHF_ALLOC` 位。否则，该位会处于禁用状态。通常，*name* 由应用重定位的节提供。因此， `.text` 的重定位节的名称通常为 `.rel.text` 或 `.rela.text` 。

.rodata 、 **.rodata1**

通常构成进程映像中的非可写段的只读数据。 有关详细信息，请参见程序头。

.shstrtab

节名。

.strtab

字符串，通常是表示与符号表各项关联的名称的字符串。如果文件具有包括符号字符串表的可装入段，则此节的属性将包括 SHF_ALLOC 位。否则，该位会处于禁用状态。

.symtab

符号表，如符号表节中所述。如果文件具有包括符号表的可装入段，则此节的属性将包括 SHF_ALLOC 位。否则，该位会处于禁用状态。

.symtab_shndx

此节包含特殊符号表的节索引数组，如 .symtab 所述。如果关联的符号表节包括 SHF_ALLOC 位，则此节的属性也将包括该位。否则，该位会处于禁用状态。

.tbss

此节包含未初始化的线程局部数据，这些数据构成程序的内存映像。根据定义，针对每个新执行流对数据进行实例化时，系统会将数据初始化为零。如节类型 SHT_NOBITS 所指明的那样，此节不会占用任何文件空间。 有关详细信息，请参见第 8 章，线程局部存储。

.tdata 、 .tdata1

这些节包含已初始化的线程局部数据，这些数据构成程序的内存映像。对于每个新执行流，系统会对数据内容的副本进行实例化。 有关详细信息，请参见第 8 章，线程局部存储。

.text

程序的**文本**或可执行指令。

.SUNW_bss

共享库的部分初始化数据，这些数据构成程序的内存映像。数据会在运行时进行初始化。如节类型 SHT_NOBITS 所指明的那样，此节不会占用任何文件空间。

.SUNW_heap

从 dldump(3C) 中创建的动态可执行文件的**堆**。

.SUNW_cap

硬件和软件的功能要求。 有关详细信息，请参见硬件和软件功能节。

.SUNW_move

部分初始化数据的附加信息。 有关详细信息，请参见移动节。

.SUNW_reloc

重定位信息，如重定位节中所述。此节是多个重定位节的串联，用于为引用各个重定位记录提供更好的临近性。由于仅有重定位记录的偏移有意义，因此节的 sh_info 值为零。

.SUNW_syminfo

其他符号表信息。 有关详细信息，请参见Syminfo 表节。

.SUNW_version

版本控制信息。 有关详细信息，请参见版本控制节。

具有点 (.) 前缀的节名为系统而保留，但如果这些节的现有含义符合要求，则应用程序也可以使用这些节。应用程序可以使用不带前缀的名称，以避免与系统节产生冲突。使用目标文件格式，可以定义非保留的节。一个目标文件可以包含多个同名的节。

保留用于处理器体系结构的节名通过在节名前加上体系结构名称的缩写而构成。该名称应来自用于 e_machine 的体系结构名称。例如， .Foo.psect 是根据 FOO 体系结构定义的 psect 节。

现有扩展使用其历史名称。

COMDAT 节

COMDAT 节由其节名 (sh_name) 唯一标识。如果链接编辑器遇到节名相同的 SHT_SUNW_COMDAT 类型的多个节，则将保留第一个节，并废弃其余的节。任何应用于已废弃的 SHT_SUNW_COMDAT 节的重定位都会被忽略。在已废弃的节中定义的任何符号都会被删除。

此外，使用 **-xF** 选项调用编译器时，链接编辑器还支持用于对节重新排序的节命名约定。如果将函数放入名为 . *sectname* % *funcname* 的 SHT_SUNW_COMDAT 节中，则最后保留的几个 SHT_SUNW_COMDAT 节都将并入名为 . *sectname* 的节中。此方法可用于将 SHT_SUNW_COMDAT 节放入 .text 、 .data 或其他任何节等最终目标位置。

组节

一些节会出现在相关的组中。例如，内置函数的外部定义除了要求包含可执行指令的节外，还会要求其他信息。此附加信息可以是包含引用的字面值的只读数据节、一个或多个调试信息节或其他信息节。

组节之间可以存在内部引用。但是，如果删除了其中某节，或者将其中某节替换为另一个目标文件的副本，则这些引用将没有意义。因此，应将这些组作为一个单位包括在链接目标文件中或从中忽略。

SHT_GROUP 类型的节可定义这样分组的一组节：其中一个所包含目标文件的符号表中的符号名称将为节组提供签名。 SHT_GROUP 节的节头会指定标识符号项。 sh_link 成员包含符号表节的节头索引，其中会包含该项。 sh_info 成员包含标识项的符号表索引。节头的 sh_flags 成员值为零。节名 (sh_name) 未指定。

SHT_GROUP 节的节数据是 Elf32_Word 项的数组。第一项是一个标志字。其余项是一系列节头索引。

当前定义了以下标志：

表 7–11 ELF 组节标志

名称	值
GRP_COMDAT	0x1

GRP_COMDAT

GRP_COMDAT 是一个 COMDAT 组。该组可以与另一个目标文件中的 COMDAT 组重复，其中，重复的定义是具有相同的组签名。在这类情况下，链接编辑器将仅保留其中一个重复组。其余组的成员会被废弃。

SHT_GROUP 节中的节头索引可标识构成该组的节。这些节必须在其 sh_flags 节头成员中设置 SHF_GROUP 标志。如果链接编辑器决定删除节组，则它将删除组的所有成员。

为了便于删除组，并且不保留未使用的引用，同时仅对符号表进行最少的处理，请遵循以下规则：

- 必须使用符号表各项中包含的 STB_GLOBAL 或 STB_WEAK 绑定和节索引 SHN_UNDEF ，才能从组外的节中引用包含该组的节。包含引用的目标文件中定义的相同符号必须具有与独立于该引用的符号表项。组外的各节不能引用对组节中包含的地址具有 STB_LOCAL 绑定的符号，包括 STT_SECTION 类型的符号。
- 不允许从组外对包含该组的节进行非符号引用。例如，不能在 sh_link 或 sh_info 成员中使用组成员的节头索引。
- 如果废弃了某个组的成员，则可以删除所定义的与该组某一节相关的符号表项。如果此符号表项包含在不属于该组的符号表节中，则会进行此删除。

硬件和软件功能节

SHT_SUNW_cap 节标识目标文件的硬件和软件功能。此节包含以下结构的数组。 请参见 sys/link.h 。

```
typedef struct {
        Elf32_Word      c_tag;
        union {
                Elf32_Word      c_val;
                Elf32_Addr      c_ptr;
        } c_un;
} Elf32_Cap;


typedef struct {
        Elf64_Xword     c_tag;
        union {
                Elf64_Xword     c_val;
                Elf64_Addr      c_ptr;
        } c_un;
} Elf64_Cap;
```

对于此类型的每一个目标文件， c_tag 会控制 c_un 的解释。

c_val

这些目标文件表示具有各种解释的整数值。

c_ptr

这些目标文件表示程序虚拟地址。

存在以下功能标记。

表 7–12 ELF 功能数组标记

名称	值	c_un
CA_SUNW_NULL	0	忽略
CA_SUNW_HW_1	1	c_val
CA_SUNW_SF_1	2	c_val

CA_SUNW_NULL

标记功能数组的结尾。

CA_SUNW_HW_1

表示硬件功能值。 c_val 元素包含用于表示关联硬件功能的值。在 SPARC 平台上，硬件功能在 sys/auxv_SPARC.h 中定义。在 x86 平台上，硬件功能在 sys/auxv_386.h 中定义。

CA_SUNW_SF_1

表示软件功能值。 c_val 元素包含用于表示 sys/elf.h 中定义的关联软件功能的值。

可重定位目标文件可以包含功能节。链接编辑器会将多个可重定位输入目标文件中的所有功能节合并到一个单独的功能节中。使用链接编辑器，还可在生成目标文件时定义功能。 请参见标识硬件和软件功能。

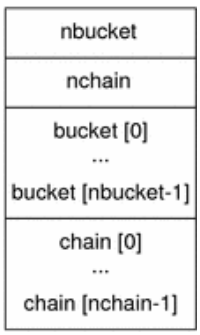
包含功能节（其中包含硬件功能信息）的动态库具有与该节关联的 PT_SUNWCAP 程序头。使用此程序头，运行时链接程序可以针对可供进程使用的硬件功能来验证目标文件。

利用不同硬件功能的动态库可以提供使用过滤器的灵活运行时环境。 请参见特定于硬件功能的共享库。

散列表节

散列表由用于符号表访问的 Elf32_Word 或 Elf64_Word 目标文件组成。 SHT_HASH 节提供了此散列表。与散列关联的符号表在散列表节头的 sh_link 项中指定。下图中使用了标签来帮助说明散列表的结构，但这些标签不属于规范的一部分。

图 7–4 符号散列表



bucket 数组包含 nbucket 项， chain 数组包含 nchain 项。索引从 0 开始。 bucket 和 chain 都包含符号表索引。链表的各项与符号表对应。符号表的项数应等于 nchain ，因此符号表索引也可选择链表的各项。

接受符号名称的散列函数会返回一个值，用于计算 bucket 索引。因此，如果散列函数为某个名称返回值 *x* ，则 bucket [*x*% nbucket] 将会计算出索引 *y*。此索引为符号表和链表的索引。如果符号表项不是需要的名称，则 chain [*y*] 将使用相同的散列值计算出符号表的下一项。

在所选符号表项具有需要的名称或者 chain 项包含值 STN_UNDEF 之前，可以遵循 chain 链接。

散列函数如下所示：

```
unsigned long

elf_Hash(const unsigned char *name)

{

    unsigned long h = 0, g;


            while (*name)

            {

                     h = (h << 4) + *name++;

                     if (g = h & 0xf0000000)

                              h ^= g >> 24;

                                   h &= ~g;

            }

            return h;

}
```

移动节

通常在 ELF 文件中，已初始化的数据变量会保留在目标文件中。如果数据变量很大，并且仅包含少量的已初始化（非零）元素，则整个变量仍会保留在目标文件中。

包含大的部分初始化数据变量的目标文件（如 FORTRAN COMMON 块）可能会产生较大的磁盘空间开销。 SHT_SUNW_move 节提供了一种压缩这些数据变量的机制。此压缩机制可减小关联目标文件的磁盘空间大小。

SHT_SUNW_move 节包含多个类型为 ELF32_Move 或 Elf64_Move 的项。使用这些项，可以将数据变量定义为暂定项目（.bss）。这些项目在目标文件中不占用任何空间，但在运行时会构成目标文件的内存映像。移动记录可确定如何初始化内存映像的数据，从而构造完整的数据变量。

ELF32_Move 和 Elf64_Move 项的定义如下：

```
typedef struct {

        Elf32_Lword       m_value;

        Elf32_Word        m_info;

        Elf32_Word        m_poffset;

        Elf32_Half        m_repeat;

        Elf32_Half        m_stride;

} Elf32_Move;


#define ELF32_M_SYM(info)       ((info)>>8)

#define ELF32_M_SIZE(info)      ((unsigned char)(info))

#define ELF32_M_INFO(sym, size) (((sym)<<8)+(unsigned char)(size))


typedef struct {

        Elf64_Lword       m_value;

        Elf64_Xword       m_info;

        Elf64_Xword       m_poffset;

        Elf64_Half        m_repeat;

        Elf64_Half        m_stride;

} Elf64_Move;
```

```
#define ELF64_M_SYM(info)       ((info)>>8)

#define ELF64_M_SIZE(info)      ((unsigned char)(info))

#define ELF64_M_INFO(sym, size) (((sym)<<8)+(unsigned char)(size))
```

这些结构的元素如下：

m_value

初始化值，即移到内存映像中的值。

m_info

符号表索引（与应用初始化相关）以及初始化的偏移的大小（以字节为单位）。成员的低 8 位定义大小，该大小可以是 1、2、4 或 8。高位字节定义符号索引。

m_poffset

与应用初始化的关联符号相关的偏移。

m_repeat

重复计数。

m_stride

幅度计数。该值表示在执行重复初始化时应跳过的单位数。单位是由 `m_info` 定义的初始化目标文件的大小。 `m_stride` 值为零表示连续对单位执行初始化。

以下数据定义以前在目标文件中会占用 `0x8000` 个字节：

```
typedef struct {
        int     one;
        char    two;
} Data


Data move[0x1000] = {
        {0, 0},       {1, '1'},     {0, 0},
        {0xf, 'F'},   {0xf, 'F'},   {0, 0},
        {0xe, 'E'},   {0, 0},       {0xe, 'E'}
};
```

`SHT_SUNW_move` 节可用于说明此数据。数据项可以在 `.bss` 节中定义并初始化为相应的移动项。

```
$  elfdump -s data | fgrep move
      [17]  0x00020868 0x00008000  OBJT GLOB 0   .bss       move
$  elfdump -m data


Move Section: .SUNW_move
        offset  ndx     size    repeat  stride  value   with respect to
        0x8     0x17    4       1       0       0x1     move
        0xc     0x17    1       1       0       0x31    move
        0x18    0x17    4       2       2       0xf     move
        0x1c    0x17    1       2       8       0x46    move
        0x28    0x17    4       2       4       0xe     move
        0x2c    0x17    1       2       16      0x45    move
```

可重定位目标文件提供的移动节可串联并在链接编辑器所创建的目标文件中输出。但是，在以下条件下链接编辑器将处理移动项，并将其内容扩展到旧的数据项中：

- 输出文件为静态可执行文件。
- 移动项的大小大于移动数据会扩展到的符号的大小。
- `-z nopartial` 选项有效。

注释节

供应商或系统工程师可能需要使用特殊信息标记目标文件，以便其他程序可根据此信息检查一致性或兼容性。为此，可使用 `SHT_NOTE` 类型的节和 `PT_NOTE` 类型的程序头元素。

节和程序头元素中的注释信息包含任意数量的项，如下图所示。对于 64 位目标文件和 32 位目标文件，每一项都是一个目标处理器格式的 4 字节字的数组。图 7–6 中所示的标签用于帮助说明注释信息的结构，但不属于规范的一部分。

图 7–5 注释信息

`namesz` 和 `name`

名称中的前 `namesz` 个字节，表示项的属主或创建者的字符（以空字符结尾）。不存在用于避免名称冲突的正式机制。根据约定，供应商使用其各自的名称（如 "XYZ Computer Company"）作为标识符。如果不存在 `name` ，则 `namesz` 值为零。如有必要，可使用填充确保描述符 4 字节对齐。 `namesz` 中不包括这种填充方式。

`descsz` 和 `desc`

`desc` 中的前 `descsz` 个字节包含注释描述符。如果不存在描述符，则 `descsz` 值为零。如有必要，可使用填充确保下一个注释项 4 字节对齐。 `descsz` 中不包括这种填充方式。

`type`

提供对描述符的解释。每个创建者可控制其各自的类型。单个 `type` 值可以存在多种解释。程序必须同时识别名称和 `type` 才能理解描述符。类型当前必须为非负数。

下图中所示的注释段包含两项。

图 7–6 注释段示例

注 –

系统会保留没有名称（`namesz == 0`）以及名称长度为零（`name[0] == '\0'`）的注释信息，但当前不定义任何类型。其他所有名称必须至少有一个非空字符。

重定位节

重定位是连接符号引用与符号定义的过程。例如，程序调用函数时，关联的调用指令必须在执行时将控制权转移到正确的目标地址。可重定位文件必须包含说明如何修改其节内容的信息。通过此信息，可执行文件和共享库文件可包含进程的程序映像的正确信息。可重定位项即是这些数据。

可重定位项可具有以下结构。 请参见 `sys/elf.h` 。

```
typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
} Elf32_Rel;

typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
        Elf32_Sword     r_addend;
} Elf32_Rela;

typedef struct {
        Elf64_Addr      r_offset;
        Elf64_Xword     r_info;
} Elf64_Rel;

typedef struct {
        Elf64_Addr      r_offset;
        Elf64_Xword     r_info;
        Elf64_Sxword    r_addend;
} Elf64_Rela;
```

`r_offset`

此成员指定应用可重定位操作的位置。不同的目标文件对于此成员的解释会稍有不同。

对于可重定位文件，该值表示节偏移。重定位节可说明如何修改文件中的其他节。重定位偏移会在第二节中指定一个存储单元。

对于可执行文件或共享库，该值表示受重定位影响的存储单元的虚拟地址。此信息使重定位项对于运行时链接程序更为有用。

虽然为了使相关程序可以更有效地访问，不同目标文件的成员的解释会发生变化，但重定位类型的含义保持相同。

`r_info`

此成员指定必须对其进行重定位的符号表索引以及要应用的重定位类型。例如，调用指令的重定位项包含所调用的函数的符号表索引。如果索引是未定义的符号索引 `STN_UNDEF` ，则重定位将使用零作为符号值。

重定位类型特定于处理器。重定位项的重定位类型或符号表索引是将 `ELF32_R_TYPE` 或 `ELF32_R_SYM` 分别应用于项的 `r_info` 成员所得的结果：

```
#define ELF32_R_SYM(info)             ((info)>>8)
#define ELF32_R_TYPE(info)            ((unsigned char)(info))
#define ELF32_R_INFO(sym, type)       (((sym)<<8)+(unsigned char)(type))
```

```
#define ELF64_R_SYM(info)             ((info)>>32)

#define ELF64_R_TYPE(info)            ((Elf64_Word)(info))

#define ELF64_R_INFO(sym, type)       (((Elf64_Xword)(sym)<<32)+ \

                                        (Elf64_Xword)(type))
```

对于 `Elf64_Rel` 和 `Elf64_Rela` 结构，`r_info` 字段可进一步细分为 8 位类型标识符和 24位类型相关数据字段：

```
#define ELF64_R_TYPE_DATA(info)       (((Elf64_Xword)(info)<<32)>>40)

#define ELF64_R_TYPE_ID(info)         (((Elf64_Xword)(info)<<56)>>56)

#define ELF64_R_TYPE_INFO(data, type) (((Elf64_Xword)(data)<<8)+ \

                                        (Elf64_Xword)(type))
```

`r_addend`

此成员指定常量加数，用于计算将存储在可重定位字段中的值。

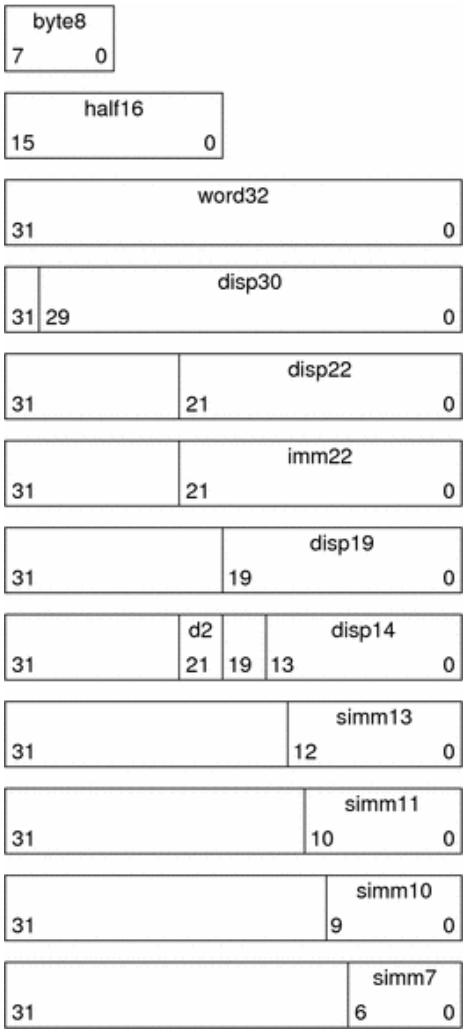
`Rela` 项包含显式加数。 `Rel` 类型的项会在要修改的位置中存储一个隐式加数。32 位 SPARC 仅使用 `Elf32_Rela` 重定位项。 64 位 SPARC 和 64 位 x86 仅使用 `Elf64_Rela` 重定位项。因此，`r_addend` 成员可用作重定位加数。x86 仅使用 `Elf32_Rel` 重定位项。要重定位的字段包含该加数。在所有情况下，加数和计算所得的结果使用相同的字节顺序。

重定位节可以引用其他两个节：符号表（由 `sh_link` 节头项标识）和要修改的节（由 `sh_info` 节头项标识）。节中指定了这些关系。如果可重定位目标文件中存在重定位节，则需要 `sh_info` 项，但对于可执行文件和共享库，该项是可选的。重定位偏移满足执行重定位的要求。

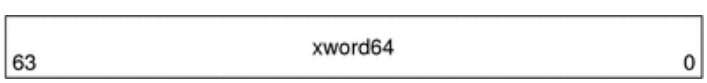
重定位类型（特定于处理器）

重定位项说明如何修改下图中的指令和数据字段。位数显示在框的下角。

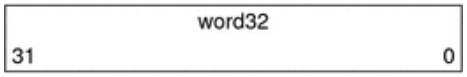
在 SPARC 平台上，重定位项应用于字节（`byte8`）、半字（`half16`）或字。



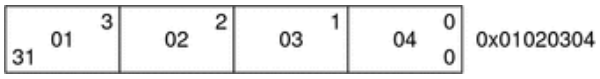
在 64 位 SPARC 和 x64 上，重定位项还会应用于扩展字（`xword64`）：



在 x86 上，重定位项应用于字（ word32 ）：



word32 可指定一个占用 4 个字节的 32 位字段，此字段以任意字节对齐。这些值使用与 x86 体系结构中的其他字值相同的字节顺序：



在所有情况下， r_offset 值都会指定受影响存储单元的第一个字节的偏移或虚拟地址。重定位类型可指定要更改的位以及计算这些位的值的方法。

针对以下重定位类型进行的计算假定，操作会将可重定位文件转换为可执行文件或共享库文件。在概念上，链接编辑器会将一个或多个可重定位文件合并以形成输出。链接编辑器首先确定如何合并和定位输入文件。然后，链接编辑器会更新符号值并执行重定位。应用于可执行文件或共享库文件的重定位类似，并会取得相同的结果。本节的表中的说明使用以下表示法：

A

用于计算可重定位字段的值的加数。

B

执行过程中将共享库装入内存的基本地址。通常，生成的共享库文件的基本虚拟地址为 0 。但是，共享库的执行地址不相同。 请参见程序头。

G

执行过程中，重定位项的符号地址所在的全局偏移表中的偏移。 请参见全局偏移表（特定于处理器）。

GOT

全局偏移表的地址。 请参见全局偏移表（特定于处理器）。

L

符号的过程链接表项的节偏移或地址。 请参见过程链接表（特定于处理器）。

P

使用 r_offset 计算出的重定位的存储单元的节偏移或地址。

S

索引位于重定位项中的符号的值。

SPARC: 重定位类型

下表中的字段名称可确定重定位类型是否会检查 overflow 。计算出的重定位值可以大于预期的字段，并且重定位类型可以验证（ V ）值是否适合结果或将结果截断（ T ）。例如， V-simm13 表示计算出的值不能包含 simm13 字段外有意义的非零位。

表 7–13 SPARC: ELF 重定位类型

名称	值	字段	计算
R_SPARC_NONE	0	无	无
R_SPARC_8	1	V-byte8	S + A
R_SPARC_16	2	V-half16	S + A
R_SPARC_32	3	V-word32	S + A
R_SPARC_DISP8	4	V-byte8	S + A - P

名称	值	字段	计算
R_SPARC_DISP16	5	V-half16	S + A - P
R_SPARC_DISP32	6	V-disp32	S + A - P
R_SPARC_WDISP30	7	V-disp30	(S + A - P) >> 2
R_SPARC_WDISP22	8	V-disp22	(S + A - P) >> 2
R_SPARC_HI22	9	T-imm22	(S + A) >> 10
R_SPARC_22	10	V-imm22	S + A
R_SPARC_13	11	V-simm13	S + A
R_SPARC_LO10	12	T-simm13	(S + A) & 0x3ff
R_SPARC_GOT10	13	T-simm13	G & 0x3ff
R_SPARC_GOT13	14	V-simm13	G
R_SPARC_GOT22	15	T-simm22	G >> 10
R_SPARC_PC10	16	T-simm13	(S + A - P) & 0x3ff
R_SPARC_PC22	17	V-disp22	(S + A - P) >> 10
R_SPARC_WPLT30	18	V-disp30	(L + A - P) >> 2
R_SPARC_COPY	19	无	请参阅此表后面的说明。
R_SPARC_GLOB_DAT	20	V-word32	S + A
R_SPARC_JMP_SLOT	21	无	请参阅此表后面的说明。
R_SPARC_RELATIVE	22	V-word32	B + A

名称	值	字段	计算
R_SPARC_UA32	23	V-word32	S + A
R_SPARC_PLT32	24	V-word32	L + A
R_SPARC_HIPLT22	25	T-imm22	(L + A) >> 10
R_SPARC_LOPLT10	26	T-simm13	(L + A) & 0x3ff
R_SPARC_PCPLT32	27	V-word32	L + A - P
R_SPARC_PCPLT22	28	V-disp22	(L + A - P) >> 10
R_SPARC_PCPLT10	29	V-simm13	(L + A - P) & 0x3ff
R_SPARC_10	30	V-simm10	S + A
R_SPARC_11	31	V-simm11	S + A
R_SPARC_HH22	34	V-imm22	(S + A) >> 42
R_SPARC_HM10	35	T-simm13	((S + A) >> 32) & 0x3ff
R_SPARC_LM22	36	T-imm22	(S + A) >> 10
R_SPARC_PC_HH22	37	V-imm22	(S + A - P) >> 42
R_SPARC_PC_HM10	38	T-simm13	((S + A - P) >> 32) & 0x3ff
R_SPARC_PC_LM22	39	T-imm22	(S + A - P) >> 10
R_SPARC_WDISP16	40	V-d2/disp14	(S + A - P) >> 2
R_SPARC_WDISP19	41	V-disp19	(S + A - P) >> 2
R_SPARC_7	43	V-imm7	S + A

名称	值	字段	计算
R_SPARC_5	44	V-imm5	S + A
R_SPARC_6	45	V-imm6	S + A
R_SPARC_HIX22	48	V-imm22	((S + A) ^ 0xffffffffffffffff) >> 10
R_SPARC_LOX10	49	T-simm13	((S + A) & 0x3ff) | 0x1c00
R_SPARC_H44	50	V-imm22	(S + A) >> 22
R_SPARC_M44	51	T-imm10	((S + A) >> 12) & 0x3ff
R_SPARC_L44	52	T-imm13	(S + A) & 0xfff
R_SPARC_REGISTER	53	V-word32	S + A
R_SPARC_UA16	55	V-half16	S + A
R_SPARC_GOTDATA_HIX22	80	T-imm22	((S + A - GOT) >> 10) ^ ((S + A - GOT) >> 31)
R_SPARC_GOTDATA_LOX22	81	T-imm13	((S + A - GOT) & 0x3ff) | (((S + A - GOT) >> 31) & 0x1c00)
R_SPARC_GOTDATA_OP_HIX22	82	T-imm22	(G >> 10) ^ (G >> 31)
R_SPARC_GOTDATA_OP_LOX22	83	T-imm13	(G & 0x3ff) | ((G >> 31) & 0x1c00)
R_SPARC_GOTDATA_OP	84	Word32	请参阅此表后面的说明。

注 –

其他重定位类型可用于线程局部存储引用。这些重定位类型将在第 8 章，线程局部存储中介绍。

一些重定位类型的语义不只是简单的计算：

R_SPARC_GOT10

与 R_SPARC_LO10 类似，不同的是此重定位指向符号的 GOT 项的地址。此外， R_SPARC_GOT10 还指示链接编辑器创建全局偏移表。

R_SPARC_GOT13

与 R_SPARC_13 类似，不同的是此重定位指向符号的 GOT 项的地址。此外， R_SPARC_GOT13 还指示链接编辑器创建全局偏移表。

R_SPARC_GOT22

与 R_SPARC_22 类似，不同的是此重定位指向符号的 GOT 项的地址。此外， R_SPARC_GOT22 还指示链接编辑器创建全局偏移表。

R_SPARC_WPLT30

与 R_SPARC_WDISP30 类似，不同的是此重定位指向符号的过程链接表项的地址。此外， R_SPARC_WPLT30 还指示链接编辑器创建过程链接表。

R_SPARC_COPY

由链接编辑器为动态可执行文件创建，用于保留只读文本段。此重定位偏移成员指向可写段中的位置。符号表索引指定应在当前目标文件和共享库中同时存在的符号。执行过程中，运行时链接程序将与共享库的符号关联的数据复制到偏移所指定的位置。 请参见复制重定位。

R_SPARC_GLOB_DAT

与 R_SPARC_32 类似，不同的是此重定位会将 GOT 项设置为所指定符号的地址。使用特殊重定位类型，可以确定符号和 GOT 项之间的对应关系。

R_SPARC_JMP_SLOT

由链接编辑器为动态库创建，用于提供延迟绑定。此重定位偏移成员可指定过程链接表项的位置。运行时链接程序会修改过程链接表项，以将控制权转移到指定的符号地址。

R_SPARC_RELATIVE

由链接编辑器为动态库创建。此重定位偏移成员可指定共享库中包含表示相对地址的值的位置。运行时链接程序通过将装入共享库的虚拟地址与相对地址相加，计算对应的虚拟地址。此类型的重定位项必须为符号表索引指定零值。

R_SPARC_UA32

与 R_SPARC_32 类似，不同的是此重定位指向未对齐的字。必须将要重定位的字作为任意对齐的四个独立字节进行处理，而不是作为根据体系结构要求对齐的字进行处理。

R_SPARC_LM22

与 R_SPARC_HI22 类似，不同的是此重定位会进行截断而不是验证。

R_SPARC_PC_LM22

与 R_SPARC_PC22 类似，不同的是此重定位会进行截断而不是验证。

R_SPARC_HIX22

与 R_SPARC_LOX10 一起用于可执行文件，这些可执行文件在 64 位地址空间中的上限为 4 GB。与 R_SPARC_HI22 类似，但会提供链接值的补码。

R_SPARC_LOX10

与 R_SPARC_HIX22 一起使用。与 R_SPARC_LO10 类似，但始终设置链接值的位 10 到 12。

R_SPARC_L44

与 R_SPARC_H44 和 R_SPARC_M44 重定位类型一起使用，以生成 44 位的绝对寻址模型。

R_SPARC_REGISTER

用于初始化寄存器符号。此重定位偏移成员包含要初始化的寄存器编号。对于此寄存器必须存在对应的寄存器符号。该符号必须为 SHN_ABS 类型。

R_SPARC_GOTDATA_OP_HIX22 、 R_SPARC_GOTDATA_OP_LOX22 和 R_SPARC_GOTDATA_OP

这些重定位类型用于代码转换。

64 位 SPARC: 重定位类型

重定位计算中使用的以下表示法是特定于 64 位 SPARC 的。

O

用于计算重定位字段的值的辅助加数。此加数通过应用 ELF64_R_TYPE_DATA 宏从 r_info 字段中提取。

下表中列出的重定位类型是扩展或修改针对 32 位 SPARC 定义的重定位类型所得的。 请参见SPARC: 重定位类型。

表 7–14 64 位 SPARC: ELF 重定位类型

名称	值	字段	计算

R_SPARC_HI22	9	V-imm22	(S + A) >> 10
R_SPARC_GLOB_DAT	20	V-xword64	S + A
R_SPARC_RELATIVE	22	V-xword64	B + A
R_SPARC_64	32	V-xword64	S + A
R_SPARC_OLO10	33	V-simm13	((S + A) & 0x3ff) + O
R_SPARC_DISP64	46	V-xword64	S + A - P
R_SPARC_PLT64	47	V-xword64	L + A
R_SPARC_REGISTER	53	V-xword64	S + A
R_SPARC_UA64	54	V-xword64	S + A
R_SPARC_H34	85	V-imm22	(S + A) >> 12

以下重定位类型的语义不只是简单的计算：

R_SPARC_OLO10

与 R_SPARC_LO10 类似，不同的是会添加额外的偏移，以充分利用 13 位带符号的直接字段。

32 位 x86: 重定位类型

下表中列出的重定位类型是针对 32 位 x86 定义的。

表 7–15 32 位 x86: ELF 重定位类型

名称	值	字段	计算
R_386_NONE	0	无	无
R_386_32	1	word32	S + A
R_386_PC32	2	word32	S + A - P
R_386_GOT32	3	word32	G + A
R_386_PLT32	4	word32	L + A - P

名称	值	字段	计算
R_386_COPY	5	无	请参阅此表后面的说明。
R_386_GLOB_DAT	6	word32	S
R_386_JMP_SLOT	7	word32	S
R_386_RELATIVE	8	word32	B + A
R_386_GOTOFF	9	word32	S + A - GOT
R_386_GOTPC	10	word32	GOT + A - P
R_386_32PLT	11	word32	L + A
R_386_16	20	word16	L + A
R_386_PC16	21	word16	L + A - P
R_386_8	22	word8	L + A
R_386_PC8	23	word8	L + A - P

注 –

其他重定位类型可用于线程局部存储引用。这些重定位类型将在第 8 章，线程局部存储中介绍。

一些重定位类型的语义不只是简单的计算：

R_386_GOT32

计算 GOT 的基本地址与符号的 GOT 项之间的距离。此重定位还指示链接编辑器创建全局偏移表。

R_386_PLT32

计算符号的过程链接表项的地址，并指示链接编辑器创建一个过程链接表。

R_386_COPY

由链接编辑器为动态可执行文件创建，用于保留只读文本段。此重定位偏移成员指向可写段中的位置。符号表索引指定应在当前目标文件和共享库中同时存在的符号。执行过程中，运行时链接程序将与共享库的符号关联的数据复制到偏移所指定的位置。 请参见复制重定位。

R_386_GLOB_DAT

用于将 GOT 项设置为所指定符号的地址。使用特殊重定位类型，可以确定符号和 GOT 项之间的对应关系。

R_386_JMP_SLOT

由链接编辑器为动态库创建，用于提供延迟绑定。此重定位偏移成员可指定过程链接表项的位置。运行时链接程序会修改过程链接表项，以将控制权转移到指定的符号地址。

R_386_RELATIVE

由链接编辑器为动态库创建。此重定位偏移成员可指定共享库中包含表示相对地址的值的位置。运行时链接程序通过将装入共享库的虚拟地址与相对地址相加，计算对应的虚拟地址。此类型的重定位项必须为符号表索引指定值零。

R_386_GOTOFF

计算符号的值与 GOT 的地址之间的差值。此重定位还指示链接编辑器创建全局偏移表。

R_386_GOTPC

与 R_386_PC32 类似，不同的是它在其计算中会使用 GOT 的地址。此重定位中引用的符号通常是 _GLOBAL_OFFSET_TABLE_ ，该符号还指示链接编辑器创建全局偏移表。

x64: 重定位类型

下表中列出的重定位是针对 x64 定义的。

表 7–16 x64: ELF 重定位类型

名称	值	字段	计算
R_AMD64_NONE	0	无	无
R_AMD64_64	1	word64	S + A
R_AMD64_PC32	2	word32	S + A - P
R_AMD64_GOT32	3	word32	G + A
R_AMD64_PLT32	4	word32	L + A - P
R_AMD64_COPY	5	无	请参阅此表后面的说明。
R_AMD64_GLOB_DAT	6	word64	S
R_AMD64_JUMP_SLOT	7	word64	S
R_AMD64_RELATIVE	8	word64	B + A
R_AMD64_GOTPCREL	9	word32	G + GOT + A - P
R_AMD64_32	10	word32	S + A
R_AMD64_32S	11	word32	S + A
R_AMD64_16	12	word16	S + A
R_AMD64_PC16	13	word16	S + A - P

名称	值	字段	计算
R_AMD64_8	14	word8	S + A
R_AMD64_PC8	15	word8	S + A - P
R_AMD64_PC64	24	word64	S + A - P
R_AMD64_GOTOFF64	25	word64	S + A - GOT
R_AMD64_GOTPC32	26	work32	GOT + A + P

注 –

其他重定位类型可用于线程局部存储引用。这些重定位类型将在第 8 章，线程局部存储中介绍。

大多数重定位类型的特殊语义与用于 x86 的语义相同。一些重定位类型的语义不只是简单的计算：

R_AMD64_GOTPCREL

此重定位类型具有与 R_AMD64_GOT32 或等效 R_386_GOTPC 重定位类型不同的语义。x64 体系结构提供了相对于指令指针的寻址模式。因此，可以使用单个指令从 GOT 装入地址。

针对 R_AMD64_GOTPCREL 重定位类型进行的计算提供了 GOT 中指定了符号地址的位置与应用重定位的位置之间的差值。

R_AMD64_32

计算出的值会截断为 32 位。链接编辑器可验证为重定位生成的值是否会使用零扩展为初始的 64 位值。

R_AMD64_32S

计算出的值会截断为 32 位。链接编辑器可验证为重定位生成的值是否会使用符号扩展为初始的 64 位值。

R_AMD64_8 、 R_AMD64_16 、 R_AMD64_PC16 和 R_AMD64_PC8

这些重定位类型不适用于 x64 ABI，在此列出是为了说明。 R_AMD64_8 重定位类型会将计算出的值截断为 8 位。 R_AMD64_16 重定位类型会将所计算的值截断为 16 位。

字符串表节

字符串表节包含以空字符结尾的字符序列，通常称为字符串。目标文件使用这些字符串表示符号和节的名称。可以将字符串作为字符串表节的索引进行引用。

第一个字节（索引零）包含空字符。同样，字符串表的最后一个字节也包含空字符，从而确保所有字符串都以空字符结尾。根据上下文，索引为零的字符串不会指定任何名称或指定空名称。

允许使用空字符串表节。节头的 sh_size 成员值为零。对于空字符串表，非零索引无效。

节头的 sh_name 成员包含节头字符串表的节索引。节头字符串表由 ELF 头的 e_shstrndx 成员指定。下图显示了具有 25 个字节的字符串表，并且其字符串与各种索引关联。

图 7–7 ELF 字符串表

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					

下表显示了上图所示的字符串表中的字符串。

表 7–17 ELF 字符串表索引

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

如示例所示，字符串表索引可以指向节中的任何字节。一个字符串可以出现多次。可以存在对子字符串的引用。一个字符串可以多次引用。另外，还允许使用未引用的字符串。

符号表节

目标文件的符号表包含定位和重定位程序的符号定义和符号引用所需的信息。符号表索引是此数组的下标。索引 `0` 指定表中的第一项并用作未定义的符号索引。 请参见表 7–21。

符号表项具有以下格式。 请参见 `sys/elf.h` 。

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;


typedef struct {
        Elf64_Word      st_name;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf64_Half      st_shndx;
        Elf64_Addr      st_value;
        Elf64_Xword     st_size;
} Elf64_Sym;
```

`st_name`

目标文件的符号字符串表的索引，其中包含符号名称的字符表示形式。如果该值为非零，则表示指定符号名称的字符串表索引。否则，符号表项没有名称。

`st_value`

关联符号的值。根据上下文，该值可以是绝对值或地址。 请参见符号值。

st_size

许多符号具有关联大小。例如，数据目标文件的大小是目标文件中包含的字节数。如果符号没有大小或大小未知，则此成员值为零。

st_info

符号的类型和绑定属性。表 7–18 中显示了值和含义的列表。以下代码说明了如何处理这些值。 请参见 sys/elf.h 。

```
#define ELF32_ST_BIND(info)          ((info) >> 4)
#define ELF32_ST_TYPE(info)          ((info) & 0xf)
#define ELF32_ST_INFO(bind, type)    (((bind)<<4)+((type)&0xf))

#define ELF64_ST_BIND(info)          ((info) >> 4)
#define ELF64_ST_TYPE(info)          ((info) & 0xf)
#define ELF64_ST_INFO(bind, type)    (((bind)<<4)+((type)&0xf))
```

st_other

符号的可见性。表 7–20 中显示了值和含义的列表。以下代码说明了如何处理 32 位目标文件和 64 位目标文件的值。其他位设置为零，并且未定义任何含义。

```
#define ELF32_ST_VISIBILITY(o)       ((o)&0x3)
#define ELF64_ST_VISIBILITY(o)       ((o)&0x3)
```

st_shndx

所定义的每一个符号表项都与某节有关。此成员包含相关节头表索引。部分节索引会表示特殊含义。 请参见表 7–4。

如果此成员包含 SHN_XINDEX ，则实际节头索引会过大而无法放入此字段中。实际值包含在 SHT_SYMTAB_SHNDX 类型的关联节中。

根据符号的 st_info 字段确定的符号绑定可确定链接可见性和行为。

表 7–18 ELF 符号绑定： ELF32_ST_BIND 和 ELF64_ST_BIND

名称	值
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOOS	10
STB_HIOS	12
STB_LOPROC	13
STB_HIPROC	15

STB_LOCAL

局部符号。这些符号在包含其定义的目标文件的外部不可见。名称相同的局部符号可存在于多个文件中而不会相互干扰。

STB_GLOBAL

全局符号。这些符号对于合并的所有目标文件都可见。一个文件的全局符号定义满足另一个文件对相同全局符号的未定义引用。

STB_WEAK

弱符号。这些符号与全局符号类似，但其定义具有较低的优先级。

STB_LOOS - STB_HIOS

此范围内包含的值保留用于特定于操作系统的语义。

STB_LOPROC - STB_HIPROC

此范围内包含的值保留用于特定于处理器的语义。

全局符号和弱符号在以下两个主要方面不同：

- 链接编辑器合并多个可重定位目标文件时，不允许多次定义相同名称的 STB_GLOBAL 符号。但是，如果存在已定义的全局符号，则出现相同名称的弱符号不会导致错误。链接编辑器会接受全局定义，并忽略弱定义。

  同样，如果存在通用符号，则出现相同名称的弱符号也不会导致错误。链接编辑器将使用通用定义，并忽略弱定义。通用符号具有包含 SHN_COMMON 的 st_shndx 字段。 请参见符号解析。

- 链接编辑器搜索归档库时，将会提取包含未定义全局符号或暂定全局符号的定义的归档成员。此成员的定义可以是全局符号或弱符号。

  缺省情况下，链接编辑器不会提取归档成员来解析未定义的弱符号。未解析的弱符号的值为零。使用 **-z weakextract** 可覆盖此缺省行为。使用此选项，弱引用可提取归档成员。

注 –

弱符号主要适用于系统软件。建议不要在应用程序中使用弱符号。

在每个符号表中，具有 STB_LOCAL 绑定的所有符号都优先于弱符号和全局符号。如节中所述，符号表节的 sh_info 节头成员包含第一个非局部符号的符号表索引。

根据符号的 st_info 字段确定的符号类型用于对关联实体进行一般分类。

表 7–19 ELF 符号类型： ELF32_ST_TYPE 和 ELF64_ST_TYPE

名称	值
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_COMMON	5
STT_TLS	6
STT_LOOS	10
STT_HIOS	12

名称	值
STT_LOPROC	13
STT_SPARC_REGISTER	13
STT_HIPROC	15

STT_NOTYPE

未指定符号类型。

STT_OBJECT

此符号与变量、数组等数据目标文件关联。

STT_FUNC

此符号与函数或其他可执行代码关联。

STT_SECTION

此符号与节关联。此类型的符号表各项主要用于重定位，并且通常具有 STB_LOCAL 绑定。

STT_FILE

通常，符号的名称会指定与目标文件关联的源文件的名称。文件符号具有 STB_LOCAL 绑定和节索引 SHN_ABS 。此符号（如果存在）位于文件的其他 STB_LOCAL 符号前面。

符号索引为 1 的 SHT_SYMTAB 是表示目标文件的 STT_FILE 符号。通常，此符号后跟文件的 STT_SECTION 符号。这些节符号又后跟已降为局部符号的任何全局符号。

STT_COMMON

此符号标记未初始化的通用块。此符号的处理与对 STT_OBJECT 的处理完全相同。

STT_TLS

此符号指定线程局部存储实体。定义后，此符号可为符号指明指定的偏移，而不是实际地址。

线程局部存储重定位只能引用 STT_TLS 类型的符号。从可分配节中引用 STT_TLS 类型的符号只能通过使用特殊线程局部存储重定位来实现。 有关详细信息，请参见第 8 章，线程局部存储。从非可分配节中引用 STT_TLS 类型的符号没有此限制。

STT_LOOS - STT_HIOS

此范围内包含的值保留用于特定于操作系统的语义。

STT_LOPROC - STT_HIPROC

此范围内包含的值保留用于特定于处理器的语义。

符号的可见性根据其 st_other 字段确定。此可见性可以在可重定位目标文件中指定。此可见性定义了在符号成为可执行文件或共享库的一部分后访问该符号的方式。

表 7–20 ELF 符号可见性

名称	值
STV_DEFAULT	0
STV_INTERNAL	1

名称	值
STV_HIDDEN	2
STV_PROTECTED	3

STV_DEFAULT

具有 STV_DEFAULT 属性的符号的可见性与符号的绑定类型指定的可见性相同。全局符号和弱符号在其定义组件（可执行文件或共享库）外部可见。局部符号处于隐藏状态。另外，还可以替换全局符号和弱符号。可以在另一个组件中通过定义相同的名称插入这些符号。

STV_PROTECTED

如果当前组件中定义的符号在其他组件中可见，但不能被替换，则该符号会处于受保护状态。定义组件中对这类符号的任何引用都必须解析为该组件中的定义。即使在另一个组件中存在按缺省规则插入的符号定义，也必须进行此解析。具有 STB_LOCAL 绑定的符号将没有 STV_PROTECTED 可见性。

STV_HIDDEN

如果当前组件中定义的符号的名称对于其他组件不可见，则该符号处于隐藏状态。必须对这类符号进行保护。此属性用于控制组件的外部接口。由这样的符号命名的目标文件仍可以在另一个组件中引用（如果将目标文件的地址传到外部）。

如果可执行文件或共享库中包括可重定位目标文件，则该目标文件中包含的隐藏符号将会删除或转换为使用 STB_LOCAL 绑定。

STV_INTERNAL

此可见性属性当前被保留。

在链接编辑过程中，可见性属性不会影响可执行文件或共享库中符号的解析。这样的解析由绑定类型控制。一旦链接编辑器选定了其解析，这些属性即会强加两种要求。两种要求都基于以下事实，即所链接的代码中的引用可能已优化，从而可利用这些属性。

- 所有非缺省可见性属性在应用于符号引用时都表示，在链接的目标文件中必须提供满足该引用的定义。如果在链接的目标文件中没有定义此类型的符号引用，则该引用必须具有 STB_WEAK 绑定。在此情况下，引用将解析为零。
- 如果任何名称的引用或定义是具有非缺省可见性属性的符号，则该可见性属性将传播给链接的目标文件中的解析符号。如果针对符号的不同实例指定不同的可见性属性，则最具约束的可见性属性将传播给链接的目标文件中的解析符号。这些属性按最低到最高约束进行排序，依次为 STV_PROTECTED 、 STV_HIDDEN 和 STV_INTERNAL 。

如果符号的值指向节中的特定位置，则符号的节索引成员 st_shndx 会包含节头表的索引。节在重定位过程中移动时，符号的值也会更改。符号的引用仍然指向程序中的相同位置。一些特殊节索引值会具有其他语义：

SHN_ABS

此符号具有不会由于重定位而发生更改的绝对值。

SHN_COMMON 和 SHN_AMD64_LCOMMON

此符号标记尚未分配的通用块。与节的 sh_addralign 成员类似，符号的值也会指定对齐约束。链接编辑器在值为 st_value 的倍数的地址位置为符号分配存储空间。符号的大小会指明所需的字节数。

SHN_UNDEF

此节表索引表示未定义符号。链接编辑器将此目标文件与用于定义所表示的符号的另一目标文件合并时，此文件中对该符号的引用将与该定义绑定。

如之前所述，索引 0 （ STN_UNDEF ）的符号表项会保留。此项具有下表中列出的值。

表 7–21 ELF 符号表项：索引 0

名称	值	说明
st_name	0	无名称
st_value	0	零值

名称	值	说明
st_size	0	无大小
st_info	0	无类型，本地绑定
st_other	0	
st_shndx	SHN_UNDEF	无节

符号值

不同目标文件类型的符号表的各项对于 st_value 成员的解释稍有不同。

- 在可重定位文件中， st_value 包含节索引为 SHN_COMMON 的符号的对齐约束。
- 在可重定位文件中， st_value 包含所定义符号的节偏移。 st_value 表示从 st_shndx 所标识的节的起始位置的偏移。
- 在可执行文件和共享库文件中， st_value 包含虚拟地址。为使这些文件的符号更适用于运行时链接程序，节偏移（文件解释）会替换为与节数无关的虚拟地址（内存解释）。

尽管符号表值对于不同的目标文件具有类似含义，但通过适当的程序可以有效地访问数据。

寄存器符号

SPARC 体系结构支持用于初始化全局寄存器的寄存器符号。下表中列出了寄存器符号的符号表项包含的各项。

表 7–22 SPARC: ELF 符号表项：寄存器符号

字段	含义
st_name	符号名称的字符串表的索引；若其值为 0 则代表临时寄存器。
st_value	寄存器编号。有关整数寄存器赋值的信息，请参见 ABI 手册。
st_size	未使用（0）。
st_info	绑定通常为 STB_GLOBAL ，类型必须是 STT_SPARC_REGISTER 。
st_other	未使用（0）。
st_shndx	如果该目标文件初始化此寄存器符号，则为 SHN_ABS ，否则为 SHN_UNDEF 。

下表中列出了为 SPARC 定义的寄存器值。

表 7–23 SPARC: ELF 寄存器编号

名称	值	含义

STO_SPARC_REGISTER_G2	0x2	%g2
STO_SPARC_REGISTER_G3	0x3	%g3

如果缺少特定全局寄存器的项，则意味着目标文件根本没有使用特定全局寄存器。

Syminfo 表节

syminfo 节包含多个类型为 Elf32_Syminfo 或 Elf64_Syminfo 的项。 .SUNW_syminfo 节中包含与关联符号表 (sh_link) 中的每一项对应的项。

如果目标文件中存在此节，则可通过采用关联符号表的符号索引，并使用该索引在此节中查找对应的 Elf32_Syminfo 项或 Elf64_Syminfo 项，从而找到其他符号信息。关联的符号表和 Syminfo 表的项数将始终相同。

索引 0 用于存储 Syminfo 表的当前版本，即 SYMINFO_CURRENT 。由于符号表项 0 始终保留用于 UNDEF 符号表项，因此该用法不会造成任何冲突。

Syminfo 项具有以下格式。 请参见 sys/link.h 。

```
typedef struct {
        Elf32_Half      si_boundto;
        Elf32_Half      si_flags;
} Elf32_Syminfo;


typedef struct {
        Elf64_Half      si_boundto;
        Elf64_Half      si_flags;
} Elf64_Syminfo;
```

si_boundto

.dynamic 节中某项的索引，由 sh_info 字段标识，该字段用于扩充 Syminfo 标志。例如， DT_NEEDED 项标识与 Syminfo 项关联的动态库。以下各项是 si_boundto 的保留值。

名称	值	含义
SYMINFO_BT_SELF	0xffff	符号与自身绑定。
SYMINFO_BT_PARENT	0xfffe	符号与父级绑定。父级是指导致此动态库被装入的第一个目标文件。
SYMINFO_BT_NONE	0xfffd	符号没有任何特殊的符号绑定。

si_flags

此位字段可以设置标志，如下表所示。

名称	值	含义
SYMINFO_FLG_DIRECT	0x01	符号引用与包含定义的目标文件直接关联。

名称	值	含义
SYMINFO_FLG_COPY	0x04	符号定义通过副本重定位生成。
SYMINFO_FLG_LAZYLOAD	0x08	符号引用应延迟装入的目标文件。
SYMINFO_FLG_DIRECTBIND	0x10	符号引用应与定义直接绑定。
SYMINFO_FLG_NOEXTDIRECT	0x20	不允许将外部引用与此符号定义直接绑定。

版本控制节

链接编辑器创建的目标文件可以包含以下两种类型的版本控制信息：

- **版本定义**，用于提供全局符号关联，并使用类型为 SHT_SUNW_verdef 和 SHT_SUNW_versym 的节实现。
- **版本依赖性**，用于指明其他目标文件依赖性的版本定义要求，并使用类型为 SHT_SUNW_verneed 的节实现。

sys/link.h 中定义了这些节的组成结构。包含版本控制信息的节名为 .SUNW_version 。

版本定义节

此节由 SHT_SUNW_verdef 类型定义。如果此节存在，则必须同时存在 SHT_SUNW_versym 节。这两种结构在文件中提供符号与版本定义之间的关联。 请参见创建版本定义。此节中的元素具有以下结构：

```
typedef struct {
        Elf32_Half      vd_version;
        Elf32_Half      vd_flags;
        Elf32_Half      vd_ndx;
        Elf32_Half      vd_cnt;
        Elf32_Word      vd_hash;
        Elf32_Word      vd_aux;
        Elf32_Word      vd_next;
} Elf32_Verdef;

typedef struct {
        Elf32_Word      vda_name;
        Elf32_Word      vda_next;
} Elf32_Verdaux;

typedef struct {
        Elf64_Half      vd_version;
        Elf64_Half      vd_flags;
        Elf64_Half      vd_ndx;
        Elf64_Half      vd_cnt;
        Elf64_Word      vd_hash;
        Elf64_Word      vd_aux;
```

```
        Elf64_Word      vd_next;
} Elf64_Verdef;


typedef struct {
        Elf64_Word      vda_name;
        Elf64_Word      vda_next;
} Elf64_Verdaux;
```

vd_version

此成员标识该结构的版本，如下表中所列。

名称	值	含义
VER_DEF_NONE	0	无效版本。
VER_DEF_CURRENT	>=1	当前版本。

值 1 表示原始节格式。扩展要求使用更大数字的新版本。 VER_DEF_CURRENT 的值可根据需要进行更改，以反映当前版本号。

vd_flags

此成员包含特定于版本定义的信息，如下表中所列。

名称	值	含义
VER_FLG_BASE	0x1	文件的版本定义。
名称 VER_FLG_WEAK	**值** 0x2	**含义** 弱版本标识符。

对文件应用版本定义或符号自动缩减后，基版本定义将始终存在。基版本可为文件保留的符号提供缺省版本。弱版本定义 (weak version definition) 没有与版本关联的符号。 请参见创建弱版本定义 (weak version definition)。

vd_ndx

版本索引。每个版本定义都有一个唯一的索引，用于将 SHT_SUNW_versym 项与相应的版本定义关联。

vd_cnt

Elf32_Verdaux 数组中的元素数目。

vd_hash

版本定义名称的散列值。该值是通过使用散列表节中介绍的同一散列函数生成的。

vd_aux

从此 Elf32_Verdef 项的开头到版本定义名称的 Elf32_Verdaux 数组的字节偏移。该数组中的第一个元素必须存在。此元素指向该结构定义的版本定义字符串。也可以存在其他元素。元素数目由 vd_cnt 值表示。这些元素表示此版本定义的依赖项。每种依赖项都会具有各自的版本定义结构。

vd_next

从此 Elf32_Verdef 结构的开头到下一个 Elf32_Verdef 项的字节偏移。

vda_name

以空字符结尾的字符串的字符串表偏移，用于提供版本定义的名称。

vda_next

从此 Elf32_Verdaux 项的开头到下一个 Elf32_Verdaux 项的字节偏移。

版本符号节

版本符号节由类型 SHT_SUNW_versym 定义。此节包含具有以下结构的元素的数组。

```
typedef Elf32_Half      Elf32_Versym;

typedef Elf64_Half      Elf64_Versym;
```

该数组的元素数目必须等于关联符号表中包含的符号表项的数目。此数目根据该节的 sh_link 值确定。该数组的每一个元素都包含一个索引，这些索引可以具有下表中所示的值。

表 7–24 ELF 版本依赖性索引

名称	值	含义
VER_NDX_LOCAL	0	符号具有局部范围的索引。
VER_NDX_GLOBAL	1	符号具有全局范围的索引，并且会指定给基版本定义。
	>1	符号具有全局范围的索引，并且会指定给用户定义的版本定义。

任何大于 VER_NDX_GLOBAL 的索引值都必须与 SHT_SUNW_verdef 节中的项的 vd_ndx 值对应。如果不存在大于 VER_NDX_GLOBAL 的索引值，则无需存在 SHT_SUNW_verdef 节。

版本依赖性节

版本依赖性节由 SHT_SUNW_verneed 类型定义。此节通过指明动态依赖项所需的版本定义，对文件的动态依赖性要求进行补充。仅当依赖项包含版本定义时，才会在此节中进行记录。此节中的元素具有以下结构：

```
typedef struct {
        Elf32_Half      vn_version;
        Elf32_Half      vn_cnt;
        Elf32_Word      vn_file;
        Elf32_Word      vn_aux;
        Elf32_Word      vn_next;
} Elf32_Verneed;


typedef struct {
        Elf32_Word      vna_hash;
        Elf32_Half      vna_flags;
        Elf32_Half      vna_other;
        Elf32_Word      vna_name;
        Elf32_Word      vna_next;
} Elf32_Vernaux;


typedef struct {
        Elf64_Half      vn_version;
        Elf64_Half      vn_cnt;
```

```
        Elf64_Word      vn_file;
        Elf64_Word      vn_aux;
        Elf64_Word      vn_next;
} Elf64_Verneed;

typedef struct {
        Elf64_Word      vna_hash;
        Elf64_Half      vna_flags;
        Elf64_Half      vna_other;
        Elf64_Word      vna_name;
        Elf64_Word      vna_next;
} Elf64_Vernaux;
```

vn_version

此成员标识该结构的版本，如下表中所列。

名称	值	含义
VER_NEED_NONE	0	无效版本。
VER_NEED_CURRENT	>=1	当前版本。

值 1 表示原始节格式。扩展要求使用更大数字的新版本。 VER_NEED_CURRENT 的值可根据需要进行更改，以反映当前版本号。

vn_cnt

Elf32_Vernaux 数组中的元素数目。

vn_file

以空字符结尾的字符串的字符串表偏移，用于提供版本依赖性的文件名。此名称与文件中找到的 .dynamic 依赖项之一匹配。 请参见动态节。

vn_aux

字节偏移，范围从此 Elf32_Verneed 项的开头到关联文件依赖项所需的版本定义的 Elf32_Vernaux 数组。必须存在至少一种版本依赖性。也可以存在其他版本依赖性，具体数目由 vn_cnt 值表示。

vn_next

从此 Elf32_Verneed 项的开头到下一个 Elf32_Verneed 项的字节偏移。

vna_hash

版本依赖性名称的散列值。该值是通过使用散列表节中介绍的同一散列函数生成的。

vna_flags

版本依赖性特定信息，如下表中所列。

名称	值	含义
VER_FLG_WEAK	0x2	弱版本标识符。

弱版本依赖性表示与弱版本定义 (weak version definition) 的原始绑定。

vna_other

目前未使用。

vna_name

以空字符结尾的字符串的字符串表偏移，用于提供版本依赖性的名称。

vna_next

从此 `Elf32_Vernaux` 项的开头到下一个 `Elf32_Vernaux` 项的字节偏移。

Previous: 第 6 章 支持接口 **Next**: 动态链接