

ELF 格式解析

—— 基于 ELF 规范 v1.2 版本

赵 凤 阳

目 录

修订历史.....	4
前言.....	5
名词对照表.....	7
第 1 章 ELF文件的静态结构	8
1.1 介绍.....	8
1.2 文件格式概述.....	9
1.3 ELF文件头	12
1.4 节.....	20
1.5 特殊节.....	28
1.6 字符串表.....	33
1.7 符号表.....	35
1.8 重定位.....	41
第 2 章 ELF文件的装载与动态连接	46
2.1 介绍.....	46
2.2 程序头.....	47
2.2.1 程序头结构.....	47
2.2.2 基地址.....	50
2.2.3 段权限.....	50
2.3 段内容.....	52
2.4 注释段.....	54
2.5 程序装载.....	56
2.6 动态连接.....	60
2.6.1 程序解析器.....	60
2.6.2 动态连接器.....	60
2.6.3 动态段.....	62
2.6.4 共享目标的依赖关系.....	67
2.6.5 全局偏移量表.....	68
2.6.6 函数地址.....	69
2.6.7 函数连接表.....	70
2.6.8 解析符号.....	71
2.7 哈希表.....	73
2.8 初始化和终止函数.....	75
2.9 程序解析器.....	77
第 3 章 示例程序.....	78
3.1 ELF文件头	78
3.2 节头表.....	80

3.3	节头字符串表.....	82
3.4	字符串表.....	82
3.5	代码节.....	83
3.6	符号表.....	85
3.7	段.....	86
3.8	动态节.....	87
附录A	源文件清单.....	90
附录B	输出文件清单.....	92

修订历史

版本	时间	概述
0.0	Aug 23, 2010	开始翻译。
0.1	Sep 19, 2010	完成原文翻译。
0.2	Sep 25, 2010	调整文档结构，重分章节。
0.3	Sep 30, 2010	加入一个 C 程序示例，并讲解。
0.4	Oct 9, 2010	补充一些动态段的内容；调整格式；加入名词对照表；修改前言。
0.5	Oct 10, 2010	校对、勘误。
1.0	Oct 10, 2010	发布第一个正式版本。

前言

关于 ELF 格式

ELF(Executable and Linking Format), 即“可执行可连接格式”, 最初由 UNIX 系统实验室(UNIX System Laboratories – USL)做为应用程序二进制接口(Application Binary Interface - ABI)的一部分而制定和发布。ELF 作为一种可移植的格式, 被 TIS 应用于基于 Intel 架构 32 位计算机的各种操作系统上。

ELF 的最大特点在于它有比较广泛的适用性, 通用的二进制接口定义使之可以平滑地移植到多种不同的操作环境上。这样, 不需要为每一种操作系统都定义一套不同的接口, 因此减少了软件的重复编码与编译, 加强了软件的可移植性。

ELF 文件格式规范由 TIS(Tool Interface Standards – 工具接口标准)委员会制定, TIS 委员会是一个微型计算机工业的联合组织, 它致力于为 32 位操作系统下的开发工具提供标准化的软件接口。这种接口包括目标标志格式、可执行文件格式, 以及调试信息的格式。

微型计算机工业中存在着各个不同的设备提供商, 硬件环境有很大区别。TIS 委员会的目标是在不同的硬件环境上建立起统一的软件规范, 使软件在各种不同的硬件平台上有尽可能大的移植性。这样, 软件开发就不会被硬件的区别所羁绊, 具有一定的“流线形”, 在不同的硬件平台间平滑地移植。TIS 为此而制定了许多规范, 有些用于主流的商用操作系统, 还有些用于 32 位的 Windows 个人电脑操作系统。这些规范最初在 TIS Portable Formats Specification 规范的 1.1 版本中一起发布。后来, 到了 1.2 版本, 这些规范被分开, 并被各自独立地发布。

TIS 委员会的成员来自 Absoft, Autodesk, Borland, IBM, Intel, Lahey, Lotus, MetaWare, Microtec Research, Microsoft, Novell, The Santa Cruz Operation, WATCOM, PharLap 以及 Symantec。

与 TIS 的其它各种规范一样, ELF 规范是基于已存在的、并已被事实证明可行的格式而制定的, 而且这种格式已经被 TIS 成员们广泛地使用在他们的软件系统中。所以当格式不能满足要求而需要改进的时候, 应首先考虑扩展已有的规范而不是制定新的。

关于本文档

由于考虑到对各种不同硬件和操作系统的适用性和扩展性, ELF(v1.2)规范在制定时, 把 ELF 格式分为了三个层次。第一层是基本的部分, 即格式中通用的部分, 这部分在各种处理器架构和操作系统上都是相同的; 第二层是处理器的扩展部分, 这部分会因处理器架构的不同而不同, 规范中只定义了针对 Intel i386 架构的

内容，针对其它的处理器的内容由处理器厂商自己提供；第三层是操作系统的扩展部分，这部分内容在不同的操作系统下面也可能是不同的，规范中只定义了针对 UNIX System V Release 4 操作系统的内容。

ELF(v1.2)是一个通用的规范，所以把上述三块内容界定得很严格。但本文只是针对于 Intel i386 架构和 UNIX 系统，所以并不把他们区分得很清楚，而是放在一起，关注知识的连贯性和完整性。

本文总体分为三章，第一章描述 ELF 文件的静态格式，第二章描述 ELF 文件在装载和连接时的动态特性，第三章以一个简单的 C 程序为示例，演示前两章所描述的内容。最后的两个附录中分别提供了第三章示例程序的源文件和相关的输出文件。示例程序是在 Intel i386 架构和 Linux 系统上完成，而 Linux 是类 UNIX 操作系统，本文述及的内容在两者之间并没有明显区别，所以本规范内容对示例程序完全适用。

由于本文描述的是 ELF 规范，是规范就会有一些比较抽象的内容。特别是第二章所描述的动态连接部分，各种操作系统可以有不同的实现，所以规范中只给出了原则性的规定，显得不够具体。如果您有兴趣研究更多的细节，我建议您阅读《程序员的自我修养——链接、装载与库》这本书，里面有非常详细的讲解。

关于作者

称为“作者”其实不太合适，本人主要是一个“译者”。最初，我只是想仔细地学习一下 ELF 文件格式规范以满足自己技术方面的好奇；在阅读 ELF 规范的过程中，觉得不如一边读一边把文本翻译过来，以分享给更多的人；当翻译完成的时候，发现“规范”文本由于要保持其严格性，因而损失了可读性，再加上我理解和翻译水平的限制，译文比原文又差了一些，所以最后决定改变文章的结构，加入一些自己的描述，并配以实例，提高可读性，这样可能会满足更多人的需要。因此本文也就不再是一篇严格的译文了。

本文是我用业余时间完成的，由于能力和精力所限，一定存在很多错误。在成文过程中，我也深感要完成一篇条理清晰、内容丰富、格式精美的文章需要很多的努力。我为本文做了版本记录，就是为了以后还能持续地改正和完善。如果您发现本文的不足，我非常愿意接受您的批评和建议。

联系方式：

E-mail: fion2009.z@gmail.com

备用邮箱及MSN: fyzhao2004@hotmail.com

作者赵凤阳 现就职于爱立信(Ericsson)。

名词对照表

可执行可连接格式	:	ELF
ELF 文件头	:	ELF header
基地址	:	base address
动态连接器	:	dynamic linker
动态连接	:	dynamic linking
全局偏移量表	:	global offset table
哈希表	:	hash table
初始化函数	:	initialization function
连接编辑器	:	link editor
目标文件	:	object file
函数连接表	:	procedure linkage table
程序头	:	program header
程序头表	:	program header table
程序解析器	:	program interpreter
重定位	:	relocation
共享目标	:	shared object
节	:	section
节头	:	section header
节头表	:	section header table
段	:	segment
字符串表	:	string table
符号表	:	symbol table
终止函数	:	termination function

第1章 ELF文件的静态结构

1.1 介绍

本章描述 ELF 文件的格式。ELF，即 Executable and Linking Format，译为“可执行可连接格式”，具有这种格式的文件称为 ELF 文件。

ELF 规范中把 ELF 文件宽泛地称为“目标文件 (object file)”，这与我们平时的理解不同。一般地，我们把经过编译但没有连接的文件(比如 Unix/Linux 上的.o 文件)称为目标文件，而 ELF 文件仅指连接好的可执行文件；在 ELF 规范中，所有符合 ELF 格式规范的都称为 ELF 文件，也称为目标文件，这两个名字是相同的，而经过编译但没有连接的文件则称为“可重定位文件 (relocatable file)”或“待重定位文件 (relocatable file)”。本文采用与此规范相同的命名方式，所以当提到可重定位文件时，一般可以理解为惯常所说的目标文件；而提到目标文件时，即指各种类型的 ELF 文件。

目标文件/ELF 文件主要分为以下三种类型：

- 可重定位文件(relocatable file)，用于与其它目标文件进行连接以构建可执行文件或动态链接库。可重定位文件就是常说的目标文件，由源文件编译而成，但还没有连接成可执行文件。在 UNIX 系统下，一般有扩展名“.o”。之所以称其为“可重定位”，是因为在这些文件中，如果引用到其它目标文件或库文件中定义的符号（变量或者函数）的话，只是给出一个名字，这里还并不知道这个符号在哪里，其具体的地址是什么。需要在连接的过程中，把对这些外部符号的引用重新定位到其真正定义的位置上，所以称目标文件为“可重定位”或者“待重定位”的。
- 共享目标文件(shared object file)，即动态链接库文件。它在以下两种情况下被使用：第一，在连接过程中与其它动态链接库或可重定位文件一起构建新的目标文件；第二，在可执行文件被加载的过程中，被动态链接到新的进程中，成为运行代码的一部分。
- 可执行文件(executable file)，经过连接的，可以执行的程序文件。

目标文件是由汇编器(assembler)和连接编辑器(link editor)生成的，内容是二进制，而非可读的文本形式，是可以直接在处理器上运行的代码。

在本章接下来的内容里，主要讨论 ELF 文件的格式和它如何用于构建一个程序。

1.2 文件格式概述

如前所述，目标文件的作用有两个，一是用于构建程序，构建动态链接库或都可执行程序，主要体现在连接的过程；二是用于运行程序。在这两种情况下，我们可以从不同的视角来看待同一个目标文件。对于同一个目标文件，当它分别被用于连接和用于执行的时候，其特性是不一样的，我们所关注的内容也不一样。

从连接的角度和运行的角度，可以分别把目标文件的组成部分做以下的划分：

图 1-1 目标文件格式

连接视图	运行视图
ELF 文件头	ELF 文件头
程序头表(可选)	程序头表
第 1 节	第 1 段
第 2 节	
...	
第 n 节	第 2 段
...	
...	
“节”头表	“段”头表(可选)

ELF 文件头 (ELF header)

位于文件的最开始处，包含有整个文件的结构信息。

节 (section)

是专用于连接过程而言的，在每个“节”中包含有指令数据、符号数据、重定位数据等等。

程序头表 (program header table)

在运行过程中是必须的，在连接过程中是可选的，因为它的作用是告诉系统如何创建进程的镜像。

节头表 (section header table)

包含有文件中所有“节”的信息。在连接视图中，“节头表”是必须存在的，文件里的每一个“节”都需要在“节头表”中有一个对应的注册项，这个注册项描述了节的名字、大小等等。

在上图中，程序头表紧跟在 ELF 文件头之后，节头表紧跟在节信息之后，但在实际的文件中，这个顺序并不是固定的。在 ELF 文件的各个组成部分中，只有 ELF 文件头的位置是固定的，其它内容的位置全都可变。

数据表示法

目标文件格式支持各种 8 位和 32 位的处理器架构，但不仅限于此。如果处理器的字长更长的话，ELF 格式仍然有效。对于控制数据，ELF 定义了自己的数据结构，所以并不依赖于所在机器的字长；其它数据使用目标处理器的数据格式，字长是在构建期间由编译器/连接器来指定的，与创建时所在的主机无关。

图 1-2 32 位数据类型

名字	字节数	对齐字节数	目的
Elf32_Addr	4	4	无符号地址
Elf32_Half	2	2	无符号半整型
Elf32_Off	4	4	无符号文件偏移量
Elf32_Sword	4	4	有符号整型
Elf32_Word	4	4	无符号整型
unsigned char	1	1	无符号短整型

目标文件中所定义的所有数据结构遵循通用的大小和对齐原则。一般来说，数据结构会向 4 字节对齐。比如，在 32 位体系结构中，一个整型数是 32 位，在目标文件的某个结构中，当出现一个整型数时，一定会保证这个整型数的地址是向 32 位对齐的，即使前面有空字节，也只会以空白填充。

在 ELF 格式中，出于可移植性的考虑，没有定义面向“比特(bit)”的数据结构。

文本字符的表示法

这里描述一下 ELF 文件中文本字符的默认表示法，并且定义出用于外部文件的标准字符集，这种字符集在不同的系统间应当是可移植的。

有些外部文件使用可读的文本来表示控制信息，这些文本使用 7 位的 ASCII 字符集来编码。在一个 ELF 文件里，记录字符常量也使用 7 位的 ASCII 字符。比如，当表示一个字符 `'/'` 或者 `'\n'` 时，出现在文件对应位置的将是单字节数字 47 和 10，而这也正是前述两个字符的 ASCII 编码。

如果要表示的字符不是基本的编码为 0~127 的 ASCII 字符，它可能仍然只占一个字节(8 比特)，也可能要占多个字节，具体占多少要看字符被如何编码。应用程序可以根据需要来选择所使用的字符集，比如，在有多国语言支持的情况下就需要使用相应的扩展字符集。TIS 委员会不限制所使用的字符集，但应用程序需要遵守以下几条基本的原则：

- 所选字符集必须与 7 位 ASCII 字符集保持兼容，即所选字符集中编码为 0~127 的字符必须与 ASCII 字符相同。
- 所选字符集中，编码值大于 127 的字符，不论是单字节还是多字节，它的每一个字节的值都不能在 0~127 之间。即一个编码值大于 127 的字符中，不能内嵌有基本 ASCII 字符。
- 所选字符集中，多字节字符必须是“自识别”的。也就是说，每一个字符是独立的，一个字符的认定/识别是不依赖于任何条件的。比如，向两个多字节字符中插入一新的字符时，并不改变原来两个字符的识别。

以上三条原则对于有多语言支持的应用程序尤其适合。

这里有几种字符串名要特别注意一下。在 ELF 文件中，有一些约定俗成的命名格式用于指明处理器定制的特别内容。ELF 文件中可能出现以 `DT_` 或 `PT_` 为前缀的名字，比如 `DT_M32_SPECIAL` 用于指明这是针对 M32 处理器的特别内容，它的命令格式为“`DT_处理器名_SPECIAL`”。但是，还有一些被定义得更早的处理器扩展并没有遵循这一命名规则，这些命名也应该被支持，比如 `DT_JMP_REL`。

1.3 ELF文件头

ELF 文件头(ELF header)位于目标文件最开始的位置，含有整个文件的一些基本信息。

文件头中含有整个文件的结构信息，包括一些控制单元的大小。有的时候，文件头所描述的大小会与控制单元实际的大小不相符，可能比实际的大，也可能会比实际的小。如果文件头所描述的大小比实际的控制单元大，程序在被加载时应忽略多余的部分；而如果文件头所描述的大小比实际的小，如何处理缺失的部分将依赖于程序运行的上下文以及是否有可用的扩展部分。

可以用以下这个数据结构体来描述文件头。

图 1-3 ELF 文件头

```
#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;
```

其中各个成员的意义如下：

e_ident

最开始处的这 16 个字节含有 ELF 文件的识别标志，并且提供了一些用于解码和解析文件内容的信息，是不依赖于具体操作系统的。

在前面提到过，ELF 格式提供的目标文件框架可以支持多种处理器，以及多种编码方式。针对不同的体系结构和编码格式，ELF 文件的内容是会截然不同的。如果不知道编码格式，系统将无法知道怎么去读取目标文件；如果系统结构与本机不同，也将无法解析和运行。这些信息需要以独立的格式存放在一个默认的地方，所有系统都约定好从文件的同一个地方来读取这些信息，这就是 ELF 标识的作用。

ELF 文件最开始的这一部分的格式是固定并通用的，在所有平台上都一样。所有处理器都可能用固定的格式去读取这一部分的内容，从而获知这个 ELF 文件中接下来的内容应该如何读取和解析。

ELF 标识，即前述 ELF 文件头结构最开始的 16 个字节，作为一个数组，它的各个索引位置的字节数据有固定的含义。

图 1-4 e_ident[] 数组各个索引位置的含义

名字	值	用途
EI_MAG0	0	文件标志
EI_MAG1	1	文件标志
EI_MAG2	2	文件标志
EI_MAG3	3	文件标志
EI_CLASS	4	文件类别
EI_DATA	5	编码格式
EI_VERSION	6	文件版本
EI_PAD	7	补齐字节开始位置
EI_NIDENT	16	e_ident[] 数组的大小

在上述各索引位置中将存放的数据如下：

EI_MAG0 ~ EI_MAG3

文件的最前面 4 字节 `e_ident[EI_MAG0] ~ e_ident[EI_MAG3]` 的内容被称为“魔数”，用于标识这是一个 ELF 文件。这四个字节存放的内容是固定的：

名字	值	意义
ELFMAG0	0x7f	<code>e_ident[EI_MAG0]</code>
ELFMAG1	'E'	<code>e_ident[EI_MAG1]</code>
ELFMAG2	'L'	<code>e_ident[EI_MAG2]</code>
ELFMAG3	'F'	<code>e_ident[EI_MAG3]</code>

ELFCLASS

接下来的一个字节 `e_ident[EI_CLASS]` 指明文件的类型，或者说容量：

名字	值	意义
ELFCLASSNONE	0	非法目标文件
ELFCLASS32	1	32 位目标文件
ELFCLASS64	2	64 位目标文件

ELF 文件格式的设计目标之一是适应于多种字长大小的系统。目前支持两种字长，32 位和 64 位。32 位字长的文件(ELFCLASS32)所支持的系统，其进程地址空间为 4G 字节，文件的最大容量也是 4G。64 位字长的文件(ELFCLASS64)格式定义目前还没有完成，这是针对 64 位处理器而设计的，它的定义目前只代表未来的趋势和努力的方向。

ELFDATA

再下面的一个字节 `e_ident[EI_DATA]` 指明了目标文件中的数据编码格式，目前支持以下几种：

名字	值	意义
ELFDATANONE	0	非法编码格式
ELFDATA2LSB	1	LSB 编码(小头编码)
ELFDATA2MSB	2	MSB 编码(大头编码)

ELFDATA2LSB

即平常说的“小头编码”，在一个多字节的数据中，“权值”比较小的低位数在前（低字节）；

ELFDATA2MSB

即平常说的“大头编码”，在一个多字节的数据中，“权值”比较大的高位数在前（低字节）。

如下图所示，左边的数字是所要表示的 16 进制数，右边的方块为字节，方块的左上角为该字节的位置。

图 1-5，LSB 编码(ELFDATA2LSB)

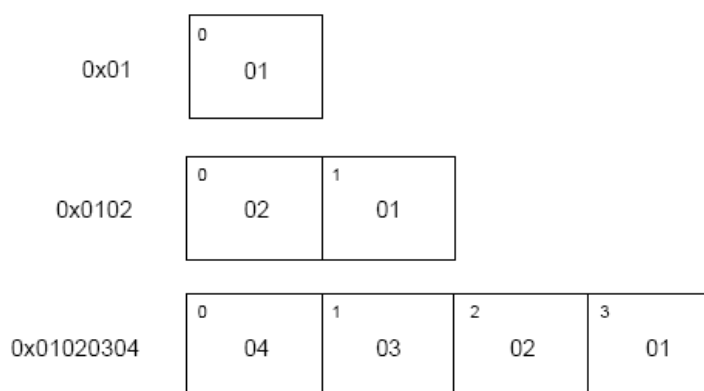
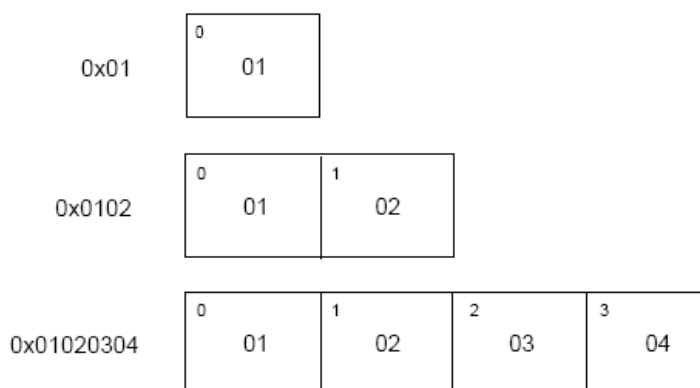


图 1-6，MSB 编码(ELFDATA2MSB)



在 Intel 架构中，e_ident[EI_DATA]取值为 ELFDATA2LSB。

EI_VERSION

接下来的字节 e_ident[EI_VERSION]指明 ELF 文件头的版本，目前这个版本号是 EV_CURRENT，即“1”。

EI_PAD

从 `e_ident[EI_PAD]` 到 `e_ident[EI_NIDENT-1]` 之间的 9 个字节目前暂时不使用，留作以后扩展，在实际的文件中应被填 0 补充，其它程序在读取 ELF 文件头时应该忽略这些字节。如果以后 ELF 文件头的内容被扩展，这 9 个字节中有一些被使用起来的话，`EI_PAD` 将被定义得更大。

e_type

此字段表明本目标文件属于哪种类型

名字	值	意义
ET_NONE	0	未知文件类型
ET_REL	1	可重定位文件
ET_EXEC	2	可执行文件
ET_DYN	3	动态链接库文件
ET_CORE	4	Core 文件
ET_LOPROC	0xff00	特定处理器文件扩展下边界
ET_HIPROC	0xffff	特定处理器文件扩展上边界

在上述文件类型中，`core` 文件类型目前还不支持，但仍然为它保留了这一关键字。

`ET_LOPROC ~ ET_HIPROC (0xff00 ~ 0xffff)` 这一范围内的文件类型是为特定处理器而保留的，如果需要为某种处理器专门设定文件格式，可以从这一范围内选取一个做为标识。

在以上已定义范围外的文件类型均为保留类型，留做以后可能的扩展。

e_machine

此字段用于指定该文件适用的处理器体系结构。

名字	值	意义
EM_NONE	0	未知体系结构
EM_M32	1	AT&T WE 32100

EM_SPARC	2	SPARC
EM_386	3	Intel Architecture
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000 Big-Endian
EM_MIPS_RS4-BE	10	MIPS RS4000 Big-Endian
RESERVED	11 ~ 16	保留未用

对于 Intel 架构处理器，`e_machine` 成员取值固定为 `EM_386`。

在以上已定义范围外的处理器类型均为保留的，在需要的时候将分配给新出现的处理器使用。这里所定义的这些处理器类型名字都以 `ET_` 为前缀，后接处理器名。这些处理器名在下文中可能会再次被用到，在下文中会遇到类似的一些定义，它们可能是一些类型或者一些属性。比如，定义某一个属性，它以 `EF_` 为前缀，以 `_WIDGET` 结尾，而中间的处理器名部分正是上表中某一项“`EM_XYZ`”的后缀“`XYZ`”，那么，这个属性就写做“`EF_XYZ_WIDGET`”。

在 ELF 文件中，处理器厂商可以定义一些自有的常量，这类常量的命名必须要遵守一定的规则，只要合乎这一规则，这些名字就都是合法的，ELF 文件解析器就需要能够识别他们。这种规则一般是以某种前缀（比如 `DT_` 或 `PT_`）开头，加上处理器的名字（比如 `M32`），再加上 `_SPECIAL` 后缀构成，比如 `DT_M32_SPECIAL`。

由于历史的原因，规范早期所定义的一些常量名字与这种命名约定相混淆，这种特殊的名字也要能够支持。但到目前为止，这类情况只有一种，即 `DT_JMP_REL`。

在 ELF 文件中，处理器厂商还可以定义一些自有的段名字。其命名规则是在原来的段名字之前再加上一个处理器架构的缩写。这个名字应该与 ELF 文件头中的 `e_machine` 字段相配套。比如，`.FOO.psect` 这个名字就是为 `FOO` 架构定义的 `psect` 节。下列已存在的名字与这一规则有混淆，需要被特别支持。

<code>.sdata</code>	<code>.lit4</code>	<code>.gptab</code>
<code>.tdesc</code>	<code>.lit8</code>	<code>.liblist</code>

.sbss	.reginfo	.conflict
-------	----------	-----------

e_version

此字段指明目标文件的版本。

名字	值	意义
EV_NONE	0	非法版本号
EV_CURRENT	1	当前版本号

EV_CURRENT 是一个动态的数字，表示最新的版本。尽管当前最新的版本号就是“1”，但如果以后有更新的版本的话，EV_CURRENT 将被更新为更大的数字，而目前的“1”将成为历史版本。

e_entry

此字段指明程序入口的虚拟地址。即当文件被加载到进程空间里后，入口程序在进程地址空间里的地址。对于可执行程序文件来说，当 ELF 文件完成加载之后，程序将从这里开始运行；而对于其它文件来说，这个值应该是 0。

e_phoff

此字段指明程序头表(program header table)开始处在文件中的偏移量。如果没有程序头表，该值应设为 0。

e_shoff

此字段指明节头表(section header table)开始处在文件中的偏移量。如果没有节头表，该值应设为 0。

e_flags

此字段含有处理器特定的标志位。标志的名字符合“EF_machine_flag”的格式。对于 Intel 架构的处理器来说，它没有定义任何标志位，所以 e_flags 应该为 0。

e_ehsize

此字段表明 ELF 文件头的大小，以字节为单位。

e_phentsize

此字段表明在程序头表中每一个表项的大小，以字节为单位。

e_phnum

此字段表明程序头表中总共有多少个表项。如果一个目标文件中没有程序头表，该值应设为 0。

e_shentsize

此字段表明在节头表中每一个表项的大小，以字节为单位。

e_shnum

此字段表明节头表中总共有多少个表项。如果一个目标文件中没有节头表，该值应设为 0。

e_shstrndx

节头表中与节名字表相对应的表项的索引。如果文件没有节名字表，此值应设置为 SHN_UNDEF。

1.4 节

在目标文件中可以包含很多“节”(section)，所有这些“节”都登记在一张称为“节头表”(section header table)的数组里。节头表的每一个表项是一个 Elf32_Shdr 结构，通过每一个表项可以定位到对应的节。

在解释 ELF 文件头的时候提到过，在文件头中有一个 `e_shoff` 成员给出节头表在 ELF 文件中的位置，即相对于文件开始处的偏移量；`e_shnum` 成员指明节头表中包含多少个表项；`e_shentsize` 成员指明了每一个表项的大小。

某些表项的索引值被保留，有特殊的含义。ELF 文件的节头表中不会出现索引值为以下各值的表项：

图 1-7 特殊节索引值

名字	值
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_HIPROC	0xff1f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_HIRESERVE	0xffff

SHN_UNDEF

该值被定义为 0，它表示一个未定义的、不存在的节的索引。

尽管索引值 0 是一个未定义的保留值，但在节头表中的索引还是会从 0 开始。比如，如果文件头中 `e_shnum` 的值为 6，表明在节头表中存在 6 个表项，其索引值分别为从 0 到 5。其中 0 号表项的意义将在后面说明。

SHN_LORESERVE

被保留索引号区间的下限。

SHN_LOPROC

为处理器定制节所保留的索引号区间的下限。

SHN_HIPROC

为处理器定制节所保留的索引号区间的上限。

SHN_ABS

此节中所定义的符号有绝对的值，这个值不会因重定位而改变。

SHN_COMMON

此节中所定义的符号是公共的，比如 **FORTRAN COMMON** 符号或者未分配的 C 外部变量。

SHN_HIRESERVE

被保留索引号区间的上限。

除 0 以外，节头表中所有保留的索引值都位于 **SHN_LORESERVE ~ SHN_HIRESERVE** 范围之内。与索引值“0”不同，这些大数值的索引并不出现在节头表中。

通常，目标文件中含有众多的“节”，“节”区是文件中最大的部分，它们需要满足下列这些条件：

- 目标文件中的每一个节一定对应有一个节头(section header)，节头中有对节的描述信息；但有的节头可以没有对应的节，而只是一个空的头。
- 每一个节所占用的空间是连续的。
- 各个节之间不能互相重叠。
- 在目标文件中，节与节之间可能会存在一些多余的字节，这些字节不属于任何节。

下图的结构体给出了节头的结构。

图 1-8 节头(Section Header)结构

```
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;
```

sh_name

本节的名字。整个名字的字符串并不存储在这里，它仅是一个索引号，指向“字符串表”节中的某个位置，那里存储了一个以'\0'结尾的字符串。

sh_type

本节的类型。下表给出了所有的节类型。

图 1-9 节类型(section type) - sh_type 成员取值

名字	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_LOPROC	0x70000000
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

SHT_NULL

此值表明本节头是一个无效的（非活动的）节头，它也没有对应的节。本节头中的其它成员的值也都是没有意义的。

SHT_PROGBITS

此值表明本节所含有的信息是由程序定义的，本节内容的格式和含义都由程序来决定。

SHT_SYMTAB/SHT_DYNSYM

这两类节都含有符号表。目前，目标文件中最多只能各包含一个这两种节，但这种限制以后可能会取消。一般来说，**SHT_SYMTAB** 提供的符号用于在创建目标文件的时候编辑连接，在运行期间也有可能会用于动态连接。**SHT_SYMTAB** 包含完整的符号表，它往往会包含很多在运行期间(动态连接)用不到的符号。所以，一个目标文件可以再有一个 **SHT_DYNSYM** 节，它含有一个较小的符号表，专门用于动态连接。

SHT_STRTAB

此值表明本节是字符串表。目标文件中可以包含多个字符串表节。

SHT_RELA

此值表明本节是一个重定位节，含有带明确加数(addend)的重定位项，对于 32 位类型的目标文件来说，这个加数就是 **Elf32_Rela**。一个目标文件可能含有多个重定位节，详情参考“1.8 重定位”节。

SHT_HASH

此值表明本节包含一张哈希表。所有参与动态连接的目标文件都必须包含一个符号哈希表。目前，一个目标文件中最多只能有一个哈希表，但这一限制以后可能会取消。

SHT_DYNAMIC

此值表明本节包含的是动态连接信息。目前，一个目标文件中最多只能有一个 *SHT_DYNAMIC* 节，但这一限制以后可能会取消。

SHT_NOTE

此值表明本节包含的信息用于以某种方式来标记本文件。

SHT_NOBITS

此值表明这一节的内容是空的，节并不占用实际的空间。在定义 *sh_offset* 时提到过，这时 *sh_offset* 只代表一个逻辑上的位置概念，并不代表实际的内容。

SHT_REL

此值表明本节是一个重定位节，含有带明确加数的重定位项，对于 32 位类型的目标文件来说，这个加数就是 *Elf32_Rel*。一个目标文件可能含有多个重定位节，详情参考“1.8 重定位”节。

SHT_SHLIB

此值是一个保留值，暂未指定语义。

SHT_DYNSYM

此值表明本节是符号表。与 *SHT_SYMTAB* 同义。

SHT_LOPROC

为特殊处理器保留的节类型索引值的下边界。

SHT_HIPROC

为特殊处理器保留的节类型索引值的上边界。*SHT_LOPROC* ~ *SHT_HIPROC* 区间是为特殊处理器节类型的保留值。

SHT_LOUSER

为应用程序保留节类型索引值的下边界。

SHT_HIUSER

为应用程序保留节类型索引值的下边界。**SHT_LOUSER ~ SHT_HIUSER** 区间的节类型可由应用程序自行定义，是一段保留值。

sh_flags

本节的一些属性，由一系列标志比特位组成，各个比特定义了节的不同属性，当某种属性被设置时，相应的标志位被设为 1，反之则设为 0。未定义的标志位被全部置 0。

以下是这些标志位的列表及含义。

图 1-10 节的属性标志位

名字	值
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MASKPROC	0xf0000000

SHF_WRITE

如果此标志被设置，表示本节所包含的内容在进程运行过程中是可写的。

SHF_ALLOC

如果此标志被设置，表示本节内容在进程运行过程中要占用内存单元。并不是所有节都会占用实际的内存，有一些起控制作用的节，在目标文件映射到进程空间时，并不需要占用内存。

SHF_EXECINSTR

如果此标志被设置，表示此节内容是指令代码。

SHF_MASKPROC

所有被此值所覆盖的位都是保留做特殊处理器扩展用的。

sh_addr

如果本节的内容需要映射到进程空间中去，此成员指定映射的起始地址；如果不需要映射，此值为 0。

sh_offset

指明了本节所在的位置，该值是节的第一个字节在文件中的位置，即相对于文件开头的偏移量。单位是字节。如果该节的类型为 **SHT_NOBITS** 的话，表明这一节的内容是空的，节并不占用实际的空间，这时 **sh_offset** 只代表一个逻辑上的位置概念，并不代表实际的内容。

sh_size

指明节的大小，单位是字节。如果该节的类型为 **SHT_NOBITS**，此值仍然可能为非零，但没有实际的意义。

sh_link

此成员是一个索引值，指向节头表中本节所对应的位置。根据节的类型不同，本成员的意义也有所不同，具体见下表。

sh_info

此成员含有此节的附加信息，根据节的类型不同，本成员的意义也有所不同。

对于某些节类型来说，**sh_link** 和 **sh_info** 含有特殊的信息，见下表。

图 1-11 不同类型的节中 **sh_link** 和 **sh_info** 的不同意义

sh_type	sh_link	sh_info
SHT_DYNAMIC	用于本节中项目的字符串表在节头表中相应的索引值	0
SHT_HASH	用于本节中哈希表的符号表在节头表中相应的索引值	0
SHT_REL / SHT_RELA	相应符号表在节头表中的索引值	本重定位节所应用到目标节在节头表中的索引值
SHT_SYMTAB / SHT_DYNSYM	相关字符串表的节头索引	符号表中最后一个本地符号的索引值加 1
其它	SHN_UNDEF	0

sh_addralign

此成员指明本节内容如何对齐字节，即该节的地址应该向多少个字节对齐。比如，在这个节中如果含有一个双字(doubleword)，系统必须保证整个节的内容向双字对齐。也就是说，本节内容在进程空间中的映射地址 **sh_addr** 必须是一个向

sh_addralign 对齐，即能被 sh_addralign 整除的值。目前，sh_addralign 只能取 0、1 或者 2 的正整数倍。如果该值为 0 或 1，表明本节没有字节对齐约束。

sh_entsize

有一些节的内容是一张表，其中每一个表项的大小是固定的，比如符号表。对于这种表来说，本成员指定其每一个表项的大小。如果此值为 0 则表明本节内容不是这种表格。

其它的节类型值是保留未用的。

下面说一下索引值为 0 的节头表项，它并不表达实际的内容，只是一个空的表项，内容见下表。

图 1-12 0 号节头表项

名字	值	意义
sh_name	0	无名字
sh_type	SHT_NULL	非活动类型
sh_flags	0	无标志
sh_addr	0	无地址
sh_offset	0	无文件内偏移量
sh_size	0	无大小
sh_link	SHN_UNDEF	无节头表对应项
sh_info	0	无附加信息
sh_addralign	0	无对齐格式
sh_entsize	0	无表项大小

1.5 特殊节

在 ELF 文件中有一些特定的节是预定义好的，其内容是指令代码或者控制信息。这些节专门为操作系统使用，对于不同的操作系统，这些节的类型和属性有所不同。

在构建可执行程序时，连接器(linker)可能需要把一些独立的目标文件和库文件连接在一起，在这个过程中，连接器要解析各个文件中的相互引用，调整某些目标文件中的绝对引用，并重定位指令码。

在第 2 章中将描述，当运行一个构建好的可执行程序时，在连接和装载过程中，需要读取目标文件的内容并把它们装入各个节中。

每种操作系统都有自己的一套连接模型，但总的来说，不外乎静态和动态两类：

静态连接：

所有的目标文件和动态连接库被静态地绑定在一起，所有的符号都被解析出来。所创建的目标文件是完整的，运行时不依赖于任何外部的库。

动态连接：

所有的目标文件、系统共享资源以及共享库以动态的形式连接在一起，外部库的内容没有完整地拷贝进来。如果创建的是可执行文件的话，程序在运行的时候，在构建时所依赖的那些库必须在系统中能找到，把它们一并装载之后，程序才能运行起来。运行期间如何解析那些动态连接进来的符号引用，不同的系统有各自不同的方式。

还有些节包含调试信息，比如.debug 和.line 节；还有些节包含程序控制信息，比如.bss, .data, .data1, .rodata 和.rodata1 这些节。

有一些节含有程序或控制信息，这些节由系统使用，有指定的类型和属性。它们中的大多数都将用于连接过程。动态连接过程所需要的信息由.dynsym、.dynstr、.interp、.hash、.dynamic、.rel、.rela、.got、.plt 等节提供。其中有些节(比如.plt 和.got)的内容依处理器而不同，但它们都支持同样的连接模型。

.init 和.fini 节用于进程的初始化和终止过程。

图 1-13 特殊节(special sections)

名字	类型	属性
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE

.comment	SHT_PROGBITS	无
.data	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.debug	SHT_PROGBITS	无
.dynamic	SHT_DYNAMIC	见下文
.dynstr	SHT_STRTAB	SHF_ALLOC
.dysym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	见下文
.line	SHT_PROGBITS	无
.note	SHT_NOTE	无
.plt	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.relname	SHT_REL	见下文
.relaname	SHT_RELA	见下文
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	无
.strtab	SHT_STRTAB	见下文
.symtab	SHT_SYMTAB	见下文
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

.bss

本节中包含目标文件中未初始化的全局变量。一般情况下，可执行程序在开始运行的时候，系统会把这一段内容清零。但是，在运行期间的 `bss` 段是由系统初始化而成的，在目标文件中 `.bss` 节并不包含任何内容，其长度为 0，所以它的节类型为 `SHT_NOBITS`。

.comment

本节包含版本控制信息。

.data/.data1

这两个节用于存放程序中被初始化过的全局变量。在目标文件中，它们是占用实际的存储空间的，与 `.bss` 节不同。

.debug

本节中含有调试信息，内容格式没有统一规定。所有以 `“.debug”` 为前缀的节名字都是保留的。

.dynamic

本节包含动态连接信息，并且可能有 `SHF_ALLOC` 和 `SHF_WRITE` 等属性。是否具有 `SHF_WRITE` 属性取决于操作系统和处理器。

.dynstr

此节含有用于动态连接的字符串，一般是那些与符号表相关的名字。更多信息参见第 2 章。

.dynsym

此节含有动态连接符号表。

.fini

此节包含进程终止时要执行的程序指令。当程序正常退出时，系统会执行这一节中的代码。

.got

此节包含全局偏移量表。

.hash

本节包含一张符号哈希表。

.init

此节包含进程初始化时要执行的程序指令。当程序开始运行时，系统会在进入主函数之前执行这一节中的代码。

.interp

此节含有 ELF 程序解析器的路径名。如果此节被包含在某个可装载的段中，那么本节的属性中应置 SHF_ALLOC 标志位，否则不置此标志。

.line

本节也是一个用于调试的节，它包含那些调试符号的行号，为程序指令码与源文件的行号建立起联系。其内容格式没有统一规定。

.note

本节所包含的信息在第 2 章“注释节(note section)”部分描述。

.plt

此节包含函数连接表。

.relname 和 *.relaname*

这两个节含有重定位信息。如果此节被包含在某个可装载的段中，那么本节的属性中应置 SHF_ALLOC 标志位，否则不置此标志。注意，这两个节的名称中“name”是可替换的部分，依照惯例，对哪一节做重定位就把“name”换成哪一节的名称。比如，.text 节的重定位节的名称将是 .rel.text 或 .rela.text。

.rodata/.rodata1

本节包含程序中的只读数据，在程序装载时，它们一般会被装入进程空间中那些只读的段中去。更详细内容参见第 2 章“程序头”部分。

.shstrtab

本节是“节名字表”，含有所有其它节的名称。

.strtab

本节用于存放字符串，主要是那些符号表项的名称。如果一个目标文件有一个可装载的段，并且其中含有符号表，那么本节的属性中应该有 SHF_ALLOC。

.symtab

本节用于存放符号表。如果一个目标文件有一个可载入的段，并且其中含有符号表，那么本节的属性中应该有 SHF_ALLOC。

.text

本节包含程序指令代码。

以点号“.”为前缀的节名字是为系统保留的。应用程序也可以构造自己的段，但最好不要取与上述系统已定义的节相同的名字，也不要取以点号开头的名字，以避免潜在的冲突。注意，目标文件中节的名字并不具有唯一性，可以存在多个相同名字的节。

1.6 字符串表

本节描述默认字符串表。

字符串表中包含有若干个以‘null’结尾的字符序列，即字符串。在目标文件中，这些字符串通常是符号的名字或者节的名字。在目标文件的其它部分中，当需要引用某个字符串时，只需要提供该字符串在字符串表中的序号即可。

字符串表中的第一个字符串（序号为 0）永远是空串，即“null”，它可以用于表示一个空的名字或者没有名字。所以，字符串表的第一个字节是‘null’。由于每一个字符串都是以‘null’结尾，所以字符串表的最后一个字节也必然为‘null’。

字符串表也可以是空的，不含有任何字符串，这时，节头中的 sh_size 成员必须是 0。

一个目标文件中可能有多个字符串表，其中一个称为“节名字表(.shstrtab)”，它包含所有节的名字。每一个节头的 sh_name 成员应该是一个索引值，它指向节名字表中的一个位置，从这个位置开始到接下来第一个‘null’字符为止的这个字符串，正是这个节的名字。

下图展示了一张长度为 25 字节的字符串表。

Index	+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					

图 1-14 字符串表序号示例

序号	字符串
0	null
1	name.
7	Variable
11	able
16	able
24	null

上面列表中左边一列中的引用全都是合法的。从这个示例中可以看出，既可以引用一个完整定义的字符串，即出现在两个'**null**'字符之间的整个串，也可以引用它的一部分。由于需要以'**null**'结尾，所以当引用一个子字符串时，一定是引用长串中靠后的那部分。比如，序号"11"就引用了一个子串。

1.7 符号表

目标文件中的“符号表(symbol table)”所包含的信息用于定位和重定位程序中的符号定义和引用。目标文件的其它部分通过一个符号在这个表中的索引值来使用该符号。索引值从 0 开始计数，但值为 0 的表项（即第一项）并没有实际的意义，它表示未定义的符号。这里用常量 `STN_UNDEF` 来表示未定义的符号。

一个符号表项的格式定义如下：

图 1-15 符号表项(Symbol Table Entry)

```
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;
```

st_name

符号的名字。但它并不是一个字符串，而是一个指向字符串表的索引值，在字符串表中对应位置上的字符串就是该符号名字的实际文本。如果此值为非 0，它代表符号名在字符串表中的索引值。如果此值为 0，那么此符号无名字。

st_value

符号的值。这个值其实没有固定的类型，它可能代表一个数值，也可以是一个地址，具体是什么要看上下文。

对于不同的目标文件类型，符号表项的 *st_value* 的含义略有不同：

- 在重定位文件中，如果一个符号对应的节的索引值是 `SHN_COMMON`，*st_value* 值是这个节内容的字节对齐数。
- 在重定位文件中，如果一个符号是已定义的，那么它的 *st_value* 值是该符号的起始地址在其所在节中的偏移量，而其所在的节的索引由 *st_shndx* 给出。

- 在可执行文件和共享库文件中，`st_value` 不再是一个节内的偏移量，而是一个虚拟地址，直接指向符号所在的内存位置。这种情况下，`st_shndx` 就不再需要了。

综合以上三点可知，在不同的目标文件中 `st_value` 的值多少有所不同，这样的设计是为了在各种类型的文件中更有效地访问数据。

如果一个可执行文件中含有一个函数的引用，而这个函数是定义在一个共享目标文件中，那么在可执行文件中，针对那个共享目标文件的符号表应该含有这个函数的符号。符号表的 `st_shndx` 成员值为 `SHN_UNDEF`，这就告诉了动态连接器，这个函数的符号定义并不在可执行文件中。如果已经在可执行文件中给这个符号申请了一个函数连接表项，而且符号表项的 `st_value` 成员不是 0，那么 `st_value` 值就将是函数连接表项中第一条指令的地址。否则，`st_value` 成员是 0。这个函数连接表项地址被动态连接器用来解析函数地址。细节参见下文“函数地址”。

st_size

符号的大小。各种符号的大小各不相同，比如一个对象的大小就是它实际占用的字节数。如果一个符号的大小为 0 或者大小未知，则这个值为 0。

st_info

符号的类型和属性。`st_info` 由一系列的比特位构成，标识了“符号绑定(symbol binding)”、“符号类型(symbol type)”和“符号信息(symbol information)”三种属性。下面几个宏分别用于读取这三种属性值。

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

下面分别说明：

符号绑定(Symbol Binding)

符号绑定属性由 `ELF32_ST_BIND` 指定。

图 1-16 符号绑定(Symbol Binding)，`ELF32_ST_BIND`

名字	值
<code>STB_LOCAL</code>	0
<code>STB_GLOBAL</code>	1

STB_WEAK	2
STB_LOPROC	13
STB_HIPROC	15

STB_LOCAL

表明本符号是一个本地符号。它只出现在本文件中，在本文件外该符号无效。所以在不同的文件中可以定义相同的符号名，它们之间不会互相影响。

STB_GLOBAL

表明本符号是一个全局符号。当有多个文件被连接在一起时，在所有文件中该符号都是可见的。正常情况下，在一个文件中定义的全局符号，一定是在其它文件中需要被引用，否则无须定义为全局。

STB_WEAK

类似于全局符号，但是相对于 ***STB_GLOBAL***，它们的优先级更低。

全局符号(global symbol)和弱符号(weak symbol)在以下两方面有区别：

- 当连接编辑器把若干个可重定位目标文件连接起来时，同名的 ***STB_GLOBAL*** 符号不允许出现多次。而如果在目标文件中已经定义了一个全局的符号(global symbol)，当一个同名的弱符号(weak symbol)出现时，并不会发生错误。连接编辑器会以全局符号为准，忽略弱符号。与全局符号相似，如果已经存在的是一个公用符号，即 `st_shndx` 域为 ***SHN_COMMON*** 值的符号，当一个同名的弱符号(weak symbol)出现时，也不会发生错误。连接编辑器会以公用符号为准，忽略弱符号。

- 在查找符号定义时，连接编辑器可能会搜索存档的库文件。如果是查找全局符号，连接编辑器会提取包含该未定义的全局符号的存档成员，存档成员可能是一个全局的符号，也可能是弱符号；而如果是查找弱符号，连接编辑器不会去提取存档成员。未解析的弱符号值为 0。

STB_LOPROC ~ STB_HIPROC

为特殊处理器保留的属性区间。

在符号表中，不同属性的符号所在位置也有不同，本地符号(***STB_LOCAL***)排在前面，全局符号(***STB_GLOBAL***/***STB_WEAK***)排在后面。

符号类型(Symbol Types)

符号类型属性由 `ELF32_ST_TYPE` 指定。

图 1-17 符号类型(Symbol Types), `ELF32_ST_TYPE`

名字	值
<code>STT_NOTYPE</code>	0
<code>STT_OBJECT</code>	1
<code>STT_FUNC</code>	2
<code>STT_SECTION</code>	3
<code>STT_FILE</code>	4
<code>STT_LOPROC</code>	13
<code>STT_HIPROC</code>	15

`STT_NOTYPE`

本符号类型未指定。

`STT_OBJECT`

本符号是一个数据对象，比如变量、数组等。

`STT_FUNC`

本符号是一个函数，或者其它的可执行代码。函数符号在共享目标文件中有特殊的意义。当另外一个目标文件引用一个共享目标文件中的函数符号时，连接编辑器为被引用符号自动创建一个连接表项。非 `STT_FUNC` 类型的共享目标符号不会通过这种连接表项被自动引用。

`STT_SECTION`

本符号与一个节相关联，用于重定位，通常具有 `STB_LOCAL` 属性。

`STT_FILE`

本符号是一个文件符号，它具有 `STB_LOCAL` 属性，它的节索引值是 `SHN_ABS`。在符号表中如果存在本类符号的话，它会出现在所有 `STB_LOCAL` 类符号的前部。

`STT_LOPROC` ~ `STT_HIPROC`

这一区间的符号类型为特殊处理器保留。

st_other

本数据成员目前暂未使用，在目标文件中一律赋值为 0。

st_shndx

任何一个符号表项的定义都与某一个“节”相联系，因为符号是为节而定义，在节中被引用。本数据成员即指明了相关联的节。本数据成员是一个索引值，它指向相关联的节在节头表中的索引。在重定位过程中，节的位置会改变，本数据成员的值也随之改变，继续指向节的新位置。当本数据成员指向下面三种特殊的节索引值时，本符号具有如下特别的意义：

SHN_ABS

符号的值是绝对的，具有常量性，在重定位过程中，此值不需要改变。

SHN_COMMON

本符号所关联的是一个还没有分配的公共节，本符号的值规定了其内容的字节对齐规则，与 *sh_addralign* 相似。也就是说，连接器会为本符号分配存储空间，而且其起始地址是向 *st_value* 对齐的。本符号的值指明了要分配的字节数。

SHN_UNDEF

当一个符号指向第 1 节(*SHN_UNDEF*)时，表明本符号在当前目标文件中未定义，在连接过程中，连接器会找到此符号被定义的文件，并把这些文件连接在一起。本文件中对该符号的引用会被连接到实际的定义上去。

符号表首项

前面提到过，符号表的第一项，即索引值为 *STN_UNDEF*(0)的这项，其内容与其它项不同，见下表。

图 1-18 符号表首项

名字	值	意义
<i>st_name</i>	0	无效名字
<i>st_value</i>	0	0 值
<i>st_size</i>	0	无效大小
<i>st_info</i>	0	无效类型

st_other	0	
st_shndx	SHN_UNDEF	无对应节

1.8 重定位

重定位(relocation)是把符号引用与符号定义连接在一起的过程。比如，当程序调用一个函数时，将从当前运行的指令跳转到一个新的指令地址去执行。在编写程序的时候，我们只需指明所要调用的函数名（即符号引用），在重定位的过程中，函数名会与实际的函数所在地址（即符号定义）联系起来，使程序知道应该跳转到哪里去。

重定位文件必须知道如何修改其所包含的“节”的内容，在构建可执行文件或共享目标文件的时候，把节中的符号引用换成这些符号在进程空间中的虚拟地址。包含这些转换信息的数据也就是“重定位项(relocation entries)”。

图 1-19 重定位项结构

```
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;
```

r_offset

本数据成员给出重定位所作用的位置。对于重定位文件来说，此值是受重定位作用的存储单元在节中的字节偏移量；对于可执行文件或共享目标文件来说，此值是受重定位作用的存储单元的虚拟地址。

r_info

本数据成员既给出了重定位所作用的符号表索引，也给出了重定位的类型。比如，如果是一个函数的重定位，本数据成员将要持有被调用函数所对应的符号表索引。如果索引值为 `STN_UNDEF`，即未定义索引，那么重定位过程中将使用 0 作为符号值。重定位类型依处理器不同而不同，各种处理器将分别定义自己的类型。如果一种处理器规定自己引用了一个重定位项的类型或者符号表索引，表明这种处理器应用了 `ELF32_R_TYPE` 或 `ELF32_R_SYM` 到其重定位项的 `r_info` 成员。

以下是应用于 `r_info` 的宏定义。

```
#define ELF32_R_SYM(i) ((i)>>8)
#define ELF32_R_TYPE(i) ((unsigned char)(i))
#define ELF32_R_INFO(s,t) (((s)<<8)+(unsigned char)(t))
```

r_addend

本成员指定了一个加数，这个加数用于计算需要重定位的域的值。

`Elf32_Rela` 与 `Elf32_Rel` 在结构上只有一处不同，就是前者有 `r_addend`。`Elf32_Rela` 中是用 `r_addend` 显式地指出加数；而对 `Elf32_Rel` 来说，加数是隐含在被修改的位置里的。`Elf32_Rel` 中加数的形式这里并不定义，它可以依处理器架构的不同而自行决定。在特定处理器上如何实现，可以指定一种固定的格式，也可以不指定格式而依据上下文来解析。

一个“重定位节(relocation section)”需要引用另外两个节：一个是符号表节，一个是被修改节。在重定位节中，节头的 `sh_info` 和 `sh_link` 成员分别指明了引用关系。不同的目标文件中，重定位项的 `r_offset` 成员的含义略有不同。

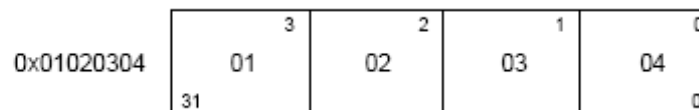
- 在重定位文件中，`r_offset` 成员含有一个节偏移量。也就是说，重定位节本身描述的是如何修改文件中的另一个节的内容，重定位偏移量(`r_offset`)指向了另一个节中的一个存储单元地址。
- 在可执行文件或共享目标文件中，`r_offset` 含有的是符号定义在进程空间中的虚拟地址。可执行文件和共享目标文件是用于运行程序而不是构建程序的，所以对它们来说更有用的信息是运行期的内存虚拟地址，而不是某个符号定义在文件中的位置。

尽管对于不同类型的目标文件，`r_offset` 的含义不同，但其重定位的作用是不变的。

重定位类型(Relocation Types)

重定位项用于描述如何修改如下的指令和数据域。

图 1-20 被重定位域



被重定位域(relocatable field)是一个 32 位的域，占 4 字节并且地址向 4 字节对齐，其字节序与所在体系结构下其他双字长数据的字节序相同。

假定以下的计算是发生在把可重定位文件转为可执行或共享目标文件的过程中。原则上，连接编辑器(link editor)要把一个或多个可重定位文件合并成一个可执行文件或共享目标文件作为输出。它首先需要决定如何把输入文件组合起来，并定位其中的符号，然后更新符号值，最后实现重定位。对于可执行文件或共享目标文件的重定位过程是相似的，不管是可执行文件还是共享目标文件，重定位这个过程的结果是相同的。

为了下面的描述方便，这里定义以下几种运算符号：

A

A 表示用于计算重定位域值的加数。

B

B 表示在程序运行期，共享目标被装入内存时的基地址。一般来说，共享目标文件在构建时基地址为 0，但在运行时则不是。

G

G 表示可重定位项在全局偏移量表中的位置，这里存储了此重定位项在运行期间的地址。更多信息参见下文“全局偏移量表”。

GOT

GOT 表示全局偏移量表的地址。

L

L 表示一个符号的函数连接表项的所在之处，可能是节内偏移量，或者是内存地址。函数连接表项把函数调用定位到合适的位置。在构建期间，连接编辑器创建初始的函数连接表；在运行期间，动态连接器会修改表项。更多信息参见第 2 章的“函数连接表”部分。

P

P 表示被重定位的存储单元在节内的偏移量或者内存地址，由 *r_offset* 计算得到。

S

S 表示重定位项中某个索引值所代表的符号的值。

一个重定位项的 `r_offset` 值指定了被重定位的数据在节内的偏移量或者在进程空间内的虚拟地址。重定位类型指定了哪些位需要被修改以及如何算计它们的值。Intel 架构只使用 `Elf32_Rel` 重定位项，被重定位的域持有重定位加数。在任何情况下，重定位加数和计算结果的字节顺序都要保持一致。

图 1-21 重定位类型(Relocation Types)

名字	值	数据类型	计算
<code>R_386_GOT32</code>	3	word32	$G + A$
<code>R_386_PLT32</code>	4	word32	$L + A - P$
<code>R_386_COPY</code>	5	none	none
<code>R_386_GLOB_DAT</code>	6	word32	S
<code>R_386_JMP_SLOT</code>	7	word32	S
<code>R_386_RELATIVE</code>	8	word32	$B + A$
<code>R_386_GOTOFF</code>	9	word32	$S + A - GOT$
<code>R_386_GOTPC</code>	10	word32	$GOT + A - P$

`R_386_GLOB_DAT`

这种重定位类型用于把指定的符号地址设置为一个全局偏移量表项。这种重定位类型在符号与全局偏移量表项之间建立起了联系。

`R_386_JMP_SLOT`

连接编辑器创建这种重定位类型，用于动态连接。此类型相应的 `offset` 成员给出了函数连接表项的位置。动态连接器修改函数连接表项来跳转到指定的符号地址。

`R_386_RELATIVE`

连接编辑器创建这种重定位类型，主要是用于动态连接。此类型相应的 `offset` 成员给出了共享目标内的一个位置，这个位置含有一个代表相对地址的值。把共享目标被加载的地址加上这个相对地址，动态连接器就可以计算得到真正需要的虚拟地址。这种类型的重定位项必须为符号表指定 0 值。

`R_386_GOTOFF`

这种重定位类型计算符号值与全局偏移量表地址之间的差值。它还指示连接编辑器去构建全局偏移量表。

R_386_GOTPC

这种重定位类型与 `R_386_PC32` 很相似，只不过在计算中它使用的是全局偏移量表的地址。一般来说，在这种类型的重定位中所引用的符号是 `_GLOBAL_OFFSET_TABLE_`，它还指示连接编辑器去构建全局偏移量表。

第2章 ELF文件的装载与动态连接

2.1 介绍

本章描述用于创建程序的目标文件信息和系统行为。可执行文件和共享目标文件（动态链接库）是程序的静态存储形式。要执行一个程序，系统要先把相应的可执行文件和动态链接库装载到进程空间中，这样形成一个可运行的进程的内存空间布局，也可以称它为“进程镜像”。一个已装载完成的进程空间会包含多个不同的“段(segment)”，比如代码段(text segment)，数据段(data segment)，堆栈段(stack segment)等等。

本章首先会描述程序头的格式，并对第一章的内容做一些补充，描述一下与运行程序相关的目标文件结构。程序头可以算是程序结构的一个总纲，它指明了文件中各个段的位置，还包含一些用于创建内存镜像的必要内容。

准备一个程序的内存镜像，可以大体上分为装载和连接两个步骤。前者把目标文件装载入内存，后者解析目标文件中的符号引用。

2.2 程序头

2.2.1 程序头结构

一个可执行文件或共享目标文件的程序头表(program header table)是一个数组，数组中的每一个元素称为“程序头(program header)”，每一个程序头描述了一个“段(segment)”或者一块用于准备执行程序的信息。一个目标文件中的“段(segment)”包含一个或者多个“节(section)”。程序头只对可执行文件或共享目标文件有意义，对于其它类型的目标文件，该信息可以忽略。在目标文件的文件头(elf header)中，`e_phentsize` 和 `e_phnum` 成员指定了程序头的大小。

图 2-1 程序头的结构

```
typedef struct {  
    Elf32_Word p_type;  
    Elf32_Off p_offset;  
    Elf32_Addr p_vaddr;  
    Elf32_Addr p_paddr;  
    Elf32_Word p_filesz;  
    Elf32_Word p_memsz;  
    Elf32_Word p_flags;  
    Elf32_Word p_align;  
} Elf32_Phdr;
```

p_type

此数据成员说明了本程序头所描述的段的类型，或者如何解析本程序头的信息。

图 2-2 段类型(segment types, p_type)

名字	值
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2

PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

PT_NULL

此类型表明本程序头是未使用的，本程序头内的其它成员值均无意义。具有此种类型的程序头应该被忽略。

PT_LOAD

此类型表明本程序头指向一个可装载的段。段的内容会被从文件中拷贝到内存中。如前所述，段在文件中的大小是 `p_filesz`，在内存中的大小是 `p_memsz`。如果 `p_memsz` 大于 `p_filesz`，在内存中多出的存储空间应填 0 补充，也就是说，段在内存中可以比在文件中占用空间更大；而相反，`p_filesz` 永远不应该比 `p_memsz` 大，因为这样的话，内存中就将无法完整地映射段的内容。在程序头表中，所有 `PT_LOAD` 类型的程序头按照 `p_vaddr` 的值做升序排列。

PT_DYNAMIC

此类型表明本段指明了动态连接的信息。

PT_INTERP

本段指向了一个以“null”结尾的字符串，这个字符串是一个 ELF 解析器的路径。这种段类型只对可执行程序有意义，当它出现在共享目标文件中时，是一个无意义的多余项。在一个 ELF 文件中它最多只能出现一次，而且必须出现在其它可装载段的表项之前。

PT_NOTE

本段指向了一个以“null”结尾的字符串，这个字符串包含一些附加的信息。

PT_SHLIB

该段类型是保留的，而且未定义语法。UNIX System V 系统上的应用程序不会包含这种表项。

PT_PHDR

此类型的程序头如果存在的话，它表明的是其自身所在的程序头表在文件或内存中的位置和大小。这样的段在文件中可以不存在，只有当所在程序头表所覆盖的段只是整个程序的一部分时，才会出现一次这种表项，而且这种表项一定出现在其它可装载段的表项之前。

PT_LOPROC ~ PT_HIPROC

类型值在这个区间的程序头是为特定处理器保留的。

除非有特别要求，否则所有程序头的段类型域 *p_type* 都是可选项，不是必须存在的。在所有程序头都不指定段类型的情况下，程序头表中所有的表项都不代表任何特别的类型，而只是作为一种索引，表明其相应的段的大小和位置。

p_offset

此数据成员给出本段内容在文件中的位置，即段内容的开始位置相对于文件开头的偏移量。

p_vaddr

此数据成员给出本段内容的开始位置在进程空间中的虚拟地址。

p_paddr

此数据成员给出本段内容的开始位置在进程空间中的物理地址。对于目前大多数现代操作系统而言，应用程序中段的物理地址事先是不可知的，所以目前这个成员多数情况下保留不用，或者被操作系统改作它用。

p_filesz

此数据成员给出本段内容在文件中的大小，单位是字节，可以是 0。

p_memsz

此数据成员给出本段内容在内容镜像中的大小，单位是字节，可以是 0。

p_flags

此数据成员给出了本段内容的属性。具体有哪些标志位请参见下文。

p_align

对于可装载的段来说，其 `p_vaddr` 和 `p_offset` 的值至少要向内存页面大小对齐。此数据成员指明本段内容如何在内存和文件中对齐。如果该值为 0 或 1，表明没有对齐要求；否则，`p_align` 应该是一个正整数，并且是 2 的幂次数。`p_vaddr` 和 `p_offset` 在对 `p_align` 取模后应该相等。

有些程序头的内容是描述进程内的段，而有的是给出一些辅助信息，并不直接成为段的内容。

2.2.2 基地址

程序头中出现的虚拟地址不能代表其相应的数据在进程内存空间中的虚拟地址。可执行文件中需要含有绝对的地址，比如变量地址，函数地址等，为了让程序正确地执行，“段”中出现的虚拟地址必须在创建可执行程序时被重新计算。另一方面，出于 ELF 通用性的要求，目标文件的段中又不能出现绝对地址，其代码是不应依赖于具体存储位置的，即同一个段在被加载到两个不同的进程中时，它的地址可能不同，但它的行为不能表现出不一样。

在被加载到进程空间里时，尽管“段”会被分配到一个不确定的地址，但是不同的段之间会有确定的“相对位置(relative position)”。也就是说，在目标文件中存储的两个段，它们的位置之间有多少偏移，当它们被加载到内存中时，这两个段的位置之间仍然保持这么大的偏移（距离）。一个段在内存中的虚拟地址与其在目标文件中的地址一般是不相等的，它们之间会有一个偏移量，这个偏移量被称为“基地址(base address)”，基地址的作用之一就是在动态连接过程中为程序重定位内存镜像。

一个可执行文件或共享目标文件的基地址是在运行期间由以下三个值计算出来的：内存加载地址，最大页面大小，程序可装载段的最低地址。为计算基地址，首先找出类型为 `PT_LOAD`（即可加载）而且 `p_vaddr`（段地址）最低的那个段，把这个段在内存中的地址与最大页面大小相除，得到一个段地址的余数；再把 `p_vaddr` 与最大页面大小相除，得到一个 `p_vaddr` 的余数。基地址就是段地址的余数与 `p_vaddr` 的余数之差。

2.2.3 段权限

虽然 ELF 文件格式中没有规定，但是一个可执行程序至少会有一个可加载的段。当为可加载段创建内存镜像时，系统会按照 `p_flags` 的指示给段赋予一定的权限。

名字	值	含义
<code>PF_X</code>	0x1	可执行
<code>PF_W</code>	0x2	只写

PF_R	0x4	只读
PF_MASKPROC	0xf0000000	未指定

被 PF_MASKPROC 所覆盖的权限值是为特殊处理器保留的，处理器可以按照自己的定义来解析它。

如果权限值为 0，表示无任何权限。实际的读写权限还要依赖于内存管理器，在不同的操作系统上，内存管理单元的做法可能会不同。在有些组合方式下，系统给出的权限会比所指定的权限大，但可写权限 PF_W 除外，如果 p_flags 中没有指定 PF_W 的话，系统一定不会给出写权限。下表给出了在一些权限组合的情况下系统实际应赋予的权限。

图 2-3 段权限(Segment Permissions)

标志位	值	所需权限	实际权限
none	0	无任何权限	无任何权限
PF_X	1	可执行	可读，可执行
PF_W	2	可写	可读，可写，可执行
PF_W + PF_X	3	可写，可执行	可读，可写，可执行
PF_R	4	可读	可读，可执行
PF_R + PF_X	5	可读，可执行	可读，可执行
PF_R + PF_W	6	可读，可写	可读，可写，可执行
PF_R + PF_W + PF_X	7	可读，可写，可执行	可读，可写，可执行

对上表总结一下，有两条：

- 可读与可执行是通用的，有其中一个就等于也有了另一个；
- 可写权限是最高权限，可以覆盖另外两个，有了可写权限，所有权限就都有了。

举两个例子，代码段(.text)有可读和可执行的权限，但没有可写权限；数据段一般都有可读和可写的权限，同时也自然有了可执行权限。

2.3 段内容

目标文件中的一个“段”由若干个“节”组成，不过程序头(program header)并不关心“节”的问题，一个段中包含多少个节也与程序装载没有多大关系。

代码段(.text)或直译为“文本段”，包含的是只读的指令和数据，一般情况下会包含以下这些节。不过这里给出的只是一个典型的例子，一个实际的更复杂的代码段可能包含更多的节。

图 2-4 代码段(text segment)

.text
.rodata
.hash
.dynsym
.dynstr
.plt
.rel.got

数据段(data segment)包含可写的数据和指令，典型的数据段包含以下节。

图 2-5 数据段(Data Segment)

.data
.dynamic
.got
.bss

一个 PT_DYNAMIC 类型的程序头项指向一个.dynamic 节。.got 节和.plt 节也含有与地址无关的代码和动态连接相关的信息。在上面的例子中，虽然.plt 节只出现在代码段中，但实际上，它也可以出现在数据段中。

在前面提到过，.bss 节的类型为 SHT_NOBITS，即它在目标文件中不占空间，但它在段中，即在进程空间中却会占有一席之地。一般地，未初始化的全局变量会

存放在.bss 节中，而整个.bss 节会出现在段的最末尾，也正是因为这样，段的内存空间大小(p_memsz)可能会比它在文件中的大小(p_filesz)大一些。

2.4 注释段

类型为 PT_NOTE 的段往往会包含类型为 SHT_NOTE 的节，SHT_NOTE 节可以为目标文件提供一些特别的信息，用于给其它的程序检查目标文件的一致性和兼容性。这些信息我们称为“注释信息”，这样的节称为“注释节(note section)”，所在的段即为“注释段(note segment)”。注释信息可以包含任意数量的“注释项”，每一个注释项是一个数组，数组的每一个成员大小为 4 字节，格式依目标处理器而定。下图解释了注释信息是如何组织的，但这仅是一种参考，不是规范的一部分。

图 2-6 注释信息(note information)

namesz
descsz
type
name
.....
desc
.....

namesz 和 name

Namesz 和 name 成对使用。Namesz 是一个 4 字节整数，而 name 是一个以 'null' 结尾的字符串。Namesz 是 name 字符串的长度。字符串 name 的内容是本项的所有者的名字。没有正式的机制来避免名字冲突，一般按照惯例，系统提供商应把他们自己的名字写进 name 项里，比如“XYZ Computer Company”。如果没有名字的话，namesz 是 0。由于数组项的大小是向 4 字节对齐的，所以如果字符串长度不是整 4 字节的话，需要填 0 补位。如果有补位的话，namesz 只计字符串长度，不计所补的空位。

descsz 和 desc

Descsz 和 desc 也成对使用，它们的格式与 namesz/name 完全相同。不过，desc 的内容没有任何规定、限制，甚至建议，它包含哪些信息完全是自由的。

type

这个字段给出描述项(desc)的解释，或者说是描述项的类型。每一个提供商都会定义自己的类型，所以同一类型值对于不同的提供商其解释也是不同的。当一个程序读取注释信息的时候，它必须同时辨认出 name 和 type 才能理解 desc 的内容。

下图给出一个注释段的示例：

图 2-7 注释段示例

	+0	+1	+2	+3	
namesz	7				No descriptor
descsz	0				
type	1				
name	X	Y	Z		
	C	o	\0	<i>pad</i>	
namesz	7				
descsz	8				
type	3				
name	X	Y	Z		
	C	o	\0	<i>pad</i>	
desc	<i>word 0</i>				
	<i>word 1</i>				

这里解释一下注释节的设计逻辑。注释节中包含三种信息：名字(name/namesz)、类型(type)和描述(desc/descsz)。名字用于区别不同的信息提供者，比如不同的厂商，“ABC Computer Company”或“XYZ Computer Company”。

一个信息提供者可能会提供很多种信息，为了区别，把信息分类，即给每一种信息加一个 ID，或者说是类型 type。比如“ABC Computer Company”公司规定，1 号分类代表这里的信息是一种花的名字，2 号分类代表这里的信息是一种香水的名字，……。

有了 name 和 type 的约束，desc 的内容才有意义，才知道它所描述的是什么。比如，如果没有前面的 name 和 type 信息的话，如果只知道 desc 包含一个字符串“Daisy”，你就不知道这到底是表示雏菊花，还是 Daisy 香水，或者是一个女孩的名字。而如果你知道它的 name 和 type 分别是“ABC Computer Company”和 2 的话，显然“Daisy”说的是香水。

2.5 程序装载

程序装载就是操作系统创建或扩充进程镜像的过程。

进程空间如何构造，内存页面如何管理，以及进程如何被处理，不同的操作系统有不同的作法。

当系统创建或者扩充一个进程镜像时，逻辑上，它要把文件中的段复制成为虚拟内存中的一个段。但是系统不一定立刻真正地去读文件，什么时候读，还要依赖于程序的行为、系统负载等等。进程在加载完成之后，很多文件的内容其实并没有真正地映射到内存中；在运行过程中，进程只有在真正需要去访问一个内存页面的时候，才会去映射它。只在需要的时候才去做物理访问，这样做可以省去很多无用的操作，提高了系统的性能和效率。为了达到这种效果，可执行文件和共享目标文件中段的镜像在文件中的偏移量或者内存虚拟地址必须是向页面大小对齐的。

对于 Intel 架构来说，页面的最大尺寸为 4KB，所以段的虚拟地址和文件内偏移量要向 4KB(0x1000)或者 4KB 的整数倍对齐。这样便于整页的换入换出，可以提高效率。

图 2-8 可执行文件示例

File Offset	File	Virtual Address
0	ELF Header	
	Program Header Table	
	Other Information	
0x100	Text Segment	0x8048100
	...	
	0x2be00 Bytes	0x8073eff
0x2bf00	Data Segment	0x8074f00
	...	
	0x4ee00 Bytes	0x8079cff
0x30d00	Other Information	
	...	

下表描述了这个文件示例。

图 2-9 程序头段

成员	代码	数据
----	----	----

p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x1000	0x1000

尽管示例文件内代码和数据段的偏移量和虚拟地址都是向 4KB 对齐的，但是文件内有 4 个页面包含不纯的代码和数据。

- 第一个代码(text)页包含 ELF 头，程序头表和其它信息。
- 最后一个代码页含有数据段开始处的拷贝。
- 第一个数据(data)页含有代码段结尾处的拷贝。
- 最后一个数据页可能含有与运行过程无关的信息。

逻辑上说，不同的段应该截然分开。根据所在的段不同，不同的页面也应该被分配不同的权限。在这个示例中，同时包含代码段结尾部分和数据段开头部分的这个区域需要被映射到内存两次，一次是映射到代码段，另一次是到数据段，当然，两次的虚拟内存地址是不同的。

当装载进内存后，未初始化的全局变量会被放在紧跟在数据段的后面，系统一般会在这种数据置为 0。所以，如果文件的最后一个数据页中包含一些不在逻辑内存页中的信息，这些附加的信息将无法被保留下来，也会被置为 0。

前述的另外“不纯”的三页中，那些不在进程镜像逻辑组成部分中的信息，系统会不会把它们清空是不一定的，这些多余信息的完整性得不到保障。

这个程序的内存镜像如下图所示。

图 2-10 进程内存镜像示例

Virtual Address	Contents	Segment
0x8048000	Header Padding 0x100 Bytes	Text
0x8048100	Text Segment ... 0x2be00 Bytes	
0x8073f00	Data Padding 0x100 Bytes	
0x8074000	Text Padding 0xf00 Bytes	Data
0x8074f00	Data Segment ... 0x4e00 Bytes	
0x8079d00	Uninitialized Data 0x1024 Zero Bytes	
0x807ad24	Page Padding 0x2dc Zero Bytes	

对于可执行文件和共享目标文件来说，段的装载是不同的。

一方面，在可执行文件中，理所当然地要包含绝对地址。为了让程序能够正确地执行，运行期间的段的虚拟地址必须与构建可执行文件时该段的地址相同，因此系统使用 `p_vaddr` 来记录不变的虚拟地址。

另一方面，在共享目标文件中，理所当然地要使用地址无关的代码，不能依赖于绝对地址。给不同的程序使用时，共享目标文件中的段被加载的地址也不同，但是不能因为地址的不同而改变了执行行为。尽管在两个不同的进程中，系统会给共享目标文件的段选择不同的地址，但是各个段之间的相对地址是不变的。与位置无关的代码正是使用段间的相对位置来互相访问的，在进程的内存中，两段的虚拟地址之差与它们在文件中的位置偏移量之差相等。

下表展示了共享目标在几个不同进程中被指定的虚拟地址，解释了在不同进程中，不同段之间的相对位置是不变的。这张表也解释了基地址的计算。

图 2-11 共享目标段地址示例

来源	代码	数据	基地址
File	0x200	0x2a400	0x0

Process	1	0x80000200	0x8002a400
Process	2	0x80081200	0x800ab400
Process	3	0x900c0200	0x900ea400
Process	4	0x900c6200	0x900f0400

2.6 动态连接

动态连接也就是解析符号引用的过程，这个过程在进程初始化和进程运行期间都可能发生。要想让一种特定的连接模型能够工作起来，需要事先设定好一些基本的机制，为达到这一目的，ELF 文件格式中保留了一些节的用法。实际的连接模型由操作系统来决定，不同操作系统的具体做法会有所不同。

2.6.1 程序解析器

一个参与动态连接的可执行文件会包含一个类型为 `PT_INTERP` 的程序头项。在 UNIX 系统上，当执行一个程序的时候，系统函数 `exec(BA_OS)` 会被调用，在这个函数中，内核会去读取这个 `PT_INTERP` 段，解析出其包含的一个路径字符串，这个串指明了一个 ELF 程序解析器，系统会转去初始化该解析器的进程镜像。也就是，在这时系统会暂停原来的工作，不是用待执行文件的段内容去初始化进程空间，而是把进程空间暂时“借”给解析器程序使用。然后，解析器程序将从系统手中接过控制权继续执行。

解析器以两种方式来接手系统的控制。第一种，解析器取得可执行文件的描述符，内容指针定位于文件开始处，解析器可以读取并映射可执行程序的段到内存中。第二种，对于有些可执行文件格式，系统直接将文件内容载入内存，并不把其文件描述符给解析器。解析器可以是一个共享目标文件，也可以是一个可执行文件。

- 一般情况下，解析器会是一个共享目标文件，并且其段内容是位置不相关的，所以在不同的进程中，它的地址会不一样，系统会使用 `mmap(KE_OS)` 系统调用在动态段区域来为解析器创建段的镜像。所以，一般情况下不用担心解析器的段内容会与待执行文件的内容发生地址冲突。
- 如果解析器是独立的可执行文件，那么系统就要按照解析器程序的程序头来加载它，在加载的时候就有可能与待执行文件的段相冲突，这种情况下由解析来负责解决冲突。

2.6.2 动态连接器

当创建一个可执行文件时，如果依赖其它的动态连接库，那么连接编辑器会在可执行文件的程序头中加入一个 `PT_INTERP` 项，告诉系统这里需要使用动态连接器。

可执行文件与动态连接器一起创建了进程的镜像，这个过程包含以下活动：

- 添加可执行文件的段到进程空间；
- 添加共享目标文件的段到进程空间；

- 为可执行文件和共享目标文件进行重定位；
- 如果动态连接器使用了可执行文件的文件描述符，应关闭它；
- 把控制权交给程序。

连接编辑器也会为动态连接器组织一些数据，以方便它的连接过程。在“程序头”部分提到过，为了方便在运行的时候访问，这些数据放在可装载的段中。当然具体的数据格式是依处理器而不同的。

- 类型为 SHT_DYNAMIC 的 .dynamic 节中包含有很多种动态连接信息。在这个节的最开始处有一个结构，其中包含有其它动态连接信息的地址。
- 类型为 SHT_HASH 的 .hash 节中含有符号哈希表。
- 类型为 SHT_PROGBITS 的 .got 和 .plt 节各包含一张表。Got 是 global offset table 的缩写，即全局偏移量表，用于位置独立的代码；plt 是 procedure linkage table 的缩写，译为函数连接表。下面几节将描述动态连接器是怎么使用和改变这两张表来创建目标文件的内存镜像。

因为每个与 UNIX System V 兼容的程序都会从一个共享目标库中导入系统调用，所以在每一个与 TIS ELF 格式一致的程序执行时，动态连接器都会起作用。

回想“程序装载”部分所描述过的，共享目标所占据的内存地址可能与文件程序头表中所记录的不同。在程序开始执行以前，动态连接器会为内存镜像做重定位，更新绝对地址。当然，库文件在被装载时，如果其内存地址与其文件中描述的完全相同的话，那些引用它们的绝对地址就是对的，不需要更新。但事实上，这种情况很少发生。

如果进程的环境变量中含有 LD_BIND_NOW，而且其值不为空，那么动态连接器就要在程序开始运行之前把所有重定位都处理完。比如，在该环境变量为以下值时，动态连接器都需要这样做：

- LD_BIND_NOW = 1
- LD_BIND_NOW = on
- LD_BIND_NOW = off

否则，如果 LD_BIND_NOW 没有出现或者其值为空，动态连接器就可以把处理重定位的工作推后，即只有当一个符号被引用的时候才去重定位它。因为在程序运行过程中，有一些函数并不会被调用到，推后重定位是一种提高效率的方法，可以避免为这些函数做不必要的重定位。

2.6.3 动态段

如果一个目标文件参与动态连接的话，它的程序头表中一定会包含一个类型为 PT_DYNAMIC 的表项，其所对应的段称为动态段(dynamic segment)，段的名字为.dynamic。动态段的作用是提供动态连接器所需要的信息，比如依赖于哪些共享目标文件、动态连接符号表的位置、动态连接重定位表的位置等等。

这个动态段中包含有动态节，动态节由符号_DYNAMIC 所标记，它包含一个由如下结构体组成的数组。

图 2-12 动态结构(dynamic structure)

```
typedef struct {
    Elf32_Sword d_tag;
    union {
        Elf32_Word d_val;
        Elf32_Addr d_ptr;
    } d_un;
} Elf32_Dyn;

extern Elf32_Dyn _DYNAMIC[];
```

对于每一个这种类型的目标项，d_tag 控制着对 d_un 的解析。

d_val

类型为 Elf32_Word 的目标项代表的是整型数。

d_ptr

类型为 Elf32_Addr 的目标项代表的是进程空间里的地址。前面描述过，目标项在文件中的地址与其在进程空间内的地址可能会不同。当系统解析到这个动态节中的地址时，动态连接器就可以根据文件地址和内存基地址来计算出实际的内存地址。为了保持一致，文件中不会包含重定位项来校正动态结构中的地址。

下表列出了可执行文件或共享目标文件中所要求的标记(d_tag)。如果一个标记被置为“必需”，那么在 ELF 文件的动态连接数组中就必须包含一个此类型的项；如果被置为“可选”，那就不是必需的，可以有也可以没有。

图 2-13 动态数组标记(Dynamic Array Tags, d_tag)

名称	值	d_un	可执行文件	共享目标文件
----	---	------	-------	--------

DT_NULL	0	忽略	必需	必需
DT_NEEDED	1	d_val	可选	可选
DT_PLTRELSZ	2	d_val	可选	可选
DT_PLTGOT	3	d_ptr	可选	可选
DT_HASH	4	d_ptr	必需	必需
DT_STRTAB	5	d_ptr	必需	必需
DT_SYMTAB	6	d_ptr	必需	必需
DT_RELA	7	d_ptr	必需	可选
DT_RELASZ	8	d_val	必需	可选
DT_RELAENT	9	d_val	必需	可选
DT_STRSZ	10	d_val	必需	必需
DT_SYMENT	11	d_val	必需	必需
DT_INIT	12	d_ptr	可选	可选
DT_FINI	13	d_ptr	可选	可选
DT_SONAME	14	d_val	忽略	可选
DT_RPATH	15	d_val	可选	忽略
DT_SYMBOLIC	16	忽略	忽略	可选
DT_REL	17	d_ptr	必需	可选
DT_RELSZ	18	d_val	必需	可选
DT_RELENT	19	d_val	必需	可选
DT_PLTREL	20	d_val	可选	可选
DT_DEBUG	21	d_ptr	可选	忽略

DT_TEXTREL	22	忽略	可选	可选
DT_JMPREL	23	d_ptr	可选	可选
DT_BIND_NOW	24	忽略	可选	可选
DT_LOPROC	0x70000000	未定义	未定义	未定义
DT_HIPROC	0x7fffffff	未定义	未定义	未定义

DT_NULL DT_NULL

用于标记_DYNAMIC 数组的结束。

DT_NEEDED

此元素指明了一个所需的库的名字。不过此元素本身并不是一个字符串，它是一个指向由”DT_STRTAB”所标记的字符串表中的索引，在表中，此索引处是一个以’null’结尾的字符串，这个字符串就是库的名字。在动态数组中可以包含若干个此类型的项，这些项出现的相对顺序是不能随意调换的。

DT_PLTRELSZ

此元素含有与函数连接表相关的所有重定位项的总大小，以字节为单位。如果数组中有 DT_JMPREL 项的话，DT_PLTRELSZ 也必须要有。

DT_PLTGOT

此元素包含与函数连接表或全局偏移量表相应的地址。在 Intel 架构中，这一项的 d_ptr 成员给出全局偏移量表中第一项的地址。如下文所述，全局偏移量表中前三项都是保留的，其中两项用于持有函数连接表信息。

DT_HASH

此元素含有符号哈希表的地址。这里所指的哈希表与 DT_SYMTAB 所指的哈希表是同一个。

DT_STRTAB

此元素包含字符串表的地址，此表中包含符号名、库名等等。

DT_SYMTAB

此元素包含符号表的地址。

DT_RELA

此元素包含一个重定位表的地址，在重定位表中存储的是显式的“加数”，比如对于 32 位文件来说，这种加数就是 `Elf32_Rela`。在一个目标文件中可以存在多个重定位节，当为可执行文件或共享目标文件创建重定位表的时候，连接编辑器会把这些重定位节连接在一起，最后形成一张大的重定位表。当连接编辑器为一个可执行文件创建进程空间，或者把一个共享目标添加到进程空间中去的时候，它会去读重定位表并执行相应的操作。如果在动态结构中包含有 `DT_RELA` 元素的话，就必须同时还包含 `DT_RELASZ` 和 `DT_RELEANT` 元素。如果一个文件需要重定位的话，`DT_RELA` 或 `DT_REL` 至少要出现一个。

DT_RELASZ

此元素持有 `DT_RELA` 相应的重定位表的大小，以字节为单位。

DT_RELAENT

此元素持有 `DT_RELA` 相应的重定位表项的大小，以字节为单位。

DT_STRSZ

此元素持有字符串表的大小，以字节为单位。

DT_SYMENT

此元素持有符号表项的大小，以字节为单位。

DT_INIT

此元素持有初始化函数的地址。参见下文“初始化和终止函数”内容。

DT_FINI

此元素持有终止函数的地址。参见下文“初始化和终止函数”内容。

DT_SONAME

此元素持有一个字符串表中的偏移量，该位置存储了一个以 `'null'` 结尾的字符串，是一个共享目标的名字。相应的字符串表由 `DT_STRTAB` 指定。

DT_RPATH

此元素持有一个字符串表中的偏移量，该位置存储了一个以 `'null'` 结尾的字符串，是一个用于搜索库文件的路径名。相应的字符串表由 `DT_STRTAB` 指定。

DT_SYMBOLIC

在共享目标文件中，此元素的出现与否决定了动态连接器解析符号时所用的算法。如果此元素不出现的话，动态连接器先搜索可执行文件再搜索库文件；如果

此元素出现的话，顺序刚好相反，动态连接器会先从本共享目标文件开始，后搜索可执行文件。

DT_REL

此元素与 *DT_RELA* 相似，只是它所指向的重定位表中，“加数”是隐含的而不是显式的。

DT_RELSZ

此元素持有 *DT_REL* 相应的重定位表的大小，以字节为单位。

DT_RELENT

此元素持有 *DT_REL* 相应的重定位表项的大小，以字节为单位。

DT_PLTREL

本成员指明了函数连接表所引用的重定位项的类型。*d_val* 成员含有 *DT_REL* 或 *DT_RELA*。函数连接表中的所有重定位类型都是相同的。

DT_DEBUG

本成员用于调试，格式未明确定义。

DT_TEXTREL

如果此元素出现的话，在重定位过程中如果需要修改的是只读段的话，连接编辑器可以做相应的修改；而如果此元素不出现的话，在重定位过程中，即使需要，也不能修改只读段。

DT_JMPREL

此类型元素如果存在的话，其 *d_ptr* 成员含有与函数连接表单独关联的重定位项地址。把多个重定位项分开可以让动态连接器在初始化的时候忽略它们，当然前提条件是“后期绑定”是激活的。如果此元素存在的话，*DT_PLTRELSZ* 和 *DT_PLTREL* 也应该出现。

DT_BIND_NOW

如果此元素存在的话，动态连接器必须在程序开始执行以前，完成所有包含此项的目标的重定位工作。如果此元素存在，即使程序应用了“后期绑定”，它对于此项所指定的目标也不适用，动态连接器仍需事先做好重定位。

DT_LOPROC ~ DT_HIPROC

这一区间的值是为处理器保留的。

以上各种类型的元素中，除了 `DT_NULL` 必须出现在数组的结尾，`DT_NEEDED` 有相对顺序要求以外，其它所有元素的出现顺序都没有限制。

在上表中没有出现的标记值都是保留未用的。

2.6.4 共享目标的依赖关系

当连接编辑器处理一个存档库的时候，它提取库的成员并把它们拷贝到输出文件中。可执行文件在创建的时候可以采用静态连接的方式，在运行的时候不依赖于外部共享库，也不需要动态连接器。而如果采用了动态连接的方式，动态连接器就需要起作用了，它必须把要用到的共享目标文件也载入到进程空间里去。

当动态连接器为一个目标文件创建内存段的时候，动态结构中的 `DT_NEEDED` 项会指明所依赖的库，动态连接器会连接被引用的符号和它们所依赖的库，这个过程会反复地执行，直到一个完整的进程镜像被构建好。当解析一个符号引用的时候，动态连接器以一种“广度优先”的算法来查找符号表。就是说，动态连接器首先查找可执行程序自己的符号表，然后是 `DT_NEEDED` 项所指明的库的符号表，再接下来是下一层依赖库的符号表，依次下去。共享目标文件必须是可读的，其它权限没有要求。

即使一个共享目标在依赖关系中被引用多次，动态连接器也只会连接它一次。

在依赖关系列表中的名字，即可以是 `DT_SONAME` 字符串，也可以是用于创建目标文件的共享目标文件的路径名。比如，在创建一个可执行文件的时候，连接编辑器使用了 `DT_SONAME` 项的共享目标 `lib1`，以及另一个名为 `"/usr/lib/lib2"` 的共享目标库，那么可执行文件就会包含 `lib1` 和 `/usr/lib/lib2` 在其依赖列表中。

如果一个共享目标名字中含有斜线（`/`）字符，比如 `"/usr/lib/lib2"` 或者 `"directory/file"`，动态连接就直接把字符串作为路径名。如果名字中没有斜线，比如 `"lib1"`，需要根据以下三种规则来查找库文件：

- 第一，动态数组标记 `DT_RPATH` 可能给出了一个含有一系列目录名的字符串，各目录名以冒号“`:`”相隔。比如，如果字符串是 `"/home/dir/lib:/home/dir2/lib:"`，表明动态连接器的查找路径依次是 `"/home/dir/lib"`、`"/home/dir2/lib"` 和当前目录。
- 第二，进程的环境变量中会有一个 `LD_LIBRARY_PATH` 变量，它也含有一个目录名列表，各目录名以冒号“`:`”相隔，各目录名列表以分号“`;`”相隔。以下几种写法的查找路径顺序相同：

- `LD_LIBRARY_PATH=/home/dir/lib:/home/dir2/lib:`
- `LD_LIBRARY_PATH=/home/dir/lib;/home/dir2/lib:`

- `LD_LIBRARY_PATH=/home/dir/lib:/home/dir2/lib;`

`LD_LIBRARY_PATH` 路径的优先级要低于 `DT_RPATH` 所指明的路径。尽管有些程序（比如连接编辑器）对待分号前后的路径有所不同，但动态连接器并不会。不过，动态连接器只接收如上所示的分号作为路径分隔符。

- 第三，如果如上两组路径都无法找到所要的库，动态连接器就搜索 `/usr/lib`。

2.6.5 全局偏移量表

全局偏移量表(global offset table)在私有数据中包含绝对地址。出于方便共享和重用的考虑，目标文件中的很多内容是“位置无关”的，其映射到进程内存中的什么位置是不一定的，所以只适合使用相对地址，全局偏移量表是一个例外。在 UNIX System V 环境下的动态连接过程中，`got` 表是必须的，它的实际内容和格式依处理器不同而不同。

总的来说，位置独立的代码不能含有绝对的虚拟地址。全局偏移量表选择了在私有数据中含有绝对地址，这种办法在没有牺牲位置独立性和可共享性的前提下保存了绝对地址。引用全局偏移量表的程序可以同时使用位置独立的地址和绝对地址，把位置无关的引用重定向到绝对地址上去。

一开始，全局偏移量表只包含其重定位项所要求的信息。当系统为可装载的目标文件创建了内存段之后，动态连接器处理重定位项，有些重定位项的类型为 `R_386_GLOB_DAT`，它们指向全局偏移量表。动态连接器决定相应的符号值，计算其绝对地址，并且为内存段设置适当的值。尽管在连接编辑器创建目标文件的时候绝对地址还是未知的，但是动态连接器却知道所有内存段的地址，因此它可以计算所含有的所有符号的绝对地址。

如果一个程序要求直接访问符号的绝对地址，那么这个符号在全局偏移量表中就必须有一个对应的项。可执行文件和共享目标文件有各自的全局偏移量表，所以一个符号的地址可能会出现在多个表中。动态连接器会在程序开始执行之前，处理好所有全局偏移量表的重定位工作，所以在程序执行的时候，可以保证所有这些符号都有正确的绝对地址。

全局偏移量表的第 0 项是保留的，它用于持有动态结构的地址，由符号 `_DYNAMIC` 引用。这样，其它程序，比如动态连接器就可以直接找到其动态结构，而不用借助重定位项。这对于动态连接器来说尤为重要，因为它必须在不依赖于其它程序重定位其内存镜像的情况下初始化自己。前面提到过，在 Intel 架构中，全局偏移量表中的第 1 项和第 2 项也是保留的，它们持有函数连接表的信息。

系统可能为同一个共享目标在不同的程序中选择不同的段地址；甚至也可能每次为同一个程序选择不同的地址。但是，在单次执行中，一旦一个进程的镜像建立起来之后，直到程序退出，内存段的地址都不会再改变了。

全局偏移量表的格式和解析方法是依处理器而不同的，就 Intel 架构而言，符号 `_GLOBAL_OFFSET_TABLE_` 会被用来访问此表。

图 2-14 全局偏移量表

```
extern Elf32_Addr _GLOBAL_OFFSET_TABLE_[];
```

符号 `_GLOBAL_OFFSET_TABLE_` 可能位于 `.got` 节中部，所以它也接受负的数组索引值。

2.6.6 函数地址

在可执行文件和共享目标文件中，当引用到同一个函数时，函数地址可能并不相同。在共享目标文件中，函数的地址被动态连接器正常地解析为它所在的虚拟地址。但在可执行文件中则不同，但可执行文件引用一个共享库中的函数时，它不是直接指向函数的虚拟地址，而是被动态连接器定向到函数连接表中的一个表项。

但是，这样的话，来自可执行文件的函数地址和来自共享目标文件的同一函数地址就会不同，为了避免在比较两个函数地址时出现这样的逻辑错误，连接编辑器和动态连接器做了一些特别操作。当可执行文件引用一个在共享目标文件中定义的函数时，连接编辑器就把这个函数的函数连接表项的地址放到其相应的符号表项中去。动态连接器会特别对待这种符号表项。在可执行文件中，如果动态连接器查找一个符号时遇到了这种符号表项，就会按照以下规则行事：

1. 如果符号表项的 `st_shndx` 成员不是 `SHN_UNDEF`，动态连接器就找到了一个符号的定义，把表项的 `st_value` 成员作为符号的地址。
2. 如果符号表项的 `st_shndx` 成员是 `SHN_UNDEF`，并且符号类型是 `STT_FUNC`，`st_value` 成员又非 0 的话，动态连接器就认定这是一个特殊的项，把 `st_value` 成员作为符号的地址。
3. 否则，动态连接器认为这个符号是在可执行文件中未定义的。

有些重定位与函数连接表项有关，这些表项用于给函数调用做定向，而不是引用函数地址。这种重定位不能像上面所描述的那样，用特别的方式去处理函数地址，因为动态连接器不可以把函数连接表项重定向到它们自己。

2.6.7 函数连接表

全局偏移量表用于把位置独立的地址重定向到绝对地址，与此功能类似，函数连接表(procedure linkage table)的作用是把位置独立的函数调用重定向到绝对地址。连接编辑器不能解析函数在不同目标文件之间的跳转，那么，它就把对其它目标文件中函数的调用重定向到一个函数连接表项中去。在 Intel 架构中，函数连接表位于共享代码段中，但它们使用全局偏移量表中的私有地址。动态连接器决定目标的绝对地址，并且会相应地修改全局偏移量表的内存镜像。这样，动态连接器就可以在不牺牲位置无关性和代码的可共享性条件下，实现到绝对地址的重定位。可执行文件和共享目标文件有各自的函数连接表。

图 2-15 绝对地址的函数连接表(Absolute Procedure Linkage Table)

```
.PLT0:          pushl  got_plus_4
                jmp     *got_plus_8
                nop; nop
                nop; nop

.PLT1:          jmp     *name1_in_GOT
                pushl  $offset
                jmp     .PLT0@PC

.PLT2:          jmp     *name2_in_GOT
                pushl  $offset
                jmp     .PLT0@PC

.....
```

图 2-16 地址无关的函数连接表(Position-Independent Procedure Linkage Table)

```
.PLT0:          pushl  4(%ebx)
                jmp     *8(%ebx)
                nop; nop
                nop; nop

.PLT1:          jmp     *name1@GOT(%ebx)
                pushl  $offset
```

```

                                jmp    .PLT0@PC
.PLT2:                        jmp    *name2@GOT(%ebx)
                                pushl  $offset
                                jmp    .PLT0@PC
.....

```

比较上面两图可知，在绝对地址代码和位置无关代码中，函数连接表中的指令使用的操作数寻址方式不同。但是它们给动态连接器的接口都是相同的。

2.6.8 解析符号

在以下的这些步骤中，动态连接器与程序合作来解析函数连接表和全局偏移量表中所有的符号引用。

1. 在一开始创建程序内存镜像的时候，动态连接器把全局偏移量表中的第 2 和第 3 个表项设为特定值。下面的步骤中会解析所设置的值。
2. 如果函数连接表是位置独立的，全局偏移量表的地址必须存储在 `%ebx` 中。进程空间中的每一个共享目标文件都有自己的函数连接表，每一个表都是用于本文件内的函数调用。那么，主调函数就要负责在调用函数连接表表项之前设置全局偏移量表。
3. 这里做个示例，假设程序要调用函数 `name1`，与之相应的函数连接表是 `.PLT1`。
4. 第一条指令跳转到 `name1` 所在全局偏移量表项中的地址。一开始，全局偏移量表中持有的是“`pushl`”指令的地址，而不是“`name1`”的地址。
5. 接下来，程序把一个重定位偏移量压入堆栈。重定位偏移量是一个 32 位的非负数，它指向重定位表内的一项。被指定的重定位项类型为 `R_386_JMP_SLOT`，它的“`offset`”指定了前一个“`jmp`”指令所用到的一个全局偏移表项。重定位项也包含一个符号表索引，为动态连接器指明了正被引用的是什么符号，在本例是即“`name1`”。
6. 把重定位偏移量压栈以后，程序就跳到 `.PLT0`，即函数连接表的第一项。“`pushl`”指令把全局偏移量表的第 2 项(`got_plus_4` or `4(%ebx)`)压入栈顶，因此给了动态连接器一个字的确认信息。接下来，程序跳转到全局偏移量表的第 3 项(`got_plus_8` or `8(%ebx)`)中的地址，控制权就又交给了动态连接器。

7. 动态连接器接过控制权后，它弹出栈顶数据，查找指定的重定位项，找到符号的值，把“name1”真正的地址存储在全局偏移量表项中，并把控制权传给指定的目标。

8. 接下来，函数连接表会把控制权直接转到 name1，不需要动态连接器再次介入。也就是说，.PLT1 处的 jmp 指令会跳转到 name1。

环境变量 LD_BIND_NOW 可以改变动态连接器的行为，如果它的值为非“null”，动态连接器在传递控制权给程序之前会估计函数连接表项。也就是说，动态连接器在进程初始化的过程中会处理类型为 R_386_JMP_SLOT 的重定位项。否则，如果其值为“null”，这种估计仍然会进行，但并不是在初始化的时候，这个过程会被推后，直到在执行过程中，该函数连接表项被用到才开始。

“后期绑定/懒绑定” (lazy binding) 一般来说都会提高应用程序的性能，因为这样可以避免用不到的符号在动态连接过程中被解析。但是，在两种情况下，后期绑定的效果并不理想。第一种情况，如果对一个共享目标函数的第一次引用比其后的引用要花更多时间的话，在第一次引用时，程序就要暂停下来，由动态连接器去解析符号，如果应用程序对这种不可预知的暂停比较敏感的话，后期绑定就不适用。第二种情况，如果动态连接器解析一个符号失败，程序将会被终止。如果没有打开后期绑定的话，这一切都发生在程序实际得到控制权之前，进程将在初始化过程中被终止。而如果打开了后期绑定的话，错误会发生在程序运行过程中，如果应用程序对这种不可预知的错误敏感的话，后期绑定也不适用。

2.7 哈希表

一个 Elf32_Word 目标组成的哈希表支持符号表的访问。下面的图解释了哈希表的组织，但它不是规范的一部分。

图 2-17 符号哈希表(Symbol Hash Table)

nbucket
nchain
bucket[0]
. . .
bucket[nbucket-1]
chain[0]
. . .
chain[nchain-1]

Bucket 数组中含有 nbucket 个项，chain 数组中含有 nchain 个项，序号都从 0 开始。Bucket 和 chain 中包含的都是符号表中的索引。符号表中的项数必须等于 nchain，所以符号表中的索引号也可以用来索引 chain 表。如下所示的一个哈希函数输入一个符号名，输出一个值用于计算 bucket 索引。如果给出一个符号名，经哈希函数计算得到值 x，那么 $x \% \text{nbucket}$ 是 bucket 表内的索引，`bucket[x%nbucket]` 给出一个符号表的索引值 y，y 同时也是 chain 表内的索引值。如果符号表内索引值为 y 的元素并不是所要的，那么 `chain[y]` 给出符号表中下一个哈希值相同的项的索引。如果所有哈希值相同的项都不是所要的，最后的一个 `chain[y]` 将包含值 STN_UNDEF，说明这个符号表中并不含有此符号。

图 2-18 哈希函数(hashing function)

```
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;  
}
```

2.8 初始化和终止函数

当动态连接器构建好进程镜像，并完成重定位后，每一个共享目标都有机会执行一些初始化代码。所有共享目标的初始化都发生在程序开始执行前。

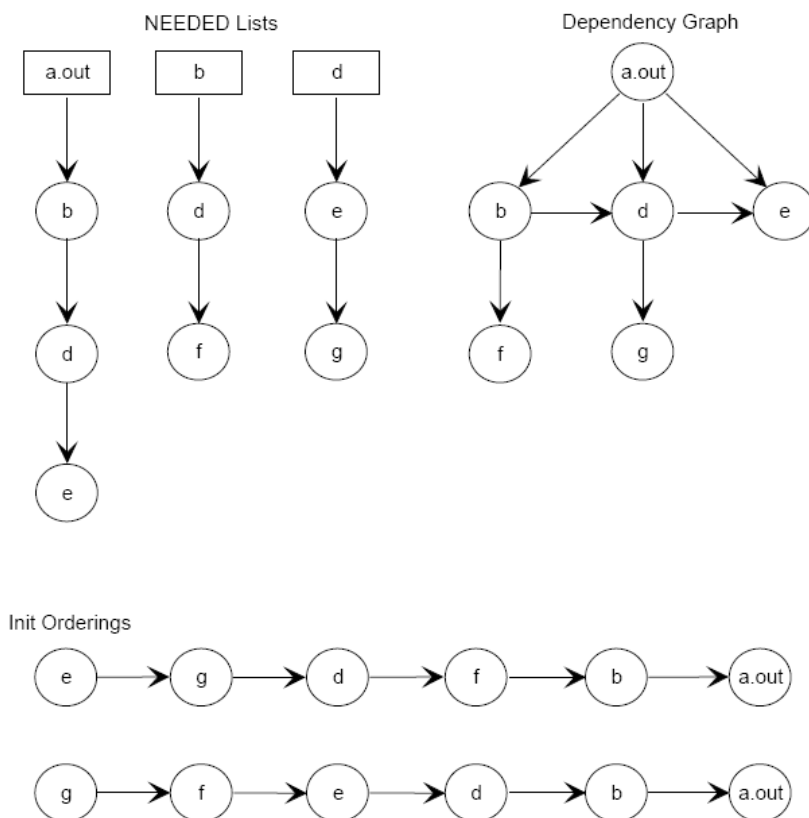
一个目标的初始化代码被执行以前，必须保证它所依赖的所有目标已经被初始化过。这里所说的“依赖”，即出现在动态结构的 `DT_NEEDED` 项里。如果两个目标，它们互相依赖，或者彼此间的依赖关系构成环状的话，哪个应该被先初始化，这里未作定义。

如果一个目标 **A** 依赖于另外一个目标 **B**，而目标 **B** 又依赖于目标 **C** 的话。当需要对 **A** 做初始化时，应先递规地初始化 **B** 和 **C**，即先初始化 **C**，然后是 **B**，最后是 **A**。

如果一个目标 **A** 依赖于另处两个目标 **B** 和 **C**，而 **B** 和 **C** 之间没有依赖关系的话，**B** 和 **C** 谁先被初始化都可以。

下面给出一个初始化顺序的例子。目标文件 `a.out` 依赖于 `b`，`d` 和 `e`，`b` 依赖于 `d` 和 `f`，而 `d` 又依赖于 `e` 和 `g`。图中左上部给出各个目标各自的依赖关系，由此可以画出右上角的依赖关系网。没有出发箭头的结点应先初始化；有出发箭头的结点必须在所有箭头的目标结点都已完成初始化之后才能初始化。根据这两条规则，图下方给出两条可行的初始化路径。当然，不只这两条，还有更多的选择。

图 2-19 初始化顺序示例



与初始化过程相似，每一个共享目标还可以有终止函数，将在进程准备终止的时候被调用。动态连接器调用终止函数的顺序正好与初始化过程相反，如果一个目标没有定义初始化函数的话，动态连接器应假设它有一个空的初始化函数并且被调用，并按照相反的顺序来调用其终止函数。

动态连接器必须保证，无论是初始化函数还是终止函数都不能被重复调用。

共享目标把初始化和终止函数分别定义在动态结构的 `DT_INIT` 和 `DT_FINI` 项中，初始化和终止函数的代码存放在 `.init` 和 `.fini` 节中。

动态连接器只负责共享目标文件中的初始化和终止化，并不负责可执行文件。即动态连接器不会去调用可执行文件的 `.init` 节的代码，也不会用 `atexit(BA_OS)` 注册 `.fini` 节的代码。用户通过 `atexit(BA_OS)` 指定的终止函数必须在共享目标的终止函数之前被调用。

2.9 程序解析器

在 Intel 架构下，系统中自带一种符合 ELF 规范的程序解析器：
`/usr/lib/libc.so.1`。

第3章 示例程序

本章以一个简单的示例程序来解释前两章所描述的内容。

这个示例包括以下源文件：

- **main.c**: 包含一个主函数，主函数中有对其它文件中函数和全局变量的引用；
- **part.c**: 其中定义了两个函数，以及全局变量；
- **part.h**: 头文件；
- **Makefile**: 用于构建可执行文件。

为了尽可能多地演示不同类型 ELF 文件的格式，分别以静态连接和动态连接两种方式构建可执行文件。以下是输出文件的列表：

- **main.o/part.o**: 相应源文件编译而成的目标文件；
- **libpart.so**: **part.o** 动态连接而成的共享目标文件；
- **main_s**: 静态连接方式构建的可执行文件；
- **main_d**: 动态连接方式构建的可执行文件。

本章所用示例的源程序及其相关的输出文件分别在附录 A 和 B 中列出。

3.1 ELF文件头

本节以 **part.o** 为例，演示 ELF 文件头。

使用以下命令可以打印出 ELF 文件的结构信息，全部输出内容请见附录部分图 B-2。

```
readelf -a part.o
```

我们从这条命令打印出的诸多信息中截取出 ELF 文件头部分，如下图所示：

图 3-1 **part.o** 文件头信息

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF32
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  REL (Relocatable file)
  Machine:                               Intel 80386
  Version:                               0x1
  Entry point address:                   0x0
  Start of program headers:              0 (bytes into file)
  Start of section headers:              284 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   52 (bytes)
  Size of program headers:               0 (bytes)
  Number of program headers:              0
  Size of section headers:               40 (bytes)
```

Number of section headers:	12
Section header string table index:	9

下面我们截取 part.o 文件的开始部分，逐条解析以下信息的字节码真实面目。

图 3-2 part.o 文件头字节码

87654321	0011	2233	4455	6677	8899	aabb	ccdd	eeff	0123456789abcdef
00000000:	7f45	4c46	0101	0100	0000	0000	0000	0000	.ELF.....
00000010:	0100	0300	0100	0000	0000	0000	0000	0000
00000020:	1c01	0000	0000	0000	3400	0000	0000	28004.....(.
00000030:	0c00	0900						

下面逐一解释各字节内容，以[xxxx]的格式来表示地址，地址值都是 16 进制数，但为了表达简洁，这里把地址的前缀“0x”省略掉。

[00 ~ 0f]

这 16 字节为 e_ident 信息。

[00 ~ 03]

这 4 个字节为“魔数”，即‘0x7f’, ‘E’, ‘L’, ‘F’4 个字符。

[04]

文件类型，这里是 1，即“ELFCLASS32”，此文件是 32 位目标文件。

[05]

文件编码格式，这里是 1，即“ELFDATA2LSB”，LSB 编码，即小头编码。

[06]

文件版本，是 1。

[07 ~ 0f]

这一段字节无意义，被 0 填充。

[10 ~ 11]

e_type，即 ELF 文件的种类。这里要注意一下，e_type 为 Elf_32_Half 类型，占两字节，虽然字节表示为“0100”，但因为本文件是 LSB 编码，即低字节在前，所以逻辑上实际的值为“0001”，即“ET_REL”，是可重定位文件。

[12 ~ 13]

e_machine，即处理器体系结构。这里是 3，即 EM_386，Intel 架构。

[14 ~ 17]

e_version, 这里是 1。

[18 ~ 1b]

e_entry, 程序入口的虚拟地址, 这里是 0, 因为想文件并非可执行文件。

[1c ~ 1f]

e_phoff, 程序头表在文件中的偏移量, 本文件没有程序头表, 所以为 0。


[20 ~ 23]

e_shoff, 节头表在文件中的偏移量, 这里为"0x011c"。

[24 ~ 27]

e_flags, 属性标志, 对于 Intel 架构, 这 4 个字节为 0。

[28 ~ 29]

e_ehsize, ELF 文件头大小, 共 0x34 个字节, 据此, 我们把上图中的 ELF 部分用  颜色标出。

[2a ~ 2b]

e_phentsize, 程序头表中表项的大小。由于本文件中没有程序头表, 所以为 0。

[2c ~ 2d]

e_phnum, 程序头表中的项数。由于本文件中没有程序头表, 所以为 0。

[2e ~ 2f]

e_shentsize, 节头表中表项的大小。这里为 0x28。

[30 ~ 31]

e_shnum, 节头表中的表项数。这里为 0xc, 即 12 项。

[32 ~ 33]

e_shstrndx, 节头字符串表索引, 此处为 9。

3.2 节头表

从上面 ELF 文件头信息得知, part.o 文件中存在一个节头表, 其起始位置在 0x011c (e_shoff) 处, 节头表中总共有 0x0c (e_shnum) 项, 每一项大小为 0x28 (e_shentsize) 字节, 这也与 Elf32_Ehdr 结构的定义相符, 每个表项中包含 10 个

数据，每个占 4 字节。由此可算出节头表的总大小为 $0x0c * 0x28 = 0x01e0$ ，即节头表的位置区域为[011c ~ 02fb]。下面把这一段拷贝出来详细分析。

图 3-3 节头表内容

87654321	0011	2233	4455	6677	8899	aabb	ccdd	eeff	0123456789abcdef
00000110:							0000	0000
00000120:	0000	0000	0000	0000	0000	0000	0000	0000
00000130:	0000	0000	0000	0000	0000	0000	0000	0000
00000140:	0000	0000	1f00	0000	0100	0000	0600	0000
00000150:	0000	0000	3400	0000	3c00	0000	0000	00004...<.....
00000160:	0000	0000	0400	0000	0000	0000	1b00	0000
00000170:	0900	0000	0000	0000	0000	0000	2804	0000(...
00000180:	2000	0000	0a00	0000	0100	0000	0400	0000
00000190:	0800	0000	2900	0000	0100	0000	0300	0000).....
000001a0:	0000	0000	7000	0000	0800	0000	0000	0000p.....
000001b0:	0000	0000	0400	0000	0000	0000	2500	0000%...
000001c0:	0900	0000	0000	0000	0000	0000	4804	0000H...
000001d0:	0800	0000	0a00	0000	0300	0000	0400	0000
000001e0:	0800	0000	2f00	0000	0800	0000	0300	0000/.
000001f0:	0000	0000	7800	0000	0000	0000	0000	0000x.....
00000200:	0000	0000	0400	0000	0000	0000	3400	00004...
00000210:	0100	0000	0200	0000	0000	0000	7800	0000x...
00000220:	2800	0000	0000	0000	0000	0000	0100	0000	(.....
00000230:	0000	0000	3c00	0000	0100	0000	3000	0000<.....0...
00000240:	0000	0000	a000	0000	2400	0000	0000	0000\$.
00000250:	0000	0000	0100	0000	0100	0000	4500	0000E...
00000260:	0100	0000	0000	0000	0000	0000	c400	0000
00000270:	0000	0000	0000	0000	0000	0000	0100	0000
00000280:	0000	0000	1100	0000	0300	0000	0000	0000
00000290:	0000	0000	c400	0000	5500	0000	0000	0000U.....
000002a0:	0000	0000	0100	0000	0000	0000	0100	0000
000002b0:	0200	0000	0000	0000	0000	0000	fc02	0000
000002c0:	f000	0000	0b00	0000	0800	0000	0400	0000
000002d0:	1000	0000	0900	0000	0300	0000	0000	0000
000002e0:	0000	0000	ec03	0000	3900	0000	0000	00009.....
000002f0:	0000	0000	0100	0000	0000	0000		

这 12 个表项中都包含哪些节，ELF 结构信息中列出了如下内容：

图 3-4 节头表信息

Section Headers:										
[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00000000	000034	00003c	00	AX	0	0	4
[2]	.rel.text	REL	00000000	000428	000020	08		10	1	4
[3]	.data	PROGBITS	00000000	000070	000008	00	WA	0	0	4
[4]	.rel.data	REL	00000000	000448	000008	08		10	3	4
[5]	.bss	NOBITS	00000000	000078	000000	00	WA	0	0	4
[6]	.rodata	PROGBITS	00000000	000078	000028	00	A	0	0	1
[7]	.comment	PROGBITS	00000000	0000a0	000024	01	MS	0	0	1
[8]	.note.GNU-stack	PROGBITS	00000000	0000c4	000000	00		0	0	1
[9]	.shstrtab	STRTAB	00000000	0000c4	000055	00		0	0	1
[10]	.symtab	SYMTAB	00000000	0002fc	0000f0	10		11	8	4
[11]	.strtab	STRTAB	00000000	0003ec	000039	00		0	0	1

Key to Flags:


W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

O (extra OS processing required) o (OS specific), p (processor specific)

这里以 4 个典型的节为例做下解释。

3.3 节头字符串表

此表项提供了节头字符串表(.shstrtab)的信息，该表存储了所有节的名字。其位置在[0284 ~ 02ab]。在图 3-3 中以  标出。

[0284 ~ 0287]

sh_name，值为 0x11。本节的名字，此值是节头字符串表中的索引。

[0288 ~ 028b]

sh_type，值为 0x03，即 SHT_STRTAB，表明此表项指向一个字符串表。

对于字符串表，我们只关心此表的起始地址和长度。

[0294 ~ 0297]





sh_offset，本节在文件中的偏移量。值是 0x0c4，这里是字符串表的起始位置。


[0298 ~ 029b]

sh_size，本节的大小，即字符串表的长度，值为 0x55。


根据 sh_offset 和 sh_size 这两个值，我们查一下这个节头字符串表的内容。

图 3-5 节头字符串表

87654321	0011	2233	4455	6677	8899	aabb	ccdd	eeff	0123456789abcdef
000000c0:		002e	7379	6d74	6162	002e	7374		..symtab..st
000000d0:	7274	6162	002e	7368	7374	7274	6162	002e	rtab.. .shstrtab.
000000e0:	7265	6c2e	7465	7874	002e	7265	6c2e	6461	rel.text..rel.da
000000f0:	7461	002e	6273	7300	2e72	6f64	6174	6100	ta..bss..rodata.
00000100:	2e63	6f6d	6d65	6e74	002e	6e6f	7465	2e47	.comment..note.G
00000110:	4e55	2d73	7461	636b	0000				NU-stack.. 

现在反查一下本节的名字 sh_name (0x11)，这里存储的字符串正是“.shstrtab”，在上图中由  标出。

3.4 字符串表

此表项提供了字符串节(.strtab)的信息。其位置在[02d4 ~ 02fb]。在图 3-3 中以  标出。

[02d4 ~ 02d7]

sh_name，值为 0x09。本节的名字，此值是节头字符串表中的索引。查上图，此处的字符串为“.strtab”。

[02d8 ~ 02db]

sh_type, 值为 0x03, 即 SHT_STRTAB, 表明此表项指向一个字符串表。

对于字符串表, 我们只关心此表的起始地址和长度。

[02e4 ~ 02e7]

sh_addr, 本节内容映射到进程空间的起始地址, 因为还没有重定位, 所以此值为 0。

[02e8 ~ 02eb]

sh_offset, 本节在文件中的偏移量。值是 0x03ec, 这里是字符串表的起始位置。

[02ec ~ 02ef]

sh_size, 本节的大小, 即字符串表的长度, 值为 0x39。

看一下这个字符串, 我们会发现我们定义的全局变量和函数的名字都在这里。但是 printf() 函数调用中用到的字符串却没有发现, 它们被定义在 .rodata 一节, 因为它们都是字符串常量。

图 3-6 .rodata 节内容

87654321	0011	2233	4455	6677	8899	aabb	ccdd	eeff	0123456789abcdef
000003e0:								0070 6172	.par
000003f0:	742e	6300	675f	696e	7431	0067	5f69	6e74	t.c.g_int1.g_int
00000400:	3200	675f	7374	7231	0067	5f73	7472	3200	2.g_str1.g_str2.
00000410:	6675	6e63	5f31	0070	7269	6e74	6600	6675	func_1.printf.fu
00000420:	6e63	5f32	0000						nc 2..

3.5 代码节

此表项提供了代码节(.text)的信息。位置是[0144 ~ 016b]。在图 3-3 中以 标出。

对于代码段, 我们只关心此表的起始地址和长度。

[0150 ~ 0153]

sh_addr, 本节内容映射到进程空间的起始地址, 因为还没有重定位, 所以此值为 0。

[0154 ~ 0157]

sh_offset，本节在文件中的偏移量。值是 0x034，这里是字符串表的起始位置。

[0158 ~ 015b]

sh_size，本节的大小，即字符串表的长度，值为 0x3c。

下面我们把[034 ~ 06f]这段字节码内容提取出来看一下。

图 3-7 代码段内容

87654321	0011	2233	4455	6677	8899	aabb	ccdd	eeff	0123456789abcdef
00000030:			5589	e583	ec28	8b45	0889	45f4	U....(.E..E.
00000040:	8b45	f489	4424	04c7	0424	0400	0000	e8fc	.E..D\$...\$.....
00000050:	ffff	ffc9	c355	89e5	83ec	188b	4508	8944U.....E..D
00000060:	2404	c704	2415	0000	00e8	fcff	ffff	c9c3	\$...\$.....

这是一段机器码。通过以下命令：

```
objdump -d part.o
```

把 part.o 反汇编一下即可了解到这一段的内容。

图 3-8 part.o 反汇编内容

00000000	<func_1>:		
0:	55	push	%ebp
1:	89 e5	mov	%esp,%ebp
3:	83 ec 28	sub	\$0x28,%esp
6:	8b 45 08	mov	0x8(%ebp),%eax
9:	89 45 f4	mov	%eax,-0xc(%ebp)
c:	8b 45 f4	mov	-0xc(%ebp),%eax
f:	89 44 24 04	mov	%eax,0x4(%esp)
13:	c7 04 24 04 00 00 00	movl	\$0x4, (%esp)
1a:	e8 fc ff ff ff	call	1b <func_1+0x1b>
1f:	c9	leave	
20:	c3	ret	
00000021	<func_2>:		
21:	55	push	%ebp
22:	89 e5	mov	%esp,%ebp
24:	83 ec 18	sub	\$0x18,%esp
27:	8b 45 08	mov	0x8(%ebp),%eax
2a:	89 44 24 04	mov	%eax,0x4(%esp)
2e:	c7 04 24 15 00 00 00	movl	\$0x15, (%esp)
35:	e8 fc ff ff ff	call	36 <func_2+0x15>
3a:	c9	leave	
3b:	c3	ret	

可见，part.o 中有两个函数的代码，它们的位置是相连的，而且总长度加在一起正是 0x3c。而且如果细心地去把每一个机器码与目标文件中的[034 ~ 06f]这段字节相比较的话，发现它们是完全相同的。.text 节中正是存储了这两个函数的机器码。

3.6 符号表

下面再来看一下符号表(.symtab)。符号表.symtab 的表头在节头表中的位置是 [02ac ~ 02d3]。在图 3-3 中以 标出。

[02b0 ~ 02b3]

sh_type 值为 2，即 SHT_SYMTAB，表明此表项指向一个符号表。

[02bc ~ 02bf]

sh_offset，即符号表在文件中的偏移量，值为 0x02fc。

[02c0 ~ 02c2]

sh_size，符号表的大小，值为 0x0f0。

[02d0 ~ 02d3]

sh_entsize，值为 0x010。即符号表中每一个表项的大小为 0x010。

下面就根据上述的位置和大小信息来查看一下符号表。

图 3-9 符号表字节码

87654321	0011	2233	4455	6677	8899	aabb	ccdd	eeff	0123456789abcdef
000002f0:							0000	0000
00000300:	0000	0000	0000	0000	0000	0000	0100	0000
00000310:	0000	0000	0000	0000	0400	f1ff	0000	0000
00000320:	0000	0000	0000	0000	0300	0100	0000	0000
00000330:	0000	0000	0000	0000	0300	0300	0000	0000
00000340:	0000	0000	0000	0000	0300	0500	0000	0000
00000350:	0000	0000	0000	0000	0300	0600	0000	0000
00000360:	0000	0000	0000	0000	0300	0800	0000	0000
00000370:	0000	0000	0000	0000	0300	0700	0800	0000
00000380:	0400	0000	0400	0000	1100	f2ff	0f00	0000
00000390:	0000	0000	0400	0000	1100	0300	1600	0000
000003a0:	0400	0000	0400	0000	1100	f2ff	1d00	0000
000003b0:	0400	0000	0400	0000	1100	0300	2400	0000\$...
000003c0:	0000	0000	2100	0000	1200	0100	2b00	0000!.....+...
000003d0:	0000	0000	0000	0000	1000	0000	3200	00002...
000003e0:	2100	0000	1b00	0000	1200	0100			!.....

符号表总大小为 0x0f0，每一个表项的大小为 0x010，所以符号表中总共包含了 15 个表项。结合第一章中符号表项的结构，很容易分析出各个表项的内容是什么。这里不再对照每一个字节数据去分析，直接给出由 readelf 给出的分析结果：

图 3-10 符号表内容解析

Symbol table '.symtab' contains 15 entries:						
Num:	Value	Size	Type	Bind	Vis	Ndx Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND
1:	00000000	0	FILE	LOCAL	DEFAULT	ABS part.c
2:	00000000	0	SECTION	LOCAL	DEFAULT	1
3:	00000000	0	SECTION	LOCAL	DEFAULT	3

4:	00000000	0	SECTION	LOCAL	DEFAULT	5	
5:	00000000	0	SECTION	LOCAL	DEFAULT	6	
6:	00000000	0	SECTION	LOCAL	DEFAULT	8	
7:	00000000	0	SECTION	LOCAL	DEFAULT	7	
8:	00000004	4	OBJECT	GLOBAL	DEFAULT	COM	g_int1
9:	00000000	4	OBJECT	GLOBAL	DEFAULT	3	g_int2
10:	00000004	4	OBJECT	GLOBAL	DEFAULT	COM	g_str1
11:	00000004	4	OBJECT	GLOBAL	DEFAULT	3	g_str2
12:	00000000	33	FUNC	GLOBAL	DEFAULT	1	func_1
13:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf
14:	00000021	27	FUNC	GLOBAL	DEFAULT	1	func_2

这张表中，第 8~14 行中包含了我们在程序中定义的全局变量和函数，数据可能有些费解，这里做个解释。

先看 Ndx 这一列，它所表示的是此符号与哪一个节相关联，这里可以理解成它定义在哪一节中。g_int2 和 g_str2 是初始化过的全局变量，所以它们出现在 .data 节中，回看图 3-4，.data 节的序号正是 3。而 g_int1 和 g_str1，它们也是全局变量，但是没有初始化，所以它们并不属于 .data，在逻辑上它们是属于 .bss，而 .bss 节在本目标文件中并没有被分配空间，所以根据第一章对 st_shndx 的描述，它们的 st_shndx 值都为 SHN_COMMON。func_1 和 func_2 是函数，它们当然应该属于 .text 节，即 st_shndx 为 1。

再来看 Value 这一列。g_int1 和 g_str1 因为 st_shndx 值都为 SHN_COMMON，所以它们的 st_value 值为字节对齐数。而 g_int2 和 g_str2 的 value 值是它们在所定义的节中，即 .data 节中的偏移量。.data 节中只有这两个数据，所以它们的偏移量分别为 0 和 4。func_1 和 func_2 是函数，它们在 .text 节中的起始位置分别为 0x00 和 0x021，这在前面已经看过。

Size 数据更容易理解，整型数和指针的大小均为 4 字节，而函数的大小为其机器码的长度。

对于 printf 这个符号，它是库中的函数，定义并不在本目标文件中。它是一个待重定位的符号，所以它的位置和大小信息都是 0。

3.7 段

“段”存在于经过连接的目标文件中。所以，在可执行文件和共享目标文件中都可以找到段的踪迹。如第一章所述，一个段由若干个节组成。下面看一下 libpart.so 和 main_d 中的段，仍然看 readelf 的输出。

图 3-11 libpart.so 程序头表

Program Headers:							
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x00000000	0x00000000	0x005f0	0x005f0	R E	0x1000
LOAD	0x000f04	0x00001f04	0x00001f04	0x00110	0x00120	RW	0x1000
DYNAMIC	0x000f18	0x00001f18	0x00001f18	0x000d0	0x000d0	RW	0x4
NOTE	0x0000f4	0x000000f4	0x000000f4	0x00024	0x00024	R	0x4
GNU STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4

GNU_RELRO	0x000f04	0x00001f04	0x00001f04	0x000fc	0x000fc	R	0x1
Section to Segment mapping:							
Segment Sections...							
00	.note.gnu.build-						
id	.hash	.gnu.hash	.dynsym	.dynstr	.gnu.version	.gnu.version_r	.rel.dyn .r
el	.plt	.init	.plt	.text	.fini	.rodata	.eh_frame
01	.ctors .dtors .jcr .dynamic .got .got.plt .data .bss						
02	.dynamic						
03	.note.gnu.build-id						
04							
05	.ctors .dtors .jcr .dynamic .got						

图 3-12 main_d 程序头表

Program Headers:							
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00100	0x00100	R E	0x4
INTERP	0x000134	0x08048134	0x08048134	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x0076c	0x0076c	R E	0x1000
LOAD	0x000f04	0x08049f04	0x08049f04	0x00118	0x00130	RW	0x1000
DYNAMIC	0x000f18	0x08049f18	0x08049f18	0x000d8	0x000d8	RW	0x4
NOTE	0x000148	0x08048148	0x08048148	0x00044	0x00044	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4
GNU_RELRO	0x000f04	0x08049f04	0x08049f04	0x000fc	0x000fc	R	0x1
Section to Segment mapping:							
Segment Sections...							
00							
01	.interp						
02	.interp .note.ABI-tag .note.gnu.build-						
id	.hash	.gnu.hash	.dynsym	.dynstr	.gnu.version	.gnu.version_r	.rel.dyn .r
el	.plt	.init	.plt	.text	.fini	.rodata	.eh_frame
03	.ctors .dtors .jcr .dynamic .got .got.plt .data .bss						
04	.dynamic						
05	.note.ABI-tag .note.gnu.build-id						
06							
07	.ctors .dtors .jcr .dynamic .got						

比较上面两图可以看出，在两个文件中各有两个类型为“LOAD”的可装载段，这两个段的内容都比较多，包含有较多的节。其中含有.text节的段即是我们常说的“代码段”，另一个含有.data的段就是“数据段”。

有一个不同的地方，在可执行文件中，有“INTERP”段，而共享目标文件中没有。在前文中讲过，这一段中存储了ELF解析器的路径名，也就是动态连接器。由于动态连接器只有在程序准备运行的时候才起作用，所以只有可执行文件含有这个段，在其它目标文件中就不需要了。

3.8 动态节

再来看另一个重要的内容，动态节。动态节也就是专门用于动态连接过程的节，所以它只会出现在经过连接的目标文件中。在本例中，*.o文件中不会有动态节，而在libpart.so和main_s及main_d中都有。

动态节是用于在不同的目标文件之间连接符号用的。在本示例中，libpart.so 在逻辑上是“导出”符号的一方，而 main_d 是“导入”符号的一方。

这里我们分别看一下它们的动态符号表，即“.dynsym”节。

图 3-13 libpart.so 的动态符号表

Symbol table '.dynsym' contains 16 entries:							
Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
2:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__Jv_RegisterClasses
3:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@GLIBC_2.0 (2)
4:	00000000	0	FUNC	WEAK	DEFAULT	UND	
__cxa_finalize@GLIBC_2.1.3 (3)							
5:	0000201c	4	OBJECT	GLOBAL	DEFAULT	23	g_str1
6:	00002020	4	OBJECT	GLOBAL	DEFAULT	23	g_int1
7:	00002024	0	NOTYPE	GLOBAL	DEFAULT	ABS	_end
8:	00002014	0	NOTYPE	GLOBAL	DEFAULT	ABS	_edata
9:	0000052c	33	FUNC	GLOBAL	DEFAULT	12	func_1
10:	0000200c	4	OBJECT	GLOBAL	DEFAULT	22	g_int2
11:	00002014	0	NOTYPE	GLOBAL	DEFAULT	ABS	_bss_start
12:	0000054d	27	FUNC	GLOBAL	DEFAULT	12	func_2
13:	00002010	4	OBJECT	GLOBAL	DEFAULT	22	g_str2
14:	0000040c	0	FUNC	GLOBAL	DEFAULT	10	_init
15:	000005a8	0	FUNC	GLOBAL	DEFAULT	13	_fini

图 3-14 main_d 的动态符号表

Symbol table '.dynsym' contains 17 entries:							
Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
2:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__Jv_RegisterClasses
3:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	
__libc_start_main@GLIBC_2.0 (2)							
4:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@GLIBC_2.0 (2)
5:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	func_2
6:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	func_1
7:	0804a01c	4	OBJECT	GLOBAL	DEFAULT	25	g_str1
8:	0804a020	4	OBJECT	GLOBAL	DEFAULT	25	g_int1
9:	0804a034	0	NOTYPE	GLOBAL	DEFAULT	ABS	_end
10:	0804a01c	0	NOTYPE	GLOBAL	DEFAULT	ABS	_edata
11:	0804871c	4	OBJECT	GLOBAL	DEFAULT	16	_IO_stdin_used
12:	0804a024	4	OBJECT	GLOBAL	DEFAULT	25	g_int2
13:	0804a01c	0	NOTYPE	GLOBAL	DEFAULT	ABS	_bss_start
14:	0804a028	4	OBJECT	GLOBAL	DEFAULT	25	g_str2
15:	0804848c	0	FUNC	GLOBAL	DEFAULT	12	_init
16:	080486fc	0	FUNC	GLOBAL	DEFAULT	15	_fini

main_d 中，func_1 和 func_2 两个符号的 Ndx 均为 UND，并且值为 0，表明它们尚未被重定位，在这里还是未知的符号。由于 main_d 是通过动态连接构建出来的，它的运行依赖于 libpart.so，所以，对于 func_1 和 func_2 的重定位将在运行期间发生。

相应地，在 libpart.so 中，func_1 和 func_2 这两个符号也出现在 .dynsym 节中，因为它们是被导出的符号。在这里，func_1 和 func_2 的 Ndx 和 Value 值都是有意

义的值，而且非零，说明它们正是定义在本目标文件里。而在 `part.c` 里调用的 `printf` 函数，因为是其它库里的，所以对于 `libpart.so` 来说，`printf` 也是导入符号，它的值也为 0，在这里是未定义的符号。

附录A 源文件清单

图 A-1 main.c 源代码

```
#include "part.h"

extern int g_int1, g_int2;
extern char *g_str1, *g_str2;

int main() {
    int i = 0;
    char *str = "abc";

    func_1(i);
    func_2(str);

    g_int1 = 9;
    g_str1 = "defg";
    printf("Global integer is %d and %d, global string is %s and %s.\n", g_int1, g_int2,
g_str1, g_str2);

    return i;
}
```

图 A-2 part.c 源代码

```
#include "part.h"

int g_int1, g_int2 = 5;
char *g_str1, *g_str2 = "xyz";

void func_1(int i) {
    int j;

    j = i;
    printf("func_1 : j = %d\n", j);
}

void func_2(char *str) {
    printf("func_2 : str = %s\n", str);
}
```

图 A-3 part.h 源代码

```
#ifndef _PART_H_
#define _PART_H_
```

```
void func_1(int i);  
void func_2(char *str);  
  
#endif
```

图 A-4 Makefile 内容

```
all: main_s main_d  
  
main_s: main.o part.o  
        gcc main.o part.o -o main_s  
  
main_d: main.o libpart.so  
        gcc main.o -L. -lpart -o main_d  
  
libpart.so: part.o  
        gcc --shared part.o -o libpart.so  
  
main.o: main.c  
        gcc -c main.c -o main.o  
  
part.o: part.c  
        gcc -c part.c -o part.o  
  
clean:  
        rm -f *.o *.so main_s main_d
```

附录B 输出文件清单

图 B-1 main.o 结构信息

```

ELF Header:
Magic:    7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:    ELF32
Data:     2's complement, little endian
Version:  1 (current)
OS/ABI:   UNIX - System V
ABI Version: 0
Type:     REL (Relocatable file)
Machine:  Intel 80386
Version:  0x1
Entry point address: 0x0
Start of program headers: 0 (bytes into file)
Start of section headers: 376 (bytes into file)
Flags:    0x0
Size of this header: 52 (bytes)
Size of program headers: 0 (bytes)
Number of program headers: 0
Size of section headers: 40 (bytes)
Number of section headers: 11
Section header string table index: 8

Section Headers:
[Nr] Name                               Type            Addr            Off             Size             ES Flg Lk  Inf Al
[ 0]                               NULL            00000000         000000          000000           00   0  0  0  0
[ 1] .text                             PROGBITS         00000000         000034          000085           00  AX  0  0  4
[ 2] .rel.text                         REL              00000000         000470          000060           08   9  1  4
[ 3] .data                             PROGBITS         00000000         0000bc          000000           00  WA  0  0  4
[ 4] .bss                              NOBITS           00000000         0000bc          000000           00  WA  0  0  4
[ 5] .rodata                           PROGBITS         00000000         0000bc          000046           00   A  0  0  4
[ 6] .comment                          PROGBITS         00000000         000102          000024           01  MS  0  0  1
[ 7] .note.GNU-stack                   PROGBITS         00000000         000126          000000           00   0  0  1
[ 8] .shstrtab                         STRTAB           00000000         000126          000051           00   0  0  1
[ 9] .symtab                           SYMTAB           00000000         000330          000100           10  10  8  4
[10] .strtab                           STRTAB           00000000         000430          00003e           00   0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

There are no program headers in this file.

Relocation section '.rel.text' at offset 0x470 contains 12 entries:
Offset      Info      Type            Sym.Value      Sym. Name
00000016    00000501  R_386_32        00000000       .rodata
00000022    00000902  R_386_PC32      00000000       func_1
0000002e    00000a02  R_386_PC32      00000000       func_2
00000034    00000b01  R_386_32        00000000       g_int1
0000003e    00000c01  R_386_32        00000000       g_str1
00000042    00000501  R_386_32        00000000       .rodata
00000048    00000d01  R_386_32        00000000       g_str2
0000004e    00000c01  R_386_32        00000000       g_str1
00000054    00000e01  R_386_32        00000000       g_int2
00000059    00000b01  R_386_32        00000000       g_int1
00000070    00000501  R_386_32        00000000       .rodata
00000075    00000f02  R_386_PC32      00000000       printf

```

There are no unwind sections in this file.

Symbol table '.symtab' contains 16 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	FILE	LOCAL	DEFAULT	ABS	main.c
2:	00000000	0	SECTION	LOCAL	DEFAULT	1	
3:	00000000	0	SECTION	LOCAL	DEFAULT	3	
4:	00000000	0	SECTION	LOCAL	DEFAULT	4	
5:	00000000	0	SECTION	LOCAL	DEFAULT	5	
6:	00000000	0	SECTION	LOCAL	DEFAULT	7	
7:	00000000	0	SECTION	LOCAL	DEFAULT	6	
8:	00000000	133	FUNC	GLOBAL	DEFAULT	1	main
9:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	func_1
10:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	func_2
11:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	g_int1
12:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	g_str1
13:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	g_str2
14:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	g_int2
15:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf

No version information found in this file.

图 B-2 part.o 结构信息

ELF Header:

Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
 Class: ELF32
 Data: 2's complement, little endian
 Version: 1 (current)
 OS/ABI: UNIX - System V
 ABI Version: 0
 Type: REL (Relocatable file)
 Machine: Intel 80386
 Version: 0x1
 Entry point address: 0x0
 Start of program headers: 0 (bytes into file)
 Start of section headers: 284 (bytes into file)
 Flags: 0x0
 Size of this header: 52 (bytes)
 Size of program headers: 0 (bytes)
 Number of program headers: 0
 Size of section headers: 40 (bytes)
 Number of section headers: 12
 Section header string table index: 9

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00000000	000034	00003c	00	AX	0	0	4
[2]	.rel.text	REL	00000000	000428	000020	08		10	1	4
[3]	.data	PROGBITS	00000000	000070	000008	00	WA	0	0	4
[4]	.rel.data	REL	00000000	000448	000008	08		10	3	4
[5]	.bss	NOBITS	00000000	000078	000000	00	WA	0	0	4
[6]	.rodata	PROGBITS	00000000	000078	000028	00	A	0	0	1
[7]	.comment	PROGBITS	00000000	0000a0	000024	01	MS	0	0	1
[8]	.note.GNU-stack	PROGBITS	00000000	0000c4	000000	00		0	0	1
[9]	.shstrtab	STRTAB	00000000	0000c4	000055	00		0	0	1
[10]	.symtab	SYMTAB	00000000	0002fc	0000f0	10		11	8	4
[11]	.strtab	STRTAB	00000000	0003ec	000039	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

```

I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

There are no program headers in this file.

Relocation section '.rel.text' at offset 0x428 contains 4 entries:
  Offset      Info      Type           Sym.Value    Sym. Name
00000016  00000501 R_386_32      00000000     .rodata
0000001b  00000d02 R_386_PC32    00000000     printf
00000031  00000501 R_386_32      00000000     .rodata
00000036  00000d02 R_386_PC32    00000000     printf

Relocation section '.rel.data' at offset 0x448 contains 1 entries:
  Offset      Info      Type           Sym.Value    Sym. Name
00000004  00000501 R_386_32      00000000     .rodata

There are no unwind sections in this file.

Symbol table '.symtab' contains 15 entries:
  Num:      Value      Size Type      Bind   Vis      Ndx Name
   0: 00000000      0 NOTYPE   LOCAL  DEFAULT  UND
   1: 00000000      0 FILE     LOCAL  DEFAULT  ABS part.c
   2: 00000000      0 SECTION  LOCAL  DEFAULT    1
   3: 00000000      0 SECTION  LOCAL  DEFAULT    3
   4: 00000000      0 SECTION  LOCAL  DEFAULT    5
   5: 00000000      0 SECTION  LOCAL  DEFAULT    6
   6: 00000000      0 SECTION  LOCAL  DEFAULT    8
   7: 00000000      0 SECTION  LOCAL  DEFAULT    7
   8: 00000004      4 OBJECT   GLOBAL  DEFAULT   COM g_int1
   9: 00000000      4 OBJECT   GLOBAL  DEFAULT    3 g_int2
  10: 00000004      4 OBJECT   GLOBAL  DEFAULT   COM g_str1
  11: 00000004      4 OBJECT   GLOBAL  DEFAULT    3 g_str2
  12: 00000000     33 FUNC     GLOBAL  DEFAULT    1 func_1
  13: 00000000      0 NOTYPE   GLOBAL  DEFAULT   UND printf
  14: 00000021     27 FUNC     GLOBAL  DEFAULT    1 func_2

No version information found in this file.

```

图 B-3 libpart.so 结构信息

```

ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                   ELF32
  Data:                     2's complement, little endian
  Version:                  1 (current)
  OS/ABI:                   UNIX - System V
  ABI Version:              0
  Type:                     DYN (Shared object file)
  Machine:                  Intel 80386
  Version:                  0x1
  Entry point address:      0x470
  Start of program headers: 52 (bytes into file)
  Start of section headers: 4404 (bytes into file)
  Flags:                    0x0
  Size of this header:      52 (bytes)
  Size of program headers:  32 (bytes)
  Number of program headers: 6
  Size of section headers:  40 (bytes)
  Number of section headers: 28
  Section header string table index: 25

```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.note.gnu.build-id	NOTE	000000f4	0000f4	000024	00	A	0	0	4
[2]	.hash	HASH	00000118	000118	000054	04	A	4	0	4
[3]	.gnu.hash	GNU HASH	0000016c	00016c	000050	04	A	4	0	4
[4]	.dynsym	DYNSYM	000001bc	0001bc	000100	10	A	5	1	4
[5]	.dynstr	STRTAB	000002bc	0002bc	0000a8	00	A	0	0	1
[6]	.gnu.version	VERSYM	00000364	000364	000020	02	A	4	0	2
[7]	.gnu.version_r	VERNEED	00000384	000384	000030	00	A	5	1	4
[8]	.rel.dyn	REL	000003b4	0003b4	000048	08	A	4	0	4
[9]	.rel.plt	REL	000003fc	0003fc	000010	08	A	4	11	4
[10]	.init	PROGBITS	0000040c	00040c	000030	00	AX	0	0	4
[11]	.plt	PROGBITS	0000043c	00043c	000030	04	AX	0	0	4
[12]	.text	PROGBITS	00000470	000470	000138	00	AX	0	0	16
[13]	.fini	PROGBITS	000005a8	0005a8	00001c	00	AX	0	0	4
[14]	.rodata	PROGBITS	000005c4	0005c4	000028	00	A	0	0	1
[15]	.eh_frame	PROGBITS	000005ec	0005ec	000004	00	A	0	0	4
[16]	.ctors	PROGBITS	00001f04	000f04	000008	00	WA	0	0	4
[17]	.dtors	PROGBITS	00001f0c	000f0c	000008	00	WA	0	0	4
[18]	.jcr	PROGBITS	00001f14	000f14	000004	00	WA	0	0	4
[19]	.dynamic	DYNAMIC	00001f18	000f18	0000d0	08	WA	5	0	4
[20]	.got	PROGBITS	00001fe8	000fe8	00000c	04	WA	0	0	4
[21]	.got.plt	PROGBITS	00001ff4	000ff4	000014	04	WA	0	0	4
[22]	.data	PROGBITS	00002008	001008	00000c	00	WA	0	0	4
[23]	.bss	NOBITS	00002014	001014	000010	00	WA	0	0	4
[24]	.comment	PROGBITS	00000000	001014	000046	01	MS	0	0	1
[25]	.shstrtab	STRTAB	00000000	00105a	0000d8	00		0	0	1
[26]	.symtab	SYMTAB	00000000	001594	0003b0	10		27	44	4
[27]	.strtab	STRTAB	00000000	001944	0001a7	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), x (unknown)
 O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x00000000	0x00000000	0x005f0	0x005f0	R E	0x1000
LOAD	0x000f04	0x00001f04	0x00001f04	0x00110	0x00120	RW	0x1000
DYNAMIC	0x000f18	0x00001f18	0x00001f18	0x000d0	0x000d0	RW	0x4
NOTE	0x0000f4	0x000000f4	0x000000f4	0x00024	0x00024	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4
GNU_RELRO	0x000f04	0x00001f04	0x00001f04	0x000fc	0x000fc	R	0x1

Section to Segment mapping:

Segment Sections...

00	.note.gnu.build-id .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
01	.ctors .dtors .jcr .dynamic .got .got.plt .data .bss
02	.dynamic
03	.note.gnu.build-id
04	
05	.ctors .dtors .jcr .dynamic .got

Dynamic section at offset 0xf18 contains 22 entries:

Tag	Type	Name/Value
0x00000001	(NEEDED)	Shared library: [libc.so.6]
0x0000000c	(INIT)	0x40c
0x0000000d	(FINI)	0x5a8

0x00000004	(HASH)	0x118
0x6ffffef5	(GNU_HASH)	0x16c
0x00000005	(STRTAB)	0x2bc
0x00000006	(SYMTAB)	0x1bc
0x0000000a	(STRSZ)	168 (bytes)
0x0000000b	(SYMENT)	16 (bytes)
0x00000003	(PLTGOT)	0x1ff4
0x00000002	(PLTRELSZ)	16 (bytes)
0x00000014	(PLTREL)	REL
0x00000017	(JMPREL)	0x3fc
0x00000011	(REL)	0x3b4
0x00000012	(RELSZ)	72 (bytes)
0x00000013	(RELENT)	8 (bytes)
0x00000016	(TEXTREL)	0x0
0x6ffffffe	(VERNEED)	0x384
0x6fffffff	(VERNEEDNUM)	1
0x6fffffff0	(VERSYM)	0x364
0x6ffffffa	(RELCOUNT)	4
0x00000000	(NULL)	0x0

Relocation section '.rel.dyn' at offset 0x3b4 contains 9 entries:

Offset	Info	Type	Sym.Value	Sym. Name
00000542	00000008	R_386_RELATIVE		
0000055d	00000008	R_386_RELATIVE		
00002008	00000008	R_386_RELATIVE		
00002010	00000008	R_386_RELATIVE		
00000547	00000302	R_386_PC32	00000000	printf
00000562	00000302	R_386_PC32	00000000	printf
00001fe8	00000106	R_386_GLOB_DAT	00000000	__gmon_start__
00001fec	00000206	R_386_GLOB_DAT	00000000	__Jv_RegisterClasses
00001ff0	00000406	R_386_GLOB_DAT	00000000	__cxa_finalize

Relocation section '.rel.plt' at offset 0x3fc contains 2 entries:

Offset	Info	Type	Sym.Value	Sym. Name
00002000	00000107	R_386_JUMP_SLOT	00000000	__gmon_start__
00002004	00000407	R_386_JUMP_SLOT	00000000	__cxa_finalize

There are no unwind sections in this file.

Symbol table '.dynsym' contains 16 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
2:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__Jv_RegisterClasses
3:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@GLIBC_2.0 (2)
4:	00000000	0	FUNC	WEAK	DEFAULT	UND	__cxa_finalize@GLIBC_2.1.3
(3)							
5:	0000201c	4	OBJECT	GLOBAL	DEFAULT	23	g_str1
6:	00002020	4	OBJECT	GLOBAL	DEFAULT	23	g_int1
7:	00002024	0	NOTYPE	GLOBAL	DEFAULT	ABS	_end
8:	00002014	0	NOTYPE	GLOBAL	DEFAULT	ABS	_edata
9:	0000052c	33	FUNC	GLOBAL	DEFAULT	12	func_1
10:	0000200c	4	OBJECT	GLOBAL	DEFAULT	22	g_int2
11:	00002014	0	NOTYPE	GLOBAL	DEFAULT	ABS	_bss_start
12:	0000054d	27	FUNC	GLOBAL	DEFAULT	12	func_2
13:	00002010	4	OBJECT	GLOBAL	DEFAULT	22	g_str2
14:	0000040c	0	FUNC	GLOBAL	DEFAULT	10	_init
15:	000005a8	0	FUNC	GLOBAL	DEFAULT	13	_fini

Symbol table '.symtab' contains 59 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	

1:	000000f4	0	SECTION	LOCAL	DEFAULT	1	
2:	00000118	0	SECTION	LOCAL	DEFAULT	2	
3:	0000016c	0	SECTION	LOCAL	DEFAULT	3	
4:	000001bc	0	SECTION	LOCAL	DEFAULT	4	
5:	000002bc	0	SECTION	LOCAL	DEFAULT	5	
6:	00000364	0	SECTION	LOCAL	DEFAULT	6	
7:	00000384	0	SECTION	LOCAL	DEFAULT	7	
8:	000003b4	0	SECTION	LOCAL	DEFAULT	8	
9:	000003fc	0	SECTION	LOCAL	DEFAULT	9	
10:	0000040c	0	SECTION	LOCAL	DEFAULT	10	
11:	0000043c	0	SECTION	LOCAL	DEFAULT	11	
12:	00000470	0	SECTION	LOCAL	DEFAULT	12	
13:	000005a8	0	SECTION	LOCAL	DEFAULT	13	
14:	000005c4	0	SECTION	LOCAL	DEFAULT	14	
15:	000005ec	0	SECTION	LOCAL	DEFAULT	15	
16:	00001f04	0	SECTION	LOCAL	DEFAULT	16	
17:	00001f0c	0	SECTION	LOCAL	DEFAULT	17	
18:	00001f14	0	SECTION	LOCAL	DEFAULT	18	
19:	00001f18	0	SECTION	LOCAL	DEFAULT	19	
20:	00001fe8	0	SECTION	LOCAL	DEFAULT	20	
21:	00001ff4	0	SECTION	LOCAL	DEFAULT	21	
22:	00002008	0	SECTION	LOCAL	DEFAULT	22	
23:	00002014	0	SECTION	LOCAL	DEFAULT	23	
24:	00000000	0	SECTION	LOCAL	DEFAULT	24	
25:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
26:	00001f04	0	OBJECT	LOCAL	DEFAULT	16	__CTOR_LIST__
27:	00001f0c	0	OBJECT	LOCAL	DEFAULT	17	__DTOR_LIST__
28:	00001f14	0	OBJECT	LOCAL	DEFAULT	18	__JCR_LIST__
29:	00000470	0	FUNC	LOCAL	DEFAULT	12	__do_global_dtors_aux
30:	00002014	1	OBJECT	LOCAL	DEFAULT	23	completed.6990
31:	00002018	4	OBJECT	LOCAL	DEFAULT	23	dtor_idx.6992
32:	000004f0	0	FUNC	LOCAL	DEFAULT	12	frame_dummy
33:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
34:	00001f08	0	OBJECT	LOCAL	DEFAULT	16	__CTOR_END__
35:	000005ec	0	OBJECT	LOCAL	DEFAULT	15	__FRAME_END__
36:	00001f14	0	OBJECT	LOCAL	DEFAULT	18	__JCR_END__
37:	00000570	0	FUNC	LOCAL	DEFAULT	12	__do_global_ctors_aux
38:	00000000	0	FILE	LOCAL	DEFAULT	ABS	part.c
39:	00001ff4	0	OBJECT	LOCAL	HIDDEN	ABS	__GLOBAL_OFFSET_TABLE__
40:	00002008	0	OBJECT	LOCAL	HIDDEN	22	__dso_handle
41:	00001f10	0	OBJECT	LOCAL	HIDDEN	17	__DTOR_END__
42:	00000527	0	FUNC	LOCAL	HIDDEN	12	__i686.get_pc_thunk.bx
43:	00001f18	0	OBJECT	LOCAL	HIDDEN	ABS	__DYNAMIC
44:	0000201c	4	OBJECT	GLOBAL	DEFAULT	23	g_str1
45:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
46:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__Jv_RegisterClasses
47:	000005a8	0	FUNC	GLOBAL	DEFAULT	13	__fini
48:	00002020	4	OBJECT	GLOBAL	DEFAULT	23	g_int1
49:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@@GLIBC_2.0
50:	0000200c	4	OBJECT	GLOBAL	DEFAULT	22	g_int2
51:	00002014	0	NOTYPE	GLOBAL	DEFAULT	ABS	__bss_start
52:	0000054d	27	FUNC	GLOBAL	DEFAULT	12	func_2
53:	00002024	0	NOTYPE	GLOBAL	DEFAULT	ABS	__end
54:	00002010	4	OBJECT	GLOBAL	DEFAULT	22	g_str2
55:	00002014	0	NOTYPE	GLOBAL	DEFAULT	ABS	edata
56:	0000052c	33	FUNC	GLOBAL	DEFAULT	12	func_1
57:	00000000	0	FUNC	WEAK	DEFAULT	UND	__cxa_finalize@@GLIBC_2.1
58:	0000040c	0	FUNC	GLOBAL	DEFAULT	10	__init

Histogram for bucket list length (total of 3 buckets):

Length	Number	% of total	Coverage
0	0	(0.0%)	

1	0	(0.0%)	0.0%
2	0	(0.0%)	0.0%
3	1	(33.3%)	20.0%
4	0	(0.0%)	20.0%
5	0	(0.0%)	20.0%
6	2	(66.7%)	100.0%

Histogram for '.gnu.hash' bucket list length (total of 3 buckets):

Length	Number	% of total	Coverage
0	0	(0.0%)	
1	1	(33.3%)	9.1%
2	0	(0.0%)	9.1%
3	0	(0.0%)	9.1%
4	0	(0.0%)	9.1%
5	2	(66.7%)	100.0%

Version symbols section '.gnu.version' contains 16 entries:
 Addr: 0000000000000364 Offset: 0x000364 Link: 4 (.dynsym)

000:	0 (*local*)	0 (*local*)	0 (*local*)	2 (GLIBC_2.0)
004:	3 (GLIBC_2.1.3)	1 (*global*)	1 (*global*)	1 (*global*)
008:	1 (*global*)	1 (*global*)	1 (*global*)	1 (*global*)
00c:	1 (*global*)	1 (*global*)	1 (*global*)	1 (*global*)

Version needs section '.gnu.version_r' contains 1 entries:
 Addr: 0x0000000000000384 Offset: 0x000384 Link: 5 (.dynstr)

000000: Version: 1 File: libc.so.6 Cnt: 2
 0x0010: Name: GLIBC_2.1.3 Flags: none Version: 3
 0x0020: Name: GLIBC_2.0 Flags: none Version: 2

Notes at offset 0x000000f4 with length 0x00000024:

Owner	Data size	Description
GNU	0x00000014	NT GNU BUILD ID (unique build ID bitstring)

图 B-4 main_s 结构信息

```

ELF Header:
  Magic:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
  Class:                                     ELF32
  Data:                                       2's complement, little endian
  Version:                                  1 (current)
  OS/ABI:                                    UNIX - System V
  ABI Version:                              0
  Type:                                       EXEC (Executable file)
  Machine:                                  Intel 80386
  Version:                                  0x1
  Entry point address:                       0x8048330
  Start of program headers:                  52 (bytes into file)
  Start of section headers:                  5200 (bytes into file)
  Flags:                                     0x0
  Size of this header:                       52 (bytes)
  Size of program headers:                   32 (bytes)
  Number of program headers:                  8
  Size of section headers:                   40 (bytes)
  Number of section headers:                  36
  Section header string table index:         33

Section Headers:
[Nr] Name                               Type             Addr             Off             Size             ES Flg Lk  Inf Al
[ 0]                               NULL             00000000         000000         000000         00      0 0  0 0
[ 1] .interp                           PROGBITS         08048134         000134         000013         00      A 0  0 1
[ 2] .note.ABI-tag                       NOTE             08048148         000148         000020         00      A 0  0 4
[ 3] .note.gnu.build-id                   NOTE             08048168         000168         000024         00      A 0  0 4
[ 4] .hash                               HASH             0804818c         00018c         000028         04      A 6  0 4

```

[5]	.gnu.hash	GNU_HASH	080481b4	0001b4	000020	04	A	6	0	4
[6]	.dynsym	DYNSYM	080481d4	0001d4	000050	10	A	7	1	4
[7]	.dynstr	STRTAB	08048224	000224	00004c	00	A	0	0	1
[8]	.gnu.version	VERSYM	08048270	000270	00000a	02	A	6	0	2
[9]	.gnu.version_r	VERNEED	0804827c	00027c	000020	00	A	7	1	4
[10]	.rel.dyn	REL	0804829c	00029c	000008	08	A	6	0	4
[11]	.rel.plt	REL	080482a4	0002a4	000018	08	A	6	13	4
[12]	.init	PROGBITS	080482bc	0002bc	000030	00	AX	0	0	4
[13]	.plt	PROGBITS	080482ec	0002ec	000040	04	AX	0	0	4
[14]	.text	PROGBITS	08048330	000330	00021c	00	AX	0	0	16
[15]	.fini	PROGBITS	0804854c	00054c	00001c	00	AX	0	0	4
[16]	.rodata	PROGBITS	08048568	000568	000076	00	A	0	0	4
[17]	.eh_frame	PROGBITS	080485e0	0005e0	000004	00	A	0	0	4
[18]	.ctors	PROGBITS	08049f0c	000f0c	000008	00	WA	0	0	4
[19]	.dtors	PROGBITS	08049f14	000f14	000008	00	WA	0	0	4
[20]	.jcr	PROGBITS	08049f1c	000f1c	000004	00	WA	0	0	4
[21]	.dynamic	DYNAMIC	08049f20	000f20	0000d0	08	WA	7	0	4
[22]	.got	PROGBITS	08049ff0	000ff0	000004	04	WA	0	0	4
[23]	.got.plt	PROGBITS	08049ffa	000ffa	000018	04	WA	0	0	4
[24]	.data	PROGBITS	0804a00c	00100c	000010	00	WA	0	0	4
[25]	.bss	NOBITS	0804a01c	00101c	000010	00	WA	0	0	4
[26]	.comment	PROGBITS	00000000	00101c	000046	01	MS	0	0	1
[27]	.debug_aranges	PROGBITS	00000000	001068	000020	00		0	0	8
[28]	.debug_pubnames	PROGBITS	00000000	001088	000025	00		0	0	1
[29]	.debug_info	PROGBITS	00000000	0010ad	0000f2	00		0	0	1
[30]	.debug_abbrev	PROGBITS	00000000	00119f	00005f	00		0	0	1
[31]	.debug_line	PROGBITS	00000000	0011fe	000082	00		0	0	1
[32]	.debug_str	PROGBITS	00000000	001280	000092	01	MS	0	0	1
[33]	.shstrtab	STRTAB	00000000	001312	00013e	00		0	0	1
[34]	.symtab	SYMTAB	00000000	0019f0	0004f0	10		35	53	4
[35]	.strtab	STRTAB	00000000	001ee0	000234	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00100	0x00100	R E	0x4
INTERP	0x000134	0x08048134	0x08048134	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x005e4	0x005e4	R E	0x1000
LOAD	0x000f0c	0x08049f0c	0x08049f0c	0x00110	0x00120	RW	0x1000
DYNAMIC	0x000f20	0x08049f20	0x08049f20	0x000d0	0x000d0	RW	0x4
NOTE	0x000148	0x08048148	0x08048148	0x00044	0x00044	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4
GNU_RELRO	0x000f0c	0x08049f0c	0x08049f0c	0x000f4	0x000f4	R	0x1

Section to Segment mapping:

Segment Sections...

00	
01	.interp
02	.interp .note.ABI-tag .note.gnu.build-
id .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt	
.init .plt .text .fini .rodata .eh_frame	
03	.ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04	.dynamic
05	.note.ABI-tag .note.gnu.build-id
06	
07	.ctors .dtors .jcr .dynamic .got

Dynamic section at offset 0xf20 contains 21 entries:

Tag	Type	Name/Value
0x00000001	(NEEDED)	Shared library: [libc.so.6]
0x0000000c	(INIT)	0x80482bc
0x0000000d	(FINI)	0x804854c
0x00000004	(HASH)	0x804818c
0x6ffffef5	(GNU_HASH)	0x80481b4
0x00000005	(STRTAB)	0x8048224
0x00000006	(SYMTAB)	0x80481d4
0x0000000a	(STRSZ)	76 (bytes)
0x0000000b	(SYMENT)	16 (bytes)
0x00000015	(DEBUG)	0x0
0x00000003	(PLTGOT)	0x8049ff4
0x00000002	(PLTRELSZ)	24 (bytes)
0x00000014	(PLTREL)	REL
0x00000017	(JMPREL)	0x80482a4
0x00000011	(REL)	0x804829c
0x00000012	(RELSZ)	8 (bytes)
0x00000013	(RELENT)	8 (bytes)
0x6ffffffe	(VERNEED)	0x804827c
0x6ffffff	(VERNEEDNUM)	1
0x6ffffff0	(VERSYM)	0x8048270
0x00000000	(NULL)	0x0

Relocation section '.rel.dyn' at offset 0x29c contains 1 entries:

Offset	Info	Type	Sym.Value	Sym. Name
08049ff0	00000106	R_386_GLOB_DAT	00000000	__gmon_start__

Relocation section '.rel.plt' at offset 0x2a4 contains 3 entries:

Offset	Info	Type	Sym.Value	Sym. Name
0804a000	00000107	R_386_JUMP_SLOT	00000000	__gmon_start__
0804a004	00000207	R_386_JUMP_SLOT	00000000	__libc_start_main
0804a008	00000307	R_386_JUMP_SLOT	00000000	printf

There are no unwind sections in this file.

Symbol table '.dynsym' contains 5 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
2:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.0
(2)	3: 00000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@GLIBC_2.0 (2)
	4: 0804856c	4	OBJECT	GLOBAL	DEFAULT	16	_IO_stdin_used

Symbol table '.symtab' contains 79 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	08048134	0	SECTION	LOCAL	DEFAULT	1	
2:	08048148	0	SECTION	LOCAL	DEFAULT	2	
3:	08048168	0	SECTION	LOCAL	DEFAULT	3	
4:	0804818c	0	SECTION	LOCAL	DEFAULT	4	
5:	080481b4	0	SECTION	LOCAL	DEFAULT	5	
6:	080481d4	0	SECTION	LOCAL	DEFAULT	6	
7:	08048224	0	SECTION	LOCAL	DEFAULT	7	
8:	08048270	0	SECTION	LOCAL	DEFAULT	8	
9:	0804827c	0	SECTION	LOCAL	DEFAULT	9	
10:	0804829c	0	SECTION	LOCAL	DEFAULT	10	
11:	080482a4	0	SECTION	LOCAL	DEFAULT	11	
12:	080482bc	0	SECTION	LOCAL	DEFAULT	12	
13:	080482ec	0	SECTION	LOCAL	DEFAULT	13	

14:	08048330	0	SECTION	LOCAL	DEFAULT	14	
15:	0804854c	0	SECTION	LOCAL	DEFAULT	15	
16:	08048568	0	SECTION	LOCAL	DEFAULT	16	
17:	080485e0	0	SECTION	LOCAL	DEFAULT	17	
18:	08049f0c	0	SECTION	LOCAL	DEFAULT	18	
19:	08049f14	0	SECTION	LOCAL	DEFAULT	19	
20:	08049f1c	0	SECTION	LOCAL	DEFAULT	20	
21:	08049f20	0	SECTION	LOCAL	DEFAULT	21	
22:	08049ff0	0	SECTION	LOCAL	DEFAULT	22	
23:	08049ff4	0	SECTION	LOCAL	DEFAULT	23	
24:	0804a00c	0	SECTION	LOCAL	DEFAULT	24	
25:	0804a01c	0	SECTION	LOCAL	DEFAULT	25	
26:	00000000	0	SECTION	LOCAL	DEFAULT	26	
27:	00000000	0	SECTION	LOCAL	DEFAULT	27	
28:	00000000	0	SECTION	LOCAL	DEFAULT	28	
29:	00000000	0	SECTION	LOCAL	DEFAULT	29	
30:	00000000	0	SECTION	LOCAL	DEFAULT	30	
31:	00000000	0	SECTION	LOCAL	DEFAULT	31	
32:	00000000	0	SECTION	LOCAL	DEFAULT	32	
33:	00000000	0	FILE	LOCAL	DEFAULT	ABS	init.c
34:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
35:	08049f0c	0	OBJECT	LOCAL	DEFAULT	18	__CTOR_LIST__
36:	08049f14	0	OBJECT	LOCAL	DEFAULT	19	__DTOR_LIST__
37:	08049f1c	0	OBJECT	LOCAL	DEFAULT	20	__JCR_LIST__
38:	08048360	0	FUNC	LOCAL	DEFAULT	14	__do_global_dtors_aux
39:	0804a01c	1	OBJECT	LOCAL	DEFAULT	25	completed.6990
40:	0804a020	4	OBJECT	LOCAL	DEFAULT	25	dtor_idx.6992
41:	080483c0	0	FUNC	LOCAL	DEFAULT	14	frame_dummy
42:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
43:	08049f10	0	OBJECT	LOCAL	DEFAULT	18	__CTOR_END__
44:	080485e0	0	OBJECT	LOCAL	DEFAULT	17	__FRAME_END__
45:	08049f1c	0	OBJECT	LOCAL	DEFAULT	20	__JCR_END__
46:	08048520	0	FUNC	LOCAL	DEFAULT	14	__do_global_ctors_aux
47:	00000000	0	FILE	LOCAL	DEFAULT	ABS	main.c
48:	00000000	0	FILE	LOCAL	DEFAULT	ABS	part.c
49:	08049ff4	0	OBJECT	LOCAL	HIDDEN	23	__GLOBAL_OFFSET_TABLE__
50:	08049f0c	0	NOTYPE	LOCAL	HIDDEN	18	__init_array_end
51:	08049f0c	0	NOTYPE	LOCAL	HIDDEN	18	__init_array_start
52:	08049f20	0	OBJECT	LOCAL	HIDDEN	21	__DYNAMIC
53:	0804a00c	0	NOTYPE	WEAK	DEFAULT	24	__data_start
54:	080484b0	5	FUNC	GLOBAL	DEFAULT	14	__libc_csu_fini
55:	08048330	0	FUNC	GLOBAL	DEFAULT	14	__start
56:	0804a024	4	OBJECT	GLOBAL	DEFAULT	25	__g_str1
57:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
58:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__Jv_RegisterClasses
59:	08048568	4	OBJECT	GLOBAL	DEFAULT	16	__fp_hw
60:	0804854c	0	FUNC	GLOBAL	DEFAULT	15	__fini
61:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@@GLIBC_
62:	0804856c	4	OBJECT	GLOBAL	DEFAULT	16	__IO_stdin_used
63:	0804a028	4	OBJECT	GLOBAL	DEFAULT	25	__g_int1
64:	0804a00c	0	NOTYPE	GLOBAL	DEFAULT	24	__data_start
65:	0804a010	0	OBJECT	GLOBAL	HIDDEN	24	__dso_handle
66:	08049f18	0	OBJECT	GLOBAL	HIDDEN	19	__DTOR_END__
67:	080484c0	90	FUNC	GLOBAL	DEFAULT	14	__libc_csu_init
68:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@@GLIBC_2.0
69:	0804a014	4	OBJECT	GLOBAL	DEFAULT	24	__g_int2
70:	0804a01c	0	NOTYPE	GLOBAL	DEFAULT	ABS	__bss_start
71:	0804848d	27	FUNC	GLOBAL	DEFAULT	14	__func_2
72:	0804a02c	0	NOTYPE	GLOBAL	DEFAULT	ABS	__end
73:	0804a018	4	OBJECT	GLOBAL	DEFAULT	24	__g_str2
74:	0804a01c	0	NOTYPE	GLOBAL	DEFAULT	ABS	__edata
75:	0804846c	33	FUNC	GLOBAL	DEFAULT	14	__func_1

76:	0804851a	0	FUNC	GLOBAL	HIDDEN	14	__i686.get_pc_thunk.bx
77:	080483e4	133	FUNC	GLOBAL	DEFAULT	14	main
78:	080482bc	0	FUNC	GLOBAL	DEFAULT	12	_init

Histogram for bucket list length (total of 3 buckets):

Length	Number	% of total	Coverage
0	0	(0.0%)	
1	2	(66.7%)	50.0%
2	1	(33.3%)	100.0%

Histogram for '.gnu.hash' bucket list length (total of 2 buckets):

Length	Number	% of total	Coverage
0	1	(50.0%)	
1	1	(50.0%)	100.0%

Version symbols section '.gnu.version' contains 5 entries:

Addr:	0000000008048270	Offset:	0x000270	Link:	6 (.dynsym)
000:	0 (*local*)	0 (*local*)	2 (GLIBC_2.0)	2 (GLIBC_2.0)	
004:	1 (*global*)				

Version needs section '.gnu.version_r' contains 1 entries:

Addr:	0x000000000804827c	Offset:	0x00027c	Link:	7 (.dynstr)
000000:	Version: 1	File:	libc.so.6	Cnt:	1
0x0010:	Name: GLIBC_2.0	Flags:	none	Version:	2

Notes at offset 0x00000148 with length 0x00000020:

Owner	Data size	Description
GNU	0x00000010	NT_GNU_ABI_TAG (ABI version tag)

Notes at offset 0x00000168 with length 0x00000024:

Owner	Data size	Description
GNU	0x00000014	NT_GNU_BUILD_ID (unique build ID bitstring)

图 B-5 main_d 结构信息

```

ELF Header:
Magic:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:                                     ELF32
Data:                                     2's complement, little endian
Version:                                1 (current)
OS/ABI:                                UNIX - System V
ABI Version:                            0
Type:                                   EXEC (Executable file)
Machine:                                Intel 80386
Version:                                0x1
Entry point address:                    0x8048520
Start of program headers:                52 (bytes into file)
Start of section headers:                5200 (bytes into file)
Flags:                                   0x0
Size of this header:                     52 (bytes)
Size of program headers:                 32 (bytes)
Number of program headers:                8
Size of section headers:                 40 (bytes)
Number of section headers:                36
Section header string table index:       33

```

```

Section Headers:
[Nr] Name                               Type             Addr             Off             Size             ES Flg Lk  Inf Al
[ 0]                               NULL             00000000         000000         000000         00      0  0  0  0
[ 1] .interp                           PROGBITS         08048134         000134         000013         00      A  0  0  1
[ 2] .note.ABI-tag                     NOTE             08048148         000148         000020         00      A  0  0  4
[ 3] .note.gnu.build-id                 NOTE             08048168         000168         000024         00      A  0  0  4
[ 4] .hash                             HASH             0804818c         00018c         000058         04      A  6  0  4

```

[5]	.gnu.hash	GNU_HASH	080481e4	0001e4	00004c	04	A	6	0	4
[6]	.dynsym	DYNSYM	08048230	000230	000110	10	A	7	1	4
[7]	.dynstr	STRTAB	08048340	000340	0000b9	00	A	0	0	1
[8]	.gnu.version	VERSYM	080483fa	0003fa	000022	02	A	6	0	2
[9]	.gnu.version_r	VERNEED	0804841c	00041c	000020	00	A	7	1	4
[10]	.rel.dyn	REL	0804843c	00043c	000028	08	A	6	0	4
[11]	.rel.plt	REL	08048464	000464	000028	08	A	6	13	4
[12]	.init	PROGBITS	0804848c	00048c	000030	00	AX	0	0	4
[13]	.plt	PROGBITS	080484bc	0004bc	000060	04	AX	0	0	4
[14]	.text	PROGBITS	08048520	000520	0001dc	00	AX	0	0	16
[15]	.fini	PROGBITS	080486fc	0006fc	00001c	00	AX	0	0	4
[16]	.rodata	PROGBITS	08048718	000718	00004e	00	A	0	0	4
[17]	.eh_frame	PROGBITS	08048768	000768	000004	00	A	0	0	4
[18]	.ctors	PROGBITS	08049f04	000f04	000008	00	WA	0	0	4
[19]	.dtors	PROGBITS	08049f0c	000f0c	000008	00	WA	0	0	4
[20]	.jcr	PROGBITS	08049f14	000f14	000004	00	WA	0	0	4
[21]	.dynamic	DYNAMIC	08049f18	000f18	0000d8	08	WA	7	0	4
[22]	.got	PROGBITS	08049ff0	000ff0	000004	04	WA	0	0	4
[23]	.got.plt	PROGBITS	08049ff4	000ff4	000020	04	WA	0	0	4
[24]	.data	PROGBITS	0804a014	001014	000008	00	WA	0	0	4
[25]	.bss	NOBITS	0804a01c	00101c	000018	00	WA	0	0	4
[26]	.comment	PROGBITS	00000000	00101c	000046	01	MS	0	0	1
[27]	.debug_aranges	PROGBITS	00000000	001068	000020	00		0	0	8
[28]	.debug_pubnames	PROGBITS	00000000	001088	000025	00		0	0	1
[29]	.debug_info	PROGBITS	00000000	0010ad	0000f2	00		0	0	1
[30]	.debug_abbrev	PROGBITS	00000000	00119f	00005f	00		0	0	1
[31]	.debug_line	PROGBITS	00000000	0011fe	000082	00		0	0	1
[32]	.debug_str	PROGBITS	00000000	001280	000092	01	MS	0	0	1
[33]	.shstrtab	STRTAB	00000000	001312	00013e	00		0	0	1
[34]	.symtab	SYMTAB	00000000	0019f0	0004e0	10		35	52	4
[35]	.strtab	STRTAB	00000000	001ed0	00022d	00		0	0	1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00100	0x00100	R E	0x4
INTERP	0x000134	0x08048134	0x08048134	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x0076c	0x0076c	R E	0x1000
LOAD	0x000f04	0x08049f04	0x08049f04	0x00118	0x00130	RW	0x1000
DYNAMIC	0x000f18	0x08049f18	0x08049f18	0x000d8	0x000d8	RW	0x4
NOTE	0x000148	0x08048148	0x08048148	0x00044	0x00044	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4
GNU_RELRO	0x000f04	0x08049f04	0x08049f04	0x000fc	0x000fc	R	0x1

Section to Segment mapping:
Segment Sections...

00	
01	.interp
02	.interp .note.ABI-tag .note.gnu.build-
id	.hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt
	.init .plt .text .fini .rodata .eh_frame
03	.ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04	.dynamic
05	.note.ABI-tag .note.gnu.build-id
06	
07	.ctors .dtors .jcr .dynamic .got

Dynamic section at offset 0xf18 contains 22 entries:

Tag	Type	Name/Value
0x00000001	(NEEDED)	Shared library: [libpart.so]
0x00000001	(NEEDED)	Shared library: [libc.so.6]
0x0000000c	(INIT)	0x804848c
0x0000000d	(FINI)	0x80486fc
0x00000004	(HASH)	0x804818c
0x6ffffef5	(GNU_HASH)	0x80481e4
0x00000005	(STRTAB)	0x8048340
0x00000006	(SYMTAB)	0x8048230
0x0000000a	(STRSZ)	185 (bytes)
0x0000000b	(SYMENT)	16 (bytes)
0x00000015	(DEBUG)	0x0
0x00000003	(PLTGOT)	0x8049ff4
0x00000002	(PLTRELSZ)	40 (bytes)
0x00000014	(PLTREL)	REL
0x00000017	(JMPREL)	0x8048464
0x00000011	(REL)	0x804843c
0x00000012	(RELSZ)	40 (bytes)
0x00000013	(RELENT)	8 (bytes)
0x6ffffffe	(VERNEED)	0x804841c
0x6fffffff	(VERNEEDNUM)	1
0x6ffffff0	(VERSYM)	0x80483fa
0x00000000	(NULL)	0x0

Relocation section '.rel.dyn' at offset 0x43c contains 5 entries:

Offset	Info	Type	Sym.Value	Sym. Name
08049ff0	00000106	R_386_GLOB_DAT	00000000	__gmon_start__
0804a01c	00000705	R_386_COPY	0804a01c	g_str1
0804a020	00000805	R_386_COPY	0804a020	g_int1
0804a024	00000c05	R_386_COPY	0804a024	g_int2
0804a028	00000e05	R_386_COPY	0804a028	g_str2

Relocation section '.rel.plt' at offset 0x464 contains 5 entries:

Offset	Info	Type	Sym.Value	Sym. Name
0804a000	00000107	R_386_JUMP_SLOT	00000000	__gmon_start__
0804a004	00000307	R_386_JUMP_SLOT	00000000	__libc_start_main
0804a008	00000407	R_386_JUMP_SLOT	00000000	printf
0804a00c	00000507	R_386_JUMP_SLOT	00000000	func_2
0804a010	00000607	R_386_JUMP_SLOT	00000000	func_1

There are no unwind sections in this file.

Symbol table '.dynsym' contains 17 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
2:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__Jv_RegisterClasses
3:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.0
(2)							
4:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@GLIBC_2.0 (2)
5:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	func_2
6:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	func_1
7:	0804a01c	4	OBJECT	GLOBAL	DEFAULT	25	g_str1
8:	0804a020	4	OBJECT	GLOBAL	DEFAULT	25	g_int1
9:	0804a034	0	NOTYPE	GLOBAL	DEFAULT	ABS	_end
10:	0804a01c	0	NOTYPE	GLOBAL	DEFAULT	ABS	_edata
11:	0804871c	4	OBJECT	GLOBAL	DEFAULT	16	_IO_stdin_used
12:	0804a024	4	OBJECT	GLOBAL	DEFAULT	25	g_int2
13:	0804a01c	0	NOTYPE	GLOBAL	DEFAULT	ABS	__bss_start
14:	0804a028	4	OBJECT	GLOBAL	DEFAULT	25	g_str2

15:	0804848c	0	FUNC	GLOBAL	DEFAULT	12	_init
16:	080486fc	0	FUNC	GLOBAL	DEFAULT	15	_fini
Symbol table '.symtab' contains 78 entries:							
Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	08048134	0	SECTION	LOCAL	DEFAULT	1	
2:	08048148	0	SECTION	LOCAL	DEFAULT	2	
3:	08048168	0	SECTION	LOCAL	DEFAULT	3	
4:	0804818c	0	SECTION	LOCAL	DEFAULT	4	
5:	080481e4	0	SECTION	LOCAL	DEFAULT	5	
6:	08048230	0	SECTION	LOCAL	DEFAULT	6	
7:	08048340	0	SECTION	LOCAL	DEFAULT	7	
8:	080483fa	0	SECTION	LOCAL	DEFAULT	8	
9:	0804841c	0	SECTION	LOCAL	DEFAULT	9	
10:	0804843c	0	SECTION	LOCAL	DEFAULT	10	
11:	08048464	0	SECTION	LOCAL	DEFAULT	11	
12:	0804848c	0	SECTION	LOCAL	DEFAULT	12	
13:	080484bc	0	SECTION	LOCAL	DEFAULT	13	
14:	08048520	0	SECTION	LOCAL	DEFAULT	14	
15:	080486fc	0	SECTION	LOCAL	DEFAULT	15	
16:	08048718	0	SECTION	LOCAL	DEFAULT	16	
17:	08048768	0	SECTION	LOCAL	DEFAULT	17	
18:	08049f04	0	SECTION	LOCAL	DEFAULT	18	
19:	08049f0c	0	SECTION	LOCAL	DEFAULT	19	
20:	08049f14	0	SECTION	LOCAL	DEFAULT	20	
21:	08049f18	0	SECTION	LOCAL	DEFAULT	21	
22:	08049ff0	0	SECTION	LOCAL	DEFAULT	22	
23:	08049ff4	0	SECTION	LOCAL	DEFAULT	23	
24:	0804a014	0	SECTION	LOCAL	DEFAULT	24	
25:	0804a01c	0	SECTION	LOCAL	DEFAULT	25	
26:	00000000	0	SECTION	LOCAL	DEFAULT	26	
27:	00000000	0	SECTION	LOCAL	DEFAULT	27	
28:	00000000	0	SECTION	LOCAL	DEFAULT	28	
29:	00000000	0	SECTION	LOCAL	DEFAULT	29	
30:	00000000	0	SECTION	LOCAL	DEFAULT	30	
31:	00000000	0	SECTION	LOCAL	DEFAULT	31	
32:	00000000	0	SECTION	LOCAL	DEFAULT	32	
33:	00000000	0	FILE	LOCAL	DEFAULT	ABS	init.c
34:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
35:	08049f04	0	OBJECT	LOCAL	DEFAULT	18	__CTOR_LIST__
36:	08049f0c	0	OBJECT	LOCAL	DEFAULT	19	__DTOR_LIST__
37:	08049f14	0	OBJECT	LOCAL	DEFAULT	20	__JCR_LIST__
38:	08048550	0	FUNC	LOCAL	DEFAULT	14	__do_global_dtors_aux
39:	0804a02c	1	OBJECT	LOCAL	DEFAULT	25	completed.6990
40:	0804a030	4	OBJECT	LOCAL	DEFAULT	25	dtor_idx.6992
41:	080485b0	0	FUNC	LOCAL	DEFAULT	14	frame_dummy
42:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
43:	08049f08	0	OBJECT	LOCAL	DEFAULT	18	__CTOR_END__
44:	08048768	0	OBJECT	LOCAL	DEFAULT	17	__FRAME_END__
45:	08049f14	0	OBJECT	LOCAL	DEFAULT	20	__JCR_END__
46:	080486d0	0	FUNC	LOCAL	DEFAULT	14	__do_global_ctors_aux
47:	00000000	0	FILE	LOCAL	DEFAULT	ABS	main.c
48:	08049ff4	0	OBJECT	LOCAL	HIDDEN	23	__GLOBAL_OFFSET_TABLE__
49:	08049f04	0	NOTYPE	LOCAL	HIDDEN	18	__init_array_end
50:	08049f04	0	NOTYPE	LOCAL	HIDDEN	18	__init_array_start
51:	08049f18	0	OBJECT	LOCAL	HIDDEN	21	__DYNAMIC
52:	0804a014	0	NOTYPE	WEAK	DEFAULT	24	__data_start
53:	08048660	5	FUNC	GLOBAL	DEFAULT	14	__libc_csu_fini
54:	08048520	0	FUNC	GLOBAL	DEFAULT	14	__start
55:	0804a01c	4	OBJECT	GLOBAL	DEFAULT	25	g_strl
56:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	gmon_start

```

57: 00000000      0 NOTYPE WEAK  DEFAULT UND _Jv_RegisterClasses
58: 08048718      4 OBJECT GLOBAL DEFAULT 16 _fp_hw
59: 080486fc      0 FUNC  GLOBAL DEFAULT 15 _fini
60: 00000000      0 FUNC  GLOBAL DEFAULT UND __libc_start_main@@GLIBC_
61: 0804871c      4 OBJECT GLOBAL DEFAULT 16 _IO_stdin_used
62: 0804a020      4 OBJECT GLOBAL DEFAULT 25 g_int1
63: 0804a014      0 NOTYPE GLOBAL DEFAULT 24 __data_start
64: 0804a018      0 OBJECT GLOBAL HIDDEN 24 __dso_handle
65: 08049f10      0 OBJECT GLOBAL HIDDEN 19 __DTOR_END
66: 08048670     90 FUNC  GLOBAL DEFAULT 14 __libc_csu_init
67: 00000000      0 FUNC  GLOBAL DEFAULT UND printf@@GLIBC_2.0
68: 0804a024      4 OBJECT GLOBAL DEFAULT 25 g_int2
69: 0804a01c      0 NOTYPE GLOBAL DEFAULT ABS __bss_start
70: 00000000      0 FUNC  GLOBAL DEFAULT UND func_2
71: 0804a034      0 NOTYPE GLOBAL DEFAULT ABS __end
72: 0804a028      4 OBJECT GLOBAL DEFAULT 25 g_str2
73: 0804a01c      0 NOTYPE GLOBAL DEFAULT ABS __edata
74: 00000000      0 FUNC  GLOBAL DEFAULT UND func_1
75: 080486ca      0 FUNC  GLOBAL HIDDEN 14 __i686.get_pc_thunk.bx
76: 080485d4     133 FUNC  GLOBAL DEFAULT 14 main
77: 0804848c      0 FUNC  GLOBAL DEFAULT 12 _init

```

Histogram for bucket list length (total of 3 buckets):

Length	Number	% of total	Coverage
0	0	(0.0%)	
1	0	(0.0%)	0.0%
2	0	(0.0%)	0.0%
3	0	(0.0%)	0.0%
4	1	(33.3%)	25.0%
5	0	(0.0%)	25.0%
6	2	(66.7%)	100.0%

Histogram for '.gnu.hash' bucket list length (total of 3 buckets):

Length	Number	% of total	Coverage
0	0	(0.0%)	
1	1	(33.3%)	10.0%
2	0	(0.0%)	10.0%
3	0	(0.0%)	10.0%
4	1	(33.3%)	50.0%
5	1	(33.3%)	100.0%

Version symbols section '.gnu.version' contains 17 entries:

```

Addr: 00000000080483fa Offset: 0x0003fa Link: 6 (.dynsym)
000: 0 (*local*)      0 (*local*)      0 (*local*)      2 (GLIBC_2.0)
004: 2 (GLIBC_2.0)     0 (*local*)      0 (*local*)      0 (*local*)
008: 0 (*local*)       1 (*global*)     1 (*global*)     1 (*global*)
00c: 0 (*local*)       1 (*global*)     0 (*local*)      1 (*global*)
010: 1 (*global*)

```

Version needs section '.gnu.version_r' contains 1 entries:

```

Addr: 0x000000000804841c Offset: 0x00041c Link: 7 (.dynstr)
000000: Version: 1 File: libc.so.6 Cnt: 1
0x0010: Name: GLIBC_2.0 Flags: none Version: 2

```

Notes at offset 0x00000148 with length 0x00000020:

Owner	Data size	Description
GNU	0x00000010	NT_GNU_ABI_TAG (ABI version tag)

Notes at offset 0x00000168 with length 0x00000024:

Owner	Data size	Description
GNU	0x00000014	NT_GNU_BUILD_ID (unique build ID bitstring)

图 B-6 part.o 反汇编码

```

part.o:      file format elf32-i386

Disassembly of section .text:

00000000 <func_1>:
  0:      55                push    %ebp
  1:      89 e5            mov     %esp,%ebp
  3:      83 ec 28         sub     $0x28,%esp
  6:      8b 45 08         mov     0x8(%ebp),%eax
  9:      89 45 f4         mov     %eax,-0xc(%ebp)
 c:      8b 45 f4         mov     -0xc(%ebp),%eax
 f:      89 44 24 04      mov     %eax,0x4(%esp)
13:      c7 04 24 04 00 00 00 movl    $0x4, (%esp)
1a:      e8 fc ff ff ff   call    1b <func_1+0x1b>
1f:      c9              leave   %eax
20:      c3              ret

00000021 <func_2>:
21:      55                push    %ebp
22:      89 e5            mov     %esp,%ebp
24:      83 ec 18         sub     $0x18,%esp
27:      8b 45 08         mov     0x8(%ebp),%eax
2a:      89 44 24 04      mov     %eax,0x4(%esp)
2e:      c7 04 24 15 00 00 00 movl    $0x15, (%esp)
35:      e8 fc ff ff ff   call    36 <func_2+0x15>
3a:      c9              leave   %eax
3b:      c3              ret

```

图 B-7 part.o 字节码

```

87654321 0011 2233 4455 6677 8899 aabb ccdd eeff 0123456789abcdef
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000 .ELF.....
00000010: 0100 0300 0100 0000 0000 0000 0000 0000 .....4.....
00000020: 1c01 0000 0000 0000 3400 0000 0000 2800 .....(.....
00000030: 0c00 0900 5589 e583 ec28 8b45 0889 45f4 ....U....(.E..E.
00000040: 8b45 f489 4424 04c7 0424 0400 0000 e8fc .E..D$....$.
00000050: ffff ffc9 c355 89e5 83ec 188b 4508 8944 ....U.....E..D
00000060: 2404 c704 2415 0000 00e8 fcff ffff c9c3 $....$.
00000070: 0500 0000 0000 0000 7879 7a00 6675 6e63 .....xyz.func
00000080: 5f31 203a 206a 203d 2025 640a 0066 756e _1 : j = %d..fun
00000090: 635f 3220 3a20 7374 7220 3d20 2573 0a00 _2 : str = %s..
000000a0: 0047 4343 3a20 2855 6275 6e74 7520 342e .GCC: (Ubuntu 4.
000000b0: 342e 312d 3475 6275 6e74 7539 2920 342e 4.1-4ubuntu9) 4.
000000c0: 342e 3100 002e 7379 6d74 6162 002e 7374 4.1...symtab..st
000000d0: 7274 6162 002e 7368 7374 7274 6162 002e rtab..shstrtab..
000000e0: 7265 6c2e 7465 7874 002e 7265 6c2e 6461 rel.text..rel.da
000000f0: 7461 002e 6273 7300 2e72 6f64 6174 6100 ta..bss..rodata.
00000100: 2e63 6f6d 6d65 6e74 002e 6e6f 7465 2e47 .comment..note.G
00000110: 4e55 2d73 7461 636b 0000 0000 0000 0000 NU-stack.....
00000120: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000130: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000140: 0000 0000 1f00 0000 0100 0000 0600 0000 .....
00000150: 0000 0000 3400 0000 3c00 0000 0000 0000 ....4...<.....
00000160: 0000 0000 0400 0000 0000 0000 1b00 0000 .....
00000170: 0900 0000 0000 0000 0000 0000 2804 0000 .....(....
00000180: 2000 0000 0a00 0000 0100 0000 0400 0000 .....
00000190: 0800 0000 2900 0000 0100 0000 0300 0000 .....).....
000001a0: 0000 0000 7000 0000 0800 0000 0000 0000 ....p.....
000001b0: 0000 0000 0400 0000 0000 0000 2500 0000 .....%...

```

000001c0:	0900	0000	0000	0000	0000	0000	4804	0000H...
000001d0:	0800	0000	0a00	0000	0300	0000	0400	0000
000001e0:	0800	0000	2f00	0000	0800	0000	0300	0000/.
000001f0:	0000	0000	7800	0000	0000	0000	0000	0000x.....
00000200:	0000	0000	0400	0000	0000	0000	3400	00004...
00000210:	0100	0000	0200	0000	0000	0000	7800	0000x...
00000220:	2800	0000	0000	0000	0000	0000	0100	0000	(.....
00000230:	0000	0000	3c00	0000	0100	0000	3000	0000<.....0...
00000240:	0000	0000	a000	0000	2400	0000	0000	0000\$.
00000250:	0000	0000	0100	0000	0100	0000	4500	0000E...
00000260:	0100	0000	0000	0000	0000	0000	c400	0000
00000270:	0000	0000	0000	0000	0000	0000	0100	0000
00000280:	0000	0000	1100	0000	0300	0000	0000	0000
00000290:	0000	0000	c400	0000	5500	0000	0000	0000U.....
000002a0:	0000	0000	0100	0000	0000	0000	0100	0000
000002b0:	0200	0000	0000	0000	0000	0000	fc02	0000
000002c0:	f000	0000	0b00	0000	0800	0000	0400	0000
000002d0:	1000	0000	0900	0000	0300	0000	0000	0000
000002e0:	0000	0000	ec03	0000	3900	0000	0000	00009.....
000002f0:	0000	0000	0100	0000	0000	0000	0000	0000
00000300:	0000	0000	0000	0000	0000	0000	0100	0000
00000310:	0000	0000	0000	0000	0400	f1ff	0000	0000
00000320:	0000	0000	0000	0000	0300	0100	0000	0000
00000330:	0000	0000	0000	0000	0300	0300	0000	0000
00000340:	0000	0000	0000	0000	0300	0500	0000	0000
00000350:	0000	0000	0000	0000	0300	0600	0000	0000
00000360:	0000	0000	0000	0000	0300	0800	0000	0000
00000370:	0000	0000	0000	0000	0300	0700	0800	0000
00000380:	0400	0000	0400	0000	1100	f2ff	0f00	0000
00000390:	0000	0000	0400	0000	1100	0300	1600	0000
000003a0:	0400	0000	0400	0000	1100	f2ff	1d00	0000
000003b0:	0400	0000	0400	0000	1100	0300	2400	0000\$.
000003c0:	0000	0000	2100	0000	1200	0100	2b00	0000!.+...
000003d0:	0000	0000	0000	0000	1000	0000	3200	00002....
000003e0:	2100	0000	1b00	0000	1200	0100	0070	6172	!.par
000003f0:	742e	6300	675f	696e	7431	0067	5f69	6e74	t.c.g_int1.g_int
00000400:	3200	675f	7374	7231	0067	5f73	7472	3200	2.g_str1.g_str2.
00000410:	6675	6e63	5f31	0070	7269	6e74	6600	6675	func_1.printf.fu
00000420:	6e63	5f32	0000	0000	1600	0000	0105	0000	nc_2.....
00000430:	1b00	0000	020d	0000	3100	0000	0105	00001.....
00000440:	3600	0000	020d	0000	0400	0000	0105	0000	6.....