Oracle® Solaris 11.2 链接程序和库指南

文件号码 E54069
2014 年 7 月

ORACLE

目录

使用此文档 13

I 使用链接编辑器和运行时链接程序 15

1 Oracle Solaris 链接编辑器介绍 17
链接编辑 17
运行时链接 18
相关主题 19

2 链接编辑器 23
调用链接编辑器 24
指定链接编辑器选项 25
输入文件处理 27
符号处理 35
生成输出文件 50
重定位处理 68
桩目标文件 70
辅助目标文件 74
压缩调试节 78
父目标文件 81
调试帮助 83

3 运行时链接程序 87
共享目标文件依赖项 88
重定位处理 91
装入其他目标文件 96
延迟装入动态依赖项 97
初始化和终止例程 100
安全性 104
运行时链接编程接口 105
调试帮助 116

4 共享目标文件 123

命名约定 123
具有依赖项的共享目标文件 126
依赖项排序 127
作为过滤器的共享目标文件 127

II 快速参考 135

5 链接编辑器快速参考 137
静态模式 137
动态模式 138

III 高级主题 141

6 直接绑定 143
观察符号绑定 143
启用直接绑定 145
直接绑定和插入 149
阻止直接绑定到某个符号 154

7 生成目标文件以优化系统性能 159
使用 elfdump 分析文件 159
底层系统 161
延迟装入动态依赖项 161
与位置无关的代码 162
删除未使用的材料 164
最大化可共享性 167
最小化分页活动 168
重定位 169
使用 -B symbolic 选项 173
配置共享目标文件 173

8 *Mapfile* 177
mapfile 结构和语法 177
mapfile 指令 183
预定义段 200
映射示例 202
链接编辑器内部：节和段的处理 204

9 接口和版本控制 209
接口兼容性 209
内部版本控制 210
外部版本控制 224

10 使用动态字符串标记建立依赖性 227
特定于功能的共享目标文件 227
特定于指令集的共享目标文件 229
特定于系统的共享目标文件 231
查找关联的依赖项 231

11 可扩展性机制 237
链接编辑器支持接口 237
运行时链接程序审计接口 243
运行时链接程序调试器接口 256

IV ELF 应用程序二进制接口 269

12 目标文件格式 271
文件格式 271
数据表示形式 273
ELF 头 274
ELF 标识 277
数据编码 280
节 281
节合并 294
节压缩 295
特殊节 297
辅助节 303
COMDAT 节 304
组节 305
功能节 306
散列表节 308
移动部分 310
注释节 312
重定位节 314
字符串表节 325
符号表节 326
Syminfo 表节 336
版本控制节 337

13 程序装入和动态链接 343
程序头 343
程序装入（特定于处理器） 348
运行时链接程序 356
动态节 356

全局偏移表（特定于处理器） 371
过程链接表（特定于处理器） 372

14 线程局部存储 381
C/C++ 编程接口 381
线程局部存储节 382
线程局部存储的运行时分配 383
线程局部存储的访问模型 386

V 附录 405

A 链接程序和库的更新及新增功能 407
Oracle Solaris 11.2 发行版 407
Oracle Solaris 11.1 发行版 407
Oracle Solaris 11 408
Oracle Solaris 10 1/13 发行版 408
Oracle Solaris 10 8/11 发行版 408
Solaris 10 5/08 发行版 410
Solaris 10 8/07 发行版 410
Solaris 10 1/06 发行版 410
Solaris 10 发行版 410

B *System V* 发行版 *4*（版本 *1*）*mapfile* 413
mapfile 结构和语法 413
映射示例 419
mapfile 选项缺省值 420
内部映射结构 421

索引 425

图

图 3-1 单个 dlopen（） 请求 109
图 3-2 多个 dlopen（） 请求 110
图 3-3 具有公共依赖项的多个 dlopen（） 请求 111
图 10-1 非绑定依赖项 231
图 10-2 非绑定共同依赖项 233
图 11-1 *rtld-debugger* 信息流程 257
图 12-1 目标文件格式 272
图 12-2 数据编码 ELFDATA2LSB 280
图 12-3 数据编码 ELFDATA2MSB 280
图 12-4 符号散列表 309
图 12-5 注释信息 312
图 12-6 注释段示例 313
图 12-7 ELF 字符串表 326
图 13-1 SPARC: 可执行文件（64 K 对齐） 349
图 13-2 32 位 x86: 可执行文件（64 K 对齐） 351
图 13-3 32 位 SPARC: 进程映像段 353
图 13-4 x86: 进程映像段 354
图 14-1 线程局部存储的运行时存储布局 384
图 14-2 线程局部存储的访问模型和转换 388
图 B-1 简单的映射结构 422

表

表 2-1 CA_SUNW_SF_1 帧指针标志组合状态表 58
表 8-1 双引号文本转义序列 178
表 8-2 *mapfile* 中的名称和其他广泛使用的字符串 179
表 8-3 段标志 179
表 8-4 预定义的条件表达式名称 180
表 8-5 条件表达式运算符 181
表 8-6 *mapfile* 指令 183
表 8-7 节 FLAGS 值 191
表 8-8 符号作用域类型 196
表 8-9 SH_ATTR 值 198
表 8-10 符号 FLAG 值 199
表 9-1 接口兼容性示例 210
表 12-1 ELF 32 位数据类型 273
表 12-2 ELF 64 位数据类型 273
表 12-3 ELF 标识索引 278
表 12-4 ELF 特殊节索引 281
表 12-5 ELF 节类型 sh_type 284
表 12-6 ELF 节头表项：索引 0 289
表 12-7 ELF 扩展的节头表项：索引 0 290
表 12-8 ELF 节属性标志 290
表 12-9 ELF sh_link 和 sh_info 解释 294
表 12-10 ELF 压缩类型 ch_type 296
表 12-11 GNU ZLIB 压缩，gch_magic 297
表 12-12 ELF 特殊节 297
表 12-13 ELF 辅助数组标记 303
表 12-14 ELF 组节标志 305
表 12-15 ELF 功能数组标记 306
表 12-16 SPARC: ELF 重定位类型 318
表 12-17 64 位 SPARC: ELF 重定位类型 321
表 12-18 32 位 x86: ELF 重定位类型 322

表 12-19 x64: ELF 重定位类型 324
表 12-20 ELF 字符串表索引 326
表 12-21 ELF 符号绑定 ELF32_ST_BIND 和 ELF64_ST_BIND 328
表 12-22 ELF 符号类型 ELF32_ST_TYPE 和 ELF64_ST_TYPE 329
表 12-23 ELF 符号可见性 330
表 12-24 ELF 符号表项：索引 0 332
表 12-25 SPARC: ELF 符号表项：寄存器符号 336
表 12-26 SPARC: ELF 寄存器编号 336
表 12-27 ELF 版本依赖性索引 342
表 13-1 ELF 段类型 344
表 13-2 ELF 段标志 347
表 13-3 ELF 段权限 347
表 13-4 SPARC: ELF 程序头段（64 K 对齐） 350
表 13-5 32 位 x86: ELF 程序头段（64 K 对齐） 351
表 13-6 32 位 SPARC: ELF 共享目标文件段地址示例 355
表 13-7 32 位 x86: ELF 共享目标文件段地址示例 355
表 13-8 ELF 动态数组标记 357
表 13-9 ELF 动态标志 DT_FLAGS 366
表 13-10 ELF 动态标志 DT_FLAGS_1 367
表 13-11 ELF 动态位置标志 DT_POSFLAG_1 370
表 13-12 ELF ASLR 值、DT_SUNW_ASLR 370
表 13-13 ELF 动态放宽标志 DT_SUNW_RELAX 371
表 13-14 32 位 SPARC: 过程链接表示例 372
表 13-15 64 位 SPARC: 过程链接表示例 375
表 13-16 32 位 x86: 绝对过程链接表示例 377
表 13-17 32 位 x86: 与位置无关的过程链接表示例 377
表 13-18 x64: 过程链接表示例 378
表 14-1 ELF PT_TLS 程序头项 383
表 14-2 SPARC: 常规动态的线程局部变量访问代码 389
表 14-3 SPARC: 局部动态的线程局部变量访问代码 390
表 14-4 32 位 SPARC: 初始可执行的线程局部变量访问代码 391
表 14-5 64 位 SPARC: 初始可执行的线程局部变量访问代码 392
表 14-6 SPARC: 局部可执行的线程局部变量访问代码 392
表 14-7 SPARC: 线程局部存储的重定位类型 393
表 14-8 32 位 x86: 常规动态的线程局部变量访问代码 395
表 14-9 32 位 x86: 局部动态的线程局部变量访问代码 395
表 14-10 32 位 x86: 初始可执行的、位置无关的线程局部变量访问代码 396
表 14-11 32 位 x86: 初始可执行的、位置相关的线程局部变量访问代码 396

表 14-12 32 位 x86: 初始可执行的、位置无关的动态线程局部变量访问代码 ... 397
表 14-13 32 位 x86: 初始可执行的、位置无关的线程局部变量访问代码 397
表 14-14 32 位 x86: 局部可执行的线程局部变量访问代码 397
表 14-15 32 位 x86: 局部可执行的线程局部变量访问代码 398
表 14-16 32 位 x86: 局部可执行的线程局部变量访问代码 398
表 14-17 32 位 x86: 线程局部存储的重定位类型 399
表 14-18 x64: 常规动态的线程局部变量访问代码 400
表 14-19 x64: 局部动态的线程局部变量访问代码 400
表 14-20 x64: 初始可执行的线程局部变量访问代码 401
表 14-21 x64: 初始可执行的线程局部变量访问代码 II 402
表 14-22 x64: 局部可执行的线程局部变量访问代码 402
表 14-23 x64: 局部可执行的线程局部变量的访问代码 II 402
表 14-24 x64: 局部可执行的线程局部变量访问代码 III 402
表 14-25 x64: 线程局部存储的重定位类型 403
表 B-1 *mapfile* 段属性 414
表 B-2 节属性 417

使用此文档

- **概述** – 在 Oracle Solaris™ 操作系统 (Oracle Solaris Operating System, Oracle Solaris OS) 中，应用程序开发者可以使用链接编辑器 `ld(1)` 创建应用程序和库，并可以借助于运行时链接程序 `ld.so.1(1)` 执行这些目标文件。

  链接程序和库指南介绍了 Oracle Solaris 链接编辑器和运行时链接程序的操作。由于动态可执行文件和共享目标文件在动态运行时环境中非常重要，因此将重点介绍这两者的生成和用法。

注 - 此 Oracle Solaris™ 发行版支持使用 SPARC® 和 x86 系列处理器体系结构的系统。支持的系统可以在 *Oracle Solaris OS: Hardware Compatibility Lists*（Oracle Solaris OS：硬件兼容性列表）中找到。本文档列举了在不同类型的平台上进行实现时的所有差别。

在本文档中，这些与 x86 相关的术语表示以下含义：

- x86 泛指 64 位和 32 位的 x86 兼容产品系列。
- x64 特指 64 位的 x86 兼容 CPU。
- “32 位 x86”指出了有关基于 x86 的系统的特定 32 位信息。

- **目标读者** – 本指南适用于对 Oracel Solaris 链接编辑器、运行时链接程序以及相关工具感兴趣的程序员（从好学的初学者到高级用户）。
  - 初级程序员可以了解链接编辑器和运行时链接程序的基本操作。
  - 中级程序员可以了解如何创建、使用以及有效地定制库。
  - 高级程序员（例如语言工具开发者）可以学习如何解释和生成目标文件。

  大多数程序员都不需要从头至尾阅读本手册。
- **必备知识** – 本指南的读者应熟悉并能够使用以下技术。
  - UNIX® SVR4 系统 – 最好是当前的 Oracle Solaris™ 发行版。
  - C 编程语言和应用程序开发。

产品文档库

位于 http://www.oracle.com/pls/topic/lookup?ctx=E56344 的文档库中包含此产品的最新信息和已知问题。

获得 Oracle 支持

Oracle 客户可通过 My Oracle Support 获得电子支持。有关信息，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info；如果您听力受损，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs。

反馈

可以在 http://www.oracle.com/goto/docfeedback 上提供有关此文档的反馈。

部分 I

使用链接编辑器和运行时链接程序

♦ ♦ ♦ 第 1 章

1 Oracle Solaris 链接编辑器介绍

本手册介绍了 Oracle Solaris 链接编辑器和运行时链接程序的操作，以及这些实用程序操作的目标文件。Oracle Solaris 链接编辑器和运行时链接程序的基本操作涉及目标文件的组合。此组合会导致从正在连接的目标文件引用另一个目标文件内的符号定义。

本手册对以下内容进行了阐述：

链接编辑器

链接编辑器 ld(1) 对来自一个或多个输入文件的数据进行串联和解释。这些文件可以是可重定位目标文件、共享目标文件或归档库。可以通过这些输入文件创建一个输出文件。此输出文件可以是可重定位目标文件、动态可执行文件或共享目标文件。在编译环境中，最常调用链接编辑器。

运行时链接程序

运行时链接程序 ld.so.1(1) 在运行时处理动态可执行文件和共享目标文件，从而将可执行文件和共享目标文件绑定在一起来创建可运行进程。

共享目标文件

共享目标文件是链接编辑阶段的一种输出形式。共享目标文件有时被称为共享库。共享目标文件在创建强大灵活的运行时环境方面非常重要。

目标文件

Oracle Solaris 链接编辑器、运行时链接程序以及相关工具处理符合可执行链接格式（又称为 ELF）的文件。

尽管可以将这些内容编写为单独的主题，但是它们之间有大量的重叠。在介绍上述每项内容的同时，本文档还将介绍其他相关内容。

链接编辑

链接编辑器 ld(1) 接受各种输入文件，这些文件通常是由编译器或汇编程序生成的文件，或是该链接编辑器之前调用的文件。链接编辑器会串联并解释这些输入文件内的数据，以形成输出文件。生成的输出文件是以下基本类型之一。

- 动态可执行文件 – 要求运行时链接程序 ld.so.1(1) 进行干预以生成可运行进程的输入可重定位目标文件的串联。动态可执行文件通常具有一个或多个以共享目标文件形式存在的依赖项。

  使用 -z type=exec 选项时将创建动态可执行文件，或者未提供控制输出文件类型的其他选项时动态可执行文件是缺省值。
- 与位置无关的可执行文件 – 共享目标文件的特殊情况，指定解释程序。应该基于与位置无关的代码创建与位置无关的可执行文件。与需要固定地址空间以执行操作的动态可执行文件不同，与位置无关的可执行文件可以在由 exec(2) 选择的任意地址处装入。

  使用 -z type=pie 选项时将创建与位置无关的可执行文件。
- 可重定位目标文件 – 可在随后的链接编辑阶段使用的输入可重定位目标文件的串联。

  使用 -z type=reloc 选项或 -r 选项时将创建可重定位目标文件。
- 共享目标文件 – 提供各种服务的输入可重定位目标文件的串联，运行时这些服务可能会绑定到动态可执行文件。应基于与位置无关的代码创建共享目标文件。共享目标文件可能依赖于其他共享目标文件。

  使用 -z type=shared 选项或 -G 选项时将创建共享目标文件。

静态可执行文件

许多发行版都建议不要创建静态可执行文件。实际上，这些版本中从未提供过 64 位系统归档库。因为静态可执行文件是基于系统归档库生成的，所以这种可执行文件包含关于系统实现的详细信息。该自包含特性有许多缺点：

- 静态可执行文件无法利用以共享目标文件形式发布的系统修补程序。因此，必须重新生成静态可执行文件，才能利用众多的系统改进功能。
- 这种可执行文件是否能够在未来的发行版上运行可能会受到影响。
- 系统实现详细信息的重复会对系统性能造成负面影响。

从 Oracle Solaris 10 发行版开始，操作系统不再包含 32 位系统归档库。如果没有这些库，尤其是 libc.a，不具备专业系统知识就无法创建静态可执行文件。请注意，链接编辑器处理静态链接选项的功能以及归档库的处理仍保持不变。

运行时链接

运行时链接涉及绑定目标文件（这些目标文件通常由以前的一个或多个链接编辑过程生成）以生成可运行的进程。在链接编辑器生成这些目标文件的过程中，会生成相应的簿记信息来表示已验证的绑定要求。利用此信息，运行时链接程序可以装入、重定位并完成绑定过程。

在进程执行过程中，运行时链接程序的功能将变为可用。通过在需要时添加附加共享目标文件，这些功能可用于扩展进程的地址空间。运行时链接过程中涉及的两个最常见组件为动态可执行文件和共享目标文件。

动态可执行文件是在运行时链接程序的控制下执行的应用程序。这些应用程序通常具有以共享目标文件（由运行时链接程序定位并绑定来创建可运行进程）形式存在的依赖项。动态可执行文件是链接编辑器生成的缺省输出文件。

共享目标文件向动态链接系统提供主要组成单元。共享目标文件类似于动态可执行文件，但是，系统尚未为其指定虚拟地址。

动态可执行文件通常依赖于一个或多个共享目标文件。通常，必须将一个或多个共享目标文件绑定到动态可执行文件，以生成可运行进程。因为共享目标文件可被许多应用程序使用，所以其构造的各个方面将直接影响可共享性、版本控制和性能。

共享目标文件是由链接编辑器处理还是由运行时链接程序处理，可以通过使用共享目标文件的环境来区分：

编译环境
: 可由链接编辑器处理共享目标文件，以生成动态可执行文件或其他共享目标文件。共享目标文件成为要生成的输出文件的依赖项。

运行时环境
: 共享目标文件可由运行时链接程序处理，并与一个动态可执行文件共同生成可运行进程。

相关主题

动态链接

动态链接通常是涵盖多个链接概念的术语。动态链接是指链接编辑过程中那些生成动态可执行文件和共享目标文件的部分。动态链接还指运行时链接这些目标文件以生成可运行进程。利用动态链接，多个应用程序可以通过在运行时将应用程序绑定到共享目标文件来使用此共享目标文件提供的代码。

通过使应用程序和标准库的服务分开，动态链接还提高了应用程序的可移植性和可扩展性。由于服务接口及其实现也彼此分开，系统可以在维持应用程序稳定性的同时进行演变。动态链接在提供应用程序二进制接口 (Application Binary Interface, ABI) 方面是至关重要的因素，而且是 Oracle Solaris 应用程序的首选编译方法。

应用程序二进制接口

根据其定义，系统组件和应用程序组件之间的二进制接口允许非同步地实现这些功能的改进。Oracle Solaris 链接编辑器和运行时链接程序依靠这些接口来对要执行的应用程序进行汇编。虽然 Oracle Solaris 链接编辑器和运行时链接程序所处理的所有组件都具有二进制接口，但是我们将系统提供的整个接口集合称为 *Oracle Solaris ABI*。

Oracle Solaris ABI 在技术上讲，是从 ABI 功能衍生而来的，这种衍生开始于 *System V* 应用程序二进制接口。ABI 功能因 SPARC International, Inc.® 为其 SPARC 处理器提供了附加特性而得以改进，该特性称为 *SPARC* 符合性定义 (SPARC Compliance Definition, SCD)。

32 位环境和 64 位环境

链接编辑器以 32 位应用程序和 64 位应用程序的形式提供。每种链接编辑器都可以对 32 位目标文件和 64 位目标文件执行操作。在运行 64 位环境的系统上，两个版本的链接编辑器都可以运行。在运行 32 位环境的系统上，只能运行链接编辑器的 32 位版本。

运行时链接程序以 32 位目标文件和 64 位目标文件的形式提供。32 位目标文件用于执行 32 位进程，而 64 位目标文件用于执行 64 位进程。

链接编辑器和运行时链接程序对 32 位目标文件和 64 位目标文件的操作相同。本文档一般使用 32 位示例。对于 64 位处理与 32 位处理不同的情况，将明确指出。

环境变量

链接编辑器和运行时链接程序支持许多以字符 `LD_` 开头的环境变量，例如，`LD_LIBRARY_PATH`。每个环境变量都可以其通用形式存在，也可以指定 `_32` 或 `_64` 后缀，例如，`LD_LIBRARY_PATH_64`。此后缀使环境变量分别特定于 32 位或 64 位进程。此后缀还将覆盖任何可能有效的通用无后缀环境变量版本。

注 - 在 Oracle Solaris 10 发行版之前，链接编辑器和运行时链接程序会忽略未指定值的环境变量。因此，在以下示例中，将使用通用环境变量设置 `/opt/lib` 来搜索 32 位应用程序 `prog` 的依赖项。

```
$ LD_LIBRARY_PATH=/opt/lib  LD_LIBRARY_PATH_32=  prog
```

从 Oracle Solaris 10 发行版开始，可以处理未指定值的带有 `_32` 或 `_64` 后缀的环境变量。这些环境变量将有效地取消任何关联的通用环境变量设置。因此在前面的示例中，将不会使用 `/opt/lib` 来搜索 32 位应用程序 `prog` 的依赖项。

在本文档中，任何对链接编辑器环境变量的引用都使用通用的无后缀变体。所有支持的环境变量均在 `ld(1)` 和 `ld.so.1(1)` 中定义。

支持工具

Oracle Solaris 操作系统还提供了几个支持工具和库。可以使用这些工具分析和检查这些目标文件和链接进程。这些工具包括 elfdump(1)、lari(1)、nm(1)、dump(1)、ldd(1)、pvs(1)、elf(3ELF)，以及一个链接程序调试支持库。在本文档中，许多讨论都含有这些工具的示例。

♦ ♦ ♦ 第 2 章

2 链接编辑器

链接编辑过程根据一个或多个输入文件创建输出文件。输出文件的创建由提供给链接编辑器的选项和输入文件提供的输入节控制。

所有文件都是以可执行链接格式 (ELF) 表示的。有关 ELF 格式的完整说明，请参见第 12 章 目标文件格式。为介绍该文件格式，需要先介绍两种 ELF 结构：节和段。

节是 ELF 文件中可以处理的不可分割的最小单元。段是节的集合，代表可通过 exec(2) 或运行时链接程序 ld.so.1(1) 映射到内存映像的最小独立单元。

虽然存在许多类型的 ELF 节，但就链接编辑阶段而言，所有节都可归为两种类别。

- 包含程序数据的节，程序数据的解释仅对应用程序有意义，这类数据包括程序指令 .text 以及关联的数据 .data 和 .bss 等等。
- 包含链接编辑信息的节，例如 .symtab 和 .strtab 中包含符号表信息，.rela.text 等节中包含重定位信息。

本质上，链接编辑器将程序数据节串联成输出文件。链接编辑器将解释链接编辑信息节，以便修改其他节。信息节还用于生成在后期处理输出文件时使用的新输出信息节。

以下对链接编辑器功能的简单细分介绍了本章中涵盖的主题。

- 对提供的所有选项进行验证和一致性检查。
- 将输入可重定位目标文件中具有相同特征的节串联起来，以在输出文件中形成新的节。串联的节又可与输出段关联。
- 处理可重定位目标文件和共享目标文件中的符号表信息，以便验证并将引用和其定义合并起来。在输出文件中生成新的符号表。
- 处理输入可重定位目标文件中的重定位信息，并将此信息应用于构成输出文件的节。此外，还可以生成输出重定位节以供运行时链接程序使用。
- 生成用于描述所创建的所有段的程序头。
- 必要时生成动态链接信息节，这些节为运行时链接程序提供信息，如共享目标文件依赖项和符号绑定。

将类似的节串联起来以及将节关联到段的处理是在链接编辑器中使用缺省信息完成的。对于大多数链接编辑操作来说，链接编辑器提供的缺省节和段处理通常已满足要求。

不过，可将 -M 选项与关联的 mapfile 配合使用来处理这些缺省行为。请参见附录 B, *System V 发行版 4（版本 1）mapfile*。

调用链接编辑器

可以从命令行直接运行链接编辑器，也可以让编译器驱动程序调用链接编辑器。以下两小节详细介绍了这两种方法。但是，首选使用编译器驱动程序。编译环境通常是复杂且有时会发生变化的一系列操作（仅对编译器驱动程序可识别）。

注 - 从 Oracle Solaris 11 开始，各种编译组件已从 /usr/ccs/bin 和 /usr/ccs/lib 移动到 /usr/bin 和 /usr/lib。但是，仍有引用原始 ccs 名称的应用程序。符号链接用于保持兼容性。

直接调用

直接调用链接编辑器时，必须提供创建预期输出所需的每个目标文件和库。链接编辑器对创建输出时要使用的目标文件模块或库不做任何假设。例如，以下命令将指示链接编辑器仅使用输入文件 test.o 创建名为 a.out 的动态可执行文件。

```
$ ld test.o
```

通常，动态可执行文件需要专用的启动代码和退出处理代码。此代码可能特定于语言或操作系统，并且通常通过编译器驱动程序提供的文件提供。

此外，您还可以提供自己的初始化代码和终止代码。必须正确封装和标记此代码，以便运行时链接程序可以正确识别并使用代码。也可以通过编译器驱动程序提供的文件提供此封装和标记。

创建运行时目标文件（如可执行文件和共享目标文件）时，应使用编译器驱动程序来调用链接编辑器。建议仅在使用 -r 选项创建中间可重定位目标文件时直接调用链接编辑器。

使用编译器驱动程序

通常通过特定于语言的编译器驱动程序来使用链接编辑器。需要为编译器驱动程序 cc(1), CC(1) 等提供组成应用程序的输入文件。编译器驱动程序将添加其他文件和缺省库以完成链接编辑。展开编译调用可以看到这些附加的文件。

```
$ cc -# -o prog main.o
/usr/bin/ld -dy /opt/COMPILER/crti.o /opt/COMPILER/crt1.o \
```

```
/usr/lib/values-Xt.o -o prog main.o \
-YP,/opt/COMPILER/lib:/lib:/usr/lib -Qy -lc \
/opt/COMPILER/crtn.o
```

注 - 编译器驱动程序包括的实际文件和用于显示链接编辑器调用的机制可能有所不同。

跨链接编辑

对于 SPARC 或 x86 目标，链接编辑器是一个跨链接编辑器，可以链接 32 位目标文件或 64 位目标文件。不允许混用 32 位目标文件和 64 位目标文件。同样，只允许单个计算机类型的目标文件。

通常，不需要使用命令行选项来区分链接编辑目标。链接编辑器使用命令行中第一个可重定位目标文件的 ELF 计算机类型来控制操作的模式。专用链接编辑（例如，完全来自 `mapfile` 或归档库的链接）不受命令行目标文件影响。这些链接编辑缺省为 32 位本机目标。要显式定义链接编辑目标，可使用 `-z target` 选项。

指定链接编辑器选项

通常，使用命令行选项充分指定链接编辑。但是，提供了各种环境变量来扩充命令行处理。这些变量提供可能与编译器选项冲突的选项。这些变量还用于覆盖或取消设置嵌入脚本和构建环境的命令行选项。

命令行选项之间的任何不一致都将导致致命错误状态。与环境变量提供的选项有关的任何不一致都将导致警告，第一个选项优先。任何 `UNSET` 操作都会伴随有警告通知。

根据环境和命令行按以下顺序解释各个选项。

- 从 `LD_OPTIONS` 环境变量。
- 从命令行。
- 从 `LD_UNSET` 环境变量。

可以使用 `LD_OPTIONS` 将本该由编译器驱动程序解释的参数传递到链接编辑器。例如，可使用 `-D` 选项进行与链接编辑相关的诊断。此选项通常由编译器预处理程序解释。

```
$ LD_OPTIONS=-Dargs  cc -o main main.c
...
debug: arg[0]  option=-D:  option-argument: args  (LD_OPTIONS)
debug:
debug: arg[0]  /usr/ccs/bin/ld
debug: arg[2]  option=-o:  option-argument: main
debug: arg[3]  option=-Q:  option-argument: y
debug: arg[4]  option=-l:  option-argument: c
```

LD_OPTIONS 也可用于覆盖具有很多变体的选项。例如，嵌入式 -z text 选项可用 -z textoff 选项覆盖。

```
$ LD_OPTIONS=-ztextoff  cc -ztext -G null.o
ld: warning: option '-ztextoff' and option '-ztext' are incompatible, \
    first option taken
```

某些选项没有替代变体，因此不可被覆盖。但可以取消设置这些选项。例如，标准的链接编辑可以创建以下节。

```
$ cc -o main main.c
$ elfdump -c main | egrep "symtab|debug"
Section Header[19]:  sh_name: .symtab
Section Header[22]:  sh_name: .debug_info
Section Header[23]:  sh_name: .debug_line
```

可以使用 -z strip-class 选项删除这些节。

```
$ cc -o main -zstrip-class=symbol -zstrip-class=debug main.c
$ elfdump -c main | egrep "symtab|debug"
$
```

可以单独取消设置各个剥离选项。以下是取消设置剥离调试节示例的示例。

```
$ LD_UNSET=-zstrip-class=debug  cc -o main -zstrip-class=symbol \
    -zstrip-class=debug main.c
ld: warning: unsetting option '-zstrip-class=debug': LD_UNSET directed
$ elfdump -c main | egrep "symtab|debug"
Section Header[20]:  sh_name: .debug_info
Section Header[21]:  sh_name: .debug_line
```

此外，对于适用于多个实例的选项（例如 -z strip-class），可以通过指定不含任何合格选项字符串的选项来取消设置所有系列成员。以下是取消设置剥离调试节和符号表节的示例。

```
$ LD_UNSET=-zstrip-class  cc -o main -zstrip-class=symbol \
    -zstrip-class=debug main.c
ld: warning: unsetting option '-zstrip-class': LD_UNSET directed
$ elfdump -c main | egrep "symtab|debug"
Section Header[19]:  sh_name: .symtab
Section Header[22]:  sh_name: .debug_info
Section Header[23]:  sh_name: .debug_line
```

根据 LD_OPTIONS、命令行和 LD_UNSET 组件确定输出目标文件类型。然后使用此目标文件类型来调查任何 LD_{object-type}_UNSET 和 LD_{object-type}_OPTIONS 环境变量，以删除或添加特定于正在构建的目标文件类型的选项。object-type 以大写形式提供 -z type 选项定义的类型，此目标文件类型为 EXEC、PIE、RELOC 或 SHARED 之一。例如，输出文件类型是动态可执行文件时，将解释 LD_EXEC_OPTIONS 选项。按以下顺序处理这些环境变量。

- 从 LD_{object-type}_UNSET 环境变量。

- 从 LD_{object-type}_OPTIONS 环境变量。

例如，创建动态可执行文件和共享目标文件的构建环境可以使用 LD_EXEC_OPTIONS 环境变量针对所有动态可执行文件启用地址空间布局随机化。

```
$ LD_EXEC_OPTIONS=-zaslr  build.sh
```

与此输出目标文件类型不一致的任何命令行选项都将导致致命错误状态。环境变量提供的任何不一致选项都将导致警告，并将忽略该选项。

有关最常用的链接编辑器选项，请参见第 5 章 链接编辑器快速参考；有关所有链接编辑器选项的完整说明，请参见 ld(1)。

输入文件处理

链接编辑器按输入文件在命令行中出现的顺序读取这些文件。将打开并检查每个文件以确定文件的 ELF 类型，从而确定文件的处理方式。作为链接编辑的输入应用的文件类型由链接编辑的绑定模式（静态或动态）确定。

在静态模式下，链接编辑器仅接受可重定位目标文件或归档库作为输入文件。在动态模式下，链接编辑器还接受共享目标文件。

对于链接编辑过程，可重定位目标文件是最基本的输入文件类型。这些文件中的程序数据节将串联成要生成的输出文件映像。组织链接编辑信息节供以后使用。这些节不会成为输出文件映像的一部分，因为将生成新的信息节替代它们。符号将被收集到内部符号表中以进行验证和解析。然后，使用此表在输出映像中创建一个或多个符号表。

虽然可以在链接编辑命令行中直接指定输入文件，但通常使用 -l 选项指定归档库和共享目标文件。请参见“与其他库链接” [29]。在链接编辑期间，归档库和共享目标文件的解释完全不同。下面两小节详细说明了这些差别。

归档处理

使用 ar(1) 生成归档。归档通常由一组可重定位目标文件和一个归档符号表组成。该符号表提供符号定义与提供这些定义的目标文件之间的关联。缺省情况下，链接编辑器有选择地提取归档成员。链接编辑器使用未解析的符号引用从归档中选择完成绑定过程所需的目标文件。也可以显式提取归档的所有成员。

在以下情况下，链接编辑器从归档中提取可重定位目标文件。

- 归档成员包含满足符号引用的符号定义，这些符号定义目前保存在链接编辑器的内部符号表中。此引用有时称为未定义符号。

- 归档成员包含满足暂定 (tentative) 符号定义（目前保存在链接编辑器的内部符号表中）的数据符号定义。例如，FORTRAN COMMON 块定义，它将导致提取定义了相同 DATA 符号的可重定位目标文件。
- 归档成员包含的某个符号定义与需要隐藏可见性或受保护可见性的一个引用相匹配。请参见表 12-23 “ELF 符号可见性”。
- 链接编辑器 -z allextract 有效。此选项会暂停选择性归档提取，并导致从正在处理的归档中提取所有归档成员。

有选择地提取归档时，除非 -z weakextract 选项有效，否则弱符号引用不会从归档中提取目标文件。有关更多信息，请参见“简单解析” [37]。

注 - 使用选项 -z weakextract、-z allextract 和 -z defaultextract 可以在多个归档之间切换归档提取机制。

在有选择地提取归档的情况下，链接编辑器会对整个归档进行多遍检查。将根据需要提取可重定位目标文件，以满足链接编辑器内部符号表中累积的符号信息。链接编辑器检查完归档一遍（但未提取任何可重定位目标文件）之后，将处理下一个输入文件。

由于遇到归档时仅从归档中提取需要的可重定位目标文件，因此命令行中归档的位置可能很重要。请参见“命令行中归档的位置” [30]。

注 - 虽然链接编辑器会通过对整个归档进行多遍检查来解析符号，但这种机制的开销会很大。对于包含随机组织的可重定位目标文件的大型归档，更是如此。在这些情况下，应使用 lorder(1) 和 tsort(1) 之类的工具对归档文件中的可重定位目标文件进行排序。该排序操作可减少链接编辑器必须检查归档的遍数。

共享目标文件处理

共享目标文件是一个或多个输入文件的上一次链接编辑生成的不可分割完整单元。链接编辑器处理共享目标文件时，共享目标文件的所有内容将成为生成的输出文件映像的逻辑部分。包含此逻辑部分意味着，链接编辑过程可以使用在共享目标文件中定义的所有符号项。

链接编辑器不使用共享目标文件的程序数据节和大多数链接编辑信息节。绑定共享目标文件以生成可运行的进程时，运行时链接程序将解释这些节。但是，会记录出现的共享目标文件。信息存储在输出文件映像中，以表明此目标文件是运行时必须使用的依赖项。

缺省情况下，链接编辑过程中指定的所有共享目标文件在要生成的目标文件中都记录为依赖项。无论要生成的目标文件实际上是否引用共享目标文件提供的符号，都会进行此记录。为了最大程度地降低运行时链接的开销，请仅指定将解析所生成目标文件中的符号引用的那些依赖项。可以使用链接编辑器的调试功能和带有 -u 选项的 ldd(1) 确定未

使用的依赖项。可以使用链接编辑器的 -z discard-unused=dependencies 选项来禁止任何未使用的共享目标文件的依赖项记录。另请参见“删除未使用的依赖项” [165]。

如果某个共享目标文件依赖于其他共享目标文件，则也会处理这些依赖项。处理完所有命令行输入文件后将进行此处理，以完成符号解析过程。不过，在要生成的输出文件映像中，不会将共享目标文件名称作为依赖项进行记录。

虽然在命令行上共享目标文件的位置没有在归档处理中那么重要，但是该位置可能会影响全局。可重定位目标文件与共享目标文件之间以及多个共享目标文件之间允许存在多个名称相同的符号。请参见“符号解析” [36]。

链接编辑器处理共享目标文件的顺序由存储在输出文件映像中的依赖项信息维护。在无延迟装入的情况下，运行时链接程序按相同的顺序装入指定的共享目标文件。因此，链接编辑器和运行时链接程序选择多重定义的一系列符号中第一次出现的某个符号。

注 - 多个符号定义在使用 -m 选项生成的装入映射输出中报告。

与其他库链接

虽然编译器驱动程序通常会确保对链接编辑器指定适当的库，但您必须经常提供自己的库。通过显式指定链接编辑器需要的输入文件可以指定共享目标文件和归档。但是，更常见且更灵活的方法涉及使用链接编辑器的 -l 选项。

库命名约定

根据约定，共享目标文件通常通过前缀 lib 和后缀 .so 指明。归档指定有前缀 lib 和后缀 .a。例如，libfoo.so 是共享目标文件版的 "*foo*" 实现，可用于编译环境。libfoo.a 是库的归档版本。

这些约定可由链接编辑器的 -l 选项识别。此选项通常用于为链接编辑提供其他库。以下示例指示链接编辑器搜索 libfoo.so。如果链接编辑器未找到 libfoo.so，则在继续搜索下一个目录之前将搜索 libfoo.a。

```
$ cc -o prog file1.c file2.c -lfoo
```

注 - 在编译环境和运行时环境中使用的共享目标文件分别遵循相应的命名约定。编译环境使用简单 .so 后缀，而运行时环境通常使用带有附加版本号的后缀。请参见“命名约定” [123]和“协调版本化文件名” [224]。

以动态模式进行链接编辑时，可以选择同时链接共享目标文件和归档。以静态模式进行链接编辑，仅接受归档库作为输入。

在动态模式下使用 -l 选项时，链接编辑器首先搜索给定目录以查找与指定名称匹配的共享目标文件。如果未找到任何匹配项，链接编辑器将在相同目录中查找归档库。在静态模式下使用 -l 选项时，将仅查找归档库。

同时链接共享目标文件和归档

动态模式下的库搜索机制会在给定目录中搜索共享目标文件，然后搜索归档库。使用 -B 选项可以更精确地控制搜索。

通过在命令行中指定 -B dynamic 和 -B static 选项，可以分别在共享目标文件或归档之间切换库搜索。例如，要将应用程序与归档 libfoo.a 和共享目标文件 libbar.so 链接，可发出以下命令。

```
$ cc -o prog main.o file1.c -Bstatic -lfoo -Bdynamic -lbar
```

-B static 和 -B dynamic 选项并不完全对称。指定 -B static 时，链接编辑器要等到下一次出现 -B dynamic 时才接受共享目标文件作为输入。但是，指定 -B dynamic 时，链接编辑器首先在任何给定目录中查找共享目标文件，然后查找归档库。

对上一个示例的准确说明如下：链接编辑器首先搜索 libfoo.a，然后链接编辑器将搜索 libbar.so，如果此搜索失败，则搜索 libbar.a。

命令行中归档的位置

命令行中归档的位置可以影响要生成的输出文件。链接编辑器搜索归档只是为了解析先前遇到过的未定义或暂定的外部引用。完成此搜索并提取所有需要的成员后，链接编辑器将继续处理命令行中的下一个输入文件。

因此，缺省情况下，不能使用归档来解析命令行中归档后面的输入文件中的任何新引用。例如，以下命令指示链接编辑器只有在解析从 file1.c 中获取的符号引用时才搜索 libfoo.a。不能使用 libfoo.a 归档来解析 file2.c 或 file3.c 中的符号引用。

```
$ cc -o prog file1.c -Bstatic -lfoo file2.c file3.c -Bdynamic
```

归档之间可以存在相互的依赖性，这样，要从一个归档中提取成员，还必须从另一个归档中提取相应成员。如果这些依赖性构成循环，则必须在命令行中重复指定归档以满足前面的引用。

```
$ cc -o prog .... -lA -lB -lC -lA -lB -lC -lA
```

确定和维护重复指定的归档是一项非常繁琐的任务。-z rescan-now 选项可以简化此过程。在命令行中遇到 -z rescan-now 选项时，链接编辑器会立即处理该选项。命令行中该选项之前的所有已处理归档都将立即重新处理。此处理过程会尝试查找可解析符号引用的其他归档成员。继续重新扫描此归档，直到扫描归档列表一遍但未提取到任何新成员为止。因此，上一个示例可进行如下简化：

```
$ cc -o prog .... -lA -lB -lC -z rescan-now
```

另外，还可以使用 -z rescan-start 和 -z rescan-end 选项将相互依赖的归档一起划分到一个归档组中。链接编辑器在命令行中遇到结束分隔符时，会立即重新处理这些组。在组中找到的归档会重新进行处理，以尝试查找可解析符号引用的其他归档成员。继续重新扫描此归档，直到扫描归档组一遍但未提取到任何新成员为止。如果使用归档组，上一个示例的编写如下：

```
$ cc -o prog .... -z rescan-start -lA -lB -lC -z rescan-end
```

注 - 应当在命令行的末尾指定任何归档，除非多重定义冲突要求采用其他方式。

链接编辑器搜索的目录

上述所有示例都假定链接编辑器知道在哪里搜索命令行中列出的库。缺省情况下，在链接 32 位目标文件时，链接编辑器只知道在两个标准目录中查找库：先搜索 /lib，再搜索 /usr/lib。在链接 64 位目标文件时，只使用两个标准目录：先搜索 /lib/64，再搜索 /usr/lib/64。必须显式地将要搜索的所有其他目录添加到链接编辑器的搜索路径中。

可以使用命令行选项或环境变量更改链接编辑器的搜索路径。

使用命令行选项

可以使用 -L 选项将新的路径名添加到库搜索路径中。在命令行中遇到此选项时，此选项将改变搜索路径。例如，以下命令将先搜索 path1，再搜索 /lib，最后搜索 /usr/lib，以查找 libfoo。此命令先搜索 path1，再搜索 path2，接着搜索 /lib 和 /usr/lib，以查找 libbar。

```
$ cc -o prog main.o -Lpath1 file1.c -lfoo file2.c -Lpath2 -lbar
```

使用 -L 选项定义的路径名仅由链接编辑器使用。这些路径名不会记录在要创建的输出文件映像中。因此，运行时链接程序不能使用这些路径名。

注 - 如果希望链接编辑器在当前目录中搜索库，必须指定 -L。可以使用句点 (.) 表示当前目录。

可以使用 -Y 选项更改链接编辑器搜索的缺省目录。随此选项提供的参数采用以冒号分隔的目录列表形式。例如，以下命令仅在 /opt/COMPILER/lib 和 /home/me/lib 目录中搜索 libfoo。

```
$ cc -o prog main.c -YP,/opt/COMPILER/lib:/home/me/lib -lfoo
```

可以使用 -L 选项补充通过使用 -Y 选项指定的目录。编译器驱动程序通常使用 -Y 选项提供特定于编译器的搜索路径。

使用环境变量

还可以使用环境变量 LD_LIBRARY_PATH 来添加链接编辑器的库搜索路径。通常，LD_LIBRARY_PATH 采用冒号分隔的目录列表形式。LD_LIBRARY_PATH 最常见的形式是以分号分隔的两个目录列表。在命令行中提供的 -Y 列表之前和之后搜索这些列表。

以下示例说明在同时设置了 LD_LIBRARY_PATH 且使用多个 -L 调用链接编辑器的情况下所得到的结果。

```
$ LD_LIBRARY_PATH=dir1:dir2;dir3
$ export LD_LIBRARY_PATH
$ cc -o prog main.c -Lpath1 .... -Lpath2 .... -Lpathn -lfoo
```

有效搜索路径为 dir1:dir2:path1:path2:....:pathn:dir3:/lib:/usr/lib。

如果在 LD_LIBRARY_PATH 定义中未指定分号，那么将在解释所有 -L 选项之后解释指定的目录列表。在以下示例中，有效搜索路径为 path1:path2:....:pathn:dir1:dir2:/lib:/usr/lib。

```
$ LD_LIBRARY_PATH=dir1:dir2
$ export LD_LIBRARY_PATH
$ cc -o prog main.c -Lpath1 .... -Lpath2 .... -Lpathn -lfoo
```

注 - 此环境变量还可用于扩充运行时链接程序的搜索路径。请参见“运行时链接程序搜索的目录” [88]。要防止此环境变量影响链接编辑器，请使用 -i 选项。

运行时链接程序搜索的目录

运行时链接程序在两个缺省位置中查找依赖项。在处理 32 位目标文件时，缺省位置为 /lib 和 /usr/lib。在处理 64 位目标文件时，缺省位置为 /lib/64 和 /usr/lib/64。其他所有要搜索的目录必须显式添加到运行时链接程序的搜索路径中。

动态可执行文件或共享目标文件与其他共享目标文件链接时，这些共享目标文件将被记录为依赖项。运行时链接程序执行进程期间必须找到这些依赖项。链接动态目标文件时，可以在输出文件中记录一个或多个搜索路径。这些搜索路径称为运行路径。运行时链接程序使用目标文件的运行路径查找该目标文件的依赖项。

可以使用 -z nodefaultlib 选项生成专用目标文件，以便在运行时不搜索任何缺省位置。此选项的用法表示可以使用目标文件的运行路径来查找该目标文件的所有依赖项。如果不使用此选项，则无论如何扩充运行时链接程序的搜索路径，所用的最后一个搜索路径始终是缺省位置。

注 - 可以使用运行时配置文件管理缺省搜索路径。请参见“配置缺省搜索路径” [90]。但是，动态目标文件的创建者不应依赖于此文件的存在。应始终确保目标文件仅使用其运行路径或缺省位置即可找到其依赖项。

可以使用 -R 选项（采用冒号分隔的目录列表形式）将运行路径记录在动态可执行文件或共享目标文件中。以下示例将运行路径 /home/me/lib:/home/you/lib 记录在动态可执行文件 prog 中。

```
$ cc -o prog main.c -R/home/me/lib:/home/you/lib -Lpath1 \
    -Lpath2 file1.c file2.c -lfoo -lbar
```

运行时链接程序使用这些路径（后接缺省位置）来获取任意共享目标文件依赖项。在本例中，此运行路径用于查找 libfoo.so.1 和 libbar.so.1。

链接编辑器接受多个 -R 选项。指定的多个选项串联在一起，用冒号分隔。因此，上一个示例还可采用如下表示方式。

```
$ cc -o prog main.c -R/home/me/lib -Lpath1 -R/home/you/lib \
    -Lpath2 file1.c file2.c -lfoo -lbar
```

对于可以安装在各种位置的目标文件，$ORIGIN 动态字符串标记提供了一种记录运行路径的灵活方法。请参见“查找关联的依赖项” [231]。

注 - 以前的一种指定 -R 选项的替代方法是设置环境变量 LD_RUN_PATH，并使链接编辑器可以使用此环境变量。LD_RUN_PATH 和 -R 的作用域和功能完全相同，但如果同时指定了这两者，则 -R 会取代 LD_RUN_PATH。

初始化节和终止节

动态目标文件可以提供用于运行时初始化和终止处理的代码。每次在进程中装入动态目标文件时，都会执行一次动态目标文件的初始化代码。每次在进程中卸载动态目标文件或进程终止时，都会执行一次动态目标文件的终止代码。可以将此代码封装在以下两种节类型的任意一种中：函数指针数组或单个代码块。这两种节类型都是通过串联输入可重定位目标文件中的相似节生成的。

.pre_initarray、.init_array 和 .fini_array 节分别提供运行时预初始化、初始化和终止函数的数组。创建动态目标文件时，链接编辑器相应地使用 .dynamic 标记对（DT_PREINIT_[ARRAY/ARRAYSZ]、DT_INIT_[ARRAY/ARRAYSZ] 和 DT_FINI_[ARRAY/ARRAYSZ]）来标识这些数组。这些标记标识关联的节，以便运行时链接程序可以调用这些节。预初始化数组仅适用于动态可执行文件。

注 - 指定给这些数组的函数必须从要生成的目标文件提供。

.init 节和 .fini 节分别提供运行时初始化代码块和终止代码块。编译器驱动程序通常向 .init 和 .fini 节提供其添加到输入文件列表开头和末尾的文件。编译器提供这些文件的作用相当于将可重定位目标文件中的 .init 和 .fini 代码封装到各个函数中。这些函数分别由保留的符号名称 _init 和 _fini 予以标识。创建动态目标文件时，链接编辑

器相应地使用 .dynamic 标记 DT_INIT 和 DT_FINI 来标识这些符号。这些标记标识关联的节，以便运行时链接程序可以调用这些节。

有关运行时执行初始化和终止代码的更多信息，请参见“初始化和终止例程” [100]。

链接编辑器可以使用 -z initarray 和 -z finiarray 选项直接注册初始化函数和终止函数。例如，以下命令将 foo（） 的地址放置在 .init_array 元素中，并将 bar（） 的地址放置在 .fini_array 元素中。

```
$ cat main.c
#include    <stdio.h>

void foo()
{
        (void) printf("initializing: foo()\n");
}

void bar()
{
        (void) printf("finalizing: bar()\n");
}

void main()
{
        (void) printf("main()\n");
}
$ cc -o main -zinitarray=foo -zfiniarray=bar main.c
$ main
initializing: foo()
main()
finalizing: bar()
```

可以使用汇编程序直接创建初始化节和终止节。但是，大多数编译器提供特殊基元来简化其声明。例如，可以使用以下 #pragma 定义重新编写上面的代码示例。这些定义导致在 .init 节中调用 foo（） ，并在 .fini 节中调用 bar（） 。

```
$ cat main.c
#include    <stdio.h>

#pragma init (foo)
#pragma fini (bar)

....
$ cc -o main main.c
$ main
initializing: foo()
main()
finalizing: bar()
```

分布在几个可重定位目标文件中的初始化和终止代码包含在归档库或共享目标文件中时，可以产生不同的行为。对使用此归档的应用程序进行链接编辑时，可能仅提取此归档中包含的部分目标文件。这些目标文件可能仅提供分布在归档成员中的部分初始化和

终止代码。在运行时仅执行此部分代码。在运行时装入依赖项时，针对共享目标文件生成的相同应用程序将执行所有累积的初始化和终止代码。

在运行时确定进程中执行初始化和终止代码的顺序是一个很复杂的问题，需要进行依赖性分析。限制初始化和终止代码的内容可简化此分析过程。简化的自包含初始化和终止代码可提供可预测的运行时行为。有关更多详细信息，请参见“初始化和终止顺序” [101]。

如果可以使用 dldump(3C) 转储其内存的动态目标文件涉及到初始化代码，那么数据初始化应该是一个独立的过程。

符号处理

可以将符号归类为局部符号或全局符号。请参见“符号可见性” [35]。

在处理输入文件期间，会将局部符号从任何输入可重定位目标文件复制到要生成的输出目标文件中，且不进行检查。

在称为符号解析的过程中，将分析并组合所有输入可重定位目标文件中的全局符号和任何外部依赖项中的全局符号。链接编辑器按照遇到符号的顺序将每个符号放入一个内部符号表中。如果某个同名符号是由早期目标文件提供的，并且已经存在于符号表中，则符号解析过程将确定保留两个符号中的哪个。作为该过程的连带效果，链接编辑器可确定如何建立对外部目标文件依赖项的引用。

成功完成输入文件处理后，链接编辑器将应用任何符号可见性调整，并决定是否保留任何未解析的符号引用。如果发生了任何致命符号解析错误，或保留了任何未解析的符号引用，链接编辑会终止。最后，将链接编辑器的内部符号表添加到要创建的映像的符号表中。

以下各节详细说明了符号可见性、符号解析和未定义符号的处理过程。

符号可见性

可以将符号归类为局部符号或全局符号。除了包含符号定义的目标文件以外，无法从其他目标文件中引用局部符号。缺省情况下，会将局部符号从任何输入可重定位目标文件复制到要生成的输出目标文件中。可以从输出目标文件中删除局部符号。请参见“删除符号” [49]。

除了包含符号定义的目标文件外，还可以从其他目标文件引用全局符号。收集并解析后，全局符号将添加到要在输出目标文件中创建的符号表中。尽管所有的全局符号是一起处理和解析的，但可以调整其最终可见性。全局符号可以定义其他可见性属性。请参见表 12-23 “ELF 符号可见性”。此外，在链接编辑期间可以使用 mapfile 符号指令指定

符号可见性。请参见表 8-8 “符号作用域类型”。这些可见性属性和指令可能会导致全局符号在写入输出目标文件中时调整其可见性。

创建可重定位目标文件时，会在输出目标文件中记录所有可见性属性和指令。但是，不会应用这些属性隐含的可见性更改。相反，任何可见性处理将延迟到读取这些目标文件作为输入的动态目标文件的后续链接编辑。在特殊情况下，可以使用 `-B reduce` 选项强制立即解释可见性属性或指令。

创建动态可执行文件或共享目标文件时，符号可见性属性和指令在将符号写入到任何符号表前应用。可见性属性可确保符号保持为全局符号，且不受任何符号缩减技术影响。可见性属性和指令还可导致全局符号降级为局部符号。后一种技术最常用于显式定义目标文件导出接口。请参见“缩减符号作用域” [46]。

符号解析

符号解析的方式很广，有简单直观的，也有错综复杂的。大多数解析由链接编辑器执行，且没有任何提示。但是，某些重定位可能伴随有警告诊断，而某些可能导致致命错误状态。

最常见的简单解析方式需要将一个目标文件中的符号引用与另一个目标文件中的符号定义绑定起来。此种绑定可用于两个可重定位目标文件之间，也可用于一个可重定位目标文件与在共享目标文件依赖项中找到的第一个定义之间。复杂解析方式通常用于两个或多个可重定位目标文件之间。

这两种符号解析方式取决于符号的属性、提供符号的文件类型以及要生成的文件类型。有关符号属性的完整说明，请参见“符号表节” [326]。但是，对于以下论述，标识了三种基本符号类型。

- 未定义 – 文件中已引用但尚未指定存储地址的符号。
- 暂定 – 已在文件中创建但尚未指定大小或分配存储空间的符号。这些符号在文件中显示为未初始化的 C 语言符号或 `FORTRAN COMMON` 块。
- 已定义 – 已在文件中创建并且分配了存储地址和空间的符号。

形式最简单的符号解析需要使用优先级关系。此关系中，已定义符号优先于暂定符号，而暂定符号又优先于未定义符号。

以下 C 代码示例说明如何生成这些符号类型。未定义符号使用 `u_` 作为前缀。暂定符号使用 `t_` 作为前缀。已定义符号使用 `d_` 作为前缀。

```
$ cat main.c
extern int u_bar;
extern int u_foo();

int t_bar;
int d_bar = 1;

int d_foo()
```

```
{
        return (u_foo(u_bar, t_bar, d_bar));
}
$ cc -o main.o -c main.c
$ elfdump -s main.o

Symbol Table Section:  .symtab
    index  value   size  type bind oth ver shndx          name
    ....
      [7]      0      0  FUNC GLOB  D    0 UNDEF          u_foo
      [8]   0x10   0x40  FUNC GLOB  D    0 .text          d_foo
      [9]    0x4    0x4  OBJT GLOB  D    0 COMMON         t_bar
     [10]      0    0x4  NOTY GLOB  D    0 UNDEF          u_bar
     [11]      0    0x4  OBJT GLOB  D    0 .data          d_bar
```

简单解析

简单符号解析是迄今最常见的解析方式。在这种情况下，将检测两个具有类似特征的符号，一个符号优先于另一个符号。此符号解析由链接编辑器执行，且没有任何提示。例如，对于具有相同绑定的符号，一个文件中的符号引用将绑定到另一个文件中的已定义或暂定符号定义。或者，一个文件中的暂定符号定义将绑定到另一个文件中的已定义符号定义。此种解析方式可用于两个可重定位目标文件之间，也可用于一个可重定位目标文件与在共享目标文件依赖项中找到的第一个定义之间。

要解析的符号可以具有全局绑定或弱绑定。处理可重定位目标文件时，弱绑定的优先级低于全局绑定。弱符号定义将在不给出任何提示的情况下被同一名称的全局定义所覆盖。

在可重定位目标文件与共享目标文件之间或者多个共享目标文件之间，还有一种简单符号解析方式，即插入。在这些情况下，如果多重定义了某个符号，链接编辑器会采用可重定位目标文件或多个共享目标文件之间的第一个定义，且不作任何提示。可重定位目标文件的定义或第一个共享目标文件的定义会在其他所有定义上插入。这种插入可用于覆盖其他共享目标文件提供的功能。可重定位目标文件与共享目标文件之间或多个共享目标文件之间的多重定义符号采用同样的处理方式。符号是弱绑定还是全局绑定无关紧要。无论哪种符号绑定，链接编辑器和运行时链接程序都行为一致，将其解析为第一个定义。

使用链接编辑器的 `-m` 选项可将所有插入的符号引用的列表和节装入地址信息写入到标准输出中。

复杂解析

如果发现两个符号的名称相同，但属性不同，则将进行复杂解析。在这些情况下，链接编辑器将选择最适合的符号同时生成一条警告消息。此消息指出符号、发生冲突的属性以及包含符号定义的文件的标识。在以下示例中，包含数据项数组定义的两个文件有不同的大小要求。

```
$ cat foo.c
int array[1];
$ cat bar.c
int array[2] = { 1, 2 };
$ ld -r -o temp.o foo.c bar.c
ld: warning: symbol 'array' has differing sizes:
    (file foo.o value=0x4; file bar.o value=0x8);
    bar.o definition taken
```

如果符号的对齐要求不同，则会产生一条类似的诊断消息。在这两种情况下，使用链接编辑器的 -t 选项可以不进行诊断。

另一种形式的属性差异是符号的类型。在以下示例中，符号 bar（） 已同时定义为数据项和函数。

```
$ cat foo.c
int bar()
{
        return (0);
}
$ cc -o libfoo.so -G -K pic foo.c
$ cat main.c
int bar = 1;

int main()
{
        return (bar);
}
$ cc -o main main.c -L. -lfoo
ld: warning: symbol 'bar' has differing types:
    (file main.o type=OBJT; file ./libfoo.so type=FUNC);
    main.o definition taken
```

注 - 此上下文中的符号类型是可以用 ELF 表示的分类。除非编程语言以最原始的方式使用数据类型，否则这些符号类型与数据类型无关。

在类似以上示例的情况下，在可重定位目标文件与共享目标文件之间进行解析时将采用可重定位目标文件定义。或者，在两个共享目标文件之间进行解析时采用第一个定义。在弱绑定符号或全局绑定符号之间进行这种解析时，还会生成警告。

链接编辑器的 -t 选项不会抑制符号类型之间的不一致性。

致命解析

无法解析的符号冲突会导致致命错误状态，并生成相应的错误消息。此消息会指示符号名称以及提供符号的文件的名称。不会生成任何输出文件。虽然致命状态足以导致链接编辑终止，但会先完成所有输入文件处理。通过这种方式，所有致命解析错误都可识别。

当两个可重定位目标文件都定义相同名称的非弱符号时，就会出现最常见的致命错误状态。

```
$ cat foo.c
int bar = 1;
$ cat bar.c
int bar()
{
        return (0);
}
$ ld -r -o temp.o foo.c bar.c
ld: fatal: symbol `bar' is multiply-defined:
    (file foo.o and file bar.o);
```

对于符号 bar 来讲，foo.c 和 bar.c 的定义相冲突。因为链接编辑器无法确定哪个符号优先，所以链接编辑通常会终止，并生成一条错误消息。可以使用链接编辑器的 -z muldefs 选项抑制出现此错误状态。此选项允许采用第一个符号定义。

未定义符号

在读取了所有输入文件并完成了所有符号解析后，链接编辑器将搜索内部符号表，以查找尚未绑定到符号定义的任何符号引用。这些符号引用称为未定义符号。未定义符号对链接编辑过程的影响因要生成的输出文件类型以及符号类型而异。

生成可执行输出文件

生成可执行输出文件时，如果有任何未定义符号，则链接编辑器的缺省行为是终止并显示相应的错误消息。如果可重定位目标文件中的符号引用从未与符号定义匹配，则符号将保持未定义状态。

```
$ cat main.c
extern int foo();

int main()
{
        return (foo());
}
$ cc -o prog main.c
Undefined           first referenced
 symbol                 in file
foo                     main.o
ld: fatal: symbol referencing errors
```

同样，如果使用共享目标文件创建动态可执行文件，并且该共享目标文件中包含未解析的符号定义，则会产生未定义符号错误。

```
$ cat foo.c
```

```
extern int bar;
int foo()
{
        return (bar);
}
$ cc -o libfoo.so -G -K pic foo.c
$ cc -o prog main.c -L. -lfoo
Undefined           first referenced
 symbol                 in file
bar                     ./libfoo.so
ld: fatal: symbol referencing errors
```

要像在上个示例中一样，允许未定义的符号，可使用链接编辑器的 `-z nodefs` 选项抑制缺省错误状态。

注 - 使用 `-z nodefs` 选项时应谨慎。如果在执行进程期间需要不可用的符号引用，会发生致命的运行时重定位错误。在初始执行和测试应用程序期间可能会检测到此错误。然而，执行路径越复杂，检测此错误状态需要的时间就越长，这将非常耗时且开销很大。

将可重定位目标文件中的符号引用绑定到隐式定义的共享目标文件中的符号定义时，符号也可能保持未定义状态。例如，继续使用上一个示例中使用的 `main.c` 和 `foo.c` 文件。

```
$ cat bar.c
int bar = 1;
$ cc -o libbar.so -R. -G -K pic bar.c -L. -lfoo
$ ldd libbar.so
        libfoo.so =>     ./libfoo.so
$ cc -o prog main.c -L. -lbar
Undefined           first referenced
 symbol                 in file
foo                     main.o  (symbol belongs to implicit \
                        dependency ./libfoo.so)
ld: fatal: symbol referencing errors
```

`prog` 是通过显式引用 `libbar.so` 而生成的。`libbar.so` 依赖于 `libfoo.so`。因此，就建立了从 `prog` 到 `libfoo.so` 的隐式引用。

因为 `main.c` 对 `libfoo.so` 提供的接口进行特定引用，所以 `prog` 确实依赖于 `libfoo.so`。但是，在要生成的输出文件中将仅记录显式共享目标文件的依赖项。因此，如果开发了一个不再依赖于 `libfoo.so` 的新版本 `libbar.so`，`prog` 将无法运行。

因此，此类型的绑定被认为是致命错误。必须通过在链接编辑 `prog` 期间直接引用库来使隐式引用变为显式引用。前面示例中显示的致命错误消息中会提示需要的引用。

生成共享目标文件输出文件

链接编辑器在生成共享目标文件输出文件时，允许在链接编辑结束时仍存在未定义符号。此缺省行为允许共享目标文件从将其定义为依赖项的动态可执行文件导入符号。

可以使用链接编辑器的 -z defs 选项在存在任何未定义符号的情况下强制生成致命错误。建议在创建任何共享目标文件时使用此选项。引用了某个应用程序中的符号的共享目标文件可以在使用 extern mapfile 指令定义符号的同时使用 -z defs 选项。请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。

自包含的共享目标文件（其中的所有对外部符号的引用都通过指定的依赖项得到满足）可提供最大的灵活性。此共享目标文件可由许多用户使用，并且这些用户无需确定和建立依赖项来满足共享目标文件的要求。

弱符号

以前，使用弱符号来禁用插入，或对可选功能进行测试。但是，经验表明，弱符号在当前编程环境中易变化且不可靠，建议不要使用它们。

在系统共享目标文件中经常使用弱符号别名。目的在于提供替代接口名称，通常符号名称使用 "_" 字符作为前缀。可从其他系统共享目标文件中引用该别名，以避免由于应用程序导出其自己的符号名称实现而导致的插入问题。在实践中，证明了该技术过于复杂且使用不一致。Oracle Solaris 的最新版本可在使用直接绑定的系统目标文件之间建立显式绑定。请参见第 6 章 直接绑定。

经常使用弱符号引用，以在运行时测试接口是否存在。该技术对生成环境、运行时环境进行限制，可通过编译器优化禁用该技术。使用具有 RTLD_DEFAULT 或 RTLD_PROBE 句柄的 dlsym(3C)，可提供一致、稳健的方式来测试符号是否存在。请参见“测试功能” [114]。

输出文件中的暂定符号顺序

构成输入文件的符号通常以这些符号的顺序出现在输出文件中。但暂定符号例外，因为完成这些符号的解析后才会完全定义这些符号。输出文件中暂定符号的顺序可能不遵循其原始顺序。

如果需要控制一组符号的顺序，应将所有暂定定义重新定义为初始化为零的数据项。例如，与源文件 foo.c 中说明的原始顺序相比，以下暂定定义将导致对输出文件中的数据项重新排序。

```
$ cat foo.c
char One_array[0x10];
char Two_array[0x20];
char Three_array[0x30];
$ cc -o libfoo.so -G -Kpic foo.c
$ elfdump -sN.dynsym libfoo.so | grep array | sort -k 2,2
    [11]   0x10614  0x20  OBJT GLOB  D    0 .bss          Two_array
```

```
     [3]   0x10634  0x30  OBJT GLOB  D    0 .bss          Three_array
     [4]   0x10664  0x10  OBJT GLOB  D    0 .bss          One_array
```

根据符号地址对符号排序将导致符号的输出顺序与其在源文件中定义的顺序不同。相反，通过将这些符号定义为已初始化的数据项，可确保这些符号在输入文件中的相对顺序传递到输出文件。

```
$ cat foo.c
char A_array[0x10] = { 0 };
char B_array[0x20] = { 0 };
char C_array[0x30] = { 0 };
$ cc -o libfoo.so -G -Kpic foo.c
$ elfdump -sN.dynsym libfoo.so | grep array | sort -k 2,2
     [4]   0x10614  0x10  OBJT GLOB  D    0 .data         One_array
    [11]   0x10624  0x20  OBJT GLOB  D    0 .data         Two_array
     [3]   0x10644  0x30  OBJT GLOB  D    0 .data         Three_array
```

定义其他符号

除输入文件中提供的符号外，还可以为链接编辑提供其他全局符号引用或全局符号定义。生成符号引用的最简单形式是使用链接编辑器的 -u 选项。使用链接编辑器的 -M 选项和关联的 mapfile 的灵活性更大。使用此 *mapfile*，可以定义全局符号引用和各种全局符号定义。可以指定符号可见性和类型之类的符号属性。有关可用选项的完整说明，请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。

使用 -u 选项定义其他符号

-u 选项提供了一种在链接编辑命令行中生成全局符号引用的机制。可以使用此选项完全从归档执行链接编辑。选择要从多个归档中提取的目标文件时，此选项还可以提供更多灵活性。有关归档提取的概述，请参见“归档处理” [27]一节。

例如，可能要从可重定位目标文件 main.o 生成动态可执行文件（此目标文件引用了符号 foo 和 bar）。您希望从 lib1.a 中包含的可重定位目标文件 foo.o 获取符号定义 foo，并从 lib2.a 中包含的可重定位目标文件 bar.o 获取符号定义 bar。

但是，归档 lib1.a 中也包含定义符号 bar 的可重定位目标文件。此可重定位目标文件的功能可能与 lib2.a 中提供的可重定位目标文件不同。要指定需要的归档提取，可以使用以下链接编辑。

```
$ cc -o prog -L. -u foo -l1 main.o -l2
```

-u 选项生成对符号 foo 的引用。此引用将导致从归档 lib1.a 中提取可重定位目标文件 foo.o。对符号 bar 的第一次引用出现在 main.o 中，这是在处理了 lib1.a 之后遇到的。因此，将从归档 lib2.a 获取可重定位目标文件 bar.o。

注 - 此简单示例假定 lib1.a 中的可重定位目标文件 foo.o 没有直接或间接地引用符号 bar。如果 lib1.a 引用 bar，在处理 lib1.a 期间还会从中提取可重定位目标文件 bar.o。有关链接编辑器处理归档多遍的介绍，请参见“归档处理” [27]。

定义符号引用

以下示例说明如何定义三种符号引用。然后，使用这些引用提取归档成员。虽然可以通过对链接编辑指定多个 -u 选项来实现归档提取，但此示例还说明了如何将符号的最终作用域缩减到局部。

```
$ cat foo.c
#include    <stdio.h>

void foo()
{
        (void) printf("foo: called from lib.a\n");
}
$ cat bar.c
#include    <stdio.h>

void bar()
{
        (void) printf("bar: called from lib.a\n");
}
$ cat main.c
extern void foo(), bar();

void main()
{
        foo();
        bar();
}
$ cc -c foo.c bar.c main.c
$ ar -rc lib.a foo.o bar.o main.o
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        local:
                foo;
                bar;
        global:
                main;
};
$ cc -o prog -M mapfile lib.a
$ prog
foo: called from lib.a
bar: called from lib.a
$ elfdump -sN.symtab prog | egrep 'main$|foo$|bar$'
    [29]   0x10f30  0x24  FUNC LOCL  H    0 .text          bar
    [30]   0x10ef8  0x24  FUNC LOCL  H    0 .text          foo
```

```
    [55]   0x10f68  0x24  FUNC GLOB  D    0 .text          main
```

在“缩减符号作用域” [46]一节中更详细地说明了将符号作用域从全局缩减为局部的重要性。

定义绝对符号

以下示例说明如何定义两种绝对符号定义。然后，使用这些定义解析输入文件 `main.c` 中的引用。

```
$ cat main.c
#include    <stdio.h>

extern int foo();
extern int bar;

void main()
{
        (void) printf("&foo = 0x%p\n", &foo);
        (void) printf("&bar = 0x%p\n", &bar);
}
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=FUNCTION; VALUE=0x400 };
                bar     { TYPE=DATA;     VALUE=0x800 };
};
$ cc -o prog -M mapfile main.c
$ prog
&foo = 0x400
&bar = 0x800
$ elfdump -sN.symtab prog | egrep 'foo$|bar$'
    [45]     0x800     0  OBJT GLOB  D    0 ABS            bar
    [69]     0x400     0  FUNC GLOB  D    0 ABS            foo
```

从输入文件获取函数或数据项的符号定义时，这些符号定义通常与数据存储元素关联。`mapfile` 定义不足以构造此数据存储，因此，这些符号必须保持为绝对值。与 `size` 关联、但无 `value` 的简单 `mapfile` 定义会导致创建数据存储。这种情况下，符号定义将带有节索引。但是，带有 `value` 的 `mapfile` 定义会导致创建绝对符号。如果在共享目标文件中定义符号，应当避免绝对定义。请参见“扩充符号定义” [45]。

定义暂定符号

还可以使用 `mapfile` 来定义 `COMMON`（即暂定）符号。与其他类型的符号定义不同，暂定符号在文件中不占用存储空间，而定义在运行时必须分配的存储空间。因此，定义此类型的符号有助于要生成的输出文件的存储分配。

暂定符号与其他符号类型的一个不同特性在于，暂定符号的 *value* 属性会指示其对齐要求。因此，可以使用 `mapfile` 定义重新对齐从链接编辑的输入文件中获取的暂定定义。

以下示例给出了两个暂定符号的定义。符号 `foo` 定义新的存储区域，而符号 `bar` 实际上用于更改文件 `main.c` 中相同暂定定义的对齐方式。

```
$ cat main.c
#include    <stdio.h>

extern int foo;

int bar[0x10];

void main()
{
        (void) printf("&foo = 0x%p\n", &foo);
        (void) printf("&bar = 0x%p\n", &bar);
}
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=COMMON; VALUE=0x4;   SIZE=0x200 };
                bar     { TYPE=COMMON; VALUE=0x102; SIZE=0x40 };
};
$ cc -o prog -M mapfile main.c
ld: warning: symbol 'bar' has differing alignments:
    (file mapfile value=0x102; file main.o value=0x4);
    largest value applied
$ prog
&foo = 0x21264
&bar = 0x21224
$ elfdump -sN.symtab prog | egrep 'foo$|bar$'
    [45]   0x21224   0x40  OBJT GLOB  D    0 .bss           bar
    [69]   0x21264  0x200  OBJT GLOB  D    0 .bss           foo
```

注 - 可以使用链接编辑器的 `-t` 选项抑制此符号解析诊断。

扩充符号定义

应避免在共享目标文件中创建绝对数据符号。从动态可执行文件对共享目标文件中数据项的外部引用通常需要创建复制重定位。请参见“复制重定位” [170]。要提供此重定位，应当将数据项与数据存储关联。可通过在可重定位目标文件中定义符号来生成此关联。也可以通过在 `mapfile` 中定义符号并使用 `size` 声明和 *no* `value` 声明来生成此关联。请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。

可以过滤数据符号。请参见“作为过滤器的共享目标文件” [127]。要提供此过滤，可以通过 `mapfile` 定义扩充目标文件定义。以下示例创建包含函数和数据定义的过滤器。

```
$ cat mapfile
```

```
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=FUNCTION;       FILTER=filtee.so.1 };
                bar     { TYPE=DATA; SIZE=0x4; FILTER=filtee.so.1 };
        local:
                *;
};
$ cc -o filter.so.1 -G -Kpic -h filter.so.1 -M mapfile -R.
$ elfdump -sN.dynsym filter.so.1 | egrep 'foo|bar'
     [1]   0x105f8     0x4  OBJT GLOB  D    1 .data          bar
     [7]         0       0  FUNC GLOB  D    1 ABS            foo
$ elfdump -y filter.so.1 | egrep 'foo|bar'
     [1]  F        [0] filtee.so.1        bar
     [7]  F        [0] filtee.so.1        foo
```

在运行时，会将从外部目标文件到以上任一符号的引用解析为 *filtee* 中的定义。

缩减符号作用域

可以使用在 `mapfile` 中定义为具有局部作用域的符号定义来缩减符号的最终绑定。对于将来使用生成的文件作为其输入一部分的链接编辑，此机制删除对这些链接编辑的符号可见性。事实上，此机制可以提供准确的文件接口定义，从而限制其他文件可以使用的功能。

例如，要从文件 `foo.c` 和 `bar.c` 生成一个简单的共享目标文件。文件 `foo.c` 包含全局符号 `foo`，此符号提供了您希望提供给其他文件的服务。文件 `bar.c` 包含符号 `bar` 和 `str`，这两个符号提供了共享目标文件的底层实现。使用这些文件创建的共享目标文件通常导致创建三个具有全局作用域的符号。

```
$ cat foo.c
extern const char *bar();

const char *foo()
{
        return (bar());
}
$ cat bar.c
const char *str = "returned from bar.c";

const char *bar()
{
        return (str);
}
$ cc -o libfoo.so.1 -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
    [41]     0x560    0x18  FUNC GLOB  D    0 .text          bar
    [44]     0x520    0x2c  FUNC GLOB  D    0 .text          foo
    [45]   0x106b8     0x4  OBJT GLOB  D    0 .data          str
```

现在，可以在另一个应用程序的链接编辑过程中使用 libfoo.so.1 提供的功能。对符号 foo 的引用被绑定到共享目标文件所提供的实现。

由于符号 bar 和 str 具有全局绑定，因此还可以直接引用这两个符号。此可见性会产生严重后果，因为以后可能会更改作为函数 foo 基础的实现。这样做可能会无意中导致已绑定到 bar 或 str 的现有应用程序失败或行为异常。

符号 bar 和 str 的全局绑定的另一个结果是，可以在这些符号中插入同名的符号。“简单解析” [37]一节中将介绍在共享目标文件中插入符号。此插入可以是有意的，用于避开共享目标文件提供的预期功能。另一方面，此插入可能是无意的，是将相同通用符号名称同时用于应用程序和共享目标文件的结果。

在开发共享目标文件时，可以通过将符号 bar 和 str 的作用域缩减为局部绑定来防止出现这种情况。在以下示例中，符号 bar 和 str 不再作为共享目标文件的接口的一部分提供。因此，外部目标文件不能引用或插入这些符号。您已经有效地定义了共享目标文件的接口。可以在隐藏底层实现详细信息的同时管理此接口。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        local:
                bar;
                str;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
    [24]     0x548    0x18  FUNC LOCL  H    0 .text          bar
    [25]   0x106a0     0x4  OBJT LOCL  H    0 .data          str
    [45]     0x508    0x2c  FUNC GLOB  D    0 .text          foo
```

缩减符号作用域还具有其他性能方面的优点。现在，以前运行时必需的针对符号 bar 和 str 的符号重定位已缩减为相对重定位。有关符号重定位开销的详细信息，请参见“执行重定位的时间” [169]。

随着链接编辑期间处理的符号数的增加，在 mapfile 中定义局部作用域缩减将变得越来越难维护。有一种更灵活的替代机制，可以根据应维护的全局符号来定义共享目标文件的接口。全局符号定义允许链接编辑器将所有其他符号缩减为局部绑定。此机制是使用特殊的自动缩减指令 "*" 实现的。例如，可以重新编写上面的 mapfile 定义，将 foo 定义为生成的输出文件中所需的唯一全局符号。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
        local:
                *;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
    [26]     0x570    0x18  FUNC LOCL  H    0 .text          bar
```

```
    [27]   0x106d8     0x4  OBJT LOCL  H    0 .data          str
    [50]     0x530    0x2c  FUNC GLOB  D    0 .text          foo
```

此示例还在 mapfile 指令中定义了一个版本名称 ISV_1.1。此版本名称建立一个内部版本定义，用于定义文件的符号接口。建议创建版本定义。此定义将成为可在文件演变过程中使用的内部更新控制机制的基础。请参见第 9 章 接口和版本控制。

注 - 如果未提供版本名称，那么将使用输出文件名作为版本定义的标签。可以使用链接编辑器的 -z noversion 选项抑制在输出文件中创建的版本更新信息。

每次指定版本名称时，都必须将所有全局符号指定给版本定义。如果有任何全局符号未指定给版本定义，链接编辑器将生成致命错误状态。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
Undefined           first referenced
 symbol                 in file
str                     bar.o  (symbol has no version assigned)
bar                     bar.o  (symbol has no version assigned)
ld: fatal: symbol referencing errors
```

可以使用 -B local 选项在命令行中声明自动缩减指令 "*"。可按照以下方式成功编译上一个示例。

```
$ cc -o libfoo.so.1 -M mapfile -B local -G foo.c bar.c
```

生成可执行文件或共享目标文件时，缩减任何符号都会导致在输出映像中记录版本定义。生成可重定位目标文件时，将创建版本定义，但不会处理符号缩减。结果是所有符号缩减的符号项仍保持全局。例如，将上一个 mapfile 与自动缩减指令和关联的可重定位目标文件配合使用时，会创建一个中间的可重定位目标文件，但不缩减任何符号。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
        local:
                *;
};
$ ld -o libfoo.o -M mapfile -r foo.o bar.o
$ elfdump -s libfoo.o | egrep 'foo$|bar$|str$'
    [29]      0x10    0x2c  FUNC GLOB  D    2 .text          foo
    [30]         0     0x4  OBJT GLOB  H    0 .data          str
```

此映像中创建的版本定义显示需要缩减符号。最终使用可重定位目标文件生成可执行文件或共享目标文件时，将会缩减符号。也就是说，链接编辑器将按照与在 mapfile 中处理版本更新数据相同的方式，来读取并解释可重定位目标文件中包含的符号缩减信息。

因此，现在可以使用上一个示例中产生的中间可重定位目标文件来生成共享目标文件。

```
$ ld -o libfoo.so.1 -G libfoo.o
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
    [24]     0x508    0x18  FUNC LOCL  H    0 .text          bar
    [25]   0x10644     0x4  OBJT LOCL  H    0 .data          str
    [42]     0x4c8    0x2c  FUNC GLOB  D    0 .text          foo
```

创建可执行文件或共享目标文件时缩减符号通常是最常见的要求。不过，使用链接编辑器的 `-B reduce` 选项可以强制在创建可重定位目标文件时缩减符号。

```
$ ld -o libfoo.o -M mapfile -B reduce -r foo.o bar.o
$ elfdump -sN.symtab libfoo.o | egrep 'foo$|bar$|str$'
    [20]      0x50    0x18  FUNC LOCL  H    0 .text          bar
    [21]         0     0x4  OBJT LOCL  H    0 .data          str
    [30]      0x10    0x2c  FUNC GLOB  D    2 .text          foo
```

删除符号

对符号缩减的扩展是指从目标文件的符号表中删除符号项。局部符号仅在目标文件的 `.symtab` 符号表中维护。使用链接编辑器的 `-z strip-class` 选项，或者在链接编辑之后使用 strip(1)，可以将此整个表从目标文件中删除。有时，可能需要保留 `.symtab` 符号表，但要删除选择的局部符号定义。

可以使用 `mapfile` 关键字 `ELIMINATE` 来执行符号删除。与 `local` 指令一样，可以单独定义符号，也可以将符号名称定义为特殊的自动删除指令 "*"。以下示例说明如何删除上一个符号缩减示例中的符号 `bar`。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
        local:
                str;
        eliminate:
                *;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
    [26]   0x10690     0x4  OBJT LOCL  H    0 .data          str
    [44]     0x4e8    0x2c  FUNC GLOB  D    0 .text          foo
```

可以使用 `-B eliminate` 选项在命令行中声明自动删除指令 "*"。

外部绑定

共享目标文件中的定义满足要创建的目标文件中的符号引用时，符号将保持未定义状态。可以在运行时查找与此符号关联的重定位信息。提供定义的共享目标文件通常会成为依赖项。

运行时链接程序在运行时使用缺省搜索模型来查找此定义。通常，将搜索每个目标文件（从动态可执行文件开始），并按装入目标文件的顺序处理每个依赖项。

还可以创建目标文件来使用直接绑定。使用此方法时，可以在要创建的目标文件中维护符号引用与提供符号定义的目标文件之间的关系。运行时链接程序使用此信息将引用直接绑定到定义符号的目标文件，从而绕过缺省符号搜索模型。请参见第 6 章 直接绑定。

字符串表压缩

链接编辑器可以通过删除重复项和尾部子串来压缩字符串表。此压缩可显著减小任何字符串表的大小。例如，`.dynstr` 表压缩后可缩小文本段，从而减少运行时分页活动。由于这些优点，缺省情况下将启用字符串表压缩。

由于字符串表压缩，提供大量符号的目标文件可能会增加链接编辑时间。为了避免开发期间产生此开销，请使用链接编辑器的 `-z nocompstrtab` 选项。可以使用链接编辑器的调试标记 `-D strtab,detail` 显示链接编辑期间执行的任何字符串表压缩。

生成输出文件

在完成输入文件处理和符号解析并且没有出现致命错误后，链接编辑器将生成输出文件。链接编辑器首先生成完成输出文件必需的其他节。这些节包括所有输入文件中的符号表，这些符号表包含局部符号定义以及已解析的全局符号和弱符号信息。

此外，还包括运行时链接程序需要的任何输出重定位和动态信息节。确定所有输出节信息后，将计算输出文件总大小。然后，相应地创建输出文件映像。

创建动态可执行文件或共享目标文件时，通常会生成两个符号表。`.dynsym` 表及其关联的字符串表 `.dynstr` 包含寄存器符号、全局符号、弱符号和节符号。这些节成为在运行时作为进程映像一部分映射的 `text` 段的一部分。请参见 `mmapobj(2)`。使用此映射，运行时链接程序可以读取这些节，以便执行任何必需的重定位。

`.symtab` 表及其关联的字符串表 `.strtab` 包含在输入文件处理过程中收集的所有符号。这些节不能作为进程映像的一部分进行映射。使用链接编辑器的 `-z strip-class` 选项，或者在链接编辑器之后使用 `strip(1)`，可以将这些节从映像中剥离。

生成符号表期间，将创建保留符号。这些符号对于链接进程有特殊意义。不能在代码中定义这些符号。

_etext
: 所有只读信息（通常称为文本段）后面的第一个位置。

_edata
: 初始化的数据后面的第一个位置。

_end
: 所有数据后面的第一个位置。

_DYNAMIC
: .dynamic 信息节的地址。

END
: 与 _end 相同。此符号具有局部作用域，它与 _START_ 符号一起提供一种确定目标文件地址范围的方法。

_GLOBAL_OFFSET_TABLE_
: 对链接编辑器提供的地址表（即 .got 节）的引用，与位置无关。此表由与位置无关的数据引用构造而成，这些数据引用出现在使用 -K pic 选项编译的目标文件中。请参见“与位置无关的代码” [162]。

_PROCEDURE_LINKAGE_TABLE_
: 对链接编辑器提供的地址表（即 .plt 节）的引用，与位置无关。此表由与位置无关的函数引用构造而成，这些数据引用出现在已使用 -K pic 选项编译的目标文件中。请参见“与位置无关的代码” [162]。

START
: 文本段中的第一个位置。此符号具有局部作用域，它与 _END_ 符号一起提供了一种确定目标文件地址范围的方法。

生成可执行文件时，链接编辑器将查找其他符号，以定义可执行文件的入口点。如果使用链接编辑器的 -e 选项指定了一个符号，那么将使用该符号。否则，链接编辑器会查找保留符号名称 _start，然后查找 main。

标识功能要求

功能标识允许代码执行所需的系统属性。可以按优先级顺序使用以下功能。

- 平台功能 – 用名称标识特定平台。
- 计算机功能 – 用名称标识特定计算机硬件。
- 硬件功能 – 用功能标志标识指令集扩展和其他硬件详细信息。
- 软件功能 – 用功能标志反映软件环境的属性。

以上每项功能既可单独定义，也可共同构成一个功能组。

仅当某些功能可用时才能执行的代码应当借助关联的 ELF 目标文件中的功能节来确定这些要求。目标文件中的记录功能要求允许系统在尝试执行关联代码之前验证目标文件。这些要求还可以提供一个框架，以便系统从目标文件系列中选择最适合的目标文件。目标文件系列由相同目标文件的变体组成，每个变体具有不同的功能要求。

动态目标文件以及目标文件中的各个函数或初始化数据项可以与功能要求相关联。理论上，功能要求记录在编译器生成的可重定位目标文件中，并反映了编译时指定的选项或优化。链接编辑器结合所有输入可重定位目标文件的功能来创建输出文件的最终功能节。请参见“功能节” [306]。

此外，还可以在链接编辑器创建输出文件时定义功能。使用 `mapfile` 和链接编辑器的 `-M` 选项标识这些功能。使用 `mapfile` 定义的功能可以扩充或覆盖在所有输入可重定位目标文件中指定的功能。`mapfile` 通常用于扩充未生成必需的功能信息的编译器。

系统功能就是对运行中系统进行描述的功能。可以使用 `uname(1)` 以及 `-i` 选项和 `-m` 选项分别显示平台名称和计算机硬件名称。可以使用 `isainfo(1)` 和 `-v` 选项显示系统硬件功能。运行时会将目标文件的功能要求与系统功能进行比较，以确定是否能装入目标文件，或者是否能在目标文件中使用符号。

目标文件功能就是与目标文件关联的功能。这些功能定义了整个目标文件的要求，并控制能否在运行时装入目标文件。如果系统无法满足目标文件所要求的功能，那么在运行时不能装入该目标文件。功能可用于提供给定目标文件的多个实例，每个实例均针对与目标文件要求匹配的系统进行了优化。运行时链接程序可以通过比较目标文件功能要求与系统提供的功能，从此类目标文件实例系列中透明地选择最佳实例。

符号功能即与目标文件中各个函数或初始化数据项关联的功能。这些功能定义了目标文件中一个或多个符号的要求，并控制在运行时能否使用符号。符号功能允许在单个目标文件中存在多个函数实例。每个函数实例均针对具有不同功能的系统进行了优化。符号功能还允许目标文件中存在多个初始化数据项实例。每个数据实例都可以定义特定于系统的数据。如果系统无法满足符号实例所要求的功能，那么在运行时不能使用该符号实例。相反，必须使用相同符号名的替代实例。通过符号功能，可以构造单个目标文件用在功能不断变化的系统上。函数系列可以为支持这些功能的系统提供经过优化的实例，并且为其他功能较弱的系统提供更为通用的实例。初始化数据项系列可以提供特定于系统的数据。运行时链接程序可以通过比较符号功能要求与系统提供的功能，从此类符号实例系列中透明地选择最佳实例。

目标文件功能和符号功能可用于为当前运行的系统选择最佳目标文件以及目标文件中的最佳符号。目标文件功能和符号功能是可选功能，彼此独立。但是，定义符号功能的目标文件也可以定义目标文件功能。这种情况下，任何功能符号系列都应当带有一个满足目标文件功能的符号实例。如果不存在目标文件功能，那么任何功能符号系列都应当带有一个不需要任何功能的符号实例。在给定系统无适用的功能实例的情况下，此符号实例会提供缺省实现。

以下 x86 示例显示了 `foo.o` 的目标文件功能。这些功能应用于整个目标文件。此示例中不存在任何符号功能。

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap
```

```
 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1      0x840  [ SSE  MMX ]
```

以下 x86 示例显示了 `bar.o` 的符号功能。这些功能应用于单个函数 `foo` 和 `bar`。每个符号存在两个实例，每个实例被指定给不同的功能集合。此示例中不存在任何目标文件功能。

```
$ elfdump -H bar.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
     index    value     size  type bind oth ver shndx    name
      [25]        0     0x21  FUNC LOCL  D    0 .text    foo%mmx
      [26]     0x24     0x1e  FUNC LOCL  D    0 .text    bar%mmx

 Symbol Capabilities:
     index  tag               value
       [3]  CA_SUNW_HW_1      0x800  [ SSE ]

  Symbols:
     index    value     size  type bind oth ver shndx    name
      [33]     0x44     0x21  FUNC LOCL  D    0 .text    foo%sse
      [34]     0x68     0x1e  FUNC LOCL  D    0 .text    bar%sse
```

注 - 在此示例中，功能符号遵循在通用符号名后附加功能标识符的命名约定。此约定可由链接编辑器在将目标文件功能转换为符号功能时生成；以后将在“将目标文件功能转换为符号功能” [65]一节中介绍此约定。

功能定义提供了许多种组合，可以使用这些组合来标识目标文件要求或目标文件中各个符号的要求。硬件功能可提供最大的灵活性。硬件功能定义硬件要求，但不决定某个特定的计算机硬件名称或平台名称。不过，有时候底层系统的某些属性只能根据计算机硬件名称或平台名称来确定。可以通过标识功能名称对非常具体的系统功能进行编码，但对已标识目标文件的使用可能会受限制。如果新计算机硬件名称或平台名称适用于目标文件，那么必须重新生成目标文件来标识新的功能名称。

以下各小节将介绍如何定义功能，以及链接编辑器如何使用功能。

标识平台功能

目标文件的平台功能标识了可在其上执行目标文件或目标文件中特定符号的系统的平台名称。可以定义多个平台功能。此标识非常具体，且优先级高于其他任何功能类型。

可通过实用程序 uname(1) 和 `-i` 选项显示系统的平台名称。

可使用以下 mapfile 语法定义平台功能要求。

```
$mapfile_version 2
    CAPABILITY {
            PLATFORM  = platform_name...;
            PLATFORM += platform_name...;
            PLATFORM -= platform_name...;
    };
```

可使用一个或多个平台名称限定 PLATFORM 属性。"+=" 形式的赋值将扩充输入目标文件所指定的平台功能，而 "=" 形式将覆盖这些功能。"-=" 形式的赋值用于从输出目标文件中排除平台功能。以下 SPARC 示例可将目标文件 foo.so.1 标识为特定于 SUNW,SPARC-Enterprise 平台。

```
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        PLATFORM = 'SUNW,SPARC-Enterprise';
};
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag              value
       [0]  CA_SUNW_PLAT      SUNW,SPARC-Enterprise
```

可重定位目标文件可以定义平台功能。可将这些功能收集到一起，定义要生成的目标文件的最终功能要求。

通过使用 "=" 形式的赋值覆盖任何输入可重定位目标文件可能提供的任何平台功能，可以从 mapfile 显式控制目标文件的平台功能。结合使用空 PLATFORM 属性和 "=" 形式的赋值，可以有效地删除要生成的目标文件中的所有平台功能要求。

运行时链接程序会根据系统的平台名称验证动态目标文件中定义的平台功能要求。仅当目标文件中记录的平台名称之一与系统的平台名称相匹配时，才使用目标文件。

在某些情况下，针对特定平台编写代码会很有用，但开发硬件功能系列可提供更大的灵活性，因此建议采用。硬件功能系列可提供经过优化的代码，应用于更广泛的系统上。

标识计算机功能

目标文件的计算机功能可以标识可在其上执行目标文件或目标文件中特定符号的系统的计算机硬件名称。可以定义多个计算机功能。此标识的优先级低于平台功能定义，但高于其他任何功能类型。

可通过实用程序 uname(1) 和 -m 选项显示系统的计算机硬件名称。

可使用以下 mapfile 语法定义计算机功能要求。

```
$mapfile_version 2
    CAPABILITY {
            MACHINE  = machine_name...;
            MACHINE += machine_name...;
            MACHINE -= machine_name...;
    };
```

可使用一个或多个计算机硬件名称限定 MACHINE 属性。"+=" 形式的赋值将扩充输入目标文件所指定的计算机功能，而 "=" 形式将覆盖这些功能。"-=" 形式的赋值用于从输出目标文件中排除计算机功能。以下 SPARC 示例可将目标文件 foo.so.1 标识为特定于 sun4u 计算机硬件名称。

```
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        MACHINE = sun4u;
};
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag              value
       [0]  CA_SUNW_MACH      sun4u
```

可重定位目标文件可以定义计算机功能。可将这些功能收集到一起，定义要生成的目标文件的最终功能要求。

通过使用 "=" 形式的赋值覆盖任何输入可重定位目标文件可能提供的任何计算机功能，从而从 mapfile 显式控制目标文件的计算机功能。结合使用空 MACHINE 属性和 "=" 形式的赋值，可以有效地删除要生成的目标文件中的所有计算机功能要求。

运行时链接程序会根据系统的计算机硬件名称验证动态目标文件中定义的计算机功能要求。仅当目标文件中记录的计算机名称之一与系统的计算机名称相匹配时，才使用目标文件。

在某些情况下，针对特定计算机编写代码会很有用，但开发硬件功能系列可提供更大的灵活性，因此建议采用。硬件功能系列可提供经过优化的代码，应用于更广泛的系统上。

标识硬件功能

目标文件的硬件功能可标识系统正确执行目标文件或特定符号所需的硬件要求。例如，标识需要在某些 x86 体系结构上可用的 MMX 或 SSE 功能的代码。

可以使用以下 mapfile 语法标识硬件功能要求。

```
$mapfile_version 2
    CAPABILITY {
            HW  = hwcap_flag...;
```

```
        HW += hwcap_flag...;
        HW -= hwcap_flag...;
};
```

使用一个或多个标识限定 CAPABILITY 指令的 HW 属性，这些标记是硬件功能的符号表示。"+=" 形式的赋值可以扩充输入目标文件指定的硬件功能，而 "=" 形式可以覆盖这些功能。"-=" 形式的赋值用于从输出目标文件中排除硬件功能。

对于 SPARC 系统，硬件功能定义为 sys/auxv_SPARC.h 中的 AV_ 值。对于 x86 系统，硬件功能定义为 sys/auxv_386.h 中的 AV_ 值。

以下 x86 示例说明了如何将 MMX 和 SSE 声明为目标文件 foo.so.1 所要求的硬件功能。

```
$ egrep "MMX|SSE" /usr/include/sys/auxv_386.h
#define AV_386_MMX    0x0040
#define AV_386_SSE    0x0800
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        HW += SSE MMX;
};
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag              value
       [0]  CA_SUNW_HW_1      0x840  [ SSE  MMX ]
```

可重定位目标文件可以包含硬件功能值。链接编辑器结合多个输入可重定位目标文件中的任何硬件功能值。产生的 CA_SUNW_HW_1 值是对关联的输入值进行按位 OR 运算的结果。缺省情况下，这些值将与 mapfile 指定的硬件功能组合。

通过使用 "=" 形式的赋值覆盖任何输入可重定位目标文件可能提供的任何硬件功能，可以从 mapfile 显式控制目标文件的硬件功能要求。结合使用空 HW 属性和 "=" 形式的赋值，可以有效地删除要生成的目标文件的所有硬件功能要求。

以下示例可抑制输入可重定位目标文件 foo.o 定义的所有硬件功能数据，使其不包括在输出文件 bar.o 中。

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag              value
       [0]  CA_SUNW_HW_1      0x840  [ SSE  MMX ]
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        HW = ;
```

```
};
$ ld -o bar.o -r -Mmapfile foo.o
$ elfdump -H bar.o
$
```

运行时链接程序会针对系统提供的硬件功能验证动态目标文件定义的任何硬件功能要求。如果无法满足任何硬件功能要求，就不会在运行时装入目标文件。例如，如果进程不能使用 SSE 功能，则 ldd(1) 将指示以下错误。

```
$ ldd prog
        foo.so.1 =>      ./foo.so.1  - hardware capability unsupported: 0x800 [ SSE ]
        ....
```

利用不同硬件功能的动态目标文件的多个变体可以提供使用过滤器的灵活运行时环境。请参见“特定于功能的共享目标文件” [227]。

硬件功能还可以用于标识单个目标文件中个别函数的功能。这种情况下，运行时链接程序可以根据当前系统功能选择要使用的最恰当的函数实例。请参见“创建符号功能函数系列” [59]。

标识软件功能

目标文件的软件功能标识对于调试或监视进程可能很重要的软件特征。软件功能也能影响进程的执行。目前，可识别的唯一软件功能与目标文件使用的帧指针以及进程地址空间限制有关。

目标文件可以声明已知其帧指针的使用状态。然后，将帧指针声明为正在使用或未使用来限定此状态。

64 位目标文件可以指明：在运行时，必须在 32 位地址空间内使用它们。

软件功能标志是在 sys/elf.h 中定义的。

```
#define  SF1_SUNW_FPKNWN    0x001
#define  SF1_SUNW_FPUSED    0x002
#define  SF1_SUNW_ADDR32    0x004
```

可以使用以下 mapfile 语法标识这些软件功能要求。

```
        $mapfile_version 2
CAPABILITY {
        SF  = sfcap_flags...;
        SF += sfcap_flags...;
        SF -= sfcap_flags...;
};
```

CAPABILITY 指令的 SF 属性可赋值为标记 FPKNWN、FPUSED 和 ADDR32 中的任意一个。

可重定位目标文件可以包含软件功能值。链接编辑器可以组合多个输入可重定位目标文件中的软件功能值。软件功能还可随 mapfile 提供。缺省情况下，任何 mapfile 值都将与可重定位目标文件提供的值组合。

通过使用 "=" 形式的赋值覆盖任何输入可重定位目标文件可能提供的任何软件功能，可以从 mapfile 显式控制目标文件的软件功能要求。结合使用空 SF 属性和 "=" 形式的赋值，可以有效地删除要生成的目标文件的所有软件功能要求。

以下示例可抑制输入可重定位目标文件 foo.o 定义的所有软件功能数据，使其不包括在输出文件 bar.o 中。

```
$ elfdump -H foo.o

Object Capabilities:
    index  tag               value
      [0]  CA_SUNW_SF_1      0x3  [ SF1_SUNW_FPKNWN  SF1_SUNW_FPUSED ]
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        SF = ;
};
$ ld -o bar.o -r -Mmapfile foo.o
$ elfdump -H bar.o
$
```

软件功能帧指针处理

可按如下方法根据两个帧指针计算 CA_SUNW_SF_1 的值。

表 2-1 CA_SUNW_SF_1 帧指针标志组合状态表

输入文件 1	输入文件 2		
	SF1_SUNW_FPKNWN SF1_ SUNW_FPUSED	**SF1_SUNW_FPKNWN**	**<unknown>**
SF1_SUNW_FPKNWN SF1_ SUNW_FPUSED	SF1_SUNW_FPKNWN SF1_ SUNW_FPUSED	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN SF1_ SUNW_FPUSED
SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN
<unknown>	SF1_SUNW_FPKNWN SF1_ SUNW_FPUSED	SF1_SUNW_FPKNWN	<unknown>

此计算方法适用于每个可重定位目标文件值和 mapfile 值。如果不存在 .SUNW_cap 节、此节未包含 CA_SUNW_SF_1 值或者未设置 SF1_SUNW_FPKNW 或 SF1_SUNW_FPUSED 标志，那么目标文件的帧指针软件功能是未知的。

软件功能地址空间限制处理

使用 SF1_SUNW_ADDR32 软件功能标志标识的 64 位目标文件可包含需要 32 位地址空间的经过优化的代码。按照此方式标识的 64 位目标文件可与任何其他 64 位目标文件交互操作，无论它们是否使用 SF1_SUNW_ADDR32 标志进行标识。在 64 位输入可重定位目标文件

中出现的 SF1_SUNW_ADDR32 标志将传播到 CA_SUNW_SF_1 值，该值是为链接编辑器将创建的输出文件创建的。

64 位可执行文件中存在的 SF1_SUNW_ADDR32 标志可确保将关联进程限定于低 32 位地址空间。此限定地址空间包括进程栈和所有进程依赖项。在此类进程内，所有目标文件都在限定的 32 位地址空间内装入，无论其是否使用 SF1_SUNW_ADDR32 标志进行标识。

64 位共享目标文件可以包含 SF1_SUNW_ADDR32 标志。但是，限定的地址空间要求只能由包含 SF1_SUNW_ADDR32 标志的 64 位可执行文件建立。因此，64 位 SF1_SUNW_ADDR32 共享目标文件必须是 64 位 SF1_SUNW_ADDR32 可执行文件的依赖项。

链接编辑器在生成不受限的 64 位可执行文件时若遇到 64 位 SF1_SUNW_ADDR32 共享目标文件，将导致生成警告。

```
$ cc -m64 -o main main.c -lfoo
ld: warning: file libfoo.so: section .SUNW_cap: software capability ADDR32: \
    requires executable be built with ADDR32 capability
```

通过不受限的 64 位可执行文件创建的进程在运行时若遇到 64 位 SF1_SUNW_ADDR32 共享目标文件，将导致生成致命错误。

```
$ ldd main
    libfoo.so =>     ./libfoo.so  - software capability unsupported: 0x4  [ ADDR32 ]
    ....
$ main
ld.so.1: main: fatal: ./libfoo.so: software capability unsupported: 0x4  [ ADDR32 ]
```

可以使用 mapfile 随 SF1_SUNW_ADDR32 生成可执行文件。

```
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        SF += ADDR32;
};
$ cc -m64 -o main main.c -Mmapfile -lfoo
$ elfdump -H main

Object Capabilities:
    index  tag               value
      [0]  CA_SUNW_SF_1      0x4  [ SF1_SUNW_ADDR32 ]
```

创建符号功能函数系列

开发者通常需要在单个目标文件中提供多个函数实例，且每个实例都针对特定功能集合进行了优化。理想情况下，这些实例的选择和使用对于任何使用者都是透明的。可以创建一个通用的前端函数以提供外部接口。此通用实例与经过优化的实例可以一起组合到一个目标文件中。此通用实例可以使用 getisax(2) 来确定系统功能，然后调用适当的已优化函数实例来处理任务。虽然此模式可行，但它不仅缺乏通用性，还会产生运行时开销。

符号功能为构造此类目标文件提供了一种替代机制。此机制更简单、更有效，而且不需要编写额外的前端代码。可以创建多个函数实例，并将其与不同的功能关联。这些实例与适合任何系统的缺省函数实例可以一起组合到单个动态目标文件中。运行时链接程序使用符号功能信息，从此符号系列中选择执行最适当的符号。

在以下示例中，x86 目标文件 foobar.mmx.o 和 foobar.sse.o 包含相同的函数 foo（） 和 bar（），这两个函数已被编译为分别使用 MMX 和 SSE 指令。

```
$ elfdump -H foobar.mmx.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
    index  tag               value
      [1]  CA_SUNW_ID        mmx
      [2]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
     index    value      size  type bind oth ver shndx    name
      [10]        0      0x21  FUNC LOCL  D    0 .text    foo%mmx
      [16]     0x24      0x1e  FUNC LOCL  D    0 .text    bar%mmx

$ elfdump -H foobar.sse.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
    index  tag               value
      [1]  CA_SUNW_ID        sse
      [2]  CA_SUNW_HW_1      0x800  [ SSE ]

  Capabilities symbols:
     index    value      size  type bind oth ver shndx    name
      [16]        0      0x2f  FUNC LOCL  D    0 .text    foo%sse
      [18]     0x48      0x30  FUNC LOCL  D    0 .text    bar%sse
```

其中的每个目标文件都包含一个局部符号，可以标识功能函数 foo%*（） 和 bar %*（）。此外，每个目标文件还会定义一个指向函数 foo（） 和 bar（） 的全局引用。对 foo（） 或 bar（） 的任何内部引用均通过这些全局引用进行重定位，这与外部引用是一样的。

现在可以将这两个目标文件与 foo（） 和 bar（） 的缺省实例相组合。这些缺省实例可满足全局引用，并提供与任何目标文件功能兼容的实现。可以说这些缺省实例引导着每个功能系列。如果不存在任何目标文件功能，那么此缺省实例也应当不需要任何功能。实际上，存在三个 foo（） 和 bar（） 的实例，全局实例提供缺省功能，局部实例提供在运行时所用的实现（如果关联的功能可用）。

```
$ cc -o libfoobar.so.1 -G foobar.o foobar.sse.o foobar.mmx.o
$ elfdump -sN.dynsym libfoobar.so.1 | egrep "foo|bar"
     [2]     0x700    0x21  FUNC LOCL  D    0 .text    foo%mmx
     [4]     0x750    0x2f  FUNC LOCL  D    0 .text    foo%sse
```

```
     [8]     0x784    0x1e  FUNC LOCL  D    0 .text    bar%mmx
     [9]     0x7b0    0x30  FUNC LOCL  D    0 .text    bar%sse
    [15]     0x7a0    0x14  FUNC GLOB  D    1 .text    foo
    [17]     0x7c0    0x14  FUNC GLOB  D    1 .text    bar
```

动态目标文件的功能信息会显示功能符号以及可用的功能系列。

```
$ elfdump -H libfoobar.so.1

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
   index  tag               value
     [1]  CA_SUNW_ID        mmx
     [2]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
   index    value     size  type bind oth ver shndx    name
     [2]    0x700     0x21  FUNC LOCL  D    0 .text    foo%mmx
     [8]    0x784     0x1e  FUNC LOCL  D    0 .text    bar%mmx

 Symbol Capabilities:
   index  tag               value
     [4]  CA_SUNW_ID        sse
     [5]  CA_SUNW_HW_1      0x800  [ SSE ]

  Symbols:
   index    value     size  type bind oth ver shndx    name
     [4]    0x750     0x2f  FUNC LOCL  D    0 .text    foo%sse
     [9]    0x7b0     0x30  FUNC LOCL  D    0 .text    bar%sse

Capabilities Chain Section:  .SUNW_capchain

 Capabilities family: foo
  chainndx  symndx      name
         1  [15]        foo
         2  [2]         foo%mmx
         3  [4]         foo%sse

 Capabilities family: bar
  chainndx  symndx      name
         5  [17]        bar
         6  [8]         bar%mmx
         7  [9]         bar%sse
```

在运行时，对 `foo（）` 和 `bar（）` 的所有引用最初都绑定到全局符号。不过，运行时链接程序将这些函数视为功能系列的引导实例。运行时链接程序会检查每个系列成员以确定是否有更好的功能函数。第一次调用函数时，此操作会发生一次性开销。对 `foo（）` 和 `bar（）` 的后续调用直接绑定到第一次调用所选的函数实例。通过使用运行时链接程序调试功能，可以观察此函数选择内容。

在以下示例中，底层系统未提供 `MMX` 或 `SSE` 支持。`foo（）` 的引导实例不需要特殊的功能支持，因此可满足任何重定位引用。

```
$ LD_DEBUG=symbols  main
....
debug: symbol=foo;  lookup in file=./libfoo.so.1  [ ELF ]
debug:   symbol=foo[15]:  capability family default
debug:   symbol=foo%mmx[2]:  capability specific (CA_SUNW_HW_1):  [ 0x40  [ MMX ] ]
debug:   symbol=foo%mmx[2]:  capability rejected
debug:   symbol=foo%sse[4]:  capability specific (CA_SUNW_HW_1):  [ 0x800  [ SSE ] ]
debug:   symbol=foo%sse[4]:  capability rejected
debug:   symbol=foo[15]:  used
```

在以下示例中，MMX 可用，但 SSE 不可用。支持 MMX 的 foo（） 实例可满足任何重定位引用。

```
$ LD_DEBUG=symbols  main
....
debug: symbol=foo;  lookup in file=./libfoo.so.1  [ ELF ]
debug:   symbol=foo[15]:  capability family default
debug:   symbol=foo%mmx[2]:  capability specific (CA_SUNW_HW_1):  [ 0x40  [ MMX ] ]
debug:   symbol=foo%mmx[2]:  capability candidate
debug:   symbol=foo%sse[4]:  capability specific (CA_SUNW_HW_1):  [ 0x800  [ SSE ] ]
debug:   symbol=foo%sse[4]:  capability rejected
debug:   symbol=foo[2]:  used
```

如果在同一个系统上可使用多个功能实例，则将使用一组优先规则来选择一个实例。

- 定义平台名称的功能组优先于不定义平台名称的功能组。
- 定义计算机硬件名称的功能组优先于不定义计算机硬件名称的功能组。
- 较大的硬件功能值优先于较小的硬件功能值。

必须通过过程链接表项访问功能函数实例系列。请参见“过程链接表（特定于处理器）” [372]。此过程链接引用需要运行时链接程序对函数进行解析。在此过程中，运行时链接程序可以处理关联的符号功能信息，并从可用的函数实例系列中选择最佳函数。

如果未使用符号功能，那么在某些情况下，链接编辑器无需过程链接表项即可解析代码引用。例如，在动态可执行文件中，可以在链接编辑时内部绑定可执行文件中存在的函数引用。共享目标文件中的隐藏函数和受保护函数也可以在链接编辑时内部绑定。在这些情况下，通常不需要运行时链接程序来解析这些函数的引用。

不过，如果使用了符号功能，就必须根据过程链接表项解析函数。为了使运行时链接程序参与适当的函数，同时维护只读文本段，此项是必需的。此机制导致功能函数的所有调用都通过过程链接表项间接进行。如果不使用符号功能，可能不一定要使用此间接方式。因此，在功能函数的调用开销与通过将功能函数用于其缺省对应项所获得的任何性能提高之间，应进行权衡。

注 - 虽然必须通过过程链接表项访问功能函数，但仍可将函数定义为隐藏函数或受保护函数。运行时链接程序会遵从这些可见性状态，并限制对这些函数的任何绑定。此行为导致的绑定与未将符号功能关联到函数时所生成的绑定相同。从外部目标文件无法绑定到隐藏函数。目标文件中对受保护函数的引用只能绑定到同一个目标文件中。

创建符号功能数据项系列

在同一个目标文件中可以提供多个初始化数据实例，其中每个实例均特定于一个系统。不过，通过功能接口提供此类数据通常更为简单，因此建议使用此方法。请参见“创建符号功能函数系列” [59]。在单个可执行文件中提供多个初始化数据实例需要特别小心。

以下示例将初始化 foo.c 中的数据项 foo，使其指向一个计算机名称字符串。此文件可针对不同计算机进行编译，每个实例均标识有计算机功能。从文件 bar.c 中的 bar（）进行对此数据项的引用。然后，通过将 bar（） 与 foo 的两个功能实例组合，创建共享目标文件 foobar.so.1。

```
$ cat foo.c
char *foo = MACHINE;
$ cat bar.c
#include <stdio.h>

extern char *foo = MACHINE;

void bar()
{
        (void) printf("machine: %s\n", foo);
}

$ elfdump -H foobar.so.1

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
   index  tag               value
     [1]  CA_SUNW_ID        sun4u
     [2]  CA_SUNW_MACH      sun4u

  Symbols:
   index    value     size  type bind oth ver shndx          name
     [1]  0x108d4      0x4  OBJT LOCL  D    0 .data          foo%sun4u

 Symbol Capabilities:
   index  tag               value
     [4]  CA_SUNW_ID        sun4v
     [5]  CA_SUNW_MACH      sun4v

  Symbols:
   index    value     size  type bind oth ver shndx          name
     [2]  0x108d8      0x4  OBJT LOCL  D    0 .data          foo%sun4v
```

应用程序可以引用 bar（），且运行时链接程序绑定到与底层系统关联的 foo 的实例。

```
$ uname -m
sun4u
$ main
machine: sun4u
```

此代码能否正确运行取决于已编译为与位置无关的代码，就像正常情况下可共享目标文件中的代码一样。请参见“与位置无关的代码” [162]。与位置无关的数据引用为间接引用，运行时链接程序可通过此引用来查找所需的引用并更新数据段元素。这种数据库重定位更新会将文本段保留为只读。

不过，可执行文件中的代码通常为位置相关代码。此外，可执行文件中的数据引用在链接编辑时进行绑定。在可执行文件中，符号功能数据引用必须通过全局数据项保持未解析状态，以便运行时链接程序从符号功能系列中进行选择。如果将上一个示例 `bar.c` 中来自 `bar（）` 的引用编译为位置相关的代码，则可执行文件的文本段必须在运行时进行重定位。缺省情况下，此状态会导致致命链接时错误。

```
$ cc -o main main.c bar.c foo.o foo.1.o foo.2.o ...
warning: Text relocation remains                referenced
    against symbol                  offset      in file
foo                                 0x0         bar.o
foo                                 0x8         bar.o
```

解决此错误状态的一种方法是将 `bar.c` 编译为与位置无关。但是，请注意可执行文件中所有符号功能数据项的所有引用都必须编译为与位置无关，否则此方法无效。

虽然可以使用符号功能机制访问数据，但如何使数据项成为目标文件公共接口的一部分是个问题。一种更为灵活的替代模式是将每个数据项封装在符号功能函数中。此函数提供了单独的数据访问方法。将数据隐藏在符号功能函数之后具有很大好处，可允许将数据定义为静态数据并保持专用状态。可以对上一个示例编码以使用符号功能函数。

```
$ cat foobar.c
cat bar.c
#include <stdio.h>

static char *foo = MACHINE;

void bar()
{
        (void) printf("machine: %s\n", foo);
}
$ elfdump -H main

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
   index  tag               value
     [1]  CA_SUNW_ID        sun4u
     [2]  CA_SUNW_MACH      sun4u

  Symbols:
   index    value     size  type bind oth ver shndx          name
     [1]  0x1111c     0x1c  FUNC LOCL  D    0 .text          bar%sun4u

 Symbol Capabilities:
   index  tag               value
     [4]  CA_SUNW_ID        sun4v
     [5]  CA_SUNW_MACH      sun4v
```

```
  Symbols:
   index    value     size  type bind oth ver shndx          name
     [2]  0x11138     0x1c  FUNC LOCL  D    0 .text          bar%sun4v

$ uname -m
sun4u
$ main
machine: sun4u
```

将目标文件功能转换为符号功能

理论上，编译器可以生成标识有符号功能的目标文件。如果编译器无法创建符号功能，链接编辑器会提供一个解决方案。

可以使用链接编辑器将定义目标文件功能的可重定位目标文件转换为定义符号功能的可重定位目标文件。任何功能数据节都可使用链接编辑器 `-z symbolcap` 选项转换为定义符号功能。目标文件中的所有全局函数都将转换为局部函数，并与符号功能关联。所有全局初始化数据项都将转换为局部数据项，并与符号功能关联。这些转换的符号附加有指定为目标文件功能组部分的任何功能标识符。如果未定义功能标识符，就会附加缺省的组名。

对于每个原始全局函数或初始化数据项，将创建一个全局引用。此引用与所有重定位要求关联，在最终结合此目标文件来创建动态目标文件时，此引用将绑定到缺省的全局符号。

注 `--z symbolcap` 选项适用于包含目标文件功能节的目标文件。对于已包含符号功能的可重定位目标文件，或同时包含目标文件功能和符号功能的可重定位目标文件，此选项无效。此设计允许链接编辑器将多个目标文件仅与那些包含受此选项影响的目标文件功能的目标文件组合。

在以下示例中，x86 可重定位目标文件包含两个全局函数 `foo（）` 和 `bar（）`。此目标文件已被编译为需要 `MMX` 和 `SSE` 硬件功能。在这些示例中，功能组已采用功能标识符项进行命名。此标识符名称会附加到经过转换的符号名称后。如果没有此显式标识符，链接编辑器会附加一个缺省的功能组名称。

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap

 Object Capabilities:
   index  tag               value
     [0]  CA_SUNW_ID        sse,mmx
     [1]  CA_SUNW_HW_1      0x840  [ SSE MMX ]

$ elfdump -s foo.o | egrep "foo|bar"
    [25]         0   0x21  FUNC GLOB  D    0 .text    foo
    [26]      0x24   0x1e  FUNC GLOB  D    0 .text    bar
```

```
$ elfdump -r foo.o | fgrep foo
  R_386_PLT32               0x38       .rel.text            foo
```

接下来将这个可重定位目标文件转换为符号功能可重定位目标文件。

```
$ ld -r -o foo.1.o -z symbolcap foo.o
$ elfdump -H foo.1.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
   index  tag               value
     [1]  CA_SUNW_ID        sse,mmx
     [2]  CA_SUNW_HW_1      0x840  [ SSE MMX ]

  Symbols:
   index    value     size  type bind oth ver shndx    name
    [25]        0     0x21  FUNC LOCL  D    0 .text    foo%sse,mmx
    [26]     0x24     0x1e  FUNC LOCL  D    0 .text    bar%sse,mmx

$ elfdump -s foo.1.o | egrep "foo|bar"
    [25]        0     0x21  FUNC LOCL  D    0 .text    foo%sse,mmx
    [26]     0x24     0x1e  FUNC LOCL  D    0 .text    bar%sse,mmx
    [37]        0        0  FUNC GLOB  D    0 UNDEF    foo
    [38]        0        0  FUNC GLOB  D    0 UNDEF    bar

$ elfdump -r foo.1.o | fgrep foo
  R_386_PLT32               0x38       .rel.text            foo
```

现在可以将此目标文件与包含相同函数实例的其他目标文件组合，与不同符号功能关联，以生成可执行文件或共享目标文件。此外，应当提供每个函数的缺省实例（即不与任何符号功能关联的实例）以引导每个功能系列。此缺省实例可供所有外部引用使用，并确保函数的实例在任何系统上均可用。

在运行时，`foo（）` 和 `bar（）` 的任何引用均定向到引导实例。不过，如果系统包含适当功能，运行时链接程序会从中选择最佳符号功能实例。

归档注意事项

归档库通常包含可重定位目标文件的集合。链接编辑器可以提取各个可重定位目标文件来解析未解析的符号引用。请参见“归档处理” [27]。

如果已将一系列功能可重定位目标文件添加到归档，任何对前置功能符号的引用只会提取定义该信号的通用可重定位目标文件。不会提取任何其他功能目标文件。

如果需要使用归档库部署功能目标文件，应创建单个功能系列可重定位目标文件。将任何功能目标文件和任何包含功能前置功能符号的通用目标文件合并为一个可重定位目标文件。将此包含整个功能系列集合的单个目标文件添加到归档中。

```
$ ld -r -o all.foo.o foo.o foo.1.o foo.2.o ....
$ ar -cr libfoo.o all.foo.o
```

功能系列试验

正常情况下，设计和生成目标文件是为了能在给定体系结构的所有系统上执行该目标文件。但是，经常要对具有特殊功能的个别系统进行优化。可以使用之前各节介绍的机制，用代码需要执行的功能来标识经过优化的代码。

要测试经过优化的实例，必须使用可提供所需功能的系统。对于每个系统，运行时链接程序会确定可用的功能，然后选择功能最强的实例。为帮助进行测试和实验，可以让运行时链接程序使用替代功能集合，而不是系统提供的功能集合。此外，可以指定应仅根据这些替代功能验证特定文件。

替代功能集合源自系统功能，可以重新初始化，也可以添加或删除功能。

以下环境变量系列可用于创建并计划使用替代功能集合。

LD_PLATCAP={*name*}
: 标识替代平台名称。

LD_MACHCAP={*name*}
: 标识替代计算机硬件名称。

LD_HWCAP=[+-]{*token* | [*index*]*number*},...
: 标识替代硬件功能值。

LD_SFCAP=[+-]{*token* | [*index*]*number*},...
: 标识替代软件功能值。

LD_CAP_FILES=*file*,...
: 标识应根据替代功能进行验证的文件。

功能环境变量 LD_PLATCAP 和 LD_MACHCAP 分别接受定义平台名称的字符串和定义计算机硬件名称的字符串。请参见“标识平台功能” [53]和“标识计算机功能” [54]。

功能环境变量 LD_HWCAP 和 LD_SFCAP 接受逗号分隔的标记列表作为功能的符号表现形式。请参见“标识硬件功能” [55]和“标识软件功能” [57]。标记也可以是数值。要为不同的掩码（例如 CA_SUNW_HW_1 和 CA_SUNW_HW_2）设置数值，可在数字前加上用方括号括起来的索引。例如，LD_HWCAP=[2]0x80 将 CA_SUNW_HW_2 设置为值 *0x80*。如果未指定索引，则采用 1。无效索引将被忽略。

"+" 前缀将导致将其后的功能添加到替代功能中。"-" 前缀将导致从替代功能中删除其后的功能。缺少 "+-" 将导致其后的功能替换替代功能。

删除某个功能会导致增加对要模拟的功能环境的限制。通常情况下，当功能实例系列可用时，还会提供一个非特定于功能的通用实例。因此，可以使用限制程度较高的功能环境来强制使用功能较弱的代码实例或通用代码实例。

添加功能会导致进一步强化要模拟的功能环境。应谨慎创建此环境，但可将其用于测试某个功能系列的框架。例如，可以创建函数系列以使用 *mapfile* 定义其预期功能。这些函数可以使用 printf(3C) 确认其执行情况。然后，采用各种功能组合验证并测试关联目标文件的创建。在对函数的实际功能要求进行编码之前，功能系列的此原型设计可以证明是有用的。不过，如果实例系列中的代码要求正确执行特定功能，但系统并未提供此功能，而是将其设置为替代功能，那么代码实例将无法正确执行。

如果建立替代功能集合时未使用 LD_CAP_FILES，那么会针对替代功能对进程的所有特定于功能的目标文件进行验证。此方法也应谨慎使用，因为许多系统目标文件要求正确执行系统功能。功能的任何更改都可能导致系统目标文件无法正确执行。

功能实验的最佳环境是使用可提供目标文件要使用的所有功能的系统。还应当使用 LD_CAP_FILES 来隔离您要实验的目标文件。随后可以使用 "-" 语法禁用功能，从而使得能够试验功能系列的各种实例。系统的真实功能会完全支持每个实例。

例如，假设有两个 x86 功能目标文件 libfoo.so 和 libbar.so。这些目标文件包含针对 SSE2 指令进行优化的功能函数、针对 MMX 指令进行优化的函数，以及不需要功能的通用函数。底层系统同时提供了 SSE2 和 MMX。缺省情况下，使用经过全面优化的 SSE2 函数。

可以使用 LD_HWCAP 定义删除 SSE2 功能，从而限制 libfoo.so 和 libbar.so 使用针对 MMX 指令进行优化的函数。定义 LD_CAP_FILES 最灵活的方法是使用所需文件的基名。

```
$ LD_HWCAP=-sse2 LD_CAP_FILES=libfoo.so,libbar.so  ./main
```

可以通过删除 SSE2 和 MMX 功能，进一步将 libfoo.so 和 libbar.so 限制为仅使用通用函数。

```
$ LD_HWCAP=-sse2,mmx LD_CAP_FILES=libfoo.so,libbar.so  ./main
```

注 - 可以使用运行时链接程序诊断来观察应用程序的可用功能以及已设置的所有替代功能。

```
$ LD_DEBUG=basic LD_HWCAP=-sse2,mmx,cx8  ./main
....
02328: hardware capabilities (CA_SUNW_HW_1) - 0x5c6f  \
    [ SSE3 SSE2 SSE FXSR MMX CMOV SEP CX8 TSC FPU ]
02328: alternative hardware capabilities (CA_SUNW_HW_1) - 0x4c2b  \
    [ SSE3 SSE FXSR CMOV SEP TSC FPU ]
....
```

重定位处理

创建了输出文件后，将输入文件中的所有数据节复制到新映像。将输入文件指定的所有重定位应用于输出映像。还要将必须生成的所有其他重定位信息写入新映像。

重定位处理通常很容易，但可能会出现错误状态并伴随有特定错误消息。有两种状态需要详细介绍。第一种状态涉及位置相关代码产生的文本重定位。“与位置无关的代码” [162]中对此状态有更详细的介绍。第二种状态可以由位移重定位产生，下一小节中将对位移重定位进行更全面的介绍。

位移重定位

如果将位移重定位应用于可以在复制重定位中使用的数据项，可能会出现错误状态。“复制重定位” [170]中介绍了有关复制重定位的详细信息。

已重定位的偏移和重定位目标保持分隔相同的位移时，位移重定位仍然有效。复制重定位是指将共享目标文件中的全局数据项复制到可执行文件的 .bss 中。此复制将保留可执行文件的只读文本段。如果对复制的数据应用了位移重定位，或者外部重定位是对复制的数据进行位移，位移重定位将变为无效。

有两个验证领域会尝试确定位移重定位问题。

- 第一个发生在生成共享目标文件时。对在复制的数据与位移重定位有关时可能产生问题的任何潜在复制可重定位数据项进行标记。在构造共享目标文件期间，链接编辑器无法确定可能会对数据项生成哪些外部引用。因此，只能标记潜在问题。
- 第二个发生在生成可执行文件时。对其数据已知与位移重定位有关的复制重定位的创建进行标记。

  然而，在链接编辑时创建共享目标文件期间，可能会完成应用于共享目标文件的位移重定位。这些位移重定位可能不会有标记。因此，对引用未标记共享目标文件的可执行文件进行的链接编辑将无法确定任何已复制重定位数据中的有效位移。

为了帮助诊断这些问题，链接编辑器使用一个或多个动态 DT_FLAGS_1 标志指示动态目标文件使用的位移重定位，如表 13-10 “ELF 动态标志 DT_FLAGS_1” 中所示。此外，可以使用链接编辑器的 -z verbose 选项显示可疑重定位。

例如，假设将创建具有全局数据项 bar[] 的共享目标文件，并且将对该数据项应用位移重定位。如果从动态可执行文件引用，那么此项可能已进行了复制重定位。链接编辑器使用以下内容对此情况提出警告。

```
$ cc -G -o libfoo.so.1 -z verbose -K pic foo.o
ld: warning: relocation warning: R_SPARC_DISP32: file foo.o: symbol foo: \
    displacement relocation to be applied to the symbol bar: at 0x194: \
    displacement relocation will be visible in output image
```

现在，如果创建一个引用数据项 bar[] 的应用程序，那么将创建复制重定位。此复制将导致位移重定位无效。由于链接编辑器可以清楚地发现此情况，所以不论是否使用 -z verbose 选项都将生成错误消息。

```
$ cc -o prog prog.o -L. -lfoo
ld: warning: relocation error: R_SPARC_DISP32: file foo.so: symbol foo: \
    displacement relocation applied to the symbol bar at: 0x194: \
```

```
the symbol bar is a copy relocated symbol
```

注 - 当 ldd(1) 与 -d 或 -r 选项配合使用时，使用位移动态标志可生成类似的重定位警告。

通过确保要重定位（偏移）的符号定义和重定位的符号目标都是局部的，可以避免这些错误状态。使用静态定义或链接编辑器的作用域设置方法。请参见“缩减符号作用域” [46]。通过使用功能接口访问共享目标文件中的数据可以避免此类型的重定位问题。

桩目标文件

桩目标文件是可提供与实际目标文件相同的链接接口但不包含代码或数据的共享目标文件，完全由 mapfile 生成。桩目标文件不能在运行时使用。不过，可以根据桩目标文件生成应用程序，桩目标文件可以提供在运行时要使用的实际目标文件名称。

在生成桩目标文件时，链接编辑器将忽略在命令行中指定的任何目标文件或库文件，无需存在这些文件即可生成桩目标文件。由于可以省略编译步骤，并且相对而言链接编辑器只需进行少量操作，因此可以很快生成桩目标文件。

桩目标文件可用于解决各种生成问题。

- 速度

  使用具有并行操作功能的 make 实用程序版本的现代计算机能够同时编译和链接多个目标文件，这可显著提高速度。但是，通常给定目标文件会依赖于其他目标文件，而且还有一个几乎其他所有目标文件都与之相关的核心目标文件集合。因此，必须对生成进行排序，使所有目标文件在被其他目标文件使用之前生成。这种排序不仅会制造瓶颈降低可能的并行操作数量，还会限制代码生成的整体速度。
- 复杂性/正确性

  在大型代码段中，各个目标文件之间可能存在大量的依赖项。这些目标文件的 makefile 或其他生成说明可能会变得非常复杂，从而难以理解或维护。依赖项可能会随系统的演变而发生变化。这会导致给定的 makefile 集合随着时间的推移慢慢变得不准确，从而导致出现竞用情况以及各种莫名其妙的生成故障。
- 依赖关系循环

  理想的情况是将代码作为协作共享目标文件进行组织，其中的每个目标文件会利用其他目标文件提供的资源。在必须在其他目标文件使用之前生成目标文件的环境中，即使运行时链接程序完全能装入并使用此类目标文件（如果它们能生成），也无法支持此类循环。

桩共享目标文件提供了一种替代方法，用于生成可避免上述问题的代码。对于该生成产生的所有共享目标文件，可以快速生成桩目标文件。然后，使用桩目标文件代替链接时的实际目标文件，按照任意顺序并行生成所有实际共享目标文件和可执行文件。之后，保留可执行文件和实际共享目标文件，并废弃桩共享目标文件。

桩目标文件由一个或多个必须共同满足以下要求的 mapfile 生成。

- 必须至少有一个 mapfile 指定 STUB_OBJECT 指令。请参见“STUB_OBJECT 指令” [195]。
- mapfile 中必须显式列出目标文件外部接口所包含的所有函数和数据符号。
- mapfile 必须使用符号作用域缩减 ('*')，从外部接口删除未显式列出的所有符号。请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。
- 从目标文件导出的所有全局数据必须在 mapfile 中具有 ASSERT 符号属性，以指定符号类型和大小。如果有多个符号引用同一个数据，那么这些符号中必须有一个符号的 ASSERT 指定 TYPE 和 SIZE 属性，其他符号必须使用 ALIAS 属性引用这个主符号。请参见“ASSERT 属性” [197]。

如果提供了此类 mapfile，则可使用相同的命令行为每个共享目标文件生成桩共享目标文件版本和实际共享目标文件版本。将 -z stub 选项添加到桩目标文件的链接编辑中，并在实际目标文件的链接编辑中省略该选项。

为说明上述方法，以下代码将实现一个名为 idx5 的共享目标文件，该共享目标文件可从包含 5 个整数元素的数组中导出数据。对每个元素进行初始化以包含从零开始的数组索引。此数据采用全局数组、使用弱绑定的替代别名数据符号形式，并通过功能接口提供。

```
$ cat idx5.c
int _idx5[5] = { 0, 1, 2, 3, 4 };
#pragma weak idx5 = _idx5

int
idx5_func(int index)
{
        if ((index < 0) || (index > 4))
                return (-1);
        return (_idx5[index]);
}
```

必须使用 mapfile 来描述此共享目标文件提供的接口。

```
$ cat mapfile
$mapfile_version 2
STUB_OBJECT;
SYMBOL_SCOPE {
        _idx5 {
                ASSERT { TYPE=data; SIZE=4[5] };
        };
        idx5 {
                ASSERT { BINDING=weak; ALIAS=_idx5 };
        };
        idx5_func;
    local:
        *;
};
```

以下主程序用于打印 idx5 共享目标文件中的所有可用索引值。

```
$ cat main.c
#include <stdio.h>

extern int _idx5[5], idx5[5], idx5_func(int);

int
main(int argc, char **argv)
{
        int i;
        for (i = 0; i < 5; i++)
                (void) printf("[%d] %d %d %d\n",
                    i, _idx5[i], idx5[i], idx5_func(i));
        return (0);
}
```

以下命令在名为 stublib 的子目录中创建此共享目标文件的桩目标文件版本。elfdump 命令用于检验生成的目标文件是否为桩目标文件。用于生成桩目标文件的命令不同于生成实际目标文件所用命令，其差别仅在于前者添加了 -z stub 选项，并使用了不同的输出文件名。这说明很容易将桩目标文件生成代码添加到现有代码中。

```
$ cc -Kpic -G -M mapfile -h libidx5.so.1 idx5.c -o stublib/libidx5.so.1 -zstub
$ ln -s libidx5.so.1 stublib/libidx5.so
$ elfdump -d stublib/libidx5.so | grep STUB
    [11]  FLAGS_1           0x4000000           [ STUB ]
```

现在可以使用桩目标文件代替实际共享目标文件，并设置会在运行时查找实际目标文件的运行路径，以此生成主程序。不过，由于实际目标文件尚未生成，此程序还不能运行。让系统装入桩目标文件的尝试将被拒绝，因为运行时链接程序知道桩目标文件缺少实际目标文件中的实际代码和数据，因而无法执行。

```
$ cc main.c -L stublib -R '$ORIGIN/lib' -lidx5 -lc
$ ./a.out
ld.so.1: a.out: fatal: libidx5.so.1: open failed: No such file or directory
Killed
$ LD_PRELOAD=stublib/libidx5.so.1  ./a.out
ld.so.1: a.out: fatal: stublib/libidx5.so.1: stub shared object \
    cannot be used at runtime
Killed
```

使用生成桩目标文件时使用的命令生成实际目标文件。省略 -z stub 选项，并指定实际输出文件的路径。

```
$ cc -Kpic -G -M mapfile -h libidx5.so.1 idx5.c -o lib/libidx5.so.1
```

一旦在 lib 子目录下生成了实际目标文件，便可以运行程序。

```
$ ./a.out
[0] 0 0 0
[1] 1 1 1
[2] 2 2 2
[3] 3 3 3
[4] 4 4 4
```

使用桩目标文件隐藏过时的接口

库不断演变，有时原始功能会被证明为不符合需求。添加新功能，废弃旧功能，这已成为常态。如果要考虑向后兼容性，则必须维护库中的此类旧功能，以便于处理现有目标文件。但您可能不希望再次用到这些功能。桩目标文件可用于实施这一策略。可使用 *mapfile* STUB_ELIMINATE 标志标记某目标文件中的功能或数据，这些功能或数据会从桩目标文件中删除，但仍保留在实际目标文件中。这样便可防止链接到此桩目标文件的新代码使用这些废弃项，并鼓励要重写的代码使用首选接口。由于实际目标文件仍然包含这些项目，现有目标文件仍可使用它们。

上一节中的 libidx5 示例就阐述了这种情况。该库演示了如何从一个目标文件中导出全局数据。然而，导出的全局数据增加了动态链接的复杂性，因此最好避免出现这种情况。较好的办法是提供一个访问此类数据的函数，例如 libidx5 提供的 idx5_func() 函数。继续这个示例，可以使用 STUB_ELIMINATE 使链接到该桩的新代码不可使用全局数据，同时在实际目标文件中提供这些旧接口，以便于处理现有程序。

重写 mapfile 以将 STUB_ELIMINATE 应用于两个全局数据符号。将 STUB_ELIMINATE 应用于全局数据的一个益处是不再需要提供 ASSERT 指令来提供数据大小。在此例中，ASSERT 被注释掉。实际的 mapfile 可能会完全省略它。

```
$ cat better_mapfile
$mapfile_version 2
STUB_OBJECT;
SYMBOL_SCOPE {
        _idx5 {
                FLAGS=STUB_ELIMINATE;
                #ASSERT { TYPE=data; SIZE=4[5] };
        };
        idx5 {
                FLAGS=STUB_ELIMINATE;
                #ASSERT { BINDING=weak; ALIAS=_idx5 };
        };
        idx5_func;
    local:
        *;
};
```

新版测试程序仅使用功能接口。

```
$ cat better_main.c
#include    <stdio.h>

extern int idx5_func(int);

int
main(int argc, char **argv)
{
        int i;
        for (i = 0; i < 5; i++)
                (void) printf("[%d] %d\n", i, idx5_func(i));
```

```
        return (0);
}
```

保存旧的测试程序，用新的 mapfile 重建桩目标文件，然后重建此测试程序并链接到采用 STUB_ELIMINATE 的新桩目标文件。

```
$ cp a.out original_a.out
$ cc -Kpic -G -M better_mapfile -h libidx5.so.1 idx5.c -o stublib/libidx5.so.1 -zstub
$ cc better_main.c -o better_a.out -L stublib -R '$ORIGIN/lib' -lidx5 -lc
$ ./better_a.out
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
```

将无法再构建原始测试程序，因为桩库缺少必要的全局数据符号。然而，早先存在的使用这些全局数据符号的二进制文件仍可正常运行，因为实际库仍提供这些全局数据符号。

```
$ cc main.c -L stublib -R '$ORIGIN/lib' -lidx5 -lc
Undefined                       first referenced
 symbol                             in file
idx5                               main.o
_idx5                              main.o
ld: fatal: symbol referencing errors
$ ./original_a.out
[0] 0 0 0
[1] 1 1 1
[2] 2 2 2
[3] 3 3 3
[4] 4 4 4
```

辅助目标文件

缺省情况下，目标文件包含可分配和不可分配节。可分配节是包含可执行代码和运行时该代码需要的数据的节。不可分配节包含补充信息，在运行时执行目标文件时这些信息不是必需的。这些节用于支持调试器和其他观察工具的操作。在运行时操作系统不会将目标文件中的不可分配节装入内存，因此，无论其大小如何，都不会影响内存使用或运行时性能的其他方面。

为方便起见，可分配节和不可分配节通常保留在同一文件中。但是，在某些情况下，将这些节分开会很有用。

- 减小目标文件的大小以便提高通过广域网对其进行复制的速度。
- 支持对高度优化的代码进行细粒度调试需要大量的调试数据。在现代系统中，调试数据可能很轻易地就大于其所描述的代码。32 位目标文件的大小限制为 4 GB。在非常大的 32 位目标文件中，调试数据可能会导致超出限制，并阻止创建目标文件。

- 限制内部实现详细信息泄漏。

传统上，将不可分配节从目标文件中剥离，以解决这些问题。剥离很有效，但是会销毁以后可能需要的数据。而 Solaris 链接编辑器可以将不可分配节写入辅助目标文件。此功能可使用 `-z ancillary` 选项启用。

```
$ cc .... -z ancillary[=outfile] ....
```

缺省情况下，辅助文件和主输出目标文件同名，具有 `.anc` 文件扩展名。但是，可以将 *outfile* 值提供给 `-z ancillary` 选项来指定一个不同的名称。

指定了 `-z ancillary` 时，链接编辑器将执行以下操作。

- 将所有可分配节写入主目标文件。此外，将包含一个或多个设置了 `SHF_SUNW_PRIMARY` 节头标志的输入节的所有不可分配节写入主目标文件。
- 将所有其余的不可分配节写入辅助目标文件。
- 将以下不可分配节写入主目标文件和辅助目标文件。

`.shstrtab`	节名称字符串表。
`.symtab`	完整非动态符号表。
`.symtab_shndx`	与 `.symtab` 关联的符号表扩展索引节。
`.strtab`	与 `.symtab` 关联的非动态字符串表。
`.SUNW_ancillary`	包含标识主目标文件、辅助目标文件和要检查的目标文件所需的信息。

- 主目标文件和所有辅助目标文件包含相同的节头数组。每个节在每个文件中具有相同的节索引。
- 尽管主目标文件和辅助目标文件均定义了相同的节头，但大多数节的数据将写入单个文件中，如上所述。如果给定文件中不存在某个节的数据，则会设置 `SHF_SUNW_ABSENT` 节头标志，且 `sh_size` 字段为 0。

通过该组织，可以从主目标文件或辅助目标文件中获取节头的完整列表、完整的符号表以及主目标文件和辅助目标文件的完整列表。

以下示例说明了辅助目标文件的底层实现。辅助目标文件是通过将 `-z ancillary` 命令行选项添加到原本采用常规方式的编译中创建的。`file` 实用程序显示生成了名为 `a.out` 的可执行文件和名为 `a.out.anc` 的关联辅助目标文件。

```
$ cat hello.c
#include    <stdio.h>

int
main(int argc, char **argv)
{
```

```
        (void) printf("hello, world\n");
        return (0);
}
$ cc -g -zancillary hello.c
$ file a.out a.out.anc
a.out: ELF 32-bit LSB executable 80386 Version 1 [FPU], dynamically \
    linked, not stripped, ancillary object a.out.anc
a.out.anc: ELF 32-bit LSB ancillary 80386 Version 1, primary object a.out
$ ./a.out
hello world
```

生成的主目标文件是可以用平常方式执行的普通可执行文件。在运行时，该文件与生成过程中不使用辅助目标文件，然后使用 `strip` 或 `mcs` 命令剥离不可分配内容的可执行文件没有区别。

如前所述，主目标文件和辅助目标文件包含相同的节头。要查看其如何工作，使用 `elfdump` 实用程序显示这些节头并对其进行比较会有所帮助。下表显示了从前一链接编辑示例中选择的部分节头的信息。

索引	节名称	类型	主标志	辅助标志	主大小	辅助大小
13	.text	PROGBITS	ALLOC EXECINSTR	ALLOC EXECINSTR SUNW_ABSENT	0x131	0
20	.data	PROGBITS	WRITE ALLOC	WRITE ALLOC SUNW_ABSENT	0x4c	0
21	.symtab	SYMTAB	0	0	0x450	0x450
22	.strtab	STRTAB	STRINGS	STRINGS	0x1ad	0x1ad
24	.debug_info	PROGBITS	SUNW_ABSENT	0	0	0x1a7
28	.shstrtab	STRTAB	STRINGS	STRINGS	0x118	0x118
29	.SUNW_ancillary	SUNW_ancillary	0	0	0x30	0x30

大多数节的数据仅存在于这两个文件之一中，不存在于另一个文件中。不存在数据时，会设置 `SHF_SUNW_ABSENT` 节头标志。运行时需要的可分配节数据在主目标文件中。用于调试但运行时不需要的不可分配节的数据放置在辅助文件中。一小组不可分配节在两个文件中都完整地存在。这些是用于将主目标文件和辅助目标文件联系在一起的 `.SUNW_ancillary` 节、节名称字符串表 `.shstrtab` 、符号表 `.symtab` 及其关联的字符串表 `.strtab`。

可以从主目标文件中剥离符号表。遇到不带符号表的目标文件的调试器可以使用 `.SUNW_ancillary` 节查找辅助目标文件并访问其中包含的符号。

主目标文件和所有相关的辅助目标文件均包含一个允许标识所有目标文件并将其关联在一起的 `.SUNW_ancillary` 节。

```
$ elfdump -T SUNW_ancillary a.out a.out.anc
a.out:
```

```
Ancillary Section:  .SUNW_ancillary
   index  tag                    value
     [0]  ANC_SUNW_CHECKSUM     0x8724
     [1]  ANC_SUNW_MEMBER       0x1         a.out
     [2]  ANC_SUNW_CHECKSUM     0x8724
     [3]  ANC_SUNW_MEMBER       0x1a3       a.out.anc
     [4]  ANC_SUNW_CHECKSUM     0xfbe2
     [5]  ANC_SUNW_NULL         0

a.out.anc:
Ancillary Section:  .SUNW_ancillary
   index  tag                    value
     [0]  ANC_SUNW_CHECKSUM     0xfbe2
     [1]  ANC_SUNW_MEMBER       0x1         a.out
     [2]  ANC_SUNW_CHECKSUM     0x8724
     [3]  ANC_SUNW_MEMBER       0x1a3       a.out.anc
     [4]  ANC_SUNW_CHECKSUM     0xfbe2
     [5]  ANC_SUNW_NULL         0
```

两个目标文件的辅助节包含相同数量的元素，除第一个元素外均相同。每个目标文件（从主目标文件开始）都是以下述方式引入的：首先是用于指定文件名的 MEMBER 元素，后面跟着是用于标识目标文件的 CHECKSUM。在此示例中，主目标文件为 a.out，其校验和为 0x8724。辅助目标文件为 a.out.anc，其校验和为 0xfbe2。.SUNW_ancillary 节中的第一个元素（位于主目标文件的 MEMBER 元素之前）始终是 CHECKSUM 元素，它包含要检查的文件的校验和。

- 目标文件中存在 .SUNW_ancillary 节表示目标文件具有关联的辅助目标文件。
- 主目标文件和所有关联的辅助目标文件的名称可以从任何一个文件中的辅助节中获得。
- 通过将第一个校验和值与接下来的每个成员的校验和进行比较，可以从较大的一组文件中确定要检查哪个文件。

注 - 链接编辑器不将辅助目标文件作为输入文件进行读取。如果用 -z ancillary 选项创建可重定位目标文件，之后引用此生成的目标文件生成另一个目标文件，则辅助目标文件中的节将不会传播到最终的目标文件中。

调试器访问及辅助目标文件使用

调试器和其他观察工具必须合并在主目标文件和辅助目标文件中找到的信息，以便生成目标文件的完整视图。这等同于处理单个文件中的信息。包含相同节头的主目标文件和辅助目标文件以及单个符号表可简化该合并。

调试器可以使用以下步骤组合这些文件中包含的信息。

1. 从主目标文件或任何辅助目标文件开始，查找 .SUNW_ancillary 节。存在此节指示该目标文件是辅助组的一部分，其中包含的信息可用于获取完整文件列表以及确定这些文件中的哪一个是当前检查的文件。

2. 使用要检查的目标文件中的节头数组作为初始模板，在内存中创建一个节头数组。
3. 依次打开并读取 .SUNW_ancillary 节标识的每个文件。对于每个文件，使用不包含 SHF_SUNW_ABSENT 标志集的每个节的信息填充内存中的节头数组。

结果是节头的一个完整内存中副本，其中包含指向所有节的数据的指针。在获得该信息后，调试器会像在使用单个文件时一样继续访问和控制正在运行的程序。

注 - 辅助目标文件的 ELF 定义提供了单个主目标文件和任意数量的辅助目标文件。当前，Oracle Solaris 链接编辑器仅生成单个包含所有不可分配节的辅助目标文件。这种情况以后可能会改变。应该编写调试器和其他观察工具来处理多个辅助目标文件的一般情况。

压缩调试节

如“辅助目标文件” [74]中所述，目标文件同时包含可分配和不可分配的节。可分配节是包含可执行代码和运行时该代码需要的数据的节。不可分配节包含补充信息，在运行时执行目标文件时这些信息不是必需的。这些节支持调试器和其他观察工具的操作，（非正式情况下）也称为调试节。

根据请求的调试信息级别，调试节相对于其说明的代码来说可能会变得非常大。将这些节写入单独文件的辅助目标文件提供了一种处理这些大型节的机制。而压缩调试节提供了第二种补充性的方法，可减小调试节大小。

调试节是使用行业标准的 ZLIB 压缩库进行压缩的。可在以下网址找到 ZLIB 的文档：http://www.zlib.net/。

链接编辑器可识别输入目标文件中的压缩调试节，并可自动解压缩这些节。此操作对链接编辑器的用户是透明的，且无需特殊操作。

缺省情况下，链接编辑器不压缩输出目标文件中的调试节。使用 -z compress-debug-sections 选项启用对输出文件中调试节的压缩。

```
$ cc .... -z compress-sections[=cmp-type] ....
```

可识别 *cmp-type* 的以下值。

none	不进行压缩。此选项等同于未指定 -z compress-sections 选项。
zlib	使用 ZLIB 压缩对候选节进行压缩。生成的输出节将设置 SHF_COMPRESSED 节标志来标识使用了压缩。
zlib-gnu	使用 ZLIB 压缩、使用 GNU 节压缩格式来压缩所有候选节。此格式要求候选节具有以 .debug 开头的名称。生成的输出节将重命名为以 .zdebug 开头来标识使用了压缩。

如果省略了 *cmp-type*，则使用 zlib 样式。

对任何节进行压缩时，如果压缩格式下的大小超过原始未压缩数据，则此压缩将被静默跳过。

要成为压缩候选项，节必须是不可分配的并且属于以下类之一。

annotate　注释节提供由内存访问工具和与覆盖范围相关的工具使用的信息。这些部分通过 SHT_SUNW_ANNOTATE 节类型来进行标识。

debug　调试节通过 .compcom、.line、.stab*、.debug* 或 .zdebug* 节名称来进行标识。这些节还通过 SHT_PROGBITS 或 SHT_SUNW_DEBUG* 节类型进行标识。

zlib-gnu 压缩类型仅限于名称以 .debug 开头的节。使用 zlib-gnu 时，将不压缩本将是压缩候选项的节。对于 zlib 和 zlib-gnu 样式，底层 ZLIB 压缩是相同的，这两种格式对于给定输入节提供相同压缩量。这两种样式的不同之处在于候选节选择、压缩头的格式以及如何标识压缩的节。请参见“节压缩” [295]。除非特别要求使用 zlib-gnu 样式，否则推荐使用更通用的缺省 zlib 样式。

以下程序演示如何使用压缩调试节。为了进行比较，程序构建了两次，一次无压缩，另一次进行了压缩。

```
$ cat hello.c
#include    <stdio.h>

int
main(int argc, char **argv)
{
        (void) printf("hello, world\n");
        return (0);
}
% cc -g hello.c -o a.out.uncompressed
% cc -g hello.c -o a.out.compressed -z compress-sections
```

现在可比较未压缩调试节和压缩调试节的节头。

```
$ elfdump -c a.out.uncompressed
....
Section Header[24]:  sh_name: .debug_info
    sh_addr:      0               sh_flags:   0
    sh_size:      0x17b           sh_type:    [ SHT_PROGBITS ]
    ....

Section Header[25]:  sh_name: .debug_line
    sh_addr:      0               sh_flags:   0
    sh_size:      0x4f            sh_type:    [ SHT_PROGBITS ]
    ....

Section Header[26]:  sh_name: .debug_abbrev
    sh_addr:      0               sh_flags:   0
```

```
    sh_size:      0x7c            sh_type:    [ SHT_PROGBITS ]
    ....

Section Header[27]:  sh_name: .debug_pubnames
    sh_addr:      0               sh_flags:   0
    sh_size:      0x1b            sh_type:    [ SHT_PROGBITS ]
    ....

$ elfdump -c a.out.compressed
....
Section Header[24]:  sh_name: .debug_info
    sh_addr:      0               sh_flags:   [ SHF_COMPRESSED ]
    sh_size:      0x14f           sh_type:    [ SHT_PROGBITS ]
    ....
    ch_size:      0x196           ch_type:    [ ELFCOMPRESS_ZLIB ]
    ch_addralign: 0x1

Section Header[25]:  sh_name: .debug_line
    sh_addr:      0               sh_flags:   0
    sh_size:      0x4f            sh_type:    [ SHT_PROGBITS ]
    ....

Section Header[26]:  sh_name: .debug_abbrev
    sh_addr:      0               sh_flags:   [ SHF_COMPRESSED ]
    sh_size:      0x79            sh_type:    [ SHT_PROGBITS ]
    ....
    ch_size:      0x7c            ch_type:    [ ELFCOMPRESS_ZLIB ]
    ch_addralign: 0x1

Section Header[27]:  sh_name: .debug_pubnames
    sh_addr:      0               sh_flags:   0
    sh_size:      0x1b            sh_type:    [ SHT_PROGBITS ]
    ....
```

各个压缩节 .debug_info 和 .debug_abbrev 都使用 SHF_COMPRESSED 节标志进行标识。此外，节头信息带有压缩头结构信息。ch_size 和 ch_addralign 字段提供未压缩数据的大小和对齐要求。请参见“节压缩” [295]。

.debug_line 和 .debug_pubnames 节的压缩格式大于其原始未压缩格式，因此保持不压缩的状态。

压缩的成本和益处

压缩调试节的主要益处是减小了目标文件的大小。然而，压缩也造成了所有开发阶段都需要额外的运行时间和内存使用

- 编译器必须生成各个未压缩格式的调试节，为压缩版本分配额外的内存，并执行压缩。
- 读取输入目标文件时，链接编辑器必须将压缩数据读入内存，分配额外的内存以保留解压缩后的数据，并执行解压缩。

- 输出时，链接编辑器必须创建生成的调试节的未压缩版本。如有压缩请求，需要额外的内存和时间来创建压缩版本。
- 当一个调试器读取带压缩调试节的目标文件时，此调试器必须分配额外的内存以保留解压缩后的数据，并执行解压缩。

此外，压缩调试节可包含较小的文件，但不适用于储存大量信息。一个常见的示例是 32 位目标文件，由于它们使用 32 位文件的偏移和大小，因此其大小基本限制于 4 GB。有时可假定压缩调试数据可能允许生成更多调试信息。然而，32 位调试数据的格式还包含 32 位偏移，因此从逻辑上来说，未压缩格式的大小不可超过 4 GB。

因此，建议压缩调试节不要用于一般开发，否则编译/链接/调试周期的速度方面的劣势会盖过小型调试数据的优势。当磁盘空间较少，或针对的是广泛复制且很少调试的生产目标文件时，压缩调试节会有一些优势。

父目标文件

提供可扩展功能的程序通常使用在运行时使用 `dlopen（）` 函数装入的共享目标文件。这些共享目标文件通常称为插件，它们提供了一种用于扩展核心系统的功能的灵活方法。装入插件的目标文件称为父目标文件。

父目标文件装入插件并从插件内访问函数和数据。对于父目标文件，通过插件提供要使用的函数和数据很常见。以下父目标文件和插件源文件对此进行了说明。为了利用插件，此处父目标文件提供了名为 `parent_callback（）` 的函数。插件为父目标文件提供了名为 `plugin_func（）` 的函数以供调用。

```
$ cat main.c
#include    <stdio.h>
#include    <dlfcn.h>
#include    <link.h>

void
parent_callback(void)
{
        (void) printf("plugin_func() has called parent_callback()\n");
}

int
main(int argc, char **argv)
{
        typedef void plugin_func_t(void);

        void            *hdl;
        plugin_func_t   *plugin_func;

        if (argc != 2) {
                (void) fprintf(stderr, "usage: main plugin\n");
                return (1);
        }
```

```
        if ((hdl = dlopen(argv[1], RTLD_LAZY)) == NULL) {
                (void) fprintf(stderr, "unable to load plugin: %s\n",
      dlerror());
                return (1);
        }

        plugin_func = (plugin_func_t *) dlsym(hdl, "plugin_func");
        if (plugin_func == NULL) {
                (void) fprintf(stderr, "unable to find plugin_func: %s\n",
                    dlerror());
                return (1);
        }

        (*plugin_func)();

        return (0);
}

$ cat plugin.c
#include    <stdio.h>

extern void parent_callback(void);

void
plugin_func(void)
{
        (void) printf("parent has called plugin_func() from plugin.so\n");
        parent_callback();
}

$ cc -o main main.c -lc
$ cc -Kpic -G -o plugin.so plugin.c -lc
$ ./main ./plugin.so
parent has called plugin_func() from plugin.so
plugin_func() has called parent_callback()
```

构建任何共享目标文件时，为了确保目标文件指定了其所有依赖项，建议使用 `-z defs` 选项。但是，由于父目标文件中有未满足的符号，使用 `-z defs` 会阻止插件目标文件进行链接。

```
$ cc -zdefs -Kpic -G -o plugin.so plugin.c -lc
Undefined                       first referenced
 symbol                             in file
parent_callback                    plugin.o
ld: fatal: symbol referencing errors
```

可以使用 *mapfile* 向链接编辑指定 `parent_callback（）` 符号由父目标文件提供。

```
$ cat plugin.mapfile
$mapfile_version 2

SYMBOL_SCOPE {
        global:
```

```
            parent_callback         { FLAGS = PARENT };
};
$ cc -zdefs -Mplugin.mapfile -Kpic -G -o plugin.so plugin.c -lc
```

构建插件的首选解决方案是使用 -z parent 选项向插件提供对父目标文件的直接访问。使用 -z parent 而非 *mapfile* 的另一好处是父目标文件的名称将记录在插件的动态节中，并由 file 实用程序显示。

```
$ cc -zdefs -zparent=main -Kpic -G -o plugin.so plugin.c -lc
$ elfdump -d plugin.so | grep PARENT
     [0]  SUNW_PARENT       0xcc               main
$ file plugin.so
plugin.so: ELF 32-bit LSB dynamic lib 80386 Version 1, parent main, \
    dynamically linked, not stripped
```

调试帮助

链接编辑器提供了调试功能，允许您详细地跟踪链接编辑过程。此功能有助于了解并调试应用程序和库的链接编辑。使用此功能显示的信息类型应保持不变。不过，信息的确切格式可能随发行版的不同而有所变化。

如果不太了解 ELF 格式，可能会不熟悉某些调试输出。不过，您可能对其中许多方面不太感兴趣。

使用 -D 选项可以启用调试。必须使用一个或多个标记扩充此选项，以指示需要的调试类型。

在命令行中键入 -D help，可以显示 -D 的可用标记。

```
$ ld -Dhelp
```

缺省情况下，会将所有调试输出发送到标准错误输出文件 stderr。可以使用 output 标记将调试输出定向到其他文件。例如，可以将帮助文本输出到名为 ld-debug.txt 的文件中。

```
$ ld -Dhelp,output=ld-debug.txt
```

大多数编译器驱动程序会为 -D 选项指定不同的含义，通常是为了定义预处理宏。可以使用 LD_OPTIONS 环境变量跳过编译器驱动程序，将 -D 选项直接提供给链接编辑器。

以下示例说明了如何跟踪输入文件。此语法可用于确定在链接编辑中使用的库。使用此语法也可以显示从归档中提取的目标文件。

```
$ LD_OPTIONS=-Dfiles  cc -o prog main.o -L. -lfoo
....
debug: file=main.o  [ ET_REL ]
debug: file=./libfoo.a  [ archive ]
```

```
debug: file=./libfoo.a(foo.o)  [ ET_REL ]
debug: file=./libfoo.a  [ archive ] (again)
....
```

其中，成员 foo.o 是从归档库 libfoo.a 中提取的，目的是满足对 prog 的链接编辑。请注意，对归档搜索了两次，以验证提取 foo.o 时没有提取其他可重定位目标文件。多个 "(again)" 诊断指示该归档文件是使用 lorder(1) 和 tsort(1) 进行排序的候选归档文件。

使用 symbols 标记，可以确定导致提取归档成员的符号和进行初始符号引用的目标文件。

```
$ LD_OPTIONS=-Dsymbols  cc -o prog main.o -L. -lfoo
....
debug: symbol table processing; input file=main.o  [ ET_REL ]
....
debug: symbol[7]=foo  (global); adding
debug:
debug: symbol table processing; input file=./libfoo.a  [ archive ]
debug: archive[0]=bar
debug: archive[1]=foo  (foo.o) resolves undefined or tentative symbol
debug:
debug: symbol table processing; input file=./libfoo(foo.o)  [ ET_REL ]
....
```

符号 foo 由 main.o 引用。将此符号添加到链接编辑器的内部符号表中。此符号引用会导致从归档 libfoo.a 中提取可重定位目标文件 foo.o。

注 - 本文档中对此输出进行了简化。

可以使用 detail 标记和 symbols 标记观察在输入文件处理期间符号解析的详细信息。

```
$ LD_OPTIONS=-Dsymbols,detail  cc -o prog main.o -L. -lfoo
....
debug: symbol table processing; input file=main.o  [ ET_REL ]
....
debug: symbol[7]=foo  (global); adding
debug:   entered  0x000000 0x000000 NOTY GLOB  UNDEF REF_REL_NEED
debug:
debug: symbol table processing; input file=./libfoo.a  [ archive ]
debug: archive[0]=bar
debug: archive[1]=foo  (foo.o) resolves undefined or tentative symbol
debug:
debug: symbol table processing; input file=./libfoo.a(foo.o)  [ ET_REL ]
debug: symbol[1]=foo.c
....
debug: symbol[7]=bar  (global); adding
debug:   entered  0x000000 0x000004 OBJT GLOB  3     REF_REL_NEED
debug: symbol[8]=foo  (global); resolving [7][0]
debug:       old  0x000000 0x000000 NOTY GLOB  UNDEF main.o
debug:       new  0x000000 0x000024 FUNC GLOB  2     ./libfoo.a(foo.o)
debug:  resolved  0x000000 0x000024 FUNC GLOB  2     REF_REL_NEED
```

....

main.o 中的原始未定义符号 foo 已被所提取的归档成员 foo.o 中的符号定义所覆盖。详细的符号信息反映每个符号的属性。

在上一个示例中，可以看到使用一些调试标记可产生大量输出。要监视部分输入文件的活动，可直接将 -D 选项放置在链接编辑命令行中。可以通过切换打开和关闭此选项。在以下示例中，只有在处理库 libbar 期间，才会打开符号处理的显示功能。

```
$ ld .... -o prog main.o -L. -Dsymbols -lbar -D!symbols ....
```

注 - 要获取链接编辑命令行，可能必须从使用的任何驱动程序展开编译行。请参见“使用编译器驱动程序” [24]。

♦ ♦ ♦ 第 3 章

3 运行时链接程序

在动态可执行文件的初始化和执行过程中，将调用解释程序来完成将应用程序绑定到其依赖项的操作。在 Oracle Solaris OS 中，此解释程序称为运行时链接程序。

在对动态可执行文件进行链接编辑过程中，将会创建一个特殊的 `.interp` 节以及关联的程序头。此节包含用于指定程序的解释程序的路径名。链接编辑器提供的缺省名称是运行时链接程序的名称：`/usr/lib/ld.so.1`（对于 32 位可执行文件）和 `/usr/lib/64/ld.so.1`（对于 64 位可执行文件）。

注 - `ld.so.1` 是共享目标文件的特例。此处使用的版本号为 1。但是，以后的 Oracle Solaris OS 发行版可能会提供更高的版本号。

在执行动态目标文件的过程中，内核将装入该文件并读取程序头信息。请参见“程序头” [343]。内核可根据此信息查找所需解释程序的名称。内核会装入并将控制权转交给此解释程序，同时传递足够的信息以便解释程序继续执行应用程序。

除了初始化应用程序以外，运行时链接程序还会提供用来使应用程序扩展其地址空间的服务。此过程涉及装入其他目标文件以及绑定到这些目标文件提供的符号。

运行时链接程序执行以下操作：

- 分析可执行文件的动态信息节 (`.dynamic`) 并确定所需的依赖项。
- 查找并装入这些依赖项，分析其动态信息节以确定是否需要其他依赖项。
- 执行所有必需的重定位以绑定这些目标文件，为执行进程做好准备。
- 调用这些依赖项提供的所有初始化函数。
- 将控制权移交给应用程序。
- 可在应用程序执行时调用，以执行延迟函数绑定。
- 可由应用程序调用以使用 `dlopen(3C)` 获取其他目标文件，并使用 `dlsym(3C)` 绑定到这些目标文件中的符号。

共享目标文件依赖项

当运行时链接程序为某一程序创建内存段时，依赖项会说明提供该程序的服务时所需的共享目标文件。通过重复连接引用的共享目标文件及其依赖项，运行时链接程序可生成完整的进程映像。

注 - 即使在依赖项列表中多次引用了某个共享目标文件，运行时链接程序也只会将该目标文件连接到进程一次。

查找共享目标文件依赖项

链接动态可执行文件时，会显式引用一个或多个共享目标文件。这些目标文件作为依赖项记录在动态可执行文件中。

运行时链接程序将使用此依赖项信息来查找并装入关联目标文件。这些依赖项的处理顺序与可执行文件的链接编辑期间对依赖项的引用顺序相同。

装入所有动态可执行文件的依赖项之后，将按依赖项的装入顺序检查每个依赖项，以找出其他依赖项。此过程会一直继续，直至找到并装入所有依赖项。此方法将导致所有依赖项按广度优先顺序排序。

运行时链接程序搜索的目录

运行时链接程序在两个缺省位置中查找依赖项。在处理 32 位目标文件时，缺省位置为 `/lib` 和 `/usr/lib`。在处理 64 位目标文件时，缺省位置为 `/lib/64` 和 `/usr/lib/64`。指定为简单文件名的任何依赖项都使用这些缺省目录名称作为前缀。生成的路径名用于查找实际文件。

使用 `ldd(1)` 可以显示动态可执行文件或共享目标文件的依赖项。例如，文件 `/usr/bin/cat` 具有下列依赖项：

```
$ ldd /usr/bin/cat
        libc.so.1 =>     /lib/libc.so.1
        libm.so.2 =>     /lib/libm.so.2
```

文件 `/usr/bin/cat` 具有依赖项文件 `libc.so.1` 和 `libm.so.2`，即需要这些文件。

可以使用 `elfdump(1)` 检查目标文件中记录的依赖项。使用此命令可以显示文件的 `.dynamic` 节，并查找具有 `NEEDED` 标记的项。在以下示例中，前面的 `ldd(1)` 示例中显示的依赖项 `libm.so.2` 没有记录在文件 `/usr/bin/cat` 中。`ldd(1)` 显示了指定文件的全部依赖项，而 `libm.so.2` 实际上是 `/lib/libc.so.1` 的依赖项。

```
$ elfdump -d /usr/bin/cat

Dynamic Section:  .dynamic:
   index  tag              value
     [0]  NEEDED           0x211              libc.so.1
     ...
```

在前面的 elfdump(1) 示例中，依赖项表示为简单文件名。也就是说，名称中没有 "/"。使用简单文件名要求运行时链接程序根据一组缺省搜索规则生成路径名。包含嵌入 "/" 的文件名均按原样使用。

记录简单文件名是记录依赖项的一种最灵活的标准机制。链接编辑器的 -h 选项记录依赖项中的简单名称。请参见“命名约定” [123]和“记录共享目标文件名称” [124]。

通常，依赖项分布在 /lib 及 /usr/lib 或 /lib/64 及 /usr/lib/64 以外的目录中。如果动态可执行文件或共享目标文件需要在其他目录中查找依赖项，则必须显式指示运行时链接程序搜索此目录。

通过在链接编辑目标文件过程中记录运行路径，可以基于每个目标文件指定其他搜索路径。有关记录此信息的详细信息，请参见“运行时链接程序搜索的目录” [32]。

使用 elfdump(1) 可以显示运行路径记录。请参考包含 RUNPATH 标记的 .dynamic 项。在以下示例中，prog 具有 libfoo.so.1 的依赖项。运行时链接程序必须先搜索目录 /home/me/lib 和 /home/you/lib，然后在缺省位置中查找。

```
$ elfdump -d prog | egrep "NEEDED|RUNPATH"
     [1]  NEEDED        0x4ce             libfoo.so.1
     [3]  NEEDED        0x4f6             libc.so.1
    [21]  RUNPATH      0x210e             /home/me/lib:/home/you/lib
```

添加运行时链接程序搜索路径的另一种方法是设置环境变量 LD_LIBRARY_PATH。可以将该环境变量（在进程启动时即时对其分析）设置为一组以冒号分隔的目录。运行时链接程序会先搜索这些目录，然后搜索指定的任何运行路径或缺省目录。

这些环境变量非常适合在调试时使用，如将应用程序强制绑定到局部依赖项。在以下示例中，前面示例中的文件 prog 将绑定到当前工作目录中的 libfoo.so.1。

```
$ LD_LIBRARY_PATH=.  prog
```

尽管 LD_LIBRARY_PATH 作为一种影响运行时链接程序搜索路径的临时机制很有用，但强烈建议不要在生产软件中使用它。可引用此环境变量的所有动态可执行文件都将扩充其搜索路径。此扩充可能导致性能整体下降。另外，根据“使用环境变量” [32]和“运行时链接程序搜索的目录” [32]中的说明，LD_LIBRARY_PATH 还会影响链接编辑器。

环境搜索路径可能导致 64 位可执行文件搜索包含与要查找的名称匹配的 32 位库的路径。反之亦然。运行时链接程序会拒绝不匹配的 32 位库，并继续搜索有效的 64 位匹配项。如果未找到匹配项，则会生成一条错误消息。通过将 LD_DEBUG 环境变量设置为包含 files 标记，可以详细观察此拒绝。请参见“调试功能” [116]。

```
$ LD_LIBRARY_PATH=/lib/64 LD_DEBUG=files /usr/bin/ls
```

```
....
00283: file=libc.so.1;  needed by /usr/bin/ls
00283:
00283: file=/lib/64/libc.so.1  rejected: ELF class mismatch: 32–bit/64–bit
00283:
00283: file=/lib/libc.so.1  [ ELF ]; generating link map
00283:     dynamic:  0xef631180  base:  0xef580000  size:      0xb8000
00283:     entry:    0xef5a1240  phdr:  0xef580034  phnum:           3
00283:      lmid:           0x0
00283:
00283: file=/lib/libc.so.1;  analyzing  [ RTLD_GLOBAL  RTLD_LAZY ]
....
```

如果无法找到某个依赖项，ldd(1) 将指出无法找到该目标文件。如果尝试执行应用程序，则会导致运行时链接程序生成相应的错误消息：

```
$ ldd prog
        libfoo.so.1 =>   (file not found)
        libc.so.1 =>     /lib/libc.so.1
        libm.so.2 =>     /lib/libm.so.2
$ prog
ld.so.1: prog: fatal: libfoo.so.1: open failed: No such file or directory
```

配置缺省搜索路径

对于 32 位应用程序，运行时链接程序使用的缺省搜索路径为 /lib 和 /usr/lib。对于 64 位应用程序，缺省搜索路径为 /lib/64 和 /usr/lib/64。使用 crle(1) 实用程序创建的运行时配置文件可以管理这些搜索路径。在为生成时未使用适当运行路径的应用程序建立搜索路径时，此文件通常很有用。

对于 32 位应用程序，可在缺省位置 /var/ld/ld.config 构造配置文件；对于 64 位应用程序，则可在缺省位置 /var/ld/64/ld.config 构造配置文件。此文件影响系统中各自类型的所有应用程序。此外，也可在其他位置创建配置文件，并且可使用运行时链接程序的 LD_CONFIG 环境变量选择这些文件。在缺省位置安装配置文件之前测试该文件时，后一种方法会很有用。

动态字符串标记

运行时链接程序允许扩展各种动态字符串标记。这些标记适用于过滤器、运行路径和依赖项定义。

- $CAPABILITY – 指示一个可在其中查找提供不同功能的目标文件的目录。请参见“特定于功能的共享目标文件” [227]。
- $ISALIST – 扩展为可在此平台上执行的本机指令集。请参见“特定于指令集的共享目标文件” [229]。
- $ORIGIN – 提供当前目标文件的目录位置。请参见“查找关联的依赖项” [231]。

- $OSNAME – 扩展为操作系统的名称。请参见“特定于系统的共享目标文件” [231]。
- $OSREL – 扩展为操作系统发行版级别。请参见“特定于系统的共享目标文件” [231]。
- $PLATFORM – 扩展为当前计算机的处理器类型。请参见“特定于系统的共享目标文件” [231]。

重定位处理

运行时链接程序在装入应用程序所需的全部依赖项之后，将会处理每个目标文件并执行所有必需的重定位。

在目标文件的链接编辑过程中，随输入可重定位目标文件提供的任何重定位信息均会应用于输出文件。但是，在创建动态可执行文件或共享目标文件时，许多重定位无法在链接编辑时完成。这些重定位需要仅在目标文件装入内存时才知道的逻辑地址。在这种情况下，链接编辑器将在输出文件映像中生成新的重定位记录。然后，运行时链接程序必须处理这些新的重定位记录。

有关许多重定位类型的更详细说明，请参见“重定位” [316]。重定位存在两个基本类型。

- 非符号重定位
- 符号重定位

使用 elfdump(1) 可以显示目标文件的重定位记录。在以下示例中，文件 libbar.so.1 包含两条重定位记录，用于指示必须更新全局偏移表或 .got 节。

```
$ elfdump -r libbar.so.1

Relocation Section:  .rel.got:
    type                     offset            section       symbol
  R_SPARC_RELATIVE          0x10438            .rel.got
  R_SPARC_GLOB_DAT          0x1043c            .rel.got      foo
```

第一个重定位是一个简单的相对重定位，这可通过重定位类型以及没有符号引用看出。此重定位需要使用将目标文件装入内存的基本地址来更新关联的 .got 偏移。

第二个重定位需要符号 foo 的地址。要完成此重定位，运行时链接程序必须从动态可执行文件或其依赖项之一查找该符号。

重定位符号查找

运行时链接程序负责搜索目标文件在运行时所需的符号。通常，用户应该熟悉应用于动态可执行文件及其依赖项的缺省搜索模型，以及应用于通过 dlopen(3C) 获取的目标文件的缺省搜索模型。但是，目标文件的符号属性或是具体的绑定要求会导致符号查找结果具有更复杂的特性。

目标文件有两个属性会影响符号查找。第一个属性是请求目标文件的符号搜索作用域。第二个属性是进程中每个目标文件提供的符号可见性。

装入目标文件时，可将这些属性作为缺省属性应用。此外，也可将这些属性作为 dlopen(3C) 的特定模式提供。在某些情况下，可在生成目标文件时将这些属性记录在目标文件中。

目标文件可以定义一个全局搜索作用域和/或一个组搜索作用域。

`world`
: 目标文件可以在进程的任何其他全局目标文件中搜索符号。

`group`
: 目标文件可在同一组的任何目标文件中搜索符号。通过使用 dlopen(3C) 获取的目标文件或使用链接编辑器的 `-B group` 选项生成的目标文件创建的依赖项树构成一个唯一的组。

目标文件可以定义目标文件的导出符号是全局可见还是局部可见。

`global`
: 可在具有全局搜索作用域的任何目标文件中引用该目标文件的导出符号。

`local`
: 只能在构成同一组的其他目标文件中引用该目标文件的导出符号。

运行时符号搜索也可由符号可见性指定。指定了 `STV_SINGLETON` 可见性的符号不受任何符号搜索作用域的影响。所有的单件符号引用都绑定到进程中第一次出现的单件定义。请参见表 12-23 “ELF 符号可见性”。

符号查找的最简单形式将在下一节“缺省符号查找” [92]中进行概述。通常，符号属性由多种形式的 dlopen(3C) 使用。这些情况将在“符号查找” [108]中进行讨论。

动态目标文件使用直接绑定时，将提供替代的符号查找模型。此模型指示运行时链接程序直接在链接编辑时提供符号的目标文件中搜索符号。请参见第 6 章 直接绑定。

缺省符号查找

动态可执行文件及随其装入的所有依赖项都被指定了 *world*（全局）搜索作用域和 *global*（全局）符号可见性。针对动态可执行文件或随其装入的任何依赖项的缺省符号查找会导致搜索每个目标文件。运行时链接程序将从动态可执行文件开始，并按目标文件的装入顺序搜索每个依赖项。

ldd(1) 将按依赖项的装入顺序列出动态可执行文件的依赖项。例如，假定动态可执行文件 `prog` 将 `libfoo.so.1` 和 `libbar.so.1` 指定为其依赖项。

```
$ ldd prog
        libfoo.so.1 =>   /home/me/lib/libfoo.so.1
        libbar.so.1 =>   /home/me/lib/libbar.so.1
```

如果需要符号 bar 来执行重定位，运行时链接程序将首先在动态可执行文件 prog 中查找 bar。如果找不到符号，运行时链接程序随后会在共享目标文件 /home/me/lib/libfoo.so.1 中查找，最后在共享目标文件 /home/me/lib/libbar.so.1 中查找。

注 - 符号查找操作的开销可能很大，尤其是在符号名称大小和依赖项数目增加的情况下。这方面的性能将在第 7 章 生成目标文件以优化系统性能中详细介绍。有关替代查找模型，请参见第 6 章 直接绑定。

缺省重定位处理模型还允许转换为延迟装入 (lazy loading) 环境。如果在当前装入的目标文件中找不到某符号，则会处理所有暂挂的延迟装入目标文件，以尝试查找该符号。此装入是对尚未完整定义其依赖项的目标文件的补偿。但是，该补偿可能会破坏延迟装入的优点。

运行时插入

缺省情况下，运行时链接程序首先在动态可执行文件中搜索符号，然后在每个依赖项中进行搜索。使用此模型时，第一次出现的所需符号满足搜索要求。因此，如果同一符号存在多个实例，则会在所有其他实例中插入第一个实例。

“简单解析” [37]中概述了插入如何影响符号解析。“缩减符号作用域” [46]中提供了有关更改符号可见性，从而减少意外插入几率的机制。

注 - 指定了 STV_SINGLETON 可见性的符号提供了一种插入形式。所有的单件符号引用都绑定到进程中第一次出现的单件定义。请参见表 12-23 “ELF 符号可见性”。

如果目标文件被显式标识为插入项，则可以对每个目标文件强制执行插入。使用环境变量 LD_PRELOAD 装入或通过链接编辑器的 -z interpose 选项创建的任何目标文件都会标识为插入项。运行时链接程序搜索符号时，将在应用程序之后、任何其他依赖项之前搜索标识为插入项的任何目标文件。

仅当在进行任何进程重定位之前装入了插入项的情况下，才能保证可以使用插入项提供的所有接口。在重定位处理开始之前，将装入使用环境变量 LD_PRELOAD 提供的插入项，或作为应用程序的非延迟装入依赖项建立的插入项。启动重定位之后，引入进程中的插入项会降级为正常依赖项。如果插入项是延迟装入的，或者是由于使用 dlopen(3C) 而装入的，则插入项可能会降级。可使用 ldd(1) 来检测前一种类别。

```
$ ldd -Lr prog
        libc.so.1 =>     /lib/libc.so.1
        foo.so.2 =>      ./foo.so.2
        libmapmalloc.so.1 =>     /usr/lib/libmapmalloc.so.1
```

```
loading after relocation has started: interposition request \
    (DF_1_INTERPOSE) ignored: /usr/lib/libmapmalloc.so.1
```

注 - 如果链接编辑器在处理延迟装入的依赖项时遇到显式定义的插入项，则插入项将被记录为非延迟可装入依赖项。

可以使用 INTERPOSE mapfile 关键字将动态可执行文件中的单个符号定义为插入项。该机制使用 -z interpose 选项，因而更具选择性，并且针对随着依赖项发展而发生的逆向插入提供更好的隔离。请参见“定义显式插入” [153]。

执行重定位的时间

根据重定位的执行时间，重定位可分为两种类型。产生这种区别是由对已重定位偏移进行的引用的类型所致。

- 即时引用
- 延迟引用

即时引用指的是必须在装入目标文件后立即确定的重定位。这些引用通常是目标文件代码使用的数据项、函数指针，甚至是通过与位置相关的共享目标文件进行的函数调用。这些重定位无法向运行时链接程序提供有关何时引用重定位项的信息。因此，必须在装入目标文件时，并在应用程序获取或重新获取控制权之前执行所有即时重定位。

延迟引用指的是可以确定为目标文件执行的重定位。这些引用通常是通过与位置无关的共享目标文件进行的全局函数调用，或者是通过动态可执行文件进行的外部函数调用。在对提供这些引用的任何动态模块进行编译和链接编辑的过程中，关联的函数调用将成为对过程链接表项的调用。这些项构成了 .plt 节。每个过程链接表项都成为包含关联重定位的延迟引用。

在首次调用过程链接表项的过程中，将控制权传递至运行时链接程序。运行时链接程序将查找所需符号，并重写关联目标文件中的项信息。将来调用此过程链接表项时，将直接转至相应函数。使用此机制，可以延迟此类型的重定位，直到调用函数的第一个实例。此过程有时称为延迟绑定。

运行时链接程序的缺省模式是在每次提供过程链接表重定位时执行延迟绑定。通过将环境变量 LD_BIND_NOW 设置为任意非空值，可以覆盖此缺省模式。此环境变量设置将导致运行时链接程序在装入目标文件时，同时执行即时引用和延迟引用重定位。这些重定位在应用程序获取或重新获取控制权之前执行。例如，根据以下环境变量来处理文件 prog 及其依赖项中的所有重定位。在将控制权转交给应用程序之前处理这些重定位。

```
$ LD_BIND_NOW=1  prog
```

此外，也可使用 dlopen(3C) 来访问目标文件，并将模式定义为 RTLD_NOW。还可使用链接编辑器的 -z now 选项来生成目标文件，以指示该目标文件需要在装入时进行完整的重定位处理。此重定位要求还将在运行时传播至所标记目标文件的所有依赖项。

注 - 前面的即时引用和延迟引用示例都很典型。但是，过程链接表项的创建最终受用作链接编辑输入的可重定位目标文件提供的重定位信息控制。R_SPARC_WPLT30 和 R_386_PLT32 等重定位记录指示链接编辑器创建过程链接表项。这些重定位由与位置无关的代码公用。

但是，通常会通过与位置相关的代码创建动态可执行文件，该代码可能不会指示需要过程链接表项。由于动态可执行文件具有固定位置，因此链接编辑器可在将引用绑定到外部函数定义时创建过程链接表项。无论原始重定位记录如何，都会创建此过程链接表项。

重定位错误

如果找不到符号，则会发生最常见的重定位错误。此情况将会产生相应的运行时链接程序错误消息并终止应用程序。在以下示例中，找不到在文件 libfoo.so.1 中引用的符号 bar。

```
$ ldd prog
        libfoo.so.1 =>   ./libfoo.so.1
        libc.so.1 =>     /lib/libc.so.1
        libbar.so.1 =>   ./libbar.so.1
        libm.so.2 =>     /lib/libm.so.2
$ prog
ld.so.1: prog: fatal: relocation error: file ./libfoo.so.1: \
    symbol bar: referenced symbol not found
```

在对动态可执行文件进行链接编辑的过程中，此类别的任何潜在重定位错误都会标记为致命未定义符号。有关示例，请参见“生成可执行输出文件” [39]。但是，如果运行时找到的依赖项与链接编辑过程中引用的原始依赖项不兼容，则可能会发生运行时重定位错误。在前面的示例中，根据包含 bar 的符号定义的 libbar.so.1 共享目标文件的版本生成了 prog。

在链接编辑过程中使用 -z nodefs 选项，将抑制验证目标文件运行时重定位要求。抑制验证还可能会导致运行时重定位错误。

如果由于找不到用作即时引用的符号而发生重定位错误，则会在进程初始化期间立即出现该错误状态。对于延迟绑定的缺省模式，如果找不到用作延迟引用的符号，则会在应用程序获取控制权后出现该错误状态。后一种情况可能需要几分钟、几个月，也可能从不发生，具体情况视整个代码中使用的执行路径而定。

为防止发生此类错误，可使用 ldd(1) 来验证任何动态可执行文件或共享目标文件的重定位要求。

如果在使用 ldd(1) 时指定 -d 选项，将显示每个依赖项并处理所有即时引用重定位。如果无法解析引用，则会生成诊断消息。在前面的示例中，-d 选项将导致以下错误诊断。

```
$ ldd -d prog
```

```
        libfoo.so.1 =>   ./libfoo.so.1
        libc.so.1 =>     /lib/libc.so.1
        libbar.so.1 =>   ./libbar.so.1
        libm.so.2 =>     /lib/libm.so.2
        symbol not found: bar           (./libfoo.so.1)
```

如果在使用 ldd(1) 时指定 -r 选项，将处理所有即时引用和延迟引用重定位。只要有一种类型的重定位无法被解析，就会生成诊断消息。

装入其他目标文件

通过运行时链接程序，可以使用环境变量 LD_PRELOAD 在进程初始化期间引入新目标文件，从而提供其他级别的灵活性。此环境变量可初始化为共享目标文件或可重定位目标文件名，也可初始化为用空格分隔的文件名字符串。这些目标文件将在装入动态可执行文件之后以及装入任何依赖项之前装入。这些目标文件都被指定了 *world*（全局）搜索作用域和 *global*（全局）符号可见性。

在以下示例中，首先装入动态可执行文件 prog，然后装入共享目标文件 newstuff.so.1。接下来装入在 prog 中定义的依赖项。

```
$ LD_PRELOAD=./newstuff.so.1  prog
```

可以使用 ldd(1) 显示这些目标文件的处理顺序。

```
$ ldd -e LD_PRELOAD=./newstuff.so.1 prog
        ./newstuff.so.1 => ./newstuff.so
        libc.so.1 =>     /lib/libc.so.1
```

在以下示例中，预装入比较复杂且耗时。

```
$ LD_PRELOAD="./foo.o ./bar.o"  prog
```

运行时链接程序首先会对可重定位目标文件 foo.o 和 bar.o 进行链接编辑，以生成在内存中保存的共享目标文件。然后，会按照前面示例中预装入共享目标文件 newstuff.so.1 的方式，在动态可执行文件及其依赖项之间插入此内存映像。同样，可以使用 ldd(1) 显示这些目标文件的处理顺序。

```
$ ldd -e LD_PRELOAD="./foo.o ./bar.o" ldd prog
        ./foo.o =>       ./foo.o
        ./bar.o =>       ./bar.o
        libc.so.1 =>     /lib/libc.so.1
```

在动态可执行文件后插入目标文件的这些机制将插入概念引入到另一个层次。您可以使用这些机制来试验驻留在标准共享目标文件中的函数的新实现。如果预装入包含此函数的目标文件，则该目标文件将插入到原始功能中。因此，原始功能会完全隐藏在新的预装入版本之后。

预装入的另一个用途是扩充驻留在标准共享目标文件中的函数。通过在原始符号中插入新符号，新函数可以执行其他处理。新函数还可调用原始函数。此机制通常会将 dlsym(3C) 与特殊句柄 RTLD_NEXT 配合使用，以获取原始符号的地址。

延迟装入动态依赖项

在内存中装入动态目标文件时，将会检查该目标文件的任何其他依赖项。缺省情况下，将立即装入存在的所有依赖项。此循环会一直继续，直到找遍整个依赖项树为止。最后，解析由重定位指定的所有目标文件间数据引用。无论应用程序在执行期间是否引用了这些依赖项中的代码，都会执行这些操作。

在延迟装入模型中，标记为延迟装入的所有依赖项仅在显式引用时装入。通过利用函数调用的延迟绑定，依赖项装入将会一直延迟，直到第一次引用该函数。因此，绝不会装入从不引用的目标文件。

重定位引用可以是即时的，也可以是延迟的。因为初始化目标文件时必须解析即时引用，所以必须即时装入满足此引用要求的任何依赖项。因此，将这类依赖项标识为延迟可装入几乎没有效果。请参见“执行重定位的时间” [94]。通常，建议不要在动态目标文件之间进行即时引用。

链接编辑器对调试库 liblddbg 的引用将使用延迟装入。由于不会经常进行调试，因此每次调用链接编辑器时装入此库既无必要又需要很大开销。通过指示可以延迟装入此库，处理库的开销将转至要求调试输出的那些调用。

实现延迟装入模型的替代方法是在需要时使用 dlopen() 和 dlsym() 来装入并绑定到依赖项。如果 dlsym() 引用数很少，则此模型非常理想。如果在链接编辑时不知道依赖项名称或位置，则此模型也很适用。对于与已知依赖项的更复杂交互，对正常符号引用进行编码并指定要延迟装入的依赖项更为简单。

通过链接编辑器选项 -z lazyload 和 -z nolazyload，可将目标文件分别指定为延迟装入或正常装入。这些选项在链接编辑命令行中与位置相关。该选项之后的任何依赖项都将采用其指定的装入属性。缺省情况下，-z nolazyload 选项有效。

以下简单程序具有依赖项 libdebug.so.1。动态节 (.dynamic) 显示 libdebug.so.1 被标记为延迟装入。符号信息节 (.SUNW_syminfo) 显示触发 libdebug.so.1 装入的符号引用。

```
$ cc -o prog prog.c -L. -zlazyload -ldebug -znolazyload -lelf -R'$ORIGIN'
$ elfdump -d prog

Dynamic Section:  .dynamic
   index  tag          value
     [0]  POSFLAG_1       0x1         [ LAZY ]
     [1]  NEEDED        0x123         libdebug.so.1
     [2]  NEEDED        0x131         libelf.so.1
     [3]  NEEDED        0x13d         libc.so.1
     [4]  RUNPATH       0x147         $ORIGIN
     ...
```

```
$ elfdump -y prog

Syminfo section: .SUNW_syminfo
   index  flgs        bound to        symbol
    ....
    [52]  DL      [1] libdebug.so.1   debug
```

值为 LAZY 的 POSFLAG_1 指示应该延迟装入下面的 NEEDED 项 libdebug.so.1。由于 libelf.so.1 没有前述的 LAZY 标志，因此该库将在程序初始启动时装入。

注 - libc.so.1 具有特殊的系统要求，该要求指出不应延迟装入该文件。如果在处理 libc.so.1 时 -z lazyload 有效，则实际上会忽略该标志。

使用延迟装入时，可能需要准确声明在应用程序使用的目标文件中的依赖项和运行路径。例如，假定有两个目标文件 libA.so 和 libB.so，它们同时引用了 libX.so 中的符号。libA.so 将 libX.so 声明为依赖项，但 libB.so 未执行此操作。通常，将 libA.so 与 libB.so 一起使用时，libB.so 可以引用 libX.so，因为 libA.so 使此依赖项可用。但是，如果 libA.so 将 libX.so 声明为延迟装入，则在 libB.so 引用此依赖项时可能不会装入 libX.so。如果 libB.so 将 libX.so 声明为依赖项，但未能提供查找该依赖项所需的运行路径，则可能会出现类似错误。

无论是否延迟装入，动态目标文件都应声明其所有依赖项以及如何查找这些依赖项。对于延迟装入，此依赖项信息更为重要。

注 - 通过将环境变量 LD_NOLAZYLOAD 设置为非空值，可在运行时禁用延迟装入。

提供 dlopen（） 的替代项

延迟装入可以提供 dlopen(3C) 和 dlsym(3C) 的替代方法。请参见“运行时链接编程接口” [105]。例如，libfoo.so.1 中的以下代码将验证是否装入了目标文件，然后调用该目标文件提供的接口。

```
void foo()
{
        void *handle;

        if ((handle = dlopen("libbar.so.1", RTLD_LAZY)) != NULL) {
                int (*fptr)();

                if ((fptr = (int (*)())dlsym(handle, "bar1")) != NULL)
                        (*fptr)(arg1);
                if ((fptr = (int (*)())dlsym(handle, "bar2")) != NULL)
                        (*fptr)(arg2);
                ....
        }
}
```

尽管非常灵活，但这种使用 dlopen（） 和 dlsym（） 的模型不是自然的编码样式，并存在一些缺点。

- 必须要知道其中的符号将要退出的目标文件。
- 使用函数指针的调用不会通过编译器或 lint(1) 提供任何验证方式。

如果提供所需接口的目标文件满足下列条件，则可以简化此代码。

- 可在链接编辑时作为依赖项建立该目标文件。
- 该目标文件始终可用。

通过使用函数引用来触发延迟装入，可以实现同样的 libbar.so.1 延迟装入。在此情况下，对函数 bar1（） 的引用将导致延迟装入关联的依赖项。这种编码要自然得多，标准函数调用可用于编译器或 lint(1) 验证。

```
void foo()
{
        bar1(arg1);
        bar2(arg2);
        ....
}
$ cc -G -o libfoo.so.1 foo.c -L. -zdefs -zlazyload -lbar -R'$ORIGIN'
```

但是，如果提供所需接口的目标文件并非始终可用，则此模型会失败。在此情况下，最好能够在不必知道依赖项名称的情况下测试依赖项是否存在。需要一种测试满足函数引用要求的依赖项可用性的方法。

一个测试函数是否存在的强大模型，可以通过显式定义延迟依赖项，以及使用 dlsym(3C) 与 RTLD_PROBE 句柄来实现。

显式定义的延迟依赖项是对延迟可装入依赖项的扩展。与延迟依赖项关联的符号引用则称为延迟符号。仅在首次引用符号时，才会处理根据该符号进行的重定位。这些重定位不会在 LD_BIND_NOW 处理过程中进行处理，也不会通过带有 RTLD_NOW 标志的 dlsym(3C) 进行处理。

延迟依赖项是在链接编辑时使用链接编辑器 -z deferred 选项建立的。

```
$ cc -G -o libfoo.so.1 foo.c -L. -zdefs -zdeferred -lbar -R'$ORIGIN'
```

使用已建立的 libbar.so.1 作为延迟依赖项时，对 bar1（） 的引用可以验证依赖项是否可用。此测试可用于控制对依赖项以使用 dlsym(3C) 的方式提供的函数的引用。然后此代码可以对 bar1（） 和 bar2（） 做出自然调用。这些调用更加清晰且更易于编写，并允许编译器捕捉调用序列中的错误。

```
void foo()
{
        if (dlsym(RTLD_PROBE, "bar1")) {
                bar1(arg1);
                bar2(arg2);
                ....
        }
```

```
}
```

延迟依赖项可提供其他级别的灵活性。如果尚未装入该依赖项，则可以在运行时对其进行更改。该机制提供的灵活性级别类似于 dlopen(3C)，其中各个目标文件可以由调用者装入和绑定。

如果已知原始依赖项名称，则可以使用带有 RTLD_DI_DEFERRED 参数的 dlinfo(3C) 将原始依赖项更换为新的依赖项。或者，可以使用与依赖项关联的延迟符号来识别使用带有 RTLD_DI_DEFERRED_SYM 参数的 dlinfo(3C) 的延迟依赖项。

初始化和终止例程

动态目标文件可以提供用于运行时初始化和终止处理的代码。每次在进程中装入动态目标文件时，都会执行一次动态目标文件的初始化代码。每次在进程中卸载动态目标文件或进程终止时，都会执行一次动态目标文件的终止代码。

在将控制权转交给应用程序之前，运行时链接程序将处理应用程序中找到的所有初始化节及所有装入的依赖项。如果在进程执行期间装入新动态目标文件，则会在装入该目标文件的过程中处理其初始化节。初始化节 .preinit_array、.init_array 和 .init 由链接编辑器在生成动态目标文件时创建。

运行时链接程序执行的函数的地址包含在 .preinit_array 和 .init_array 节中。这些函数的执行顺序与其地址在数组中的显示顺序相同。运行时链接程序将 .init 节作为单独的函数执行。如果某目标文件同时包含 .init 节和 .init_array 节，则会首先处理 .init 节，然后再处理该目标文件的 .init_array 节定义的函数。

动态可执行文件可在 .preinit_array 节中提供预初始化函数。这些函数将在运行时链接程序生成进程映像并执行重定位之后但执行任何其他初始化函数之前执行。预初始化函数不允许在共享目标文件中执行。

注 - 编译器驱动程序提供的进程启动机制通过应用程序来调用动态可执行文件中的任何 .init 节。执行所有依赖项初始化节之后，最后会调用动态可执行文件中的 .init 节。

动态目标文件还可提供终止节。终止节 .fini_array 和 .fini 由链接编辑器在生成动态目标文件时创建。

所有终止节都传递给 atexit(3C)。当进程调用 exit(2) 时，将调用这些终止例程。使用 dlclose(3C) 从运行的进程中删除目标文件时，也会调用终止节。

运行时链接程序执行的函数的地址包含在 .fini_array 节中。这些函数的执行顺序与其地址在数组中的显示顺序相反。运行时链接程序将 .fini 节作为单独的函数执行。如果某目标文件同时包含 .fini 节和 .fini_array 节，则会首先处理 .fini_array 节定义的函数，然后再处理该目标文件的 .fini 节。

注 - 编译器驱动程序提供的进程终止机制通过应用程序来调用动态可执行文件中的任何 `.fini` 节。在执行所有依赖项终止节之前，首先会调用动态可执行文件的 `.fini` 节。

有关链接编辑器创建初始化节和终止节的更多信息，请参见“初始化节和终止节” [33]。

初始化和终止代码的限制和缺点

ELF 初始化节和例程以及终止节和例程在目标文件生命周期的某个敏感时间执行。初始化过程中，目标文件已装入到内存中，但未完全初始化。终结化过程中，目标文件仍装入内存中，但不再能够安全使用，并且部分可能会从进程状态下删除。在任一上下文中，进程状态都不是完全一致的，而且在可以安全执行的代码方面有很大的限制。常见的缺点包括但不限于以下各项：

- 循环依赖项导致死锁，其中，一个目标文件的初始化代码触发另一个目标文件的装入操作，反过来又回调到初始目标文件。
- 当某共享目标文件用于多线程应用程序时，线程序列化将失败。两个线程会同时尝试访问延迟装入的库。最先成功访问的线程将导致运行时链接程序装入目标文件并开始运行初始化代码。程序员经常会误以为在 ELF 初始化代码和终止代码运行时，运行时链接程序可以防止一个以上的线程访问给定目标文件，但事实上并非如此。当初始化代码正在运行时，运行时链接程序并不能防止其他线程尝试访问库。因此可能会有第二个线程在不一致的状态下访问目标文件。目标文件的责任是提供必要的锁定或要求调用方提供必要锁定来对此类访问进行序列化。

ELF 初始化节和例程以及终止节和例程允许执行任意代码，这给人以一种错觉，即它们能够执行在正常上下文中代码可以执行的任何操作。在这种观念下，此类代码看起来不外是一种无需显式函数调用便可执行初始化或清除的便利方法。这种误解会导致难以诊断的故障。

程序员在使用 ELF 初始化代码和终止代码时应格外注意，并限制操作的范围和复杂性。链接编辑器和运行时链接程序并不了解此类代码的内容或目的，无法诊断或阻止不安全的代码。自包含的小操作是安全的。要访问其他目标文件或进程状态的操作则可能不安全。库应提供显式初始化函数和终止函数供调用方运行，并记录执行此类操作的要求，而不是尝试在初始化代码和终止代码中执行复杂的操作。

以下各节详细说明了这些问题。

初始化和终止顺序

确定运行时进程中初始化和终止代码的执行顺序是一个很复杂的过程，需要进行依赖项分析。此过程在原来初始化节和终止节的基础上有了很大发展。此过程试图达到现代语言和当前编程技术的预期目标。但是，可能存在很难满足用户期望的情况。通过了解这些情况以及限制初始化代码和终止代码的内容，可以实现灵活的可预测运行时行为。

初始化节的目标是在引用同一目标文件中的任何其他代码之前执行一小节代码。终止节的目标是在目标文件完成执行后执行一小节代码。自包含的初始化节和终止节可以轻松满足这些要求。

但是，初始化节通常更为复杂，它会引用其他目标文件提供的外部接口。因此，将会建立依赖项，并且在从其他目标文件进行引用之前，必须在该依赖项中执行某个目标文件的初始化节。应用程序可建立详细的依赖项分层结构。此外，依赖项还可在其分层结构中创建循环。如果初始化节装入其他目标文件或更改已装入目标文件的重定位模式，会使得情况更加复杂。这些问题已经导致产生了各种排序和执行方法，以尝试达到这些节的原始目标。

运行时链接程序会构造以拓扑方式排序的已装入目标文件列表。此列表是根据每个目标文件表示的依赖项关系以及所表示依赖项外部的符号绑定生成的。

初始化节是按依赖项的相反拓扑顺序执行的。如果发现循环依赖项，则无法对构成循环的目标文件进行拓扑排序。任何循环依赖项的初始化节都会以其相反装入顺序执行。同样，会以依赖项的拓扑顺序调用终止节。任何循环依赖项的终止节以其装入顺序执行。

可通过使用带有 -i 选项的 ldd(1)，就目标文件的依赖项的初始化顺序进行静态分析。例如，以下动态可执行文件及其依赖项显示了循环依赖项：

```
$ elfdump -d B.so.1 | grep NEEDED
     [1]     NEEDED      0xa9    C.so.1
$ elfdump -d C.so.1 | grep NEEDED
     [1]     NEEDED      0xc4    B.so.1
$ elfdump -d main | grep NEEDED
     [1]     NEEDED      0xd6    A.so.1
     [2]     NEEDED      0xc8    B.so.1
     [3]     NEEDED      0xe4    libc.so.1
$ ldd -i main
        A.so.1 =>        ./A.so.1
        B.so.1 =>        ./B.so.1
        libc.so.1 =>     /lib/libc.so.1
        C.so.1 =>        ./C.so.1
        libm.so.2 =>     /lib/libm.so.2

    cyclic dependencies detected, group[1]:
        ./libC.so.1
        ./libB.so.1

    init object=/lib/libc.so.1
    init object=./A.so.1
    init object=./C.so.1 - cyclic group [1], referenced by:
        ./B.so.1
    init object=./B.so.1 - cyclic group [1], referenced by:
        ./C.so.1
```

上述分析完全从显式依赖项关系的拓扑排序得出。但是，频繁创建了未定义所需依赖项的目标文件。为此，进行依赖项分析时还会引入符号绑定。将符号绑定与显式依赖项结合有助于生成更准确的依赖性关系。通过使用带有 -i 和 -d 选项的 ldd(1)，可以获取更准确的初始化顺序静态分析。

装入目标文件的最常见模型使用延迟绑定。对于此模型，将仅在初始化处理之前处理即时引用符号绑定。延迟引用中的符号绑定可能仍会处于暂挂状态。这些绑定可以扩展到现在为止已建立的依赖项关系。通过使用带有 -i 和 -r 选项的 ldd(1)，可以对引入所有符号绑定的初始化顺序进行静态分析。

实际上，大多数应用程序都使用延迟绑定。因此，计算初始化顺序之前完成的依赖项分析会遵循使用 ldd -id 的静态分析。但是，由于此依赖项分析可能不完整，并且可能存在循环依赖项，因此运行时链接程序允许进行动态初始化。

动态初始化会尝试在调用同一目标文件中的任何函数之前执行该目标文件的初始化节。在延迟符号绑定过程中，运行时链接程序将确定是否已调用要绑定到的目标文件的初始化节。如果未调用，则运行时链接程序将在从符号绑定过程返回之前执行初始化节。

不能使用 ldd(1) 来显示动态初始化。但是，通过将 LD_DEBUG 环境变量设置为包括标记 init，可在运行时观察初始化调用的确切顺序。请参见“调试功能” [116]。通过添加调试标记 detail，可以捕获丰富的运行时初始化信息和终止信息。此信息包括依赖项列表、拓扑处理以及循环依赖项的标识。

仅当处理延迟引用时，动态初始化才可用。以下各项禁用此动态初始化。

- 使用环境变量 LD_BIND_NOW。
- 使用 -z now 选项生成的目标文件。
- 由 dlopen(3C) 在模式 RTLD_NOW 下装入的目标文件。

到现在为止已介绍的初始化方法可能仍然不足以处理某些动态活动。初始化节可显式使用 dlopen(3C) 或隐式使用延迟装入和过滤器来装入其他目标文件。初始化节还可提升现有目标文件的重定位。对于已装入来应用延迟绑定的目标文件，均解析这些绑定（如果使用 dlopen(3C) 在模式 RTLD_NOW 下引用同一目标文件）。此重定位提升将有效地抑制在动态解析函数调用时可用的动态初始化功能。

每次装入新目标文件或者提升现有目标文件的重定位时，都会启动这些目标文件的拓扑排序。实际上，在建立新的初始化要求并执行关联的初始化节时，将暂停原始初始化执行。此模型尝试确保新引用的目标文件进行适当的初始化，以供原始初始化节使用。但是，此并行操作可能会导致不需要的递归。

处理使用延迟绑定的目标文件时，运行时链接程序可以检测某些级别的递归。可通过设置 LD_DEBUG=init 来显示此递归。例如，执行 foo.so.1 的初始化节可能会导致调用另一目标文件。如果此目标文件随后引用了 foo.so.1 中的接口，则会创建循环。运行时链接程序可在绑定对 foo.so.1 的延迟函数引用时检测此递归。

```
$ LD_DEBUG=init  prog
00905: ....
00905: warning: calling foo.so.1 whose init has not completed
00905: ....
```

运行时链接程序无法检测通过已重定位的引用导致的递归。

递归开销可能很大并且存在问题。因此，应减少可通过初始化节触发的外部引用数和动态装入活动数，以便消除递归。

对于使用 dlopen(3C) 添加到运行的进程的所有目标文件，可以重复执行初始化处理。另外，对于因为调用 dlclose(3C) 而从进程卸载的所有目标文件，也可执行终止处理。

以上各节按尝试满足用户期望的方式，介绍了用于执行初始化节和终止节的各种方法。但是，还应采用编码样式和链接编辑做法来简化依赖项之间的初始化和终止关系。该简化有助于进行可预测的初始化处理和终止处理，同时不会轻易出现意外依赖项排序带来的负面影响。

应尽量精简初始化节和终止节的内容。通过运行时初始化目标文件来避免创建全局构造函数。减少初始化和终止代码对其他依赖项的依赖。定义所有动态目标文件的依赖项要求。请参见“生成共享目标文件输出文件” [40]。不要表示不需要的依赖项。请参见“共享目标文件处理” [28]。避免循环依赖项。不要依赖于初始化或终止序列的顺序。目标文件的排序可能会受到共享目标文件和应用程序开发的影响。请参见“依赖项排序” [127]。

安全性

在安全进程中，对其依赖项及运行路径的评估应用了一些限制，以避免产生恶意依赖项替换或符号插入。

对于某个进程来说，如果 issetugid(2) 系统调用返回的结果为 True，运行时链接程序会将该进程归类为安全进程。

对于 32 位目标文件，运行时链接程序已知的缺省可信目录为 /lib/secure 和 /usr/lib/secure。对于 64 位目标文件，运行时链接程序已知的缺省可信目录为 /lib/secure/64 和 /usr/lib/secure/64。实用程序 crle(1) 可用于指定适用于安全应用程序的其他可信目录。使用此技术的管理员应确保已对目标目录进行了适当的保护，以防受到恶意入侵。

如果 LD_LIBRARY_PATH 系列环境变量对安全进程有效，则仅将此变量指定的可信目录用于扩充运行时链接程序的搜索规则。请参见“运行时链接程序搜索的目录” [88]。

在安全进程中，将使用应用程序或其任何依赖项指定的运行路径。但是，运行路径必须是完整路径名，即路径名必须以 "/" 开头。

在安全进程中，仅当 $ORIGIN 字符串扩展为可信目录时，才允许对其进行扩展。请参见“安全性” [234]。但是，如果 $ORIGIN 扩展与一个已提供依赖项的目录匹配，则该目录是隐式安全的。该目录可用于提供其他依赖项。

在安全进程中，LD_CONFIG 会被忽略。但是，会使用安全应用程序中记录的配置文件。请参见 ld(1) 的 -c 选项。记录的配置文件必须使用全路径名，即路径名必须以“/”开头。

使用 $ORIGIN 字符串的已记录配置文件仅限于已知的可信目录。在安全应用程序中记录配置文件的开发者应确保配置文件目录受到适当的保护，以避免恶意侵入。如果缺少记录的配置文件，则安全进程将使用缺省配置文件（如果该配置文件存在）。请参见 crle(1)。

在安全进程中，LD_SIGNAL 会被忽略。

在安全进程中使用 LD_PRELOAD 或 LD_AUDIT 环境变量可装入其他目标文件。必须将这些目标文件指定为完整路径名或简单文件名。完整路径名仅限于已知的可信目录。不含 "/" 的简单文件名在定位时将遵循前述搜索路径限制。简单文件名只能解析为已知的可信目录。

在安全进程中，将使用前面描述的路径名限制来处理组成简单文件名的所有依赖项。以完整路径名或相对路径名表示的依赖项按原样使用。因此，安全进程的开发者应确保作为这些依赖项之一引用的目标目录受到适当的保护，以避免恶意侵入。

在创建安全进程时，不要使用相对路径名来表示依赖项或构造 dlopen(3C) 路径名。此限制适用于应用程序及所有依赖项。

运行时链接编程接口

在应用程序的链接编辑期间指定的依赖项，由运行时链接程序在进程初始化过程中处理。除了此机制以外，应用程序还可在执行期间通过绑定到其他目标文件来扩展其地址空间。应用程序将有效地使用处理应用程序标准依赖项所用的相同运行时链接程序服务。

延迟目标文件绑定有以下几个优点。

- 通过在需要目标文件时而不是在应用程序初始化期间处理目标文件，可以极大地缩短启动时间。如果在应用程序的特定运行期间不需要某目标文件提供的服务，则表示不需要该目标文件。用于提供帮助或调试信息的目标文件可能会出现此情况。
- 根据所需的确切服务（如用于网络协议），应用程序可在若干不同目标文件间进行选择。
- 在执行期间添加到进程地址空间的所有目标文件都可在使用后释放。

应用程序可使用下列典型方案来访问其他共享目标文件。

- 使用 dlopen(3C) 来查找共享目标文件并将其添加到运行的应用程序的地址空间。同时查找并添加此共享目标文件的所有依赖项。
- 重定位添加的共享目标文件及其依赖项。调用这些目标文件中的所有初始化节。
- 应用程序使用 dlsym(3C) 来查找已添加目标文件中的符号。然后，应用程序可引用该数据或调用这些新符号定义的函数。

- 在应用程序处理完这些目标文件后，可使用 dlclose(3C) 来释放地址空间。此时将调用要释放的目标文件中的所有终止节。
- 可使用 dlerror(3C) 来显示由于使用运行时链接程序接口例程而导致的所有错误状态。

运行时链接程序的服务将在头文件 dlfcn.h 中进行定义，并且通过共享目标文件 libc.so.1 使该服务可用于应用程序。在以下示例中，文件 main.c 可以引用任何 dlopen(3C) 系列例程，并且应用程序 prog 可在运行时绑定到这些例程。

```
$ cc -o prog main.c
```

注 - 在以前的 Oracle Solaris OS 发行版中，共享目标文件 libdl.so.1 提供了动态链接接口。libdl.so.1 仍然可用于支持所有现有依赖项。但是，libdl.so.1 提供的动态链接接口现在可从 libc.so.1 获取。不再需要使用 -ldl 进行链接。

装入其他目标文件

使用 dlopen(3C)，可将其他目标文件添加到运行的进程的地址空间。此函数将路径名和绑定模式作为参数，并向应用程序返回一个句柄。通过 dlsym(3C)，可使用此句柄来查找供应用程序使用的符号。

如果将路径名指定为简单文件名（名称中没有 "/"），则运行时链接程序将使用一组规则来生成相应的路径名。包含 "/" 的路径名均按原样使用。

这些搜索路径规则与用于查找所有初始依赖项的规则完全相同。请参见“运行时链接程序搜索的目录” [88]。例如，文件 main.c 包含以下代码片段。

```
#include    <stdio.h>
#include    <dlfcn.h>

int main(int argc, char **argv)
{
        void *handle;
        ....

        if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
                (void) printf("dlopen: %s\n", dlerror());
                return (1);
        }
        ....
```

要查找共享目标文件 foo.so.1，运行时链接程序应使用进程初始化时存在的 LD_LIBRARY_PATH 定义。接下来，使用链接编辑 prog 过程中指定的运行路径。最后，使用缺省位置 /lib 和 /usr/lib（对于 32 位目标文件）或 /lib/64 和 /usr/lib/64（对于 64 位目标文件）。

如果将路径名指定为：

```
if ((handle = dlopen("./foo.so.1", RTLD_LAZY)) == NULL) {
```

则运行时链接程序只会在进程的当前工作目录中搜索该文件。

注 - 应该通过版本化文件名来引用使用 dlopen(3C) 指定的任何共享目标文件。有关版本化的更多信息，请参见“协调版本化文件名” [224]。

如果无法找到需要的目标文件，dlopen(3C) 将返回 NULL 句柄。在此情况下，可使用 dlerror(3C) 来显示失败的真正原因。例如：

```
$ cc -o prog main.c
$ prog
dlopen: ld.so.1: prog: fatal: foo.so.1: open failed: No such file or directory
```

如果 dlopen(3C) 添加的目标文件依赖于其他目标文件，则也会将这些目标文件引入进程的地址空间中。此进程将一直继续，直到装入指定目标文件的所有依赖项。此依赖项树称为组。

如果 dlopen(3C) 指定的目标文件或其任何依赖项已经是进程映像的一部分，则不会进一步处理这些目标文件，而是向应用程序返回一个有效句柄。此机制可避免多次装入同一目标文件，并且使应用程序可以获取指向自身的句柄。例如，前面的示例 main.c 可以包含以下 dlopen（） 调用。

```
if ((handle = dlopen(0, RTLD_LAZY)) == NULL) {
```

通过指定了 RTLD_GLOBAL 标志的 dlopen(3C)，可使用从该 dlopen(3C) 返回的句柄在应用程序本身、进程初始化时装入的任何依赖项或添加到进程地址空间的所有目标文件中查找符号。

重定位处理

在找到并装入所有目标文件后，运行时链接程序必须处理每个目标文件并执行所有必需的重定位。另外，还必须以同一方式重定位使用 dlopen(3C) 引入进程地址空间中的所有目标文件。

对于简单应用程序，此过程很简单。但是，对于使用较复杂的应用程序（包含涉及多个目标文件的 dlopen(3C) 调用，且可能包含公共依赖项）的用户，则此过程可能相当重要。

可以根据重定位发生的时间对其进行分类。运行时链接程序的缺省行为是在初始化时处理所有即时引用重定位，以及在进程执行期间处理所有延迟引用（此机制通常称为延迟绑定）。

当模式定义为 RTLD_LAZY 时，此相同的机制可应用于任何使用 dlopen(3C) 添加的目标文件。替代方法是要求在添加目标文件时立即执行目标文件的所有重定位。您可以使用 RTLD_NOW 模式，也可以使用链接编辑器的 -z now 选项在生成目标文件时将此要求记录在目标文件中。此重定位要求将传播至要打开的目标文件的所有依赖项。

重定位还可分为非符号重定位和符号重定位。本节的余下部分将讨论有关符号重定位的问题（无论这些重定位何时进行），并重点介绍符号查找的某些细节信息。

符号查找

如果 dlopen(3C) 获取的目标文件引用全局符号，则运行时链接程序必须从构成进程的目标文件池中查找此符号。如果缺少直接绑定，则会将缺省符号搜索模型应用于通过 dlopen（） 获取的目标文件。但是，dlopen（） 模式以及构成进程的目标文件的属性允许使用替代的符号搜索模型。

需要直接绑定的目标文件将直接在关联的依赖项中搜索符号（尽管要维护后面介绍的所有属性）。请参见第 6 章 直接绑定。

注 - 指定了 STV_SINGLETON 可见性的符号使用缺省符号搜索绑定，无论 dlopen(3C) 属性如何都是如此。请参见表 12-23 “ELF 符号可见性”。

缺省情况下，使用 dlopen(3C) 获取的目标文件将被指定全局符号搜索作用域和局部符号可见性。“缺省符号查找模型” [108]一节将使用此缺省模型说明典型的目标文件组交互。“定义全局目标文件” [111]、“隔离组” [112]和“目标文件分层结构” [112]部分介绍使用 dlopen(3C) 模式和文件属性扩展缺省符号查找模型的示例。

缺省符号查找模型

对于通过基本 dlopen(3C) 添加的每个目标文件，运行时链接程序将首先在动态可执行文件中查找符号。之后，运行时链接程序在进程初始化期间提供的每个目标文件中查找。如果找不到该符号，运行时链接程序将继续搜索。接下来，运行时链接程序会在通过 dlopen(3C) 获取的目标文件及其依赖项中查找。

使用缺省符号查找模型时，可以转换到延迟装入环境。如果在当前装入的目标文件中找不到某符号，则会处理所有暂挂的延迟装入目标文件，以尝试查找该符号。此装入是对尚未完整定义其依赖项的目标文件的补偿。但是，该补偿可能会破坏延迟装入的优点。

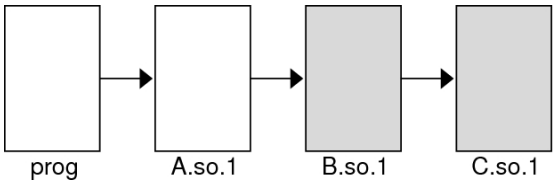
在以下示例中，动态可执行文件 prog 和共享目标文件 B.so.1 具有下列依赖项。

```
$ ldd prog
        A.so.1 =>        ./A.so.1
$ ldd B.so.1
```

```
        C.so.1 =>        ./C.so.1
```

如果 prog 通过 dlopen(3C) 获取共享目标文件 B.so.1，则会首先在 prog 中查找重定位共享目标文件 B.so.1 和 C.so.1 所需的任何符号，然后依次在 A.so.1、B.so.1 和 C.so.1 中进行查找。在此简单示例中，将通过 dlopen(3C) 获取的共享目标文件视作已在对应用程序的原始链接编辑结束时将它们添加进来。例如，可以采用图解方式表示前面列表中引用的目标文件，如下图所示。

图 3-1 单个 dlopen（） 请求



通过 dlopen(3C) 获取的目标文件（显示为阴影块）所需的任何符号查找，将从动态可执行文件 prog 继续执行一直到最后一个共享目标文件 C.so.1。

此符号查找是按装入目标文件时为目标文件指定的属性建立的。请记住，已为动态可执行文件及随其装入的所有依赖项指定了全局符号可见性，并且为新目标文件指定了全局符号搜索作用域。因此，新目标文件能够在原始目标文件中查找符号。新目标文件也会构成一个唯一的组，其中每个目标文件都具有局部符号可见性。因此，该组中的每个目标文件都可在其他组成员中查找符号。

这些新目标文件不会影响应用程序或其初始依赖项所需的正常符号查找。例如，如果 A.so.1 要求在执行了前面的 dlopen(3C) 后进行函数重定位，则运行时链接程序对重定位符号的正常搜索是首先在 prog 中查找，然后在 A.so.1 中查找。运行时链接程序不会继续在 B.so.1 或 C.so.1 中查找。

此符号查找同样是按装入目标文件时为目标文件指定的属性建立的。对于动态可执行文件及随其装入的所有依赖项，都指定全局符号搜索作用域。此作用域不允许它们在仅提供局部符号可见性的新目标文件中查找符号。

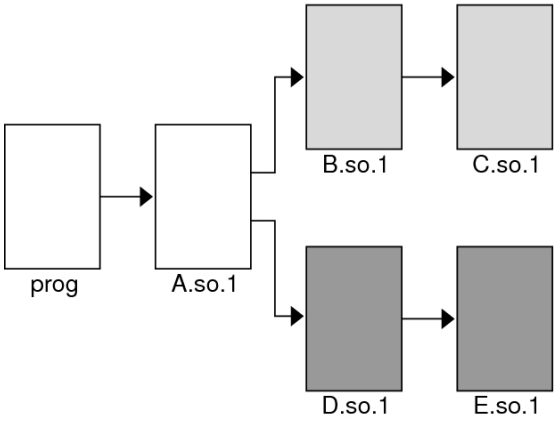
这些符号搜索和符号可见性属性用于维护目标文件之间的关联。这些关联基于它们引入进程地址空间的情况以及目标文件之间的依赖项关系。将与给定 dlopen(3C) 关联的目标文件指定给唯一的组，可以确保仅允许与同一 dlopen(3C) 关联的目标文件在目标文件本身及其关联的依赖项中查找符号。

定义目标文件之间的关联的概念在多次执行 dlopen(3C) 的应用程序中更为清晰。例如，假定共享目标文件 D.so.1 具有以下依赖项：

```
$ ldd D.so.1
        E.so.1 =>        ./E.so.1
```

并且 prog 应用程序使用 dlopen(3C) 装入此共享目标文件及共享目标文件 B.so.1。下图说明了目标文件之间的符号查找关系。

图 3-2 多个 dlopen（） 请求



假定 B.so.1 和 D.so.1 都包含符号 foo 的定义，C.so.1 和 E.so.1 都包含需要此符号的重定位。由于目标文件与唯一的组关联，因此 C.so.1 绑定到 B.so.1 中的定义，而 E.so.1 绑定到 D.so.1 中的定义。此机制用于为通过多次调用 dlopen(3C) 获取的目标文件提供最直观的绑定。

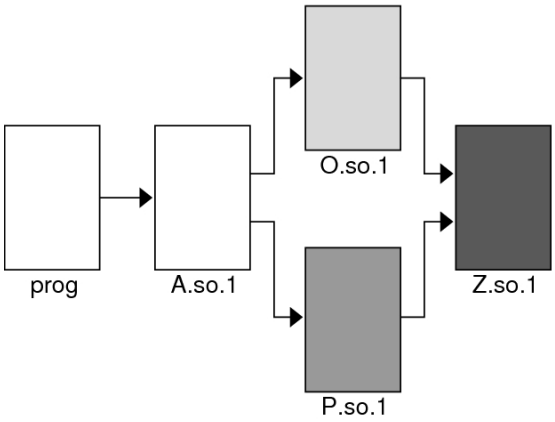
在到现在为止已介绍的情况中使用目标文件时，执行每个 dlopen(3C) 的顺序对生成的符号绑定没有影响。但是，如果目标文件具有公共依赖项，则生成的绑定可能会受到进行 dlopen(3C) 调用的顺序的影响。

在以下示例中，共享目标文件 O.so.1 和 P.so.1 具有相同的公共依赖项。

```
$ ldd O.so.1
        Z.so.1 =>        ./Z.so.1
$ ldd P.so.1
        Z.so.1 =>        ./Z.so.1
```

在此示例中，prog 应用程序将对其中每个共享目标文件执行 dlopen(3C)。由于共享目标文件 Z.so.1 是 O.so.1 和 P.so.1 的公共依赖项，因此将为与两次 dlopen(3C) 调用关联的两个组指定 Z.so.1。此关系如下图所示。

图 3-3 具有公共依赖项的多个 dlopen（） 请求



Z.so.1 可同时供 O.so.1 和 P.so.1 用于查找符号。更重要的是，就 dlopen(3C) 排序而言，Z.so.1 还可用于同时在 O.so.1 和 P.so.1 中查找符号。

因此，如果 O.so.1 和 P.so.1 同时包含符号 foo 的定义（这是 Z.so.1 重定位所需的），则进行的实际绑定不可预测，因为它会受到 dlopen(3C) 调用的顺序的影响。如果符号 foo 的功能在定义它的两个共享目标文件间不同，则在 Z.so.1 中执行代码的整体结果可能会因应用程序的 dlopen(3C) 排序而异。

定义全局目标文件

通过使用 RTLD_GLOBAL 标志扩充模式参数，可将为通过 dlopen(3C) 获取的目标文件缺省指定的局部符号可见性提升为全局可见性。在此模式下，具有全局符号搜索作用域的任何其他目标文件可使用通过 dlopen(3C) 获取的所有目标文件来查找符号。

此外，通过 dlopen(3C) 获取的、并且带有 RTLD_GLOBAL 标志的任何目标文件都可供使用 dlopen() 及值为 0 的路径名的符号查找使用。

注 - 如果某个组成员定义了局部符号可见性，并且被另一个定义了全局符号可见性的组引用，则该目标文件的可见性将变为局部和全局可见性的串联。即使以后删除该全局组引用，也会保留此属性提升。

隔离组

通过使用 RTLD_GROUP 标志扩充模式参数，可将为通过 dlopen(3C) 获取的目标文件缺省指定的全局符号搜索作用域缩小为组。在此模式下，仅允许通过 dlopen(3C) 获取的所有目标文件在其各自的组中查找符号。

使用链接编辑器的 -B group 选项，可在生成目标文件时为其指定组符号搜索作用域。

注 - 如果某个组成员定义了组搜索要求，并且被另一个定义了全局搜索要求的组引用，则该目标文件的搜索要求将变为组和全局搜索的串联。即使以后删除该全局组引用，也会保留此属性提升。

目标文件分层结构

如果初始目标文件是从 dlopen(3C) 获取的，并且使用 dlopen () 打开第二个目标文件，则这两个目标文件都会被指定给一个唯一的组。这种情况可以防止一个目标文件在另一个目标文件中查找符号。

在某些实现中，初始目标文件必须导出符号以便重定位第二个目标文件。通过以下两种机制之一，可以满足此要求：

- 使初始目标文件成为第二个目标文件的显式依赖项。
- 使用 RTLD_PARENT 模式标志对第二个目标文件执行 dlopen(3C)。

如果初始目标文件是第二个目标文件的显式依赖项，则会将该初始目标文件指定给第二个目标文件所在的组。因此，初始目标文件可以为第二个目标文件的重定位提供符号。

如果许多目标文件可使用 dlopen(3C) 打开第二个目标文件，并且每个初始目标文件必须导出相同符号以满足第二个目标文件重定位的需要，则不能对第二个目标文件指定显式依赖项。在此情况下，可使用 RTLD_PARENT 标志扩充第二个目标文件的 dlopen(3C) 模式。此标志将导致第二个目标文件所在的组以显式依赖项的方式传播至初始目标文件。

这两种方法之间有一点区别。如果指定显式依赖项，则依赖项本身将成为第二个目标文件的 dlopen(3C) 依赖关系树的一部分，从而可用于使用 dlsym(3C) 的符号查找。如果使用 RTLD_PARENT 获取第二个目标文件，则使用 dlsym(3C) 的符号查找不能使用初始目标文件。

如果第二个目标文件是通过 dlopen(3C) 从具有全局符号可见性的初始目标文件获取的，则 RTLD_PARENT 模式既是冗余的，也是无害的。从应用程序或应用程序的依赖项之一调用 dlopen(3C) 时，通常会发生这种情况。

获取新符号

进程可以使用 dlsym(3C) 获取特定符号的地址。此函数采用句柄和符号名称，并将符号地址返回给调用者。该句柄通过以下方式指示符号搜索：

- 可通过指定目标文件的 dlopen(3C) 返回句柄。该句柄允许从指定目标文件及定义其依赖项树的目标文件获取符号。使用模式 RTLD_FIRST 返回的句柄仅允许从指定目标文件获取符号。
- 可通过其值为 0 的路径名的 dlopen(3C) 返回句柄。该句柄允许从关联链接映射的启动目标文件及定义其依赖项树的目标文件获取符号。通常，启动目标文件为动态可执行文件。对于关联链接映射，该句柄还允许从通过 dlopen(3C) 获取且模式为 RTLD_GLOBAL 的任何目标文件获取符号。使用模式 RTLD_FIRST 返回的句柄仅允许从关联链接映射的启动目标文件获取符号。
- 特殊句柄 RTLD_DEFAULT 和 RTLD_PROBE 允许从关联链接映射的启动目标文件及定义其依赖项树的目标文件获取符号。此句柄还允许从通过 dlopen(3C) 获取且与调用者同属一组的任何目标文件获取符号。使用 RTLD_DEFAULT 或 RTLD_PROBE 时采用在解析调用目标文件中的符号重定位时所用的同一模型。
- 特殊句柄 RTLD_NEXT 允许从调用者链接映射列表中的下一个关联目标文件获取符号。

在以下可能很常见的示例中，应用程序首先会将其他目标文件添加到其地址空间。然后，应用程序会使用 dlsym(3C) 来查找函数或数据符号。接下来，应用程序将使用这些符号来调用这些新目标文件中提供的服务。文件 main.c 包含以下代码：

```
#include    <stdio.h>
#include    <dlfcn.h>

int main()
{
        void *handle;
        int  *dptr, (*fptr)();

        if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
                (void) printf("dlopen: %s\n", dlerror());
                return (1);
        }

        if (((fptr = (int (*)())dlsym(handle, "foo")) == NULL) ||
            ((dptr = (int *)dlsym(handle, "bar")) == NULL)) {
                (void) printf("dlsym: %s\n", dlerror());
                return (1);
        }

        return ((*fptr)(*dptr));
}
```

首先会在文件 foo.so.1 中搜索符号 foo 和 bar，然后在与此文件关联的所有依赖项中搜索。然后，使用 return () 语句中的单个参数 bar 来调用函数 foo。

使用前面的文件 main.c 生成的应用程序 prog 包含下列依赖项。

```
$ ldd prog
        libc.so.1 =>     /lib/libc.so.1
```

如果在 dlopen(3C) 中指定的文件名的值为 0，则会首先在 prog 中搜索符号 foo 和 bar，然后在 /lib/libc.so.1 中搜索。

该句柄指示启动符号搜索所在根。搜索机制将从此根开始采用“重定位符号查找” [91]中所述的模型。

如果无法找到需要的符号，dlsym(3C) 将返回 NULL 值。在此情况下，可使用 dlerror(3C) 来指示失败的真正原因。在以下示例中，应用程序 prog 无法找到符号 bar。

```
$ prog
dlsym: ld.so.1: main: fatal: bar: can't find symbol
```

测试功能

使用特殊句柄 RTLD_DEFAULT 和 RTLD_PROBE，应用程序可以测试是否存在其他符号。

RTLD_DEFAULT 句柄采用运行时链接程序所用的同一规则来解析调用目标文件中的任何符号引用。请参见“缺省符号查找模型” [108]。应注意该模型的两个方面。

- 与动态可执行文件中的同一符号引用匹配的符号引用将绑定到与可执行文件中的引用关联的过程链接表项。请参见“过程链接表（特定于处理器）” [372]。动态链接的这种人为因素确保进程中的所有组件看到的是函数的单一地址。
- 如果在进程中目前装入的目标文件中无法找到可以满足非弱符号引用的符号定义，将启动延迟装入回退。将对装入的所有动态目标文件重复执行此回退，从而装入所有暂挂的延迟可装入目标文件，以尝试解析符号。该模型对尚未完整定义其依赖项的目标文件进行补偿。但是，该补偿可能会破坏延迟装入的优点。如果找不到重定位符号，可能会装入不需要的目标文件，或是全面装入所有的延迟可装入目标文件。

RTLD_PROBE 采用与 RTLD_DEFAULT 类似的模型，但是与标有 RTLD_DEFAULT 的模型在两个方面有所不同。RTLD_PROBE 只能绑定到显式符号定义，而不能绑定到可执行文件中的任何过程链接表项。此外，RTLD_PROBE 不会启动一个全面的延迟装入回退。RTLD_PROBE 是用来检测现有进程中符号是否存在的最合适的标志。

RTLD_DEFAULT 和 RTLD_PROBE 都可以启动显式延迟装入。目标文件可以引用函数，且该引用可以通过一个延迟可装入依赖项建立。调用该函数之前，可以使用 RTLD_DEFAULT 或 RTLD_PROBE 测试函数是否存在。由于目标文件会引用函数，因此首先要尝试装入关联的延迟依赖项。随后遵循 RTLD_DEFAULT 和 RTLD_PROBE 的规则以绑定到函数。在下面的示例中，RTLD_PROBE 调用用于触发延迟装入，以及在存在依赖项时绑定到装入的依赖项。

```
void foo()
{
```

```
        if (dlsym(RTLD_PROBE, "foo1")) {
                foo1(arg1);
                foo2(arg2);
                ....
}
```

要为功能性测试提供一个强大而灵活的模型，关联的延迟依赖项应显式标记为延迟。请参见“提供 dlopen（） 的替代项” [98]。该标记还在运行时提供更改延迟依赖项的方式。

RTLD_DEFAULT 或 RTLD_PROBE 的使用为未定义的弱引用的使用提供了一个更强大的替代方案，如“弱符号” [41]中所述。

使用插入

使用特殊句柄 RTLD_NEXT，应用程序可在符号作用域内查找下一个符号。例如，如果应用程序 prog 包含以下代码片段：

```
        if ((fptr = (int (*)())dlsym(RTLD_NEXT, "foo")) == NULL) {
                 (void) printf("dlsym: %s\n", dlerror());
                 return (1);
        }

        return ((*fptr)());
```

则会在与 prog 关联的共享目标文件（在此情况下为 /lib/libc.so.1）中搜索 foo。如果此代码片段包含在 图 3-1 “单个 dlopen（） 请求” 中显示的示例的文件 B.so.1 中，则仅会在 C.so.1 中搜索 foo。

使用 RTLD_NEXT 提供了使用符号插入的方法。例如，可通过前面的目标文件插入目标文件中的函数，然后扩充原始函数的处理。例如，在共享目标文件 malloc.so.1 中放置以下代码片段。

```
#include    <sys/types.h>
#include    <dlfcn.h>
#include    <stdio.h>

void *
malloc(size_t size)
{
        static void *(*fptr)() = 0;
        char        buffer[50];

        if (fptr == 0) {
                fptr = (void *(*)())dlsym(RTLD_NEXT, "malloc");
                if (fptr == NULL) {
                        (void) printf("dlopen: %s\n", dlerror());
                        return (NULL);
                }
        }
```

```
        (void) sprintf(buffer, "malloc: %#x bytes\n", size);
        (void) write(1, buffer, strlen(buffer));
        return ((*fptr)(size));
}
```

malloc.so.1 可插入到 malloc(3C) 通常所在的系统库 /lib/libc.so.1 之前。现在，在调用原始函数以完成分配之前，插入对 malloc（） 的调用：

```
$ cc -o malloc.so.1 -G -K pic malloc.c
$ cc -o prog file1.o file2.o .... -R. malloc.so.1
$ prog
malloc: 0x32 bytes
malloc: 0x14 bytes
....
```

或者，可使用以下命令实现相同插入：

```
$ cc -o malloc.so.1 -G -K pic malloc.c
$ cc -o prog main.c
$ LD_PRELOAD=./malloc.so.1  prog
malloc: 0x32 bytes
malloc: 0x14 bytes
....
```

注 - 使用任何插入方法的用户在处理任何可能的递归时都必须小心。前面的示例使用 sprintf(3C)，而不是直接使用 printf(3C) 来格式化诊断消息，以避免由于 printf(3C) 可能使用 malloc(3C) 而导致产生递归。

在动态可执行文件或预装入的目标文件中使用 RTLD_NEXT，可提供可预测的插入方法。在一般目标文件依赖项中使用此方法时应该十分小心，因为目标文件的实际装入顺序有时无法预测。

调试帮助

Oracle Solaris 运行时链接程序附带有调试库和调试 mdb(1) 模块。使用调试库，可以更详细地跟踪运行时链接过程。使用 mdb(1) 模块，可以进行交互式进程调试。

调试功能

运行时链接程序提供了调试功能，允许您详细地跟踪应用程序的运行时链接及其依赖项情况。使用此功能显示的信息类型应保持不变。不过，信息的确切格式可能随发行版的不同而有所变化。

不了解运行时链接程序的用户可能不熟悉某些调试输出。不过，您可能对其中许多方面不太感兴趣。

可使用环境变量 LD_DEBUG 来启用调试。所有调试输出都使用进程标识符作为前缀。必须使用一个或多个标记来扩充此环境变量，以指示所需调试的类型。

使用 LD_DEBUG=help，可以显示 LD_DEBUG 中可用的标记。

```
$ LD_DEBUG=help  prog
```

prog 可以是任何动态可执行文件。在将控制权转交给 prog 之前，进程终止并随后显示帮助信息。可执行文件的选择并不重要。

缺省情况下，会将所有调试输出发送到标准错误输出文件 stderr。可以使用 output 标记将调试输出定向到其他文件。例如，可以在名为 rtld-debug.txt 的文件中捕获帮助文本。

```
$ LD_DEBUG=help,output=rtld-debug.txt  prog
```

此外，还可以通过设置环境变量 LD_DEBUG_OUTPUT 来重定向调试输出。使用 LD_DEBUG_OUTPUT 时，进程标识符作为后缀添加到输出文件名。

如果同时指定 LD_DEBUG_OUTPUT 和 output 标记，则 LD_DEBUG_OUTPUT 优先。如果同时指定 LD_DEBUG_OUTPUT 和 output 标记，则 LD_DEBUG_OUTPUT 优先。将 output 标记与调用 fork(2) 的程序结合使用会导致每个进程都将调试输出写入同一文件。调试输出因此会变得混乱且不完整。在这种情况下应使用 LD_DEBUG_OUTPUT，以将每个进程的调试输出定向到唯一的文件。

对于安全的应用程序，不允许进行调试。

其中一个最有用的调试选项是显示运行时进行的符号绑定。以下示例使用很简单的动态可执行文件，该可执行文件依赖于两个局部共享目标文件。

```
$ cat bar.c
int bar = 10;
$ cc -o bar.so.1 -K pic -G bar.c

$ cat foo.c
int foo(int data)
{
return (data);
}
$ cc -o foo.so.1 -K pic -G foo.c

$ cat main.c
extern int foo();
extern int bar;

int main()
{
        return (foo(bar));
}
```

```
$ cc -o prog main.c -R/tmp:. foo.so.1 bar.so.1
```

通过设置 LD_DEBUG=bindings，可以显示运行时符号绑定。

```
$ LD_DEBUG=bindings  prog
11753: ....
11753: binding file=prog to file=./bar.so.1: symbol bar
11753: ....
11753: transferring control: prog
11753: ....
11753: binding file=prog to file=./foo.so.1: symbol foo
11753: ....
```

即时重定位需要的符号 bar 是在应用程序获得控制权之前绑定的。然而，符号 foo 是延迟重定位所需的，将在应用程序获得第一次调用函数的控制权之后绑定。此重定位说明了延迟绑定的缺省模式。如果设置了环境变量 LD_BIND_NOW，则所有符号绑定都会在应用程序获取控制权之前进行。

通过设置 LD_DEBUG=bindings,detail，可提供有关实际绑定位置的实际地址和相对地址的其他信息。

可以使用 LD_DEBUG 来显示使用的各个搜索路径。例如，通过设置 LD_DEBUG=libs，可以显示用于查找所有依赖项的搜索路径机制。

```
$ LD_DEBUG=libs  prog
11775:
11775: find object=foo.so.1; searching
11775:  search path=/tmp:.  (RUNPATH/RPATH from file prog)
11775:  trying path=/tmp/foo.so.1
11775:  trying path=./foo.so.1
11775:
11775: find object=bar.so.1; searching
11775:  search path=/tmp:.  (RUNPATH/RPATH from file prog)
11775:  trying path=/tmp/bar.so.1
11775:  trying path=./bar.so.1
11775: ....
```

应用程序 prog 中记录的运行路径将影响两个依赖项 foo.so.1 和 bar.so.1 的搜索。

同样，可通过设置 LD_DEBUG=symbols 来显示每个符号查找的搜索路径。组合使用 symbols 和 bindings 可生成符号重定位进程的完整信息。

```
$ LD_DEBUG=bindings,symbols  prog
11782: ....
11782: symbol=bar;  lookup in file=./foo.so.1  [ ELF ]
11782: symbol=bar;  lookup in file=./bar.so.1  [ ELF ]
11782: binding file=prog to file=./bar.so.1: symbol bar
11782: ....
11782: transferring control: prog
11782: ....
11782: symbol=foo;  lookup in file=prog  [ ELF ]
11782: symbol=foo;  lookup in file=./foo.so.1  [ ELF ]
11782: binding file=prog to file=./foo.so.1: symbol foo
```

```
11782: ....
```

在前面的示例中，未在应用程序 prog 中搜索符号 bar。省略数据引用查找是因为在处理复制重定位时使用了优化。有关此重定位类型的更多详细信息，请参见“复制重定位” [170]。

调试器模块

调试器模块提供了一组可在 mdb(1) 下装入的 dcmds 和 walkers。此模块可用于检查运行时链接程序的各种内部数据结构。许多调试信息都要求您熟悉运行时链接程序的内部构造。并且该信息会随发行版的不同而有所变化。但是，这些数据结构中的某些元素显示了动态链接进程的基本组件，有助于进行一般调试。

以下示例显示了一些将 mdb(1) 与调试器模块一起使用的简单方案。

```
$ cat main.c
#include    <dlfnc.h>

int main()
{
        void *handle;
        void (*fptr)();

        if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL)
                return (1);

        if ((fptr = (void (*)())dlsym(handle, "foo")) == NULL)
                return (1);

        (*fptr)();
        return (0);
}
$ cc -o main main.c -R.
```

如果 mdb(1) 尚未自动装入调试器模块 ld.so，则显式装入该模块。之后可以检查调试器模块的功能。

```
$ mdb main
> ::load ld.so
> ::dmods -l ld.so

ld.so
------------------------------------------------------------------
  dcmd Bind                 - Display a Binding descriptor
  dcmd Callers              - Display Rt_map CALLERS binding descriptors
  dcmd Depends              - Display Rt_map DEPENDS binding descriptors
  dcmd ElfDyn               - Display Elf_Dyn entry
  dcmd ElfEhdr              - Display Elf_Ehdr entry
  dcmd ElfPhdr              - Display Elf_Phdr entry
  dcmd Groups               - Display Rt_map GROUPS group handles
```

```
  dcmd GrpDesc              - Display a Group Descriptor
  dcmd GrpHdl               - Display a Group Handle
  dcmd Handles              - Display Rt_map HANDLES group descriptors
  ....
> ::bp main
> :r
```

进程中的每个动态目标文件都表示为链接映射 `Rt_map`，该映射在链接映射列表中对其进行维护。可使用 `Rt_maps` 显示进程的所有链接映射。

```
> ::Rt_maps
Link-map lists (dynlm_list): 0xffbfe0d0
-----------------------------------------------
  Lm_list: 0xff3f6f60  (LM_ID_BASE)
  -----------------------------------------------
    lmco        rtmap       ADDR()     NAME()
    -----------------------------------------------
    [0xc]       0xff3f0fdc 0x00010000 main
    [0xc]       0xff3f1394 0xff280000 /lib/libc.so.1
  -----------------------------------------------
  Lm_list: 0xff3f6f88  (LM_ID_LDSO)
  -----------------------------------------------
    [0xc]       0xff3f0c78 0xff3b0000 /lib/ld.so.1
```

可使用 `Rt_map` 显示单个链接映射。

```
> 0xff3f9040::Rt_map
Rt_map located at: 0xff3f9040
     NAME: main
 PATHNAME: /export/home/user/main
     ADDR: 0x00010000         DYN: 0x000207bc
     NEXT: 0xff3f9460        PREV: 0x00000000
      FCT: 0xff3f6f18    TLSMODID:          0
     INIT: 0x00010710        FINI: 0x0001071c
   GROUPS: 0x00000000     HANDLES: 0x00000000
  DEPENDS: 0xff3f96e8     CALLERS: 0x00000000
....
```

可使用 `ElfDyn dcmd` 显示目标文件的 `.dynamic` 节。以下示例显示了前 4 项。

```
> 0x000207bc,4::ElfDyn
Elf_Dyn located at: 0x207bc
    0x207bc  NEEDED       0x0000010f
Elf_Dyn located at: 0x207c4
    0x207c4  NEEDED       0x00000124
Elf_Dyn located at: 0x207cc
    0x207cc  INIT         0x00010710
Elf_Dyn located at: 0x207d4
    0x207d4  FINI         0x0001071c
```

mdb(1) 在设置延迟断点时也很有用。在此示例中，函数 `foo ()` 中的断点可能会很有用。但是，在对 `foo.so.1` 执行 dlopen(3C) 之前，调试器不知道此符号。延迟断点会指示调试器在装入动态目标文件时设置实际断点。

```
> ::bp foo.so.1`foo
> :c
> mdb: You've got symbols!
> mdb: stop at foo.so.1`foo
mdb: target stopped at:
foo.so.1`foo:   save      %sp, -0x68, %sp
```

此时，已经装入了新目标文件：

```
> *ld.so`lml_main::Rt_maps
lmco    rtmap       ADDR()     NAME()
-----------------------------------------------
[0xc]   0xff3f0fdc 0x00010000 main
[0xc]   0xff3f1394 0xff280000 /lib/libc.so.1
[0xc]   0xff3f9ca4 0xff380000 ./foo.so.1
[0xc]   0xff37006c 0xff260000 ./bar.so.1
```

foo.so.1 的链接映射显示了 dlopen(3C) 返回的句柄。可使用 Handles 来扩展此结构。

```
> 0xff3f9ca4::Handles -v
HANDLES for ./foo.so.1
-----------------------------------------------
  HANDLE: 0xff3f9f60 Alist[used 1: total 1]
  -------------------------------------------
    Group Handle located at: 0xff3f9f28
    -------------------------------------------
    owner:               ./foo.so.1
    flags: 0x00000000    [ 0 ]
   refcnt:          1    depends: 0xff3f9fa0 Alist[used 2: total 4]
    -------------------------------------------
    Group Descriptor located at: 0xff3f9fac
       depend: 0xff3f9ca4    ./foo.so.1
        flags: 0x00000003    [ AVAIL-TO-DLSYM,ADD-DEPENDENCIES ]
    -------------------------------------------
    Group Descriptor located at: 0xff3f9fd8
       depend: 0xff37006c    ./bar.so.1
        flags: 0x00000003    [ AVAIL-TO-DLSYM,ADD-DEPENDENCIES ]
```

句柄的依赖项由一组链接映射组成，这些链接映射表示可满足 dlsym(3C) 请求的句柄的目标文件。在此情况下，依赖项为 foo.so.1 和 bar.so.1。

注 - 前面的示例提供了调试器模块功能的基本指南，但确切的命令、用法和输出可能会随发行版的不同而有所变化。有关系统中可用的确切功能，请参阅 mdb(1) 用法和帮助信息。

♦ ♦ ♦ 第 4 章

4 共享目标文件

共享目标文件是一种由链接编辑器创建并通过指定 -G 选项生成的输出形式。在以下示例中，共享目标文件 libfoo.so.1 根据输入文件 foo.c 生成。

```
$ cc -o libfoo.so.1 -G -K pic foo.c
```

共享目标文件是一个不可分割的单元，根据一个或多个可重定位目标文件生成。共享目标文件可以与动态可执行文件绑定在一起以形成可运行进程。顾名思义，共享目标文件可供多个应用程序共享。由于这种潜在的深远影响，因此与先前章节相比，本章更深入地介绍了这种链接编辑器输出形式。

对于要绑定到动态可执行文件或其他共享目标文件的共享目标文件，首先它必须可用于链接编辑所需的输出文件。在此链接编辑过程中，会解释所有输入共享目标文件，就像已将这些共享目标文件添加到要生成的输出文件的逻辑地址空间。共享目标文件的所有功能均可供输出文件使用。

所有输入共享目标文件都将成为此输出文件的依赖项。此输出文件中维护了少量簿记信息以描述这些依赖项。运行时链接程序将在创建可运行进程的过程中解释此信息并完成这些共享目标文件的处理。

以下各节详述了如何在编译环境和运行时环境中使用共享目标文件。“运行时链接” [18]中介绍了这些环境。

命名约定

链接编辑器和运行时链接程序都不会根据文件名解释文件。将检查所有文件以确定其 ELF 类型（请参见“ELF 头” [274]）。链接编辑器使用此信息来推导文件的处理要求。但是，共享目标文件通常遵循两种命名约定之一，具体取决于这些目标文件是用作编译环境的一部分还是用作运行时环境的一部分。

用作编译环境的一部分时，共享目标文件由链接编辑器读取并处理。虽然可以在传递到链接编辑器的命令中通过显式文件名指定这些共享目标文件，但是通常使用 -l 选项来利用链接编辑器的库搜索功能。请参见“共享目标文件处理” [28]。

应该使用前缀 lib 和后缀 .so 来指定适用于此链接编辑器处理的共享目标文件。例如，/lib/libc.so 便是可用于编译环境的标准 C 库的共享目标文件。根据约定，64 位共享目

标文件位于 lib 目录中名为 64 的子目录中。例如，/lib/libc.so.1 的对应 64 位名称是 /lib/64/libc.so.1。

用作运行时环境的一部分时，共享目标文件由运行时链接程序读取和处理。要允许在一系列软件发行版中对共享目标文件的导出接口进行更改，可将共享目标文件作为版本化文件名提供。

版本化文件名通常采用 .so 后缀后跟版本号的形式。例如，/lib/libc.so.1 便是可用于运行时环境的第一版标准 C 库的共享目标文件。

如果从不打算在编译环境中使用共享目标文件，则可能会从共享目标文件名称中删除常规的 lib 前缀。仅用于 dlopen(3C) 的共享目标文件便是此类共享目标文件。仍然建议使用后缀 .so 来指示实际文件类型。此外，强烈建议使用版本号以便在一系列软件发行版中提供正确的共享目标文件绑定。第 9 章 接口和版本控制更详细地介绍了版本控制。

注 - dlopen(3C) 中使用的共享目标文件名称通常表示为不包含 "/" 的简单文件名。然后，运行时链接程序可以使用一组规则来查找实际文件。有关更多详细信息，请参见“装入其他目标文件” [96]。

记录共享目标文件名称

缺省情况下，动态可执行文件或共享目标文件中的依赖项记录将是链接编辑器所引用的关联共享目标文件的文件名。例如，根据同一共享目标文件 libfoo.so 生成的以下动态可执行文件会导致对同一依赖项具有不同的解释。

```
$ cc -o ../tmp/libfoo.so -G foo.o
$ cc -o prog main.o -L../tmp -lfoo
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED        0x123         libfoo.so.1

$ cc -o prog main.o ../tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED        0x123         ../tmp/libfoo.so

$ cc -o prog main.o /usr/tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED        0x123         /usr/tmp/libfoo.so
```

如这些示例所示，这种记录依赖项机制会因编译技术的不同而出现不一致性。此外，在链接编辑过程中引用的共享目标文件的位置也可能会与已安装系统上的共享目标文件的最终位置有所不同。为了提供更一致的指定依赖项的方法，共享目标文件可以在自身中记录运行时据以引用共享目标文件的文件名。

在链接编辑共享目标文件过程中，可以使用 -h 选项在共享目标文件自身中记录其运行时名称。在以下示例中，将在文件自身中记录共享目标文件的运行时名称 libfoo.so.1。此标识称为 *soname*。

```
$ cc -o ../tmp/libfoo.so -G -K pic -h libfoo.so.1 foo.c
```

以下示例说明如何使用 elfdump(1) 和引用具有 SONAME 标记的项来显示 *soname* 记录。

```
$ elfdump -d ../tmp/libfoo.so | grep SONAME
     [1]  SONAME        0x123         libfoo.so.1
```

链接编辑器处理包含 *soname* 的共享目标文件时，此名称便是作为要生成的输出文件中的依赖项记录的名称。

如果在上一示例的创建动态可执行文件 prog 的过程中使用此新版本的 libfoo.so，则所有三种创建可执行文件的方法都会记录同一依赖项。

```
$ cc -o prog main.o -L../tmp -lfoo
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED        0x123         libfoo.so

$ cc -o prog main.o ../tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED        0x123         libfoo.so

$ cc -o prog main.o /usr/tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED        0x123         libfoo.so
```

在上述示例中，使用 -h 选项来指定不包含 "/" 的简单文件名。借助此约定，运行时链接程序可以使用一组规则来查找实际文件。有关更多详细信息，请参见“查找共享目标文件依赖项” [88]。

在归档中包含共享目标文件

如果共享目标文件始终通过归档库进行处理，则在共享目标文件中记录 *soname* 是基本的机制。

归档可以根据一个或多个共享目标文件生成，并可用于生成动态可执行文件或共享目标文件。可以从归档中提取共享目标文件来满足链接编辑的要求。与处理可重定位目标文件（串联成要创建的输出文件）不同，从归档中提取的任何共享目标文件都将记录为依赖项。有关归档提取条件的更多详细信息，请参见“归档处理” [27]。

归档成员的名称由链接编辑器构造，并且由归档名称和归档中的目标文件串联而成。例如：

```
$ cc -o libfoo.so.1 -G -K pic foo.c
$ ar -r libfoo.a libfoo.so.1
$ cc -o main main.o libfoo.a
$ elfdump -d main | grep NEEDED
     [1]  NEEDED        0x123         libfoo.a(libfoo.so.1)
```

由于具有此串联名称的文件无法在运行时存在，因此，在共享目标文件中提供 *soname* 是生成有意义的依赖项运行时文件名的唯一方法。

注 - 运行时链接程序不会从归档中提取目标文件。因此，在此示例中，必须从归档中提取所需的共享目标文件依赖项，并使其可用于运行时环境。

已记录名称冲突

使用共享目标文件创建动态可执行文件或其他共享目标文件时，链接编辑器会执行多项一致性检查。这些检查可确保输出文件中记录的任何依赖项名称都是唯一的。

如果在链接编辑过程中用作输入文件的两个共享目标文件包含相同的 *soname*，依赖项名称可能会发生冲突。例如：

```
$ cc -o libfoo.so -G -K pic -h libsame.so.1 foo.c
$ cc -o libbar.so -G -K pic -h libsame.so.1 bar.c
$ cc -o prog main.o -L. -lfoo -lbar
ld: fatal: recording name conflict: file './libfoo.so' and \
    file './libbar.so' provide identical dependency names: libsame.so.1
```

如果某个没有已记录 *soname* 的共享目标文件的文件名与同一链接编辑过程使用的其他共享目标文件的 *soname* 匹配，则也会出现类似的错误情况。

如果要生成的共享目标文件的运行时名称与它的某个依赖项匹配，则链接编辑器也会报告名称冲突。

```
$ cc -o libbar.so -G -K pic -h libsame.so.1 bar.c -L. -lfoo
ld: fatal: recording name conflict: file './libfoo.so' and \
    -h option provide identical dependency names: libsame.so.1
```

具有依赖项的共享目标文件

共享目标文件可以有其自己的依赖项。运行时链接程序查找共享目标文件依赖项使用的搜索规则在“运行时链接程序搜索的目录” [88]中介绍。如果共享目标文件没有位于其中一个缺省搜索目录中，则必须将查找位置明确告知运行时链接程序。对于 32 位目标文件，缺省搜索目录为 `/lib` 和 `/usr/lib`。对于 64 位目标文件，缺省搜索目录为 `/lib/64` 和 `/usr/lib/64`。指明非缺省搜索路径要求的首选机制是在具有依赖项的目标文件中记录运行路径。使用链接编辑器的 `-R` 选项可以记录 运行路径。

在以下示例中，共享目标文件 `libfoo.so` 依赖于 `libbar.so`；后者在运行时应位于目录 `/home/me/lib` 中，否则将位于缺省位置中。

```
$ cc -o libbar.so -G -K pic bar.c
$ cc -o libfoo.so -G -K pic foo.c -R/home/me/lib -L. -lbar
$ elfdump -d libfoo.so | egrep "NEEDED|RUNPATH"
    [1]  NEEDED        0x123         libbar.so.1
    [2]  RUNPATH       0x456         /home/me/lib
```

共享目标文件负责指定查找其依赖项所需的所有运行路径。所有在动态可执行文件中指定的运行路径只用于查找动态可执行文件的依赖项。不能使用这些运行路径来查找共享目标文件的任何依赖项。

LD_LIBRARY_PATH 系列的环境变量的作用域更具全局性。运行时链接程序使用借助此变量指定的所有路径名来搜索任意共享目标文件依赖项。虽然这些环境变量可用作影响运行时链接程序搜索路径的临时机制，但是强烈建议不要在生产软件中使用这些环境变量。有关更全面的介绍，请参见“运行时链接程序搜索的目录” [88]。

依赖项排序

动态可执行文件和共享目标文件都依赖于相同的常用共享目标文件时，这些目标文件的处理顺序可能会变得更难以预测。

例如，假设共享目标文件开发者生成的 libfoo.so.1 具有以下依赖项：

```
$ ldd libfoo.so.1
        libA.so.1 =>     ./libA.so.1
        libB.so.1 =>     ./libB.so.1
        libC.so.1 =>     ./libC.so.1
```

如果使用此共享目标文件创建动态可执行文件 prog，并定义对 libC.so.1 的显式依赖项，则生成的共享目标文件顺序将如下：

```
$ cc -o prog main.c -R. -L. -lC -lfoo
$ ldd prog
        libC.so.1 =>     ./libC.so.1
        libfoo.so.1 =>   ./libfoo.so.1
        libA.so.1 =>     ./libA.so.1
        libB.so.1 =>     ./libB.so.1
```

动态可执行文件 prog 的构造将会影响对共享目标文件 libfoo.so.1 依赖项处理顺序的任何要求。

专门致力于符号插入和 .init 节处理的开发者应意识到共享目标文件处理顺序可能存在这种变化。

作为过滤器的共享目标文件

可以定义共享目标文件以将其用作过滤器。此技术涉及将过滤器提供的接口与备用共享目标文件进行关联。在运行时，此备用共享目标文件可提供过滤器所提供的一个或多个接口。此备用共享目标文件称为 *filtee*。*filtee* 的生成方式与任意共享目标文件的生成方式相同。

过滤提供了一种从运行时环境中提取编译环境的机制。在链接编辑时，将绑定到过滤器接口的符号引用解析为过滤器符号定义。在运行时，可以将绑定到过滤器接口的符号引用重定向到备用共享目标文件。

通过使用 mapfile 关键字 FILTER 或 AUXILIARY，可以将在共享目标文件中定义的单个接口定义为过滤器。或者，共享目标文件可以使用链接编辑器的 -F 或 -f 选项将共享目标文件提供的所有接口定义为过滤器。通常单独使用这些技术，但也可在同一共享目标文件中组合使用这些技术。

存在两种过滤形式。

标准过滤

此过滤只需要一个用于要过滤的接口的符号表项。在运行时，必须通过 *filtee* 实现过滤器符号定义。

使用链接编辑器的 mapfile 关键字 FILTER 或链接编辑器的 -F 选项，可以定义接口以将其用作标准过滤器。此 mapfile 关键字或选项使用必须在运行时提供符号定义的一个或多个 *filtee* 的名称进行限定。

在运行时无法处理的 *filtee* 会被跳过。如果在 *filtee* 中找不到标准过滤器符号，则也将导致跳过此 *filtee*。在这两种情况下，无法使用过滤器提供的符号定义来实现此符号查找。

辅助过滤

此过滤提供类似于标准过滤的机制，而且该过滤器还提供一种对应于辅助过滤器接口的回退实现。在运行时，可以通过 *filtee* 实现符号定义。

使用链接编辑器的 mapfile 关键字 AUXILIARY 或链接编辑器的 -f 选项，可以定义接口以将其用作辅助过滤器。此 mapfile 关键字或选项使用在运行时可提供符号定义的一个或多个 *filtee* 的名称进行限定。

在运行时无法处理的 *filtee* 会被跳过。如果在 *filtee* 中找不到辅助过滤器符号，则也将导致跳过 *filtee*。在这两种情况下，将使用过滤器提供的符号定义来实现此符号查找。

生成标准过滤器

要生成标准过滤器，应先定义要对其应用过滤的 *filtee*。以下示例将生成 *filtee* filtee.so.1，并提供符号 foo 和 bar。

```
$ cat filtee.c
char *bar = "defined in filtee";

char *foo()
{
        return("defined in filtee");
}
```

```
$ cc -o filtee.so.1 -G -K pic filtee.c
```

可以通过以下两种方法之一提供标准过滤。要将共享目标文件提供的所有接口都声明为过滤器，可使用链接编辑器的 -F 选项。要将共享目标文件的单个接口声明为过滤器，可使用链接编辑器 mapfile 和 FILTER 关键字。

在以下示例中，会将共享目标文件 filter.so.1 定义为过滤器。filter.so.1 提供符号 foo 和 bar ，是针对 *filtee* filtee.so.1 的过滤器。在此示例中，使用环境变量 LD_OPTIONS 来阻止编译器动程序解释 -F 选项。

```
$ cat filter.c
char *bar = NULL;

char *foo()
{
        return (NULL);
}
$ LD_OPTIONS='-F filtee.so.1' \
cc -o filter.so.1 -G -K pic -h filter.so.1 -R. filter.c
$ elfdump -d filter.so.1 | egrep "SONAME|FILTER"
     [2]  SONAME           0xee     filter.so.1
     [3]  FILTER           0xfb     filtee.so.1
```

创建动态可执行文件或共享目标文件时，链接编辑器可将标准过滤器 filter.so.1 作为依赖项引用。链接编辑器使用此过滤器符号表中的信息来实现任何符号解析。但是，在运行时，对该过滤器符号的任何引用都会导致增加对该 *filtee* filtee.so.1 的装入。运行时链接程序使用该 *filtee* 来解析 filter.so.1 定义的所有符号。如果未找到此 *filtee*，或者在此 *filtee* 中未找到过滤器符号，则查找该符号时会跳过此过滤器。

例如，以下动态可执行文件 prog 引用符号 foo 和 bar；这两个符号在链接编辑过程中通过过滤器 filter.so.1 进行解析。执行 prog 会导致从 *filtee* filtee.so.1 中获取 foo 和 bar，而不是从过滤器 filter.so.1 中获取。

```
$ cat main.c
extern char *bar, *foo();

void main()
{
        (void) printf("foo is %s: bar is %s\n", foo(), bar);
}
$ cc -o prog main.c -R. filter.so.1
$ prog
foo is defined in filtee: bar is defined in filtee
```

在以下示例中，共享目标文件 filter.so.2 将它的一个接口 foo 定义为 *filtee* filtee.so.1 的过滤器。

注 - 由于未提供 foo（） 的源代码，因此将使用 mapfile 关键字 FUNCTION 以确保创建 foo 的符号表项。

```
$ cat filter.c
char *bar = "defined in filter";
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=FUNCTION; FILTER=filtee.so.1 };
};
$ cc -o filter.so.2 -G -K pic -h filter.so.2 -M mapfile -R. filter.c
$ elfdump -d filter.so.2 | egrep "SONAME|FILTER"
     [2]  SONAME           0xd8     filter.so.2
     [3]  SUNW_FILTER      0xfb     filtee.so.1
$ elfdump -y filter.so.2 | egrep "foo|bar"
     [1]  F    [3] filtee.so.1      foo
    [10]  D        <self>           bar
```

在运行时，对过滤器符号 `foo` 的任何引用都会导致增加对 *filtee* `filtee.so.1` 的装入。运行时链接程序使用该 *filtee* 仅解析 `filter.so.2` 定义的符号 `foo`。对符号 `bar` 的引用始终使用 `filter.so.2` 中的符号，因为没有为此符号定义 *filtee* 处理。

例如，以下动态可执行文件 `prog` 引用符号 `foo` 和 `bar`，这两个符号在链接编辑过程中通过过滤器 `filter.so.2` 进行解析。执行 `prog` 会导致从 *filtee* `filtee.so.1` 中获取 `foo`，从过滤器 `filter.so.2` 中获取 `bar`。

```
$ cc -o prog main.c -R. filter.so.2
$ prog
foo is defined in filtee: bar is defined in filter
```

在这些示例中，*filtee* `filtee.so.1` 仅与过滤器关联。不能使用该 *filtee* 从任何其他可能作为 `prog` 执行结果而装入的目标文件中实现符号查找。

标准过滤器提供了一种用于定义现有共享目标文件的子集接口的便捷机制。使用标准过滤器，可以跨多个现有共享目标文件创建接口组。标准过滤器还提供一种将接口重定向到实现的方法。在 Oracle Solaris OS 中，可以使用多个标准过滤器。

`/lib/libxnet.so.1` 过滤器使用多个 *filtee*。该库提供来自 `/lib/libsocket.so.1`、`/lib/libnsl.so.1` 和 `/lib/libc.so.1` 的套接字和 XTI 接口。

`libc.so.1` 定义运行时链接程序的接口过滤器。这些接口在以下两者之间提供了抽象：`libc.so.1` 的编译环境中引用的符号，以及 `ld.so.1`(1) 的运行时环境中生成的实际实现绑定。

`libnsl.so.1` 针对 `libc.so.1` 定义标准过滤器 `gethostname`(3C)。以前，`libnsl.so.1` 和 `libc.so.1` 针对该符号提供相同的实现。通过将 `libnsl.so.1` 建立为过滤器，只需要存在一种 `gethostname`() 实现。由于 `libnsl.so.1` 继续导出 `gethostname`() ，因此此库的接口将继续保持与以前的发行版兼容。

由于在运行时从不引用标准过滤器中的代码，因此，无需向任何定义为过滤器的函数中添加内容。任何过滤器代码都可能需要重定位，这样会在运行时处理过滤器期间

导致不必要的开销。建议将函数定义为空例程，或者直接从 mapfile 进行定义。请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。

在过滤器中生成数据符号时，始终将数据与节关联。可通过在可重定位目标文件中定义符号来生成此关联。也可以通过在 mapfile 中定义符号并使用 size 声明和 *no* value 声明来生成此关联。请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。生成的数据定义可确保正确建立来自动态可执行文件的引用。

链接编辑器执行某些更为复杂的符号解析时，需要了解符号的属性（包括符号大小）。因此，应该在过滤器中生成符号，以便符号属性与 *filtee* 中的符号属性匹配。维护属性一致性可确保链接编辑过程使用与运行时所用的符号定义兼容的方式来分析过滤器。请参见“符号解析” [36]。

注 - 链接编辑器使用所处理的第一个可重定位文件的 ELF 类来控制所创建的目标文件类。使用链接编辑器的 -64 选项可以仅根据 mapfile 创建 64 位过滤器。

生成辅助过滤器

要生成辅助过滤器，应先定义要对其应用过滤的 *filtee*。以下示例将生成一个 *filtee* filtee.so.1，并提供符号 foo。

```
$ cat filtee.c
char *foo()
{
        return("defined in filtee");
}
$ cc -o filtee.so.1 -G -K pic filtee.c
```

可以通过以下两种方法之一提供辅助过滤。要将共享目标文件提供的所有接口都声明为辅助过滤器，可使用链接编辑器的 -f 选项。要将共享目标文件的单个接口声明为辅助过滤器，可使用链接编辑器 mapfile 和 AUXILIARY 关键字。

以下示例将 filter.so.1 共享目标文件定义为辅助过滤器。filter.so.1 提供 foo 和 bar 符号，并且是 *filtee* filtee.so.1 的辅助过滤器。在此示例中，使用环境变量 LD_OPTIONS 来阻止编译器动程序解释 -f 选项。

```
$ cat filter.c
char *bar = "defined in filter";

char *foo()
{
        return ("defined in filter");
}
$ LD_OPTIONS='-f filtee.so.1' \
cc -o filter.so.1 -G -K pic -h filter.so.1 -R. filter.c
```

```
$ elfdump -d filter.so.1 | egrep "SONAME|AUXILIARY"
     [2]  SONAME          0xee     filter.so.1
     [3]  AUXILIARY       0xfb     filtee.so.1
```

创建动态可执行文件或共享目标文件时，链接编辑器可将辅助过滤器 `filter.so.1` 引用为依赖项。链接编辑器使用此过滤器符号表中的信息来实现任何符号解析。但是，在运行时，对此过滤器符号的任何引用都会导致搜索 *filtee* `filtee.so.1`。如果找到了 *filtee*，运行时链接程序就会使用 *filtee* 解析由 `filter.so.1` 定义的任何符号。如果未找到此 *filtee*，或者在此 *filtee* 中未找到过滤器符号，则会使用过滤器中的原始符号。

例如，以下动态可执行文件 `prog` 引用符号 `foo` 和 `bar`；这两个符号在链接编辑过程中通过过滤器 `filter.so.1` 进行解析。执行 `prog` 会导致从 *filtee* `filtee.so.1` 中获取 `foo`，而不是从 `filter.so.1` 过滤器中获取。但是，`bar` 是从过滤器 `filter.so.1` 中获取的，因为此符号在 *filtee* `filtee.so.1` 中没有备选定义。

```
$ cat main.c
extern char *bar, *foo();

void main()
{
        (void) printf("foo is %s: bar is %s\n", foo(), bar);
}
$ cc -o prog main.c -R. filter.so.1
$ prog
foo is defined in filtee: bar is defined in filter
```

在以下示例中，共享目标文件 `filter.so.2` 将接口 `foo` 定义为针对 *filtee* `filtee.so.1` 的一个辅助过滤器。

```
$ cat filter.c
char *bar = "defined in filter";

char *foo()
{
        return ("defined in filter");
}
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { AUXILIARY=filtee.so.1 };
};
$ cc -o filter.so.2 -G -K pic -h filter.so.2 -M mapfile -R. filter.c
$ elfdump -d filter.so.2 | egrep "SONAME|AUXILIARY"
     [2]  SONAME          0xd8     filter.so.2
     [3]  SUNW_AUXILIARY  0xfb     filtee.so.1
$ elfdump -y filter.so.2 | egrep "foo|bar"
     [1]  A    [3] filtee.so.1     foo
    [10]  D        <self>          bar
```

在运行时，对过滤器符号 `foo` 的任何引用都会导致搜索 *filtee* `filtee.so.1`。如果找到 *filtee*，就会将 *filtee* 装入。然后，使用 *filtee* 来解析由 `filter.so.2` 定义的 `foo`。如果

未找到 *filtee*，则将使用由 `filter.so.2` 定义的 `foo` 符号。对符号 `bar` 的引用始终使用 `filter.so.2` 中的符号，因为没有为此符号定义 *filtee* 处理。

例如，以下动态可执行文件 `prog` 引用符号 `foo` 和 `bar`，这两个符号在链接编辑过程中通过过滤器 `filter.so.2` 进行解析。如果存在 *filtee* `filtee.so.1`，执行 `prog` 将导致从 *filtee* `filtee.so.1` 中获取 `foo`，并从过滤器 `filter.so.2` 获取 `bar`。

```
$ cc -o prog main.c -R. filter.so.2
$ prog
foo is defined in filtee: bar is defined in filter
```

如果不存在 *filtee* `filtee.so.1`，执行 `prog` 将导致从过滤器 `filter.so.2` 获取 `foo` 和 `bar`。

```
$ prog
foo is defined in filter: bar is defined in filter
```

在这些示例中，*filtee* `filtee.so.1` 仅与过滤器关联。不能使用该 *filtee* 从任何其他可能作为 `prog` 执行结果而装入的目标文件中实现符号查找。

辅助过滤器提供了一种用于定义现有共享目标文件的备用接口的机制。在 Oracle Solaris OS 中使用此机制可以提供优化的硬件功能以及平台特定的共享目标文件。有关示例，请参见“特定于功能的共享目标文件” [227]、“特定于指令集的共享目标文件” [229]和“特定于系统的共享目标文件” [231]。

注 - 环境变量 `LD_NOAUXFLTR` 可设置为禁用运行时链接程序辅助过滤器处理。由于通常使用辅助过滤器来提供平台特定的优化，因此，该选项在评估 *filtee* 用法及其性能影响方面很有用。

过滤组合

可以在同一个共享目标文件中定义个别接口，这些接口可定义标准过滤器，以及定义辅助过滤器的个别接口。通过使用 `mapfile` 关键字 `FILTER` 和 `AUXILIARY` 来指定所需的 *filtees*，可以实现这种过滤器定义组合。

使用 `-F` 或 `-f` 选项将其所有接口都定义为过滤器的共享目标文件可以是标准过滤器，也可以是辅助过滤器。

共享目标文件可以定义单个接口以将其用作过滤器，同时还可以将目标文件的所有接口都定义为过滤器。在这种情况下，首先处理针对接口定义的单个过滤。如果无法针对单个接口过滤器建立 *filtee*，则针对过滤器的所有接口定义的 *filtee* 会在适用时提供回退。

例如，考虑 `filter.so.1` 过滤器。此过滤器使用链接编辑器的 `-f` 选项，针对 *filtee* `filtee.so.1` 将所有接口都定义为辅助过滤器。`filter.so.1` 还使用 `mapfile` 关键字 `FILTER`，针对 *filtee* `foo.so.1` 将单个接口 `foo` 定义为标准过滤器。`filter.so.1` 还使用 `mapfile` 关键字 `AUXILIARY`，针对 *filtee* `bar.so.1` 将单个接口 `bar` 定义为辅助过滤器。

对 foo 的外部引用会导致处理 *filtee* foo.so.1。如果在 foo.so.1 中找不到 foo，则不会进一步处理过滤器。在这种情况下，不会执行回退处理，因为已将 foo 定义为标准过滤器。

对 bar 的外部引用会导致处理 *filtee* bar.so.1。如果在 bar.so.1 中找不到 bar，则处理会回退到 *filtee* filtee.so.1。在这种情况下，会执行回退处理，因为已将 bar 定义为辅助过滤器。如果在 filtee.so.1 中找不到 bar，则最终会使用 filter.so.1 过滤器中的 bar 的定义来解析外部引用。

filtee 处理

运行时链接程序处理过滤器时会延迟装入 *filtee*，直到引用过滤器符号。这种实现类似于执行 dlopen(3C) 的过滤器，在需要 filtee 时，对其每个 *filtee* 使用 *RTLD_LOCAL* 模式。这种实现考虑了由 ldd(1) 等生成的依赖项报告中存在的差异。

创建过滤器时，可以使用链接编辑器的 -z loadfltr 选项，实现在运行时立即处理 *filtees*。此外，通过将 LD_LOADFLTR 环境变量设置为任意值，可触发在某个进程中立即处理所有 *filtee*。

部分 II

快速参考

♦ ♦ ♦ 第 5 章

5 链接编辑器快速参考

以下各节提供了最常用的链接编辑器方案的简单概述（也可称为备忘单）。有关链接编辑器生成的输出模块种类的介绍，请参见“链接编辑” [17]。

提供的示例说明了提供给编译器驱动程序的链接编辑器选项，即调用链接编辑器最常用的机制。在这些示例中，使用了 cc(1)。请参见“使用编译器驱动程序” [24]。

链接编辑器不会对任何输入文件名赋予任何意义。每个文件都会被打开并检查，以确定其需要的处理类型。请参见“输入文件处理” [27]。

可以使用 -l 选项输入遵循 lib*X*.so 命名约定的共享目标文件以及遵循 lib*X*.a 命名约定的归档库。请参见“库命名约定” [29]。这在允许使用 -L 选项指定搜索路径方面提供了更大的灵活性。请参见“链接编辑器搜索的目录” [31]。

随着时间的推移，链接编辑器添加了许多功能以用于创建高质量的目标文件。利用这些功能，可以在各种运行时环境中高效而可靠地使用目标文件。但是，为了确保与现有生成环境的向下兼容性，其中许多功能在缺省情况下未启用。例如，直接绑定和延迟装入等功能必须显式启用。链接编辑器提供了 -z guidance 选项，以帮助简化选择所要应用的功能的过程。请求指导时，链接编辑器可以发出警告指导消息。这些消息会建议要使用的选项以及其他相关更改，从而有助于生成更高质量的目标文件。由于链接编辑器中会添加新功能，或者会发现更好的做法来生成更高品质的目标文件，因此指导消息可能会随着时间而变化。请参见 ld(1)。

链接编辑器本质上以静态或动态两种模式之一运行。

静态模式

使用 -d n 选项时会选择静态模式，通过此模式可创建可重定位目标文件和静态可执行文件。在此模式下，可以接受的输入形式只有可重定位目标文件和归档库。使用 -l 选项可以对归档库进行搜索。

创建可重定位目标文件

要创建可重定位目标文件，请使用 -r 选项。

```
$ ld -r -o temp.o file1.o file2.o file3.o ....
```

创建静态可执行文件

注 - 静态可执行文件的使用将受到限制。请参见"静态可执行文件" [18]。静态可执行文件通常包含特定于平台的实现详细信息，这会限制可执行文件在备用平台或操作系统版本上运行的能力。Oracle Solaris 共享目标文件的许多实现取决于动态链接功能，例如 dlopen(3C) 和 dlsym(3C)。请参见"装入其他目标文件" [96]。这些功能对于静态可执行文件不可用。

要创建静态可执行文件，请使用 -d n 选项而不要使用 -r 选项。

```
$ cc -dn -o prog file1.o file2.o file3.o ....
```

-a 选项可用于指示静态可执行文件的创建。使用 -d n 而不使用 -r 选项隐含表示为使用 -a。

动态模式

动态模式是链接编辑器的缺省操作模式。通过指定 -d y 选项可以强制执行此模式；但是只要不使用 -d n 选项，便隐含表示为使用此模式。

在此模式下，可以接受的输入形式包括可重定位目标文件、共享目标文件和归档库。使用 -l 选项可以进行目录搜索，即搜索每个目录以查找共享目标文件。如果未找到任何共享目标文件，则会搜索同一目录来查找归档库。使用 -B static 选项可以强制仅对归档库执行搜索。请参见"同时链接共享目标文件和归档" [30]。

创建共享目标文件

- 要创建共享目标文件，请使用 -G 选项。由于缺省情况下隐含表示执行 -d y 选项，故此选项是可选的。
- 建议使用链接编辑器的 -z guidance 选项。指导消息提供有关链接编辑器选项和其他操作的建议，以帮助改善生成的目标文件。
- 输入可重定位目标文件应当通过与位置无关的代码生成。例如，C 编译器可以使用 -K pic 选项生成与位置无关的代码。请参见"与位置无关的代码" [162]。使用 -z text 选项可以强制实施此要求。
- 避免包含未使用的可重定位目标文件。或者，请使用 -z discard-unused=sections 选项，此选项指示链接编辑器删除未引用的 ELF 节。请参见"删除未使用的材料" [164]。

- 应用程序寄存器是 SPARC 体系结构的一项功能，可保留以供最终用户使用。供外部使用的 SPARC 共享目标文件应当对 C 编译器使用 `-xregs=no%appl` 选项，以便确保共享目标文件不使用任何应用程序寄存器。这样可以使应用程序寄存器对于任何外部用户均可用，同时不影响共享目标文件的实现。
- 通过定义应从共享目标文件可见的全局符号并将其他任何全局符号缩减到局部作用域，来建立共享目标文件的公共接口。该定义由 `-M` 选项与关联的 `mapfile` 共同提供。请参见第 9 章 接口和版本控制。
- 请针对共享目标文件使用版本化名称以便将来可以升级。请参见“协调版本化文件名” [224]。
- 自包含的共享目标文件可以提供最大的灵活性。目标文件表示所有依赖性需要时会生成这些共享目标文件。使用 `-z defs` 可强制实现这种自包含。请参见“生成共享目标文件输出文件” [40]。
- 避免包含不需要的依赖项。请使用带有 `-u` 选项的 `ldd` 来检测并删除不需要的依赖项。请参见“共享目标文件处理” [28]。或者，使用 `-z discard-unused=dependencies` 选项，此选项指示链接编辑器将依赖项仅记录到所引用的目标文件中。
- 如果要生成的共享目标文件依赖于其他共享目标文件，则表明应该使用 `-z lazyload` 选项以延迟方式装入这些依赖项。请参见“延迟装入动态依赖项” [97]。
- 如果要生成的共享目标文件依赖于其他共享目标文件，并且这些依赖项并不位于缺省的搜索位置，请使用 `-R` 选项将其路径名记录在输出文件中。请参见“具有依赖项的共享目标文件” [126]。
- 如果没有针对此目标文件或其依赖项使用插入符号，请使用 `-B direct` 建立直接绑定信息。请参见第 6 章 直接绑定。

以下示例结合了以上几点。

```
$ cc -c -o foo.o -K pic -xregs=no%appl foo.c
$ cc -M mapfile -G -o libfoo.so.1 -z text -z defs -B direct -z lazyload \
    -z discard-unused=sections -R /home/lib foo.o -L. -lbar -lc
```

- 如果要生成的共享目标文件用作其他链接编辑的输入，请使用 `-h` 选项在其中记录共享目标文件的运行时名称。请参见“记录共享目标文件名称” [124]。
- 请通过创建指向非版本化共享目标文件名称的文件系统链接，使共享目标文件可用于编译环境中。请参见“协调版本化文件名” [224]。

以下示例结合了以上几点。

```
$ cc -M mapfile -G -o libfoo.so.1 -z text -z defs -B direct -z lazyload \
    -z discard-unused=sections -R /home/lib -h libfoo.so.1 foo.o -L. -lbar -lc
$ ln -s libfoo.so.1 libfoo.so
```

- 请考虑共享目标文件的性能含义：最大化共享性，如“最大化可共享性” [167]中所述；最小化分页活动，如“最小化分页活动” [168]中所述；减少重定位开销，尤其是通过最大程度上减少符号重定位的次数，如“缩减符号作用域” [46]中所述；允许通过功能接口访问数据，如“复制重定位” [170]中所述。

创建动态可执行文件

- 要创建动态可执行文件，请勿使用 -G 或 -d n 选项。
- 建议使用链接编辑器的 -z guidance 选项。指导消息提供有关链接编辑器选项和其他操作的建议，以帮助改善生成的目标文件。
- 表明应使用 -z lazyload 选项延迟装入动态可执行文件的依赖项。请参见"延迟装入动态依赖项" [97]。
- 避免包含不需要的依赖项。请使用带有 -u 选项的 ldd 来检测并删除不需要的依赖项。请参见"共享目标文件处理" [28]。或者，使用 -z discard-unused=dependencies 选项，此选项指示链接编辑器将依赖项仅记录到所引用的目标文件中。
- 如果动态可执行文件的依赖项不位于缺省的搜索位置，请使用 -R 选项将其路径名记录在输出文件中。请参见"运行时链接程序搜索的目录" [32]。
- 使用 -B direct 建立直接绑定信息。请参见第 6 章 直接绑定。

以下示例结合了以上几点。

```
$ cc -o prog -R /home/lib -z discard-unused=dependencies -z lazyload -B direct -L. \
    -lfoo file1.o file2.o file3.o ....
```

部分 III

高级主题

♦ ♦ ♦ 第 6 章

6 直接绑定

在基于动态可执行文件和许多依赖项构造进程的过程中，运行时链接程序必须将符号引用绑定到符号定义。缺省情况下，符号定义是通过简单搜索模型搜索的。通常，将搜索每个目标文件（从动态可执行文件开始），并按装入目标文件的顺序处理每个依赖项。自首次引入动态链接后一直在使用此模型。此简单模型通常会导致所有的符号引用被绑定到一个定义。绑定的定义是在已装入的依赖项系列中找到的第一个定义。

动态可执行文件已发展为更复杂的进程，其复杂程度远高于在动态链接尚处于起步阶段时所开发的可执行文件。依赖项的数目已从几十增长到了几百。动态目标文件之间引用的符号接口的数目也有了大幅增长。符号名称的大小也随着用来支持语言（例如 C++）的技术（例如名称改编）有了显著增长。由于符号引用被绑定到符号定义，因此对于许多应用程序来说，这些因素导致启动时间增加。

进程内符号数目的增加还导致了名称空间污染的增加。多个符号实例具有相同的名称正变得越来越常见。由具有相同符号的多个实例产生的意外的错误绑定经常导致难以对进程故障进行诊断。

此外，现有的进程的各个目标文件需要绑定到多次定义的同名符号的不同实例。

为解决缺省搜索模型的开销问题，同时提供更大的符号绑定灵活性，已创建了一个替代的符号搜索模型。该模型称为直接绑定。

使用直接绑定，可以在进程的目标文件之间建立准确的绑定关系。通过杜绝关联目标文件的意外绑定，直接绑定关系可以帮助避免任何意外的名称空间冲突。这一保护机制增加了进程内目标文件的可靠性，有助于避免意外的难以诊断的绑定情况。

直接绑定可以影响插入。使用直接绑定可以避免意外插入。不过，直接绑定可以禁用有意插入。

本章介绍了直接绑定模型，并讨论了转换目标文件以使用该模型时应考虑的插入问题。

观察符号绑定

为了解缺省的符号搜索模型并将其与直接绑定进行比较，我们使用以下组件来生成一个进程。

```
$ cat main.c
```

```
extern int W(), X();

int main() { return (W() + X()); }
$ cat W.c
extern int b();

int a() { return (1); }
int W() { return (a() - b()); }
$ cat w.c
int b() { return (2); }
$ cat X.c
extern int b();

int a() { return (3); }
int X() { return (a() - b()); }
$ cat x.c
int b() { return (4); }
$ cc -o w.so.1 -G -Kpic w.c
$ cc -o W.so.1 -G -Kpic W.c -R. w.so.1
$ cc -o x.so.1 -G -Kpic x.c
$ cc -o X.so.1 -G -Kpic X.c -R. x.so.1
$ cc -o prog1 -R. main.c W.so.1 X.so.1
```

应用程序的组件按以下顺序装入。

```
$ ldd prog1
        W.so.1 =>        ./W.so.1
        X.so.1 =>        ./X.so.1
        w.so.1 =>        ./w.so.1
        x.so.1 =>        ./x.so.1
```

W.so.1 和 X.so.1 文件都定义了一个名为 a（） 的函数。w.so.1 和 x.so.1 文件都定义了一个名为 b（） 的函数。此外，W.so.1 和 X.so.1 文件都引用了函数 a（） 和 b（） 。

通过设置 LD_DEBUG 环境变量，可以观察使用缺省搜索模型及最终绑定的运行时符号搜索。从运行时链接程序诊断信息中，可以发现到函数 a（） 和 b（） 的绑定。

```
$ LD_DEBUG=symbols,bindings prog1
....
17375: symbol=a;  lookup in file=prog1  [ ELF ]
17375: symbol=a;  lookup in file=./W.so.1  [ ELF ]
17375: binding file=./W.so.1 to file=./W.so.1: symbol 'a'
....
17375: symbol=b;  lookup in file=prog1  [ ELF ]
17375: symbol=b;  lookup in file=./W.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./X.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./w.so.1  [ ELF ]
17375: binding file=./W.so.1 to file=./w.so.1: symbol 'b'
....
17375: symbol=a;  lookup in file=prog1  [ ELF ]
17375: symbol=a;  lookup in file=./W.so.1  [ ELF ]
17375: binding file=./X.so.1 to file=./W.so.1: symbol 'a'
....
```

```
17375: symbol=b;  lookup in file=prog1  [ ELF ]
17375: symbol=b;  lookup in file=./W.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./X.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./w.so.1  [ ELF ]
17375: binding file=./X.so.1 to file=./w.so.1: symbol 'b'
```

对函数 a（） 或 b（） 其中之一的每个引用都会导致从应用程序 prog1 开始搜索关联的符号。对 a（） 的每个引用都绑定到在 W.so.1 中发现的符号的第一个实例。对 b（） 的每个引用都绑定到在 w.so.1 中发现的符号的第一个实例。此示例揭示了 W.so.1 和 w.so.1 中的函数定义如何插入到 X.so.1 和 x.so.1 中的函数定义。使用直接绑定时会发生插入，这是使用直接绑定时需要考虑的一个重要因素。下面的几节中详细介绍了插入。

此示例很简洁，很容易遵循关联的诊断信息。然而，大多数应用程序是基于许多动态组件构造的，要复杂得多。这些组件是从不同的源根基生成的，通常异步提交。

对来自复杂进程的诊断信息进行分析可能比较有难度。分析动态目标文件的接口需求的另一种方法是使用 lari(1) 实用程序。lari 分析进程的绑定信息以及每个目标文件提供的接口定义。该信息使得 lari 能够简明地转换关于进程符号依赖项的受关注信息。在分析伴随着直接绑定发生的插入时，该信息非常有用。

缺省情况下，lari 将转换它认为受关注的信息。该信息源自一个符号定义的多个实例。lari 为 prog1 显示以下信息。

```
$ lari prog1
[2:2ES]: a(): ./W.so.1
[2:0]: a(): ./X.so.1
[2:2E]: b(): ./w.so.1
[2:0]: b(): ./x.so.1
```

在此示例中，从 prog1 建立的进程包含两个多次定义的符号：a（） 和 b（） 。输出诊断信息中的初始元素（括在方括号中的那些元素）描述了关联的符号。

第一个十进制值表示关联符号的实例数。a（） 和 b（） 都存在两个实例。第二个十进制值表示已解析为此符号的绑定数。来自 W.so.1 的符号定义 a（） 揭示已建立了到此依赖项的两个绑定。同样，来自 w.so.1 的符号定义 b（） 揭示已建立了到此依赖项的两个绑定。绑定数之后的字母用于限定绑定。字母 "E" 表示已从外部 (external) 目标文件建立了绑定。字母 "S" 表示已从同一 (same) 目标文件建立了绑定。

在接下来的几节中，将使用 LD_DEBUG、lari 以及从这些组件生成的进程示例来进一步研究直接绑定情况。

启用直接绑定

使用直接绑定的目标文件维护符号引用与提供定义的依赖项之间的关系。运行时链接程序使用该信息直接在关联目标文件中搜索符号，而不执行缺省符号搜索模型。

动态目标文件的直接绑定信息是在链接编辑时记录的。只能为通过该目标文件的链接编辑指定的依赖项建立该信息。可使用 -z defs 选项来确保所有必需的依赖项都提供为链接编辑的一部分。

使用直接绑定的目标文件可以与不使用直接绑定的目标文件共存于同一进程中。那些不使用直接绑定的目标文件将使用缺省的符号搜索模型。

可以使用下列链接编辑机制之一来建立符号引用到符号定义的直接绑定。

- 使用 -B direct 选项。此选项可在要生成的目标文件与所有目标文件依赖项之间建立直接绑定。此选项还可在要生成的目标文件中的任何符号引用与符号定义之间建立直接绑定。

  使用 -B direct 选项还会启用延迟装载。该启用行为等效于在链接编辑命令行的前面添加 -z lazyload 选项。“延迟装入动态依赖项” [97]中介绍了该属性。
- 使用 -z direct 选项。此选项可在要生成的目标文件与命令行中该选项之后的任何依赖项之间建立直接绑定。可将此选项与 -z nodirect 选项配合使用，以切换依赖项之间直接绑定的使用。此选项不在要生成的目标文件中的任何符号引用与符号定义之间建立直接绑定。
- 使用 DIRECT mapfile 关键字。此关键字用于直接绑定各个符号。“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]中描述了此关键字。

注 - 通过将环境变量 LD_NODIRECT 设置为非空值，可在运行时禁用直接绑定。通过设置该环境变量，进程内的所有符号绑定都将通过缺省搜索模型来执行。

以下几节介绍了每种直接绑定机制的使用方法。

使用 -B direct 选项

-B direct 选项提供了为任何动态目标文件启用直接绑定的最简单机制。此选项可在要生成的目标文件内建立到任何依赖项的直接绑定。

从前面的示例中使用的组件，可以生成一个直接绑定的目标文件 W.so.2。

```
$ cc -o W.so.2 -G -Kpic W.c -R. -Bdirect w.so.1
$ cc -o prog2 -R. main.c W.so.2 X.so.1
```

直接绑定信息是在 W.so.2 中的符号信息部分 .SUNW_syminfo 中维护的。可以使用 elfdump(1) 查看该部分。

```
$ elfdump -y W.so.2
     [6]  DB          <self>         a
     [7]  DBL     [1] w.so.1         b
```

字母 "DB" 表示已经为关联符号记录了直接绑定 (direct binding)。函数 a () 已绑定到包含目标文件 W.so.2。函数 b () 已直接绑定到依赖项 w.so.1。字母 "L" 表示依赖项 w.so.1 也应当延迟装载。

可以使用 LD_DEBUG 环境变量观察为 W.so.2 建立的直接绑定。detail 标记用于向绑定诊断信息添加额外的信息。对于 W.so.2，该标记指明绑定是直接绑定这一特性。detail 标记还提供有关绑定地址的额外信息。为简化起见，在以下示例生成的输出中已省略了该地址信息。

```
$ LD_DEBUG=symbols,bindings,detail  prog2
....
18452: symbol=a;  lookup in file=./W.so.2  [ ELF ]
18452: binding file=./W.so.2 to file=./W.so.2: symbol 'a'  (direct)
18452: symbol=b;  lookup in file=./w.so.1  [ ELF ]
18452: binding file=./W.so.2 to file=./w.so.1: symbol 'b'  (direct)
```

lari(1) 实用程序也可以揭示直接绑定信息。

```
$ lari prog2
[2:2ESD]: a(): ./W.so.2
[2:0]: a(): ./X.so.1
[2:2ED]: b(): ./w.so.1
[2:0]: b(): ./x.so.1
```

字母 "D" 表示由 W.so.2 定义的函数 a（） 已被直接绑定。同样，在 w.so.1 中定义的函数 b（） 也已被直接绑定。

注 - 对于函数a（） ，W.so.2 到 W.so.2 的直接绑定产生的效果与使用 -B symbolic 选项生成 W.so.2 时产生的效果类似。不过，-B symbolic 选项能够在链接编辑时完成可以内部解析的引用（例如 a（）） 。该符号解析不会将任何绑定留到运行时进行解析。

与 -B symbolic 绑定不同，-B direct 绑定留到运行时进行解析。因此，可以使用显式插入覆盖该绑定，或者通过将环境变量 LD_NODIRECT 的值设置为一个非空值来禁用该绑定。

符号绑定通常用来降低在装入复杂目标文件时产生的运行时重定位开销。直接绑定可用来建立完全相同的符号绑定。不过，在创建每个直接绑定时仍然需要进行运行时重定向。直接绑定需要的开销比符号绑定大，但能够提供更强的灵活性。

使用 -z direct 选项

-z direct 选项提供的机制用来建立与链接编辑命令行中该选项后的所有依赖项的直接绑定。与 -B direct 选项不同，这不会在将要生成的目标文件内建立直接绑定。

此选项适用于生成作为插入基础的目标文件。例如，共享目标文件有时会设计包含许多缺省或回退接口。应用程序可以本着在运行时将应用程序定义绑定到这些接口的目的，自由地定义这些接口的定义。要允许应用程序插入到共享目标文件的接口，请使用 -z direct 选项而不是 -B direct 选项来生成共享目标文件。

如果您希望有选择地直接绑定到一个或多个依赖项，-z direct 选项也比较有用。-z nodirect 选项用于在通过链接编辑提供的依赖项之间切换直接绑定的使用。

从之前示例中使用的组件，可以生成一个直接绑定的目标文件 X.so.2。

```
$ cc -o X.so.2 -G -Kpic X.c -R. -zdirect x.so.1
$ cc -o prog3 -R. main.c W.so.2 X.so.2
```

可以通过 elfdump(1) 查看直接绑定信息。

```
$ elfdump -y X.so.2
     [6]  D           <self>        a
     [7]  DB      [1] x.so.1        b
```

函数 b () 已直接绑定到依赖项 x.so.1。函数 a () 被定义为与目标文件 X.so.2 之间有一个可能的直接绑定 "D"，但没有建立直接绑定。

可以使用 LD_DEBUG 环境变量来观察运行时绑定。

```
$ LD_DEBUG=symbols,bindings,detail  prog3
....
06177: symbol=a;  lookup in file=prog3  [ ELF ]
06177: symbol=a;  lookup in file=./W.so.2  [ ELF ]
06177: binding file=./X.so.2 to file=./W.so.2: symbol 'a'
06177: symbol=b;  lookup in file=./x.so.1  [ ELF ]
06177: binding file=./X.so.2 to file=./x.so.1: symbol 'b'  (direct)
```

lari(1) 实用程序也可以揭示直接绑定信息。

```
$ lari prog3
[2:2ESD]: a(): ./W.so.2
[2:0]: a(): ./X.so.2
[2:1ED]: b(): ./w.so.1
[2:1ED]: b(): ./x.so.1
```

由 W.so.2 定义的函数 a () 继续服务于 X.so.2 执行的缺省符号引用。但是，在 x.so.1 中定义的函数 b () 已从 X.so.2 执行的引用直接绑定。

使用 DIRECT mapfile 关键字

DIRECT mapfile 关键字提供了一种为各个符号建立直接绑定的方法。该机制用于特定的链接编辑情况。

在之前示例使用的组件中，main () 函数引用了外部函数 W () 和 X () 。这些函数的绑定遵循缺省的搜索模型。

```
$ LD_DEBUG=symbols,bindings  prog3
....
18754: symbol=W;  lookup in file=prog3  [ ELF ]
18754: symbol=W;  lookup in file=./W.so.2  [ ELF ]
18754: binding file=prog3 to file=./W.so.2: symbol 'W'
....
18754: symbol=X;  lookup in file=prog3  [ ELF ]
```

```
18754: symbol=X;  lookup in file=./W.so.2  [ ELF ]
18754: symbol=X;  lookup in file=./X.so.2  [ ELF ]
18754: binding file=prog3 to file=./X.so.2: symbol 'X'
```

可以使用 DIRECT mapfile 关键字来重新生成 prog3，以便建立到函数 W() 和 X() 的直接绑定。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                W       { FLAGS = EXTERN DIRECT };
                X       { FLAGS = EXTERN DIRECT };
};
$ cc -o prog4 -R. main.c W.so.2 X.so.2 -Mmapfile
```

可以使用 LD_DEBUG 环境变量来观察运行时绑定。

```
$ LD_DEBUG=symbols,bindings,detail prog4
....
23432: symbol=W;  lookup in file=./W.so.2  [ ELF ]
23432: binding file=prog4 to file=./W.so.2: symbol 'W'  (direct)
23432: symbol=X;  lookup in file=./X.so.2  [ ELF ]
23432: binding file=prog4 to file=./x.so.2: symbol 'X'  (direct)
```

lari(1) 实用程序也可以揭示直接绑定信息。然而，在本例中没有多次定义 W() 和 X() 函数。因此，缺省情况下，lari 不会发现这些函数受关注。必须使用 -a 选项才能显示所有符号信息。

```
$ lari -a prog4
....
[1:1ED]: W(): ./W.so.2
....
[2:1ED]: X(): ./X.so.2
....
```

注 - 通过使用 -B direct 选项或 -z direct 选项生成 prog4，可以生成到 W.so.2 和 X.so.1 的相同直接绑定。本示例的目的只是为了说明如何使用 mapfile 关键字。

直接绑定和插入

当已装入到进程中的不同动态目标文件中存在同一符号的多个同名实例时可能会发生插入。在缺省搜索模型下，符号引用被绑定到在已装入的依赖项系列中发现的第一个定义。这第一个符号被称为在同名的其他符号上的插入。

直接绑定可以禁用任何隐式插入。因为将在与引用关联的依赖项中搜索直接绑定的引用，所以会跳过启用了插入的缺省符号搜索模型。在直接绑定环境中，可以建立到具有相同名称的同一符号的不同定义的绑定。

能够绑定到具有相同名称的同一符号的不同定义是直接绑定的一个非常有用的功能。然而，如果某个应用程序依赖于一个插入实例，则使用直接绑定可能会破坏应用程序的预期执行。在决定为现有应用程序使用直接绑定之前，应当对应用程序进行分析以确定是否存在插入。

要确定应用程序内是否可能存在插入，请使用 lari(1)。缺省情况下，lari 输出受关注的信息。该信息源自同一符号定义的多个实例，这些实例可能会依次导致插入。

只有在绑定到符号的一个实例时才会发生插入。插入中可能不会涉及 lari 收集的同一符号的多个实例。其他多个实例符号可以存在，但可能不会被引用。这些未引用的符号仍然用作插入的候选者，因为将来的代码开发可能会引用这些符号。在考虑使用直接绑定时，应当对多次定义的符号的所有实例进行分析。

如果存在同一符号的多个同名实例，特别是已观察到插入时，应当执行下列操作之一。

- 对符号实例进行本地化以消除名称空间冲突。
- 删除多个实例以便只留下一个符号定义。
- 显式定义任何插入需求。
- 识别可以作为插入基础的符号以阻止直接绑定到该符号。

以下几节更详细地讨论了这些操作。

本地化符号实例

应当对提供不同实现的已多次定义的同名符号进行隔离以避免意外的插入。从目标文件导出的接口中删除符号的最简单方法是将符号降级为局部符号。可以通过将符号定义为 "static" 或使用编译器提供的符号属性将符号降级为局部符号。

还可以通过使用链接编辑器和 mapfile 将符号降级为局部符号。以下示例显示了一个 mapfile，它通过使用 local 作用域指令将全局函数 error（） 降级为一个局部符号。

```
$ cc -o A.so.1 -G -Kpic error.c a.c b.c ....
$ elfdump -sN.symtab A.so.1 | fgrep error
    [36]     0x2d0   0x14  FUNC GLOB  D    0 .text      error
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        local:
                error;
};
$ cc -o A.so.2 -G -Kpic -M mapfile error.c a.c b.c ....
$ elfdump -sN.symtab A.so.2 | fgrep error
    [24]     0x2c8   0x14  FUNC LOCL  H    0 .text      error
```

虽然可以通过使用显式的 mapfile 定义将各个符号降级为局部符号，但建议通过符号版本控制来定义整个接口系列。请参见第 9 章 接口和版本控制。

版本控制是一项通常用于标识从共享目标文件导出的接口的有用技术。类似地，可以对动态可执行文件进行版本控制以定义其导出的接口。动态可执行文件只需要导出必须可供要绑定到的目标文件的依赖项使用的接口。通常，您添加到动态可执行文件的代码不需要导出接口。

从动态可执行文件删除导出的接口时应考虑已由编译器驱动程序建立的任何符号定义。这些定义源自编译器驱动程序添加到最终链接编辑的辅助文件。请参见“使用编译器驱动程序” [24]。

以下示例 `mapfile` 导出了编译器驱动程序可以建立的一组常见的符号定义，同时将所有其他全局定义降级为局部定义。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                __Argv;
                __environ_lock;
                _environ;
                _lib_version;
                environ;
        local:
                *;
};
```

您应当决定您的编译器驱动程序建立的符号定义。在动态可执行文件内使用的这些定义中的任何一个都应保留为全局定义。

通过从动态可执行文件删除任何导出的接口，可以防止可执行文件出现比目标文件依赖项进化时更多的插入问题。

删除多次定义的同名符号

如果由与符号关联的实现维护自身状态，则在直接绑定环境内多次定义的同名符号可能会出现问题。从这一点来说，数据符号通常是违规者，然而维护自身状态的函数也可能会出现问题。

在直接绑定环境中，可以绑定到同一符号的多个实例。因此，不同的绑定实例可以在一个进程内操纵原本打算成为单个实例的不同状态变量。

例如，假设两个共享目标文件包含相同的数据项 `errval`。另外，假设两个函数 `action（）` 和 `inspect（）` 存在于不同的共享目标文件中。这些函数预期分别读取和写入 `errval` 值。

使用缺省搜索模型时，`errval` 的一个定义将插入到另一个定义上。函数 `action（）` 和 `inspect（）` 都将绑定到 `errval` 的同一实例。因此，如果 `action（）` 向 `errval` 写入了一个错误代码，则 `inspect（）` 可以读取和处理该错误状态。

但是，假设包含 action() 和 inspect() 的目标文件分别被绑定到了各自定义了 errval 的不同依赖项。在直接绑定环境内，这些函数将被绑定到 errval 的不同定义。action() 可以向 errval 的一个实例写入错误代码，而 inspect() 读取 errval 的另一个未初始化的定义。结果是，inspect() 未检测到要处理的错误状态。

如果符号是在标头中声明的，则通常会出现数据符号的多个实例。

```
int bar;
```

该数据声明会导致包含该标头的每个编译单元生成一个数据项。此暂定结果数据项会导致在不同的动态目标文件中定义符号的多个实例。

但是，通过将数据项显式定义为外部数据项，可以为包含该标头的每个编译单元生成对数据项的引用。

```
extern int bar;
```

然后，在运行时可以将这些引用解析为一个数据实例。

有时候，应当保留您要删除的符号实现的接口。同一接口的多个实例可以向量化到一个实现，同时保留任何现有接口。通过使用 FILTER mapfile 关键字创建单个符号过滤器，可以实现此模型。“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]中描述了此关键字。

当依赖项期望在已删除了某个符号实现的目标文件中找到该符号时，创建单个符号过滤器很有用。

例如，假设函数 error() 存在于两个共享目标文件 A.so.1 和 B.so.1 中。为消除符号重复，您希望从 A.so.1 中删除该实现。不过，其他依赖项正依赖于从 A.so.1 提供的 error()。以下示例显示了 A.so.1 中 error() 的定义。然后，使用一个 mapfile 来允许删除 error() 实现，同时为定向到 B.so.1 的该符号保留一个过滤器。

```
$ cc -o A.so.1 -G -Kpic error.c a.c b.c ....
$ elfdump -sN.dynsym A.so.1 | fgrep error
     [3]     0x300    0x14  FUNC GLOB  D    0 .text      error
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                error { TYPE=FUNCTION; FILTER=B.so.1 };
};
$ cc -o A.so.2 -G -Kpic -M mapfile a.c b.c ....
$ elfdump -sN.dynsym A.so.2 | fgrep error
     [3]        0       0  FUNC GLOB  D    0 ABS        error
$ elfdump -y A.so.2 | fgrep error
     [3]  F       [0] B.so.1         error
```

error() 函数是全局的，并保留 A.so.2 的一个导出接口。但是，到该符号的任何运行时绑定都将指向 *filtee* B.so.1。字母 "F" 表示该符号是过滤器。

该模型在向量化到一个实现时保留现有接口，多个 Oracle Solaris 库中均使用该模型。例如，曾经在 libc.so.1 中定义的许多数学接口现已向量化到 libm.so.2 中函数的首选实现。

定义显式插入

缺省搜索模型可能会导致同名符号的实例插入到同名的后续实例上。即使没有任何显式标签，仍然会发生插入，以便从所有引用绑定到同一个符号定义。发生该隐式插入是符号搜索的结果，而不是因为向运行时链接程序发出了任何显式指令。使用直接绑定可以禁用该隐式插入。

虽然直接绑定能够将符号引用直接解析到关联的符号定义，但是显式插入是在任何直接绑定搜索之前处理的。因此，即使是在直接绑定环境内，也可以对插入项进行设计并预期它在任意直接绑定关联上进行插入。可以使用下列技术显式定义插入项。

- 使用 LD_PRELOAD 环境变量。
- 使用链接编辑器 -z interpose 选项。
- 使用 INTERPOSE mapfile 关键字。
- singleton 符号定义产生的结果。

LD_PRELOAD 环境变量的插入功能和 -z interpose 选项已经使用了一段时间。请参见“运行时插入” [93]。因为这些目标文件是显式定义为插入项的，因此运行时链接程序将在处理任何直接绑定之前检查这些目标文件。

为一个共享目标文件建立的插入将应用于该动态目标文件的所有接口。系统使用 LD_PRELOAD 环境变量装入目标文件时，会建立该目标文件插入。在装入已使用 -z interpose 选项生成的目标文件时，也会建立目标文件插入。当使用具有特殊句柄 RTLD_NEXT 的 dlsym(3C) 等技术时，该目标文件模型很重要。插入目标文件应始终具有下一个目标文件的一致视图。

动态可执行文件具有额外的灵活性，因为该可执行文件可以使用 INTERPOSE mapfile 关键字定义单个插入符号。因为动态可执行文件是进程中装入的第一个目标文件，所以可执行文件的下一个目标文件视图始终是一致的。

以下示例显示了要显式插入到 exit（） 函数的应用程序。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                exit    { FLAGS = INTERPOSE };
};
$ cc -o prog -M mapfile exit.c a.c b.c ....
$ elfdump -y prog | fgrep exit
[6]  DI          <self>         exit
```

字母 "I" 表示该符号是插入符号。实现此 exit（） 函数可以直接引用系统函数 _exit（） ，也可以使用带有 RTLD_NEXT 句柄的 dlsym（） 调用系统函数 exit（） 。

最初，您可能考虑使用 -z interpose 选项识别此目标文件。但是，此技术的开销相当大，因为应用程序导出的所有接口都将用作插入项。较好的替代方法是，结合使用 -z interpose 选项将应用程序提供的所有符号（插入项除外）本地化。

但是，使用 INTERPOSE mapfile 关键字会提供更大的灵活性。通过使用此关键字，应用程序可以导出多个接口，同时选择应当用作插入项的接口。

指定了 STV_SINGLETON 可见性的符号可以有效地提供一种插入形式。请参见表 12-23 “ELF 符号可见性”。编译系统可将这些符号指定给一个实现，该实现可能会在进程内的多个目标文件中多次实例化。所有的单件符号引用都绑定到进程中第一次出现的单件符号。

阻止直接绑定到某个符号

可以使用显式插入来覆盖直接绑定。请参见“定义显式插入” [153]。不过，可能会存在您无法控制显式插入的建立的情况。

例如，您可能交付了您希望使用直接绑定的一系列共享目标文件。用户知道要在该系列的共享目标文件提供的符号上进行插入。如果这些用户没有显式定义他们的插入要求，则重新交付使用直接绑定的共享目标文件可能会破坏它们的插入。

共享目标文件也可设计提供许多缺省接口，并期望用户提供自己的插入例程。

为防止破坏现有的应用程序，可以交付显式阻止直接绑定到一个或多个应用程序接口的共享目标文件。

可以使用下列选项之一阻止直接绑定到动态目标文件。

- 使用 -B nodirect 选项。此选项阻止直接绑定到由要生成的目标文件提供的任何接口。
- 使用 NODIRECT mapfile 关键字。使用此关键字可阻止直接绑定到各个符号。“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]中描述了此关键字。
- singleton 符号定义产生的结果。

不能从外部目标文件直接绑定到标签为 nodirect 的接口。此外，也不能从同一目标文件内直接绑定到标签为 nodirect 的接口。

以下几节介绍了每种直接绑定阻止机制的使用方法。

使用 -B nodirect 选项

-B nodirect 选项提供了最简单的机制，它阻止从任何动态目标文件进行直接绑定。此选项阻止从任何其他目标文件进行直接绑定，并阻止从要生成的目标文件内进行直接绑定。

以下组件用于生成三个共享目标文件：A.so.1、O.so.1 和 X.so.1。-B nodirect 选项用于阻止 A.so.1 直接绑定到 O.so.1。不过，O.so.1 可以使用 -z direct 选项继续建立到 X.so.1 的直接绑定。

```
$ cat a.c
extern int o(), p(), x(), y();

int a() { return (o() + p() - x() - y()); }
$ cat o.c
extern int x(), y();

int o() { return (x()); }
int p() { return (y()); }
$ cat x.c
int x() { return (1); }
int y() { return (2); }
$ cc -o X.so.1 -G -Kpic x.c
$ cc -o O.so.1 -G -Kpic o.c -Bnodirect -zdirect -R. X.so.1
$ cc -o A.so.1 -G -Kpic a.c -Bdirect -R. O.so.1 X.so.1
```

可以使用 elfdump(1) 来查看 A.so.1 和 O.so.1 的符号信息。

```
$ elfdump -y A.so.1
     [1]  DBL     [3] X.so.1           x
     [5]  DBL     [3] X.so.1           y
     [6]  DL      [1] O.so.1           o
     [9]  DL      [1] O.so.1           p
$ elfdump -y O.so.1
     [3]  DB      [0] X.so.1           x
     [4]  DB      [0] X.so.1           y
     [6]  N                            o
     [7]  N                            p
```

字母 "N" 表示不允许直接绑定到函数 o（） 和 p（）。尽管 A.so.1 已使用 -B direct 选项来请求直接绑定，但是没有建立到函数 o（） 和 p（） 的直接绑定。O.so.1 仍然可以使用 -z direct 选项来请求到其依赖项 X.so.1 的直接绑定。

Oracle Solaris 库 libproc.so.1 是使用 -B nodirect 选项生成的。该库的用户应该为许多 libproc 函数提供他们自己的回调接口。从 libproc 的任何依赖项对 libproc 函数的引用都应当绑定到任何用户定义（如果存在这样的定义）。

使用 NODIRECT mapfile 关键字

NODIRECT mapfile 关键字提供了一种阻止直接绑定到各个符号的方法。在阻止直接绑定方面，该关键字提供了比 -B nodirect 选项更细粒度的控制。

从之前示例使用的组件中，可以将 O.so.2 生成为阻止直接绑定到函数 o（）。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                o       { FLAGS = NODIRECT };
};
$ cc -o O.so.2 -G -Kpic o.c -Mmapfile -zdirect -R. X.so.1
$ cc -o A.so.2 -G -Kpic a.c -Bdirect -R. O.so.2 X.so.1
```

可以使用 elfdump(1) 来查看 A.so.2 和 O.so.2 的符号信息。

```
$ elfdump -y A.so.2
     [1]  DBL     [3] X.so.1           x
     [5]  DBL     [3] X.so.1           y
     [6]  DL      [1] O.so.1           o
     [9]  DBL     [1] O.so.1           p
$ elfdump -y O.so.1
     [3]  DB      [0] X.so.1           x
     [4]  DB      [0] X.so.1           y
     [6]  N                            o
     [7]  D           <self>           p
```

O.so.1 只声明了不能直接绑定到函数 o（）。因此，A.so.2 能够直接绑定到 O.so.1 中的函数 p（）。

Oracle Solaris 库内的各个接口已被定义为不允许直接绑定。其中之一是数据项 errno。该数据项是在 libc.so.1 中定义的。可以通过包含头文件 stdio.h 来引用该数据项。但是，通常会指示许多应用程序定义其自己的 errno。如果交付了直接绑定到 libc.so.1 中定义的 errno 的一系列系统库，则这些应用程序会被损害。

已定义为阻止绑定到的另一个接口系列是 malloc(3C) 系列。malloc（） 系列是用户应用程序内经常会实现的另一组接口。这些用户实现的目的是在任何系统定义上进行插入。

注 - 随 Oracle Solaris OS 提供了各种系统插入库，这些库提供了替代的 malloc（） 实现。此外，每个实现都期望成为在一个进程内使用的唯一实现。所有 malloc（） 插入库都是使用 -z interpose 选项生成的。该选项实际上不是必需的，因为 libc.so.1 内的 malloc（） 系列已标记为阻止任何直接绑定。

不过，仍然使用了 -z interpose 选项生成这些插入库是为了给插入项的生成设定一个惯例。该显式插入与在 libc.so.1 中建立的直接绑定阻止定义之间没有任何不利的相互影响。

不能直接绑定到指定了 STV_SINGLETON 可见性的符号。请参见表 12-23 “ELF 符号可见性”。编译系统可将这些符号指定给一个实现，该实现可能会在进程内的多个目标文件中多次实例化。所有的单件符号引用都绑定到进程中第一次出现的单件符号。

♦ ♦ ♦ 第 7 章

7 生成目标文件以优化系统性能

动态可执行文件和共享目标文件要求运行时处理建立这些目标文件所参与的进程。在任意时刻，一个进程可以有多个实例处于活动状态，且共享目标文件可以同时供不同的进程使用。动态目标文件的构造会影响目标文件运行时初始化及其在进程之间的潜在共享，以及总体系统性能。

以下各节分析了动态目标文件的运行时初始化和处理，其中检查了影响动态目标文件运行时性能的因素（如文本大小和纯度以及重定位开销）。

使用 elfdump 分析文件

可以使用各种工具分析 ELF 文件的内容，包括标准的 Unix 实用程序 dump(1)、nm(1) 和 size(1)。在 Oracle Solaris 中，这些工具大都已被 elfdump(1) 取代。

使用 eldump 诊断 ELF 目标文件的内容对于调查以下几节中介绍的各种性能问题很有用。

ELF 格式可以将数据组织到多个节中。各节会被依次指定给称为段的单元。段描述如何将文件的各部分映射到内存。请参见 mmapobj(2)。可以通过使用 elfdump(1) 命令并检查 PT_LOAD 项来显示这些可装入段。

```
$ elfdump -p -NPT_LOAD libfoo.so.1
Program Header[0]:
    p_vaddr:      0           p_flags:    [ PF_X PF_R ]
    p_paddr:      0           p_type:     [ PT_LOAD ]
    p_filesz:     0x53c       p_memsz:    0x53c
    p_offset:     0           p_align:    0x10000

Program Header[1]:
    p_vaddr:      0x1053c     p_flags:    [ PF_X PF_W PF_R ]
    p_paddr:      0           p_type:     [ PT_LOAD ]
    p_filesz:     0x114       p_memsz:    0x13c
    p_offset:     0x53c       p_align:    0x10000
```

在 libfoo.so.1 共享目标文件中存在两种可装入段，通常称为文本段和数据段。对文本段进行了映射以允许读取和执行其内容（PF_X 和 PF_R）。数据段映射的目的是允许同时

修改其内容 PF_W。数据段的内存大小 p_memsz 不同于文件大小 p_filesz。该差异说明存在 .bss 节，此节属于数据段并在装入数据段时动态创建。

程序员通常根据定义其代码中的函数和数据元素的符号来考虑文件。通过在 elfdump 命令中使用 -s 选项可显示这些符号。

```
$ elfdump -sN.symtab libfoo.so.1

Symbol Table Section:  .symtab
   index    value     size  type bind oth ver shndx          name
    ....
    [36]  0x10628     0x28  OBJT GLOB  D    0 .data          data
    ....
    [38]  0x10650     0x28  OBJT GLOB  D    0 .bss           bss
    ....
    [40]    0x520      0xc  FUNC GLOB  D    0 .init          _init
    ....
    [44]    0x508     0x14  FUNC GLOB  D    0 .text          foo
    ....
    [46]    0x52c      0xc  FUNC GLOB  D    0 .fini          _fini
```

elfdump 显示的符号表信息包括与符号关联的节。elfdump -c 选项可用于显示与这些节有关的信息。

```
$ elfdump -c libfoo.so.1
....
Section Header[6]:  sh_name: .text
    sh_addr:      0x4f8           sh_flags:   [ SHF_ALLOC SHF_EXECINSTR ]
    sh_size:      0x28            sh_type:    [ SHT_PROGBITS ]
    sh_offset:    0x4f8           sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x8

Section Header[7]:  sh_name: .init
    sh_addr:      0x520           sh_flags:   [ SHF_ALLOC SHF_EXECINSTR ]
    sh_size:      0xc             sh_type:    [ SHT_PROGBITS ]
    sh_offset:    0x520           sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x4

Section Header[8]:  sh_name: .fini
    sh_addr:      0x52c           sh_flags:   [ SHF_ALLOC SHF_EXECINSTR ]
    sh_size:      0xc             sh_type:    [ SHT_PROGBITS ]
    sh_offset:    0x52c           sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x4
....
Section Header[12]:  sh_name: .data
   sh_addr:      0x10628         sh_flags:   [ SHF_WRITE SHF_ALLOC ]
   sh_size:      0x28            sh_type:    [ SHT_PROGBITS ]
   sh_offset:    0x628           sh_entsize: 0
   sh_link:      0               sh_info:    0
   sh_addralign: 0x4
....
```

```
Section Header[14]:  sh_name: .bss
   sh_addr:      0x10650         sh_flags:   [ SHF_WRITE SHF_ALLOC ]
   sh_size:      0x28            sh_type:    [ SHT_NOBITS ]
   sh_offset:    0x650           sh_entsize: 0
   sh_link:      0               sh_info:    0
   sh_addralign: 0x4
....
```

以上示例中 elfdump(1) 的输出显示了 _init、foo 和 _fini 函数与 .init、.text 和 .fini 节的关联关系。这些节由于具有只读性质，因此属于文本段。

同样，数据数组 data 和 bss 分别与节 .data 和 .bss 关联。这些节由于具有可写性质，因此属于数据段。

底层系统

应用程序是从动态可执行文件和一个或多个共享目标文件依赖项中生成的。动态可执行文件和共享目标文件的整个可装入内容在运行时映射到此进程的虚拟地址空间。每个进程通过引用内存中动态可执行文件和共享目标文件的单个副本进行启动。

处理动态目标文件中的重定位以将符号引用绑定到相应的定义。这会导致计算那些无法在链接编辑器生成目标文件时得到的实际虚拟地址。通常，这些重定位会导致更新进程数据段中的项。

基于动态目标文件链接的内存管理方案将按照页粒度在各进程之间共享内存。只要在运行时未修改内存页，便可在进程间共享这些内存页。如果某个进程在写入数据项或在重定位对共享目标文件的引用时写入到一个共享目标文件页，则会生成此页的专用副本。此专用副本不会影响此目标文件的其他用户。但是，其他进程无法共享此页。通过此方式修改的文本页称为不纯文本页。

映射到内存的动态目标文件中的段分为两种基本类别：只读的文本段和可读写的数据段。有关如何从 “使用 elfdump 分析文件” [159] 文件获取信息，请参见Analyzing Files With elfdump。开发动态目标文件时的最重要目标是最大化文本段以及最小化数据段。该分区可优化代码共享量，同时减少初始化和使用动态目标文件所需的处理量。本节介绍有助于实现此目标的机制。

延迟装入动态依赖项

通过将共享目标文件建立为延迟可装入目标文件，可以延迟装入该共享目标文件依赖项，直到首次引用依赖项。请参见“延迟装入动态依赖项” [97]。

对于小型应用程序，典型的执行线程可以引用所有的应用程序依赖项。应用程序将装入所有的依赖项，而无论是否将这些依赖项定义为延迟可装入依赖项。但是，使用延迟装入，会使依赖项处理从进程启动一直延迟到整个进程执行过程。

对于具有许多依赖项的应用程序，延迟装入通常会导致根本没有装入某些依赖项。仅装入针对特定执行线程引用的依赖项。

与位置无关的代码

动态可执行文件中的代码通常是位置相关的，并且与内存中的某个固定地址相关联。相反，共享目标文件可装入不同进程中的不同地址。位置无关代码不与特定地址关联。这种无关性允许在每个使用此类代码的进程中的不同地址有效地执行代码。建议在创建共享目标文件时使用与位置无关的代码。

使用 `-K pic` 选项，编译器可以生成与位置无关的代码。

如果共享目标文件根据位置相关代码生成，则在运行时可能需要修改文本段。通过此修改，可以为已装入目标文件的位置指定可重定位引用。本段的重定位需要将此段重映射为可写段。这种修改需要预留交换空间，并且会形成此进程的文本段专用副本。此文本段不再供多个进程共享。通常，位置相关代码比相应的与位置无关的代码需要更多的运行时重定位。总体而言，处理文本重定位的开销可能会严重降低性能。

根据与位置无关的代码生成共享目标文件时，会通过共享目标文件数据段中的数据间接生成可重定位引用。文本段中的代码不需要进行任何修改。所有重定位更新都会应用于数据段中的相应项。有关特定间接技术的更多详细信息，请参见“全局偏移表（特定于处理器）” [371]和“过程链接表（特定于处理器）” [372]。

如果存在文本重定位，运行时链接程序便会尝试处理这些重定位。但是，某些重定位无法在运行时实现。

x64 位置相关代码序列可生成的代码只能装入内存的低 32 位。任何地址的高 32 位必须全部为零。由于共享目标文件通常装入内存高位，因此需要地址的高 32 位。这样，x64 共享目标文件中位置相关代码便无法满足重定位要求。在共享目标文件中使用此类代码会导致出现运行时重定位错误。

```
$ prog
ld.so.1: prog: fatal: relocation error: R_AMD64_32: file \
    libfoo.so.1: symbol (unknown): value 0xfffffd7fff0cd457 does not fit
```

与位置无关的代码可以装入内存中的任何区域，从而可以满足 x64 共享目标文件的要求。

这种情况不同于用于 64 位 SPARCV9 代码的缺省 `ABS64` 模式。这种位置相关代码通常兼容整个 64 位地址范围。因此，位置相关代码序列可以存在于 SPARCV9 共享目标文件中。针对 64 位 SPARCV9 代码使用 `ABS32` 模式或 `ABS44` 模式仍会导致无法在运行时解析的重定位。但是，这两种模式都需要运行时链接程序对文本段进行重定位。

无论运行时链接程序功能如何，也无论重定位要求的差异如何，共享目标文件都应该使用与位置无关的代码生成。

可以根据文本段确定需要重定位的共享目标文件。以下示例使用 elfdump(1) 确定是否存在 TEXTREL 项动态项。

```
$ cc -o libfoo.so.1 -G -R. foo.c
$ elfdump -d libfoo.so.1 | grep TEXTREL
    [9]  TEXTREL       0
```

注 - TEXTREL 项的值无关紧要。共享目标文件中存在此项表示存在文本重定位。

要防止创建包含文本重定位的共享目标文件，可使用链接编辑器的 -z text 标志。此标志会导致链接编辑器生成指示将位置相关代码源用作输入的诊断。以下示例显示位置相关代码如何导致无法生成共享目标文件。

```
$ cc -o libfoo.so.1 -z text -G -R. foo.c
Text relocation remains                         referenced
  against symbol                offset      in file
foo                               0x0         foo.o
bar                               0x8         foo.o
ld: fatal: relocations remain against allocatable but \
non-writable sections
```

因为通过 foo.o 文件生成了位置相关代码，因此将根据文本段生成两个重定位。如有可能，这些诊断会指明执行重定位所需的任何符号引用。在这种情况下，将根据符号 foo 和 bar 进行重定位。

如果包括手写汇编程序代码，但不包括相应的位置无关原型，则在共享目标文件中也会出现文本重定位。

注 - 可能需要使用一些简单的源文件进行实验，以确定启用位置无关性的编码序列。请使用编译器功能来生成中间汇编程序输出。

-K pic 和 -K PIC 选项

对于 SPARC 二进制文件，-K pic 选项与备用 -K PIC 选项之间的细微差异会影响对全局偏移表项的引用。请参见“全局偏移表（特定于处理器）” [371]。

全局偏移表是一个指针数组，对于 32 位（4 个字节）和 64 位（8 个字节）目标文件，其项大小为常量。以下代码序列可引用 -K pic 下的某个项。

```
        ld    [%l7 + j], %o0    ! load &j into %o0
```

其中，%l7 是执行引用的目标文件的 _GLOBAL_OFFSET_TABLE_ 符号的预计算值。

此代码序列为全局偏移表项提供了 13 位位移常量。因此，此位移为 32 位目标文件提供了 2048 个唯一项，为 64 位目标文件提供了 1024 个唯一项。如果创建目标文件需要的项数多于可用项数，则链接编辑器会生成一个致命错误。

```
$ cc -K pic -G -o lobfoo.so.1 a.o b.o .... z.o
ld: fatal: too many symbols require 'small' PIC references: \
    have 2050, maximum 2048 -- recompile some modules -K PIC.
```

要克服这种错误情况，可使用 -K PIC 选项编译某些输入可重定位目标文件。此选项为全局偏移表项提供 32 位常量。

```
sethi %hi(j), %g1
or    %g1, %lo(j), %g1    ! get 32–bit constant GOT offset
ld    [%l7 + %g1], %o0    ! load &j into %o0
```

可以使用带 -G 选项的 elfdump(1) 查看目标文件的全局偏移表要求。还可以使用链接编辑器调试标记 -D got,detail 在链接编辑过程中检查这些项的处理。

理论上，使用 -K pic 模型对经常访问的数据项有益。可以使用这两种模型引用单个项。但是，确定哪些可重定位目标文件应该使用其中一个选项进行编译可能会相当耗时，并且不会显著改善性能。通常，使用 -K PIC 选项可轻松重新编译所有可重定位目标文件。

删除未使用的材料

当要生成的目标文件并不使用输入可重定位目标文件中的函数和数据时，将这些函数和数据包括在内只是一种浪费。这种不需要的材料会使目标文件过大，在运行时使用该目标文件时会导致额外开销。

引用未使用的共享目标文件依赖项也是一种浪费。特别是不存在延迟装入时，这些引用会导致在运行时不必要地装入和处理这些共享目标文件。

在链接编辑期间可以使用链接编辑器的调试选项 -D unused 诊断未使用的节、未使用的可重定位目标文件和未使用的共享目标文件依赖项。

使用 -z guidance 选项时还会诊断未使用的文件和依赖项。

应该从链接编辑中删除未使用的节、未使用的文件和未使用的依赖项。此删除可减少链接编辑的成本，减少使用要生成的目标文件的运行时成本。但是，如果删除这些项会产生问题，则可以使用 -z discard-unused 选项从要生成的目标文件中丢弃未使用的材料。

删除未使用的节

当三个条件为真时，确定不使用输入可重定位目标文件中的 ELF 节。

- 节不提供任何全局符号。
- 该节是某个可分配段的构成部分。

- 参与链接编辑的任何目标文件中的任何其他已使用的节均不引用该节。

可以使用 -z discard-unused=sections 选项从链接编辑中丢弃未使用的节。

通过定义动态目标文件的外部接口可以改进链接编辑器诊断和丢弃节的功能。请参见第 9 章 接口和版本控制。通过定义接口，可以将未定义为此接口一部分的全局符号降级为局部符号。此时将未从其他目标文件引用的降级后符号明确标识为供丢弃的目标文件。

如果将单个函数和数据变量指定给其自己的节，则使用链接编辑器可以丢弃这些项。可以使用 -xF 编译器选项完善此节。

删除未使用的文件

如果由可重定位目标文件提供的所有可分配节都未使用，则会将输入可重定位目标文件判定为未使用的。

可以通过 -z guidance 选项诊断未使用的文件，也可以使用 -z discard-unused=files 选项从链接编辑中丢弃未使用的文件。

-z discard-unused 选项提供了对未使用的节和未使用的文件的独立控制以便支持 -z guidance 处理。通过 -z guidance，可以识别哪些文件被判定为未使用的。通常可以很容易地从链接编辑中删除未使用的文件。不过，通过 -z guidance 处理无法识别哪些节被判定为未使用的。未使用的节可能涉及更多分析和删除工作，可能会导致编译器操作超出您的控制。

组合使用 -z discard-unused=sections 选项和 -z guidance 选项，可自动删除未使用的节，同时为您识别未使用的文件以便从链接编辑中删除。

删除未使用的依赖项

显式的共享目标文件依赖项是指使用路径名或使用 -l 选项（更常用）在命令行定义的依赖项。显式依赖项包含可能由编译器驱动程序提供的依赖项，如 -lc。

隐式依赖项是显式依赖项的依赖项。可以将隐式依赖项处理为链接编辑的一部分，以完成所有符号解析的关闭。此符号关闭确保要生成的目标文件是自包含的，不保留未引用的符号。

所有动态目标文件都应定义其需要的依赖项。在生成动态可执行文件时，缺省情况下将实施此要求，而生成共享目标文件时则不会缺省实施。使用 -z defs 选项可在生成共享目标文件时实施此要求。

所有动态目标文件都应避免定义它们不需要的依赖项。在运行时装入此类未使用的依赖项是一种无谓和浪费的操作。

如果两个条件为真，则会判定显式依赖项是未使用的。

- 不会从要生成的目标文件中引用依赖项提供的全局符号。
- 该依赖项不会针对任何隐式依赖项的要求进行补偿。

可使用 -z guidance 选项诊断未使用的依赖项。应从链接编辑中删除这些依赖项。但是，如果删除这些项会产生问题，则可以使用 -z discard-unused=dependencies 选项从要生成的目标文件中丢弃未使用的依赖项。

遗憾的是，有些共享目标文件并未定义所有必要的依赖项。在这种情况下，开发者通常会将缺失的依赖项添加到可执行文件或他们正在生成的其他共享目标文件中，而非重新生成正确的原始依赖项。此类依赖项称为补偿依赖项。

例如，假设有一个共享目标文件 foo.so，它引用共享目标文件 bar.so 中的符号 bar（）。但是，foo.so 并未表现出对 bar.so 的依赖性。对 foo.so 进行检查后发现缺少必要的依赖项，因为找不到 bar（） 符号。

```
% ldd -r foo.so
     libc.so.1 =>     /lib/libc.so.1
     symbol not found: bar           (foo.so)
```

现在假设某应用程序开发者希望创建一个引用共享目标文件 foo.so 中符号 foo（） 的可执行文件。已指定所需的对 foo.so 的依赖性，但可执行文件的链接编辑失败。

```
% cc -Bdirect -o main main.c -L. -lfoo
Undefined                       first referenced
  symbol                             in file
  bar                                 ./libfoo.so
ld: fatal: symbol referencing errors
```

开发者通过添加对 bar.so 的补偿依赖项来强制更正此情况。

```
% cc -Bdirect -o main main.c -L. -lfoo -lbar
```

此次更正创建了一个在运行时装入所有必要依赖项的应用程序，因此看上去这个问题已得到解决。然而，这个结果并不可靠。如果今后要传送 foo.so，并且不需要 bar.so 中的符号，则此应用程序仍将毫无缘由地装入 bar.so。更好的解决方案是通过添加缺失的依赖项 bar.so 来更正 foo.so。

可通过指导来诊断是否存在补偿依赖项。

```
% cc -Bdirect -zguidance -o main main.c -L. -lfoo -lbar
ld: guidance: removal of compensating dependency recommended: libbar.so
```

可通过指导来诊断补偿依赖项，但不会从 -z discard-unused=dependencies 下删除它们。尽管要创建的目标文件可能未使用依赖项，但其他链接编辑的组件可能会使用它。删除此依赖项可能会导致创建在运行时无法执行的目标文件。

系统性地使用 -z defs 选项可以消除对补偿依赖项的需求，以生成所有动态目标文件。

-z ignore 和 -z record 选项是可以与 -z discard-unused=dependencies 选项结合使用的位置选项。这些位置选项针对选定的目标文件有选择地打开和关闭丢弃功能。

最大化可共享性

如“底层系统” [161]中所述，只有共享目标文件的文本段才可供所有使用此目标文件的进程共享。目标文件的数据段通常无法共享。在数据段中写入数据项时，每个使用共享目标文件的进程都会生成一个其完整数据段的专用内存副本。可以通过把永远不会修改的数据元素移到文本段或者完全删除数据项来减小数据段。

本节介绍了几种可用于减小数据段大小的机制。

将只读数据移动到文本中

应该使用 const 声明将只读数据元素移动到文本段中。例如，以下字符串位于 .data 节中，此节属于可写数据段。

```
char *rdstr = "this is a read-only string";
```

相反，以下字符串位于 .rodata 节中，此节是文本段中的只读数据节。

```
const char *rdstr = "this is a read-only string";
```

通过将只读元素移动到文本段中来减小数据段是一种极好的方法。但是，移动需要重定位的数据元素可能会达不到预期目标。例如，请查看以下字符串数组。

```
char *rdstrs[] = { "this is a read-only string",
                   "this is another read-only string" };
```

较好的定义可能会使用以下定义。

```
const char *const rdstrs[] = { .... };
```

此定义可确保将字符串以及指向这些字符串的指针数组放在 .rodata 节中。遗憾的是，虽然用户将地址数组视为只读，但是在运行时必须重定位这些地址。因此，此定义会导致创建文本重定位。将此定义表示为：

```
const char *rdstrs[] = { .... };
```

将确保在可重定位数组指针的可写数据段中维护这些指针。数组字符串将在只读文本段中维护。

注 - 某些编译器在生成与位置无关的代码时可以检测到会导致运行时重定位的只读指定。这些编译器会安排将此类项放在可写段中。例如，.picdata。

折叠多重定义数据

可以通过折叠多重定义数据来减小数据大小。多次出现相同错误消息的程序可以通过定义全局数据来加以改进，并可使所有其他实例都引用此全局数据。例如：

```
const char *Errmsg = "prog: error encountered: %d";

foo()
{
        ....
        (void) fprintf(stderr, Errmsg, error);
        ....
```

进行此类数据缩减的主要目标文件是字符串。可以使用 strings(1) 查看共享目标文件中的字符串用法。以下示例在 libfoo.so.1 文件中生成数据字符串的有序表。此列表中的每项都使用字符串的出现次数作为前缀。

```
$ strings -10 libfoo.so.1 | sort | uniq -c | sort -rn
```

使用自动变量

如果将关联的功能设计为使用自动（栈）变量，则可以完全删除数据项的永久性存储。通常，任何永久性存储删除操作都会导致所需运行时重定位数的相应地减少。

动态分配缓冲区

大型数据缓冲区通常应该动态分配，而不是使用永久性存储进行定义。通常，这样会从整体上节省内存，因为只分配当前调用应用程序所需的那些缓冲区。动态分配还可在不影响兼容性的情况下通过允许更改缓冲区大小来提供更大的灵活性。

最小化分页活动

任何访问新页的进程都会导致页面错误，这是一种开销很大的操作。由于共享目标文件可供许多进程使用，因此，减少由于访问共享目标文件而生成的页面错误数会对进程和整个系统有益。

将常用例程及其数据组织到一组相邻页中通常会改善性能，因为这样改善了引用的邻近性。当进程调用其中一个函数时，此函数可能已在内存中，因为它与其他常用函数邻近。同样，将相互关联的函数组织在一起也会改善引用的邻近性。例如，如果每次调用

foo（） 函数都会导致调用 bar（） 函数，则应将这些函数放在同一页中。可以使用诸如 cflow(1)、tcov(1)、prof(1) 和 gprof(1) 工具来确定代码适用范围和配置。

应将相关功能与其共享目标文件隔离开来。以前，生成的标准 C 库包含许多无关函数。仅在极少数情况下，某个可执行文件才可能会使用此库中的所有函数。由于这些函数用途广泛，因此，确定实际上最常用的函数组也具有一定的难度。相反，刚开始设计共享目标文件时，只在此共享目标文件中维护相关函数。这样会改善引用的邻近性，并会产生减小目标文件总体大小的负面影响。

重定位

在“重定位处理” [91]中，介绍了运行时链接程序重定位动态可执行文件和共享目标文件以创建可运行进程所依据的机制。“重定位符号查找” [91]和“执行重定位的时间” [169]将此重定位处理分为两类，以简化和帮助说明所涉及的机制。理论上，考虑重定位对性能的影响时也要区分这两种类别。

符号查找

当运行时链接程序需要查找符号时，缺省情况下它会搜索每个目标文件进行查找。运行时链接程序首先搜索动态可执行文件，然后按照共享目标文件的装入顺序搜索每个共享目标文件。在多数情况下，需要符号重定位的共享目标文件最终都是符号定义的提供者。

在这种情况下，如果不需要此重定位所用的符号成为共享目标文件接口的一部分，则首选将此符号转换为静态或自动变量。还可以应用符号缩减以从共享目标文件接口中删除符号。有关更多详细信息，请参见“缩减符号作用域” [46]。通过进行上述转换，链接编辑器在创建共享目标文件过程中，会产生针对这些符号处理符号重定位的开销。

应在共享目标文件中可见的全局数据项只是那些属于此共享目标文件用户接口的数据项。以前，这是要实现的硬性目标，因为通常将全局数据定义为允许从两个或多个位于不同源文件中的函数进行引用。通过应用符号缩减，可以删除不必要的全局符号。请参见“缩减符号作用域” [46]。减少从共享目标文件导出的全局符号数会降低重定位成本，并全面改善性能。

在具有许多符号重定位和依赖项的动态进程中，使用直接绑定也可以显著降低符号查找开销。请参见第 6 章 直接绑定。

执行重定位的时间

在应用程序获得控制权之前，必须在进程初始化过程中执行所有立即引用重定位。但是，可以延迟所有延迟引用重定位，直到调用第一个函数实例。立即重定位通常由于数据引用而产生。因此，减少数据引用数也会缩短进程的运行时初始化时间。

将数据引用转换为函数引用也可延迟初始化重定位成本。例如，可以通过功能接口返回数据项。此转换通常会显著改善性能，因为初始化重定位成本有效分布在整个进程执行过程中。特定的进程调用可能从不调用某些功能接口，因此完全避免了这些接口的重定位开销。

“复制重定位” [170]一节中介绍了使用功能接口的优势。该节介绍了一种在动态可执行文件与共享目标文件之间使用的开销稍大的特殊重定位机制。此外，还提供了如何避免此重定位开销的示例。

组合重定位节

可重定位目标文件中的重定位节通常与重定位必须应用到的节有一对一的关系。但是，当链接编辑器创建可执行文件或共享目标文件时，会将过程链接表重定位之外的所有重定位都放在名为 `.SUNW_reloc` 的单个公用节中。

通过此方式组合重定位记录会将所有的 `RELATIVE` 重定位组织在一起。所有符号重定位均按符号名称进行排序。组织 `RELATIVE` 重定位可允许使用 `DT_RELACOUNT/DT_RELCOUNT` `.dynamic` 项优化运行时处理。有序符号项有助于缩短运行时符号查找时间。

复制重定位

共享目标文件通常使用与位置无关的代码生成。对此类型代码的外部数据项的引用通过一组表实现间接寻址。有关更多详细信息，请参见“与位置无关的代码” [162]。在运行时，将使用数据项的实际地址更新这些表。使用这些已更新的表，无需修改代码本身即可访问数据。

但是，动态可执行文件通常并不使用与位置无关的代码创建。它们所执行的任何外部数据引用看似只能在运行时通过修改执行引用的代码来实现。应避免修改只读文本段。可以使用复制重定位技术来解决此引用。

假设使用链接编辑器创建动态可执行文件，并且发现对数据项的引用位于其中一个相关共享目标文件中。将在动态可执行文件的 `.bss` 中分配空间，空间的大小等于共享目标文件中的数据项的大小。还为此空间指定在共享目标文件中定义的符号名称。分配此数据时，链接编辑器会生成特殊的复制重定位记录，指示运行时链接程序将数据从共享目标文件复制到动态可执行文件中的已分配空间。

由于指定给此空间的符号为全局符号，因此，使用此符号可以实现任何共享目标文件引用。动态可执行文件可继承数据项。进程中对此项进行引用的任何其他目标文件都绑定到副本。生成此副本所依据的原始数据实际上变成了未使用的数据。

此机制的以下示例使用一组在标准 C 库中维护的系统错误消息。在 SunOS 操作系统早期发行版中，通过两个全局变量 `sys_errlist[]` 和 `sys_nerr` 提供此信息接口。第一个变量提供错误消息字符串数组，而第二个变量告知数组本身的大小。这些变量通常按照以下方式在应用程序中使用：

```
$ cat foo.c
extern int sys_nerr;
extern char *sys_errlist[];

char *
error(int errnumb)
{
        if ((errnumb < 0) || (errnumb >= sys_nerr))
                return (0);
        return (sys_errlist[errnumb]);
}
```

应用程序使用 `error` 函数提供焦点以获取与编号 `errnumb` 关联的系统错误消息。

对使用此代码生成的动态可执文件进行检查，可以更详细地显示复制重定位的实现。

```
$ cc -o prog main.c foo.c
$ elfdump -sN.dynsym prog | grep ' sys_'
    [24]   0x21240   0x260  OBJT GLOB  D    1 .bss           sys_errlist
    [39]   0x21230     0x4  OBJT GLOB  D    1 .bss           sys_nerr
$ elfdump -c prog
....
Section Header[19]:  sh_name: .bss
    sh_addr:      0x21230         sh_flags:   [ SHF_WRITE SHF_ALLOC ]
    sh_size:      0x270           sh_type:    [ SHT_NOBITS ]
    sh_offset:    0x1230          sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x8
....
$ elfdump -r prog

Relocation Section:  .SUNW_reloc
      type                    offset     addend  section        symbol
  ....
  R_SPARC_COPY               0x21240          0  .SUNW_reloc    sys_errlist
  R_SPARC_COPY               0x21230          0  .SUNW_reloc    sys_nerr
  ....
```

链接编辑器已在动态可执行文件的 `.bss` 中分配了空间，以便接收由 `sys_errlist` 和 `sys_nerr` 表示的数据。这些数据是运行时链接程序在进程初始化时从 C 库中复制。因此，每个使用这些数据的应用程序都在其自己的数据段中获取数据的专用副本。

此技术存在两个缺点。第一，每个应用程序都会由于运行时产生的复制数据开销而降低了性能。第二，数据数组 `sys_errlist` 的大小现在已成为 C 库接口的一部分。假设要更改此数组的大小，则可能是因为添加了新的错误消息。任何引用此数组的动态可执行文件都必须进行新的链接编辑，以便可以访问所有新错误消息。如果不进行这种新的链接编辑，则动态可执行文件中的已分配空间不足以包含新的数据。

如果动态可执行文件所需的数据由功能接口提供，则不会存在这些缺点。ANSI C 函数 `strerror(3C)` 基于提供给它的错误号返回指向相应错误字符串的指针。此函数的一种实现可能如下所示：

```
$ cat strerror.c
static const char *sys_errlist[] = {
    "Error 0",
    "Not owner",
    "No such file or directory",
    ....
};
static const int sys_nerr = sizeof (sys_errlist) / sizeof (char *);

char *
strerror(int errnum)
{
        if ((errnum < 0) || (errnum >= sys_nerr))
                return (0);
        return ((char *)sys_errlist[errnum]);
}
```

现在，可以将 `foo.c` 中的错误例程简化为使用此功能接口。通过这种简化，无需在进程初始化时执行原始复制重定位。

此外，由于数据现在对于共享目标文件而言是局部数据，因此数据不再是其接口的一部分。因此，共享目标文件可以灵活地更改数据，而不会对任何使用此数据的动态可执行文件造成不良影响。通常，从共享目标文件接口中删除数据项会改善性能，同时使得共享接口和代码更易于维护。

ldd(1) 与 `-d` 或 `-r` 选项一起使用时，可以检验动态可执行文件中存在的任何复制重定位。

例如，假设动态可执行文件 `prog` 最初根据共享目标文件 `libfoo.so.1` 生成，并且已记录以下两个复制重定位。

```
$ cat foo.c
int _size_gets_smaller[16];
int _size_gets_larger[16];
$ cc -o libfoo.so -G foo.c
$ cc -o prog main.c -L. -R. -lfoo
$ elfdump -sN.symtab prog | grep _size
    [49]   0x211d0   0x40  OBJT GLOB  D    0 .bss           _size_gets_larger
    [59]   0x21190   0x40  OBJT GLOB  D    0 .bss           _size_gets_smaller
$ elfdump -r prog | grep _size
  R_SPARC_COPY               0x211d0          0  .SUNW_reloc    _size_gets_larger
  R_SPARC_COPY               0x21190          0  .SUNW_reloc    _size_gets_smaller
```

以下是此共享目标文件的新版本，其中包含这些符号的不同数据大小。

```
$ cat foo2.c
int _size_gets_smaller[4];
int _size_gets_larger[32];
$ cc -o libfoo.so -G foo2.c
$ elfdump -sN.symtab libfoo.so | grep _size
    [37]   0x105cc   0x10  OBJT GLOB  D    0 .bss           _size_gets_smaller
    [41]   0x105dc   0x80  OBJT GLOB  D    0 .bss           _size_gets_larger
```

针对此动态可执行文件运行 ldd(1) 将显示以下内容：

```
$ ldd -d prog
    libfoo.so.1 =>   ./libfoo.so.1
    ....
    relocation R_SPARC_COPY sizes differ: _size_gets_larger
        (file prog size=0x40; file ./libfoo.so size=0x80)
        prog size used; possible data truncation
    relocation R_SPARC_COPY sizes differ: _size_gets_smaller
        (file prog size=0x40; file ./libfoo.so size=0x10)
        ./libfoo.so size used; possible insufficient data copied
....
```

ldd(1) 显示此动态可执行文件将复制此共享目标文件必须提供的所有数据，但只接受其已分配空间所允许的数据量。

通过根据与位置无关的代码生成应用程序可以删除复制重定位。请参见“与位置无关的代码” [162]。

使用 -B symbolic 选项

使用链接编辑器的- B symbolic 选项，可以将符号引用绑定到共享目标文件中的相应全局定义。此选项由来已久，因为设计它是为了在创建运行时链接程序本身时使用。

定义目标文件接口并将非公共符号降级为局部符号时，首选使用 -B symbolic 选项。请参见“缩减符号作用域” [46]。使用 -B symbolic 通常会产生某些非直观的负面影响。

如果插入以符号形式绑定的符号，则从以符号形式绑定的目标文件外部对此符号的引用将绑定到插入项。目标文件本身已在内部绑定。实际上，现在可以从进程中引用两个同名符号。导致复制重定位的以符号形式绑定的数据符号的插入情况同上。请参见“复制重定位” [170]。

注 - 以符号形式绑定的共享目标文件由 .dynamic 标志 DF_SYMBOLIC 标识。此标志仅用于提供信息。运行时链接程序在这些目标文件中处理符号查找的方式与在任何其他目标文件中的方式相同。假设任一符号绑定均在链接编辑阶段创建。

配置共享目标文件

运行时链接程序可以针对任何在运行应用程序时处理的共享目标文件生成配置信息。运行时链接程序负责将共享目标文件绑定到应用程序，因此它可以拦截任何全局函数绑定。这些绑定通过 .plt 项执行。有关此机制的详细信息，请参见“执行重定位的时间” [169]。

LD_PROFILE 环境变量将指定配置文件的共享目标文件名称。可以使用此环境变量分析单个共享目标文件。可以使用此环境变量的设置来分析一个或多个应用程序使用此共享目标文件的方式。在以下示例中，将分析命令 ls(1) 的单次调用如何使用 libc。

```
$ LD_PROFILE=libc.so.1  ls -l
```

在以下示例中，将在配置文件中记录此环境变量设置。此设置会导致某个应用程序使用 libc 来累积经过分析的信息。

```
# crle  -e LD_PROFILE=libc.so.1
$ ls -l
$ make
$ ....
```

启用配置时，便会创建配置数据文件（如果尚未存在）。此文件由运行时链接程序进行映射。在上述示例中，此数据文件为 /var/tmp/libc.so.1.profile。64 位库需要扩展配置文件格式，并使用 .profilex 后缀写入。还可以指定备用目录，使用 LD_PROFILE_OUTPUT 环境变量存储配置数据。

该配置数据文件用于存储 profil(2) 数据，并调用与使用指定共享目标文件相关的计数信息。使用 gprof(1) 可以直接检查此配置数据。

注 - gprof(1) 最常用于分析由可执行文件（已使用 cc(1) 的 -xpg 选项进行编译）创建的 gmon.out 配置数据。运行时链接程序的配置文件分析不要求使用此选项编译任何代码。其相关共享目标文件正在配置的应用程序不应该调用 profil(2)，因为此系统调用不会在同一进程中提供多次调用。由于相同的原因，不能使用 cc(1) 的 -xpg 选项编译这些应用程序。这种编译器生成的配置机制也会在 profil(2) 的基础上生成。

此配置机制最强大的功能之一就是可以分析由多个应用程序使用的共享目标文件。通常，使用一个或两个应用程序执行配置分析。但是，共享目标文件就其本质而言可供多个应用程序使用。分析这些应用程序使用共享目标文件的方式，可以了解应在何处投入更多精力以改善共享目标文件的整体性能。

以下示例说明了在某个源分层结构中创建多个应用程序时对 libc 进行的性能分析。

```
$ LD_PROFILE=libc.so.1 ; export LD_PROFILE
$ make
$ gprof -b /lib/libc.so.1 /var/tmp/libc.so.1.profile
....

granularity: each sample hit covers 4 byte(s) ....

                                called/total     parents
index  %time    self descendents  called+self    name      index
                                called/total     children
....
-----------------------------------------------
                0.33        0.00      52/29381     _gettxt [96]
```

```
                1.12        0.00     174/29381     _tzload [54]
               10.50        0.00    1634/29381     <external>
               16.14        0.00    2512/29381     _opendir [15]
              160.65        0.00   25009/29381     _endopen [3]
[2]     35.0  188.74        0.00   29381          _open [2]
-----------------------------------------------
....
granularity: each sample hit covers 4 byte(s) ....

   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call  name
 35.0     188.74   188.74    29381     6.42     6.42  _open [2]
 13.0     258.80    70.06    12094     5.79     5.79  _write [4]
  9.9     312.32    53.52    34303     1.56     1.56  _read [6]
  7.1     350.53    38.21     1177    32.46    32.46  _fork [9]
 ....
```

特殊名称 *<external>* 指示要从正在配置的共享目标文件的地址范围之外进行引用。因此，在上一示例中，从动态可执行文件，或者从其他共享目标文件（正在进行配置分析时绑定到 `libc`），对 `libc` 中的函数 `open(2)` 发出了 1634 次调用。

注 - 共享目标文件配置具有多线程安全性，但以下情况除外：一个线程调用 `fork(2)`，而另一个线程正在更新配置数据信息。使用 `fork(2)` 可取消这个限制。

♦ ♦ ♦ 第 8 章

Mapfile

mapfile 提供了对链接编辑器的操作、以及所生成的输出目标文件的广泛控制。

- 创建和/或修改输出段。
- 定义如何将输入节指定给各段，以及这些节的相对顺序。
- 指定符号作用域和/或版本控制，为可共享目标文件创建稳定的向下兼容接口。
- 根据可共享目标文件依赖项定义要使用的版本。
- 在输出目标文件中设置标头选项。
- 为动态可执行文件设置进程栈属性。
- 设置或覆盖硬件和软件功能。

注 - 所使用的不带 `mapfile` 的链接编辑器始终会生成一个有效的 ELF 输出文件。`mapfile` 选项可为用户提供大量灵活性以及针对输出目标文件的广泛控制，其中的某些功能可能会生成无效或不可用的目标文件。用户需要了解控制 ELF 格式的规则和约定。

`-M` 命令行选项用于指定要使用的 `mapfile`。单个链接操作中可以使用多个 *mapfile*。指定了多个 `mapfile` 时，链接编辑器将按给定顺序处理每个 mapfile，如同它们表示一个逻辑 `mapfile`。这一过程发生在处理任何输入目标文件之前。

系统在 `/usr/lib/ld` 目录中提供了用于解决常见问题的样例 *mapfile*。

mapfile 结构和语法

mapfile 指令可以超出一行，并且可以具有任意数量的空格，包括换行符。

对于所有语法讨论，以下表示法都适用。

- 空格或换行符可以出现在除名称或值的中间位置以外的任何位置。
- 以井号 (`#`) 开始并以换行符结束的注释可以出现在任何可以出现空格的位置。链接编辑器不会解释注释，注释仅用于提供说明。
- 所有指令均以分号 (`;`) 结束。`{....}` 节中的最后一个分号可以省略。
- 所有项都为固定宽度，所有冒号 (`:`)、分号 (`;`)、赋值运算符（`=`、`+=`、`-=`）和花括号 `{....}` 都按原样输入。

- 所有以斜体表示的项都是可替换的。
- 方括号 [. . . .] 用于表示可选语法。方括号不是表达形式的一部分，不会显示在实际指令中。
- 名称是区分大小写的字符串。表 8-2 “*mapfile* 中的名称和其他广泛使用的字符串” 列出了 *mapfile* 中的常见名称和其他字符串。可以通过三种不同形式指定名称。
  - 不带引号

    非引用名称是一个字母和数字序列。第一个字符必须是字母，后面可以没有或跟有多个字母或数字。百分号 (%)、斜杠 (/)、句点 (.) 和下划线 (_) 将视为字母。美元符号 ($) 和连字符 (-) 将视为数字。
  - 单引号

    在单引号 (') 中，名称可以包含除单引号或换行符以外的任何字符。所有字符将解释为字面字符。指定文件路径或其他包含非引用名称中所不允许的常规可打印字符的名称时，这种引用形式会很方便。
  - 双引号

    在双引号 (") 中，名称可以包含除双引号或换行符以外的任何字符。反斜杠 (\) 是转义符，其工作方式与在 C 编程语言的串文字中的使用方式类似。带有反斜杠前缀的字符将替换为其所代表的字符，如表 8-1 “双引号文本转义序列” 中所示。除了表 8-1 “双引号文本转义序列” 中所示的字符外，任何字符跟在反斜杠后都是错误的。
- *value* 表示数字值，可以是十六进制、十进制或八进制，并遵循 C 语言的整型常数所使用的规则。所有值都是无符号整数值，对于 32 位输出目标文件是 32 位的，对于 64 位输出目标文件是 64 位的。
- *segment_flags* 将内存访问权限指定为表 8-3 “段标志” 中所列一个或多个值的空格分隔列表，对应于 <sys/elf.h> 中定义的 PF_ 值。

表 8-1 双引号文本转义序列

转义序列	含义
\a	警报（响铃）
\b	退格键
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符
\\	反斜杠
\'	单引号
\"	双引号
ooo	一个八进制常数，其中 *ooo* 是一到三个八进制数字 (0....7)

表 8-2 *mapfile* 中的名称和其他广泛使用的字符串

名称	用途
segment_name	ELF 段的名称
section_name	ELF 节的名称
symbol_name	ELF 符号的名称
file_path	用于引用 ELF 目标文件或包含 ELF 目标文件的归档文件的斜杠 (/) 分隔名称的 Unix 文件路径
file_basename	*file_path* 的最终组件 (basename(1))
objname	*file_basename* 或包含在归档文件中的目标文件的名称
soname	可共享目标文件名称，用于可共享目标文件的 SONAME（例如 `libc.so.1`）
version_name	符号版本的名称，用在 ELF 版本控制节中
inherited_version_name	由另一个符号版本继承的符号版本的名称

表 8-3 段标志

标志值	含义
`READ`	段是可读的
`WRITE`	段是可写的
`EXECUTE`	段是可执行的
`0`	清除所有权限标志
`DATA`	适用于目标平台上的数据段的 READ、WRITE 和 EXECUTE 标志的组合
`STACK`	适用于目标平台的 READ、WRITE 和 EXECUTE 标志的组合，由平台 ABI 定义

mapfile 版本

mapfile 中的第一个非注释、非空行应是 *mapfile* 版本声明。该声明确定文件其余部分使用的 *mapfile* 语言的版本。本手册中所探讨的 *mapfile* 语言为版本 2。

```
$mapfile_version 2
```

没有以版本声明开始的 *mapfile* 将被视为是以 AT&T 为 System V 发行版 4 Unix (SVR4) 定义的原始 *mapfile* 语言编写的。链接编辑器保留了处理此类 *mapfile* 的功能。附录 B, *System V 发行版 4（版本 1）mapfile* 说明了其语法。

条件输入

mapfile 中的行可以设置条件以只应用于特定 ELFCLASS（32 或 64 位）或计算机类型。

```
$if expr
....
[$elif expr]
....
[$else]
....
$endif
```

条件输入表达式将计算得出一个逻辑 *true* 或 *false* 值。每个指令（`$if`、`$elif`、`$else` 和 `$endif`）都单独出现在一行中。`$if` 和后续 `$elif` 行中的表达式将依序计算，直至找到计算结果为 *true* 的表达式。值为 *false* 的行后面的文本将被放弃。成功指令行后面的文本将得到正常处理。这里的文本是指不属于条件结构的任何内容。找到成功的 `$if` 或 `$elif` 并处理了其文本后，后续的 `$elif` 和 `$else` 行及其文本将被放弃。如果所有表达式都为零，并且有一个 `$else`，则会正常处理该 `$else` 之后的文本。

`$if` 指令的作用域不能跨越多个 *mapfile*。`$if` 指令必须由使用该 `$if` 指令的 `mapfile` 中的匹配 `$endif` 终止，否则链接编辑器将发出错误。

链接编辑器会维护一个内部名称表，这些名称可用在由 `$if` 和 `$elif` 计算的逻辑表达式中。启动时，该表将使用下表中适用于要创建的输出目标文件的每个名称进行初始化。

表 8-4　预定义的条件表达式名称

名称	含义
`_ELF32`	32 位目标文件
`_ELF64`	64 位目标文件
`_ET_DYN`	共享目标文件
`_ET_EXEC`	可执行目标文件
`_ET_REL`	可重定位目标文件
`_sparc`	*Sparc* 计算机（32 或 64 位）
`_x86`	x86 计算机（32 或 64 位）
`true`	始终定义

名称区分大小写，并且必须完全按照所示形式使用。例如，定义了 `true`，但未定义 `TRUE`。其中的任何名称都可以由自身用作逻辑表达式。例如：

```
$if _ELF64
....
$endif
```

该示例的计算结果为 *true*，并且在输出目标文件为 64 位时允许链接编辑器处理所含的文本。虽然这些逻辑表达式中不允许使用数字值，但值 `1` 和 `0` 例外，它们的计算结果分别为 *true* 和 *false*。

任何未定义的名称计算结果将为 *false*。使用未定义的名称 `false` 标记应无条件跳过的输入行是一种常见做法。

```
$if false
....
$endif
```

使用下表中所示的运算符可以编写更复杂的逻辑表达式。

表 8-5　　条件表达式运算符

运算符	含义
&&	逻辑 AND
||	逻辑 OR
(expr)	子表达式
!	对后面表达式的布尔值求反

表达式从左至右计算。子表达式先于外围表达式进行计算。

例如，在为 x86 平台生成 64 位目标文件时，将计算以下构造中的行。

```
$if _ELF64 && _x86
....
$endif
```

`$add` 指令可用于向链接编辑器的已知名称表中添加新名称。在上述示例中，定义名称 `amd64` 以表示 64 位 x86 目标文件可能会便于简化 `$if` 指令。

```
$if _ELF64 && _x86
$add amd64
$endif
```

这可用于简化上述示例。

```
$if amd64
....
$endif
```

使用链接编辑器的 `-z mapfile-add` 选项，还可以将新的名称添加到链接编辑器的已知名称表。当 `mapfile` 输入需要基于外部环境属性（例如正在使用的编译器）有条件启用时，该选项非常有用。

`$clear` 指令是 `$add` 指令的逆操作。它用于从内部表中删除名称。

```
$clear amd64
```

`$add` 指令的作用一直持续到使用该 `$add` 的 `mapfile` 的最后，并且对于由相同链接操作中的链接编辑器处理的任何后续 `mapfile` 均可见。如果不希望这样，可在包含 `$add` 的 `mapfile` 的最后使用 `$clear` 删除该定义。

最后，`$error` 指令会导致链接编辑器将行中的所有剩余文本打印为致命错误，并停止链接操作。`$error` 指令可用于确保将目标文件移植到新计算机类型的程序员将无法在无提示的情况下生成缺少必要 `mapfile` 定义的错误目标文件。

```
$if _sparc
....
$elif _x86
....
$else
$error unknown machine type
$endif
```

C 语言程序员会发现用于 `mapfile` 条件输入的语法与 C 预处理程序宏语言类似。这种相似性是有意为之的。不过，`mapfile` 条件输入指令远不如 C 预处理程序提供的指令强大，这也是有意如此的。它们只能提供支持跨平台环境中的链接操作所需的最基本的功能。

这两种语言之间存在显著差异，其中包括：

- C 预处理程序可定义完整的宏语言，并且这些宏适用于源文本以及由 `#if` 和 `#elif` 预处理程序语句计算的表达式。链接编辑器 *mapfile* 不实现宏功能。
- C 预处理程序计算的表达式涉及各种数字类型以及一组丰富的运算符。*Mapfile* 逻辑表达式涉及布尔值 *true* 和 *false*，以及一组有限的运算符。
- C 预处理程序表达式涉及任意数字值，可能定义为宏，并使用 `defined()` 计算是否定义了给定宏，同时生成 *true*（非零）或 *false*（零）值。*mapfile* 逻辑表达式只处理布尔值，并且直接使用名称而不含 `defined()` 运算。如果指定的名称存在于链接编辑器的已知名称表中，则视为 *true*，否则视为 *false*。

如果需要进行更复杂的宏处理，则应考虑使用外部宏处理程序，例如 `m4`(1)。

指令语法

mapfile 指令可用于指定输出目标文件的很多方面。这些指令具有通用的语法，即为属性使用名称值对，并使用 {....} 构造表示分层结构和分组。

mapfile 指令的语法以下列通用形式为基础。

最简单的形式是一个不带值的指令名称。

directive;

第二种形式是带有一个值或由空格分隔的值列表的指令名称。

directive = *value....*;

除了所示的 "=" 赋值运算符外，还允许使用 "+=" 和 "-=" 形式的赋值。"=" 运算符将给定指令设置为给定值或值列表。"+=" 运算符用于将右侧值添加到当前值中，"-=" 运算符用于删除值。

更复杂的指令可以处理将多个属性放在花括号 {....} 中以便将其分组为一个单元的项目。

```
directive [name] {
        attribute [directive = value];
....
} [name];
```

左括号 ({) 前面可以有一个名称，用于命名给定语句的结果。类似地，在右括号 (}) 后面、终止分号 (;) 之前可以带有一个或多个可选名称。这些名称用于表示所定义的项目与其他命名项目具有某种关系。

请注意，分组中的属性的格式采用与前面所述的带有值的简单指令相同的语法，即带有一个赋值运算符 (=, +=, -=) 并后跟一个值，或是一个由空格分隔的值列表，并由分号 (;) 终止。

指令可以具有属性，后者又可以具有子属性。在这种情况下，子属性也分组在嵌套的花括号 {....} 中以反映此分层结构。

```
directive [name] {
        attribute {
                subattribute [= value];
        ....
        };
} [name....];
```

mapfile 语法对于这种嵌套所允许的深度没有限制。嵌套深度只取决于指令的要求。

mapfile 指令

链接编辑器接受以下指令。

表 8-6 *mapfile* 指令

指令	用途
CAPABILITY	硬件、软件、计算机和平台功能
DEPEND_VERSIONS	指定可共享目标文件依赖项的允许版本
HDR_NOALLOC	ELF 头和程序头不可分配
LOAD_SEGMENT	创建新的可装入段，或者修改现有装入段
NOTE_SEGMENT	创建注释段，或者修改现有注释段
NULL_SEGMENT	创建空段，或者修改现有空段
PHDR_ADD_NULL	添加空程序头项
SEGMENT_ORDER	指定输出目标文件和程序头数组中的段顺序
STACK	进程栈属性
STUB_OBJECT	指定该目标文件可生成为桩目标文件
SYMBOL_SCOPE	设置未命名全局版本中的符号属性和作用域

指令	用途
SYMBOL_VERSION	设置显式命名版本中的符号属性和作用域

后面的各个小节显示了每个支持的 *mapfile* 指令的特定语法。

CAPABILITY 指令

通常在编译时会在目标文件中记录可重定位目标文件的硬件、软件、计算机和平台功能。链接编辑器结合所有输入可重定位目标文件的功能来创建输出文件的最终功能节。可以在 mapfile 中定义功能以扩充或完全取代由输入可重定位目标文件提供的功能。

```
CAPABILITY [capid] {
        HW  = [hwcap_flag....];
        HW += [hwcap_flag....];
        HW -= [hwcap_flag....];

        HW_1  = [value....];
        HW_1 += [value....];
        HW_1 -= [value....];

        HW_2  = [value....];
        HW_2 += [value....];
        HW_2 -= [value....];

        MACHINE  = [machine_name....];
        MACHINE += [machine_name....];
        MACHINE -= [machine_name....];

        PLATFORM  = [platform_name....];
        PLATFORM += [platform_name....];
        PLATFORM -= [platform_name....];

        SF  = [sfcap_flag....];
        SF += [sfcap_flag....];
        SF -= [sfcap_flag....];

        SF_1  = [value....];
        SF_1 += [value....];
        SF_1 -= [value....];
};
```

如果存在可选的 *capid* 名称，则它可以为目标文件功能提供一个符号名称，从而在输出目标文件中获得一个 CA_SUNW_ID 功能项。如果发现多个 CAPABILITY 指令，则会使用最后一个指令提供的 *capid*。

可以使用空 CAPABILITY 指令为目标文件功能指定一个 *capid* 而不指定任何功能值。

```
CAPABILITY capid;
```

对于每种功能类型，链接编辑器会维护一个当前值 (*value*) 和一组要排除的值 (*exclude*)。对于硬件和软件功能，这些值是位掩码。对于计算机和平台功能，它们是名称列表。在处理 *mapfile* 之前，必须清除所有功能的 *value* 和 *exclude* 值。赋值运算符的工作方式如下。

- 如果使用 "+=" 运算符，则指定值将添加到该功能的当前 *value* 中，并从该功能的 *exclude* 值中删除。
- 如果使用 "-=" 运算符，则指定值将添加到该功能的 *exclude* 值中，并从该功能的当前 *value* 中删除。
- 如果使用 "=" 运算符，则指定值将替换之前的 *value*，并且 *exclude* 将重置为 0。此外，使用 "=" 将覆盖通过输入文件处理收集的任何功能。

输入目标文件在读取 *mapfile* 后进行处理。输入目标文件指定的功能值将与来自 *mapfile* 的值合并，除非使用 "=" 运算符，在这种情况下，在输入目标文件中遇到时将忽略该功能。因此，"=" 运算符将覆盖输入目标文件，而 "+=" 运算符则用于扩充它们。

在将所获得的功能值写入到输出目标文件之前，链接编辑器会减去使用 "-=" 运算符指定的任何功能值。

要从输出目标文件中完全排除某个给定功能，使用 "=" 运算符和一个空值列表已足够。例如，以下示例将抑制输入目标文件提供的任何硬件功能：

```
$mapfile_version 2
CAPABILITY {
        HW = ;
};
```

在 ELF 目标文件中，硬件和软件功能表示为目标文件功能节中的一个或多个位掩码中的位赋值。`HW` 和 `SF` `mapfile` 属性提供了这一实现的更抽象的视图，即接受一个空格分隔的符号功能名称列表，链接编辑器会将其转换为相应掩码和位。带有编号的属性（`HW_1`、`HW_2`、`SF_1`）旨在允许对底层功能位掩码进行直接数字访问。它们可用于指定尚未正式定义的功能位。如有可能，则建议使用 `HW` 和 `SF` 属性。

HW 属性

硬件功能指定为一个空格分隔的符号功能名称列表。对于 SPARC 平台，硬件功能定义为 `<sys/auxv_SPARC.h>` 中的 `AV_` 值。对于 x86 系统，硬件功能定义为 `<sys/auxv_386.h>` 中的 `AV_` 值。*mapfile* 使用相同的名称，但不带 `AV_` 前缀。例如，x86 `AV_SSE` 硬件功能在 `mapfile` 中称为 `SSE`。该列表可以包含为 `CA_SUNW_HW_` 功能掩码定义的任意功能名称。

HW_1 / HW_2 属性

`HW_1` 和 `HW_2` 属性允许将 `CA_SUNW_HW_1` 和 `CA_SUNW_HW_2` 功能掩码直接指定为数字值，或者指定为与该掩码相对应的符号硬件功能名称。

MACHINE 属性

MACHINE 属性指定目标文件可在其中执行的系统的计算机硬件名称。可通过实用程序 uname(1) 和 -m 选项显示系统的计算机硬件名称。一个 CAPABILITY 指令可以指定多个计算机名称。每个名称将在输出目标文件中获得一个 CA_SUNW_MACH 功能项。

PLATFORM 属性

PLATFORM 属性指定目标文件可在其中执行的系统的平台名称。可通过实用程序 uname(1) 和 -i 选项显示系统的平台名称。一个 CAPABILITY 指令可以指定多个平台名称。每个名称将在输出目标文件中获得一个 CA_SUNW_PLAT 功能项。

SF 属性

软件功能指定为一个空格分隔的符号功能名称列表。软件功能定义为 <sys/elf.h> 中的 SF1_SUNW_ 值。*Mapfile* 使用相同的名称，但不带 SF1_SUNW_ 前缀。例如，SF1_SUNW_ADDR32 软件功能在 mapfile 中称为 ADDR32。该列表可以包含为 CA_SUNW_SF_1 定义的任意功能名称。

SF_1 属性

SF_1 属性允许将 CA_SUNW_SF_1 功能掩码直接指定为数字值，或者指定为与该掩码相对应的符号软件功能名称。

DEPEND_VERSIONS 指令

链接可共享目标文件时，来自由目标文件导出的所有版本的符号通常可供链接编辑器使用。DEPEND_VERSIONS 指令用于将访问只限定于指定的版本。限制版本访问可用于确保给定输出目标文件不会使用在系统的较早版本上可能不可用的较新功能。

DEPEND_VERSIONS 指令的语法如下。

```
DEPEND_VERSIONS objname {
        ALLOW = version_name;
        REQUIRE = version_name;
        ....
};
```

objname 是可共享目标文件的名称，在命令行中指定。在使用 -l 命令行选项指定目标文件的一般情况下，这将是带有 lib 前缀的指定名称。例如，libc 通常在命令行中引用为 -lc，因此在 DEPEND_VERSIONS 指令中指定为 libc.so。

ALLOW 属性

ALLOW 属性指定所指定的版本以及该版本所继承的版本可供链接编辑器用来在输出目标文件中解析符号。链接编辑器会向输出目标文件要求中添加一个针对包含此版本的继承链中使用的最高版本的要求。

REQUIRE 属性

REQUIRE 将指定版本添加到输出目标文件要求中，而不管该版本是否是满足链接操作所实际需要的。

HDR_NOALLOC 指令

每个 ELF 目标文件在文件的偏移 0 处都有一个 ELF 头。可执行和可共享目标文件还包含通过 ELF 头访问的程序头。链接编辑器通常会将这些项目安排为包含在第一个可装入段中。因此，这些头中所含的信息会显示在映射的映像中，并通常由运行时链接程序使用。HDR_NOALLOC 指令会防止这种情况。

```
HDR_NOALLOC;
```

指定 HDR_NOALLOC 时，ELF 头和程序头数组仍显示在所获得的输出目标文件的开始处，但不包含在一个可装入段中，并且映像的虚拟地址计算始于第一个段的第一节，而非基于 ELF 头。

PHDR_ADD_NULL 指令

PHDR_ADD_NULL 指令会导致链接编辑器在程序头数组末尾添加指定数量的类型为 PT_NULL 的附加程序头项。额外的 PT_NULL 项可供后期处理实用程序使用。

```
PHDR_ADD_NULL = value;
```

value 必须是正整数值，并给出要创建的额外 PT_NULL 项数。所获得的程序头项的所有字段将设置为 0。

LOAD_SEGMENT / NOTE_SEGMENT / NULL_SEGMENT 指令

段是输出目标文件的一个连续部分，其中包含节。*mapfile* 段指令允许指定三种不同的段类型。

- LOAD_SEGMENT

  可装入段包含在运行时映射到进程地址空间的代码或数据。链接编辑器会为每个可分配段创建一个 PT_LOAD 程序头项，运行时链接程序使用它定位和映射段。

- NOTE_SEGMENT

  注释段包含注释节。链接编辑器会创建一个引用该段的 PT_NOTE 程序头项。注释段不可分配。

- NULL_SEGMENT

  空段中的节包含在输出目标文件中，但在运行时不能用于该目标文件。这种节的常见示例包括 .symtab 符号表以及为便于调试器操作而生成的各种节。针对空段不会创建程序头。

段指令用于在输出文件中创建新段，或更改现有段的属性值。现有段是之前定义的段，或者是“预定义段” [200]中所讨论的内置段。每个新段将添加到目标文件中相同类型的最后一个此类段之后。先添加可装入段，然后是注释段，最后添加空段。与这些段相关联的任何程序头将按照与段本身相同的相对顺序放在程序头数组中。通过为可装入段设置显式地址或者使用 SEGMENT_ORDER 指令可以更改这一缺省放置方式。

如果 *segment_name* 是一个预先存在的段，则所指定的属性将修改现有段。否则，将创建一个新段并为新段应用指定属性。链接编辑器会为未明确提供的属性填入缺省值。

注 - 选择段名称时，请注意链接编辑器的未来版本可能会添加新的预定义段。如果您的段指令中使用的名称与此新名称相匹配，则新的预定义段会将 mapfile 的含义从创建新段更改为修改现有段。防止这种情况的最好方式是避免为段使用通用名称，并为您的所有段名称指定一个唯一的前缀，例如公司/项目标识符，甚至程序的名称。例如，名为 hello_world 的程序可以使用段名称 hello_world_data_segment。

所有这三个段指令共享一组通用的核心属性。段的声明如下，可用 LOAD_SEGMENT、NOTE_SEGMENT 和 NULL_SEGMENT 其中之一取代 *directive*。

```
directive segment_name {
        ASSIGN_SECTION [assign_name];
        ASSIGN_SECTION [assign_name] {
                FILE_BASENAME = file_basename;
                FILE_OBJNAME = objname;
                FILE_PATH = file_path;
                FLAGS = section_flags;
                IS_NAME = section_name;
                TYPE = section_type;
        };

        DISABLE;

        IS_ORDER  = assign_name....;
        IS_ORDER += assign_name....;

        OS_ORDER  = section_name....;
        OS_ORDER += section_name....;
};
```

LOAD_SEGMENT 指令接受一组特定于可装入段的附加属性。这些附加属性的语法如下。

```
LOAD_SEGMENT segment_name {
        ALIGN = value;

        FLAGS  = segment_flags;
        FLAGS += segment_flags;
        FLAGS -= segment_flags;

        MAX_SIZE = value;

        NOHDR;

        PADDR = value;
        ROUND = value;

        SIZE_SYMBOL  = symbol_name....;
        SIZE_SYMBOL += symbol_name....;

        VADDR = value;
};
```

其中的任何段指令都可指定为空指令。当空段指令创建新段时，将为所有段属性设置缺省值。空段的声明如下。

```
LOAD_SEGMENT segment_name;

NOTE_SEGMENT segment_name;

NULL_SEGMENT segment_name;
```

下面将说明一个或多个段指令所接受的所有属性。

ALIGN 属性（仅限 LOAD_SEGMENT）

ALIGN 属性用于指定可装入段的对齐方式。所指定的值设置在对应于该段的程序头的 p_align 字段中。段对齐用于计算段开头的虚拟地址。

指定的对齐方式必须是 0 或 2 的幂。缺省情况下，链接编辑器将段的对齐方式设置为内置缺省值。此缺省值随 CPU 的不同而不同，甚至也可能随软件修订版的不同而不同。

ALIGN 属性与 PADDR 和 VADDR 属性相关，且必须与它们兼容。

ASSIGN_SECTION 属性

ASSIGN_SECTION 指定一个节属性组合，例如节名称、类型和标志，它们共同确定将某个节分配给某个给定段。每个此类属性集合称为一个入口条件。当节属性与入口条件的那些属性精确匹配时，该节即匹配。未指定任何属性的 ASSIGN_SECTION 将匹配与条件进行比较的任何节。

针对某个给定段允许使用多个 ASSIGN_SECTION 属性。每个 ASSIGN_SECTION 属性都独立于其他属性。如果某节与其中任何一个与段关联的 ASSIGN_SECTION 定义相匹配，则将该节指定给该段。除非段具有至少一个 ASSIGN_SECTION 属性，否则链接编辑器不会将节指定给段。

链接编辑器使用一个内部入口条件列表将节指定给段。在 mapfile 中遇到的每个 ASSIGN_SECTION 声明都按照所遇到的顺序放在此列表中。“预定义段” [200]中所讨论的内置段的入口条件将紧接最后一个 mapfile 定义的项放在此列表中。

可以为入口条件指定一个可选名称 (*assign_name*)。该名称可与 IS_ORDER 属性一起使用，以指定输入节在输出节中的放置顺序。

为放置输入节，链接编辑器从入口条件列表的开头开始，将节的属性依次与每个入口条件进行比较。节将指定给与节属性精确匹配的第一个入口条件所关联的段。如果没有匹配项，该节将放置在文件的最后，所有不可分配节通常都是如此。

ASSIGN_SECTION 接受以下各项。

FILE_BASENAME、FILE_OBJNAME、FILE_PATH

这些属性允许基于其所来自的文件的路径 (FILE_PATH)、基名 (FILE_BASENAME) 或目标文件名称 (FILE_OBJNAME) 选择节。

文件路径使用标准 Unix 斜杠分隔约定来指定。最后的路径段是路径的基名，也可以简单地称为文件名。对于归档文件，可以使用归档成员的名称对基名进行扩充，即采用 archive_name(component_name) 的形式。例如，/lib/libfoo.a(bar.o) 指定在名为 /lib/libfoo.a 的归档文件中找到的目标文件 bar.o。

FILE_BASENAME 和 FILE_OBJNAME 在应用于非归档文件时是等效的，它们将文件的给定名称根基名进行比较。在应用至归档文件时，FILE_BASENAME 将检查归档文件名称的基名，而 FILE_OBJNAME 将检查包含在归档文件中的目标文件的名称。

每个 ASSIGN_SECTION 会维护所有 FILE_BASENAME、FILE_PATH 和 FILE_OBJNAME 值的一个列表。如果其中任何一个定义与某个输入文件相匹配，则文件匹配。

IS_NAME

输入节名称。

TYPE

指定 ELF *section_type*，它可以是 <sys/elf.h> 中定义的任何 SHT_ 常量，且不带 SHT_ 前缀。例如，PROGBITS、SYMTAB 或 NOBITS。

FLAGS

FLAGS 属性使用 *section_flags* 将节属性指定为表 8-7 “节 FLAGS 值” 中所给定的一个或多个值的空格分隔列表，这些值与 <sys/elf.h> 中定义的 SHF_ 值相对应。如果某个标志前面带有叹号 (!)，则不能显式存在该属性。在下面的示例中，节定义为可分配但不可写入。

```
ALLOC !WRITE
```

未明确出现在 *section_flags* 列表中的标志将予以忽略。在上面的示例中，将节与指定标志进行匹配时，将只检查 ALLOC 和 WRITE 的值。其他节标志可以具有任意值。

表 8-7 节 FLAGS 值

标志值	含义
ALLOC	节是可分配的
WRITE	节是可写的
EXECUTE	节是可执行的
AMD64_LARGE	节可以大于 2 GB

DISABLE 属性

DISABLE 属性会导致链接编辑器忽略段。不会将任何节指定给禁用的段。当由以下段指令引用时，将自动重新启用段。因此，空引用即足以重新启用禁用的段。

```
segment segment_name;
```

FLAGS 属性（仅限 LOAD_SEGMENT）

FLAGS 属性将段权限指定为表 8-3 “段标志” 中所示权限的空格分隔列表。缺省情况下，用户定义的段将获得 READ、WRITE 和 EXECUTE 权限。“预定义段” [200]中所描述的预定义段的缺省标志由链接编辑器提供，在某些情况下可能与平台相关。

该属性允许采用三种形式。

```
FLAGS  = segment_flags....;
FLAGS += segment_flags....;
FLAGS -= segment_flags....;
```

简单的 "=" 赋值运算符将当前标志替换为新集合，"+=" 形式将新标志添加到现有集合中，而 "-=" 形式将指定标志从现有集合中删除。

IS_ORDER 属性

链接编辑器通常按照遇到输出节的顺序将其放入段中。类似地，构成输出节的输入节也按照遇到它们的顺序进行放置。IS_ORDER 属性可用于改变输入节的这种缺省放置方式。IS_ORDER 指定入口条件名称 (*assign_name*) 的空格分隔列表。通过其中一个入口条件匹配的节将放置在输出节的开头，并按 IS_ORDER 指定的顺序排序。通过未在 IS_ORDER 列表中的入口条件匹配的节将按照遇到它们的顺序放置在已排序节之后。

使用 "=" 形式的赋值时，将放弃给定段的 IS_ORDER 的先前值并替换为新列表。"+=" 形式的 IS_ORDER 将新列表串联到现有列表的最后。

IS_ORDER 属性与编译器的 -xF 选项一起使用时会更有用。使用 -xF 选项编译文件时，将该文件中的每个函数都放置在与 text 节具有相同属性的单独节中。这些节称为 .text %function_name。

例如，使用 -xF 选项编译包含 main（）、foo（） 和 bar（） 这三个函数的文件时，会生成一个可重定位的目标文件，并会将三个函数的文本放置在名为 .text%main、.text %foo 和 .text%bar 的节中。当链接编辑器将这些节放入输出中时，将删除 % 和 % 之后的任何内容。因此，这三个函数都将放在 .text 输出节中。IS_ORDER 属性可用于强制将它们以相对于彼此的特定顺序放在 .text 输出节中。

请考虑以下用户定义的 mapfile。

```
$mapfile_version 2
LOAD_SEGMENT text {
        ASSIGN_SECTION text_bar  { IS_NAME = .text%bar };
        ASSIGN_SECTION text_main { IS_NAME = .text%main };
        ASSIGN_SECTION text_foo  { IS_NAME = .text%foo };
        IS_ORDER = text_foo text_bar text_main;
};
```

不管这三个函数在源代码中的顺序或者链接编辑器遇到它们的顺序如何，它们在输出目标文件文本段中的顺序都将是 foo（） 、bar（） 和 main（） 。

MAX_SIZE 属性（仅限 LOAD_SEGMENT）

缺省情况下，链接编辑器允许段增大到段内容所需要的大小。MAX_SIZE 属性可用于指定段的最大大小。如果设置了 MAX_SIZE，链接编辑器会在段增长超出指定大小时生成错误。

NOHDR 属性（仅限 LOAD_SEGMENT）

如果设置了 NOHDR 属性的段成为输出目标文件中的第一个可装入段，则 ELF 头和程序头将不会包含在该段中。

NOHDR 属性与顶级 HDR_NOALLOC 指令的区别在于 HDR_NOALLOC 是一个基于段的值，仅当段成为第一个可装入段时才有效。该功能主要用于为早期的 *mapfile* 提供功能奇偶校验。有关更多详细信息，请参见附录 B, *System V 发行版 4（版本 1）mapfile*。

建议优先使用 HDR_NOALLOC 指令，而不是段 NOHDR 属性。

OS_ORDER 属性

链接编辑器通常按照遇到输出节的顺序将其放入段中。OS_ORDER 属性可用于改变输出节的这种缺省放置方式。OS_ORDER 指定输出节名称 (*section_name*) 的空格分隔列表。列

出的节放在段的开头，并按照 OS_ORDER 指定的顺序排序。未在 OS_ORDER 中列出的节将按照遇到它们的顺序放在已排序节之后。

使用 "=" 形式的赋值时，将放弃给定段的 OS_ORDER 的先前值并替换为新列表。"+=" 形式的 OS_ORDER 将新列表串联到现有列表的最后。

PADDR 属性（仅限 LOAD_SEGMENT）

PADDR 属性用于为段指定一个显式物理地址。指定的值必须是 0 或 2 的幂。所指定的值设置在对应于该段的程序头的 p_addr 字段中。缺省情况下，链接编辑器将段的物理地址设置为 0，因为此字段对于用户模式目标文件没有意义，它主要用于非用户级目标文件，如操作系统内核。

ROUND 属性（仅限 LOAD_SEGMENT）

ROUND 属性用于指定段的大小应向上舍入为给定值。指定的舍入值必须是 0 或 2 的幂。缺省情况下，链接编辑器将段的舍入系数设置为 1，即不对段大小向上舍入。

SIZE_SYMBOL 属性（仅限 LOAD_SEGMENT）

SIZE_SYMBOL 属性定义要由链接编辑器创建的节大小符号名称的空格分隔列表。大小符号是全局绝对符号，以字节为单位表示段的大小。可以在目标文件中引用这些符号。为访问代码中的符号，应确保 *symbol_name* 是该语言中的合法标识符。建议采用 C 编程语言的符号命名规则，因为这种符号可能能够由任何其他语言访问。

可以使用 "=" 形式的赋值建立初始值，并且针对每个链接编辑器会话只能使用一次。"+=" 形式的 SIZE_SYMBOL 将新列表串联到现有列表的最后，并可根据需要使用任意次。

VADDR（仅限 LOAD_SEGMENT）

VADDR 属性用于为段指定一个显式虚拟地址。所指定的值设置在对应于该段的程序头的 p_vaddr 字段中。缺省情况下，链接编辑器会在创建输出文件时将虚拟地址指定给段。

SEGMENT_ORDER 指令

SEGMENT_ORDER 指令用于为输出目标文件中的段指定非缺省顺序。

SEGMENT_ORDER 接受段名称的空格分隔列表。

```
SEGMENT_ORDER  = segment_name....;
SEGMENT_ORDER += segment_name....;
```

使用 "=" 形式的赋值时，将放弃之前的段顺序列表并替换为新列表。"+=" 形式的赋值将新列表串联到现有列表的最后。

缺省情况下，链接编辑器按以下方式确定段顺序。

1. 具有通过 LOAD_SEGMENT 指令的 VADDR 属性设置的显式地址的可装入段，按地址排序。
2. 使用 SEGMENT_ORDER 指令排序的段，按指定顺序排序。
3. 没有显式地址且未在 SEGMENT_ORDER 列表中列出的可装入段。
4. 没有显式地址且未在 SEGMENT_ORDER 列表中列出的注释段。
5. 没有显式地址且未在 SEGMENT_ORDER 列表中列出的空段。

注 - ELF 具有一些格式正确的目标文件所必须遵循的隐式约定。

- 第一个可装入段应该是只读、可分配和可执行的，并接收 ELF 头和程序头数组。这通常是预定义的文本段。
- 可执行文件中的最后一个可装入段应该是可写的，并且动态堆的开头通常紧随其后放置在相同的虚拟内存映射中。

可以使用 *mapfile* 创建不遵循这些要求的目标文件。但应避免如此操作，因为运行这种目标文件的结果是不确定的。

除非指定 HDR_NOALLOC 指令，否则链接编辑器将强制要求第一个段必须是可装入段，而不是注释段或空段。HDR_NOALLOC 不能用于用户级目标文件，因此没什么实用价值。生成操作系统内核时会使用该功能。

STACK 指令

STACK 指令指定进程栈的属性。

```
STACK {
        FLAGS  = segment_flags....;
        FLAGS += segment_flags....;
        FLAGS -= segment_flags....;
};
```

FLAGS 属性指定段权限的空格分隔列表，其中包含表 8-3 “段标志” 中所描述的任意值。

该属性允许采用三种形式。简单的 "=" 赋值运算符将当前标志替换为新集合，"+=" 形式将新标志添加到现有集合中，而 "-=" 形式将指定标志从现有集合中删除。

缺省栈权限由平台 ABI 定义，并随平台的不同而变化。目标平台的值使用段标志名称 STACK 指定。

在某些平台上，ABI 要求的缺省权限包括 EXECUTE。一般很少需要 EXECUTE，并且它通常被视为一个潜在的安全风险。建议从栈中删除 EXECUTE 权限。

```
STACK {
        FLAGS -= EXECUTE;
};
```

STACK 指令在输出 ELF 目标文件中反映为一个 PT_SUNWSTACK 程序头项。

STUB_OBJECT 指令

STUB_OBJECT 指令通知链接编辑器可以将 mapfile 所描述的目标文件生成为一个桩目标文件。

```
STUB_OBJECT;
```

桩共享目标文件是完全通过命令行中提供的 mapfile 中的信息生成的。当指定 -z stub 选项生成桩目标文件时，要求 mapfile 中存在 STUB_OBJECT 指令，链接编辑器使用符号 ASSERT 属性中的信息创建与实际目标文件的符号相匹配的全局符号。

SYMBOL_SCOPE / SYMBOL_VERSION 指令

SYMBOL_SCOPE 和 SYMBOL_VERSION 指令用于指定全局符号的作用域和属性。SYMBOL_SCOPE 在未命名的基础符号版本的上下文中工作，而 SYMBOL_VERSION 用于将符号收集到显式命名的全局版本中。SYMBOL_VERSION 指令允许创建稳定接口，该接口支持目标文件以向下兼容方式发展。

SYMBOL_VERSION 的语法如下。

```
SYMBOL_VERSION version_name {
        symbol_scope:
                *;

        symbol_name;
        symbol_name {
                ASSERT = {
                        ALIAS = symbol_name;
                        BINDING = symbol_binding;
                        TYPE = symbol_type;

                        SIZE = size_value;
                        SIZE = size_value[count];
          VALUE = value;
```

```
                };
                AUXILIARY = soname;
                FILTER = soname;
                FLAGS = symbol_flags....;

                SIZE = size_value;
                SIZE = size_value[count];

                TYPE = symbol_type;
                VALUE = value;
        };
} [inherited_version_name....];
```

SYMBOL_SCOPE 不接受版本名称，除此之外都相同。

```
SYMBOL_SCOPE {
        ....
};
```

在 SYMBOL_VERSION 指令中，*version_name* 为该符号定义集合提供一个标签。该标签标识输出目标文件中的 *version definition*。可以指定一个或多个由空格分隔的继承版本 (*inherited_version_name*)，这时新定义的版本将从命名的版本继承。请参见第 9 章 接口和版本控制。

symbol_scope 定义 SYMBOL_SCOPE 或 SYMBOL_VERSION 指令中符号的作用域。缺省情况下，符号被假定为具有全局作用域。通过指定 *symbol_scope* 并后跟一个冒号 (:) 可以修改此缺省设置。这些行确定其后所有符号的符号作用域，直至后续作用域声明做出更改。下表给出了可能的作用域值及其含义。

表 8-8　　符号作用域类型

作用域	含义
default / global	此作用域的全局符号对所有外部目标文件都可见。从目标文件内部对这种符号的引用在运行时绑定，从而允许进行插入。这种可见性作用域提供了一种缺省设置，可通过其他符号可见性技术进行降级或消除。此作用域定义与具有 STV_DEFAULT 可见性的符号产生相同的效果。请参见表 12-23 “ELF 符号可见性”。
hidden / local	此作用域的全局符号缩减为具有局部绑定的符号。此作用域的符号对其他外部目标文件不可见。此作用域定义与具有 STV_HIDDEN 可见性的符号产生相同的效果。请参见表 12-23 “ELF 符号可见性”。
protected / symbolic	此作用域的全局符号对所有外部目标文件都可见。从目标文件内部对这些符号的引用在链接编辑时绑定，从而可以防止运行时插入。这种可见性作用域可通过其他符号可见性技术进行降级或消除。此作用域定义与具有 STV_PROTECTED 可见性的符号产生相同的效果。请参见表 12-23 “ELF 符号可见性”。
exported	此作用域的全局符号对所有外部目标文件都可见。从目标文件内部对这种符号的引用在运行时绑定，从而允许进行插入。这种符号可见性不能通过任何其他符号可见性技术进行降级或消除。此作用域定义与具有 STV_EXPORTED 可见性的符号产生相同的效果。请参见表 12-23 “ELF 符号可见性”。
singleton	此作用域的全局符号对所有外部目标文件都可见。从目标文件内部对这种符号的引用在运行时绑定，并确保进程中的所有引用只绑定到一个符号实

作用域	含义
	例。这种符号可见性不能通过任何其他符号可见性技术进行降级或消除。此作用域定义与具有 STV_SINGLETON 可见性的符号产生相同的效果。请参见表 12-23 “ELF 符号可见性”。
eliminate	此作用域的全局符号处于隐藏状态，并会删除其符号表项。此作用域定义与具有 STV_ELIMINATE 可见性的符号产生相同的效果。请参见表 12-23 “ELF 符号可见性”。

symbol_name 是符号的名称。该名称可生成符号定义或符号引用，具体取决于任何限定属性。在最简单的没有任何限定属性的形式中，将创建符号引用。该引用与使用-Defining Additional Symbols with the -u option中所讨论的 “使用 -u 选项定义其他符号” [42] 选项生成的引用完全相同。通常，如果符号名称后跟任何限定属性，则使用关联的属性生成符号定义。

定义了局部作用域时，可以将符号名称定义为特殊自动缩减指令 "*"。没有显式定义可见性的符号将在所生成的动态目标文件中降级为局部绑定。显式可见性定义源自 mapfile 定义或封装在可重定位目标文件中的可见性定义。类似地，定义了 eliminate（删除）作用域时，可以将符号名称定义为特殊自动删除指令 "*"。没有显式定义可见性的符号将从所生成的动态目标文件中删除。

如果指定了 SYMBOL_VERSION 指令，或者如果使用 SYMBOL_VERSION 或 SYMBOL_SCOPE 指定了自动缩减，则会在所创建的映像中记录版本控制信息。如果此映像是可执行目标文件或共享目标文件，则还会应用任何符号缩减。

如果要创建的映像是可重定位目标文件，则缺省情况下不会应用符号缩减。在这种情况下，任何符号缩减都将记录在版本控制信息中。当最终使用可重定位目标文件来生成可执行文件或共享目标文件时，将应用这些符号缩减。在生成可重定位目标文件时，可以使用链接编辑器的 -B reduce 选项强制执行符号缩减。

第 9 章 接口和版本控制中提供了版本控制信息的更详细说明。

注 - 为了确保接口定义的稳定性，定义符号名称时不提供通配符扩展功能。

---

symbol_name 可单独列出，以便仅将符号分配给一个版本和/或指定其作用域。可以在 {} 括号中指定可选符号属性。下面将说明各个有效属性。

ASSERT 属性

ASSERT 属性用于指定符号的预期特征。链接编辑器会比较通过链接编辑所获得的符号特征与 ASSERT 属性指定的符号特征。如果实际属性与声明的属性不一致，则会发出致命错误并且不会创建输出目标文件。

ASSERT 属性的解释取决于是使用 STUB_OBJECT 指令还是使用 -z stub 命令行选项。三种可能的情况如下所述。

1. 不使用 STUB_OBJECT 指令时，将不需要 ASSERT 属性。但是，如果 ASSERT 属性存在，则会根据通过链接编辑收集到的实际值对其属性进行验证。如果任何 ASSERT 属性与其关联的实际值不匹配，则链接编辑过程将终止且不成功。
2. 使用 STUB_OBJECT 指令且指定了 -z stub 命令行选项时，链接编辑器将使用 ASSERT 指令定义目标文件提供的全局符号的属性。请参见“桩目标文件” [70]。
3. 使用 STUB_OBJECT 指令但未指定 -z stub 命令行选项时，链接编辑器要求所获得的目标文件中的所有全局数据都有一个将其声明为数据并提供大小的关联 ASSERT 指令。在此模式下，如果未指定 TYPE ASSERT 属性，则假定为 GLOBAL。类似地，如果未指定 SH_ATTR，则假定为缺省值 BITS。这些缺省值可确保桩目标文件和实际目标文件的数据属性是兼容的。所获得的 ASSERT 语句的计算方式与上述第一种情况相同。请参见“STUB_OBJECT 指令” [195]。

ASSERT 接受以下属性。

ALIAS

为之前定义的符号定义别名。别名符号具有与主符号相同的类型、值和大小。ALIAS 属性不能与 TYPE、SIZE 和 SH_ATTR 属性一起使用。指定了 ALIAS 时，类型、大小和节属性将从别名符号中获取。

BIND

指定 ELF *symbol_binding*，它可以是 <sys/elf.h> 中定义的任意 STB_ 值，但去掉 STB_ 前缀。例如，GLOBAL 或 WEAK。

TYPE

指定 ELF *symbol_type*，它可以是 <sys/elf.h> 中定义的任何 STT_ 常量，且不带 STT_ 前缀。例如，OBJECT、COMMON 或 FUNC。此外，为与其他 mapfile 使用情况兼容，还可以分别为 STT_FUNC 和 STT_OBJECT 指定 FUNCTION 和 DATA。TYPE 不能与 ALIAS 一起使用。

SH_ATTR

指定与符号关联的节的属性。表 8-9 “SH_ATTR 值” 列出了可指定的 *section_attributes*。SH_ATTR 不能与 ALIAS 一起使用。

SIZE

指定预期符号大小。SIZE 不能与 ALIAS 一起使用。*size_value* 参数的语法如 SIZE 属性的讨论中所述。请参见 “SIZE 属性” [200]。

VALUE

指定预期的符号值。

表 8-9 SH_ATTR 值

节属性	含义
BITS	节的类型不是 SHT_NOBITS

节属性	含义
NOBITS	节的类型是 SHT_NOBITS

AUXILIARY 属性

指示此符号是共享目标文件名称 (*soname*) 的辅助过滤器。请参见“生成辅助过滤器” [131]。

FILTER 属性

指示此符号是共享目标文件 *name* 的过滤器。请参见“生成标准过滤器” [128]。过滤器符号不需要输入可重定位目标文件提供任何后备实现。因此，使用此指令并定义符号的类型可以创建绝对符号表项。

FLAGS 属性

symbol_flags 将符号属性指定为一个或多个以下值的空格分隔列表。

表 8-10 符号 FLAG 值

标志	含义
DIRECT	指示应直接绑定到此符号。与符号定义一起使用时，此关键字会将来自所生成目标文件中的任何引用直接绑定到该定义。与符号引用一起使用时，此标志会导致直接绑定到提供定义的依赖项。请参见第 6 章 直接绑定。此标志还可与 PARENT 标志一起使用以便在运行时建立与任何父项的直接绑定。
DYNSORT	指示此符号应包含在排序节中。请参见“符号排序节” [334]。符号类型必须为 STT_FUNC、STT_OBJECT、STT_COMMON 或 STT_TLS。
EXTERN	指示在要创建的目标文件外部定义符号。通常定义此关键字以为回调例程设置标签。此标志会抑制将使用 -z defs 选项标记的未定义符号。此标志仅在生成符号引用时才有意义。如果在链接编辑时合并的目标文件内出现此符号的定义，则会在无提示的情况下忽略该关键字。
INTERPOSE	指示此符号将充当插入项。仅在生成动态可执行文件时才能使用此标志。在定义插入符号方面，此标志可提供比使用 -z interpose 选项更精细的控制。
NODIRECT	指示不应直接绑定到此符号。此状态适用于要创建的目标文件中的引用或外部引用中的引用。请参见第 6 章 直接绑定。此标志还可与 PARENT 标志一起使用以防止在运行时直接绑定到任何父项。
NODYNSORT	指示此符号不应包含在排序节中。请参见“符号排序节” [334]。
PARENT	指示在要创建的目标文件的父项中定义符号。父项是在运行时将此目标文件作为显式依赖项进行引用的目标文件。父项还可以使用 dlopen(3C) 在运行时引用此目标文件。通常定义此标志以为回调例程设置标签。此标志可与 DIRECT 或 NODIRECT 标志一起使用以建立对父项的单个直接或非直接引用。此标志会抑制将使用 -z defs 选项标记的未定义符号。此标志仅在生成符号引用时才有意义。如果在链接编辑时合并的目标文件内出现此符号的定义，则会在无提示的情况下忽略该关键字。
STUB_ELIMINATE	表示应在桩目标文件中省略此符号。请参见“使用桩目标文件隐藏过时的接口” [73]。

SIZE 属性

设置大小属性。此属性会导致创建符号定义。

size_value 参数可以是数字值，或者是符号名称 `addrsize`。`addrsize` 表示可保存一个内存地址的计算机字的大小。在生成 32 位目标文件时，链接编辑器会将 `addrsize` 替换为值 4，在生成 64 位目标文件时替换为值 8。`addrsize` 对于表示类型为 `long` 的指针变量和 C 变量的大小很有用，因为它会自动针对 32 位和 64 位目标文件进行调整，而无需使用条件输入。

可以选择为 *size_value* 参数添加一个括在方括号中的 *count* 值后缀。如果存在 *count*，则 *size_value* 和 *count* 将相乘以得出最终的大小值。

TYPE 属性

符号类型属性。此属性可以是 `COMMON`、`DATA` 或 `FUNCTION`。`COMMON` 会生成一个暂定符号定义。`DATA` 和 `FUNCTION` 会生成一个节符号定义或绝对符号定义。请参见“符号表节” [326]。

数据属性会导致创建 `OBJT` 符号。带有大小但不含值的数据属性会通过将符号与某个 ELF 节相关联来创建节符号。此节将填充零。函数属性会导致创建 `FUNC` 符号。

带有大小但不含值的函数属性会通过将符号与某个 ELF 节相关联来创建节符号。一个由链接编辑器生成的带有以下签名的 void 函数将指定给此节。

```
void (*)(void)
```

含有值的数据或函数属性将生成相应符号类型并带有一个绝对 `ABS` 节索引。

创建节数据符号对于创建过滤器很有用。从可执行文件中对过滤器的节数据符号的外部引用会导致生成相应的复制重定位。请参见“复制重定位” [170]。

VALUE属性

指示值属性。此属性会导致创建符号定义。

预定义段

链接编辑器提供一组预定义的输出段描述符和入口条件。这些定义可满足多数链接情况的需要，并且符合系统预期的 ELF 布局规则和约定。

text、data 和 extra 段是常用段，其他段可提供更专门的用途，如下所述。

- text

  text 段定义只读可执行文件的可装入段，后者接受可分配、非可写节。这包括可执行代码、程序所需的只读数据以及链接编辑器生成的供运行时链接程序（如动态符号表）使用的只读数据。

  text 段是进程中的第一个段，因此链接编辑器会为其指定 ELF 头和程序头数组。使用 HDR_NOALLOC *mapfile* 指令可以防止这种情况。

- data

  data 段定义可写的可装入段。data 段用于程序所需的可写数据和运行时链接程序使用的可写数据，例如全局偏移表 (Global Offset Table, GOT) 和诸如 SPARC 等要求 PLT 节为可写的体系结构上的过程链接表 (Procedure Linkage Table, PLT)。

- extra

  extra 段捕获未指定在其他位置并由最后的入口条件记录引向该处的所有节。常见的示例包括完整符号表 (.symtab) 以及为便于调试器操作而生成的各种节。这是一个空段，并且没有对应的程序头表项。

- note

  note 段捕获所有类型为 SHT_NOTE 的节。链接编辑器会提供一个引用 note 段的 PT_NOTE 程序头项。

- lrodata / ldata

  x86-64 ABI 定义小型、中型和大型编译模型。ABI 要求中型和大型模型的节设置 SHF_AMD64_LARGE 节标志。缺少 SHF_AMD64_LARGE 的输入节必须放在大小不超过 2 GB 的输出段中。lrodata 和 ldata 预定义段仅适用于 x86-64 输出目标文件，用于处理设置了 SHF_AMD64_LARGE 标志的节。lrodata 接收只读节，ldata 接收其他节。

- bss

  ELF 允许任何段包含 NOBITS 节。链接编辑器将这种节放在其所指定到的段的最后。这可以使用程序头项 p_filesz 和 p_memsz 字段来实现，并且必须遵循以下规则。

  ```
  p_memsz >= p_filesz
  ```

  如果 p_memsz 大于 p_filesz，则多余的字节为 NOBITS。第一个 p_filesz 字节来自目标文件，其余直至 p_memsz 的任何字节会在使用前由系统归零。

  缺省赋值规则将只读 NOBITS 节指定给 text 段，将可写 NOBITS 节指定给 data 段。链接编辑器将 bss 段定义为可接受可写 NOBITS 节的备选段。该段缺省为禁用，并且必须显式启用才能使用。

  由于可写 NOBITS 节可轻松作为 data 段的一部分进行处理，因此设置单独的 bss 段的好处可能不会立即显现出来。根据约定，进程动态内存堆始于最后一个段的末尾，必须是可写的。这通常是 data 段，但如果启用了 bss，bss 将成为最后一个段。在生成动态可执行文件时，启用具有适当对齐方式的 bss 段可用于启用堆的大型页分配。例如，以下示例将启用 bss 段并设置 4MB 对齐。

  ```
  LOAD_SEGMENT bss {
          ALIGN=0x400000;
  };
  ```

注 - 用户应注意，对齐规范可能是特定于计算机的，并且在不同硬件平台上的优势可能并不相同。未来的发行版中可能会出现更灵活的请求最佳底层页面大小的方式。

映射示例

下面是用户定义的 *mapfile* 示例。示例中左边的编号用于教学演示。实际上只有编号右边的信息会出现在 `mapfile` 中。

示例：节到段的分配

此示例说明如何定义段并为其指定输入节。

例 8-1 基本节到段的分配

```
1    $mapfile_version 2
2    LOAD_SEGMENT elephant {
3            ASSIGN_SECTION {
4                    IS_NAME=.data;
5                    FILE_PATH=peanuts.o;
6            };
7            ASSIGN_SECTION {
8                    IS_NAME=.data;
9                    FILE_OBJNAME=popcorn.o;
10           };
11   };
12
13   LOAD_SEGMENT monkey {
14           VADDR=0x80000000;
15           MAX_SIZE=0x4000;
16           ASSIGN_SECTION {
17                   TYPE=progbits;
18                   FLAGS=ALLOC EXECUTE;
19           };
20           ASSIGN_SECTION {
21                   IS_NAME=.data
22           };
23   };
24
25   LOAD_SEGMENT donkey {
26           FLAGS=READ EXECUTE;
27           ALIGN=0x1000;
28           ASSIGN_SECTION {
29                   IS_NAME=.data;
```

```
30          };
31  };
32
33  LOAD_SEGMENT text {
34          VADDR=0x80008000
35  };
```

在此示例中处理四个单独的段。每个 mapfile 都以一个 $mapfile_version 声明开始，如行 1 所示。段 elephant（行 2-11）接收来自文件 peanuts.o 或 popcorn.o 的所有数据节。目标文件 popcorn.o 可以来自归档文件，这时归档文件可以具有任意名称。或者，popcorn.o 可以来自具有基名 popcorn.o 的任意文件。与此相反，peanuts.o 只能来自具有该确切名称的文件。例如，提供给链接编辑的文件 /var/tmp/peanuts.o 与 peanuts.o 并不匹配。

段 monkey（行 13-23）具有一个虚拟地址 *0x80000000* 和最大长度 *0x4000*。此段接收类型为 PROGBITS 且可分配、可执行的所有节，以及尚未在段 elephant 中的名为 .data 的所有节。进入 monkey 段的 .data 节不需要是 PROGBITS 或是可分配、可执行的，因为它们与第 20 行而不是第 16 行上的入口条件相匹配。这表明 ASSIGN_SECTION 属性中的子属性之间存在 *and*（与）关系，而单个段的不同 ASSIGN_SECTION 属性之间存在 *or*（或）关系。

donkey 段（行 25-31）指定了非缺省权限标志和对齐方式，并将接受名为 .data 的所有节。但是，此段不会指定任何节，因此，段 donkey 永远不会出现在输出目标文件中。其原因是链接编辑器按照入口条件在 mapfile 中出现的顺序对其进行检查。在此 mapfile 中，段 elephant 接受某些 .data 节，段 monkey 接受其余的所有节，而没有为 donkey 留下任何节。

行 33-35 将 text 段的虚拟地址设置为 *0x80008000*。text 段是其中一种标准预定义段，如“预定义段” [200]中所述，因此该语句将修改现有段，而不是创建一个新段。

示例：预定义节的修改

以下 mapfile 示例将处理预定义的 text 和 data 段、头选项以及排序段中的节。

例 8-2　　预定义节的处理和节到段的分配

```
1   $mapfile_version 2
2   HDR_NOALLOC;
3
4   LOAD_SEGMENT text {
5           VADDR=0xf0004000;
6           FLAGS=READ EXECUTE;
7           OS_ORDER=.text .rodata;
9           ASSIGN_SECTION {
10                  TYPE=PROGBITS;
11                  FLAGS=ALLOC !WRITE;
12          };
```

```
13   };
14
15   LOAD_SEGMENT data {
16           FLAGS=READ WRITE EXECUTE;
17           ALIGN=0x1000;
18           ROUND=0x1000;
19   };
```

与往常一样，第一行声明要使用的 *mapfile* 语言版本。HDR_NOALLOC 指令（行 2）指定所获得的目标文件不应在目标文件的第一个可分配段中包含 ELF 头或程序头数组，该段是预定义的 text 段。

行 4-13 中的段指令为 text 段设置虚拟地址和权限标志。该指令还指定名为 .text 的节应放在段的开头，后跟任何名为 .rodata 的节，再后面是所有其他节。最后将可分配、非可写的 PROGBITS 节指定给该段。

行 15-19 中的段指令指定 data 段必须在边界 *0x1000* 上对齐。其效果是以相同的对齐方式对齐段中的第一个节。段的长度将向上舍入为与对齐方式相同的值的倍数。段权限设置为读取、写入和执行。

链接编辑器内部：节和段的处理

下面说明链接编辑器用来将节指定给输出段的内部过程。这些信息并不是使用 *mapfile* 所必需的。这些信息主要提供给对链接编辑器的内部过程感兴趣的读者，以及希望进一步了解链接编辑器如何解释和执行段的 *mapfile* 指令的读者。

节到段的分配

将输入节分配给输出段的过程涉及以下数据结构。

- 输入节

  输入节是从可重定位目标文件输入读入到链接编辑器中的。其中某些由链接编辑器检查和处理，另一些则只是传递至输出而不检查其内容（例如 PROGBITS）。
- 输出节

  输出节是写入到输出目标文件中的节。其中某些由通过输入目标文件传递的节串联而成。另一些（如符号表和重定位节）由链接编辑器自己生成，并且通常会并入从输入目标文件中读取的信息。

  当链接编辑器传递输入节以使之成为输出节时，该节通常会保留输入节名称。不过，链接编辑器在某些情况下会修改名称。例如，链接编辑器会转换 name%XXX 形式的输入节名称，在输出节名称中删除 % 字符及其后的任何字符。
- 段描述符

  链接编辑器会维护一个已知段列表。该列表初始时包含预定义段，如“预定义段” [200]中所述。当使用 LOAD_SEGMENT、NOTE_SEGMENT 或 NULL_SEGMENT *mapfile*

指令创建新段时，会向此列表中添加新段的附加段描述符。除非通过设置虚拟地址 (LOAD_SEGMENT) 或使用 SEGMENT_ORDER 指令进行显式排序，否则新段将位于列表的末尾（位于相同类型的其他段之后）。

创建输出目标文件时，链接编辑器只会为接收节的段创建程序头。将以无提示方式忽略空段。因此，用户指定的段定义可完全取代预定义的段定义的使用，尽管没有明确的工具用于从链接编辑器列表中删除段定义。

- 入口条件

  为将节放入给定段中所需的一组节属性称为该段的入口条件。给定段可以具有任意数量的入口条件。

  链接编辑器会维护一个所有定义的入口条件的内部列表。该列表用于将节放入段中，如下所述。每个 mapfile 按照在 mapfile 中遇到 ASSIGN_SECTION 属性所创建的入口条件的顺序将它们插入到此列表顶部的 LOAD_SEGMENT、NOTE_SEGMENT 或 NULL_SEGMENT *mapfile* 指令中。“预定义段” [200]中所讨论的内置段的入口条件放在此列表的最后。因此，mapfile 定义的入口条件将优先于内置规则，而位于命令行最后的 *mapfile* 将优先于开始处的 mapfile。

对于写入到输出目标文件中的每个节，链接编辑器将执行以下步骤，以便将节放入输出段中。

1. 节的属性将与内部入口条件列表中的每个记录进行比较，从列表的开头开始，依次考虑每个入口条件。当入口条件中的每个属性都精确匹配时，则表示匹配，并且不会禁用与该入口条件相关联的段。搜索将在匹配的第一个入口条件处停止，并且节将定向到关联的段。

   如果不与任何入口条件匹配，则将节放置在输出文件的末尾（位于所有其他段之后）。不会为此信息创建任何程序头项。多数不可分配节（例如调试节）都结束于此区域。
2. 当节位于段下时，链接编辑器会检查此段中现有输出节的列表，方式如下：

   如果节属性值与现有输出节的属性值完全匹配，则将此节放置在与该输出节关联的节列表的末尾。

   如果未找到匹配的输出节，则使用要放置的节的属性创建一个新输出节，并将输入节放在新输出节中。此新输出节位于段内具有相同节类型的任何其他输出节之后，如果没有相同类型的其他输出节，则位于段的末尾。

注 - 如果输入节的用户定义的节类型值介于 SHT_LOUSER 和 SHT_HIUSER 之间，则将其视为 PROGBITS 节。没有在 mapfile 中命名此节类型值的方法，但可以使用入口条件中指定的其他属性值（节标志、节名称）重定向这些节。

预定义段和入口条件的 *mapfile* 指令

链接编辑器提供一组预定义的输出段描述符和入口条件，如“预定义段” [200]中所述。链接编辑器已经知道这些节，因此创建这些节无需 *mapfile* 指令。所显示的可用于生成

它们的 *mapfile* 指令仅为提供说明，并作为相对复杂的 `mapfile` 规范的一个示例。可以使用 *mapfile* 段指令修改或扩充这些内置定义。

通常，节到段的分配可在单个段指令中完成。但是，预定义节具有更复杂的要求，即要求按照不同于段在内存中的布局顺序处理这些节的入口条件。为此可以使用两个过程，第一个用于按照所需顺序定义所有段，第二个用于按照实现所需结果的顺序建立入口条件。用户 `mapfile` 很少需要采用此策略。

```
# Predefined segments and entrance criteria for the Oracle Solaris
# link-editor
$mapfile_version 2

# The lrodata and ldata segments only apply to x86-64 objects.
# Establish amd64 as a convenient token for conditional input
$if _ELF64 && _x86
$add amd64
$endif

# Pass 1: Define the segments and their attributes, but
# defer the entrance criteria details to the 2nd pass.
LOAD_SEGMENT text {
        FLAGS = READ EXECUTE;
};
LOAD_SEGMENT data {
        FLAGS = READ WRITE EXECUTE;
};
LOAD_SEGMENT bss {
        DISABLE;
        FLAGS=DATA;
};
$if amd64
LOAD_SEGMENT lrodata {
        FLAGS = READ
};
LOAD_SEGMENT ldata {
        FLAGS = READ WRITE;
};
$endif
NOTE_SEGMENT note;
NULL_SEGMENT extra;

# Pass 2: Define ASSIGN_SECTION attributes for the segments defined
# above, in the order the link-editor should evaluate them.

# All SHT_NOTE sections go to the note segment
NOTE_SEGMENT note {
        ASSIGN_SECTION {
                TYPE = NOTE;
        };
};
$if amd64
# Medium/large model x86-64 readonly sections to lrodata
LOAD_SEGMENT lrodata {
        ASSIGN_SECTION {
```

```
                FLAGS = ALLOC AMD64_LARGE;
        };
};
$endif

# text receives all readonly allocable sections
LOAD_SEGMENT text {
        ASSIGN_SECTION {
                FLAGS = ALLOC !WRITE;
        };
};

# If bss is enabled, it takes the writable NOBITS sections
# that would otherwise end up in ldata or data.
LOAD_SEGMENT bss {
        DISABLE;
        ASSIGN_SECTION {
                FLAGS = ALLOC WRITE;
                TYPE = NOBITS;
        };
};

$if amd64
# Medium/large model x86-64 writable sections to ldata
LOAD_SEGMENT ldata {
        ASSIGN_SECTION {
                FLAGS = ALLOC WRITE AMD64_LARGE;
        };
        ASSIGN_SECTION {
                TYPE = NOBITS;
                FLAGS = AMD64_LARGE
        };
};
$endif

# Any writable allocable sections not taken above go to data
LOAD_SEGMENT data {
        ASSIGN_SECTION {
                FLAGS = ALLOC WRITE;
        };
};

# Any section that makes it to this point ends up at the
# end of the object file in the extra segment. This accounts
# for the bulk of non-allocable sections.
NULL_SEGMENT extra {
        ASSIGN_SECTION;
};
```

♦ ♦ ♦ 第 9 章

9 接口和版本控制

由链接编辑器和运行时链接程序处理的 ELF 目标文件提供了许多其他目标文件可以绑定到的全局符号。这些符号描述了目标文件的应用程序二进制接口 (Application Binary Interface, ABI)。在目标文件演变过程中，此接口可能会由于添加或删除全局符号而发生更改。此外，目标文件演变还可能涉及内部实现更改。

版本控制是一些可以应用于某个目标文件以指示接口与实现更改的技术。这些技术使目标文件的演变过程受到控制，同时又维护向下兼容性。

本章介绍了如何定义目标文件的 ABI。此外，还介绍了对此 ABI 接口的更改影响下兼容性的方式。我们将借助模型来探讨这些概念，说明如何将接口和实现的更改纳入目标文件的新发行版。

本章重点介绍动态可执行文件与共享目标文件的运行时接口。对说明和管理这些动态目标文件内更改的方法只作一般性介绍。

动态目标文件的开发者必须注意接口更改的结果，并了解管理此类更改的方法，尤其是关于维护与以前所发布的目标文件的向下兼容性。

由任何动态目标文件实现可用的全局符号都代表目标文件的公共接口。通常，进行链接编辑之后留在目标文件中的全局符号数多于希望公开的符号数。这些全局符号源于用于创建目标文件的可重定位目标文件之间所需的符号状态。这些符号代表目标文件内部的专用接口。

要定义目标文件的 ABI，应当首先确定目标文件中那些希望使其公开可用的全局符号。可以在最终的链接编辑过程中使用链接编辑器的 `-M` 选项和关联的 `mapfile` 来建立这些公共符号。“缩减符号作用域” [46]中介绍了此技术。此公共接口可在正在创建的目标文件内建立一个或多个版本定义。这些定义构成目标文件演变时添加新接口的基础。

以下各节基于此初始公共接口。不过，首先应了解如何对接口的各种更改进行分类，以便对这些接口进行适当管理。

接口兼容性

可以对目标文件进行许多类型的更改。可以用最简单的术语将这些更改分类为以下两组之一。

- 兼容更新。这些更新中会新增接口，所有先前可用的接口仍保持不变。
- 不兼容更新。这些更新会更改现有接口，使此接口的现有用户操作失败或不正确地执行操作。

下表对一些常见的目标文件更改进行了分类。

表 9-1 接口兼容性示例

目标文件更改	更新类型
添加符号	兼容
删除符号	不兼容
向 *non-variadic* 函数添加参数	不兼容
从函数中删除参数	不兼容
更改函数数据项的大小或内容或将其作为外部定义	不兼容
目标文件的语义属性保持不变时，对函数进行的错误修复或内部增强	兼容
目标文件的语义属性发生更改时对函数进行的错误修复或内部增强	不兼容

注 - 添加符号（实质上是插入）可构成不兼容更新，使得新符号可能与应用程序对此符号的使用产生冲突。但是，由于通常使用源级名称空间管理，因此实际上这种形式的不兼容性非常少见。

可以通过维护所生成的目标文件的内部版本定义来适应兼容更新。可以通过生成具有新的外部版本化名称的新目标文件来适应不兼容更新。通过以上两种版本控制技术，可以选择应用程序的绑定，还可以在运行时验证正确版本绑定。以下各节将更详细地探讨这两种技术。

内部版本控制

动态目标文件可以包含一个或多个与其关联的内部版本定义。每个版本定义通常与一个或多个符号名称关联。一个符号名称只能与一个版本定义关联。但是，一个版本定义可以继承来自其他版本定义的符号。因此，我们有一种结构可以定义所创建的目标文件内的一个或多个独立的或相关的版本定义。当对目标文件进行了新的更改时，可以添加新的版本定义来表示这些更改。

共享目标文件内的版本定义有两项功能。

- 根据版本化共享目标文件生成的动态目标文件可以记录它们对所绑定的版本定义的依赖性。运行时将检验这些版本依赖项，以确保提供相应的接口或功能来正确执行应用程序。
- 动态目标文件可以在其链接编辑期间选择要绑定的共享目标文件版本定义。通过此机制，开发者可以针对接口或功能控制其对共享目标文件的依赖性，以提供最大的灵活性。

创建版本定义

版本定义通常包括符号名称与唯一版本名称之间的关联。这些关联建立在 mapfile 中，并使用链接编辑器的 -M 选项提供给目标文件的最终链接编辑。“缩减符号作用域” [46]一节中介绍了此技术。

任何时候作为 mapfile 指令的一部分指定一个版本名称时，都会创建一个版本定义。在以下示例中，两个源文件进行合并，并配合 mapfile 指令，生成一个定义了公共接口的目标文件。

```
$ cat foo.c
#include    <stdio.h>

extern const char *_foo1;

void foo1()
{
        (void) printf(_foo1);
}
$ cat data.c
const char *_foo1 = "string used by foo1()\n";
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                 # Release X
        global:
                foo1;
        local:
                *;
};
$ cc -c -Kpic foo.c data.c
$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ elfdump -sN.symtab libfoo.so.1 | grep 'foo.$'
    [32]   0x1074c    0x4  OBJT LOCL  H    0 .data      _foo1
    [53]     0x560   0x38  FUNC GLOB  D    0 .text      foo1
```

符号 foo1 是为提供共享目标文件的公共接口而定义的唯一全局符号。特殊的自动缩减指令 "*" 缩减了其他所有全局符号，从而在所生成的目标文件中实现局部绑定。“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]部分对该自动缩减指令进行了介绍。关联的版本名称 SUNW_1.1 导致生成版本定义。因此，共享目标文件的公共接口包括与内部版本定义 SUNW_1.1 关联的全局符号 foo1。

任何时候使用版本定义或自动缩减指令生成目标文件时，都会同时创建一个基版本定义。此基版本使用所生成的目标文件的名称来定义。此基版本用于关联链接编辑器生成的任何保留符号。有关保留符号的列表，请参见“生成输出文件” [50]。

使用 pvs(1) 和 -d 选项可以显示目标文件中包含的版本定义。

```
$ pvs -d libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
```

目标文件 libfoo.so.1 包含一个名为 SUNW_1.1 的内部版本定义和一个名为 libfoo.so.1 的基版本定义。

注 - 使用链接编辑器的 -z noversion 选项，可以由 mapfile 来定向符号缩减，但是会抑制创建版本定义。

目标文件可以从该初始版本定义开始，通过添加新的接口和更新功能不断演变。例如，通过更新源文件 foo.c 和 data.c，可以将新的函数 foo2 及其支持的数据结构添加到目标文件中。

```
$ cat foo.c
#include    <stdio.h>

extern const char *_foo1, *_foo2;

void foo1()
{
        (void) printf(_foo1);
}

void foo2()
{
        (void) printf(_foo2);
}
$ cat data.c
const char *_foo1 = "string used by foo1()\n";
const char *_foo2 = "string used by foo2()\n";
```

可以创建一个新的版本定义 SUNW_1.2 来定义一个新接口，代表符号 foo2。此外，还可以将此新接口定义为继承原始版本定义 SUNW_1.1。

创建此新接口很重要，因为该接口描述了目标文件的演变。这些接口使用户可以验证和选择要绑定的接口。“绑定到版本定义” [215]和“指定版本绑定” [219]部分更详细地介绍了这些概念。

以下示例介绍了创建上述两个接口的 mapfile 指令。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                   # Release X
        global:
                foo1;
        local:
                *;
};

SYMBOL_VERSION SUNW_1.2 {                   # Release X+1
        global:
                foo2;
} SUNW_1.1;
```

```
$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ elfdump -sN.symtab libfoo.so.1 | grep 'foo.$'
    [28]   0x107a4    0x4  OBJT LOCL  H    0 .data      _foo1
    [29]   0x107a8    0x4  OBJT LOCL  H    0 .data      _foo2
    [48]     0x5e8   0x20  FUNC GLOB  D    0 .text      foo1
    [51]     0x618   0x20  FUNC GLOB  D    0 .text      foo2
```

符号 foo1 和 foo2 均定义为共享目标文件公共接口的一部分。但是，每个符号指定给了一个不同的版本定义。foo1 指定给了版本 SUNW_1.1。foo2 指定给了版本 SUNW_1.2。

使用 pvs(1) 以及 -d、-v 和 -s 选项可以显示这些版本定义及其继承性和符号关联。

```
$ pvs -dsv libfoo.so.1
       libfoo.so.1:
               _end;
               _GLOBAL_OFFSET_TABLE_;
               _DYNAMIC;
               _edata;
               _PROCEDURE_LINKAGE_TABLE_;
               _etext;
       SUNW_1.1:
               foo1;
               SUNW_1.1;
       SUNW_1.2:               {SUNW_1.1}:
               foo2;
               SUNW_1.2
```

版本定义 SUNW_1.2 具有一个对版本定义 SUNW_1.1 的依赖项。

不同版本定义之间的继承是一项很有用的技术。这种继承可以减少任何绑定到版本依赖项的目标文件最终记录的版本信息。“绑定到版本定义” [215]一节对版本继承进行了更详细的介绍。

版本定义符号创建后会与版本定义相关联。在上面的 pvs(1) 示例中，使用 -v 选项时，将显示这些符号。

创建弱版本定义

对于不需要引入新接口定义的目标文件内部更改，可以通过创建弱版本定义来定义。例如，这类更改可以是错误修复或性能改进。这类版本定义是空的，版本定义没有任何全局接口符号与之关联。

例如，假定之前示例中使用的数据文件 data.c 更新后提供更为详细的字符串定义。

```
$ cat data.c
const char *_foo1 = "string used by function foo1()\n";
const char *_foo2 = "string used by function foo2()\n";
```

这时便可以引入一个弱版本定义来标识此项更改。

```
$ cat mapfile
```

```
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                   # Release X
        global:
                foo1;
        local:
                *;
};

SYMBOL_VERSION SUNW_1.2 {                   # Release X+1
        global:
                foo2;
} SUNW_1.1;

SYMBOL_VERSION SUNW_1.2.1 { } SUNW_1.2;     # Release X+2

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:               {SUNW_1.1};
        SUNW_1.2.1 [WEAK]:      {SUNW_1.2};
```

空版本定义由弱标签表示。利用这些弱版本定义，应用程序可以验证某个特定的实现详细信息是否存在。应用程序可以绑定到与其要求的实现详细信息相关联的版本定义。“绑定到版本定义” [215]一节更详细地说明了如何使用这些定义。

定义不相关接口

上面的示例说明了添加到目标文件的新版本定义如何继承任何现有的版本定义。您还可以创建具有唯一性的、独立的版本定义。在以下示例中，两个新文件 `bar1.c` 和 `bar2.c` 添加到目标文件 `libfoo.so.1`。这两个文件分别提供了两个新符号：`bar1` 和 `bar2`。

```
$ cat bar1.c
extern void foo1();

void bar1()
{
        foo1();
}
$ cat bar2.c
extern void foo2();

void bar2()
{
        foo2();
}
```

这两个新符号旨在定义两个新的公共接口。这些新接口彼此并不相关。但是，每个接口均表现出对原始 `SUNW_1.2` 接口的依赖性。

以下 `mapfile` 定义创建了必要的关联。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                   # Release X
        global:
                foo1;
        local:
                *;
};

SYMBOL_VERSION SUNW_1.2 {                   # Release X+1
        global:
                foo2;
} SUNW_1.1;

SYMBOL_VERSION SUNW_1.2.1 { } SUNW_1.2;     # Release X+2

SYMBOL_VERSION SUNW_1.3a {                  # Release X+3
        global:
                bar1;
} SUNW_1.2;

SYMBOL_VERSION SUNW_1.3b {                  # Release X+3
        global:
                bar2;
} SUNW_1.2;
```

使用此 `mapfile` 时在 `libfoo.so.1` 中创建的版本定义及其相关的依赖项可以使用 pvs(1) 进行检查。

```
$ cc -o libfoo.so.1 -M mapfile -G foo.o bar1.o bar2.o data.o
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:               {SUNW_1.1};
        SUNW_1.2.1 [WEAK]:      {SUNW_1.2};
        SUNW_1.3a:              {SUNW_1.2};
        SUNW_1.3b:              {SUNW_1.2};
```

版本定义可用于验证运行时绑定要求，还可以用于控制目标文件创建期间的目标文件绑定。以下各节将更详细地探讨这些版本定义的用法。

绑定到版本定义

根据其他共享目标文件生成动态可执行文件或共享目标文件时，会在生成的目标文件中记录这些依赖项。有关更多详细信息，请参见“共享目标文件处理” [28]和“记录共享目标文件名称” [124]。如果依赖项还包含版本定义，则会在所生成的目标文件中记录关联的版本依赖项。

以下示例使用上一节中的数据文件生成一个适用于编译时环境的共享目标文件 `libfoo.so.1`。

```
$ cc -o libfoo.so.1 -h libfoo.so.1 -M mapfile -G foo.o bar.o data.o
$ ln -s libfoo.so.1 libfoo.so
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
                SUNW_1.1;
        SUNW_1.2:               {SUNW_1.1}:
                foo2;
                SUNW_1.2;
        SUNW_1.2.1 [WEAK]:      {SUNW_1.2}:
                SUNW_1.2.1;
        SUNW_1.3a:              {SUNW_1.2}:
                bar1;
                SUNW_1.3a;
        SUNW_1.3b:              {SUNW_1.2}:
                bar2;
                SUNW_1.3b
```

共享目标文件 libfoo.so.1 提供六个公共接口。其中 SUNW_1.1、SUNW_1.2、SUNW_1.3a 以及 SUNW_1.3b 这四个接口定义了导出的符号名称。接口 SUNW_1.2.1 描述了目标文件的一个内部实现更改。接口 libfoo.so.1 定义了若干保留标签。使用 libfoo.so.1 作为依赖项创建的动态目标文件记录了动态目标文件绑定到的接口的版本名称。

以下示例创建了一个引用 foo1 和 foo2 符号的应用程序。使用 pvs(1) 及 -r 选项可以检查该应用程序中记录的版本控制依赖项信息。

```
$ cat prog.c
extern void foo1();
extern void foo2();

main()
{
        foo1();
        foo2();
}
$ cc -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.2, SUNW_1.2.1);
```

在本示例中，应用程序 prog 绑定到两个接口：SUNW_1.1 和 SUNW_1.2。这两个接口分别提供全局符号 foo1 和 foo2。

因为版本定义 SUNW_1.1 在 libfoo.so.1 中定义为由版本定义 SUNW_1.2 继承，所以您只需记录一个依赖项。这种继承是为了实现版本定义依赖项的标准化。这种标准化可以减少目标文件中保留的版本信息量，并减少运行时需要的版本验证处理。

因为应用程序 prog 是根据包含弱版本定义 SUNW_1.2.1 的共享目标文件实现而生成的，所以也记录了该依赖项。虽然此版本定义被定义为继承版本定义 SUNW_1.2，但是其弱版本的性质阻止了其使用 SUNW_1.1 进行标准化。弱版本定义会导致单独记录依赖项的情况。

如果有多个弱版本定义彼此继承，则这些定义将以与非弱版本定义一样的方式进行标准化。

注 - 可以使用链接编辑器的 -z noversion 选项来抑制记录版本依赖项。

在执行应用程序时，运行时链接程序会验证是否存在任何来自绑定到的目标文件的已记录版本定义。使用 ldd(1) 及 -v 选项可以显示此验证。例如，通过对应用程序 prog 运行 ldd(1)，显示出在依赖项 libfoo.so.1 中正确发现版本定义依赖项。

```
$ ldd -v prog

    find object=libfoo.so.1; required by prog
        libfoo.so.1 =>   ./libfoo.so.1
    find version=libfoo.so.1;
        libfoo.so.1 (SUNW_1.2) =>           ./libfoo.so.1
        libfoo.so.1 (SUNW_1.2.1) =>         ./libfoo.so.1
    ....
```

注 - 在 ldd(1) 中使用 -v 选项可以输出详细结果。将生成一份包含所有依赖项和所有版本控制要求的递归列表。

如果找不到非弱版本定义依赖项，应用程序初始化期间会发生致命错误。将在不给出任何提示的情况下忽略任何无法找到的弱版本定义依赖项。例如，如果在应用程序 prog 运行的环境中，libfoo.so.1 仅包含版本定义 SUNW_1.1，则会发生以下致命错误。

```
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
$ prog
ld.so.1: prog: fatal: libfoo.so.1: version 'SUNW_1.2' not \
    found (required by file prog)
```

如果 prog 未记录任何版本定义依赖项，则符号 foo2 的缺失会导致运行时的致命重定位错误。此重定位错误可能在进程初始化或进程执行期间发生。如果应用程序的执行路径不调用函数 foo2，则错误情况可能根本不会发生。请参见“重定位错误” [95]。

版本定义依赖项也可以即时指出应用程序所需接口的可用性。

例如，prog 可能运行在这样一个环境中，在该环境中，libfoo.so.1 仅包含版本定义 SUNW_1.1 和 SUNW_1.2 。在这种情况下，所有非弱版本定义的要求均得到满足。缺少弱版本定义 SUNW_1.2.1 被认为是非致命的。在这种情况下，不会出现运行时错误。

```
$ pvs -dv libfoo.so.1
```

```
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:               {SUNW_1.1};
$ prog
string used by foo1()
string used by foo2()
```

使用 ldd(1) 可以显示所有无法找到的版本定义。

```
$ ldd prog
        libfoo.so.1 =>   ./libfoo.so.1
        libfoo.so.1 (SUNW_1.2.1) =>          (version not found)
        ....
```

在运行时，如果依赖项的实现不包含版本定义信息，则会在不给出任何提示的情况下忽略该依赖项的任何版本验证。在从非版本化的共享目标文件过渡到版本化的共享目标文件时，此策略可以提供一定程度的向下兼容性。ldd(1) 总是可以用于显示任何版本要求差异。

注 - 环境变量 LD_NOVERSION 可以用于抑制所有运行时版本控制验证。

验证新增目标文件中的版本

版本定义符号还提供一种机制，用于验证 dlopen(3C) 所获取的目标文件的版本要求。使用 dlopen(3C) 添加到进程地址空间的目标文件不会自动进行版本依赖项验证。因此，dlopen(3C) 的调用者负责验证是否符合任何版本控制要求。

通过使用 dlsym(3C) 查找关联的版本定义符号，可以验证某个需要的版本定义是否存在。以下示例使用 dlopen(3C) 将共享目标文件 libfoo.so.1 添加到一个进程。然后验证接口 SUNW_1.2 的可用性。

```
#include    <stdio.h>
#include    <dlfcn.h>

main()
{
        void       *handle;
        const char *file = "libfoo.so.1";
        const char *vers = "SUNW_1.2";
        ....

        if ((handle = dlopen(file, (RTLD_LAZY | RTLD_FIRST))) == NULL) {
                (void) printf("dlopen: %s\n", dlerror());
                return (1);
        }

        if (dlsym(handle, vers) == NULL) {
```

```
        (void) printf("fatal: %s: version '%s' not found\n", file, vers);
        return (1);
}
....
```

注 - 使用 dlopen(3C) 的标志 RTLD_FIRST 可以确保将 dlsym(3C) 搜索限制在 libfoo.so.1。

指定版本绑定

创建动态目标文件时，如果该动态目标文件基于包含版本定义的共享目标文件进行链接，则可以指示链接编辑器将绑定仅限于特定版本定义。实际上，使用链接编辑器可以控制目标文件到特定接口的绑定。

使用 DEPEND_VERSIONS *mapfile* 指令可以控制目标文件的绑定要求。此指令通过链接编辑器的 -M 选项和关联的 mapfile 来提供。DEPEND_VERSIONS 指令使用以下语法。

```
$mapfile_version 2
DEPEND_VERSIONS objname {
        ALLOW   = version_name;
        REQUIRE = version_name;
        ....
};
```

- *objname* 代表共享目标文件依赖项的名称。此名称应当与链接编辑器所使用的共享目标文件编译环境名称一致。请参见“库命名约定” [29]。
- ALLOW 属性用于指定共享目标文件内应设为可用于绑定的版本定义名称。可以指定多个 ALLOW 属性。
- REQUIRE 属性允许记录新增的版本定义。可以指定多个 REQUIRE 属性。

在以下情况中，版本绑定的控制非常实用。

- 当共享目标文件定义了独立的、具有唯一性的版本时。在定义不同标准接口时可以使用这种版本控制方法。可以使用绑定控制生成目标文件，以确保目标文件仅绑定到特定的接口。
- 当共享目标文件经过了若干软件发行版的版本化时。可以使用绑定控制生成目标文件，以将其绑定限定到之前的软件发行版中可用的接口。这样，目标文件在使用最新发行版的共享目标文件生成之后，仍可以与旧发行版的共享目标文件依赖项一起运行。

以下示例说明了版本控制机制的使用。此示例使用共享目标文件 libfoo.so.1，其中包含以下版本接口定义。

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
```

```
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
                foo2;
                SUNW_1.1;
        SUNW_1.2:           {SUNW_1.1}:
                bar;
```

版本定义 SUNW_1.1 和 SUNW_1.2 在 libfoo.so.1 中分别代表软件 Release X 和 Release X+1 中设为可用的接口。

使用以下版本控制 mapfile 指令，可以将应用程序生成为仅绑定到 Release X 中可用的接口。

```
$ cat mapfile
$mapfile_version 2
DEPEND_VERSIONS libfoo.so {
        ALLOW = SUNW_1.1;
}
```

例如，假定您开发一个应用程序 prog，并希望确保该应用程序可以运行于 Release X。该应用程序必须仅使用 Release X 中可用的接口。如果应用程序错误地引用了符号 bar，则该应用程序将无法符合所要求的接口。发生这种情况时，链接编辑器会提示未定义的符号错误。

```
$ cat prog.c
extern void foo1();
extern void bar();

main()
{
        foo1();
        bar();
}
$ cc -o prog prog.c -M mapfile -L. -R. -lfoo
Undefined           first referenced
 symbol                 in file
bar                     prog.o  (symbol belongs to unavailable \
                        version ./libfoo.so (SUNW_1.2))
ld: fatal: symbol referencing errors
```

要符合 SUNW_1.1 接口，必须删除对 bar 的引用。您可以重新编写应用程序，删除对 bar 的要求，或向应用程序的创建添加一个 bar 的实现。

注 - 缺省情况下，还会对照任何文件控制指令验证在链接编辑过程中遇到的共享目标文件依赖项。使用环境变量 LD_NOVERSION 可以抑制对任何共享目标文件依赖项进行版本验证。

绑定到额外的版本定义

要记录目标文件的正常符号绑定可能生成的更多版本依赖项，可以将 REQUIRE 属性用于 DEPEND_VERSIONS *mapfiile* 指令。以下各节说明了可以使用这种额外绑定的情形。

重新定义接口

情形之一是，将特定于 ISV 的接口转换为公共标准接口。

在之前的 libfoo.so.1 示例中，假定在 Release X+2 中，版本定义 SUNW_1.1 细分为两个标准发行版：STAND_A 和 STAND_B。要保持兼容性，必须保留 SUNW_1.1 版本定义。在此示例中，该版本定义表现为继承两个标准定义。

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:           {STAND_A, STAND_B}:
                SUNW_1.1;
        SUNW_1.2:           {SUNW_1.1}:
                bar;
        STAND_A:
                foo1;
                STAND_A;
        STAND_B:
                foo2;
                STAND_B;
```

如果应用程序 prog 的唯一要求是接口符号 foo1，则该应用程序将仅具有一个对版本定义 STAND_A 的依赖项。这将使 prog 无法运行在 libfoo.so.1 小于 Release X+2 的系统上。版本定义 STAND_A 在之前的发行版中不存在，尽管接口 foo1 在之前的发行版中存在。

通过创建对 SUNW_1.1 的依赖项，可以生成应用程序 prog，使其要求与之前的发行版一致。

```
$ cat mapfile
$mapfile_version 2
DEPEND_VERSIONS libfoo.so {
        ALLOW = SUNW_1.1;
        REQUIRE = SUNW_1.1;
};
$ cat prog
extern void foo1();

main()
```

```
{
        foo1();
}
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.1);
```

此显式依赖项足以封装真实的依赖项要求。此依赖项可以满足与旧发行板的兼容性。

绑定到弱版本

“创建弱版本定义” [213]描述了如何使用弱版本定义标记内部实现更改。这些版本定义非常适合指示对目标文件所做的错误修复和性能改进。如果弱版本的存在是必需的，可以生成一个对此版本定义的显式依赖项。当某项错误修复或性能改进对于目标文件的正常工作至关重要时，创建这类依赖项是非常重要的。

在之前的 `libfoo.so.1` 示例中，假定一项错误修复作为弱版本定义 `SUNW_1.2.1` 纳入软件 `Release X+3` 中：

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:           {STAND_A, STAND_B}:
                SUNW_1.1;
        SUNW_1.2:           {SUNW_1.1}:
                bar;
        STAND_A:
                foo1;
                STAND_A;
        STAND_B:
                foo2;
                STAND_B;
        SUNW_1.2.1 [WEAK]:  {SUNW_1.2}:
                SUNW_1.2.1;
```

通常，如果应用程序依据此 `libfoo.so.1` 进行生成，将记录一个对版本定义 `SUNW_1.2.1` 的弱依赖项。此依赖项仅用于提供信息。如果在运行时使用的 `libfoo.so.1` 的实现中不存在版本定义，此依赖项不会导致应用程序终止。

使用 `DEPEND_VERSIONS` *mapfile* 指令的 `REQUIRE` 属性可以生成对版本定义的显式依赖项。如果此定义是一个弱定义，则此显式引用同时也是要提升为强依赖项的版本定义。

使用以下文件控制指令，可以将应用程序 `prog` 生成为强制要求 `SUNW_1.2.1` 接口在运行时可用。

```
$ cat mapfile
```

```
$mapfile_version 2
DEPEND_VERSIONS libfoo.so {
        ALLOW = SUNW_1.1;
        REQUIRE = SUNW_1.2.1;
};
$ cat prog
extern void foo1();

main()
{
        foo1();
}
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.2.1);
```

prog 具有一个对接口 STAND_A 的显式依赖项。因为版本定义 SUNW_1.2.1 提升为强版本，所以版本 SUNW_1.2.1 使用依赖项 STAND_A 进行标准化。在运行时，如果无法找到版本定义 SUNW_1.2.1，将生成一条致命错误。

注 - 当处理的依赖项数量较少时，可以使用链接编辑器的 -u 选项显式绑定到版本定义。使用此选项可以引用版本定义符号。但是，符号引用是不可选择的。当处理多个包含类似命名的版本定义的依赖项时，使用此方法可能不足以创建显式绑定。

版本稳定性

已介绍了用于绑定到目标文件中的版本定义的不同模型。这些模型允许对接口要求进行运行时验证。仅当各个版本定义在目标文件的生命周期内保持不变时，此验证才能保持有效。

可以为一个目标文件创建一个版本定义，以供其他目标文件进行绑定。此版本定义必须在后续的目标文件发行版中一直存在。版本名称及与该版本关联的符号均必须保持不变。要强制执行这些要求，版本定义中定义的符号名称的通配符扩展将不受支持。与通配符匹配的符号的数量在目标文件的演变过程中可能会有所不同。这种差异可能会导致意外的接口不稳定性。

可重定位目标文件

之前的各节说明了如何在动态目标文件中记录版本信息。可重定位的目标文件可以按类似的方式保留版本控制信息。但是，在如何使用此信息方面，存在着细微的差别。

任何提供给可重定位目标文件的链接编辑的版本定义都记录在该目标文件中。这些定义采取与动态目标文件中记录的版本定义相同的格式。但是，缺省情况下，不会对所创建的可重定位目标文件执行符号缩减。当可重定位目标文件用于创建动态目标文件时，由版本控制信息定义的符号缩减将应用到该目标文件。

此外，可重定位目标文件中找到的任何版本定义都将传播到动态目标文件。有关在可重定位目标文件中进行版本处理的示例，请参见“缩减符号作用域” [46]。

注 - 使用链接编辑器的 -B reduce 选项可以将版本定义所暗示的符号缩减应用于可重定位目标文件。

外部版本控制

对共享目标文件的运行时引用应始终引用版本化的文件名。版本化的文件名通常表示为文件名加版本号后缀。

如果共享目标文件的接口以不兼容的方式更改，会导致旧的应用程序中断。在这种情况下，应当使用新的版本化文件名分发新的共享目标文件。另外，还必须分发原始的版本化文件名，以提供旧的应用程序所需要的接口。

在跨一系列软件发行版生成应用程序时，应当在运行时环境中以单独的版本化文件名提供共享目标文件。这样可以保证生成应用程序所依据的接口在应用程序执行期间可用于绑定。

以下各节介绍了如何在编译和运行时环境之间协调接口的绑定。

协调版本化文件名

链接编辑通常使用链接编辑器的 -l 选项引用共享目标文件依赖项。此选项使用链接编辑器的库搜索机制定位前缀为 lib、后缀为 .so 的共享目标文件。

但是，在运行时，任何共享目标文件依赖项都应当作为版本化文件名存在。我们不保留两个遵循不同命名约定的不同共享目标文件，而是在两个文件名之间创建文件系统链接。

例如，可以使用符号链接将共享目标文件 libfoo.so.1 设为可用于编译环境。编译文件名是指向运行时文件名的一个符号链接。

```
$ cc -o libfoo.so.1 -G -K pic foo.c
$ ln -s libfoo.so.1 libfoo.so
$ ls -l libfoo*
lrwxrwxrwx  1 usr grp          11 1991 libfoo.so -> libfoo.so.1
-rwxrwxr-x  1 usr grp        3136 1991 libfoo.so.1
```

符号链接和硬链接均可使用。但是，作为一种文档和诊断的辅助手段，符号链接更为实用。

已经为运行时环境生成了共享目标文件 libfoo.so.1。符号链接 libfoo.so 还使该文件可以用在编译环境中。

```
$ cc -o prog main.o -L. -lfoo
```

链接编辑器使用共享目标文件 libfoo.so.1 描述的接口处理可重定位目标文件 main.o，按照符号链接 libfoo.so 可以找到该共享目标文件。

经过一系列软件发行版，新版本的 libfoo.so 可以使用已更改的接口进行分发。通过更改符号链接，可以将编译环境构造为使用适用的接口。

```
$ ls -l libfoo*
lrwxrwxrwx  1 usr grp          11 1993 libfoo.so -> libfoo.so.3
-rwxrwxr-x  1 usr grp        3136 1991 libfoo.so.1
-rwxrwxr-x  1 usr grp        3237 1992 libfoo.so.2
-rwxrwxr-x  1 usr grp        3554 1993 libfoo.so.3
```

在此示例中，有三个主要的共享目标文件版本可用。libfoo.so.1 和 libfoo.so.2 这两个版本提供现有应用程序的依赖项。libfoo.so.3 提供用于创建和运行新应用程序的最新主要发行版。

仅仅使用这种符号链接机制，并不足以协调编译共享目标文件与运行时版本化文件名。如示例所体现的，链接编辑器将其处理的共享目标文件的文件名记录在动态可执行文件 prog 中。在这种情况下，链接编辑器看到的文件名是编译环境文件。

```
$ elfdump -d prog | grep libfoo
     [0]  NEEDED       0x1b7           libfoo.so
```

当执行应用程序 prog 时，运行时链接器会搜索依赖项 libfoo.so。prog 会绑定到该符号链接所指向的文件。

为了确保将正确的运行时名称记录为依赖项，应当使用 *soname* 定义来生成共享目标文件 libfoo.so.1。此定义可以识别共享目标文件的运行时名称。此名称将作为依赖项名称，供任何依据该共享目标文件进行链接的目标文件使用。在创建共享目标文件期间，可以使用 -h 选型提供此定义。

```
$ cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 foo.c
$ ln -s libfoo.so.1 libfoo.so
$ cc -o prog main.o -L. -lfoo
$ elfdump -d prog | grep libfoog
     [0]  NEEDED       0x1b7           libfoo.so.1
```

此符号链接和 *soname* 机制在编译环境和运行时环境的共享目标文件命名约定之间建立了一种强健的协调方式。链接编辑期间处理的接口将准确记录在所生成的输出文件中。这种记录功能可以确保在运行时提供所需要的接口。

同一进程中的多个外部版本化文件

创建新的外部版本化共享目标文件属于一项重大更改。请确保您了解任何使用其中一个外部版本化共享目标文件的进程的完整依赖项。

例如，某个应用程序可能具有一个对 libfoo.so.1 的依赖项和一个外部提供的目标文件 libISV.so.1。后者也可能具有对 libfoo.so.1 的依赖项。应用程序可以重新设计，以使用 libfoo.so.2 中的新接口。但是，应用程序可能无法更改对外部目标文件 libISV.so.1 的使用。根据运行时装入的 libfoo.so 实现的可见性作用域，文件的两个主要版本均可引入正在运行的进程。更改 libfoo.so 的版本的唯一原因是要标记一项不兼容的更改。因此，在一个进程内使用目标文件的两个版本会导致不正确的符号绑定，并因此而产生意外的交互。

应当避免不兼容的接口更改。仅当您可以完全控制接口定义和引用此定义的所有目标文件时，才可以考虑进行不兼容的更改。

♦ ♦ ♦ 第 10 章

10 使用动态字符串标记建立依赖性

动态目标文件可以显式建立依赖性，也可以通过过滤器建立依赖性。其中每一种机制都可以用运行路径来扩展，该路径指示运行时链接程序搜索并装入所需依赖项。用于记录过滤器、依赖项和运行路径信息的字符串名称可以用以下保留的动态字符串标记来扩展：

- `$CAPABILITY` (`$HWCAP`)
- `$ISALIST`
- `$OSNAME`、`$OSREL`、`$PLATFORM` 和 `$MACHINE`
- `$ORIGIN`

以下各节提供了如何使用这些标记的示例。

特定于功能的共享目标文件

动态标记 `$CAPABILITY` 可用于指定特定于功能的共享目标文件所在的目录。此标记可用于过滤器和依赖项。由于此标记可以扩展到多个目标文件，因此它与依赖项一起使用时应受到控制。通过 `dlopen(3C)` 获取的依赖项可以在 `RTLD_FIRST` 模式下使用此标记。使用此标记的显式依赖项将装入找到的第一个适当的依赖项。

注 - 原始功能实现完全基于硬件功能。标记 `$HWCAP` 用于选择此功能处理。自从此功能被扩展超出硬件功能后，`$HWCAP` 标记也已经被 `$CAPABILITY` 标记取代。出于兼容性方面的考虑，`$HWCAP` 标记被解释为 `$CAPABILITY` 标记的别名。

路径名称指定必须包含以 `$CAPABILITY` 标记结束的完整路径名。由 `$CAPABILITY` 标记指定的目录中的共享目标文件将在运行时受到检查。这些目标文件应指明其功能要求。请参见“标识功能要求” [51]。将根据可用于此进程的功能验证每个目标文件。适用于进程的那些目标文件按其功能值的降序进行排序。这些已排序的 *filtee* 用于解析过滤器内定义的符号。

功能目录中的 *filtee* 在命名方面没有限制。以下示例说明了如何设计辅助过滤器 `libfoo.so.1` 以使其访问硬件功能 *filtee*。

```
$ LD_OPTIONS='-f /opt/ISV/lib/cap/$CAPABILITY' \
```

```
    cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ elfdump -d libfoo.so.1 | egrep 'SONAME|AUXILIARY'
     [2]  SONAME             0x1                libfoo.so.1
     [3]  AUXILIARY          0x96               /opt/ISV/lib/cap/$CAPABILITY
$ elfdump -H /opt/ISV/lib/cap/*

/opt/ISV/lib/cap/filtee.so.3:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
   index  tag               value
     [0]  CA_SUNW_HW_1     0x1000  [ SSE2 ]

/opt/ISV/lib/cap/filtee.so.1:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
   index  tag               value
     [0]  CA_SUNW_HW_1     0x40  [ MMX ]

/opt/ISV/lib/cap/filtee.so.2:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
   index  tag               value
     [0]  CA_SUNW_HW_1     0x800  [ SSE ]
```

如果在具有 MMX 和 SSE 硬件功能的系统上处理过滤器 libfoo.so.1，则出现以下 *filtee* 搜索顺序。

```
$ cc -o prog prog.c -R. -lfoo
$ LD_DEBUG=symbols prog
....
01233: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]
01233: symbol=foo;  lookup in file=cap/filtee.so.2  [ ELF ]
01233: symbol=foo;  lookup in file=cap/filtee.so.1  [ ELF ]
....
```

请注意，filtee.so.2 的功能值大于 filtee.so.1 的功能值。由于 SSE2 功能不可用，因此 filtee.so.3 不会包括在符号搜索中。

减少 *filtee* 搜索

通过在过滤器内使用 $CAPABILITY，可使一个或多个 *filtee* 实现过滤器内定义的接口。

指定的 $CAPABILITY 目录中的所有共享目标文件都会被检查，以验证其可用性并对找到的那些适用于进程的目标文件进行排序。排序后，将装入所有目标文件以备使用。

可以使用链接编辑器的 -z endfiltee 选项生成 *filtee*，以指明它是最后一个可用的 *filtee*。使用此选项标识的 *filtee* 将终止此过滤器的已排序 *filtee* 列表。不会为过滤器装入任何排在此 *filtee* 之后的目标文件。在前面的示例中，如果使用 -z endfiltee 标记了 filter.so.2 *filtee*，则 *filtee* 搜索将如下所示：

```
$ LD_DEBUG=symbols prog
....
01424: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]
01424: symbol=foo;  lookup in file=cap/filtee.so.2  [ ELF ]
....
```

特定于指令集的共享目标文件

将在运行时扩展动态标记 $ISALIST，以反映可在此平台上执行的本机指令集，如实用程序 isalist(1) 所示。此标记可用于过滤器、运行路径定义和依赖项。由于此标记可以扩展到多个目标文件，因此它与依赖项一起使用时应受到控制。通过 dlopen(3C) 获取的依赖项可以在 RTLD_FIRST 模式下使用此标记。使用此标记的显式依赖项将装入找到的第一个适当的依赖项。

注 - 该标记已废弃，以后的 Oracle Solaris 版本中将不再包括该标记。有关处理指令集扩展名的建议技术，请参见“特定于功能的共享目标文件” [227]。

引入 $ISALIST 标记的任何字符串名称将有效地复制到多个字符串中。并且会为每个字符串指定一个可用的指令集。

以下示例说明了如何设计辅助过滤器 libfoo.so.1 以使其访问特定于指令集的 *filtee* libbar.so.1。

```
$ LD_OPTIONS='-f /opt/ISV/lib/$ISALIST/libbar.so.1' \
    cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ elfdump -d libfoo.so.1 | egrep 'SONAME|AUXILIARY'
     [2]  SONAME            0x1               libfoo.so.1
     [3]  AUXILIARY         0x96              /opt/ISV/lib/$ISALIST/libbar.so.1
```

或者，也可以使用运行路径。

```
$ LD_OPTIONS='-f libbar.so.1' \
    cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R'/opt/ISV/lib/$ISALIST' foo.c
$ elfdump -d libfoo.so.1 | egrep 'RUNPATH|AUXILIARY'
     [3]  AUXILIARY         0x96              libbar.so.1
     [4]  RUNPATH           0xa2              /opt/ISV/lib/$ISALIST
```

在这两种情况下，运行时链接程序均使用平台上可用的指令列表来构造多个搜索路径。例如，以下应用程序依赖于 libfoo.so.1，并且在 SUNW,Ultra-2 上执行：

```
$ ldd -ls prog
....
  find object=libbar.so.1; required by ./libfoo.so.1
    search path=/opt/ISV/lib/$ISALIST  (RPATH from file ./libfoo.so.1)
      trying path=/opt/ISV/lib/sparcv9+vis/libbar.so.1
      trying path=/opt/ISV/lib/sparcv9/libbar.so.1
      trying path=/opt/ISV/lib/sparcv8plus+vis/libbar.so.1
      trying path=/opt/ISV/lib/sparcv8plus/libbar.so.1
      trying path=/opt/ISV/lib/sparcv8/libbar.so.1
      trying path=/opt/ISV/lib/sparcv8-fsmuld/libbar.so.1
      trying path=/opt/ISV/lib/sparcv7/libbar.so.1
      trying path=/opt/ISV/lib/sparc/libbar.so.1
```

或者，在配置了 `MMX` 的 `Pentium Pro` 上执行具有类似依赖项的应用程序：

```
$ ldd -ls prog
....
  find object=libbar.so.1; required by ./libfoo.so.1
    search path=/opt/ISV/lib/$ISALIST  (RPATH from file ./libfoo.so.1)
      trying path=/opt/ISV/lib/pentium_pro+mmx/libbar.so.1
      trying path=/opt/ISV/lib/pentium_pro/libbar.so.1
      trying path=/opt/ISV/lib/pentium+mmx/libbar.so.1
      trying path=/opt/ISV/lib/pentium/libbar.so.1
      trying path=/opt/ISV/lib/i486/libbar.so.1
      trying path=/opt/ISV/lib/i386/libbar.so.1
      trying path=/opt/ISV/lib/i86/libbar.so.1
```

减少 *filtee* 搜索

通过在过滤器内使用 `$ISALIST`，可使一个或多个 *filtee* 实现过滤器内定义的接口。

过滤器内定义的任何接口都可能导致全面搜索所有可能的 *filtee*，以尝试找到所需接口。如果使用 *filtee* 以提供性能关键的功能，则这种全面的 *filtee* 搜索可能会对效率带来负面影响。

可以使用链接编辑器的 `-z endfiltee` 选项生成 *filtee*，以指明它是最后一个可用的 *filtee*。此选项将终止该过滤器的任何进一步 *filtee* 搜索。在前面的 SPARC 示例中，如果存在 SPARCV9 *filtee*，并且它使用了 `-z endfiltee` 标记，则 *filtee* 搜索将如下所示：

```
$ ldd -ls prog
....
  find object=libbar.so.1; required by ./libfoo.so.1
    search path=/opt/ISV/lib/$ISALIST  (RPATH from file ./libfoo.so.1)
      trying path=/opt/ISV/lib/sparcv9+vis/libbar.so.1
      trying path=/opt/ISV/lib/sparcv9/libbar.so.1
```

特定于系统的共享目标文件

将在运行时扩展动态标记 $OSNAME、$OSREL、$PLATFORM 和 $MACHINE，以提供特定于系统的信息。这些标记可用于过滤器、运行路径或依赖项定义。

将扩展 $OSNAME 以反映操作系统的名称，如组合使用实用程序 uname(1) 和 -s 选项时所示。将扩展 $OSREL 以反映操作系统的发行版级别，如 uname -r 所示。将扩展 $PLATFORM 以反映底层平台名称，如 uname -i 所示。将扩展 $MACHINE 以反映底层计算机硬件名称，如 uname -m 所示。

以下示例说明了如何设计辅助过滤器 libfoo.so.1 以使其访问特定于平台的 *filtee* libbar.so.1。

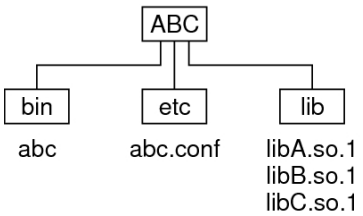
```
$ LD_OPTIONS='-f /platform/$PLATFORM/lib/libbar.so.1' \
    cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ elfdump -d libfoo.so.1 | egrep 'SONAME|AUXILIARY'
     [2]  SONAME            0x1               libfoo.so.1
     [3]  AUXILIARY         0x96              /platform/$PLATFORM/lib/libbar.so.1
```

此机制在 Oracle Solaris OS 中用于提供特定于平台的共享目标文件 /lib/libc.so.1 的扩展。

查找关联的依赖项

通常，非绑定产品用于在唯一的位置上安装。此产品由二进制文件、共享目标文件依赖项和关联的配置文件组成。例如，非绑定产品 ABC 可能具有下图所示的布局。

图 10-1 非绑定依赖项



假定此产品适用于安装在 /opt 下。通常，您会向 PATH 中增加 /opt/ABC/bin，以定位产品的二进制代码。每个二进制代码均使用硬编码的运行路径在二进制代码内查找其依赖项。对于应用程序 abc，此运行路径将如下所示：

```
$ cc -o abc abc.c -R/opt/ABC/lib -L/opt/ABC/lib -lA
$ elfdump -d abc | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED           0x1b5               libA.so.1
     ....
     [4]  RUNPATH          0x1bf               /opt/ABC/lib
```

类似地，对于依赖项 libA.so.1，此运行路径将如下所示：

```
$ cc -o libA.so.1 -G -Kpic A.c -R/opt/ABC/lib -L/opt/ABC/lib -lB
$ elfdump -d libA.so.1 | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED           0x96                libB.so.1
     [4]  RUNPATH          0xa0                /opt/ABC/lib
```

此依赖项表示法将一直有效，直到将产品安装到除建议的缺省目录之外的某个目录中。

动态标记 $ORIGIN 扩展到目标文件的原始目录中。此标记可用于过滤器、运行路径或依赖项定义。可以使用此技术重新定义非绑定应用程序，根据 $ORIGIN 查找其依赖项：

```
$ cc -o abc abc.c '-R$ORIGIN/../lib' -L/opt/ABC/lib -lA
$ elfdump -d abc | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED           0x1b5               libA.so.1
     ....
     [4]  RUNPATH          0x1bf               $ORIGIN/../lib
```

而依赖项 libA.so.1 也可以根据 $ORIGIN 进行定义：

```
$ cc -o libA.so.1 -G -Kpic A.c '-R$ORIGIN' -L/opt/ABC/lib -lB
$ elfdump -d lib/libA.so.1 | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED           0x96                libB.so.1
     [4]  RUNPATH          0xa0                $ORIGIN
```

如果此产品目前安装在 /usr/local/ABC 下，并在您的 PATH 中增加 /usr/local/ABC/bin，则调用应用程序 abc 将产生如下所示的路径名查询，以查找其依赖项：

```
$ ldd -s abc
....
  find object=libA.so.1; required by abc
    search path=$ORIGIN/../lib  (RUNPATH/RPATH from file abc)
      trying path=/usr/local/ABC/lib/libA.so.1
        libA.so.1 =>     /usr/local/ABC/lib/libA.so.1

  find object=libB.so.1; required by /usr/local/ABC/lib/libA.so.1
    search path=$ORIGIN  (RUNPATH/RPATH from file /usr/local/ABC/lib/libA.so.1)
      trying path=/usr/local/ABC/lib/libB.so.1
        libB.so.1 =>     /usr/local/ABC/lib/libB.so.1
```

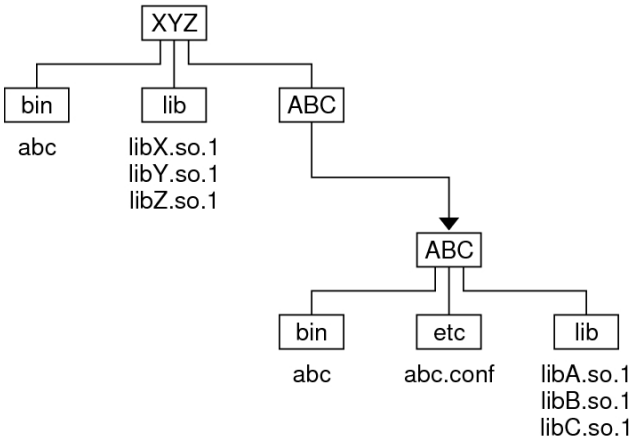
注 - 包含 $ORIGIN 标记的目标文件可使用符号链接进行引用。在这种情况下，符号链接完全解析以确定目标文件的真正来源。

非绑定产品之间的依赖性

另一个与依赖项位置相关的问题是如何建立非绑定产品能借以表达相互之间依赖性的模型。

例如，非绑定产品 XYZ 可能依赖于产品 ABC。可以通过主机软件包安装脚本来建立此依赖性。此脚本生成一个指向 ABC 产品安装点的符号链接，如下图所示：

图 10-2　　非绑定共同依赖项



XYZ 产品的二进制代码和共享目标文件可使用符号链接来表示其对于 ABC 产品的依赖性。此链接现在是一个稳定的参照点。对于应用程序 xyz，此运行路径将如下所示：

```
$ cc -o xyz xyz.c '-R$ORIGIN/../lib:$ORIGIN/../ABC/lib' \
    -L/opt/ABC/lib -lX -lA
$ elfdump -d xyz | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED           0x1b5              libX.so.1
     [1]  NEEDED           0x1bf              libA.so.1
     ....
     [2]  NEEDED           0x18f              libc.so.1
     [5]  RUNPATH          0x1c9              $ORIGIN/../lib:$ORIGIN/../ABC/lib
```

类似地，对于依赖项 libX.so.1，此运行路径将如下所示：

```
$ cc -o libX.so.1 -G -Kpic X.c '-R$ORIGIN:$ORIGIN/../ABC/lib' \
    -L/opt/ABC/lib -lY -lC
$ elfdump -d libX.so.1 | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED           0x96               libY.so.1
     [1]  NEEDED           0xa0               libC.so.1
```

```
    [5]  RUNPATH          0xaa               $ORIGIN:$ORIGIN/../ABC/lib
```

如果此产品目前安装在 /usr/local/XYZ 下，则需要使用其后安装脚本来建立以下内容的符号链接：

```
$ ln -s ../ABC /usr/local/XYZ/ABC
```

如果在您的 PATH 中增加 /usr/local/XYZ/bin，则调用应用程序 xyz 将产生如下所示的路径名查询，以查找其依赖项：

```
$ ldd -s xyz
....
  find object=libX.so.1; required by xyz
    search path=$ORIGIN/../lib:$ORIGIN/../ABC/lib  (RUNPATH/RPATH from file xyz)
      trying path=/usr/local/XYZ/lib/libX.so.1
        libX.so.1 =>     /usr/local/XYZ/lib/libX.so.1

  find object=libA.so.1; required by xyz
    search path=$ORIGIN/../lib:$ORIGIN/../ABC/lib  (RUNPATH/RPATH from file xyz)
      trying path=/usr/local/XYZ/lib/libA.so.1
      trying path=/usr/local/ABC/lib/libA.so.1
        libA.so.1 =>     /usr/local/ABC/lib/libA.so.1

  find object=libY.so.1; required by /usr/local/XYZ/lib/libX.so.1
    search path=$ORIGIN:$ORIGIN/../ABC/lib \
               (RUNPATH/RPATH from file /usr/local/XYZ/lib/libX.so.1)
      trying path=/usr/local/XYZ/lib/libY.so.1
        libY.so.1 =>     /usr/local/XYZ/lib/libY.so.1

  find object=libC.so.1; required by /usr/local/XYZ/lib/libX.so.1
    search path=$ORIGIN:$ORIGIN/../ABC/lib \
               (RUNPATH/RPATH from file /usr/local/XYZ/lib/libX.so.1)
      trying path=/usr/local/XYZ/lib/libC.so.1
      trying path=/usr/local/ABC/lib/libC.so.1
        libC.so.1 =>     /usr/local/ABC/lib/libC.so.1

  find object=libB.so.1; required by /usr/local/ABC/lib/libA.so.1
    search path=$ORIGIN  (RUNPATH/RPATH from file /usr/local/ABC/lib/libA.so.1)
      trying path=/usr/local/ABC/lib/libB.so.1
        libB.so.1 =>     /usr/local/ABC/lib/libB.so.1
```

注 - 在运行时，可以通过组合使用 dlinfo(3C) 和 RTLD_DI_ORIGIN 标志来获取目标文件源。该来源路径可用于访问关联的产品分层结构中的其他文件。

安全性

在安全的进程中，只有将 $ORIGIN 字符串扩展到可信目录时才允许对其进行扩展。出现其他相对路径名将产生安全风险。

$ORIGIN/../lib 之类的路径明显指向由可执行文件的位置决定的固定位置。但是，此位置实际上并不固定。相同文件系统上的可写入目录可能会利用使用 $ORIGIN 的安全程序。

以下示例显示，如果在安全进程中随意扩展 $ORIGIN，可能会发生这种安全违规。

```
$ cd /worldwritable/dir/in/same/fs
$ mkdir bin lib
$ ln $ORIGIN/bin/program bin/program
$ cp ~/crooked-libc.so.1 lib/libc.so.1
$ bin/program
.... using crooked-libc.so.1
```

可以使用实用程序 crle(1) 指定让安全应用程序能够使用 $ORIGIN 的可信目录。使用此技术的管理员应确保已对目标目录进行了适当的保护，以防受到恶意入侵。

♦ ♦ ♦ 第 11 章

11 可扩展性机制

链接编辑器和运行时链接程序提供了接口，用于实现链接编辑器的监视和修改功能以及运行时链接程序处理功能。要使用这些接口，除了了解前几章中所介绍的概念之外，通常还需要对链接编辑的概念有更深入的了解。本章介绍了以下接口：

- *ld-support* – “链接编辑器支持接口” [237]
- *rtld-audit* – “运行时链接程序审计接口” [243]
- *rtld-debugger* – “运行时链接程序调试器接口” [256]

链接编辑器支持接口

链接编辑器可执行许多操作，其中包括打开文件以及串联这些文件中的各节。监视并不时修改这些操作通常会对编译系统的各组件有利。

本节介绍了 *ld-support* 接口。通过此接口可以检查输入文件，并在某种程度上还可以修改链接编辑过程中所用到的那些文件的输入文件数据。使用此接口的两个应用程序分别为链接编辑器以及 `make(1S)` 实用程序。链接编辑器使用此接口处理可重定位目标文件内的调试信息。`make` 实用程序使用此接口保存状态信息。

ld-support 接口由提供一个或多个支持接口例程的支持库组成。该库是在链接编辑过程中装入的。在链接编辑的不同阶段会调用该库中的某些支持例程。

使用此接口时，应该熟悉 `elf(3ELF)` 结构和文件格式。

调用支持接口

链接编辑器可接受一个或多个通过 `SGS_SUPPORT` 环境变量或链接编辑器的 `-S` 选项提供的支持库。此环境变量由冒号分隔的支持库列表组成：

```
$ SGS_SUPPORT=support.so.1:support.so.2  cc ....
```

`-S` 选项用于指定单个支持库。可以指定多个 `-S` 选项：

```
$ LD_OPTIONS="-Ssupport.so.1 -Ssupport.so.2"  cc ....
```

支持库是共享目标文件。链接编辑器会使用 dlopen(3C) 按照指定库的顺序打开每个支持库。如果遇到环境变量和 -S 选项，则首先处理通过此环境变量指定的支持库。然后，使用 dlsym(3C) 搜索每个支持库以查找所有支持接口例程。这些支持例程随后会在链接编辑的不同阶段调用。

支持库必须与所调用的链接编辑器的 ELF 类保持一致，可以是 32 位或 64 位。有关更多详细信息，请参见“32 位环境和 64 位环境” [238]。

注 - 缺省情况下，Solaris OS 支持库 libldstab.so.1 由链接编辑器用于处理和压缩输入可重定位目标文件内提供的由编译器生成的调试信息。如果您调用了包含任何使用 -S 选项指定的支持库的链接编辑器，将禁用此默认处理。除支持库服务外，还可以要求 libldstab.so.1 的默认处理。在这种情况下，将 libldstab.so.1 显式添加到提供给链接编辑器的支持库的列表。

---

32 位环境和 64 位环境

如“32 位环境和 64 位环境” [20]中所述，64 位链接编辑器 ld(1) 可以生成 32 位目标文件。此外，32 位链接编辑器可以生成 64 位目标文件。对于其中每个目标文件，都定义关联支持接口。

64 位目标文件的支持接口类似于 32 位目标文件的接口，但是以 *64* 为后缀结尾。例如 ld_start() 和 ld_start64()。通过此约定，两种方式实现的支持接口可以分别位于 32 位类和 64 位类的单个共享目标文件中。

可以为 SGS_SUPPORT 环境变量指定 _32 或 _64 后缀，并且可以使用链接编辑器选项 -z ld32 和 -z ld64 定义 -S 选项的要求。这些定义只能分别通过链接编辑器的 32 位或 64 位类来解释。通过此操作，可在可能不知道链接编辑器的类的情况下指定两类支持库。

支持接口函数

所有 *ld-support* 接口均在头文件 link.h 中定义。所有接口参数均为基本的 C 类型或 ELF 类型。可以通过 ELF 访问库 libelf 检查 ELF 数据类型。有关 libelf 内容的说明，请参见 elf(3ELF)。以下接口函数由 *ld-support* 接口提供，并且按照预期的使用顺序进行了说明。

ld_version()

此函数提供了链接编辑器与支持库之间的初次握手。

```
uint_t ld_version(uint_t version);
```

链接编辑器使用其可以支持的最高版本的 *ld-support* 接口来调用此接口。支持库可以检验此版本是否达到使用的最低要求，并返回支持库要求使用的版本。此版本通常为 LD_SUP_VCURRENT。

如果支持库没有提供此接口，则采用初始支持级别 LD_SUP_VERSION1。

如果支持库返回版本 LD_SUP_VNONE，则链接编辑器将其卸载而无任何提示，且在不使用它的情况下继续执行。如果返回的版本高于链接编辑器所支持的 *ld-support* 接口的版本，则将发出致命错误，且链接编辑器将终止执行。否则，使用指定的 *ld-support* 接口版本的支持库继续执行。

`ld_start ()`

此函数在初始验证链接编辑器命令行之后调用，表示开始处理输入文件。

```
void ld_start(const char *name, const Elf32_Half type,
    const char *caller);

void ld_start64(const char *name, const Elf64_Half type,
    const char *caller);
```

name 是所创建的输出文件名。type 是输出文件类型，可以为 ET_DYN、ET_REL 或 ET_EXEC，在 sys/elf.h 中定义。caller 是调用接口的应用程序，通常为 /usr/bin/ld 或 /usr/ccs/bin/ld。

`ld_open ()`

此函数会针对每个输入链接编辑的文件调用。版本 LD_SUP_VERSION3 中添加的此函数比 ld_file () 函数具有更大的灵活性。此函数允许支持库替换文件描述符、ELF 描述符以及关联的文件名。此函数提供以下可能的使用情况。

- 将新的节添加到现有 ELF 文件。在这种情况下，应使用支持更新 ELF 文件的描述符替换原始 ELF 描述符。请参见 elf_begin(3ELF) 的 ELF_C_RDWR 参数。
- 可以使用替代文件替换整个输入文件。在这种情况下，应使用与新文件关联的描述符替换原始文件描述符和 ELF 描述符。

在这两种情况下，可以使用表示输入文件已修改的替代名称替换路径名和文件名。

```
void ld_open(const char **pname, const char **fname, int *fd,
    int flags, Elf **elf, Elf *ref, size_t off, Elf_Kind kind);

void ld_open64(const char **pname, const char **fname, int *fd,
    int flags, Elf **elf, Elf *ref, size_t off, Elf_Kind kind);
```

pname 是要处理的输入文件的路径名。fname 是要处理的输入文件的文件名。fname 通常是 pname 的基本名称。pname 和 fname 均可由支持库修改。

fd 是输入文件的文件描述符。此描述符可由支持库关闭，且新的文件描述符可返回至链接编辑器。可以返回值为 -1 的文件描述符，以表示应忽略该文件。

注 - 如果链接编辑器不允许 ld_open () 关闭文件描述符，则将传递至 ld_open () 的 *fd* 的值设为 -1。处理归档成员时最常发生这种情况。如果将值 -1 传递至 ld_open () ，则无法关闭此描述符，支持库也不应返回替代描述符。

---

flags 字段表示链接编辑器获取文件的方式，可以是以下一个或多个定义：

- LD_SUP_DERIVED – 文件名不是在命令行中显式指定的。文件是从 -l 扩展派生而来，或者文件标识提取的归档成员。
- LD_SUP_EXTRACTED – 文件提取自归档。
- LD_SUP_INHERITED – 文件作为命令行共享目标文件的依赖项获取。

如果未指定 flags 值，则表明已在命令行中显式指定了输入文件。

elf 是输入文件的 ELF 描述符。此描述符可由支持库关闭，且新的 ELF 描述符可返回至链接编辑器。可以返回值为 0 的 ELF 描述符，以表示应忽略该文件。如果 *elf* 描述符与归档库的成员关联，则 *ref* 描述符是底层归档文件的 ELF 描述符。*off* 表示归档文件中归档成员的偏移。

kind 表示输入文件类型，可以是 ELF_K_AR 或 ELF_K_ELF，在 libelf.h 中定义。

ld_file ()

此函数针对每个输入链接编辑的文件调用。并且在执行任何文件数据处理之前即会调用此函数。

```
void ld_file(const char *name, const Elf_Kind kind, int flags,
    Elf *elf);

void ld_file64(const char *name, const Elf_Kind kind, int flags,
    Elf *elf);
```

name 是要处理的输入文件。kind 表示输入文件类型，可以是 ELF_K_AR 或 ELF_K_ELF，在 libelf.h 中定义。flags 字段表示链接编辑器获取文件的方式，此字段可包含与 ld_open () 的 flags 字段相同的定义。

- LD_SUP_DERIVED – 文件名不是在命令行中显式指定的。文件是从 -l 扩展派生而来，或者文件标识提取的归档成员。
- LD_SUP_EXTRACTED – 文件提取自归档。
- LD_SUP_INHERITED – 文件作为命令行共享目标文件的依赖项获取。

如果未指定 flags 值，则表明已在命令行中显式指定了输入文件。

elf 是输入文件的 ELF 描述符。

section ()

此函数针对输入文件的每一节调用，并且在链接编辑器确定是否应将节传播给输出文件之前即会调用版本 LD_SUP_VERSION2 中添加的此函数。此函数不同于 ld_section () 处理，后者仅针对组成输出文件的各节进行调用。

```
void ld_input_section(const char *name, Elf32_Shdr **shdr,
    Elf32_Word sndx, Elf_Data *data, Elf *elf, unit_t flags);

void ld_input_section64(const char *name, Elf64_Shdr **shdr,
    Elf64_Word sndx, Elf_Data *data, Elf *elf, uint_t flags);
```

name 是输入节的名称。shdr 是指向关联节头的指针。sndx 是输入文件内的节索引。data 是指向关联数据缓冲区的指针。elf 是指向文件的 ELF 描述符的指针。flags 保留供将来使用。

节头的修改是通过重新分配节头并为新头重新指定 *shdr 来完成的。链接编辑器使用从 ld_input_section () 返回时 *shdr 所指向的节头信息来处理节。

通过重新分配数据并重新指 Elf_Data 缓冲区的 d_buf 指针，可以修改数据。对数据进行任何修改都应确保正确设置 Elf_Data 缓冲区的 d_size 元素。对于成为输出映像一部分的输入节，将 d_size 元素设置为零可以有效地删除输出映像中的数据。

在解压缩压缩的节之前即会调用此函数，这会使检查或替换数据的任务变得更为复杂。因此，建议延迟 ld_section () 的节数据的检查和可能的替换。

flags 字段指向初始值为零的 uint_t 数据字段。虽然在将来的更新中可通过链接编辑器或支持库来指定标志，但是当前未指定任何标志。

ld_section ()

系统将针对传播给输出文件的输入文件的每一节调用此函数，并且在执行任何节数据处理之前即会调用此函数。但是，在调用此函数之前，会解压缩包含压缩数据的节。

```
void ld_section(const char *name, Elf32_Shdr *shdr,
    Elf32_Word sndx, Elf_Data *data, Elf *elf);

void ld_section64(const char *name, Elf64_Shdr *shdr,
    Elf64_Word sndx, Elf_Data *data, Elf *elf);
```

name 是输入节的名称。shdr 是指向关联节头的指针。sndx 是输入文件内的节索引。data 是指向关联数据缓冲区的指针。elf 是指向文件 ELF 描述符的指针。

通过重新分配数据并重新指 Elf_Data 缓冲区的 d_buf 指针，可以修改数据。对数据进行任何修改都应确保正确设置 Elf_Data 缓冲区的 d_size 元素。对于成为输出映像一部分的输入节，将 d_size 元素设置为零可以有效地删除输出映像中的数据。

注 - 不会将从输出文件中删除的节报告给 ld_section () 。使用链接编辑器的 -z strip-class 选项剥离各节。由于 SHT_SUNW_COMDAT 处理或 SHF_EXCLUDE 标识而废弃各节。请参见“*COMDAT* 节” [304]和表 12-8 “ELF 节属性标志”。

---

ld_input_done ()

在输入文件处理完成之后，将调用版本 LD_SUP_VERSION2 中添加的此函数。

此时，所有输入节均已被指定给输出文件映像。此外，已收集创建和更新此映像所需的信息以备应用于初始映像。任何时侯尝试通过 ld_input_done () 更改之前支持例程记录的任何数据时，均应格外小心。对节的标识或关系进行的所有更改都将丢失，或者可能会影响输出文件映像的创建。可应用无需重定位或不会影响现有定位的次要更新，例如添加节数据。

```
void ld_input_done(uint_t *flags);
```

flags 字段指向初始值为零的 uint_t 数据字段。虽然在将来的更新中可通过链接编辑器或支持库来指定标志，但是当前未指定任何标志。

`ld_atexit` ()

此函数在完成链接编辑时调用。

```
void ld_atexit(int status);

void ld_atexit64(int status);
```

`status` 是将由链接编辑器返回的 exit(2) 代码，可以是 `EXIT_FAILURE` 或 `EXIT_SUCCESS`，在 `stdlib.h` 中定义。

支持接口示例

以下示例创建了一个支持库，其中显示了在 32 位链接编辑过程中处理的任何可重定位目标文件的节名称。

```
$ cat support.c
#include    <link.h>
#include    <stdio.h>

static int indent = 0;

void
ld_start(const char *name, const Elf32_Half type, const char *caller)
{
        (void) printf("output image: %s\n", name);
}

void
ld_file(const char *name, const Elf_Kind kind, int flags, Elf *elf)
{
        if (flags & LD_SUP_EXTRACTED)
                indent = 4;
        else
                indent = 2;

        (void) printf("%*sfile: %s\n", indent, "", name);
}

void
ld_section(const char *name, Elf32_Shdr *shdr, Elf32_Word sndx,
    Elf_Data *data, Elf *elf)
{
        Elf32_Ehdr *ehdr = elf32_getehdr(elf);

        if (ehdr->e_type == ET_REL)
                (void) printf("%*s   section [%ld]: %s\n", indent,
                    "", (long)sndx, name);
}
```

此支持库依赖于 `libelf` 来提供用于确定输入文件类型的 ELF 访问函数 `elf32_getehdr(3ELF)`。此支持库通过使用以下命令生成：

```
$ cc -o support.so.1 -G -K pic support.c -lelf -lc
```

以下示例说明了从可重定位目标文件和本地归档库构造普通应用程序所产生的节诊断信息。使用 -S 选项不仅可以处理缺省调试信息，还可以调用支持库。

```
$ LD_OPTIONS=-S./support.so.1 cc -o prog main.c -L. -lfoo

output image: prog
  file: /opt/COMPILER/crti.o
    section [1]: .shstrtab
    section [2]: .text
    ....
  file: /opt/COMPILER/crt1.o
    section [1]: .shstrtab
    section [2]: .text
    ....
  file: /opt/COMPILER/values-xt.o
    section [1]: .shstrtab
    section [2]: .text
    ....
  file: main.o
    section [1]: .shstrtab
    section [2]: .text
    ....
  file: ./libfoo.a
    file: ./libfoo.a(foo.o)
      section [1]: .shstrtab
      section [2]: .text
      ....
  file: /lib/libc.so
  file: /opt/COMPILER/crtn.o
    section [1]: .shstrtab
    section [2]: .text
    ....
```

注 - 为了简化输出，已经减少了本示例中显示的节数。另外，编译器驱动程序所包含的文件也会有所不同。

运行时链接程序审计接口

您可以使用 *rtld-audit* 接口访问与进程的运行时链接有关的信息。*rtld-audit* 接口实现为提供一个或多个审计接口例程的审计库。如果将该库作为进程的一部分装入，则运行时链接程序会在进程执行的不同阶段调用审计例程。审计库可以使用这些接口访问以下各项：

- 依赖项搜索。可以通过审计库替换搜索路径。
- 与装入的目标文件相关的信息。
- 装入的目标文件之间进行的符号绑定。可以通过审计库更改这些绑定。

- 通过利用过程链接表各项所提供的延迟绑定机制，可以审计函数调用及其返回值。请参见“过程链接表（特定于处理器）” [372]。可以通过审计库修改函数参数及其返回值。

通过预装入专用的共享目标文件可以获取其中的部分信息。但是，预装入的目标文件与应用程序的目标文件存在于同一名称空间内。这种预装入通常会限制预装入共享目标文件的实现或者使实现变得更为复杂。*rtld-audit* 接口会为您提供唯一的名称空间，用于在其中执行审计库。此名称空间可确保在应用程序内进行正常绑定时审计库不会侵入。

“配置共享目标文件” [173]中介绍对共享目标文件的运行时配置即是使用此 *rtld-audit interface* 的一个示例。

建立名称空间

运行时链接程序将动态可执行文件与其依赖项绑定时，会生成链接映射的链接列表，用于对此应用程序进行说明。链接映射结构说明了应用程序中的每个目标文件。`/usr/include/sys/link.h` 中定义了此链接映射结构。绑定应用程序的目标文件所需的符号搜索机制会遍历此链接映射的列表。此链接映射列表用于提供应用程序符号解析的名称空间。

运行时链接程序也通过链接映射来进行说明。此链接映射以不同于应用程序目标文件列表的列表中进行维护。因此，运行时链接程序驻留在其自己唯一的名称空间中，从而可防止应用程序查看或直接访问运行时链接程序内的任意服务。因此，应用程序只能通过 `libc.so.1` 或 `libdl.so.1` 提供的过滤器来访问运行时链接程序。

`/usr/include/link.h` 中定义了两个标识符，用于定义应用程序和运行时链接程序的链接映射列表：

```
#define LM_ID_BASE      0     /* application link-map list */
#define LM_ID_LDSO      1     /* runtime linker link-map list */
```

除了这两个标准的链接映射列表以外，运行时链接程序还允许再创建任意数量的链接映射列表。其中每个链接映射列表都提供一个唯一的名称空间。*rtld-audit* 接口使用自己的用于维护审计库的链接映射列表。因此，在应用程序的符号绑定要求中，不涉及审计库。针对每个 *rtld-audit* 支持库会指定一个唯一的新链接映射标识符。

审计库可以使用 `dlmopen(3C)` 检查应用程序链接映射列表。将 `dlmopen（）` 与 `RTLD_NOLOAD` 标志结合使用时，审计库可以在不装入目标文件的情况下查询该目标文件是否存在。

创建审计库

审计库的生成方式与其他任何共享目标文件的生成方式相同。但是，必须注意进程内审计库名称空间的唯一性。

- 该库必须提供所有依赖性需求。
- 该库不应使用无法用于进程内多个接口实例的系统接口。

如果审计库引用外部接口，则审计库必须定义提供接口定义的依赖性。例如，如果审计库调用 printf(3C)，则审计库必须定义与 libc 之间的依赖性。请参见“生成共享目标文件输出文件” [40]。由于审计库具有唯一的名称空间，因此，所审计的应用程序中提供的 libc 无法满足符号引用。如果审计库依赖于 libc，则会向进程中装入两种版本的 libc.so.1。一种版本用于满足应用程序链接映射列表的绑定要求，另一种版本用于满足审计链接映射列表的绑定要求。

要确保生成的审计库会记录所有的依赖项，请使用链接编辑器的 -z defs 选项。

部分系统接口会假定其是进程内实现的唯一实例，例如信号和 malloc(3C)。审计库应该避免使用此类接口，因为这样做可能会无意中更改应用程序的行为。

注 - 审计库可以使用 mapmalloc(3MALLOC) 来分配内存，因为此分配方法可以与应用程序通常使用的任何分配方案同时存在。

调用审计接口

rtld-audit 接口可通过以下两种方法之一来启用。每种方法都会指示一个所审计的目标文件的作用域。

- 通过在生成目标文件时定义一个或多个审计程序来启用局部审计。请参见“记录局部审计程序” [246]。通过此方法在运行时可用的审计库附带有与请求局部审计的动态目标文件相关的信息。
- 通过使用环境变量 LD_AUDIT 定义一个或多个审计程序来启用全局审计。通过将局部审计定义与 -z globalaudit 选项组合，也可以为应用程序启用全局审计。请参见“记录全局审计程序” [246]。通过此方法在运行时可用的审计库附带有与应用程序所用的所有动态目标文件相关的信息。

这两种定义审计程序的方法均使用一个字符串，其中包含通过 dlmopen(3C) 装入的以冒号分隔的共享目标文件列表。每个目标文件都装入各自的审计链接映射列表中。使用 dlsym(3C) 可搜索每个目标文件中的审计例程。在应用程序执行过程中的不同阶段会调用找到的审计例程。

安全应用程序只能从可信目录中获取审计库。缺省情况下，用于 32 位目标文件的运行时链接程序可识别的可信目录仅有 /lib/secure 和 /usr/lib/secure。对于 64 位目标文件，可信目录是 /lib/secure/64 和 /usr/lib/secure/64。

注 - 通过将环境变量 LD_NOAUDIT 设置为非空值，可在运行时禁用审计。

记录局部审计程序

使用链接编辑器选项 -p 或 -P 生成目标文件时，可以确定局部审计要求。例如，要使用审计库 audit.so.1 审计 libfoo.so.1，请在链接编辑时使用 -p 选项记录要求：

```
$ cc -G -o libfoo.so.1 -Wl,-paudit.so.1 -K pic foo.c
$ elfdump -d libfoo.so.1 | grep AUDIT
     [2]  AUDIT            0x96               audit.so.1
```

在运行时，如果存在此审计标识符，则会装入审计库。然后，将与标识目标文件相关的信息传递到此审计库。

如果单独使用此机制，则在装入审计库之前会显示搜索标识目标文件之类的信息。要提供尽可能多的审计信息，需要将存在的要求局部审计的目标文件传播给此目标文件的用户。例如，如果生成的应用程序依赖于 libfoo.so.1，则会对此应用程序进行标识，指明其依赖项需要审计：

```
$ cc -o main main.c libfoo.so.1
$ elfdump -d main | grep AUDIT
     [4]  DEPAUDIT         0x1be              audit.so.1
```

通过此机制启用的审计会导致向审计库中传递与所有应用程序显式依赖项有关的信息。使用链接编辑器的 -P 选项，还可以在创建目标文件时直接记录此依赖项审计：

```
$ cc -o main main.c -Wl,-Paudit.so.1
$ elfdump -d main | grep AUDIT
     [3]  DEPAUDIT         0x1b2              audit.so.1
```

记录全局审计程序

通过设置环境变量 LD_AUDIT 可以确定全局审计要求。例如，针对审计库 audit.so.1，此环境变量可用于审计应用程序 main 以及该应用程序的所有依赖项。

```
$ LD_AUDIT=audit.so.1 main
```

通过记录应用程序中的局部审计程序以及 -z globalaudit 选项，还可以实现全局审计。例如，通过使用链接编辑器的 -P 选项和 -z globalaudit 选项，可以生成应用程序 main 以启用全局审计。

```
$ cc -o main main.c -Wl,-Paudit.so.1 -z globalaudit
$ elfdump -d main | grep AUDIT
     [3]  DEPAUDIT         0x1b2              audit.so.1
    [26]  FLAGS_1       0x1000000            [ GLOBAL-AUDITING ]
```

通过以上任一机制启用的审计会导致向审计库中传递与应用程序的所有动态目标文件有关的信息。

审计接口交互

为审计例程提供了一个或多个 *cookie*。cookie 是描述单个动态目标文件的数据项。最初装入动态目标文件时，为 `la_objopen()` 例程提供了一个初始 cookie。此 cookie 是指向装入的动态目标文件的关联 `Link_map` 的指针。但是，`la_objopen()` 例程可自由分配，并返回至运行时链接程序（备用 cookie）。此机制为审计程序提供了一种使用每个动态目标文件维护其自身数据并使用所有后续审计例程调用接收此数据的方法。

通过 *rtld-audit* 接口可以提供多个审计库。在这种情况下，从一个审计程序返回的信息将传递至下一个审计程序的同一审计例程。同样，由一个审计程序建立的 cookie 将传递至下一个审计程序。设计预期与其他审计库共存的审计库时应谨慎。一种安全的方法是不应更改通常由运行时链接程序返回的绑定或 cookie。更改这些数据会在后面的审计库中产生意外结果。否则，应设计所有审计程序相互协作以安全更改任何绑定信息或 cookie 信息。

审计接口函数

rtld-audit 接口提供了以下例程。这些例程按照其预期的使用顺序进行说明。

注 - 为了简化讨论，对体系结构或目标文件类特定接口的引用会缩减为其通用名称。例如，对 `la_symbind32()` 和 `la_symbind64()` 的引用会指定为 `la_symbind()`。

`la_version()`

此例程可提供运行时链接程序与审计库之间的初次握手。必须提供此接口才能装入审计库。

```
uint_t la_version(uint_t version);
```

运行时链接程序通过其可以支持的最高 `version` 的 *rtld-audit* 接口来调用此接口。审计库可以检验此版本是否足以供其使用，并返回审计库预期使用的版本。此版本通常为 `/usr/include/link.h` 中定义的 `LAV_CURRENT`。

如果审计库返回零，或者返回的版本高于运行时链接程序所支持的 *rtld-audit* 接口的版本，则会废弃该审计库。

为其余审计例程提供了一个或多个 *cookie*。请参见“审计接口交互” [247]。

在 `la_version()` 调用之后，对 `la_objopen()` 例程进行了两次调用。第一次调用提供动态可执行文件的链接映射信息，第二次调用提供运行时链接程序的链接映射信息。

`la_objopen()`

此例程在运行时链接程序装入新目标文件时调用。

```
uint_t la_objopen(Link_map *lmp, Lmid_t lmid, uintptr_t *cookie);
```

lmp 提供说明新目标文件的链接映射结构。lmid 标识添加了目标文件的链接映射列表。cookie 提供指向某个标识符的指针。此标识符会初始化为目标文件 lmp。审计库可以重新指定此标识符，以便更好地标识其他 *rtld-audit* 接口例程的目标文件。

la_objopen（） 例程会返回表示与此目标文件相关的符号绑定的值。返回值是 /usr/include/link.h 中定义的以下值的掩码：

- LA_FLG_BINDTO – 审计到此目标文件的符号绑定。
- LA_FLG_BINDFROM – 审计来自此目标文件的符号绑定。

通过这些值，审计程序可以选择要使用 la_symbind（） 监视的目标文件。返回值为零表示绑定信息与此目标文件无关。

例如，审计程序可以监视从 libfoo.so 到 libbar.so 的绑定。将 la_objopen（） 用于 libfoo.so 会返回 LA_FLG_BINDFROM。将 la_objopen（） 用于 libbar.so 会返回 LA_FLG_BINDTO。

审计程序可以监视 libfoo.so 与 libbar.so 之间的所有绑定。将 la_objopen（） 用于这两个目标文件会返回 LA_FLG_BINDFROM 和 LA_FLG_BINDTO。

审计程序可以监视到 libbar.so 的所有绑定。将 la_objopen（） 用于 libbar.so 会返回 LA_FLG_BINDTO。所有 la_objopen（） 调用都会返回 LA_FLG_BINDFROM。

使用审计版本 LAV_VERSION5，将一个表示动态可执行文件的 la_objopen（） 调用提供给局部审计程序。在这种情况下，审计程序不会返回符号绑定标志，因为它可能装入过晚，而无法监视与该动态可执行文件关联的符号绑定。忽略审计程序返回的任何标志。la_objopen（） 调用为局部审计程序提供了一个初始 cookie，它是任意后续 la_preinit（） 或 la_activity（） 调用所需的。

la_activity（）

此例程可通知审计程序正在进行链接映射活动。

```
void la_activity(uintptr_t *cookie, uint_t flags);
```

cookie 标识作为链接映射标题的目标文件。flags 表示活动类型，如 /usr/include/link.h 中所定义：

- LA_ACT_ADD – 正在向链接映射列表中添加目标文件。
- LA_ACT_DELETE – 正在从链接映射列表中删除目标文件。
- LA_ACT_CONSISTENT – 已经完成目标文件活动。

在动态可执行文件和运行时链接程序进行 la_objopen（） 调用之后，在进程启动时调用 LA_ACT_ADD 活动，以指示添加了新的依赖项。对于延迟装入和 dlopen(3C) 事件，也会调用此活动。使用 dlclose(3C) 删除目标文件时，也会调用 LA_ACT_DELETE 活动。

LA_ACT_ADD 和 LA_ACT_DELETE 活动均为预期随后发生的事件的提示。在一些情况下，呈现的事件可能有所不同。例如，如果目标文件未能完全重定位，则添加新目标文件可能会导致一些新目标文件被删除。如果 .fini 执行导致延迟装入新目标文件，则删除目标文件还会导致添加新目标文件。LA_ACT_CONSISTENT 活动在任何目标

文件添加或删除操作之后发生，可依赖它来指示应用程序链接映射列表的一致性。审计程序应谨慎检验实际结果，而不是盲目相信 LA_ACT_ADD 和 LA_ACT_DELETE 提示。

对于审计版本 LAV_VERSION1 至 LAV_VERSION4，仅针对全局审计程序调用 la_activity () 。对于审计版本 LAV_VERSION5，局部审计程序可获取活动事件。活动事件提供一个表示应用程序链接映射的 cookie。要准备此活动并允许审计程序控制此 cookie 的内容，请首先对局部审计程序进行 la_objopen () 调用。la_objopen () 调用提供一个表示应用程序链接映射的初始 cookie。请参见"审计接口交互" [247]。

la_objsearch ()

此例程可通知审计程序将要搜索目标文件。

```
char *la_objsearch(const char *name, uintptr_t *cookie, uint_t flags);
```

name 表示所搜索的文件名或路径名。cookie 标识启动搜索的目标文件。flags 标识 name 的来源和创建方式，如 /usr/include/link.h 中所定义：

- LA_SER_ORIG – 初始搜索名称。通常，此名称表示记录为 DT_NEEDED 项的文件名或者提供给 dlopen(3C) 的参数。
- LA_SER_LIBPATH – 已经通过 LD_LIBRARY_PATH 组件创建了路径名。
- LA_SER_RUNPATH – 已经通过运行路径组件创建了路径名。
- LA_SER_DEFAULT – 已经通过缺省搜索路径组件创建了路径名。
- LA_SER_CONFIG – 路径组件源自配置文件。请参见 crle(1)。
- LA_SER_SECURE – 路径组件特定于安全目标文件。

返回值会指明运行时链接程序应该继续处理的搜索路径名。值为零表示应该忽略此路径。监视搜索路径的审计库会返回 name。

la_objfilter ()

此例程在过滤器装入新的 *filtee* 时调用。请参见"作为过滤器的共享目标文件" [127]。

```
int la_objfilter(uintptr_t *fltrcook, const char *fltestr,
    uintptr_t *fltecook, uint_t flags);
```

fltrcook 标识过滤器。fltestr 指向 *filtee* 字符串。fltecook 标识 *filtee*。flags 当前未使用。对于过滤器和 *filtee*，la_objfilter () 在调用 la_objopen () 之后调用。

值为零表示应该忽略此 *filtee*。监视过滤器使用情况的审计库会返回非零值。

la_preinit ()

此例程在为应用程序装入所有即时依赖性之后调用一次。

```
void la_preinit(uintptr_t *cookie);
```

cookie 标识启动进程的主目标文件，通常为动态可执行文件。

调用 la_preinit() 后，进程仍需要线程初始化，包括创建任何初始线程局部存储。请参见“程序启动” [384]。此外，所有装入的目标文件的初始化节在执行之前仍需要收集和排序。请参见“初始化和终止例程” [100]。此函数提供了一个便利的控制点，用于将其他目标文件添加到初始进程。这些目标文件可用于初始线程局部存储和进程初始化。

对于审计版本 LAV_VERSION1 至 LAV_VERSION4，仅针对全局审计程序调用 la_preinit()。对于审计版本 LAV_VERSION5，局部审计程序可获取 preinit 事件。preinit 事件提供一个表示应用程序链接映射的 cookie。要准备此 preinit 并允许审计程序控制此 cookie 的内容，请首先对局部审计程序进行 la_objopen() 调用。la_objopen() 调用提供一个表示应用程序链接映射的初始 cookie。请参见“审计接口交互” [247]。

la_callinit()

在完成线程初始化且建立所有初始线程局部存储之后，将调用此例程。此外，已收集并对所有初始化例程进行排序，以待执行。

```
void la_callinit(uintptr_t *cookie);
```

将针对 la_preinit() 介绍 cookie 以及在全局或局部审计程序中的调用。

审计版本 LAV_VERSION6 添加的此接口标志到执行应用程序代码的转换。

la_callentry()

在执行所有初始化例程之后，将调用此例程。

```
void la_callentry(uintptr_t *cookie);
```

将针对 la_preinit() 介绍 cookie 以及在全局或局部审计程序中的调用。

审计版本 LAV_VERSION6 添加的此接口将标记到应用程序入口点的转换。

la_symbind()

在已经通过 la_objopen() 标记用于绑定通知的两个目标文件之间进行绑定时，会调用此例程。

```
uintptr_t la_symbind32(Elf32_Sym *sym, uint_t ndx,
    uintptr_t *refcook, uintptr_t *defcook, uint_t *flags);

uintptr_t la_symbind64(Elf64_Sym *sym, uint_t ndx,
    uintptr_t *refcook, uintptr_t *defcook, uint_t *flags,
    const char *sym_name);
```

sym 是构造的符号结构，其 sym->st_value 表示所绑定的符号定义的地址。请参见 /usr/include/sys/elf.h。la_symbind32() 可将 sym->st_name 调整为指向实际符号名称。la_symbind64() 可保留 sym->st_name 作为绑定目标文件字符串表的索引。

ndx 表示绑定目标文件的动态符号表内的符号索引。refcook 标识引用此符号的目标文件。此标识符与传递给 la_objopen() 例程的标识符相同，此例程会返

回 LA_FLG_BINDFROM。defcook 标识定义此符号的目标文件。此标识符与传递给 la_objopen（） 的标识符相同，此函数会返回 LA_FLG_BINDTO。

flags 指向可以传送与绑定相关的信息的数据项。此数据项还可用于修改对此过程链接表项的继续审计。该值是 /usr/include/link.h 中定义的符号绑定标志的掩码。

可以为 la_symbind（） 提供以下标志：

- LA_SYMB_DLSYM – 由于调用 dlsym(3C) 而发生的符号绑定。
- LA_SYMB_ALTVALUE (LAV_VERSION2) – 通过先前调用 la_symbind（） 为符号值返回替换值。

如果 la_pltenter（） 或 la_pltexit（） 例程存在，则对于过程链接表的各项，这些例程在 la_symbind（） 之后调用。每次引用符号时都会调用这些例程。另请参见“审计接口限制” [255]。

la_symbind（） 可以提供以下标志来更改此缺省行为。这些标志可应用于按位或运算（包含边界值），并通过 flags 参数来指示其值。

- LA_SYMB_NOPLTENTER – 请勿针对此符号调用 la_pltenter（） 例程。
- LA_SYMB_NOPLTEXIT – 请勿针对此符号调用 la_pltexit（） 例程。

返回值表示在此调用后将控制权传递到的地址。监视符号绑定的审计库应该返回值 sym->st_value，以便将控制权传递给绑定符号定义。审计库可以通过返回不同的值对符号绑定进行专门重定向。

sym_name 仅适用于 la_symbind64（），其中包含所处理的符号的名称。对于 32 位接口，可在 sym->st_name 字段中使用此名称。

la_pltenter（）

这些例程是系统特定的。调用过程链接表中位于已经标记用于绑定通知的两个目标文件之间的一项时，会调用这些例程。

```
uintptr_t la_sparcv8_pltenter(Elf32_Sym *sym, uint_t ndx,
    uintptr_t *refcook, uintptr_t *defcook,
    La_sparcv8_regs *regs, uint_t *flags);

uintptr_t la_sparcv9_pltenter(Elf64_Sym *sym, uint_t ndx,
    uintptr_t *refcook, uintptr_t *defcook,
    La_sparcv9_regs *regs, uint_t *flags,
    const char *sym_name);

uintptr_t la_i86_pltenter(Elf32_Sym *sym, uint_t ndx,
    uintptr_t *refcook, uintptr_t *defcook,
    La_i86_regs *regs, uint_t *flags);

uintptr_t la_amd64_pltenter(Elf64_Sym *sym, uint_t ndx,
    uintptr_t *refcook, uintptr_t *defcook,
    La_amd64_regs *regs, uint_t *flags, const char *sym_name);
```

sym、ndx、refcook、defcook 和 sym_name 提供的信息与传递给 la_symbind（） 的信息相同。

对于 la_sparcv8_pltenter（） 和 la_sparcv9_pltenter（） ，regs 指向输出寄存器。对于 la_i86_pltenter（） ，regs 指向栈寄存器和帧寄存器。对于 la_amd64_pltenter（） ，regs 指向栈寄存器、帧寄存器以及用于传递整数参数的寄存器。regs 在 /usr/include/link.h 中定义。

flags 指向可以传送与绑定相关的信息的数据项。此数据项可用于修改对此过程链接表项的继续审计。此数据项与 la_symbind（） 中的 flags 指向的数据项相同。

la_pltenter（） 可以提供以下标志来更改当前的审计行为。这些标志可应用于按位或运算（包含边界值），并通过 flags 参数来指示其值。

- LA_SYMB_NOPLTENTER – 不能针对此符号再次调用 la_pltenter（） 。
- LA_SYMB_NOPLTEXIT – 不能针对此符号再次调用 la_pltexit（） 。

返回值表示在此调用后将控制权传递到的地址。监视符号绑定的审计库应该返回值 sym->st_value，以便将控制权传递给绑定符号定义。审计库可以通过返回不同的值对符号绑定进行专门重定向。

la_pltexit（）

返回过程链接表中位于已经标记用于绑定通知的两个目标文件之间的一项时，会调用此例程。此例程在调用者获取控制权之前调用。

```
uintptr_t la_pltexit(Elf32_Sym *sym, uint_t ndx, uintptr_t *refcook,
    uintptr_t *defcook, uintptr_t retval);

uintptr_t la_pltexit64(Elf64_Sym *sym, uint_t ndx, uintptr_t *refcook,
    uintptr_t *defcook, uintptr_t retval, const char *sym_name);
```

sym、ndx、refcook、defcook 和 sym_name 提供的信息与传递给 la_symbind（） 的信息相同。retval 是绑定函数的返回代码。监视符号绑定的审计库应该返回 retval。审计库可以专门返回不同的值。

注 - la_pltexit（） 接口是实验接口。请参见“审计接口限制” [255]。

---

la_objclose（）

此例程在执行目标文件的任何终止代码之后和卸载目标文件之前调用。

```
uint_t la_objclose(uintptr_t *cookie);
```

cookie 标识目标文件，并从先前的 la_objopen（） 中获取。当前会忽略任何返回值。

审计接口控制流量

以下各节介绍了审计库可使用每个接口执行的审计接口例程和操作。重点介绍了进程初始化。这些例程将按照其在常规全局审计程序（在进程启动时提供）中的调用顺序显示。

审计接口分为两个类别：信息性和控制性。

信息性接口提供有关执行进程的审计库信息，例如目标文件搜索、目标文件装入和符号绑定。此外，这些接口还允许审计程序修改装入的目标文件，以及请求接收未来符号绑定事件的通知。

通过调用控制性接口，审计库可以跟踪进程执行过程中活动的起始阶段或结束阶段。通过这些接口，可以允许审计程序安全地检查一致性目标文件集合，甚至可以允许装入新的目标文件。

第一次装入一个审计库后，将即时调用此库的 `la_version（）` 接口。此握手会验证此审计库是否受支持，且允许此库通过运行时链接程序定义此库需要的接口版本。

在进程启动时，可通过使用 `LD_AUDIT` 或通过启动进程的可执行目标文件中的局部审计定义建立审计库。请参见“调用审计接口” [245]。在此方案中，将向审计库提供针对可执行目标文件和运行时链接程序的 `la_objopen（）` 调用。

此时，进程仍处于构建的早期阶段。审计程序应避免执行任何可能干扰此构建的操作，例如将其他目标文件添加到进程或对进程进行全面符号搜索。这些操作会导致提前装入和重定位目标文件，以试图满足符号查找。

进程初始化时即时装入的所有依赖项都要报告给审计库的 `la_objopen（）` 接口。对于使用延迟装入的进程，进程初始化时仅装入几个依赖项。请参见“延迟装入动态依赖项” [97]。每个装入的目标文件都将进行重定位，从而导致将在符号引用与符号定义之间建立符号绑定。这些绑定都将报告给审计库的 `la_symbind（）` 接口。

一旦装入并重定位了所有即时依赖项，将立即调用审计库的 `la_preinit（）` 接口。此时，进程仍在构建中。线程初始化和初始化例程收集仍处于暂挂状态。但是，此接口提供了一个便利的控制点，用于将其他目标文件添加到初始进程。

一旦线程初始化完成，将立即调用审计库的 `la_callinit（）` 接口。此时，所有装入的目标文件可随时执行，同时已收集并对所有初始化例程进行排序，以待执行。请参见“初始化和终止例程” [100]。`la_callinit（）` 控制点用于标记到执行应用程序代码的转换。

执行应用程序代码会导致在符号引用与符号定义之间建立函数调用绑定。这些绑定都将报告给审计库的 `la_symbind（）` 和/或 `la_pltenter（）` 接口。使用延迟装入，可装入其他目标文件以满足符号引用，这些引用都将报告给审计库的 `la_objopen（）`。

一旦执行了所有初始化代码，将立即调用审计库的 `la_callentry（）` 接口。`la_callentry（）` 控制点用于标记进程初始化的结束，以及到应用程序入口点（通常为 `start（）` 或 `main（）`）的转换。

继续执行进程时，会出现更多的符号绑定，从而会调用 `la_symbind（）` 和/或 `la_pltenter（）`。可装入其他依赖项，从而会调用 `la_objopen（）`。还可卸载新的依赖项，从而会调用 `la_objclose（）`。目标文件的任何装入或卸载均由一对 `la_activity（）` 调用进行绑定。在执行有针对性的行为（添加或删除目标文件）时，会提示第一个 `la_activity（）`。第二个 `la_activity（）` 表示进程中的依赖关系结构不一致。审计程序应限制其对进程进行的检查，以遵循一致的通知。

审计接口示例

以下简单示例创建了一个审计库，其中显示了动态可执行文件 date(1) 装入的每个共享目标文件依赖项的名称。

```
$ cat audit.c
#include    <link.h>
#include    <stdio.h>

uint_t
la_version(uint_t version)
{
        return (LAV_CURRENT);
}

uint_t
la_objopen(Link_map *lmp, Lmid_t lmid, uintptr_t *cookie)
{
        if (lmid == LM_ID_BASE)
                (void) printf("file: %s loaded\n", lmp->l_name);
        return (0);
}
$ cc -o audit.so.1 -G -K pic -z defs audit.c -lmapmalloc -lc
$ LD_AUDIT=./audit.so.1  date
file: date loaded
file: /lib/libc.so.1 loaded
file: /lib/libm.so.2 loaded
file: /usr/lib/locale/en_US/en_US.so.2 loaded
Thur Aug  10 17:03:55 PST 2012
```

审计接口演示

在 /usr/demo/link_audit 下的 pkg:/solaris/source/demo/system 软件包中提供了许多使用 *rtld-audit* 接口的演示应用程序。

sotruss
: 此演示跟踪指定的应用程序的动态目标文件之间的过程调用。

whocalls
: 此演示跟踪指定的应用程序每次调用指定函数时所用栈。

perfcnt
: 此演示跟踪指定的应用程序的每个函数执行时间。

symbindrep
: 此演示报告为装入指定的应用程序而执行的所有符号绑定。

sotruss(1) 和 whocalls(1) 包括在 pkg:/developer/linker 软件包中。perfcnt 和 symbindrep 是程序示例。这些应用程序不适用于生产环境。

审计接口限制

rtld-audit 实现中存在一些限制。设计审计库时应谨慎了解这些限制。

使用应用程序代码

将目标文件添加到进程时审计库会接收到信息。审计库接收到这种信息时，所监视目标文件可能无法执行。例如，对于装入的目标文件，审计程序可能会接收到 la_objopen() 调用。但是，在该目标文件中的任何代码可以使用之前，该目标文件必须装入其自身的依赖项并进行重定位。审计库可能需要通过使用 dlopen(3C) 获取句柄来检查装入的目标文件。然后，通过使用 dlsym(3C)，可将该句柄用于搜索接口。但是，除非已知目标文件的初始化已完成，否则不应调用采用这种方式获取的接口。

la_pltexit() 的用法

使用 la_pltexit() 系列存在一些限制。这些限制是由于需要在调用者和被调用者之间插入额外栈帧，以提供 la_pltexit() 返回值。此要求在仅调用 la_pltenter() 例程时不会产生问题。在这种情况下，可以在将控制权转交给目标函数之前清除任何干预栈。

由于存在这些限制，因此应该将 la_pltexit() 视为实验接口。在不确定的情况下，请避免使用 la_pltexit() 例程。

直接检查栈的函数

有少量函数可以直接检查栈或对其状态做出假设。这些函数的一些示例包括 setjmp(3C) 系列、vfork(2) 以及返回结构而不是指向结构的指针的任何函数。为支持 la_pltexit() 而创建的额外栈会破坏这些函数。

由于运行时链接程序无法检测此类型的函数，因此，审计库创建者会负责针对此类例程禁用 la_pltexit()。

运行时链接程序调试器接口

运行时链接程序可执行许多操作，包括将目标文件映射到内存中以及绑定符号。调试程序通常需要在分析应用程序的过程中访问说明这些运行时链接程序操作的信息。这些调试器作为不同于其所分析的应用程序的进程运行。

本节介绍了用于监视和修改其他进程中的动态链接应用程序的 *rtld-debugger* 接口。此接口的体系结构采用 `libc_db(3LIB)` 中所使用的模型。

使用 *rtld-debugger* 接口时，至少涉及两个进程：

- 一个或多个目标进程。目标进程必须动态链接，对于 32 位进程，使用运行时链接程序 `/usr/lib/ld.so.1`，对于 64 位进程，使用 `/usr/lib/64/ld.so.1`。
- 控制进程与 *rtld-debugger* 接口库链接，并使用该接口来检查目标进程的动态方面。64 位控制进程可以调试 64 位目标和 32 位目标。但是，32 位控制进程只能调试 32 位目标。

当控制进程为调试器并且其目标为动态可执行文件时，最需要使用 *rtld-debugger* 接口。

rtld-debugger 接口可启用目标进程的以下活动：

- 与运行时链接程序初次会合。
- 通知装入和卸载动态目标文件。
- 检索与任何装入的目标文件相关的信息。
- 跳过过程链接表项。
- 启用目标文件填充。

控制进程和目标进程之间的交互

要检查和处理目标进程，*rtld-debugger* 接口需要使用导出接口、导入接口以及代理在这些接口之间进行通信。

控制进程与 `librtld_db.so.1` 所提供的 *rtld-debugger* 接口链接，并会请求从该库导出的接口。此接口在 `/usr/include/rtld_db.h` 中定义。与此相反，`librtld_db.so.1` 会请求从控制进程导入的接口。通过此交互，*rtld-debugger* 接口可以执行以下操作：

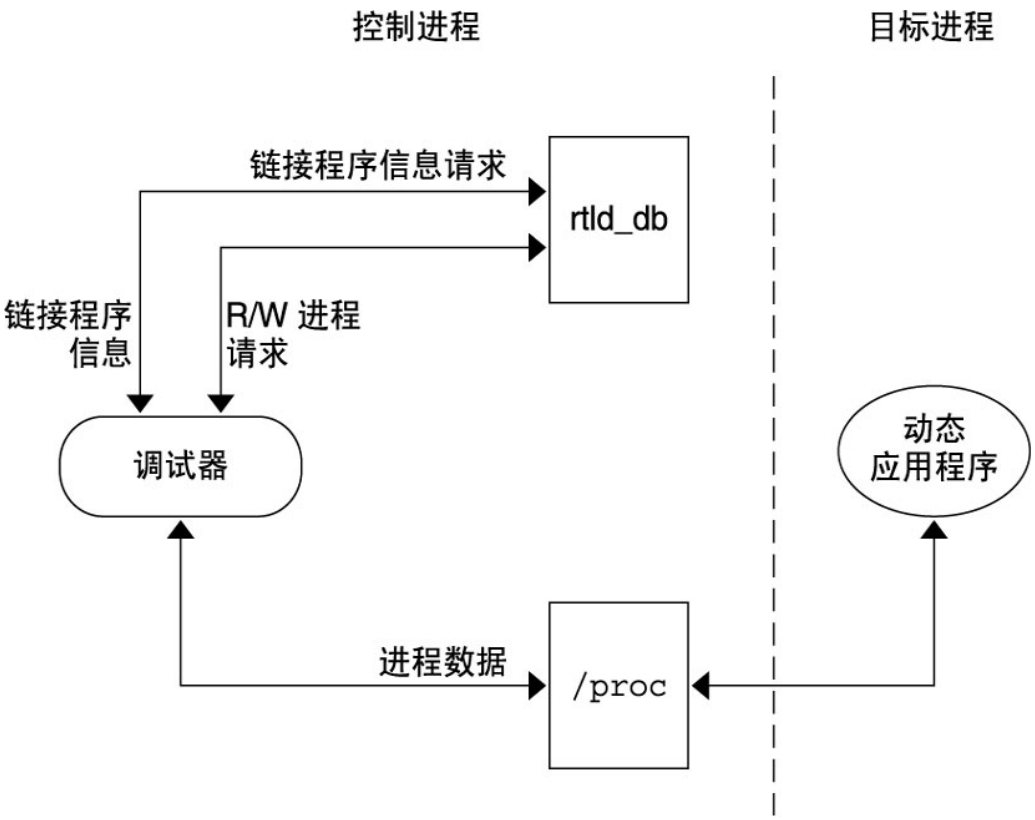
- 在目标进程中查找符号。
- 在目标进程中读写内存。

导入接口由许多 `proc_service` 例程组成，大多数调试器已经使用这些例程来分析进程。这些例程将在“调试器导入接口” [266]中进行介绍。

rtld-debugger 接口假定请求 *rtld-debugger* 接口时会停止进程分析。如果未停止分析，则目标进程的运行时链接程序内的数据结构在检查时可能处于不一致状态。

下图中显示了 `librtld_db.so.1`、控制进程（调试器）和目标进程（动态可执行文件）之间的信息流程。

图 11-1 *rtld-debugger* 信息流程



注 - *rtld-debugger* 接口依赖于 `proc_service` 接口 `/usr/include/proc_service.h`，后者被视为实验接口。*rtld-debugger* 接口可能必须跟踪 `proc_service` 接口在发展中的变化。

在 `/usr/demo/librtld_db` 下的 `pkg:/solaris/source/demo/system` 软件包中提供了使用 *rtld-debugger* 接口的控制进程的实现样例。此调试器 `rdb` 提供了使用 `proc_service` 导入接口的示例，并说明了所有 `librtld_db.so.1` 导出接口所需的调用顺序。以下各节介绍 *rtld-debugger* 接口。可以查看调试器样例，获取更多详细信息。

调试器接口代理

代理提供了可以描述内部接口结构的不透明处理方式，还提供了导出接口与导入接口之间的通信机制。*rtld-debugger* 接口旨在供可以同时处理多个进程的调试器使用，这些代理用于标识进程。

`struct ps_prochandle`

控制进程创建的不透明结构，用于标识在导出接口与导入接口之间传递的目标进程。

`struct rd_agent`

rtld-debugger 接口创建的不透明结构，用于标识在导出接口与导入接口之间传递的目标进程。

调试器导出接口

本节介绍 `/usr/lib/librtld_db.so.1` 审计库所导出的各种接口。可以将这些接口分为不同的功能组。

代理处理接口

`rd_init ()`

此函数可确定 *rtld-debugger* 的版本要求。基本 `version` 会定义为 `RD_VERSION1`。当前 `version` 始终由 `RD_VERSION` 定义。

```
rd_err_e rd_init(int version);
```

Solaris 8 10/00 发行版中添加的版本 `RD_VERSION2` 扩展了 `rd_loadobj_t` 结构。请参见“扫描可装入目标文件” [259]中的 `rl_flags`、`rl_bend` 和 `rl_dynamic` 字段。

Solaris 8 01/01 发行版中添加的版本 `RD_VERSION3` 扩展了 `rd_plt_info_t` 结构。请参见“跳过过程链接表” [264]中的 `pi_baddr` 和 `pi_flags` 字段。

如果控制进程要求的版本高于可用的 *rtld-debugger* 接口版本，则会返回 `RD_NOCAPAB`。

`rd_new ()`

此函数可创建新的导出接口代理。

```
rd_agent_t *rd_new(struct ps_prochandle *php);
```

`php` 是控制进程所创建的 cookie，用于标识目标进程。此 cookie 供控制进程提供的导入接口用于维护上下文，并且对于 *rtld-debugger* 接口是不透明的。

`rd_reset()`

此函数可基于为 `rd_new()` 提供的相同 `ps_prochandle` 结构重置代理内的信息。

```
rd_err_e rd_reset(struct rd_agent *rdap);
```

此函数在重新启动目标进程时调用。

`rd_delete()`

此函数可删除代理并释放与其关联的任何状态。

```
void rd_delete(struct rd_agent *rdap);
```

错误处理

rtld-debugger 接口（在 `rtld_db.h` 中定义）会返回以下错误状态：

```
typedef enum {
        RD_ERR,
        RD_OK,
        RD_NOCAPAB,
        RD_DBERR,
        RD_NOBASE,
        RD_NODYNAM,
        RD_NOMAPS
} rd_err_e;
```

以下接口可用于收集错误信息。

`rd_errstr()`

此函数可返回说明错误代码 `rderr` 的描述性错误字符串。

```
char *rd_errstr(rd_err_e rderr);
```

`rd_log()`

此函数可启用 (1) 或禁用 (0) 日志记录。

```
void rd_log(const int onoff);
```

启用日志记录时，会使用更多详细诊断信息来调用控制进程所提供的导入接口函数 `ps_plog()`。

扫描可装入目标文件

可以获取运行时链接程序中维护的每个目标文件的信息。通过使用 `rtld_db.h` 中定义的以下结构，可实现链接映射：

```
typedef struct rd_loadobj {
```

```
        psaddr_t        rl_nameaddr;
        unsigned        rl_flags;
        psaddr_t        rl_base;
        psaddr_t        rl_data_base;
        unsigned        rl_lmident;
        psaddr_t        rl_refnameaddr;
        psaddr_t        rl_plt_base;
        unsigned        rl_plt_size;
        psaddr_t        rl_bend;
        psaddr_t        rl_padstart;
        psaddr_t        rl_padend;
        psaddt_t        rl_dynamic;
        unsigned long   rl_tlsmodid;
} rd_loadobj_t;
```

请注意，在此结构中提供的所有地址（包括字符串指针）都是目标进程中的地址，而不是控制进程本身的地址空间中的地址。

`rl_nameaddr`

指向包含动态目标文件名称的字符串的指针。

`rl_flags`

在修订版 `RD_VERSION2` 中，使用 `RD_FLG_MEM_OBJECT` 标识动态装入的可重定位目标文件。

`rl_base`

动态目标文件的基本地址。

`rl_data_base`

动态目标文件数据段的基本地址。

`rl_lmident`

链接映射标识符（请参见“建立名称空间” [244]）。

`rl_refnameaddr`

如果动态目标文件是标准过滤器，则指向 *filtee* 的名称。

`rl_plt_base`、`rl_plt_size`

提供这些元素是为了向下兼容，当前未使用。

`rl_bend`

目标文件的结束地址 (`text + data + bss`)。在修订版 `RD_VERSION2` 中，动态装入的可重定位目标文件将导致此元素指向创建的目标文件（包括其节头）的结尾。

`rl_padstart`

动态目标文件之前填充的基本地址（请参阅“动态目标文件填充” [265]）。

rl_padend

动态目标文件之后填充的基本地址（请参阅“动态目标文件填充” [265]）。

rl_dynamic

添加了 RD_VERSION2 的此字段可提供目标文件动态节的基本地址，从而可允许引用 DT_CHECKSUM 之类的项（请参见表 13-8 “ELF 动态数组标记”）。

rl_tlsmodid

随 RD_VERSION4 添加的此字段为线程局部存储 TLS 引用提供了模块标识符。模块标识符是对目标文件唯一的一个小整数。可以将该标识符传递到 libc_db 函数 td_thr_tlsbase（），来为相关目标文件获取线程的 TLS 块的基本地址。请参见td_thr_tlsbase(3C_DB)。

rd_loadobj_iter（） 例程使用此目标文件数据结构来访问运行时链接程序的链接映射列表中的信息。

rd_loadobj_iter（）

对当前在目标进程中装入的所有动态目标文件重复执行此函数。

```
typedef int rl_iter_f(const rd_loadobj_t *, void *);

rd_err_e rd_loadobj_iter(rd_agent_t *rap, rl_iter_f *cb,
    void *clnt_data);
```

每次重复时都会调用 cb 指定的导入函数。可以使用 clnt_data 将数据传递给 cb 调用。通过指向可变（已分配的栈）rd_loadobj_t 结构的指针可返回有关每个目标文件的信息。

cb 例程中的返回代码通过 rd_loadobj_iter（） 进行检查，并具有以下含义：

- 1 – 继续处理链接映射。
- 0 – 停止处理链接映射并将控制权返回给控制进程。

rd_loadobj_iter（） 运行成功时会返回 RD_OK。返回 RD_NOMAPS 表示运行时链接程序尚未装入初始链接映射。

事件通知

控制进程可以跟踪运行时链接程序作用域内发生的特定事件。这些事件包括：

RD_PREINIT

运行时链接程序已经装入并重定位所有动态目标文件，并且即将开始调用每个装入的目标文件的 .init 节。

RD_POSTINIT

运行时链接程序已经完成调用所有的 .init 节，并且即将会将控制权转交给主可执行文件。

RD_DLACTIVITY

已经调用运行时链接程序来装入或卸载动态目标文件。

可以使用 sys/link.h 和 rtld_db.h 中定义的以下接口来监视这些事件：

```
typedef enum {
        RD_NONE = 0,
        RD_PREINIT,
        RD_POSTINIT,
        RD_DLACTIVITY
} rd_event_e;

/*
 * Ways that the event notification can take place:
 */
typedef enum {
        RD_NOTIFY_BPT,
        RD_NOTIFY_AUTOBPT,
        RD_NOTIFY_SYSCALL
} rd_notify_e;

/*
 * Information on ways that the event notification can take place:
 */
typedef struct rd_notify {
        rd_notify_e     type;
        union {
                psaddr_t        bptaddr;
                long            syscallno;
        } u;
} rd_notify_t;
```

以下函数可跟踪事件：

rd_event_enable ()

此函数可启用 (1) 或禁用 (0) 事件监视。

```
rd_err_e rd_event_enable(struct rd_agent *rdap, int onoff);
```

注 - 目前，由于性能原因，运行时链接程序会忽略事件禁用。控制进程应假定可以访问指定的断点，因为最后调用了此例程。

---

rd_event_addr ()

此函数可指定如何通知控制程序指定的事件。

```
rd_err_e rd_event_addr(rd_agent_t *rdap, rd_event_e event,
    rd_notify_t *notify);
```

根据事件类型，通过调用 notify->u.syscallno 标识的运行正常的低成本系统调用或者在 notify->u.bptaddr 指定的地址执行断点可实现控制进程通知。控制进程负责跟踪系统调用或定位实际断点。

事件发生后，可以通过 rtld_db.h 中定义的此接口获取其他信息：

```
typedef enum {
        RD_NOSTATE = 0,
        RD_CONSISTENT,
        RD_ADD,
        RD_DELETE
} rd_state_e;

typedef struct rd_event_msg {
        rd_event_e      type;
        union {
                rd_state_e      state;
        } u;
} rd_event_msg_t;
```

rd_state_e 值包括：

RD_NOSTATE
: 没有其他可用的状态信息。

RD_CONSISTANT
: 链接映射处于稳定状态，可以对其进行检查。

RD_ADD
: 正在装入动态目标文件，链接映射未处于稳定状态。应该在达到 RD_CONSISTANT 状态之后再检查这些链接映射。

RD_DELETE
: 正在删除动态目标文件，链接映射未处于稳定状态。应该在达到 RD_CONSISTANT 状态之后再检查这些链接映射。

rd_event_getmsg() 函数用于获取此事件状态信息。

rd_event_getmsg()
: 此函数可提供有关事件的其他信息。

```
rd_err_e rd_event_getmsg(struct rd_agent *rdap, rd_event_msg_t *msg);
```

下表显示了各种不同事件类型的可能状态。

RD_PREINIT	RD_POSTINIT	RD_DLACTIVITY
RD_NOSTATE	RD_NOSTATE	RD_CONSISTANT
		RD_ADD
		RD_DELETE

跳过过程链接表

通过使用 *rtld-debugger* 接口，控制进程可以跳过过程链接表项。第一次要求控制进程（如调试器）步入函数时，通过过程链接表处理可将控制权传递给运行时链接程序以搜索函数定义。

通过使用以下接口，控制进程可以跳过运行时链接程序的过程链接表处理。控制进程可以基于 ELF 文件中提供的外部信息来确定何时遇到过程链接表项。

目标进程步入过程链接表项之后，便会调用 `rd_plt_resolution()` 接口。

`rd_plt_resolution()`

此函数可返回当前过程链接表项的解析状态以及有关如何跳过此状态的信息。

```
rd_err_e rd_plt_resolution(rd_agent_t *rdap, paddr_t pc,
    lwpid_t lwpid, paddr_t plt_base, rd_plt_info_t *rpi);
```

`pc` 表示过程链接表项的第一条指令。`lwpid` 提供 `lwp` 标识符，`plt_base` 提供过程链接表的基本地址。这三个变量提供的信息足以供多个体系结构用于处理过程链接表。

`rpi` 提供有关以下数据结构（在 `rtld_db.h` 中定义）中定义的过程链接表项的详细信息：

```
typedef enum {
        RD_RESOLVE_NONE,
        RD_RESOLVE_STEP,
        RD_RESOLVE_TARGET,
        RD_RESOLVE_TARGET_STEP
} rd_skip_e;

typedef struct rd_plt_info {
        rd_skip_e       pi_skip_method;
        long            pi_nstep;
        psaddr_t        pi_target;
        psaddr_t        pi_baddr;
        unsigned int    pi_flags;
} rd_plt_info_t;

#define RD_FLG_PI_PLTBOUND     0x0001
```

`rd_plt_info_t` 结构的元素包括：

`pi_skip_method`

标识遍历过程链接表项的方法。此方法可设置为 `rd_skip_e` 值之一。

`pi_nstep`

标识返回 `RD_RESOLVE_STEP` 或 `RD_RESOLVE_TARGET_STEP` 时跳过的指令数。

`pi_target`

指定返回 `RD_RESOLVE_TARGET_STEP` 或 `RD_RESOLVE_TARGET` 时设置断点的地址。

pi_baddr

添加了 RD_VERSION3 的过程链接表的目标地址。设置 pi_flags 字段的 RD_FLG_PI_PLTBOUND 标志之后，此元素可标识已解析（绑定）的目标地址。

pi_flags

添加了 RD_VERSION3 的标志字段。标志 RD_FLG_PI_PLTBOUND 可将过程链接项标识为已解析（绑定）到其目标地址，此地址可用于 pi_baddr 字段。

rd_plt_info_t 返回值表明了以下可能的情况：

- 必须由运行时链接程序解析通过此过程链接表进行的首次调用。在这种情况下，rd_plt_info_t 包含以下内容：

  {RD_RESOLVE_TARGET_STEP, M, <BREAK>, 0, 0}

  控制进程会在 BREAK 处设置断点，从而使目标进程继续运行。到达断点时，即会完成过程链接表项处理。然后，控制进程可以将 M 条指令转到目标函数。请注意，由于这是通过过程链接表项进行的首次调用，因此尚未设置绑定地址 (pi_baddr)。
- 通过此过程链接表第 N 次进行调用时，rd_plt_info_t 会包含以下内容：

  {RD_RESOLVE_STEP, M, 0, <BoundAddr>, RD_FLG_PI_PLTBOUND}

  过程链接表项已经过解析，并且控制进程可以将 M 条指令转到目标函数。过程链接表项绑定到的地址为 <BoundAddr>，并且已在标志字段中设置了 RD_FLG_PI_PLTBOUND 位。

动态目标文件填充

运行时链接程序的缺省行为取决于要装入动态目标文件的操作系统（可以在其中最有效地引用这些目标文件）。如果能够对装入目标进程内存的目标文件执行填充，有些控制进程会从中受益。控制进程可以使用此接口请求此填充。

rd_objpad_enable ()

此函数可启用或禁用对目标进程的任何随后装入的目标文件的填充。可以在装入目标文件的两端进行填充。

```
rd_err_e rd_objpad_enable(struct rd_agent *rdap, size_t padsize);
```

padsize 指定将任何目标文件装入内存前后要保留的填充大小（以字节为单位）。该填充保留为 mmapobj(2) 请求中的内存映射。实际上，运行时链接程序可保留与任何装入目标文件相邻的目标进程虚拟地址空间区域。控制进程随后可以利用这些空间区域。

如果 padsize 为 0，则对于后续目标文件将禁用目标文件填充。

注 - 通过使用 proc(1) 工具并引用 rd_loadobj_t 中提供的链接映射信息，可报告使用 mmapobj(2) 获取的预留空间。

调试器导入接口

控制进程必须提供给 librtld_db.so.1 的导入接口在 /usr/include/proc_service.h 中定义。可以在 rdb 演示调试器中找到这些 proc_service 函数的实现样例。*rtld-debugger* 接口仅使用一部分可用的 proc_service 接口。将来版本的 *rtld-debugger* 接口可能会利用其他 proc_service 接口，而不会创建不兼容的更改。

当前 *rtld-debugger* 接口会使用以下接口：

ps_pauxv()

此函数可返回指向 auxv 向量副本的指针。

```
ps_err_e ps_pauxv(const struct ps_prochandle *ph, auxv_t **aux);
```

由于 auxv 向量信息会复制到已分配的结构，因此只要 ps_prochandle 有效，便会保留指针。

ps_pread()

此函数可从目标进程中读取数据。

```
ps_err_e ps_pread(const struct ps_prochandle *ph, paddr_t addr,
    char *buf, int size);
```

将 size 字节从目标进程中的地址 addr 复制到 buf。

ps_pwrite()

此函数可将数据写入目标进程。

```
ps_err_e ps_pwrite(const struct ps_prochandle *ph, paddr_t addr,
    char *buf, int size);
```

将 size 字节从 buf 复制到目标进程的地址 addr。

ps_plog()

此函数通过 *rtld-debugger* 接口中的其他诊断信息调用。

```
void ps_plog(const char *fmt, ....);
```

控制进程会确定在何处或者是否记录此诊断信息。ps_plog() 的参数采用 printf(3C) 格式。

ps_pglobal_lookup()

此函数可在目标进程中搜索符号。

```
ps_err_e ps_pglobal_lookup(const struct ps_prochandle *ph,
    const char *obj, const char *name, ulong_t *sym_addr);
```

在目标进程 ph 中的名为 obj 的目标文件中搜索名为 name 的符号。如果找到此符号，则将符号地址存储在 sym_addr 中。

ps_pglobal_sym ()

此函数可在目标进程中搜索符号。

```
ps_err_e ps_pglobal_sym(const struct ps_prochandle *ph,
    const char *obj, const char *name, ps_sym_t *sym_desc);
```

在目标进程 ph 中的名为 obj 的目标文件中搜索名为 name 的符号。如果找到此符号，则将符号描述符存储在 sym_desc 中。

如果在创建任何链接映射之前，*rtld-debugger* 接口需要在应用程序或运行时链接程序内查找符号，则可以使用 obj 的以下保留值：

```
#define PS_OBJ_EXEC ((const char *)0x0)  /* application id */
#define PS_OBJ_LDSO ((const char *)0x1)  /* runtime linker id */
```

控制进程可以使用以下伪代码将 procfs 文件系统用于这些目标文件：

```
ioctl(.., PIOCNAUXV, ....)       - obtain AUX vectors
ldsoaddr = auxv[AT_BASE];
ldsofd = ioctl(...., PIOCOPENM, &ldsoaddr);

/* process elf information found in ldsofd .... */

execfd = ioctl(.., PIOCOPENM, 0);

/* process elf information found in execfd .... */
```

找到文件描述符之后，控制程序即可检查 ELF 文件来查找其符号信息。

部分 IV

ELF 应用程序二进制接口

♦ ♦ ♦ 第 12 章 12

目标文件格式

本章介绍由汇编程序和链接编辑器生成的目标文件的可执行链接格式 (Excutable and Linking Format, ELF)。存在三种重要类型的目标文件。

- 可重定位目标文件包含代码节和数据节。此文件适合与其他可重定位目标文件链接，从而创建动态可执行文件、共享目标文件或其他可重定位目标文件。
- 动态可执行文件包含可随时执行的程序。此文件指定了 exec(2) 创建程序的进程映像的方式。此文件通常在运行时绑定到共享目标文件以创建进程映像。
- 共享目标文件文件包含适用于进行其他链接的代码和数据。链接编辑器可将此文件与其他可重定位目标文件和共享目标文件一起处理，以创建其他目标文件。运行时链接程序会将此文件与动态可执行文件和其他共享目标文件合并，以创建进程映像。

程序可以使用由 ELF 访问库 `libelf` 提供的函数处理目标文件。有关 `libelf` 内容的说明，请参见 elf(3ELF)。`/usr/demo/ELF` 目录下的 `pkg:/solaris/source/demo/system` 软件包中提供了使用 `libelf` 的源代码样例。

文件格式

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

图 12-1 目标文件格式

ELF 头位于目标文件的起始位置，其中包含用于说明文件结构的指南。

注 - 只有 ELF 头在文件中具有固定位置。由于 ELF 格式的灵活性，不要求头表、节或段具有指定的顺序。但是，此图是 Oracle Solaris OS 中使用的典型布局。

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

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

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

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

数据表示形式

目标文件格式支持 8 位字节、32 位体系结构和 64 位体系结构的各种处理器。不过，数据表示形式最好可扩展为更大或更小的体系结构。表 12-1 “ELF 32 位数据类型” 和表 12-2 “ELF 64 位数据类型” 列出了 32 位数据类型和 64 位数据类型。

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

表 12-1 ELF 32 位数据类型

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

表 12-2 ELF 64 位数据类型

名称	大小	对齐方式	用途
Elf64_Addr	8	8	无符号程序地址
Elf64_Half	2	2	无符号中整数
Elf64_Off	8	8	无符号文件偏移
Elf64_Sword	4	4	带符号整数

名称	大小	对齐方式	用途
Elf64_Word	4	4	无符号整数
Elf64_Xword	8	8	无符号长整数
Elf64_Sxword	8	8	带符号长整数
unsigned char	1	1	无符号小整数

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

注 - 为便于移植，ELF 不使用位字段。

ELF 头

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

ELF 头具有以下结构。请参见 sys/elf.h。

```
#define EI_NIDENT       16

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

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf64_Half      e_type;
        Elf64_Half      e_machine;
        Elf64_Word      e_version;
```

```
        Elf64_Addr      e_entry;
        Elf64_Off       e_phoff;
        Elf64_Off       e_shoff;
        Elf64_Word      e_flags;
        Elf64_Half      e_ehsize;
        Elf64_Half      e_phentsize;
        Elf64_Half      e_phnum;
        Elf64_Half      e_shentsize;
        Elf64_Half      e_shnum;
        Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

`e_ident`

将文件标记为目标文件的初始字节。这些字节提供与计算机无关的数据，用于解码和解释文件的内容。“ELF 标识” [277]中提供了完整说明。

`e_type`

标识目标文件类型，如下表中所列。

名称	值	含义
ET_NONE	0	无文件类型
ET_REL	1	可重定位文件
ET_EXEC	2	可执行文件
ET_DYN	3	共享目标文件
ET_CORE	4	核心文件
ET_LOSUNW	0xfefe	开始特定于操作系统的范围
ET_SUNW_ANCILLARY	0xfefe	辅助目标文件
ET_HISUNW	0xfefd	结束特定于操作系统的范围
ET_LOPROC	0xff00	开始特定于处理器的范围
ET_HIPROC	0xffff	结束特定于处理器的范围

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

`e_machine`

指定独立文件所需的体系结构。下表中列出了相关体系结构。

名称	值	含义
EM_NONE	0	无计算机
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_SPARC32PLUS	18	Sun SPARC 32+

名称	值	含义
EM_SPARCV9	43	SPARC V9
EM_AMD64	62	AMD 64

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

e_version

标识目标文件版本，如下表中所列。

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

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

e_entry

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

e_phoff

程序头表的文件偏移（字节）。如果文件没有程序头表，则此成员值为零。

e_shoff

节头表的文件偏移（字节）。如果文件没有节头表，则此成员值为零。

e_flags

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

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

名称	值	含义
EF_SPARCV9_RMO	0x2	非严格内存排序

e_ehsize

ELF 头的大小（字节）。

e_phentsize

文件的程序头表中某一项的大小（字节）。所有项的大小都相同。

e_phnum

程序头表中的项数。e_phentsize 和 e_phnum 的积指定了表的大小（字节）。如果文件没有程序头表，则 e_phnum 值为零。

如果程序头的数量大于或等于 PN_XNUM (0xffff)，则此成员的值为 PN_XNUM (0xffff)。程序头表的实际项数包含在节头中索引为 0 的 sh_info 字段中。否则，初始节头项的 sh_info 成员值为零。请参见表 12-6 “ELF 节头表项：索引 0” 和表 12-7 “ELF 扩展的节头表项：索引 0”。

e_shentsize

节头的大小（字节）。节头是节头表中的一项。所有项的大小都相同。

e_shnum

节头表中的项数。e_shentsize 和 e_shnum 的积指定了节头表的大小（字节）。如果文件没有节头表，则 e_shnum 值为零。

如果节数大于或等于 SHN_LORESERVE (0xff00)，则 e_shnum 值为零。节头表的实际项数包含在节头中索引为 0 的 sh_size 字段中。否则，初始节头项的 sh_size 成员值为零。请参见表 12-6 “ELF 节头表项：索引 0” 和表 12-7 “ELF 扩展的节头表项：索引 0”。

e_shstrndx

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

如果节名称字符串表的节索引大于或等于 SHN_LORESERVE (0xff00)，则此成员值为 SHN_XINDEX (0xffff)，并且节名称字符串表的实际节索引包含在节头中索引为 0 的 sh_link 字段中。否则，初始节头项的 sh_link 成员值为零。请参见表 12-6 “ELF 节头表项：索引 0” 和表 12-7 “ELF 扩展的节头表项：索引 0”。

ELF 标识

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

ELF 头和目标文件的初始字节对应于 e_ident 成员。

表 12-3 ELF 标识索引

名称	值	用途
EI_MAG0	0	文件标识
EI_MAG1	1	文件标识
EI_MAG2	2	文件标识
EI_MAG3	3	文件标识
EI_CLASS	4	文件类
EI_DATA	5	数据编码
EI_VERSION	6	文件版本
EI_OSABI	7	操作系统/ABI 标识
EI_ABIVERSION	8	ABI 版本
EI_PAD	9	填充字节的开头
EI_NIDENT	16	e_ident[] 的大小

这些索引可访问值为以下各项的字节。

EI_MAG0 - EI_MAG3

4 字节魔数，用于将文件标识为 ELF 目标文件，如下表中所列。

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

EI_CLASS

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

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

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

类 ELFCLASS32 支持文件和虚拟地址空间最高为 4 GB 的计算机。该类使用表 12-1 “ELF 32 位数据类型” 中定义的基本类型。

类 ELFCLASS64 保留用于 64 位体系结构，如 64 位 SPARC 和 x64。该类使用表 12-2 “ELF 64 位数据类型” 中定义的基本类型。

EI_DATA

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

名称	值	含义
ELFDATANONE	0	无效数据编码
ELFDATA2LSB	1	请参见图 12-2 “数据编码 ELFDATA2LSB”。
ELFDATA2MSB	2	请参见图 12-3 “数据编码 ELFDATA2MSB”。

“数据编码” [280]一节中提供了有关这些编码的更多信息。其他值保留供将来使用。

EI_VERSION

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

EI_OSABI

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

名称	值	含义
ELFOSABI_NONE / ELFOSABI_SYSV	0	无扩展或未指定
ELFOSABI_SOLARIS	6	Solaris

EI_ABIVERSION

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

EI_PAD

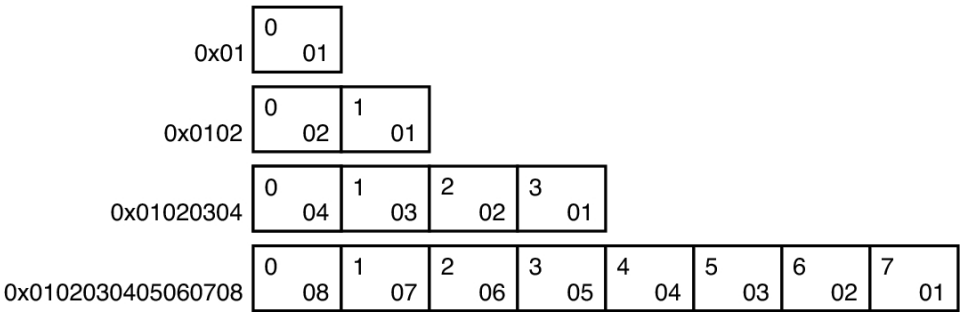
该值用于标记 e_ident 中未使用字节的起始位置。这些字节会保留并设置为零。读取目标文件的程序应忽略这些值。

数据编码

文件的数据编码指定解释文件中的整数类型的方式。类 ELFCLASS32 文件和类 ELFCLASS64 文件使用占用 1、2、4 和 8 个字节的整数来表示偏移、地址和其他信息。按照定义的编码，目标文件使用如下描述的数字表示。字节编号显示在左上角。

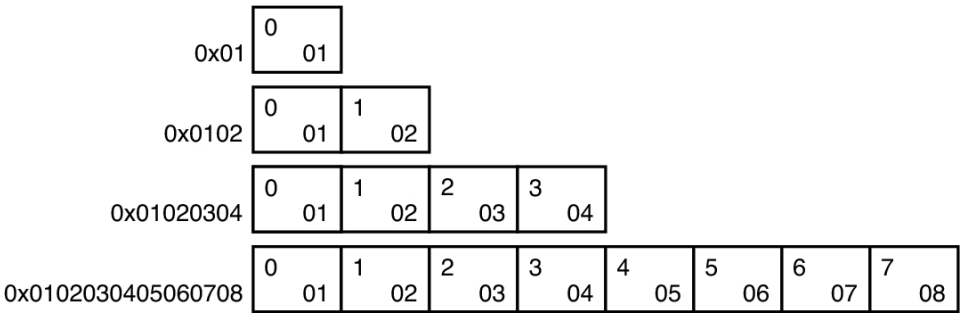
ELFDATA2LSB 编码用于指定 2 的补码值，其中最低有效字节占用最低地址。在非正式情况下，这种编码通常称为小尾数法字节排序。

图 12-2 数据编码 ELFDATA2LSB



ELFDATA2MSB 编码用于指定 2 的补码值，其中最高有效字节占用最低地址。在非正式情况下，这种编码通常称为大尾数法字节排序。

图 12-3 数据编码 ELFDATA2MSB



节

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

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

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

表 12-4 ELF 特殊节索引

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

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

SHN_UNDEF

未定义、缺少、无关或无意义的节引用。例如，定义的相对于节编号 SHN_UNDEF 的符号即是未定义符号。

SHN_LORESERVE

所保留索引的范围下界。

SHN_LOPROC - SHN_HIPROC

此范围内包含的值（包括这两个值）保留用于特定于处理器的语义。

SHN_LOOS - SHN_HIOS

此范围内包含的值（包括这两个值）保留用于特定于操作系统的语义。

SHN_LOSUNW - SHN_HISUNW

此范围内包含的值（包括这两个值）保留用于特定于 Sun 的语义。

SHN_SUNW_IGNORE

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

SHN_BEFORE、SHN_AFTER

与 SHF_LINK_ORDER 和 SHF_ORDERED 节标志一起提供初始和最终节排序。请参见表 12-8 “ELF 节属性标志”。

SHN_AMD64_LCOMMON

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

SHN_ABS

对应引用的绝对值。例如，相对于节编号 SHN_ABS 而定义的符号具有绝对值，不受重定位影响。

SHN_COMMON

相对于此节而定义的符号为通用符号，如 FORTRAN COMMON 或未分配的 C 外部变量。这些符号有时称为暂定符号。

SHN_XINDEX

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

SHN_HIRESERVE

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

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

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

节头具有以下结构。请参见 `sys/elf.h`。

```
typedef struct {
        elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;

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

`sh_name`

节的名称。此成员值是节头字符串表节的索引，用于指定以空字符结尾的字符串的位置。表 12-12 “ELF 特殊节” 中列出了节名称及其说明。

`sh_type`

用于将节的内容和语义分类。表 12-5 “ELF 节类型 `sh_type`” 中列出了节类型及其说明。

`sh_flags`

节可支持用于说明杂项属性的 1 位标志。表 12-8 “ELF 节属性标志” 中列出了标志定义。

`sh_addr`

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

sh_offset

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

sh_size

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

sh_link

节头表索引链接，其解释依赖于节类型。表 12-9 “ELF sh_link 和 sh_info 解释” 说明了相应的值。

sh_info

额外信息，其解释依赖于节类型。表 12-9 “ELF sh_link 和 sh_info 解释” 说明了相应的值。如果此节头的 sh_flags 字段包含属性 SHF_INFO_LINK，则此成员表示节头表索引。

sh_addralign

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

sh_entsize

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

节头的 sh_type 成员用于指定节的语义，如下表中所示。

表 12-5 ELF 节类型 sh_type

名称	值
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7
SHT_NOBITS	8
SHT_REL	9

名称	值
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_INIT_ARRAY	14
SHT_FINI_ARRAY	15
SHT_PREINIT_ARRAY	16
SHT_GROUP	17
SHT_SYMTAB_SHNDX	18
SHT_LOOS	0x60000000
SHT_LOSUNW	0x6ffffffe
SHT_SUNW_ancillary	0x6ffffffe
SHT_SUNW_capchain	0x6fffffef
SHT_SUNW_capinfo	0x6ffffff0
SHT_SUNW_symsort	0x6ffffff1
SHT_SUNW_tlssort	0x6ffffff2
SHT_SUNW_LDYNSYM	0x6ffffff3
SHT_SUNW_dof	0x6ffffff4
SHT_SUNW_cap	0x6ffffff5
SHT_SUNW_SIGNATURE	0x6ffffff6
SHT_SUNW_ANNOTATE	0x6ffffff7
SHT_SUNW_DEBUGSTR	0x6ffffff8
SHT_SUNW_DEBUG	0x6ffffff9
SHT_SUNW_move	0x6ffffffa
SHT_SUNW_COMDAT	0x6ffffffb
SHT_SUNW_syminfo	0x6ffffffc
SHT_SUNW_verdef	0x6ffffffd
SHT_SUNW_verneed	0x6ffffffe
SHT_SUNW_versym	0x6fffffff
SHT_HISUNW	0x6fffffff
SHT_HIOS	0x6fffffff
SHT_LOPROC	0x70000000
SHT_SPARC_GOTDATA	0x70000000
SHT_AMD64_UNWIND	0x70000001
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

SHT_NULL

将节头标识为无效。此节头没有关联的节。节头的其他成员具有未定义的值。

SHT_PROGBITS

标识由程序定义的信息，这些信息的格式和含义完全由程序确定。

SHT_SYMTAB、SHT_DYNSYM、SHT_SUNW_LDYNSYM

标识符号表。通常，SHT_SYMTAB 节会提供用于链接编辑的符号。作为完整的符号表，该表可能包含许多对于动态链接不必要的符号。因此，目标文件还可以包含一个 SHT_DYNSYM 节，其中包含一组尽可能少的动态链接符号，从而节省空间。

SHT_DYNSYM 还可以使用 SHT_SUNW_LDYNSYM 节进行扩充。此附加节为运行时环境提供局部函数符号，但对于动态链接来说不是必需的。当不可分配的 SHT_SYMTAB 不可用或已从文件中剥离时，调试器通过使用此节可在运行时上下文中产生精确的栈跟踪。此节还可以为运行时环境提供其他符号信息，以便与 dladdr(3C) 一起使用。

如果 SHT_SUNW_LDYNSYM 节和 SHT_DYNSYM 节同时存在，链接编辑器会将这两者的数据区域紧邻彼此放置。SHT_SUNW_LDYNSYM 节位于 SHT_DYNSYM 节的前面。这种放置方式可以使两个表看起来像是一个更大的连续符号表，其中包含 SHT_SYMTAB 中的缩减符号集合。

有关详细信息，请参见“符号表节” [326]。

SHT_STRTAB、SHT_DYNSTR

标识字符串表。目标文件可以有多个字符串表节。有关详细信息，请参见“字符串表节” [325]。

SHT_RELA

标识包含显式加数的重定位项，如 32 位目标文件类的 Elf32_Rela 类型。目标文件可以有多个重定位节。有关详细信息，请参见“重定位节” [314]。

SHT_HASH

标识符号散列表。动态链接的目标文件必须包含符号散列表。当前，目标文件只能有一个散列表，但此限制在将来可能会放宽。有关详细信息，请参见“散列表节” [308]。

SHT_DYNAMIC

标识动态链接的信息。当前，目标文件只能有一个动态节。有关详细信息，请参见“动态节” [356]。

SHT_NOTE

标识以某种方法标记文件的信息。有关详细信息，请参见“注释节” [312]。

SHT_NOBITS

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

SHT_REL

标识不包含显式加数的重定位项，如 32 位目标文件类的 Elf32_Rel 类型。目标文件可以有多个重定位节。有关详细信息，请参见“重定位节” [314]。

SHT_SHLIB

标识具有未指定的语义的保留节。包含此类型的节的程序不符合 ABI。

SHT_INIT_ARRAY

标识包含指针数组的节，这些指针指向初始化函数。数组中的每个指针都被视为不返回任何值的无参数过程。有关详细信息，请参见“初始化节和终止节” [33]。

SHT_FINI_ARRAY

标识包含指针数组的节，这些指针指向终止函数。数组中的每个指针都被视为不返回任何值的无参数过程。有关详细信息，请参见“初始化节和终止节” [33]。

SHT_PREINIT_ARRAY

标识包含指针数组的节，这些指针指向在其他所有初始化函数之前调用的函数。数组中的每个指针都被视为不返回任何值的无参数过程。有关详细信息，请参见“初始化节和终止节” [33]。

SHT_GROUP

标识节组。节组标识一组相关的节，这些节必须作为一个单元由链接编辑器进行处理。SHT_GROUP 类型的节只能出现在可重定位目标文件中。有关详细信息，请参见“组节” [305]。

SHT_SYMTAB_SHNDX

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

SHT_SYMTAB_SHNDX 节是 Elf32_Word 值的数组。对于关联的符号表项中的每一项，此数组都包含对应的一项。这些值表示针对其定义符号表项的节头索引。仅当对应符号表项的 st_shndx 字段包含转义值 SHN_XINDEX 时，匹配的 Elf32_Word 才会包含实际的节头索引。否则，该项必须为 SHN_UNDEF (0)。

SHT_LOOS – SHT_HIOS

此范围内包含的值（包括这两个值）保留用于特定于操作系统的语义。

SHT_LOSUNW – SHT_HISUNW

此范围内包含的值（包括这两个值）保留用于 Oracle Solaris OS 语义。

SHT_SUNW_ancillary

表示该目标文件是一组辅助目标文件的一部分。包含标识构成该组的所有文件所需要的信息。有关详细信息，请参见“辅助节” [303]。

SHT_SUNW_capchain

收集功能系列成员的索引数组。该数组的第一个元素是链版本号。此元素的后面是以 0 结尾的功能符号索引链。每个以 0 结尾的索引组都表示一个功能系列。每个系列的第一个元素是功能前置符号。后面的元素指向系列成员。有关详细信息，请参见“功能节” [306]。

SHT_SUNW_capinfo

将符号表项与功能要求及其前置功能符号关联起来的索引数组。定义符号功能的目标文件包含 SHT_SUNW_cap 节。SHT_SUNW_cap 节头信息指向关联的 SHT_SUNW_capinfo 节。SHT_SUNW_capinfo 节头信息指向关联的符号表节。有关详细信息，请参见“功能节” [306]。

SHT_SUNW_symsort

由相邻的 SHT_SUNW_LDYNSYM 节和 SHT_DYNSYM 节构成的动态符号表的索引数组。这些索引相对于 SHT_SUNW_LDYNSYM 节的开头。这些索引引用包含内存地址的符号。该索引已进行排序，以使索引按照地址递增的顺序引用符号。

SHT_SUNW_tlssort

由相邻的 SHT_SUNW_LDYNSYM 节和 SHT_DYNSYM 节构成的动态符号表的索引数组。这些索引相对于 SHT_SUNW_LDYNSYM 节的开头。这些索引引用线程局部存储符号。请参见第 14 章 线程局部存储。该索引已进行排序，以使索引按照偏移递增的顺序引用符号。

SHT_SUNW_LDYNSYM

非全局符号的动态符号表。请参见前面的 SHT_SYMTAB、SHT_DYNSYM、SHT_SUNW_LDYNSYM 说明。

SHT_SUNW_dof

保留供 dtrace(1M) 内部使用。

SHT_SUNW_cap

指定功能要求。有关详细信息，请参见“功能节” [306]。

SHT_SUNW_SIGNATURE

标识模块验证签名。

SHT_SUNW_ANNOTATE

注释节的处理遵循处理节的所有缺省规则。仅当注释节位于不可分配的内存中时，才会发生异常。如果未设置节头标志 SHF_ALLOC，则链接编辑器将在不给出任何提示的情况下忽略针对此节的所有未满足的重定位。

SHT_SUNW_DEBUGSTR、SHT_SUNW_DEBUG

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

SHT_SUNW_move

标识用于处理部分初始化的符号的数据。有关详细信息，请参见“移动部分” [310]。

SHT_SUNW_COMDAT
: 标识允许将相同数据的多个副本减少为单个副本的节。有关详细信息，请参见“*COMDAT* 节” [304]。

SHT_SUNW_syminfo
: 标识其他符号信息。有关详细信息，请参见“*Syminfo* 表节” [336]。

SHT_SUNW_verdef
: 标识此文件定义的细分版本。有关详细信息，请参见“版本定义章节” [338]。

SHT_SUNW_verneed
: 标识此文件所需的细分依赖性。有关详细信息，请参见“版本依赖性节” [339]。

SHT_SUNW_versym
: 标识用于说明符号与文件提供的版本定义之间关系的表。有关详细信息，请参见“版本符号节” [341]。

SHT_LOPROC - SHT_HIPROC
: 此范围内包含的值（包括这两个值）保留用于特定于处理器的语义。

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

SHT_AMD64_UNWIND
: 标识特定于 x64 的数据，其中包含对应于栈展开的展开函数表的各项。

SHT_LOUSER
: 指定保留用于应用程序的索引范围的下界。

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

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

表 12-6 ELF 节头表项：索引 0

名称	值	备注
sh_name	0	无名称
sh_type	SHT_NULL	无效
sh_flags	0	无标志

名称	值	备注
sh_addr	0	无地址
sh_offset	0	无文件偏移
sh_size	0	无大小
sh_link	SHN_UNDEF	无链接信息
sh_info	0	无辅助信息
sh_addralign	0	无对齐
sh_entsize	0	无项

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

表 12-7 ELF 扩展的节头表项：索引 0

名称	值	备注
sh_name	0	无名称
sh_type	SHT_NULL	无效
sh_flags	0	无标志
sh_addr	0	无地址
sh_offset	0	无文件偏移
sh_size	e_shnum	节头表中的项数
sh_link	e_shstrndx	与节名称字符串表关联的项的节头索引
sh_info	e_phnum	程序头表中的项数
sh_addralign	0	无对齐
sh_entsize	0	无项

节头的 sh_flags 成员包含用于说明节属性的 1 位标志。

表 12-8 ELF 节属性标志

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

名称	值
SHF_GROUP	0x200
SHF_TLS	0x400
SHF_COMPRESSED	0x800
SHF_MASKOS	0x0ff00000
SHF_SUNW_NODISCARD	0x00100000
SHF_SUNW_ABSENT	0x00200000
SHF_SUNW_PRIMARY	0x00400000
SHF_MASKPROC	0xf0000000
SHF_AMD64_LARGE	0x10000000
SHF_ORDERED	0x40000000
SHF_EXCLUDE	0x80000000

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

SHF_WRITE

标识在进程执行过程中应可写的节。

SHF_ALLOC

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

SHF_EXECINSTR

标识包含可执行计算机指令的节。

SHF_MERGE

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

SHF_STRINGS

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

SHF_INFO_LINK

此节头的 sh_info 字段包含节头表索引。

SHF_LINK_ORDER

此节向链接编辑器中添加特殊排序要求。如果此节头的 sh_link 字段引用其他节（链接到的节），则会应用这些要求。如果将此节与输出文件中的其他节合并，则

此节将按照相同的相对顺序（相对于这些节）显示。同样，链接到的节也将按照相同的相对顺序（相对于与其合并的节）显示。链接到的节必须是未排序的，因而不能指定 SHF_LINK_ORDER 或 SHF_ORDERED。

特殊的 sh_link 值 SHN_BEFORE 和 SHN_AFTER（请参见表 12-4 “ELF 特殊节索引”）分别表示已排序的节将位于要排序的集合中的其他所有节之前或之后。如果已排序集合中的多个节包含这些特殊值之一，则会保持输入文件链接行的顺序。

此标志的一个典型用法是生成按地址顺序引用文本或数据节的表。

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

SHF_OS_NONCONFORMING

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

SHF_GROUP

此节是节组的一个成员（可能是唯一的成员）。此节必须由 SHT_GROUP 类型的节引用。只能对可重定位目标文件中包含的节设置 SHF_GROUP 标志。有关详细信息，请参见“组节” [305]。

SHF_TLS

此节包含线程局部存储。进程中的每个线程都包含此数据的一个不同实例。有关详细信息，请参见第 14 章 线程局部存储。

SHF_COMPRESSED

标识包含压缩数据的节。SHF_COMPRESSED 仅适用于不可分配节，且不能与 SHF_ALLOC 一起使用。此外，SHF_COMPRESSED 不适用于类型为 SHT_NOBITS 的节。有关详细信息，请参见“节压缩” [295]。

SHF_MASKOS

此掩码中包括的所有位都保留用于特定于操作系统的语义。

SHF_SUNW_NODISCARD

此节无法通过链接编辑器丢弃，并且始终会被复制到输出目标文件中。链接编辑器提供了通过链接编辑放弃未使用的输入节的功能。SHF_SUNW_NODISCARD 节标志将该节排除在此类优化之外。

SHF_SUNW_ABSENT

表示该节的数据在此文件中不存在。创建辅助目标文件后，主目标文件和任何辅助目标文件都具有相同的节头数组。该组织便利了这些目标文件中包含的信息的合并，并允许使用单个符号表。每个文件都包含节数据的一个子集。可分配节的数据将写入主目标文件，而不可分配节的数据将写入辅助文件。SHF_SUNW_ABSENT 标志表

示所检查的目标文件中不存在该节的数据。设置 SHF_SUNW_ABSENT 标志后，节头的 sh_size 字段必须是 0。遇到 SHF_SUNW_ABSENT 节的应用程序可以选择忽略该节，或选择在其中一个相关辅助文件中搜索节数据。请参见"调试器访问及辅助目标文件使用" [77]。

SHF_SUNW_PRIMARY

创建辅助目标文件时的缺省行为是：将所有可分配节写入主目标文件并将所有不可分配节写入辅助目标文件。SHF_SUNW_PRIMARY 标志可覆盖此行为。包含另一个设置有 SHF_SUNW_PRIMARY 标志的输入节的任何输出节将写入到主目标文件中。

SHF_MASKPROC

此掩码中包括的所有位都保留用于特定于处理器的语义。

SHF_AMD64_LARGE

x64 的缺省编译模型仅用于 32 位位移。此位移限制了节的大小，并最终限制段的大小不得超过 2 GB。此属性标志用于标识可包含超过 2 GB 数据的节。此标志允许链接使用不同代码模型的目标文件。

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

SHF_ORDERED

SHF_ORDERED 是 SHF_LINK_ORDER 所提供的旧版本功能，并且已被 SHF_LINK_ORDER 取代。SHF_ORDERED 提供两种不同的功能。首先，可以指定输出节，其次，链接编辑器具有特殊排序要求。

SHF_ORDERED 节的 sh_link 字段可以构成节的链接列表。此列表以包含指向其自身的 sh_link 的最终节结束。此列表中的所有节都将指定给具有此列表中最终节的名称的输出节。

如果已排序节的 sh_info 项在相同输入文件中是有效节，则将基于由 sh_info 项所指向的节的输出文件内的相对排序，对已排序的节进行排序。sh_info 项指向的节必须尚未排序，因而不能指定 SHF_LINK_ORDER 或 SHF_ORDERED。

特殊的 sh_info 值 SHN_BEFORE 和 SHN_AFTER（请参见表 12-4 "ELF 特殊节索引"）分别表示已排序的节将位于要排序的集合中的其他所有节之前或之后。如果已排序集合中的多个节包含这些特殊值之一，则会保持输入文件链接行的顺序。

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

SHF_EXCLUDE

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

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

表 12-9 ELF sh_link 和 sh_info 解释

sh_type	sh_link	sh_info
SHT_DYNAMIC	关联的字符串表的节头索引。	0
SHT_HASH	关联的符号表的节头索引。	0
SHT_REL SHT_RELA	关联的符号表的节头索引。	应用重定位的节的节头索引，否则是 0。另请参见表 12-12 “ELF 特殊节” 和“重定位节” [314]。
SHT_SYMTAB SHT_DYNSYM	关联的字符串表的节头索引。	比上一个局部符号 STB_LOCAL 的符号表索引大一。
SHT_GROUP	关联的符号表的节头索引。	关联的符号表中项的符号表索引。指定的符号表项的名称用于提供节组的签名。
SHT_SYMTAB_SHNDX	关联的符号表的节头索引。	0
SHT_SUNW_ancillary	关联的字符串表的节头索引。	0
SHT_SUNW_cap	如果符号功能存在，则为关联的 SHT_SUNW_capinfo 表的节头索引，否则为 0。	如果任何功能引用指定的字符串，则为关联的字符串表的节头索引，否则为 0。
SHT_SUNW_capinfo	关联的符号表的节头索引。	对于动态目标文件，为关联的 SHT_SUNW_capchain 表的节头索引，否则为 0。
SHT_SUNW_symsort	关联的符号表的节头索引。	0
SHT_SUNW_tlssort	关联的符号表的节头索引。	0
SHT_SUNW_LDYNSYM	关联的字符串表的节头索引。此索引是与 SHT_DYNSYM 节所使用的字符串表相同的字符串表。	比上一个局部符号 STB_LOCAL 的符号表索引大一。由于 SHT_SUNW_LDYNSYM 仅包含局部符号，因此 sh_info 等于表中的符号数。
SHT_SUNW_move	关联的符号表的节头索引。	0
SHT_SUNW_COMDAT	0	0
SHT_SUNW_syminfo	关联的符号表的节头索引。	关联的 .dynamic 节的节头索引。
SHT_SUNW_verdef	关联的字符串表的节头索引。	节中版本定义的数量。
SHT_SUNW_verneed	关联的字符串表的节头索引。	节中版本依赖性的数量。
SHT_SUNW_versym	关联的符号表的节头索引。	0

节合并

SHF_MERGE 节标志可用于标记可重定位目标文件中的 SHT_PROGBITS 节。请参见表 12-8 “ELF 节属性标志”。此标志表示该节可以与其他目标文件中的兼容节合并。这类合并有可能减小通过这些可重定位目标文件生成的任何可执行文件或共享目标文件的大小。减小文件大小还有助于改善生成的目标文件的运行时性能。

带有 SHF_MERGE 标志的节表示该节遵循以下特征：

- 该节为只读。包含该节的程序在运行时绝对不可能修改该节的数据。
- 从单独的重定位记录可以访问该节中的每一项。生成访问这些项的代码时，程序代码无法针对该节中的项的相对位置做出任何假设。
- 如果该节还设置了 SHF_STRINGS 标志，那么该节只能包含以空字符结尾的字符串。空字符只能作为字符串结束符，而不能出现在任何字符串的中间位置。

SHF_MERGE 是一个可选标志，用于表示进行优化的可能性。允许链接编辑器执行优化，或忽略优化。任一情况下，链接编辑器都会创建一个有效的输出目标文件。当前，链接编辑器仅对包含使用 SHF_STRINGS 标志进行标记的字符串数据的节执行节合并。

同时设置了 SHF_STRINGS 节标志和 SHF_MERGE 标志时，该节中的字符串就可以与其他兼容节中的字符串合并。链接编辑器使用与用于压缩 SHT_STRTAB 字符串表 (.strtab 和 .dynstr) 的字符串压缩算法相同的字符串压缩算法来合并此类节。

- 重复字符串会缩减为一个。
- 会消除尾部字符串。例如，如果输入节包含字符串 "*bigdog*" 和 "dog"，那么将消除较小的 "dog"，并使用较大的字符串的尾部表示较小的字符串。

目前，链接编辑器仅对由没有特殊对齐限制的单字节大小字符组成的字符串执行字符串合并。具体来说，必须具备以下节特征。

- sh_entsize 必须为 0 或 1。不支持包含宽字符的节。
- 仅合并字节对齐的节，其中 sh_addralign 为 0 或 1。

注 - 可以使用链接编辑器的 -z nocompstrtab 选项抑制任何字符串表压缩。

节压缩

SHF_COMPRESSED 节标志标识包含压缩数据的节。SHF_COMPRESSED 仅适用于不可分配节，且不能与 SHF_ALLOC 一起使用。此外，SHF_COMPRESSED 不适用于类型为 SHT_NOBITS 的节。

任何必须应用于压缩节的重定位都将指定未压缩节数据的偏移。因此，必须在应用重定位之前解压缩节数据。每个压缩节都会独立指定算法。给定 ELF 目标文件中的不同节可以采用不同的压缩算法。

压缩节以可标识压缩算法的压缩头结构开头。

```
typedef struct {
        Elf32_Word      ch_type;
        Elf32_Word      ch_size;
        Elf32_Word      ch_addralign;
} Elf32_Chdr;
```

```
typedef struct {
        Elf64_Word      ch_type;
        Elf64_Word      ch_reserved;
        Elf64_Xword     ch_size;
        Elf64_Xword     ch_addralign;
} Elf64_Chdr;
```

ch_type

指定压缩算法。表 12-10 “ELF 压缩类型 ch_type” 中列出了支持的算法及相关说明。

ch_size

未压缩数据的大小（字节）。请参见 sh_size。

ch_addralign

未压缩数据所需的对齐。请参见 sh_addralign。

压缩节头的 sh_size 和 sh_addralign 字段反映了压缩节的要求。压缩头的 ch_size 和 ch_addralign 字段提供未压缩数据的对应值，从而提供此节未压缩时 sh_size 和 sh_addralign 字段应具有的值。

压缩头后的数据的布局和解释特定于各个算法。除了压缩数据字节，此布局还可能包含特定于算法的参数和对齐填充。

压缩头的 ch_type 成员指定采用的压缩算法，如下表中所示。

表 12-10 ELF 压缩类型 ch_type

名称	值
ELFCOMPRESS_ZLIB	1
ELFCOMPRESS_LOOS	0x60000000
ELFCOMPRESS_HIOS	0x6fffffff
ELFCOMPRESS_LOPROC	0x70000000
ELFCOMPRESS_HIPROC	0x7fffffff

ELFCOMPRESS_ZLIB

使用 ZLIB 压缩算法对节数据进行压缩。压缩 ZLIB 数据字节以紧跟在压缩头后的字节开头，一直到节的末尾。可在以下网址找到 ZLIB 的文档：http://www.zlib.net/。

ELFCOMPRESS_LOOS - ELFCOMPRESS_HIOS

此范围内包含的值（包括这两个值）保留用于特定于操作系统的语义。

ELFCOMPRESS_LOPROC - ELFCOMPRESS_HIPROC

此范围内包含的值（包括这两个值）保留用于特定于处理器的语义。

GNU 样式的节压缩

除了上述压缩格式，Oracle Solaris 链接编辑器还识别一种由 GNU 工具链使用的备用格式。此格式不采用节标志来表示压缩。而是用以 .zdebug 前缀开头的节名称标识包含压缩数据的节。GNU 样式的压缩节以以下压缩头结构开头：

```
typedef struct {
        uchar_t         gch_magic[4];
        uchar_t         gch_size[8];
} Chdr_GNU;
```

gch_magic

用 4 字节魔数标识压缩算法。目前仅支持 ZLIB 压缩。ZLIB 压缩的 gch_magic 值如表 12-11 “GNU ZLIB 压缩，gch_magic” 中所示。

gch_size

未压缩数据的大小（字节），编码为 64 位 ELFDATA2MSB 大尾数整数值。

表 12-11 GNU ZLIB 压缩，gch_magic

名称	值
gch_magic[0]	'Z'
gch_magic[1]	'L'
gch_magic[2]	'I'
gch_magic[3]	'B'

特殊节

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

表 12-12 ELF 特殊节

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

名称	类型	属性
.fini_array	SHT_FINI_ARRAY	SHF_ALLOC + SHF_WRITE
.got	SHT_PROGBITS	请参见“全局偏移表（特定于处理器）” [371]
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.init_array	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE
.interp	SHT_PROGBITS	请参见“程序的解释程序” [355]
.note	SHT_NOTE	无
.lbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_AMD64_LARGE
.ldata、.ldata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_AMD64_LARGE
.lrodata、.lrodata1	SHT_PROGBITS	SHF_ALLOC + SHF_AMD64_LARGE
.plt	SHT_PROGBITS	请参见“过程链接表（特定于处理器）” [372]
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC + SHF_WRITE
.rela	SHT_RELA	无
.rel*name*	SHT_REL	请参见“重定位节” [314]
.rela*name*	SHT_RELA	请参见“重定位节” [314]
.rodata、.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	无
.strtab	SHT_STRTAB	请参阅此表后面的说明。
.symtab	SHT_SYMTAB	请参见“符号表节” [326]
.symtab_shndx	SHT_SYMTAB_SHNDX	请参见“符号表节” [326]
.tbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata、.tdata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.SUNW_ancillary	SHT_SUNW_ancillary	无
.SUNW_bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.SUNW_cap	SHT_SUNW_cap	SHF_ALLOC
.SUNW_capchain	SHT_SUNW_capchain	SHF_ALLOC
.SUNW_capinfo	SHT_SUNW_capinfo	SHF_ALLOC
.SUNW_heap	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.SUNW_ldynsym	SHT_SUNW_LDYNSYM	SHF_ALLOC
.SUNW_dynsymsort	SHT_SUNW_symsort	SHF_ALLOC
.SUNW_dymtlssort	SHT_SUNW_tlssort	SHF_ALLOC
.SUNW_move	SHT_SUNW_move	SHF_ALLOC
.SUNW_reloc	SHT_REL	SHF_ALLOC
	SHT_RELA	

名称	类型	属性
.SUNW_syminfo	SHT_SUNW_syminfo	SHF_ALLOC
.SUNW_version	SHT_SUNW_verdef	SHF_ALLOC
	SHT_SUNW_verneed	
	SHT_SUNW_versym	

.bss

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

.comment

注释信息，通常由编译系统的组件提供。此节可以由 mcs(1) 进行处理。

.data、.data1

构成程序的内存映像的已初始化数据。

.dynamic

动态链接信息。有关详细信息，请参见“动态节” [356]。

.dynstr

进行动态链接所需的字符串，通常是表示与符号表各项关联的名称的字符串。

.dynsym

动态链接符号表。有关详细信息，请参见“符号表节” [326]。

.eh_frame_hdr、.eh_frame

用于展开栈的调用帧信息。

.fini

可执行指令，用于构成包含此节的可执行文件或共享目标文件的单个终止函数。有关详细信息，请参见“初始化和终止例程” [100]。

.fini_array

函数指针数组，用于构成包含此节的可执行文件或共享目标文件的单个终止数组。有关详细信息，请参见“初始化和终止例程” [100]。

.got

全局偏移表。有关详细信息，请参见“全局偏移表（特定于处理器）” [371]。

.hash

符号散列表。有关详细信息，请参见“散列表节” [308]。

`.init`

可执行指令，用于构成包含此节的可执行文件或共享目标文件的单个初始化函数。有关详细信息，请参见“初始化和终止例程” [100]。

`.init_array`

函数指针数组，用于构成包含此节的可执行文件或共享目标文件的单个初始化数组。有关详细信息，请参见“初始化和终止例程” [100]。

`.interp`

程序的解释程序的路径名。有关详细信息，请参见“程序的解释程序” [355]。

`.lbss`

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

`.ldata`、`.ldata1`

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

`.lrodata`、`.lrodata1`

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

`.note`

“注释节” [312]中说明了该格式的信息。

`.plt`

过程链接表。有关详细信息，请参见“过程链接表（特定于处理器）” [372]。

`.preinit_array`

函数指针数组，用于构成包含此节的可执行文件或共享目标文件的单个预初始化数组。有关详细信息，请参见“初始化和终止例程” [100]。

`.rela`

不适用于特定节的重定位。此节的用途之一是用于寄存器重定位。有关详细信息，请参见“寄存器符号” [335]。

`.rel`*name*、`.rela`*name*

重定位信息，如“重定位节” [314]中所述。如果文件具有包括重定位的可装入段，则此节的属性将包括 `SHF_ALLOC` 位。否则，该位会处于禁用状态。通常，*name* 由应用重定位的节提供。因此，`.text` 的重定位节的名称通常为 `.rel.text` 或 `.rela.text`。

`.rodata`、`.rodata1`

通常构成进程映像中的非可写段的只读数据。有关详细信息，请参见“程序头” [343]。

.shstrtab

节名称。

.strtab

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

.symtab

符号表，如“符号表节” [326]中所述。如果文件具有包括符号表的可装入段，则此节的属性将包括 SHF_ALLOC 位。否则，该位会处于禁用状态。

.symtab_shndx

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

.tbss

此节包含构成程序的内存映像的未初始化线程局部数据。根据定义，为每个新执行流实例化数据时，系统都会将数据初始化为零。如节类型 SHT_NOBITS 所指明的那样，此节不会占用任何文件空间。有关详细信息，请参见第 14 章 线程局部存储。

.tdata、.tdata1

这些节包含已初始化的线程局部数据，这些数据构成程序的内存映像。对于每个新执行流，系统会对其内容的副本进行实例化。有关详细信息，请参见第 14 章 线程局部存储。

.text

程序的文本或可执行指令。

.SUNW_ancillary

辅助组信息。有关详细信息，请参见“辅助节” [303]。

.SUNW_bss

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

.SUNW_cap

功能要求。有关详细信息，请参见“功能节” [306]。

.SUNW_capchain

功能链表。有关详细信息，请参见“功能节” [306]。

.SUNW_capinfo

功能符号信息。有关详细信息，请参见“功能节” [306]。

`.SUNW_heap`

从 dldump(3C) 中创建的动态可执行文件的 堆。

`.SUNW_dynsymsort`

`.SUNW_ldynsym` – `.dynsym` 组合符号表中符号的索引数组。该索引进行排序，以按照地址递增的顺序引用符号。不表示变量或函数的符号不包括在内。对于冗余全局符号和弱符号，仅保留弱符号。有关详细信息，请参见“符号排序节” [334]。

`.SUNW_dyntlssort`

`.SUNW_ldynsym` – `.dynsym` 组合符号表中线程局部存储符号的索引数组。该索引进行排序，以按照偏移递增的顺序引用符号。不表示 TLS 变量的符号不包括在内。对于冗余全局符号和弱符号，仅保留弱符号。有关详细信息，请参见“符号排序节” [334]。

`.SUNW_ldynsym`

扩充 `.dynsym` 节。此节包含局部函数符号，以在完整的 `.symtab` 节不可用时在上下文中使用。链接编辑器始终将 `.SUNW_ldynsym` 节的数据放置在紧邻 `.dynsym` 节之前。这两个节始终使用相同的 `.dynstr` 字符串表节。这种放置和组织方式使两个符号表可以被视为一个更大的符号表。请参见“符号表节” [326]。

`.SUNW_move`

部分初始化数据的附加信息。有关详细信息，请参见“移动部分” [310]。

`.SUNW_reloc`

重定位信息，如“重定位节” [314]中所述。此节是多个重定位节的串联，用于为引用各个重定位记录提供更好的临近性。由于仅有重定位记录的偏移有意义，因此节的 `sh_info` 值为零。

`.SUNW_syminfo`

其他符号表信息。有关详细信息，请参见“*Syminfo* 表节” [336]。

`.SUNW_version`

版本控制信息。有关详细信息，请参见“版本控制节” [337]。

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

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

现有扩展使用其历史名称。

辅助节

除了主输出目标文件之外，Solaris 链接编辑器还可以生成一个或多个辅助目标文件。辅助目标文件包含通常写入到主目标文件中的不可分配节。生成辅助目标文件后，主目标文件和所有关联的辅助目标文件将包含 SHT_SUNW_ancillary 节，该节包含用于标识这些相关目标文件的信息。这些目标文件中任何一个的辅助节都提供了标识和解释组的其他成员所需的信息。

此节包含以下结构的数组。请参见 sys/elf.h。

```
typedef struct {
        Elf32_Word      a_tag;
        union {
                Elf32_Word      a_val;
                Elf32_Addr      a_ptr;
        } a_un;
} Elf32_Ancillary;

typedef struct {
        Elf64_Xword     a_tag;
        union {
                Elf64_Xword     a_val;
                Elf64_Addr      a_ptr;
        } a_un;
} Elf64_Ancillary;
```

对于此类型的每个目标文件，a_tag 都控制着 a_un 的解释。

a_val
: 这些目标文件表示具有各种解释的整数值。

a_ptr
: 这些目标文件表示程序虚拟地址。

存在以下辅助标记。

表 12-13 ELF 辅助数组标记

名称	值	c_un
ANC_SUNW_NULL	0	已忽略
ANC_SUNW_CHECKSUM	1	a_val
ANC_SUNW_MEMBER	2	a_ptr

ANC_SUNW_NULL
: 标记辅助节组的结尾。

ANC_SUNW_CHECKSUM

在 c_val 元素中提供文件的校验和。当 ANC_SUNW_CHECKSUM 位于第一个 ANC_SUNW_MEMBER 实例之前时，它将提供将从中读取辅助节的目标文件的校验和。如果它位于 ANC_SUNW_MEMBER 标记之后，则它将提供该成员的校验和。

ANC_SUNW_MEMBER

指定目标文件名称。a_ptr 元素包含以空字符结尾的字符串的字符串表偏移，该偏移提供文件名称。

辅助节中必须始终在第一个 ANC_SUNW_MEMBER 实例（标识当前目标文件）之前包含一个 ANC_SUNW_CHECKSUM。在其之后，针对构成完整目标文件集合的每个目标文件，都应当有一个 ANC_SUNW_MEMBER。每个 ANC_SUNW_MEMBER 之后都应当跟有该目标文件的 ANC_SUNW_CHECKSUM。因此，典型辅助节的结构如下所示。

标签	含义
ANC_SUNW_CHECKSUM	此目标文件的校验和
ANC_SUNW_MEMBER	目标文件 1 的名称
ANC_SUNW_CHECKSUM	目标文件 1 的校验和
…	
ANC_SUNW_MEMBER	目标文件 N 的名称
ANC_SUNW_CHECKSUM	目标文件 N 的校验和
ANC_SUNW_NULL	

因此，目标文件可以通过将初始 ANC_SUNW_CHECKSUM 与其之后的每项进行比较，直到找到匹配项，来标识其自己。

COMDAT 节

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

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

组节

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

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

SHT_GROUP 类型的节可定义这样分组的一组节：包含方目标文件的其中一个符号表中的某个符号名称为节组提供签名。SHT_GROUP 节的节头会指定标识符号项。sh_link 成员包含含有该项的符号表节的节头索引。sh_info 成员包含标识项的符号表索引。节头的 sh_flags 成员值为零。节名 (sh_name) 未指定。

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

当前定义了以下标志：

表 12-14　ELF 组节标志

名称	值
GRP_COMDAT	0x1

GRP_COMDAT

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

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

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

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

功能节

SHT_SUNW_cap 节标识目标文件的功能要求。这些功能称为目标文件功能。此节还可以标识目标文件中的函数或初始化的数据项的功能要求。这些功能称为符号功能。此节包含以下结构的数组。请参见 sys/elf.h。

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

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

对于此类型的每一个目标文件，c_tag 都会控制 c_un 的解释。

c_val

这些目标文件表示具有各种解释的整数值。

c_ptr

这些目标文件表示程序虚拟地址。

存在以下功能标记。

表 12-15 ELF 功能数组标记

名称	值	c_un
CA_SUNW_NULL	0	已忽略
CA_SUNW_HW_1	1	c_val
CA_SUNW_SF_1	2	c_val
CA_SUNW_HW_2	3	c_val
CA_SUNW_PLAT	4	c_ptr
CA_SUNW_MACH	5	c_ptr
CA_SUNW_ID	6	c_ptr

CA_SUNW_NULL

标记功能组的结尾。

CA_SUNW_HW_1、CA_SUNW_HW_2

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

CA_SUNW_SF_1

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

CA_SUNW_PLAT

指定平台名称。c_ptr 元素包含以空字符结尾的字符串的字符串表偏移，该偏移定义平台名称。

CA_SUNW_MACH

指定计算机名。c_ptr 元素包含以空字符结尾的字符串的字符串表偏移，该偏移定义计算机硬件名称。

CA_SUNW_ID

指定功能标识符名称。c_ptr 元素包含以空字符结尾的字符串的字符串表偏移，该偏移定义标识符名称。此元素并不定义功能，而是为功能组指定唯一的符号名称，通过该名称可以引用该功能组。此标识符名称附加在任何在链接编辑器的 -z symbolcap 处理过程中转换为局部符号的全局符号名称中。请参见“将目标文件功能转换为符号功能” [65]。

可重定位目标文件可以包含功能节。链接编辑器会将多个输入可重定位目标文件中的所有功能节合并到一个单独的功能节中。使用链接编辑器，还可在生成目标文件时定义功能。请参见“标识功能要求” [51]。

一个目标文件中可以存在多个以 CA_SUNW_NULL 结尾的功能组。从索引 0 开始的第一个组标识目标文件功能。定义目标文件功能的动态目标文件包含与该节关联的 PT_SUNWCAP 程序头。使用此程序头，运行时链接程序可以对照可供进程使用的系统功能来验证目标文件。使用不同目标文件功能的动态目标文件可以提供使用过滤器的灵活运行时环境。请参见“特定于功能的共享目标文件” [227]。

其他功能组标识符号功能。符号功能允许一个目标文件中存在同一符号的多个实例。每个实例都关联一组必须可供要使用的实例使用的功能。存在符号功能时，SHT_SUNW_cap 节的 sh_link 元素指向关联的 SHT_SUNW_capinfo 表。使用符号功能的动态目标文件可以提供针对特定的系统启用优化功能的灵活方式。请参见“创建符号功能函数系列” [59]。

SHT_SUNW_capinfo 表与关联的符号表对应。SHT_SUNW_capinfo 节的 sh_link 元素指向关联的符号表。与功能关联的函数在 SHT_SUNW_capinfo 表中具有索引，这些索引用于标识 SHT_SUNW_cap 节中的功能组。

在动态目标文件中，SHT_SUNW_capinfo 节的 sh_info 元素指向功能链表 SHT_SUNW_capchain。运行时链接程序使用此表查找功能系列的成员。

SHT_SUNW_capinfo 表项具有以下格式。请参见 sys/elf.h。

```
typedef Elf32_Word    Elf32_Capinfo;
typedef Elf64_Xword   Elf64_Capinfo;
```

此表中的元素可以使用以下宏进行解释。请参见 sys/elf.h。

```
#define ELF32_C_SYM(info)       ((info)>>8)
#define ELF32_C_GROUP(info)     ((unsigned char)(info))
#define ELF32_C_INFO(sym, grp)  (((sym)<<8)+(unsigned char)(grp))

#define ELF64_C_SYM(info)       ((info)>>32)
#define ELF64_C_GROUP(info)     ((Elf64_Word)(info))
#define ELF64_C_INFO(sym, grp)  (((Elf64_Xword)(sym)<<32)+(Elf64_Xword)(grp))
```

SHT_SUNW_capinfo 项的 group（组）元素包含此符号关联的 SHT_SUNW_cap 表的索引。因此该元素可以将符号与功能组关联起来。保留的组索引 CAPINFO_SUNW_GLOB 标识功能实例系列的前置符号，该符号提供缺省实例。

名称	值	含义
CAPINFO_SUNW_GLOB	0xff	标识缺省符号。此符号不与任何特定的功能关联，但指向一个符号功能系列。

SHT_SUNW_capinfo 项的 symbol 元素包含与此符号关联的前置符号的索引。链接编辑器使用组和符号信息处理可重定位目标文件中的功能符号系列，并在任何输出目标文件中构造所需的功能信息。在动态目标文件中，前置符号的符号元素（使用组 CAPINFO_SUNW_GLOB 标记）是 SHT_SUNW_capchain 表的索引。此索引使运行时链接程序可以遍历功能链表，从该索引开始检查其后的每一项，直至找到 0 项为止。这些链项包含每个功能系列成员的符号索引。

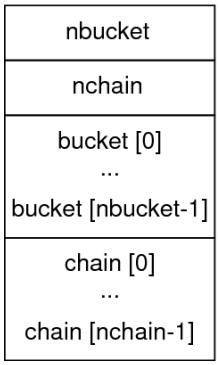
定义符号功能的动态目标文件具有一个 DT_SUNW_CAP 动态项和一个 DT_SUNW_CAPINFO 动态项。这些项分别标识 SHT_SUNW_cap 节和 SHT_SUNW_capinfo 节。目标文件还包含 DT_SUNW_CAPCHAIN、 DT_SUNW_CAPCHAINENT 和 DT_SUNW_CAPCHAINSZ 项，这些项标识 SHT_SUNW_capchain 节、节项大小和总大小。通过这些项，运行时链接程序可以使用符号功能实例系列建立最好的符号以供使用。

目标文件可以仅定义目标文件功能，或仅定义符号功能，也可以同时定义这两种功能。目标文件功能组从索引 0 开始。符号功能组从 0 以外的其他任何索引开始。如果目标文件定义符号功能，但不定义目标文件功能，那么在索引 0 处必须存在一个 CA_SUNW_NULL 项以表示符号功能的开始。

散列表节

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

图 12-4 符号散列表



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

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

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

散列函数如下所示：

```
unsigned long
elf_Hash(const unsigned char *name)
{
        unsigned int h = 0, g;

        while (*name)
        {
                h = (h << 4) + *name++;
                if (g = h & 0xf0000000)
                        h ^= g >> 24;
                h &= ~g;
        }
        return h;
}
```

移动部分

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

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

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

ELF32_Move 和 Elf64_Move 项的定义如下：

```
typedef struct {
        Elf32_Lword       m_value;
        Elf32_Word        m_info;
        Elf32_Word        m_poffset;
        Elf32_Half        m_repeat;
        Elf32_Half        m_stride;
} Elf32_Move;

#define ELF32_M_SYM(info)       ((info)>>8)
#define ELF32_M_SIZE(info)      ((unsigned char)(info))
#define ELF32_M_INFO(sym, size) (((sym)<<8)+(unsigned char)(size))

typedef struct {
        Elf64_Lword       m_value;
        Elf64_Xword       m_info;
        Elf64_Xword       m_poffset;
        Elf64_Half        m_repeat;
        Elf64_Half        m_stride;
} Elf64_Move;

#define ELF64_M_SYM(info)       ((info)>>8)
#define ELF64_M_SIZE(info)      ((unsigned char)(info))
#define ELF64_M_INFO(sym, size) (((sym)<<8)+(unsigned char)(size))
```

这些结构的元素如下：

`m_value`

初始化值，即移到内存映像中的值。

`m_info`

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

m_poffset
: 相对于应用初始化的关联符号的偏移。

m_repeat
: 重复计数。

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

以下数据定义以前在目标文件中会占用 0x8000 个字节：

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

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

SHT_SUNW_move 节可用于说明此数据。数据项在 .bss 节中定义。使用相应的移动项初始化数据项中的非零元素。

```
$ elfdump -s data | fgrep move
    [17]    0x20868   0x8000  OBJT GLOB 0   .bss       move
$ elfdump -m data

Move Section:  .SUNW_move
  symndx     offset   size repeat stride            value  with respect to
    [17]       0x44      4      1      1       0x45000000  move
    [17]       0x40      4      1      1             0xe  move
    [17]       0x34      4      1      1       0x45000000  move
    [17]       0x30      4      1      1             0xe  move
    [17]       0x1c      4      2      1       0x46000000  move
    [17]       0x18      4      2      1             0xf  move
    [17]        0xc      4      1      1       0x31000000  move
    [17]        0x8      4      1      1             0x1  move
```

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

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

注释节

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

节和程序头元素中的注释信息包含任意数量的项，如下图所示。对于 64 位目标文件和 32 位目标文件，每一项都是一个目标处理器格式的 4 字节字的数组。图 12-6 “注释段示例” 中所示的标签用于帮助说明注释信息的结构，但不属于规范的一部分。

图 12-5 注释信息

namesz 和 name

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

descsz 和 desc

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

type

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

下图中所示的注释段包含两项。

图 12-6 注释段示例

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

重定位节

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

重定位项可具有以下结构。请参见 sys/elf.h。

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

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

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

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

`r_offset`

此成员指定应用重定位操作的位置。不同的目标文件对于此成员的解释会稍有不同。

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

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

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

`r_info`

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

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

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

#define ELF64_R_SYM(info)             ((info)>>32)
#define ELF64_R_TYPE(info)            ((Elf64_Word)(info))
#define ELF64_R_INFO(sym, type)       (((Elf64_Xword)(sym)<<32)+ \
                                          (Elf64_Xword)(type))
```

对于 64 位 SPARC `Elf64_Rela` 结构，`r_info` 字段可进一步细分为 8 位类型标识符和 24 位类型相关数据字段。对于现有的重定位类型，数据字段为零。但是，新的重定位类型可能会使用数据位。

```
#define ELF64_R_TYPE_DATA(info)       (((Elf64_Xword)(info)<<32)>>40)
#define ELF64_R_TYPE_ID(info)         (((Elf64_Xword)(info)<<56)>>56)
#define ELF64_R_TYPE_INFO(data, type) (((Elf64_Xword)(data)<<8)+ \
                                          (Elf64_Xword)(type))
```

`r_addend`

此成员指定常量加数，用于计算将存储在可重定位字段中的值。

`Rela` 项包含显式加数。类型为 `Rel` 的项存储要修改的位置中的隐式加数。在所有情况下，加数和计算所得的结果使用相同的字节顺序。加数值的重定位项类型和解释由特定于平台的 ABI 定义。

SPARC — 32 位 SPARC 使用 `Elf32_Rela` 重定位项。64 位 SPARC 使用 `Elf64_Rela` 重定位项。要重定位的字段的先前值被添加到 `r_addend` 成员中，用作重定位加数。

32 位 x86 — 32 位 x86 使用 `Elf32_Rel` 重定位项。要重定位的字段包含该加数。

64 位 x86 — 64 位 x86 使用 `Elf64_Rela` 重定位项。`r_addend` 成员用作重定位加数。忽略要重定位的字段的先前值。

重定位节可以引用其他两个节：符号表（由 `sh_link` 节头项标识）和要修改的节（由 `sh_info` 节头项标识）。“节” [281]中指定了这些关系。如果可重定位目标文件中存在重定位节，则需要 `sh_info` 项，但对于可执行文件和共享目标文件，该项是可选的。重定位偏移满足执行重定位的要求。

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

重定位计算

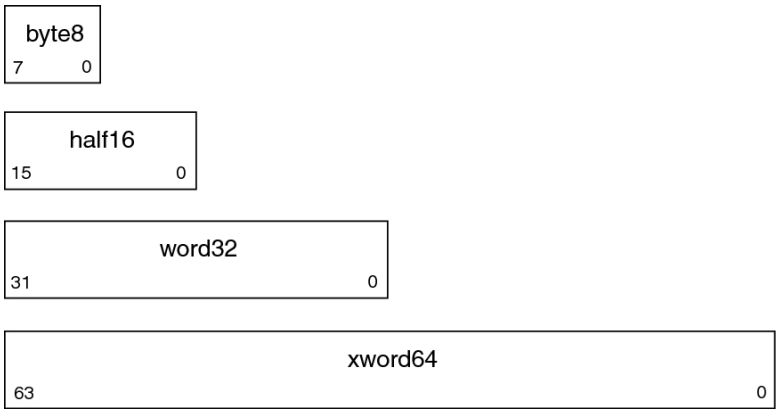
以下表示法用于说明重定位计算。

`A` — 用于计算可重定位字段的值的加数。

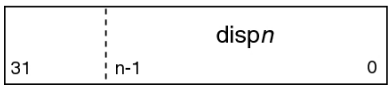
B	执行过程中将共享目标文件装入内存的基本地址。通常，生成的共享目标文件的基本虚拟地址为 0。但是，共享目标文件的执行地址不相同。请参见“程序头” [343]。
G	执行过程中，重定位项的符号地址所在的全局偏移表中的偏移。请参见“全局偏移表（特定于处理器）” [371]。
GOT	全局偏移表的地址。请参见“全局偏移表（特定于处理器）” [371]。
L	符号的过程链接表项的节偏移或地址。请参见“过程链接表（特定于处理器）” [372]。
P	使用 r_offset 计算出的重定位的存储单元的节偏移或地址。
S	索引位于重定位项中的符号的值。
Z	索引位于重定位项中的符号的大小。

SPARC: 重定位

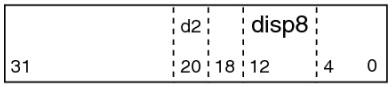
在 SPARC 平台上，重定位项应用于字节 (byte8)、半字 (half16)、字 (word32) 和扩展字 (xword64)。



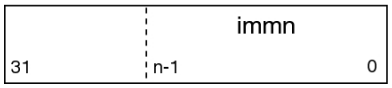
重定位字段 (disp19, disp22, disp30) 的 disp*n* 系列都是字对齐、带符号扩展的 PC 相对位移。全部将值编码为其最低有效位都位于字的位置 0，仅在分配给值的位数方面有所不同。



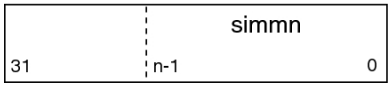
d2/disp8 和 d2/disp14 变体使用两个非连续位字段 d2 和 disp*n* 对 16 位和 10 位位移值进行编码。



重定位字段的 imm*n* 系列（imm5、imm6、imm7、imm10、imm13 和 imm22）表示无符号整型常数。全部将值编码为其最低有效位都位于字的位置 0，仅在分配给值的位数方面有所不同。



重定位字段的 simm*n* 系列（simm10、simm11、simm13 和 simm22）表示带符号的整型常数。全部将值编码为其最低有效位都位于字的位置 0，仅在分配给值的位数方面有所不同。



SPARC: 重定位类型

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

表 12-16 SPARC: ELF 重定位类型

名称	值	字段	计算
R_SPARC_NONE	0	无	无
R_SPARC_8	1	V-byte8	S + A
R_SPARC_16	2	V-half16	S + A
R_SPARC_32	3	V-word32	S + A
R_SPARC_DISP8	4	V-byte8	S + A - P
R_SPARC_DISP16	5	V-half16	S + A - P
R_SPARC_DISP32	6	V-disp32	S + A - P
R_SPARC_WDISP30	7	V-disp30	(S + A - P) >> 2
R_SPARC_WDISP22	8	V-disp22	(S + A - P) >> 2
R_SPARC_HI22	9	T-imm22	(S + A) >> 10
R_SPARC_22	10	V-imm22	S + A
R_SPARC_13	11	V-simm13	S + A
R_SPARC_LO10	12	T-simm13	(S + A) & 0x3ff
R_SPARC_GOT10	13	T-simm13	G & 0x3ff
R_SPARC_GOT13	14	V-simm13	G
R_SPARC_GOT22	15	T-simm22	G >> 10
R_SPARC_PC10	16	T-simm13	(S + A - P) & 0x3ff
R_SPARC_PC22	17	V-disp22	(S + A - P) >> 10
R_SPARC_WPLT30	18	V-disp30	(L + A - P) >> 2
R_SPARC_COPY	19	无	请参阅此表后面的说明。
R_SPARC_GLOB_DAT	20	V-word32	S + A
R_SPARC_JMP_SLOT	21	无	请参阅此表后面的说明。
R_SPARC_RELATIVE	22	V-word32	B + A
R_SPARC_UA32	23	V-word32	S + A
R_SPARC_PLT32	24	V-word32	L + A
R_SPARC_HIPLT22	25	T-imm22	(L + A) >> 10
R_SPARC_LOPLT10	26	T-simm13	(L + A) & 0x3ff
R_SPARC_PCPLT32	27	V-word32	L + A - P
R_SPARC_PCPLT22	28	V-disp22	(L + A - P) >> 10
R_SPARC_PCPLT10	29	V-simm13	(L + A - P) & 0x3ff
R_SPARC_10	30	V-simm10	S + A
R_SPARC_11	31	V-simm11	S + A
R_SPARC_HH22	34	V-imm22	(S + A) >> 42
R_SPARC_HM10	35	T-simm13	((S + A) >> 32) & 0x3ff
R_SPARC_LM22	36	T-imm22	(S + A) >> 10
R_SPARC_PC_HH22	37	V-imm22	(S + A - P) >> 42
R_SPARC_PC_HM10	38	T-simm13	((S + A - P) >> 32) & 0x3ff

名称	值	字段	计算
R_SPARC_PC_LM22	39	T-imm22	(S + A - P) >> 10
R_SPARC_WDISP16	40	V-d2/disp14	(S + A - P) >> 2
R_SPARC_WDISP19	41	V-disp19	(S + A - P) >> 2
R_SPARC_7	43	V-imm7	S + A
R_SPARC_5	44	V-imm5	S + A
R_SPARC_6	45	V-imm6	S + A
R_SPARC_HIX22	48	V-imm22	((S + A) ^ 0xffffffffffffffff) >> 10
R_SPARC_LOX10	49	T-simm13	((S + A) & 0x3ff) | 0x1c00
R_SPARC_H44	50	V-imm22	(S + A) >> 22
R_SPARC_M44	51	T-imm10	((S + A) >> 12) & 0x3ff
R_SPARC_L44	52	T-imm13	(S + A) & 0xfff
R_SPARC_REGISTER	53	V-word32	S + A
R_SPARC_UA16	55	V-half16	S + A
R_SPARC_GOTDATA_HIX22	80	V-imm22	((S + A - GOT) >> 10) ^ ((S + A - GOT) >> 31)
R_SPARC_GOTDATA_LOX10	81	T-imm13	((S + A - GOT) & 0x3ff) | (((S + A - GOT) >> 31) & 0x1c00)
R_SPARC_GOTDATA_OP_HIX22	82	T-imm22	(G >> 10) ^ (G >> 31)
R_SPARC_GOTDATA_OP_LOX10	83	T-imm13	(G & 0x3ff) | ((G >> 31) & 0x1c00)
R_SPARC_GOTDATA_OP	84	Word32	请参阅此表后面的说明。
R_SPARC_SIZE32	86	V-word32	Z + A
R_SPARC_WDISP10	88	V-d2/disp8	(S + A - P) >> 2

注 - 其他重定位类型可用于线程局部存储引用。这些重定位类型将在第 14 章 线程局部存储中介绍。

一些重定位类型的语义不只是简单的计算：

R_SPARC_GOT10

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

R_SPARC_GOT13

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

R_SPARC_GOT22

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

R_SPARC_WPLT30

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

R_SPARC_COPY

由链接编辑器为动态可执行文件创建，用于保留只读文本段。此重定位偏移成员指向可写段中的位置。符号表索引指定应在当前目标文件和共享目标文件中同时存在的符号。执行过程中，运行时链接程序将与共享目标文件的符号关联的数据复制到偏移所指定的位置。请参见“复制重定位” [170]。

R_SPARC_GLOB_DAT

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

R_SPARC_JMP_SLOT

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

R_SPARC_RELATIVE

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

R_SPARC_UA32

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

R_SPARC_LM22

与 R_SPARC_HI22 类似，不同的是此重定位会进行截断而不是验证。

R_SPARC_PC_LM22

与 R_SPARC_PC22 类似，不同的是此重定位会进行截断而不是验证。

R_SPARC_HIX22

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

R_SPARC_LOX10

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

R_SPARC_L44

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

R_SPARC_REGISTER

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

R_SPARC_GOTDATA_OP_HIX22、R_SPARC_GOTDATA_OP_LOX10 和 R_SPARC_GOTDATA_OP

这些重定位类型用于代码转换。

64 位 SPARC: 重定位类型

重定位计算中使用的以下表示法是特定于 64 位 SPARC 的。

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

下表中列出的重定位类型是扩展或修改针对 32 位 SPARC 定义的重定位类型所得的。请参见“重定位类型” [317]。

表 12-17　64 位 SPARC: ELF 重定位类型

名称	值	字段	计算
R_SPARC_HI22	9	V-imm22	(S + A) >> 10
R_SPARC_GLOB_DAT	20	V-xword64	S + A
R_SPARC_RELATIVE	22	V-xword64	B + A
R_SPARC_64	32	V-xword64	S + A
R_SPARC_OLO10	33	V-simm13	((S + A) & 0x3ff) + O
R_SPARC_DISP64	46	V-xword64	S + A - P
R_SPARC_PLT64	47	V-xword64	L + A
R_SPARC_REGISTER	53	V-xword64	S + A
R_SPARC_UA64	54	V-xword64	S + A
R_SPARC_H34	85	V-imm22	(S + A) >> 12
R_SPARC_SIZE64	87	V-xword64	Z + A

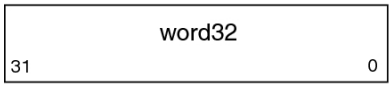
以下重定位类型的语义不只是简单的计算：

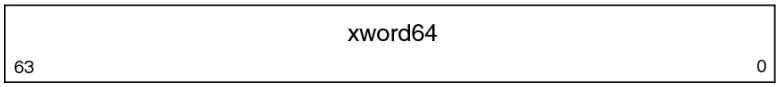
R_SPARC_OLO10

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

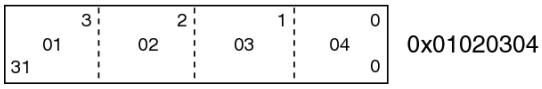
x86: 重定位

在 x86 上，重定位项应用于字 (word32) 和扩展字 (xword64)。





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



0x01020304

32 位 x86: 重定位类型

下表中列出的重定位类型是针对 32 位 x86 定义的。

表 12-18 32 位 x86: ELF 重定位类型

名称	值	字段	计算
R_386_NONE	0	无	无
R_386_32	1	word32	S + A
R_386_PC32	2	word32	S + A - P
R_386_GOT32	3	word32	G + A
R_386_PLT32	4	word32	L + A - P
R_386_COPY	5	无	请参阅此表后面的说明。
R_386_GLOB_DAT	6	word32	S
R_386_JMP_SLOT	7	word32	S
R_386_RELATIVE	8	word32	B + A
R_386_GOTOFF	9	word32	S + A - GOT
R_386_GOTPC	10	word32	GOT + A - P
R_386_32PLT	11	word32	L + A

名称	值	字段	计算
R_386_16	20	word16	S + A
R_386_PC16	21	word16	S + A - P
R_386_8	22	word8	S + A
R_386_PC8	23	word8	S + A - P
R_386_SIZE32	38	word32	Z + A

注 - 其他重定位类型可用于线程局部存储引用。这些重定位类型将在第 14 章 线程局部存储中介绍。

一些重定位类型的语义不只是简单的计算：

R_386_GOT32

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

R_386_PLT32

计算符号的过程链接表项的地址，并指示链接编辑器创建一个过程链接表。

R_386_COPY

由链接编辑器为动态可执行文件创建，用于保留只读文本段。此重定位偏移成员指向可写段中的位置。符号表索引指定应在当前目标文件和共享目标文件中同时存在的符号。执行过程中，运行时链接程序将与共享目标文件的符号关联的数据复制到偏移所指定的位置。请参见“复制重定位” [170]。

R_386_GLOB_DAT

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

R_386_JMP_SLOT

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

R_386_RELATIVE

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

R_386_GOTOFF

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

R_386_GOTPC

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

x64: 重定位类型

下表中列出的重定位是针对 x64 定义的。

表 12-19 x64: ELF 重定位类型

名称	值	字段	计算
R_AMD64_NONE	0	无	无
R_AMD64_64	1	word64	S + A
R_AMD64_PC32	2	word32	S + A - P
R_AMD64_GOT32	3	word32	G + A
R_AMD64_PLT32	4	word32	L + A - P
R_AMD64_COPY	5	无	请参阅此表后面的说明。
R_AMD64_GLOB_DAT	6	word64	S
R_AMD64_JUMP_SLOT	7	word64	S
R_AMD64_RELATIVE	8	word64	B + A
R_AMD64_GOTPCREL	9	word32	G + GOT + A - P
R_AMD64_32	10	word32	S + A
R_AMD64_32S	11	word32	S + A
R_AMD64_16	12	word16	S + A
R_AMD64_PC16	13	word16	S + A - P
R_AMD64_8	14	word8	S + A
R_AMD64_PC8	15	word8	S + A - P
R_AMD64_PC64	24	word64	S + A - P
R_AMD64_GOTOFF64	25	word64	S + A - GOT
R_AMD64_GOTPC32	26	word32	GOT + A + P
R_AMD64_SIZE32	32	word32	Z + A
R_AMD64_SIZE64	33	word64	Z + A

注 - 其他重定位类型可用于线程局部存储引用。这些重定位类型将在第 14 章 线程局部存储中介绍。

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

R_AMD64_GOTPCREL

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

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

R_AMD64_32

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

R_AMD64_32S

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

R_AMD64_8、R_AMD64_16、R_AMD64_PC16 和 R_AMD64_PC8

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

字符串表节

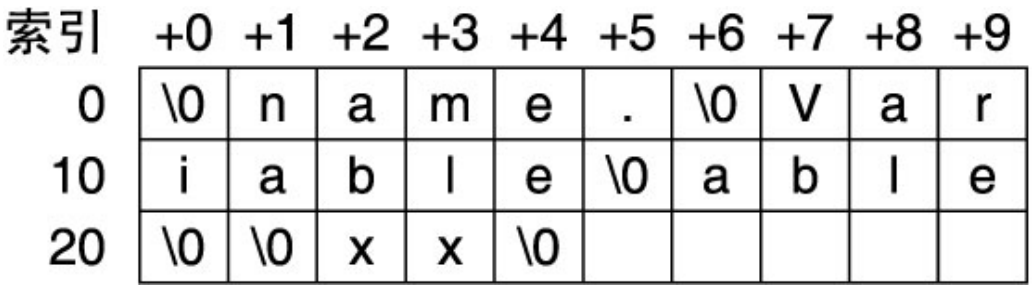
字符串表节包含以空字符结尾的字符序列，通常称为字符串。目标文件使用这些字符串表示符号和节的名称。可以将字符串作为字符串表节的索引进行引用。

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

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

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

图 12-7 ELF 字符串表



下表显示了上图所示的字符串表中的字符串。

表 12-20 ELF 字符串表索引

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

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

符号表节

目标文件的符号表包含定位和重定位程序的符号定义和符号引用所需的信息。符号表索引是此数组的下标。索引 0 指定表中的第一项并用作未定义的符号索引。请参见表 12-24 “ELF 符号表项：索引 0”。

符号表项具有以下格式。请参见 sys/elf.h。

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
```

```
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;

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

`st_name`

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

`st_value`

关联符号的值。根据上下文，该值可以是绝对值或地址。请参见“符号值” [332]。

`st_size`

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

`st_info`

符号的类型和绑定属性。表 12-21 “ELF 符号绑定 `ELF32_ST_BIND` 和 `ELF64_ST_BIND`” 中显示了值和含义的列表。以下代码说明了如何处理这些值。请参见 `sys/elf.h`。

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

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

`st_other`

符号的可见性。表 12-23 “ELF 符号可见性” 中显示了值和含义的列表。以下代码说明了如何处理 32 位目标文件和 64 位目标文件的值。其他位设置为零，并且未定义任何含义。

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

`st_shndx`

所定义的每一个符号表项都与某节有关。此成员包含相关节头表索引。部分节索引会表示特殊含义。请参见表 12-4 “ELF 特殊节索引”。

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

根据符号的 st_info 字段确定的符号绑定可确定链接可见性和行为。

表 12-21 ELF 符号绑定 ELF32_ST_BIND 和 ELF64_ST_BIND

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

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

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

STB_WEAK
: 弱符号。这些符号与全局符号类似，但其定义具有较低的优先级。

STB_LOOS - STB_HIOS
: 此范围内包含的值（包括这两个值）保留用于特定于操作系统的语义。

STB_LOPROC - STB_HIPROC
: 此范围内包含的值（包括这两个值）保留用于特定于处理器的语义。

全局符号和弱符号主要在以下两个方面不同：

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

  同样，如果存在通用符号，则出现相同名称的弱符号也不会导致错误。链接编辑器将使用通用定义，并忽略弱定义。通用符号具有包含 SHN_COMMON 的 st_shndx 字段。请参见“符号解析” [36]。
- 链接编辑器搜索归档库时，将会提取包含未定义全局符号或暂定全局符号的定义的归档成员。此成员的定义可以是全局符号或弱符号。

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

注 - 弱符号主要适用于系统软件。建议不要在应用程序中使用弱符号。

在每个符号表中，具有 STB_LOCAL 绑定的所有符号都优先于弱符号和全局符号。如“节” [281]中所述，符号表节的 sh_info 节头成员包含第一个非局部符号的符号表索引。

根据符号的 st_info 字段确定的符号类型用于对关联实体进行一般等级划分。

表 12-22 ELF 符号类型 ELF32_ST_TYPE 和 ELF64_ST_TYPE

名称	值
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_COMMON	5
STT_TLS	6
STT_LOOS	10
STT_HIOS	12
STT_LOPROC	13
STT_SPARC_REGISTER	13
STT_HIPROC	15

STT_NOTYPE
: 未指定符号类型。

STT_OBJECT
: 此符号与变量、数组等数据目标文件关联。

STT_FUNC
: 此符号与函数或其他可执行代码关联。

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

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

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

STT_COMMON

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

STT_TLS

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

线程局部存储重定位只能引用 STT_TLS 类型的符号。从可分配节中引用 STT_TLS 类型的符号只能通过使用特殊线程局部存储重定位来实现。有关详细信息，请参见第 14 章 线程局部存储。从不可分配节中引用 STT_TLS 类型的符号没有此限制。

STT_LOOS - STT_HIOS

此范围内包含的值（包括这两个值）保留用于特定于操作系统的语义。

STT_LOPROC - STT_HIPROC

此范围内包含的值（包括这两个值）保留用于特定于处理器的语义。

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

表 12-23 ELF 符号可见性

名称	值
STV_DEFAULT	0
STV_INTERNAL	1
STV_HIDDEN	2
STV_PROTECTED	3
STV_EXPORTED	4
STV_SINGLETON	5
STV_ELIMINATE	6

STV_DEFAULT

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

STV_PROTECTED

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

STV_HIDDEN

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

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

STV_INTERNAL

此可见性属性以与 STV_HIDDEN 相同的方式进行解释。

STV_EXPORTED

此可见性属性确保符号保持为全局。不能使用任何其他符号可见性技术对此可见性进行降级或消除。具有 STB_LOCAL 绑定的符号将没有 STV_EXPORTED 可见性。

STV_SINGLETON

此可见性属性确保符号保持为全局，并且符号定义的一个实例绑定到一个进程中的所有引用。不能使用任何其他符号可见性技术对此可见性进行降级或消除。具有 STB_LOCAL 绑定的符号将没有 STV_SINGLETON 可见性。不能直接绑定到 STV_SINGLETON。

STV_ELIMINATE

此可见性属性扩展 STV_HIDDEN。当前组件中定义为要消除的符号对其他组件不可见。该符号未写入使用该组件的动态可执行文件或共享目标文件的任何符号表中。

STV_SINGLETON 可见性属性可以影响链接编辑过程中可执行文件或共享目标文件中的符号的解析。从一个进程中的任意引用只能绑定到单件的一个实例。

STV_SINGLETON 可以与 STV_DEFAULT 可见性属性组合，其中 STV_SINGLETON 优先级较高。STV_EXPORT 可以与 STV_DEFAULT 可见性属性组合使用，其中 STV_EXPORT 优先级较高。STV_SINGLETON 或 STV_EXPORT 可见性不能与任何其他可见性属性组合。链接编辑会将此类事件视为致命错误。

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

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

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

SHN_ABS
: 此符号具有不会由于重定位而发生更改的绝对值。

SHN_COMMON 和 SHN_AMD64_LCOMMON
: 此符号标记尚未分配的通用块。与节的 sh_addralign 成员类似，符号的值也会指定对齐约束。链接编辑器在值为 st_value 的倍数的地址位置为符号分配存储空间。符号的大小会指明所需的字节数。

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

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

表 12-24 ELF 符号表项：索引 0

名称	值	备注
st_name	0	无名称
st_value	0	零值
st_size	0	无大小
st_info	0	无类型，局部绑定
st_other	0	
st_shndx	SHN_UNDEF	无节

符号值

不同目标文件类型的符号表的各项对于 st_value 成员的解释稍有不同。

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

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

符号表布局和约定

按照以下顺序将符号写入符号表。

- 任何符号表中的索引 0 用于表示未定义的符号。符号表中的第一项始终完全为零。因此符号类型为 STT_NOTYPE。
- 如果符号表包含任何局部符号，符号表中的第二项是提供文件名的 STT_FILE 符号。
- STT_SECTION 类型的节符号。
- STT_REGISTER 类型的寄存器符号。
- 缩减到局部作用域的全局符号。
- 对于提供局部符号的每个输入文件，提供输入文件名称的 STT_FILE 符号，后跟相关符号。
- 在符号表中，全局符号紧跟局部符号。第一个全局符号由符号表 sh_info 值标识。局部符号和全局符号始终以这种方式保持彼此独立，不能混合。

这些符号表是 Oracle Solaris OS 中的特殊内容。

.symtab (SHT_SYMTAB)

此符号表包含说明关联的 ELF 文件的每个符号。此符号表通常是不可分配的，因此在进程的内存映像中不可用。

通过使用 mapfile 和 ELIMINATE 关键字可以从 .symtab 中消除全局符号。请参见“删除符号” [49]和“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。

.dynsym (SHT_DYNSYM)

此表包含 .symtab 表中支持动态链接所需的符号的子集。此符号表可供分配，因此在进程的内存映像中可用。

.dynsym 表以标准 NULL 符号开始，后跟文件全局符号。STT_FILE 符号通常不包含在此符号表中。如果重定位项需要，可能会包含 STT_SECTION 符号。

.SUNW_ldynsym (SHT_SUNW_LDYNSYM)

扩充 .dynsym 表中包含的信息的可选符号表。.SUNW_ldynsym 表包含局部函数符号。此符号表可供分配，因此在进程的内存映像中可用。当不可分配的 .symtab 不可用或已从文件中剥离时，调试器通过使用此节可在运行时上下文中产生精确的栈跟踪。此节还可以为运行时环境提供其他符号信息，以便与 dladdr(3C) 一起使用。

仅当 .dynsym 表存在时，才存在 .SUNW_ldynsym 表。当 .SUNW_ldynsym 节和 .dynsym 节同时存在时，链接编辑器会将其数据区域紧邻彼此放置，其中 .SUNW_ldynsym 放置在前面。这种放置方式可以使两个表看起来像是一个更大的连续符号表。此符号表遵从先前枚举的标准布局规则。

.SUNW_ldynsym 表可以通过使用链接编辑器的 -z noldynsym 选项进行消除。

符号排序节

由相邻的 .SUNW_ldynsym 节和 .dynsym 节构成的动态符号表可用于将内存地址映射到其对应的符号。这种映射可用于确定给定地址表示哪个函数或变量。但是，按照符号写入符号表的顺序分析符号表以确定映射是很复杂的。请参见“符号表布局和约定” [333]。此布局使地址与符号名称之间的关联变得复杂，具体表现在以下几个方面。

- 符号不是按地址排序的，这将强制对整个表进行线性搜索，从而增大开销。
- 可能有多个符号指向给定地址。尽管这些符号均有效且正确，但调试工具应选择使用这些等效名称中的哪一个可能并不明确。不同的工具可能会使用不同的备用名称。这些问题可能会给用户带来困惑。
- 许多符号提供无地址信息。搜索时不应考虑这类符号。

符号排序节就是用于解决这些问题的。符号排序节是 Elf32_Word 或 Elf64_Word 目标文件的数组。此数组中的每个元素都是 .SUNW_ldynsym – .dynsym 组合符号表的索引。对数组中的元素进行排序，从而使引用的符号也按照排好的顺序提供。其中仅包含表示函数或变量的符号。使用 elfdump(1) 和 -S 选项来显示与排序数组关联的符号。

不能同时排序常规符号和线程局部存储符号。常规符号的值是符号所引用的函数或变量的地址。线程局部存储符号的值是变量的线程偏移。因此，常规符号和线程局部存储符号使用两种不同的排序节。

.SUNW_dynsymsort

SHT_SUNW_SYMSORT 类型的节，其中包含 .SUNW_ldynsym – .dynsym 组合符号表中常规符号的索引，该索引按地址排序。不表示变量或函数的符号不包括在内。

.SUNW_dyntlssort

SHT_SUNW_TLSSORT 类型的节，其中包含 .SUNW_ldynsym – .dynsym 组合符号表中 TLS 符号的索引，该索引按偏移排序。仅当目标文件包含 TLS 符号时会生成此节。

链接编辑器按照显示的顺序使用以下规则来选择排序节引用的符号。

- 符号必须具有函数或变量类型：STT_FUNC、STT_OBJECT、STT_COMMON 或 STT_TLS。
- 以下符号如果存在，则始终包含在内：_DYNAMIC、_end、_fini、_GLOBAL_OFFSET_TABLE_、_init、_PROCEDURE_LINKAGE_TABLE_ 和 _start。
- 如果全局符号和弱符号引用同一个项，则会包含弱符号而排除全局符号。
- 符号必须已定义。
- 符号的大小不得为零。

这些规则可筛选出由编译器自动生成以及由链接编辑器生成的符号。所选择的符号是用户关注的符号。然而，在两种情况下可能需要手动干预来改进选择过程。

- 规则未能选择所需的特殊符号。例如，某些特殊符号大小为零。

- 选择了不需要的多余符号。例如，共享目标文件可以定义引用相同地址并且大小相同的多个符号。这些别名符号均有效地引用相同的项。您可能希望在排序节中仅包含多符号系列中的一个。

mapfile 的关键字 DYNSORT 和 NODYNSORT 可以为符号选择提供更多的控制。请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。

DYNSORT

标识应包含在排序节中的符号。符号类型必须为 STT_FUNC、STT_OBJECT、STT_COMMON 或 STT_TLS。

NODYNSORT

标识不应包含在排序节中的符号。

例如，目标文件可能会提供以下符号表定义。

```
$ elfdump -sN.symtab foo.so.1 | egrep "foo$|bar$"
    [37]    0x4b0   0x1c  FUNC GLOB  D   0 .text      bar
    [38]    0x4b0   0x1c  FUNC WEAK  D   0 .text      foo
```

符号 foo 和 bar 表示一个别名对。缺省情况下，创建排序数组时，仅表示符号 foo。

```
$ cc -o foo.so.1 -G foo.c
$ elfdump -S foo.so.1 | egrep "foo$|bar$"
    [13]    0x4b0   0x1c  FUNC WEAK  D   0 .text      foo
```

如果链接编辑器发现一个全局符号和一个弱符号引用同一个项，通常会保留弱符号。排序数组中忽略符号 bar 是因为与弱符号 foo 之间的关联。

以下 mapfile 会导致在排序数组中表示符号 bar。忽略了符号 foo。

```
$ cat mapfile
{
        global:
                bar = DYNSORT;
                foo = NODYNSORT;
};
$ cc -M mapfile -o foo.so.2 -Kpic -G foo.c
$ elfdump -S foo.so.2 | egrep "foo$|bar$"
    [13]    0x4b0   0x1c  FUNC GLOB  D   0 .text      bar
```

.SUNW_dynsymsort 节和 .SUNW_dyntlssort 节，要求 .SUNW_ldynsym 节存在。因此，使用 -z noldynsym 选项还会阻止创建任何排序节。

寄存器符号

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

表 12-25 SPARC: ELF 符号表项：寄存器符号

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

下表中列出了为 SPARC 定义的寄存器值。

表 12-26 SPARC: ELF 寄存器编号

名称	值	含义
STO_SPARC_REGISTER_G2	0x2	%g2
STO_SPARC_REGISTER_G3	0x3	%g3

如果缺少特定全局寄存器的项，则意味着目标文件根本没有使用特定全局寄存器。

Syminfo 表节

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

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

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

Syminfo 项具有以下格式。请参见 sys/link.h。

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

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

```
} Elf64_Syminfo;
```

si_boundto

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

名称	值	含义
SYMINFO_BT_SELF	0xffff	符号与自身绑定。
SYMINFO_BT_PARENT	0xfffe	符号与父级绑定。父级是指导致此动态目标文件被装入的第一个目标文件。
SYMINFO_BT_NONE	0xfffd	符号没有任何特殊的符号绑定。
SYMINFO_BT_EXTERN	Oxfffc	符号定义是外部的。

si_flags

此位字段可以设置标志，如下表所示。

名称	值	含义
SYMINFO_FLG_DIRECT	0x01	符号引用与包含定义的目标文件直接关联。
SYMINFO_FLG_FILTER	0x02	符号定义可用作标准过滤器。
SYMINFO_FLG_COPY	0x04	符号定义通过副本重定位生成。
SYMINFO_FLG_LAZYLOAD	0x08	符号引用应延迟装入的目标文件。
SYMINFO_FLG_DIRECTBIND	0x10	符号引用应与定义直接绑定。
SYMINFO_FLG_NOEXTDIRECT	0x20	不允许将外部引用与此符号定义直接绑定。
SYMINFO_FLG_AUXILIARY	0x40	符号定义可用作辅助过滤器。
SYMINFO_FLG_INTERPOSE	0x80	符号定义可用作插入项。此属性仅适用于动态可执行文件。
SYMINFO_FLG_CAP	0x100	符号与功能相关联。
SYMINFO_FLG_DEFERRED	0x200	符号不应包含在 BIND_NOW 重定位中。

版本控制节

链接编辑器创建的目标文件可以包含以下两种类型的版本控制信息：

- 版本定义，用于提供全局符号关联，并使用类型为 SHT_SUNW_verdef 和 SHT_SUNW_versym 的节进行实现。
- 版本依赖性，用于表示其他目标文件依赖性的版本定义要求，并使用类型为 SHT_SUNW_verneedSHT_SUNW_versym 的节进行实现。

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

版本定义章节

此节由 SHT_SUNW_verdef 类型定义。如果此节存在，则必须同时存在 SHT_SUNW_versym 节。这两种结构在文件中提供符号与版本定义之间的关联。请参见“创建版本定义” [211]。此节中的元素具有以下结构：

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

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

typedef struct {
        Elf64_Half      vd_version;
        Elf64_Half      vd_flags;
        Elf64_Half      vd_ndx;
        Elf64_Half      vd_cnt;
        Elf64_Word      vd_hash;
        Elf64_Word      vd_aux;
        Elf64_Word      vd_next;
} Elf64_Verdef;

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

vd_version

此成员标识该结构的版本，如下表中所列。

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

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

vd_flags

此成员包含特定于版本定义的信息，如下表中所列。

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

对文件应用版本定义或符号自动缩减后，基版本定义将始终存在。基版本可为文件保留的符号提供缺省版本。弱版本定义 (weak version definition) 没有与版本关联的符号。请参见“创建弱版本定义” [213]。

vd_ndx

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

vd_cnt

Elf32_Verdaux 数组中的元素数目。

vd_hash

版本定义名称的散列值。该值是通过使用“散列表节” [308]中介绍的同一散列函数生成的。

vd_aux

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

vd_next

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

vda_name

以空字符结尾的字符串的字符串表偏移，用于提供版本定义的名称。

vda_next

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

版本依赖性节

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

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

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

typedef struct {
        Elf64_Half      vn_version;
        Elf64_Half      vn_cnt;
        Elf64_Word      vn_file;
        Elf64_Word      vn_aux;
        Elf64_Word      vn_next;
} Elf64_Verneed;

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

`vn_version`

此成员标识该结构的版本，如下表中所列。

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

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

`vn_cnt`

Elf32_Vernaux 数组中的元素数目。

`vn_file`

以空字符结尾的字符串的字符串表偏移，用于提供版本依赖性的文件名。此名称与文件中找到的 .dynamic 依赖项之一匹配。请参见“动态节” [356]。

vn_aux

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

vn_next

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

vna_hash

版本依赖性名称的散列值。该值是通过使用“散列表节” [308]中介绍的同一散列函数生成的。

vna_flags

版本依赖性特定信息，如下表中所列。

名称	值	含义
VER_FLG_WEAK	0x2	弱版本标识符。
VER_FLG_INFO	0x4	SHT_SUNW_versym 引用用于提供信息，在运行时无需进行验证。

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

vna_other

如果为非零，则会向此依赖项版本指定版本索引。此索引用于在 SHT_SUNW_versym 内向此版本指定全局符号引用。

在 Oracle Solaris 10 发行版及更低 Solaris 版本中，不向依赖项版本指定版本符号索引。在这些目标文件中，vna_other 的值为 0。

vna_name

以空字符结尾的字符串的字符串表偏移，用于提供版本依赖性的名称。

vna_next

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

版本符号节

版本符号节由 SHT_SUNW_versym 类型定义。此节由具有以下结构的元素数组构成。

```
typedef Elf32_Half      Elf32_Versym;
typedef Elf64_Half      Elf64_Versym;
```

数组元素的数量必须等于关联的符号表中包含的符号表项的数量。此数量由节的 sh_link 值确定。数组的每个元素都包含一个索引，该索引可以包含下表中显示的值。

表 12-27 ELF 版本依赖性索引

名称	值	含义
VER_NDX_LOCAL	0	符号具有局部作用域。
VER_NDX_GLOBAL	1	符号具有全局作用域，并指定给基版本定义。
	>1	符号具有全局作用域，并指定给用户定义的版本定义 SHT_SUNW_verdef 或版本依赖性 SHT_SUNW_verneed。

可以为符号指定特殊的保留索引 0。可以出于以下任意原因指定此索引。

- 始终为非全局符号指定 VER_NDX_LOCAL。然而，这在实际操作中很少见。版本控制节通常仅与只包含全局符号的动态符号表 .dynsym 一起创建。
- 没有 SHT_SUNW_verdef 版本定义节的目标文件中定义的全局符号。
- 没有 SHT_SUNW_verneed 版本依赖性节的目标文件中定义的未定义全局符号。或者，其中的版本依赖性节未指定版本索引的目标文件中定义的未定义全局符号。
- 符号表的第一项始终为 NULL。此项始终接收 VER_NDX_LOCAL，但该值没有任何特定意义。

目标文件所定义的版本指定从 1 开始的版本索引，并且随版本变化每次增加 1。索引 1 保留用于第一个全局版本。如果该目标文件没有 SHT_SUNW_verdef 版本定义节，则该目标文件定义的所有全局符号均接收索引 1。如果目标文件具有版本定义节，则 VER_NDX_GLOBAL 将仅指向第一个此类版本。

其他 SHT_SUNW_verneed 依赖项中目标文件所需的版本将指定从 1 到最终版本定义索引之间的版本索引。这些索引也随版本变化每次增加 1。由于索引 1 始终保留用于 VER_NDX_GLOBAL，依赖项版本可用的第一个索引为 2。

在 Oracle Solaris 10 发行版及更低 Solaris 版本中，不向 SHT_SUNW_verneed 依赖项版本指定版本索引。在此类目标文件中，任何符号引用都具有版本索引 0，这表示该符号没有可用的版本控制信息。

♦ ♦ ♦ 第 13 章

13 程序装入和动态链接

本章介绍用于创建运行程序的目标文件信息和系统操作。此处介绍的大多数信息适用于所有系统。特定于某处理器的信息位于带有相应标记的各节中。

可执行文件和共享目标文件静态表示应用程序。要执行这类程序，系统可使用这些文件创建动态程序表示形式（即进程映像）。进程映像具有包含其文本、数据、栈等内容的段。提供了以下主要节。

- “程序头” [343]，其中介绍了直接参与程序执行的目标文件结构。主数据结构是一种程序头表，用于定位文件中的段映像，并包含创建程序内存映像所需的其他信息。
- “程序装入（特定于处理器）” [348]，其中介绍了用于将程序装入内存的信息。
- “运行时链接程序” [356]，其中介绍了用于指定和解析进程映像的目标文件之间的符号引用的信息。

程序头

可执行文件或共享目标文件的程序头表是一个结构数组。每种结构都描述了系统准备程序执行所需的段或其他信息。目标文件段包含一个或多个节，如“段内容” [348]中所述。

程序头仅对可执行文件和共享目标文件有意义。文件使用 ELF 头的 `e_phentsize` 和 `e_phnum` 成员来指定其自己的程序头大小。

程序头具有以下结构。请参见 `sys/elf.h`。

```
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;
```

```
typedef struct {
        Elf64_Word      p_type;
        Elf64_Word      p_flags;
        Elf64_Off       p_offset;
        Elf64_Addr      p_vaddr;
        Elf64_Addr      p_paddr;
        Elf64_Xword     p_filesz;
        Elf64_Xword     p_memsz;
        Elf64_Xword     p_align;
} Elf64_Phdr;
```

`p_type`

此数组元素描述的段类型或解释此数组元素的信息的方式。表 13-1 “ELF 段类型” 中指定了类型值及其含义。

`p_offset`

相对段的第一个字节所在文件的起始位置的偏移。

`p_vaddr`

段的第一个字节在内存中的虚拟地址。

`p_paddr`

段在与物理寻址相关的系统中的物理地址。由于此系统忽略了应用程序的物理地址，因此该成员对于可执行文件和共享目标文件具有未指定的内容。

`p_filesz`

段的文件映像中的字节数，可以为零。

`p_memsz`

段的内存映像中的字节数，可以为零。

`p_flags`

与段相关的标志。表 13-2 “ELF 段标志” 中指定了类型值及其含义。

`p_align`

可装入的进程段必须具有 `p_vaddr` 和 `p_offset` 的同余值（以页面大小为模数）。此成员可提供一个值，用于在内存和文件中根据该值对齐各段。值 `0` 和 `1` 表示无需对齐。另外，`p_align` 应为 `2` 的正整数幂，并且 `p_vaddr` 应等于 `p_offset`（以 `p_align` 为模数）。请参见“程序装入（特定于处理器）” [348]。

某些项用于描述进程段。其他项则提供补充信息，并且不会构成进程映像。除非明确指定了顺序，否则段的各项可以任何顺序显示。下表中列出了定义的类型值。

表 13-1 ELF 段类型

名称	值
PT_NULL	0

名称	值
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_TLS	7
PT_LOOS	0x60000000
PT_SUNW_UNWIND	0x6464e550
PT_SUNW_EH_FRAME	0x6474e550
PT_LOSUNW	0x6ffffffa
PT_SUNWBSS	0x6ffffffa
PT_SUNWSTACK	0x6ffffffb
PT_SUNWDTRACE	0x6ffffffc
PT_SUNWCAP	0x6ffffffd
PT_HISUNW	0x6fffffff
PT_HIOS	0x6fffffff
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

PT_NULL

未使用。没有定义成员值。使用此类型，程序头表可以包含忽略的项。

PT_LOAD

指定由 p_filesz 和 p_memsz 描述的可装入段。文件中的字节会映射到内存段的起始位置。如果段的内存大小 (p_memsz) 大于文件大小 (p_filesz)，则将多余字节的值定义为 0。这些字节跟在段的已初始化区域后面。文件大小不能大于内存大小。程序头表中的可装入段的各项按升序显示，并基于 p_vaddr 成员进行排列。

PT_DYNAMIC

指定动态链接信息。请参见“动态节” [356]。

PT_INTERP

指定要作为解释程序调用的以空字符结尾的路径名的位置和大小。对于动态可执行文件，必须设置此类型。此类型可出现在共享目标文件中。此类型不能在一个文件中多次出现。此类型（如果存在）必须位于任何可装入段的各项的前面。有关详细信息，请参见“程序的解释程序” [355]。

PT_NOTE

指定辅助信息的位置和大小。有关详细信息，请参见“注释节” [312]。

PT_SHLIB

保留类型，但具有未指定的语义。

PT_PHDR

指定程序头表在文件及程序内存映像中的位置和大小。此段类型不能在一个文件中多次出现。此外，仅当程序头表是程序内存映像的一部分时，才可以出现此段。此类型（如果存在）必须位于任何可装入段的各项的前面。有关详细信息，请参见“程序的解释程序” [355]。

PT_TLS

指定线程局部存储模板。有关详细信息，请参见“线程局部存储节” [382]。

PT_LOOS - PT_HIOS

此范围内包含的值保留用于特定于操作系统的语义。

PT_SUNW_UNWIND

此段包含栈扩展表。

PT_SUNW_EH_FRAME

此段包含栈扩展表。PT_SUNW_EH_FRAME 与 PT_SUNW_EH_UNWIND 等效。

PT_LOSUNW - PT_HISUNW

此范围内包含的值（包括这两个值）保留用于特定于 Sun 的语义。

PT_SUNWBSS

与 PT_LOAD 元素相同的属性，用于描述 .SUNW_bss 节。

PT_SUNWSTACK

描述进程栈。只能存在一个 PT_SUNWSTACK 元素。仅访问权限（如 p_flags 字段中所定义）有意义。

PT_SUNWDTRACE

保留供 dtrace(1M) 内部使用。

PT_SUNWCAP

指定功能要求。有关详细信息，请参见“功能节” [306]。

PT_LOPROC - PT_HIPROC

此范围内包含的值（包括这两个值）保留用于特定于处理器的语义。

注 - 除非在其他位置具体要求，否则所有程序头的段类型都是可选的。文件的程序头表只能包含与其内容相关的那些元素。

基本地址

可执行文件和共享目标文件都有一个基本地址，该地址是与程序目标文件的内存映像关联的最低虚拟地址。基本地址的其中一种用途是在动态链接过程中重定位程序的内存映像。

可执行文件或共享目标文件的基本地址是在执行过程中通过以下三个值计算得出的：内存装入地址、最大页面大小和程序可装入段的最低虚拟地址。程序头中的虚拟地址可能并不表示程序内存映像的实际虚拟地址。请参见“程序装入（特定于处理器）” [348]。

要计算基本地址，首先需要确定与 PT_LOAD 段的最低 p_vaddr 值关联的内存地址。然后，将内存地址截断为最大页面大小的最接近倍数，从而获取基本地址。根据装入内存的文件的类型，内存地址可能与 p_vaddr 值不匹配。

段权限

系统要装入的程序必须至少包含一个可装入段，即使文件格式并不要求此限制也是如此。系统创建可装入段的内存映像时，将会授予如 p_flags 成员中所指定的访问权限。PF_MASKPROC 掩码中包括的所有位都保留用于特定于处理器的语义。

表 13-2 ELF 段标志

名称	值	含义
PF_X	0x1	执行
PF_W	0x2	写
PF_R	0x4	读
PF_MASKPROC	0xf0000000	未指定

如果权限位是 0，则会拒绝该位的访问类型。实际内存权限取决于内存管理单元，该单元可随系统的不同而变化。尽管所有标志组合均有效，但系统仍可授予比请求更多的访问权限。不过，如果不显式指定写权限，则段在任何情况下都不会具有该权限。下表列出了确切的标志解释及允许的标志解释。

表 13-3 ELF 段权限

标志	值	精确	允许
无	0	拒绝所有访问	拒绝所有访问
PF_X	1	仅执行	读、执行
PF_W	2	只写	读、写、执行
PF_W + PF_X	3	写、执行	读、写、执行
PF_R	4	只读	读、执行
PF_R + PF_X	5	读、执行	读、执行

标志	值	精确	允许
PF_R + PF_W	6	读、写	读、写、执行
PF_R + PF_W + PF_X	7	读、写、执行	读、写、执行

例如，典型的文本段具有读和执行权限，但没有写权限。数据段通常具有读、写和执行权限。

段内容

目标文件段由一节或多节组成，但此事实对程序头是透明的。另外，无论文件段包含一节还是包含多节，对程序装入都没有实际意义。但是，必须存在各种数据以便执行程序、进行动态链接等操作。下图使用一般术语说明了段内容。段中各节的顺序和成员关系可能会有所变化。

文本段包含只读指令和数据。数据段包含可写数据和指令。有关所有特殊节的列表，请参见表 12-12 “ELF 特殊节”。

PT_DYNAMIC 程序头元素指向 .dynamic 节。.got 和 .plt 节还包含与位置无关的代码和动态链接的相关信息。

.plt 可以位于文本或数据段中，具体取决于处理器。有关详细消息，请参见“全局偏移表（特定于处理器）” [371]和“过程链接表（特定于处理器）” [372]。

SHT_NOBITS 类型的节在文件中不占用任何空间，但会参与段的内存映像的构成。通常，这些未初始化的数据驻留在段尾，从而使 p_memsz 大于关联的程序头元素中的 p_filesz。

程序装入（特定于处理器）

系统创建或扩充进程映像时，系统会以逻辑方式将文件的段复制到虚拟内存段。系统以物理方式读取文件的时间和可能性取决于程序的执行行为、系统负载等。

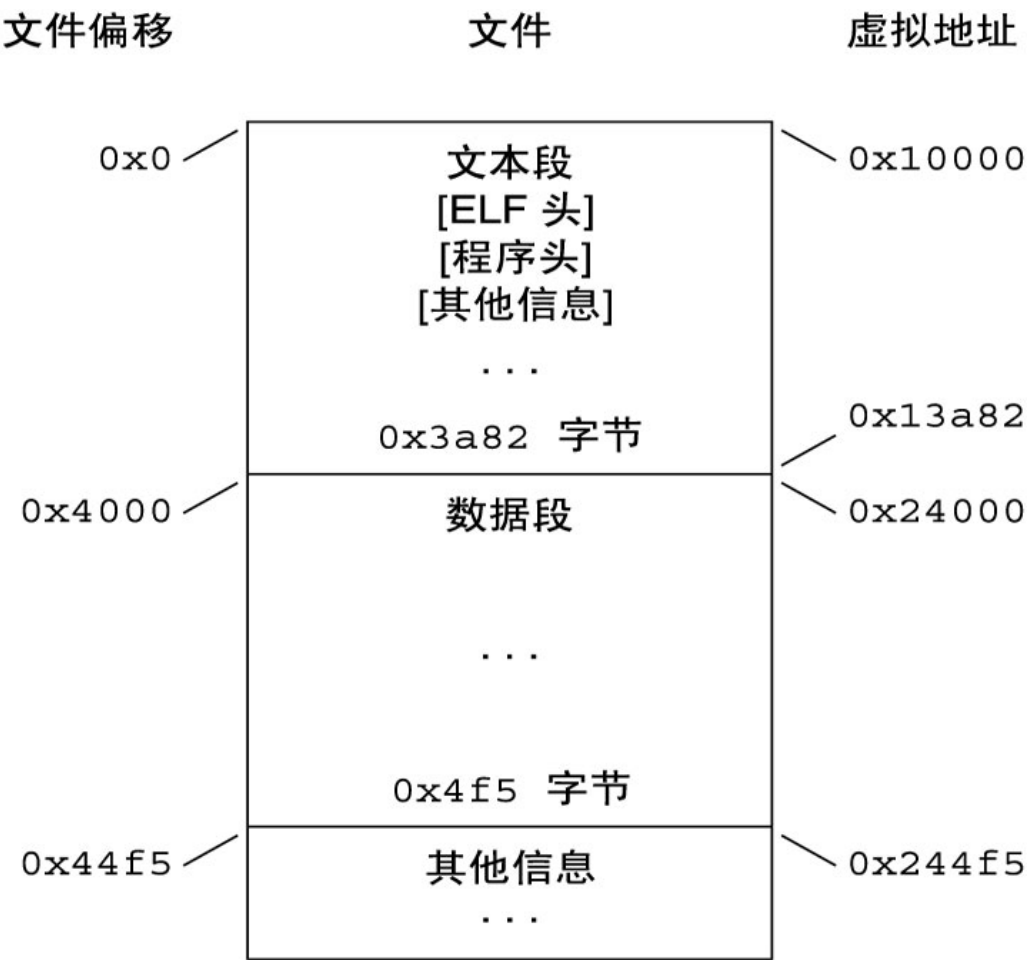
除非进程在执行过程中引用了逻辑页，否则进程不需要物理页。进程通常会保留许多页面不对其进行引用。因此，延迟物理读取可以提高系统性能。要实际达到这种效率，可执行文件和共享目标文件必须具有文件偏移和虚拟地址同余（以页面大小为模数）的段映像。

32 位段的虚拟地址和文件偏移对模数 64 K (0x10000) 同余。64 位段的虚拟地址和文件偏移对模数 1 MB (0x100000) 同余。通过将各段与最大页面大小对齐，无论物理页大小如何，文件都适合进行分页。

缺省情况下，64 位 SPARC 程序与 0x100000000 的起始地址链接。整个程序位于 4 GB 以上的地址空间内，包括其文本、数据、堆、栈和共享目标文件依赖项。这有助于确保 64 位程序正确，因为如果程序截断其任何指针，则程序在其最低有效的 4 GB 地址空间中将出现错误。尽管 64 位程序在 4 GB 以上的地址空间内进行链接，但您仍可以使用 mapfile 和链接编辑器的 -M 选项，链接 4 GB 以下的地址空间内的程序。请参见 /usr/lib/ld/sparcv9/map.below4G。

下图显示了 SPARC 版本的可执行文件。

图 13-1 SPARC: 可执行文件（64 K 对齐）



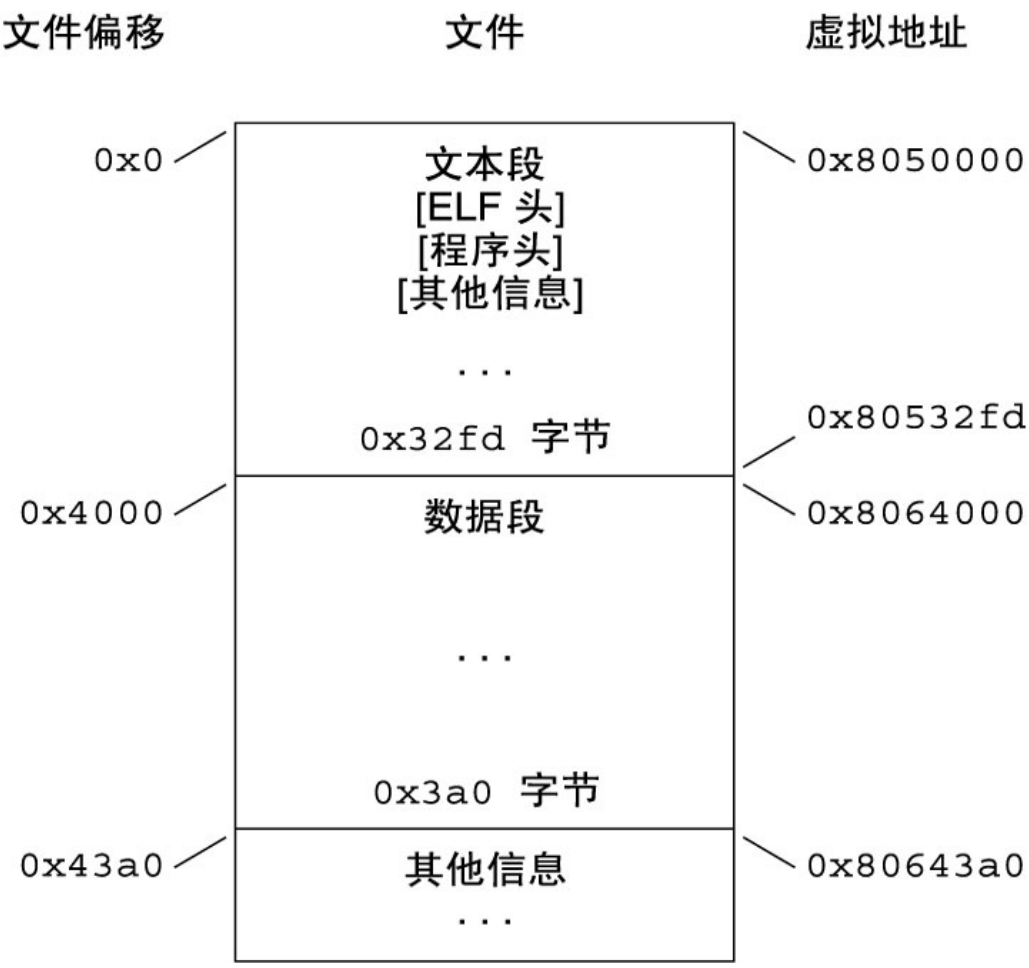
下表定义了上图中可装入段的各元素。

表 13-4 SPARC: ELF 程序头段（64 K 对齐）

成员	文本	数据
p_type	PT_LOAD	PT_LOAD
p_offset	0x0	0x4000
p_vaddr	0x10000	0x24000
p_paddr	未指定	未指定
p_filesize	0x3a82	0x4f5
p_memsz	0x3a82	0x10a4
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x10000	0x10000

下图显示了 x86 版本的可执行文件。

图 13-2　　32 位 x86: 可执行文件（64 K 对齐）



下表定义了上图中可装入段的各元素。

表 13-5　　32 位 x86: ELF 程序头段（64 K 对齐）

成员	文本	数据
p_type	PT_LOAD	PT_LOAD
p_offset	0x0	0x4000
p_vaddr	0x8050000	0x8064000
p_paddr	未指定	未指定

成员	文本	数据
p_filesize	0x32fd	0x3a0
p_memsz	0x32fd	0xdc4
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x10000	0x10000

此示例的文件偏移和虚拟地址以文本和数据的最大页面大小为模数同余。根据页面大小和文件系统块大小，最多可有四个文件页包含混合文本或数据。

- 第一个文本页包含 ELF 头、程序头表和其他信息。
- 最后一个文本页包含数据起始部分的副本。
- 第一个数据页包含文本结尾的副本。
- 最后一个数据页可以包含与运行的进程无关的文件信息。从逻辑上而言，系统会强制执行内存权限，如同每个段是完整而独立的一样。为确保地址空间中的每个逻辑页都具有单独一组权限，各段的地址会进行调整。在前面的示例中，包含文本结尾和数据起始部分的文件区域映射了两次：一次映射到文本的虚拟地址，另一次映射到数据的与之不同的虚拟地址。

注 - 前面的示例反映了对文本段取整的典型的 Oracle Solaris OS 二进制文件。

数据段结尾要求对未初始化的数据进行特殊处理，系统将其定义为从零值开始。如果文件的最后一个数据页包含不属于逻辑内存页的信息，则必须将无关数据设置为零，而不是设置为可执行文件的未知内容。

其他三页中的混合内容逻辑上不是进程映像的一部分。没有指定系统是否会清除这些混合内容。以下各图中显示了此程序的内存映像，假定页面大小为 4 KB (0x1000)。为简单起见，这些图仅对一种页面大小进行说明。

图 13-3 32 位 SPARC: 进程映像段

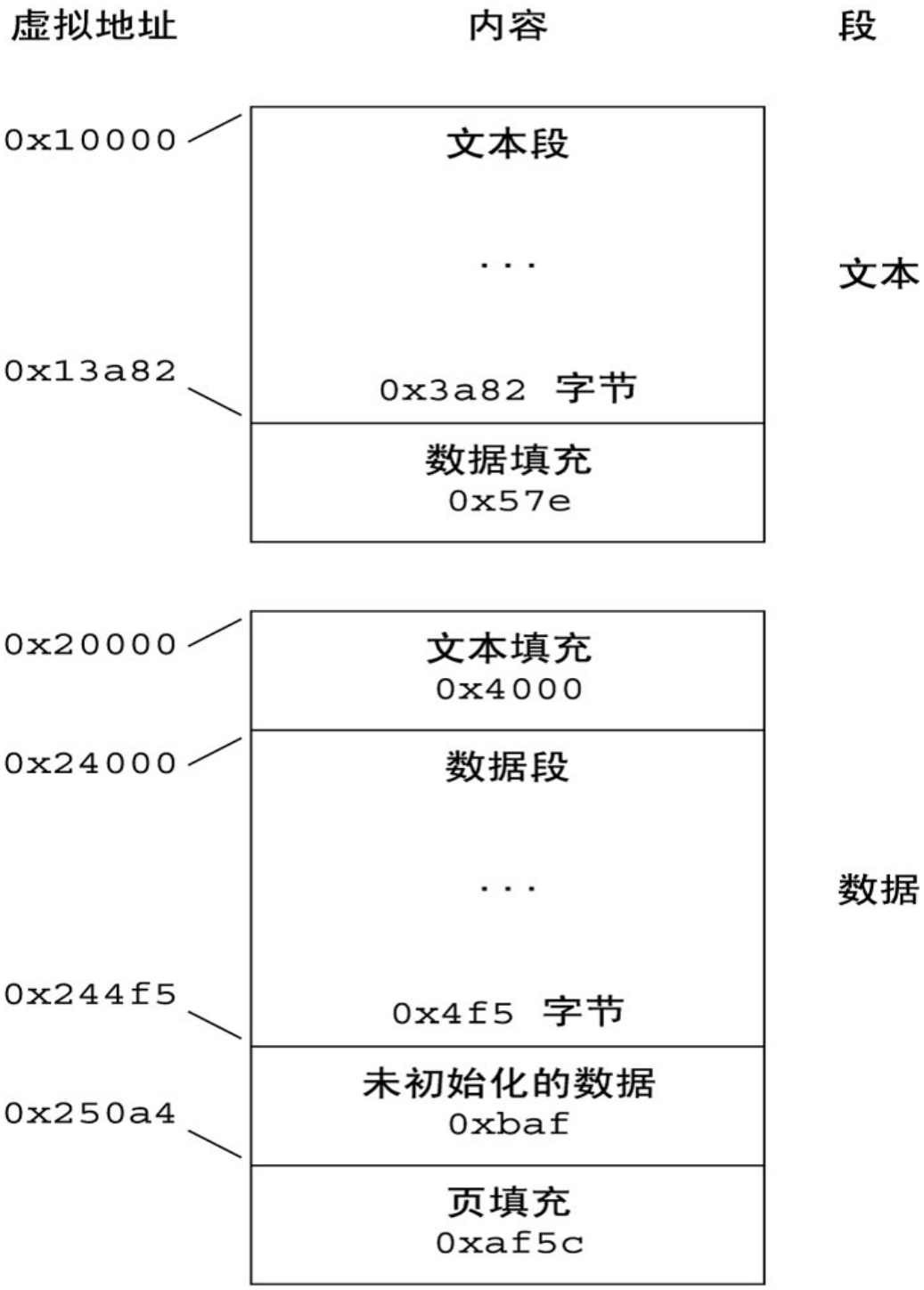
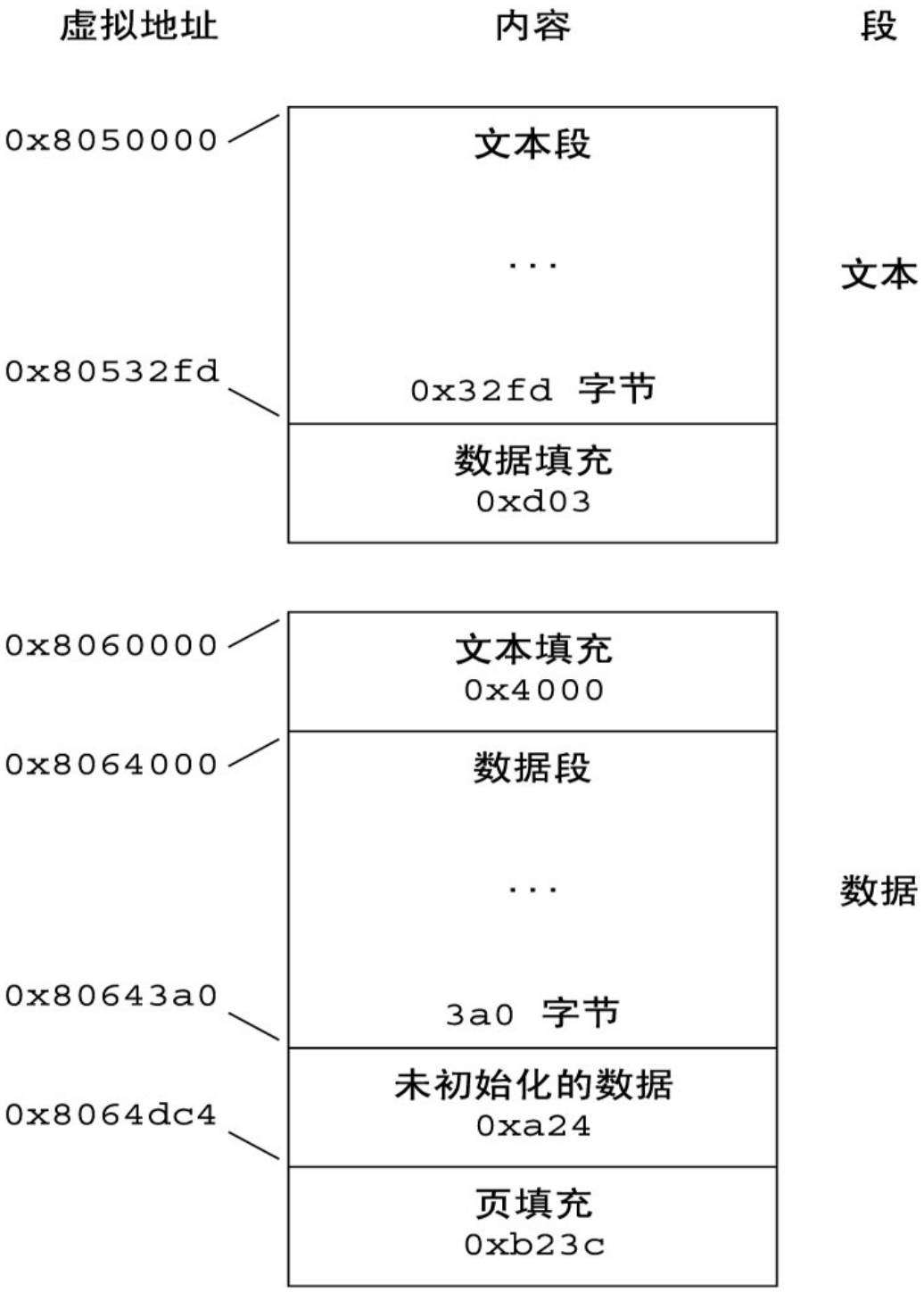


图 13-4 x86: 进程映像段



可执行文件和共享目标文件在段装入的某个方面有所不同。可执行文件段通常包含绝对代码。为使进程正确执行，段必须位于用于创建可执行文件的虚拟地址处。系统会使用未更改的 p_vaddr 值作为虚拟地址。

另一方面，共享目标文件段通常包含与位置无关的代码。使用此代码，段的虚拟地址在不同进程之间会进行更改，而不会使执行行为无效。

尽管系统会为各个进程选择虚拟地址，但仍会保持各段之间的相对位置。由于与位置无关的代码在各段之间使用相对地址，因此内存中虚拟地址之间的差值必须与文件中虚拟地址之间的差值匹配。

以下各表显示针对多个进程可能指定的共享目标文件虚拟地址，从而说明了固定的相对位置。此外，这些表中还包括基本地址计算。

表 13-6 32 位 SPARC: ELF 共享目标文件段地址示例

来源	文本	数据	基本地址
文件	0x0	0x4000	0x0
进程 1	0xc0000000	0xc0024000	0xc0000000
进程 2	0xc0010000	0xc0034000	0xc0010000
进程 3	0xd0020000	0xd0024000	0xd0020000
进程 4	0xd0030000	0xd0034000	0xd0030000

表 13-7 32 位 x86: ELF 共享目标文件段地址示例

来源	文本	数据	基本地址
文件	0x0	0x4000	0x0
进程 1	0x8000000	0x8004000	0x80000000
进程 2	0x80081000	0x80085000	0x80081000
进程 3	0x900c0000	0x900c4000	0x900c0000
进程 4	0x900c6000	0x900ca000	0x900c6000

程序的解释程序

启动动态链接的动态可执行文件或共享目标文件可以包含一个 PT_INTERP 程序头元素。在 exec(2) 过程中，系统将从 PT_INTERP 段检索路径名，并通过解释程序文件段创建初始进程映像。解释程序负责从系统接收控制并为应用程序提供环境。

在 Oracle Solaris OS 中，解释程序称为运行时链接程序，即 ld.so.1(1)。

运行时链接程序

创建启动动态链接的动态目标文件时，链接编辑器将向可执行文件中添加一个类型为 PT_INTERP 的程序头元素。该元素指示系统将运行时链接程序作为程序的解释程序进行调用。exec(2) 和运行时链接程序进行协作，为程序创建进程映像。

链接编辑器可为可执行文件和共享目标文件构造协助运行时链接程序运行的各种数据。这些数据位于可装入段中，从而使数据在执行过程中可用。这些段包括：

- 类型为 SHT_DYNAMIC 的 .dynamic 节，其中包含各种数据。位于该节起始位置的结构包含其他动态链接信息的地址。
- 类型为 SHT_PROGBITS 的 .got 和 .plt 节，其中分别包含以下两个表：全局偏移表和过程链接表。以下各节说明了运行时链接程序如何使用和更改这些表，以便为目标文件创建内存映像。
- 类型为 SHT_HASH 的 .hash 节，其中包含符号散列表。

共享目标文件可以占用虚拟内存地址，这些虚拟内存地址与文件的程序头表中记录的地址不同。运行时链接程序会重定位内存映像，从而在应用程序获取控制权之前更新绝对地址。

动态节

如果目标文件参与动态链接，则其程序头表将包含一个类型为 PT_DYNAMIC 的元素。此段包含 .dynamic 节。特殊符号 _DYNAMIC 用于标记包含以下结构的数组的节。请参见 sys/link.h。

```
typedef struct {
        Elf32_Sword d_tag;
        union {
                Elf32_Word      d_val;
                Elf32_Addr      d_ptr;
                Elf32_Off       d_off;
        } d_un;
} Elf32_Dyn;

typedef struct {
        Elf64_Xword d_tag;
        union {
                Elf64_Xword     d_val;
                Elf64_Addr      d_ptr;
        } d_un;
} Elf64_Dyn;
```

对于此类型的每个目标文件，d_tag 将控制 d_un 的解释。

d_val

这些目标文件表示具有各种解释的整数值。

d_ptr

这些目标文件表示程序虚拟地址。在执行过程中，文件虚拟地址可能与内存虚拟地址不匹配。对动态结构中包含的地址进行解释时，运行时链接程序会根据原始文件值和内存基本地址来计算实际地址。为确保一致性，文件不应包含用于更正动态结构中的地址的重定位项。

通常，每个动态标记的值决定了 d_un 联合的解释。借助此约定，第三方工具可进行更简单的动态标记解释。值为偶数的标记表示使用 d_ptr 的动态节项。值为奇数的标记表示使用 d_val 的动态节项，或表示该标记既不使用 d_ptr 也不使用 d_val。值包含在以下特殊兼容性范围中的标记不遵循这些规则。第三方工具必须逐项明确处理这些例外范围。

- 其值小于特定值 DT_ENCODING 的标记。
- 其值介于 DT_LOOS 和 DT_SUNW_ENCODING 之间的标记。
- 其值介于 DT_HIOS 和 DT_LOPROC 之间的标记。

下表概述了可执行文件和共享目标文件的标记要求。如果某标记带有强制标志，则动态链接数组必须包含此类型的项。同样，可选表示该标记的项可以出现但不是必需的。

表 13-8 ELF 动态数组标记

名称	值	d_un	可执行文件	共享目标文件
DT_NULL	0	已忽略	必选	必选
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	必选	必选
DT_STRTAB	5	d_ptr	必选	必选
DT_SYMTAB	6	d_ptr	必选	必选
DT_RELA	7	d_ptr	必选	可选
DT_RELASZ	8	d_val	必选	可选
DT_RELAENT	9	d_val	必选	可选
DT_STRSZ	10	d_val	必选	必选
DT_SYMENT	11	d_val	必选	必选
DT_INIT	12	d_ptr	可选	可选
DT_FINI	13	d_ptr	可选	可选
DT_SONAME	14	d_val	已忽略	可选
DT_RPATH	15	d_val	可选	可选
DT_SYMBOLIC	16	已忽略	已忽略	可选
DT_REL	17	d_ptr	必选	可选

名称	值	d_un	可执行文件	共享目标文件
DT_RELSZ	18	d_val	必选	可选
DT_RELENT	19	d_val	必选	可选
DT_PLTREL	20	d_val	可选	可选
DT_DEBUG	21	d_ptr	可选	已忽略
DT_TEXTREL	22	已忽略	可选	可选
DT_JMPREL	23	d_ptr	可选	可选
DT_BIND_NOW	24	已忽略	可选	可选
DT_INIT_ARRAY	25	d_ptr	可选	可选
DT_FINI_ARRAY	26	d_ptr	可选	可选
DT_INIT_ARRAYSZ	27	d_val	可选	可选
DT_FINI_ARRAYSZ	28	d_val	可选	可选
DT_RUNPATH	29	d_val	可选	可选
DT_FLAGS	30	d_val	可选	可选
DT_ENCODING	32	未指定	未指定	未指定
DT_PREINIT_ARRAY	32	d_ptr	可选	已忽略
DT_PREINIT_ARRAYSZ	33	d_val	可选	已忽略
DT_MAXPOSTAGS	34	未指定	未指定	未指定
DT_LOOS	0x6000000d	未指定	未指定	未指定
DT_SUNW_AUXILIARY	0x6000000d	d_ptr	未指定	可选
DT_SUNW_RTLDINF	0x6000000e	d_ptr	可选	可选
DT_SUNW_FILTER	0x6000000e	d_ptr	未指定	可选
DT_SUNW_CAP	0x60000010	d_ptr	可选	可选
DT_SUNW_SYMTAB	0x60000011	d_ptr	可选	可选
DT_SUNW_SYMSZ	0x60000012	d_val	可选	可选
DT_SUNW_ENCODING	0x60000013	未指定	未指定	未指定
DT_SUNW_SORTENT	0x60000013	d_val	可选	可选
DT_SUNW_SYMSORT	0x60000014	d_ptr	可选	可选
DT_SUNW_SYMSORTSZ	0x60000015	d_val	可选	可选
DT_SUNW_TLSSORT	0x60000016	d_ptr	可选	可选
DT_SUNW_TLSSORTSZ	0x60000017	d_val	可选	可选
DT_SUNW_CAPINFO	0x60000018	d_ptr	可选	可选
DT_SUNW_STRPAD	0x60000019	d_val	可选	可选
DT_SUNW_CAPCHAIN	0x6000001a	d_ptr	可选	可选
DT_SUNW_LDMACH	0x6000001b	d_val	可选	可选
DT_SUNW_CAPCHAINENT	0x6000001d	d_val	可选	可选
DT_SUNW_CAPCHAINSZ	0x6000001f	d_val	可选	可选
DT_SUNW_PARENT	0x60000021	d_val	可选	可选
DT_SUNW_ASLR	0x60000023	d_val	可选	已忽略

名称	值	d_un	可执行文件	共享目标文件
DT_SUNW_RELAX	0x60000025	d_val	可选	可选
DT_HIOS	0x6ffff000	未指定	未指定	未指定
DT_VALRNGLO	0x6ffffd00	未指定	未指定	未指定
DT_CHECKSUM	0x6ffffdf8	d_val	可选	可选
DT_PLTPADSZ	0x6ffffdf9	d_val	可选	可选
DT_MOVEENT	0x6ffffdfa	d_val	可选	可选
DT_MOVESZ	0x6ffffdfb	d_val	可选	可选
DT_POSFLAG_1	0x6ffffdfd	d_val	可选	可选
DT_SYMINSZ	0x6ffffdfe	d_val	可选	可选
DT_SYMINENT	0x6ffffdff	d_val	可选	可选
DT_VALRNGHI	0x6ffffdff	未指定	未指定	未指定
DT_ADDRRNGLO	0x6ffffe00	未指定	未指定	未指定
DT_CONFIG	0x6ffffefa	d_ptr	可选	可选
DT_DEPAUDIT	0x6ffffefb	d_ptr	可选	可选
DT_AUDIT	0x6ffffefc	d_ptr	可选	可选
DT_PLTPAD	0x6ffffefd	d_ptr	可选	可选
DT_MOVETAB	0x6ffffefe	d_ptr	可选	可选
DT_SYMINFO	0x6ffffeff	d_ptr	可选	可选
DT_ADDRRNGHI	0x6ffffeff	未指定	未指定	未指定
DT_RELACOUNT	0x6ffffff9	d_val	可选	可选
DT_RELCOUNT	0x6ffffffa	d_val	可选	可选
DT_FLAGS_1	0x6ffffffb	d_val	可选	可选
DT_VERDEF	0x6ffffffc	d_ptr	可选	可选
DT_VERDEFNUM	0x6ffffffd	d_val	可选	可选
DT_VERNEED	0x6ffffffe	d_ptr	可选	可选
DT_VERNEEDNUM	0x6fffffff	d_val	可选	可选
DT_LOPROC	0x70000000	未指定	未指定	未指定
DT_SPARC_REGISTER	0x70000001	d_val	可选	可选
DT_AUXILIARY	0x7ffffffd	d_val	未指定	可选
DT_USED	0x7ffffffe	d_val	可选	可选
DT_FILTER	0x7fffffff	d_val	未指定	可选
DT_HIPROC	0x7fffffff	未指定	未指定	未指定

DT_NULL

标记 _DYNAMIC 数组的结尾。

DT_NEEDED

以空字符结尾的字符串的 DT_STRTAB 字符串表偏移，用于提供所需依赖项的名称。动态数组可以包含多个此类型的项。尽管这些项与其他类型的项的关系不重要，但其相对顺序却很重要。请参见“共享目标文件依赖项” [88]。

DT_PLTRELSZ

与过程链接表关联的重定位项的总大小（字节）。请参见“过程链接表（特定于处理器）” [372]。

DT_PLTGOT

与过程链接表或全局偏移表关联的地址。请参见“过程链接表（特定于处理器）” [372]和“全局偏移表（特定于处理器）” [371]。

DT_HASH

符号散列表的地址。该表引用 DT_SYMTAB 元素指示的符号表。请参见“散列表节” [308]。

DT_STRTAB

字符串表的地址。运行时链接程序所需的符号名称、依赖项名称和其他字符串位于该表中。请参见“字符串表节” [325]。

DT_SYMTAB

符号表的地址。请参见“符号表节” [326]。

DT_RELA

重定位表的地址。请参见“重定位节” [314]。

目标文件可以有多个重定位节。为可执行文件或共享目标文件创建重定位表时，链接编辑器会连接这些节以形成一个表。尽管这些节在目标文件中可以保持独立，但运行时链接程序将看到一个表。运行时链接程序为可执行文件创建进程映像或将共享目标文件添加到进程映像中时，运行时链接程序将会读取该重定位表并执行关联操作。

此元素要求同时存在 DT_RELASZ 和 DT_RELAENT 元素。如果某个文件必须重定位，则可以存在 DT_RELA 和 DT_REL 中的一个。

DT_RELASZ

DT_RELA 重定位表的总大小（字节）。

DT_RELAENT

DT_RELA 重定位项的大小（字节）。

DT_STRSZ

DT_STRTAB 字符串表的总大小（字节）。

DT_SYMENT

DT_SYMTAB 符号项的大小（字节）。

DT_INIT

初始化函数的地址。请参见“初始化节和终止节” [33]。

DT_FINI

终止函数的地址。请参见“初始化节和终止节” [33]。

DT_SONAME

以空字符结尾的字符串的 DT_STRTAB 字符串表偏移，用于标识共享目标文件的名称。请参见“记录共享目标文件名称” [124]。

DT_RPATH

以空字符结尾的库搜索路径字符串的 DT_STRTAB 字符串表偏移。此元素的用途已被 DT_RUNPATH 取代。请参见“运行时链接程序搜索的目录” [88]。

DT_SYMBOLIC

表示目标文件包含在其链接编辑过程中应用的符号绑定。此元素已被 DF_SYMBOLIC 标志取代。请参见“使用 -B symbolic 选项” [173]。

DT_REL

与 DT_RELA 类似，但其表中包含隐式加数。此元素要求同时存在 DT_RELSZ 和 DT_RELENT 元素。

DT_RELSZ

DT_REL 重定位表的总大小（字节）。

DT_RELENT

DT_REL 重定位项的大小（字节）。

DT_PLTREL

表示过程链接表指向的重定位项的类型（DT_REL 或 DT_RELA）。过程链接表中的所有重定位都必须使用相同的重定位项。请参见“过程链接表（特定于处理器）” [372]。此元素要求同时存在 DT_JMPREL 元素。

DT_DEBUG

用于调试。

DT_TEXTREL

表示一个或多个重定位项可能会要求修改非可写段，并且运行时链接程序会进行相应准备。此元素已被 DF_TEXTREL 标志取代。请参见“与位置无关的代码” [162]。

DT_JMPREL

与过程链接表单独关联的重定位项的地址。请参见“过程链接表（特定于处理器）” [372]。通过分隔这些重定位项，运行时链接程序可在装入启用了延迟绑定的目标文件时忽略这些项。此元素要求同时存在 DT_PLTRELSZ 和 DT_PLTREL 元素。

DT_POSFLAG_1

应用于紧邻的 DT_ 元素的各种状态标志。请参见表 13-11 “ELF 动态位置标志 DT_POSFLAG_1”。

DT_BIND_NOW

表示在将控制权返回给程序之前，必须处理此目标文件的所有重定位项。通过环境或 dlopen(3C) 指定时，提供的此项优先于使用延迟绑定的指令。此元素的用途已被 DF_BIND_NOW 标志取代。请参见“执行重定位的时间” [169]。

DT_INIT_ARRAY

初始化函数的指针数组的地址。此元素要求同时存在 DT_INIT_ARRAYSZ 元素。请参见“初始化节和终止节” [33]。

DT_FINI_ARRAY

终止函数的指针数组的地址。此元素要求同时存在 DT_FINI_ARRAYSZ 元素。请参见“初始化节和终止节” [33]。

DT_INIT_ARRAYSZ

DT_INIT_ARRAY 数组的总大小（字节）。

DT_FINI_ARRAYSZ

DT_FINI_ARRAY 数组的总大小（字节）。

DT_RUNPATH

以空字符结尾的库搜索路径字符串的 DT_STRTAB 字符串表偏移。请参见“运行时链接程序搜索的目录” [88]。

DT_FLAGS

特定于此目标文件的标志值。请参见表 13-9 “ELF 动态标志 DT_FLAGS”。

DT_ENCODING

大于或等于 DT_ENCODING、小于或等于 DT_LOOS 的动态标记值遵循 d_un 联合的解释规则。

DT_PREINIT_ARRAY

预初始化函数的指针数组的地址。此元素要求同时存在 DT_PREINIT_ARRAYSZ 元素。仅在可执行文件中处理该数组。如果该数组包含在共享目标文件中，则会被忽略。请参见“初始化节和终止节” [33]。

DT_PREINIT_ARRAYSZ

DT_PREINIT_ARRAY 数组的总大小（字节）。

DT_MAXPOSTAGS

正动态数组标记值的数量。

DT_LOOS - DT_HIOS

此范围内包含的值（包括这两个值）保留用于特定于操作系统的语义。所有这类值都遵循 d_un 联合的解释规则。

DT_SUNW_AUXILIARY

以空字符结尾的字符串的 DT_STRTAB 字符串表偏移，用于逐符号指定一个或多个辅助 *filtee*。请参见“生成辅助过滤器” [131]。

DT_SUNW_RTLDINF

保留供运行时链接程序内部使用。

DT_SUNW_FILTER

以空字符结尾的字符串的 DT_STRTAB 字符串表偏移，用于逐符号指定一个或多个标准 *filtee*。请参见“生成标准过滤器” [128]。

DT_SUNW_CAP

功能节的地址。请参见“功能节” [306]。

DT_SUNW_SYMTAB

符号表的地址，其中包含用于扩充 DT_SYMTAB 所提供的符号的局部函数符号。这些符号始终在紧邻 DT_SYMTAB 所提供的符号之前的位置。请参见“符号表节” [326]。

DT_SUNW_SYMSZ

DT_SUNW_SYMTAB 和 DT_SYMTAB 提供的符号表的组合大小。

DT_SUNW_ENCODING

大于或等于 DT_SUNW_ENCODING、小于或等于 DT_HIOS 的动态标记值遵循 d_un 联合的解释规则。

DT_SUNW_SORTENT

DT_SUNW_SYMSORT 和 DT_SUNW_TLSSORT 符号排序项的大小（字节）。

DT_SUNW_SYMSORT

符号表索引数组的地址，这些索引提供对 DT_SUNW_SYMTAB 所引用的符号表中函数和变量符号的排序访问。请参见“符号排序节” [334]。

DT_SUNW_SYMSORTSZ

DT_SUNW_SYMSORT 数组的总大小（字节）。

DT_SUNW_TLSSORT

符号表索引数组的地址，这些索引提供对 DT_SUNW_SYMTAB 所引用的符号表中线程局部符号的排序访问。请参见“符号排序节” [334]。

DT_SUNW_TLSSORTSZ

DT_SUNW_TLSSORT 数组的总大小（字节）。

DT_SUNW_CAPINFO

符号表索引数组的地址，这些索引提供符号与其功能要求之间的关联。请参见“功能节” [306]。

DT_SUNW_STRPAD

动态字符串表末尾未使用的保留空间的总大小（字节）。如果目标文件中不存在 DT_SUNW_STRPAD，则没有保留空间可用。

DT_SUNW_CAPCHAIN

功能系列索引数组的地址。每个索引系列都以 0 项结尾。

DT_SUNW_LDMACH

生成目标文件的链接编辑器的计算机体系结构。DT_SUNW_LDMACH 与 ELF 头的 e_machine 字段使用相同的 EM_ 整数值。请参见“ELF 头” [274]。DT_SUNW_LDMACH 用于标识生成目标文件的链接编辑器的类（32 位或 64 位）和平台。此信息不会用于运行时链接程序，而仅用于说明目的。

DT_SUNW_CAPCHAINENT

DT_SUNW_CAPCHAIN 项的大小（字节）。

DT_SUNW_CAPCHAINSZ

DT_SUNW_CAPCHAIN 链的总大小（字节）。

DT_SUNW_PARENT

以空字符结尾的父目标文件名称的 DT_STRTAB 字符串表偏移。所提供的名称是基名，其中仅包含一个文件名且没有任何路径组件。请参见“父目标文件” [81]。

DT_SUNW_ASLR

特定于此目标文件的地址空间布局随机化 (Address Space Layout Randomization, ASLR) 标志值。请参见表 13-12 “ELF ASLR 值、DT_SUNW_ASLR”。

DT_SUNW_RELAX

生成目标文件时，与链接编辑器的 -z relax 选项一同指定的有效性检查放宽选项。请参见表 13-13 “ELF 动态放宽标志 DT_SUNW_RELAX”。

DT_SYMINFO

符号信息表的地址。此元素要求同时存在 DT_SYMINENT 和 DT_SYMINSZ 元素。请参见“*Syminfo* 表节” [336]。

DT_SYMINENT

DT_SYMINFO 信息项的大小（字节）。

DT_SYMINSZ

DT_SYMINFO 表的总大小（字节）。

DT_VERDEF

版本定义表的地址。该表中的元素包含字符串表 DT_STRTAB 的索引。此元素要求同时存在 DT_VERDEFNUM 元素。请参见“版本定义章节” [338]。

DT_VERDEFNUM

DT_VERDEF 表中的项数。

DT_VERNEED

版本依赖性表的地址。该表中的元素包含字符串表 DT_STRTAB 的索引。此元素要求同时存在 DT_VERNEEDNUM 元素。请参见“版本依赖性节” [339]。

DT_VERNEEDNUM

DT_VERNEEDNUM 表中的项数。

DT_RELACOUNT

表示 RELATIVE 重定位计数，该计数是通过串联所有 Elf32_Rela 或 Elf64_Rela 重定位项生成的。请参见“组合重定位节” [170]。

DT_RELCOUNT

表示 RELATIVE 重定位计数，该计数是通过串联所有 Elf32_Rel 重定位项生成的。请参见“组合重定位节” [170]。

DT_AUXILIARY

以空字符结尾的字符串的 DT_STRTAB 字符串表偏移，用于指定一个或多个辅助 *filtee*。请参见“生成辅助过滤器” [131]。

DT_FILTER

以空字符结尾的字符串的 DT_STRTAB 字符串表偏移，用于指定一个或多个标准 *filtee*。请参见“生成标准过滤器” [128]。

DT_CHECKSUM

目标文件中选定的节的简单校验和。请参见 gelf_checksum(3ELF)。

DT_MOVEENT

DT_MOVETAB 移动项的大小（字节）。

DT_MOVESZ

DT_MOVETAB 表的总大小（字节）。

DT_MOVETAB

移动表的地址。此元素要求同时存在 DT_MOVEENT 和 DT_MOVESZ 元素。请参见“移动部分” [310]。

DT_CONFIG

以空字符结尾的字符串的 DT_STRTAB 字符串表偏移，用于定义配置文件。该配置文件仅在可执行文件中有意义，并且通常是特定于此目标文件的。请参见“配置缺省搜索路径” [90]。

DT_DEPAUDIT

以空字符结尾的字符串的 DT_STRTAB 字符串表偏移，用于定义一个或多个审计库。请参见“运行时链接程序审计接口” [243]。

DT_AUDIT

以空字符结尾的字符串的 DT_STRTAB 字符串表偏移，用于定义一个或多个审计库。请参见“运行时链接程序审计接口” [243]。

DT_FLAGS_1

特定于此目标文件的标志值。请参见表 13-10 “ELF 动态标志 DT_FLAGS_1”。

DT_VALRNGLO - DT_VALRNGHI

此范围内包含的值（包括这两个值）使用动态结构的 d_un.d_val 字段。

DT_ADDRRNGLO - DT_ADDRRNGHI

此范围内包含的值（包括这两个值）使用动态结构的 d_un.d_ptr 字段。如果生成 ELF 目标文件后对其进行了任何调整，则必须相应地更新这些项。

DT_SPARC_REGISTER

DT_SYMTAB 符号表中 STT_SPARC_REGISTER 符号的索引。该符号表中的每个 STT_SPARC_REGISTER 符号都存在一个动态项。请参见“寄存器符号” [335]。

DT_LOPROC - DT_HIPROC

此范围内包含的值（包括这两个值）保留用于特定于处理器的语义。

除动态数组末尾的 DT_NULL 元素以及 DT_NEEDED 和 DT_POSFLAG_1 元素的相对顺序以外，各项可以采用任何顺序显示。未显示在该表中的标记值为保留值。

表 13-9 ELF 动态标志 DT_FLAGS

名称	值	含义
DF_ORIGIN	0x1	要求 $ORIGIN 处理
DF_SYMBOLIC	0x2	要求符号解析
DF_TEXTREL	0x4	存在文本重定位项
DF_BIND_NOW	0x8	要求非延迟绑定
DF_STATIC_TLS	0x10	目标文件使用静态线程局部存储方案

DF_ORIGIN

表示目标文件要求 $ORIGIN 处理。请参见“查找关联的依赖项” [231]。

DF_SYMBOLIC

表示目标文件包含在其链接编辑过程中应用的符号绑定。请参见“使用 -B symbolic 选项” [173]。

DF_TEXTREL

表示一个或多个重定位项可能会要求修改非可写段，并且运行时链接程序可以相应地进行准备。请参见“与位置无关的代码” [162]。

DF_BIND_NOW

表示在将控制权返回给程序之前，必须处理此目标文件的所有重定位项。通过环境或 dlopen(3C) 指定时，提供的此项优先于使用延迟绑定的指令。请参见“执行重定位的时间” [169]。

DF_STATIC_TLS

表示目标文件包含使用静态线程局部存储方案的代码。在通过 dlopen(3C) 或延迟装入而动态装入的目标文件中，不能使用静态线程局部存储。

表 13-10 ELF 动态标志 DT_FLAGS_1

名称	值	含义
DF_1_NOW	0x1	执行完整的重定位处理。
DF_1_GLOBAL	0x2	未使用。
DF_1_GROUP	0x4	表示目标文件是组的成员。
DF_1_NODELETE	0x8	不能从进程中删除目标文件。
DF_1_LOADFLTR	0x10	确保立即装入 *filtee*。
DF_1_INITFIRST	0x20	首先进行目标文件初始化。
DF_1_NOOPEN	0x40	目标文件不能用于 dlopen(3C)。
DF_1_ORIGIN	0x80	要求 $ORIGIN 处理。
DF_1_DIRECT	0x100	已启用直接绑定。
DF_1_INTERPOSE	0x400	目标文件是插入项。
DF_1_NODEFLIB	0x800	忽略缺省的库搜索路径。
DF_1_NODUMP	0x1000	不能使用 dldump(3C) 转储目标文件。
DF_1_CONFALT	0x2000	目标文件是配置替代项。
DF_1_ENDFILTEE	0x4000	*filtee* 终止过滤器搜索。
DF_1_DISPRELDNE	0x8000	已执行位移重定位。
DF_1_DISPRELPND	0x10000	位移重定位暂挂。
DF_1_NODIRECT	0x20000	目标文件包含非直接绑定。
DF_1_IGNMULDEF	0x40000	内部使用。

名称	值	含义
DF_1_NOKSYMS	0x80000	内部使用。
DF_1_NOHDR	0x100000	内部使用。
DF_1_EDITED	0x200000	目标文件在最初生成后已被修改。
DF_1_NORELOC	0x400000	内部使用。
DF_1_SYMINTPOSE	0x800000	存在各个符号插入项。
DF_1_GLOBAUDIT	0x1000000	建立全局审计。
DF_1_SINGLETON	0x2000000	存在单件符号。
DF_1_STUB	0x4000000	目标文件是桩目标文件。
DF_1_PIE	0x8000000	目标文件是与位置无关的可执行文件。

DF_1_NOW

表示在将控制权返回给程序之前，必须处理此目标文件的所有重定位项。通过环境或 dlopen(3C) 指定时，提供的此标志优先于使用延迟绑定的指令。请参见“执行重定位的时间” [169]。

DF_1_GROUP

表示目标文件是组的成员。此标志通过链接编辑器的 -B group 选项记录在目标文件中。请参见“目标文件分层结构” [112]。

DF_1_NODELETE

表示不能从进程中删除目标文件。如果使用 dlopen(3C) 通过直接或依赖性方式将目标文件装入进程，则无法使用 dlclose(3C) 卸载该目标文件。此标志通过使用链接编辑器的 -z nodelete 选项记录在目标文件中。

DF_1_LOADFLTR

仅对过滤器有意义。表示立即处理所有关联 *filtee*。此标志通过使用链接编辑器的 -z loadfltr 选项记录在目标文件中。请参见“*filtee* 处理” [134]。

DF_1_INITFIRST

表示在装入其他任何目标文件之前首先运行此目标文件的初始化节。此标志仅适用于专用系统库，并通过使用链接编辑器的 -z initfirst 选项记录在目标文件中。

DF_1_NOOPEN

表示无法使用 dlopen(3C) 将目标文件添加到正在运行的进程。此标志通过使用链接编辑器的 -z nodlopen 选项记录在目标文件中。

DF_1_ORIGIN

表示目标文件要求 $ORIGIN 处理。请参见“查找关联的依赖项” [231]。

DF_1_DIRECT

表示目标文件应使用直接绑定信息。请参见第 6 章 直接绑定。

DF_1_INTERPOSE

表示目标文件符号表将在除主装入目标文件（通常为可执行文件）外的所有符号之前插入。通过使用链接编辑器的 -z interpose 选项记录此标志。请参见“运行时插入” [93]。

DF_1_NODEFLIB

表示此目标文件的依赖性搜索会忽略所有缺省的库搜索路径。此标志通过使用链接编辑器的 -z nodefaultlib 选项记录在目标文件中。请参见“运行时链接程序搜索的目录” [32]。

DF_1_NODUMP

表示 dldump(3C) 不转储该目标文件。此选项的替代选项包括没有重定位项的目标文件，这些目标文件可能会包括在使用 crle(1) 生成的替代目标文件中。此标志通过使用链接编辑器的 -z nodump 选项记录在目标文件中。

DF_1_CONFALT

将此目标文件标识为 crle(1) 生成的配置替代目标文件。此标志可触发运行时链接程序来搜索配置文件 $ORIGIN/ld.config.*app-name*。

DF_1_ENDFILTEE

仅对 *filtee* 有意义。终止对其他任何 *filtee* 的过滤器搜索。此标志通过使用链接编辑器的 -z endfiltee 选项记录在目标文件中。请参见“减少 *filtee* 搜索” [230]。

DF_1_DISPRELDNE

表示此目标文件应用了位移重定位。由于位移重定位记录在应用重定位后被丢弃，因此该目标文件中将不再存在这些记录。请参见“位移重定位” [69]。

DF_1_DISPRELPND

表示此目标文件暂挂了位移重定位。由于此目标文件中存在位移重定位，因此可在运行时完成重定位。请参见“位移重定位” [69]。

DF_1_NODIRECT

表示此目标文件包含无法直接绑定的符号。请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。

DF_1_IGNMULDEF

保留供内核运行时链接程序内部使用。

DF_1_NOKSYMS

保留供内核运行时链接程序内部使用。

DF_1_NOHDR

保留供内核运行时链接程序内部使用。

DF_1_EDITED

表示此目标文件在最初由链接编辑器构造后，已被编辑或被修改。此标志用于警告调试器，某个目标文件在最初生成后进行了更改。

DF_1_NORELOC

保留供内核运行时链接程序内部使用。

DF_1_SYMINTPOSE

表示目标文件包含应在除主装入目标文件（通常为可执行文件）外的所有符号之前插入的各个符号。使用 `mapfile` 和 `INTERPOSE` 关键字生成目标文件时记录此标志。请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。

DF_1_GLOBAUDIT

表示动态可执行文件要求全局审计。请参见“记录全局审计程序” [246]。

DF_1_SINGLETON

表示目标文件定义或引用 `singleton` 符号。请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。

DF_1_STUB

表示目标文件是桩目标文件。请参见“桩目标文件” [70]。

DF_1_PIE

表示目标文件是与位置无关的可执行文件，这是共享目标文件的特例，用于指定解释程序。请参见链接编辑器的 `-z type` 选项。

表 13-11 ELF 动态位置标志 `DT_POSFLAG_1`

名称	值	含义
`DF_P1_LAZYLOAD`	`0x1`	标识延迟装入的依赖项。
`DF_P1_GROUPPERM`	`0x2`	标识组依赖性。

DF_P1_LAZYLOAD

将以下 `DT_NEEDED` 项标识为要延迟装入的目标文件。此标志通过使用链接编辑器的 `-z lazyload` 选项记录在目标文件中。请参见“延迟装入动态依赖项” [97]。

DF_P1_GROUPPERM

将以下 `DT_NEEDED` 项标识为要作为组装入的目标文件。此标志通过使用链接编辑器的 `-z groupperm` 选项记录在目标文件中。请参见“隔离组” [112]。

表 13-12 ELF ASLR 值、`DT_SUNW_ASLR`

名称	值	含义
`DV_SUNW_ASLR_DEFAULT`	`0`	采用系统缺省设置

名称	值	含义
DV_SUNW_ASLR_DISABLE	1	禁用 ASLR
DV_SUNW_ASLR_ENABLE	2	启用 ASLR

使用链接编辑器的 -z aslr 选项将 DV_SUNW_ASLR_DISABLE 和 DV_SUNW_ASLR_ENABLE 记录在目标文件中。

表 13-13 ELF 动态放宽标志 DT_SUNW_RELAX

DF_SUNW_RELAX_COMDAT	0x1	重定位符号替换废弃的 COMDAT
DF_SUNW_RELAX_SECADJ	0x2	已禁用节邻接验证
DF_SUNW_RELAX_SYMBOUND	0x4	已禁用符号/节边界验证
DF_SUNW_RELAX_COMMON	0x8	已启用不同大小或不同对齐方式的暂定（一般）数据。

使用链接编辑器的 -z relax 选项后，将在目标文件中记录 DF_SUNW_RELAX_ 标志。

全局偏移表（特定于处理器）

通常，与位置无关的代码不能包含绝对虚拟地址。全局偏移表在专用数据中包含绝对地址。因此这些地址可用，并且不会破坏程序文本的位置独立性和共享性。程序使用与位置无关的地址来引用其 GOT 并提取绝对值。此方法可将与位置无关的引用重定向到绝对位置。

最初，GOT 包含其重定位项所需的信息。系统为可装入目标文件创建内存段后，运行时链接程序将会处理这些重定位项。某些重定位项的类型可以为 R_xxxx_GLOB_DAT，用于引用 GOT。

运行时链接程序可确定关联符号值，计算其绝对地址以及将相应的内存表各项设置为正确的值。尽管链接编辑器创建目标文件时绝对地址未知，但运行时链接程序知道所有内存段的地址，因此可以计算其中包含的符号的绝对地址。

如果程序要求直接访问某符号的绝对地址，则该符号将具有一个 GOT 项。由于可执行文件和共享目标文件具有不同的 GOT，因此一个符号的地址可以出现在多个表中。运行时链接程序在向进程映像中的任何代码授予控制权之前，将首先处理所有的 GOT 重定位。此处理操作可确保绝对地址在执行过程中可用。

表项零保留用于存储动态结构（使用符号 _DYNAMIC 引用）的地址。使用此符号，运行时链接程序等程序可在尚未处理其重定位项的情况下查找各自的动态结构。此方法对于运行时链接程序尤其重要，因为它必须对自身进行初始化，而不依赖于其他程序来重定位其内存映像。

系统可为不同程序中的同一共享目标文件选择不同的内存段地址。系统甚至可以为同一程序的不同执行方式选择不同的库地址。但是，一旦建立进程映像，内存段即不会更改各地址。只要存在进程，其内存段就会位于固定的虚拟地址。

GOT 的格式和解释是特定于处理器的。可以使用符号 _GLOBAL_OFFSET_TABLE_ 访问该表。此符号可以位于 .got 节的中间，以提供地址数组的负下标和非负下标。对于 32 位代码，符号类型是 Elf32_Addr 数组；对于 64 位代码，符号类型是 Elf64_Addr 数组。

```
extern  Elf32_Addr  _GLOBAL_OFFSET_TABLE_[];
extern  Elf64_Addr  _GLOBAL_OFFSET_TABLE_[];
```

过程链接表（特定于处理器）

全局偏移表可将与位置无关的地址计算结果转换为绝对位置。同样，过程链接表也可将与位置无关的函数调用转换为绝对位置。链接编辑器无法解析不同动态目标文件之间的执行传输（如函数调用）。因此，链接编辑器会安排程序将控制权转移给过程链接表中的各项。这样，运行时链接程序就会重定向各项，而不会破坏程序文本的位置独立性和共享性。可执行文件和共享目标文件包含不同的过程链接表。

32 位 SPARC: 过程链接表

对于 32 位 SPARC 动态目标文件，过程链接表位于专用数据中。运行时链接程序可确定目标的绝对地址，并相应地修改过程链接表的内存映像。

过程链接表的前四项是保留项。尽管表 13-14 “过程链接表示例” 中显示了过程链接表的示例，但未指定这些项的原始内容。该表中的每一项都占用 3 个字（12 字节），并且表的最后一项后跟 nop 指令。

重定位表与过程链接表关联。_DYNAMIC 数组中的 DT_JMP_REL 项指定了第一个重定位项的位置。对于非保留的过程链接表的每一项，重定位表中都包含相同顺序的对应项。所有这些项的重定位类型均为 R_SPARC_JMP_SLOT。重定位偏移可指定关联的过程链接表项的第一个字节的地址。符号表索引会指向相应的符号。

为说明过程链接表，表 13-14 “过程链接表示例” 显示了四项。其中，前两项是初始保留项。第三项是对 name101 的调用。第四项是对 name102 的调用。此示例假定对应 name102 的项是表的最后一项。在该最后一项的后面是 nop 指令。左列显示了进行动态链接之前目标文件中的指令。右列说明了运行时链接程序会用于修复过程链接表各项的可能的指令序列。

表 13-14　32 位 SPARC: 过程链接表示例

目标文件	内存段
.PLT0:	.PLT0:

目标文件	内存段
unimp	save %sp, -64, %sp
unimp	call runtime_linker
unimp	nop
.PLT1:	.PLT1:
unimp	.word identification
unimp	unimp
unimp	unimp
.PLT101:	.PLT101:
sethi (.-.PLT0), %g1	nop
ba,a .PLT0	ba,a name101
nop	nop
.PLT102:	.PLT102:
sethi (.-.PLT0), %g1	sethi (.-.PLT0), %g1
ba,a .PLT0	sethi %hi(name102), %g1
nop	jmpl %g1+%lo(name102), %g0
nop	nop

以下步骤介绍了运行时链接程序和程序如何通过过程链接表来共同解析符号引用。所介绍的这些步骤仅用于说明。没有指定运行时链接程序的准确执行时行为。

1. 初始创建程序的内存映像时，运行时链接程序会更改初始过程链接表的各项。修改这些项是为了可将控制权转移给运行时链接程序自己的其中一个例程。运行时链接程序还会在第二项中存储一个字的标识信息。运行时链接程序获取控制权后，会检查该字以标识调用者。
2. 过程链接表的其他所有项最初都会传输给第一项。因此，运行时链接程序会在首次执行表项时获取控制权。例如，该程序会调用 `name101`，以将控制权转移给标签 `.PLT101`。
3. `sethi` 指令可分别计算当前过程链接表各项和初始过程链接表各项（`.PLT101` 和 `.PLT0`）之间的距离。该值会占用 `%g1` 寄存器最高有效的 22 位。
4. 接下来，`ba,a` 指令会跳至 `.PLT0` 以建立栈帧，然后调用运行时链接程序。
5. 通过标识值，运行时链接程序可获取其用于目标文件的数据结构，包括重定位表。
6. 通过将 `%g1` 值移位并除以过程链接表各项的大小，运行时链接程序可计算对应 `name101` 的重定位项的索引。重定位项 `101` 的类型为 `R_SPARC_JMP_SLOT`。此重定位偏移可指定 `.PLT101` 的地址，并且其符号表索引会指向 `name101`。因此，运行时链接程序可获取符号的实际值、展开栈、修改过程链接表项并将控制权转移给所需目标。

运行时链接程序不必在内存段列下创建指令序列。如果运行时链接程序创建了指令序列，则某些点需要更多说明。

- 要使代码可重复执行，可按特定顺序更改过程链接表的指令。如果运行时链接程序在修复函数的过程链接表项时收到信号，则信号处理代码必须能够调用具有可预测的正确结果的原始函数。
- 运行时链接程序更改三个字才能转换一项。对于指令执行，运行时链接程序只能自动更新一个字。因此，通过以相反顺序更新每个字可实现重复执行。如果仅在最后一个

修补程序之前调用可重复执行的函数，则运行时链接程序会再次获取控制权。尽管两次调用运行时链接程序修改的过程链接表项都相同，但这些更改不会相互干扰。

- 过程链接表项的第一条 `sethi` 指令可以填充 `jmp1` 指令的延迟插槽。尽管 `sethi` 会更改 `%g1` 寄存器的值，但可以安全放弃以前的内容。
- 转换之后，过程链接表的最后一项 `.PLT102` 需要一条延迟指令用于其 `jmp1`。所需的结尾 `nop` 将填充此延迟插槽。

注 - 为 `.PLT101` 和 `.PLT102` 显示的不同指令序列说明了如何优化关联目标的更新。

`LD_BIND_NOW` 环境变量更改动态链接行为。如果其值不为空，则运行时链接程序会在将控制权转移给程序之前处理 `R_SPARC_JMP_SLOT` 重定位项。

64 位 SPARC: 过程链接表

对于 64 位 SPARC 动态目标文件，过程链接表位于专用数据中。运行时链接程序可确定目标的绝对地址，并相应地修改过程链接表的内存映像。

过程链接表的前四项是保留项。尽管表 13-15 “过程链接表示例” 中显示了过程链接表的示例，但未指定这些项的原始内容。在该表中，前 32,768 项的每一项都占用 8 个字（32 字节），并且必须与 32 字节边界对齐。整个表必须与 256 字节边界对齐。如果所需项数大于 32,768，则其余各项由 6 个字（24 字节）和 1 个指针（8 字节）组成。指令以 160 项并后跟 160 个指针的块方式收集到一起。最后一组项和指针可以包含的项数少于 160。不需要进行填充。

注 - 数字 32,768 和 160 分别基于分支和装入目标文件位移的限制，并且位移会向下舍入以使代码和数据之间的分区落到 256 字节边界上，从而提高高速缓存的性能。

重定位表与过程链接表关联。`_DYNAMIC` 数组中的 `DT_JMP_REL` 项指定了第一个重定位项的位置。对于非保留的过程链接表的每一项，重定位表中都包含相同顺序的对应项。所有这些项的重定位类型均为 `R_SPARC_JMP_SLOT`。对于前 32,767 个插槽，重定位偏移将指定关联过程链接表项的第一个字节的地址，并且加数字段为零。符号表索引会指向相应的符号。对于插槽 32,768 及其之后的插槽，重定位偏移将指定关联指针的第一个字节的地址。加数字段是未重定位的值 `-(.PLTN + 4)`。符号表索引会指向相应的符号。

为说明过程链接表，表 13-15 “过程链接表示例” 显示了若干个项。前三项显示了初始保留项。接下来的三项显示了初始的 32,768 个项的示例以及可能的解析格式，这些格式分别应用于目标地址位于项上下 2 GB 的地址空间内、目标地址位于低位的 4 GB 地址空间内或目标地址位与其他任意位置的情形。最后两项显示了后续各项的示例，这些项由指令和指针对组成。左列显示了进行动态链接之前目标文件中的指令。右列说明了运行时链接程序会用于修复过程链接表各项的可能指令序列。

表 13-15　64 位 SPARC: 过程链接表示例

目标文件	内存段

.PLT0:
    unimp
    unimp
    unimp
    unimp
    unimp
    unimp
    unimp
.PLT1:
    unimp
    unimp
    unimp
    unimp
    unimp
    unimp
    unimp
    unimp
.PLT2:
    unimp
	.PLT0:
	    save %sp, -176, %sp
	    sethi %hh(runtime_linker_0), %l0
	    sethi %lm(runtime_linker_0), %l1
	    or %l0, %hm(runtime_linker_0), %l0
	    sllx %l0, 32, %l0
	    or %l0, %l1, %l0
	    jmpl %l0+%lo(runtime_linker_0), %o1
	    mov %g1, %o0
	.PLT1:
	    save %sp, -176, %sp
	    sethi %hh(runtime_linker_1), %l0
	    sethi %lm(runtime_linker_1), %l1
	    or %l0, %hm(runtime_linker_1), %l0
	    sllx %l0, 32, %l0
	    or %l0, %l1, %l0
	    jmpl %l0+%lo(runtime_linker_0), %o1
	    mov %g1, %o0
	.PLT2:
	    .xword identification

.PLT101:
    sethi   (.-.PLT0), %g1
    ba,a   %xcc, .PLT1
    nop
    nop
    nop;   nop
    nop;   nop
	.PLT101:
	    nop
	    mov %o7, %g1
	    call name101
	    mov %g1, %o7
	    nop; nop
	    nop; nop

.PLT102:
    sethi   (.-.PLT0), %g1
    ba,a   %xcc, .PLT1
    nop
    nop
    nop;   nop
    nop;   nop
	.PLT102:
	    nop
	    sethi %hi(name102), %g1
	    jmpl %g1+%lo(name102), %g0
	    nop
	    nop; nop
	    nop; nop

.PLT103:
    sethi   (.-.PLT0), %g1
    ba,a   %xcc, .PLT1
    nop
    nop
    nop
    nop
    nop
    nop
	.PLT103:
	    nop
	    sethi %hh(name103), %g1
	    sethi %lm(name103), %g5
	    or %hm(name103), %g1
	    sllx %g1, 32, %g1
	    or %g1, %g5, %g5
	    jmpl %g5+%lo(name103), %g0
	    nop

.PLT32768:
    mov   %o7, %g5
    call   .+8
    nop
    ldx   [%o7+.PLTP32768 -
              (.PLT32768+4)], %g1
    jmpl   %o7+%g1, %g1
	.PLT32768:
	<unchanged>
	<unchanged>
	<unchanged>
	<unchanged>
	<unchanged>

目标文件	内存段
`    mov     %g5, %o7`	`    <unchanged>`
`    ....`	`    ....`
`.PLT32927:`	`.PLT32927:`
`    mov     %o7, %g5`	`    <unchanged>`
`    call    .+8`	`    <unchanged>`
`    nop`	`    <unchanged>`
`    ldx     [%o7+.PLTP32927 -` `              (.PLT32927+4)], %g1`	`    <unchanged>`
`    jmpl    %o7+%g1, %g1`	`    <unchanged>`
`    mov     %g5, %o7`	`    <unchanged>`
`.PLTP32768`	`.PLTP32768`
`    .xword  .PLT0 -` `              (.PLT32768+4)`	`    .xword  name32768 -` `                (.PLT32768+4)`
`    ....`	`    ....`
`.PLTP32927`	`    .PLTP32927`
`    .xword  .PLT0 -` `              (.PLT32927+4)`	`    .xword  name32927 -` `                (.PLT32927+4)`

以下步骤介绍了运行时链接程序和程序如何通过过程链接表来共同解析符号引用。所介绍的这些步骤仅用于说明。没有指定运行时链接程序的准确执行时行为。

1. 初始创建程序的内存映像时，运行时链接程序会更改初始过程链接表的各项。修改这些项是为了将控制权转移给运行时链接程序自己的例程。运行时链接程序还会在第三项中存储一个扩展字的标识信息。运行时链接程序获取控制权后，会检查该字以标识调用者。
2. 过程链接表的其他所有项最初都会传输给第一项或第二项。这些项将建立栈帧并调用运行时链接程序。
3. 通过标识值，运行时链接程序可获取其用于目标文件的数据结构，包括重定位表。
4. 运行时链接程序会计算表插槽对应的重定位项的索引。
5. 使用索引信息，运行时链接程序可获取符号的实际值、展开栈、修改过程链接表项并将控制权转移给所需目标。

运行时链接程序不必在内存段列下创建指令序列。如果运行时链接程序创建了指令序列，则某些点需要更多说明。

- 要使代码可重复执行，可按特定顺序更改过程链接表的指令。如果运行时链接程序在修复函数的过程链接表项时收到信号，则信号处理代码必须能够调用具有可预测的正确结果的原始函数。
- 运行时链接程序最多可更改八个字来转换一项。对于指令执行，运行时链接程序只能自动更新一个字。因此，通过以下方法可实现重复执行：首先将 `nop` 指令覆写为其替换指令，如果使用 64 位存储，则之后还要修补 `ba,a` 和 `sethi`。如果仅在最后一个修补程序之前调用可重复执行的函数，则运行时链接程序会再次获取控制权。尽管两次调用运行时链接程序修改的过程链接表项都相同，但这些更改不会相互干扰。
- 如果更改初始 `sethi` 指令，则只能将该指令替换为 `nop`。

按照更改第二种格式的项的方式更改指针是通过使用一个原子的 64 位存储器完成的。

注 - 为 .PLT101、.PLT102 和 .PLT103 显示的不同指令序列说明了如何优化关联目标的更新。

LD_BIND_NOW 环境变量可更改动态链接行为。如果其值不为空，则运行时链接程序会在将控制权转移给程序之前处理 R_SPARC_JMP_SLOT 重定位项。

32 位 x86: 过程链接表

对于 32 位 x86 动态目标文件，过程链接表位于共享文本中，但使用专用全局偏移表中的地址。运行时链接程序可确定目标的绝对地址，并相应地修改全局偏移表的内存映像。这样，运行时链接程序就会重定向各项，而不会破坏程序文本的位置独立性和共享性。可执行文件和共享目标文件包含不同的过程链接表。

表 13-16　32 位 x86: 绝对过程链接表示例

```
.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
```

表 13-17　32 位 x86: 与位置无关的过程链接表示例

```
.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
```

注 - 如前面的示例所示，对于绝对代码和与位置无关的代码，过程链接表指令会使用不同的操作数寻址模式。但是，它们的运行时链接程序接口却相同。

以下步骤介绍了运行时链接程序和程序如何通过过程链接表和全局偏移表来协作解析符号引用。

1. 初始创建程序的内存映像时，运行时链接程序会将全局偏移表中的第二项和第三项设置为特殊值。以下步骤说明了这些值。
2. 如果过程链接表与位置无关，则全局偏移表的地址必须位于 `%ebx` 中。进程映像中的每个共享目标文件都有各自的过程链接表，并且控制权仅转移给位于同一目标文件内的过程链接表项。因此，调用函数在调用过程链接表项之前，必须首先设置全局偏移表基本寄存器。
3. 例如，该程序会调用 `name1`，以将控制权转移给标签 `.PLT1`。
4. 第一条指令会跳至全局偏移表项中对应于 `name1` 的地址。最初，全局偏移表保存以下 `pushl` 指令的地址，而不是 `name1` 的实际地址。
5. 该程序将在栈中推送一个重定位偏移 (`offset`)。该重定位偏移是重定位表中一个 32 位的非负字节偏移。指定的重定位项的类型为 `R_386_JMP_SLOT`，其偏移指定了前面的 `jmp` 指令中使用的全局偏移表项。该重定位项还包含符号表索引，以供运行时链接程序用于获取引用的符号 `name1`。
6. 推送该重定位偏移后，程序将跳至过程链接表中的第一项 `.PLT0`。`pushl` 指令会在栈中推送全局偏移表的第二项（`got_plus_4` 或 `4(%ebx)`）的值，从而为运行时链接程序提供一个字的标识信息。然后，程序将跳至全局偏移表的第三项（`got_plus_8` 或 `8(%ebx)`）中的地址，以继续跳至运行时链接程序。
7. 运行时链接程序将展开栈、检查指定的重定位项、获取符号的值、在全局偏移项表中存储 `name1` 的实际地址并跳至目标。
8. 过程链接表项的后续执行结果会直接传输给 `name1`，而不会再次调用运行时链接程序。位于 `.PLT1` 的 `jmp` 指令将跳至 `name1`，而不是对 `pushl` 指令失败。

`LD_BIND_NOW` 环境变量可更改动态链接行为。如果其值不为空，则运行时链接程序会在将控制权转移给程序之前处理 `R_386_JMP_SLOT` 重定位项。

x64: 过程链接表

对于 x64 动态目标文件，过程链接表位于共享文本中，但使用专用全局偏移表中的地址。运行时链接程序可确定目标的绝对地址，并相应地修改全局偏移表的内存映像。这样，运行时链接程序就会重定向各项，而不会破坏程序文本的位置独立性和共享性。可执行文件和共享目标文件包含不同的过程链接表。

表 13-18　x64: 过程链接表示例

```
.PLT0:
```

```
    pushq   GOT+8(%rip)                         # GOT[1]
    jmp     *GOT+16(%rip)                       # GOT[2]
    nop;    nop
    nop;    nop
.PLT1:
    jmp     *name1@GOTPCREL(%rip)               # 16 bytes from .PLT0
    pushq   $index1
    jmp     .PLT0
.PLT2:
    jmp     *name2@GOTPCREL(%rip)               # 16 bytes from .PLT1
    pushl   $index2
    jmp     .PLT0
```

以下步骤介绍了运行时链接程序和程序如何通过过程链接表和全局偏移表来协作解析符号引用。

1. 初始创建程序的内存映像时，运行时链接程序会将全局偏移表中的第二项和第三项设置为特殊值。以下步骤说明了这些值。
2. 进程映像中的每个共享目标文件都有各自的过程链接表，并且控制权仅转移给位于同一目标文件内的过程链接表项。
3. 例如，该程序会调用 name1，以将控制权转移给标签 .PLT1。
4. 第一条指令会跳至全局偏移表项中对应于 name1 的地址。最初，全局偏移表保存以下 pushq 指令的地址，而不是 name1 的实际地址。
5. 该程序将在栈中推送一个重定位索引 (index1)。该重定位索引是重定位表中一个 32 位的非负索引。重定位表由 DT_JUMPREL 动态节项标识。指定的重定位项的类型为 R_AMD64_JMP_SLOT，其偏移指定了前面的 jmp 指令中使用的全局偏移表项。该重定位项还包含符号表索引，以供运行时链接程序用于获取引用的符号 name1。
6. 推送该重定位索引后，程序将跳至过程链接表中的第一项 .PLT0。pushq 指令会在栈中推送全局偏移表的第二项 (GOT+8) 的值，从而为运行时链接程序提供一个字的标识信息。然后，程序将跳至第三个全局偏移表项 (GOT+16) 中的地址，以继续跳至运行时链接程序。
7. 运行时链接程序将展开栈、检查指定的重定位项、获取符号的值、在全局偏移项表中存储 name1 的实际地址并跳至目标。
8. 过程链接表项的后续执行结果会直接传输给 name1，而不会再次调用运行时链接程序。位于 .PLT1 的 jmp 指令将跳至 name1，而不是对 pushq 指令失败。

LD_BIND_NOW 环境变量可更改动态链接行为。如果其值不为空，则运行时链接程序会在将控制权转移给程序之前处理 R_AMD64_JMP_SLOT 重定位项。

♦ ♦ ♦ 第 14 章

14 线程局部存储

编译环境支持声明线程局部数据。此类数据有时称为线程特定数据或线程专用数据，但更多时候以首字母缩略词 TLS 表示。通过将变量声明为线程局部变量，编译器可自动安排针对每个线程分配这些变量。

提供对此功能的内置支持有三个目的。

- 提供生成 POSIX 接口的基础，此接口用于分配线程特定数据。
- 提供一种方便高效的机制，以便应用程序和库直接使用线程局部变量。
- 执行循环并行优化时，编译器可以根据需要分配 TLS。

C/C++ 编程接口

使用 `__thread` 关键字可将变量声明为线程局部变量，如下例所示。

```
__thread int i;
__thread char *p;
__thread struct state s;
```

在循环优化期间，编译器可根据需要选择创建临时线程局部变量。

适用性

`__thread` 关键字可以应用于任何全局变量、文件作用域静态变量或函数作用域静态变量。它对于始终是线程局部变量的自动变量没有影响。

初始化

在 C++ 中，如果初始化需要静态构造函数，将无法初始化线程局部变量。否则，可以将线程局部变量初始化为对于普通静态变量合法的任何值。

无论是线程局部变量还是其他变量，都不能静态地初始化为线程局部变量的地址。

绑定

线程局部变量可以在外部声明和引用。线程局部变量遵循与普通符号相同的插入规则。

动态装入限制

可用的 TLS 访问模型有多种。请参见“线程局部存储的访问模型” [386]。共享目标文件开发者应意识到访问模型所带来的与目标文件装入有关的限制。共享目标文件可以在进程启动期间或进程启动之后通过延迟装入、过滤器或 dlopen(3C) 动态地装入。进程启动完成后，将建立主线程的线程指针。在线程指针建立前，会计算所有的静态 TLS 存储需求。

引用线程局部变量的共享目标文件应确保每个包含引用的转换单元都使用动态的 TLS 模型进行编译。这种访问模型可在装入共享目标文件方面提供更大的灵活性。但是，静态 TLS 模型可生成执行速度更快的代码。使用静态 TLS 模型的共享目标文件可以作为进程初始化的一部分装入。但是，进程初始化之后，使用静态 TLS 模型的共享目标文件将仅在有足够的备份 TLS 存储空间可用时才能装入。请参见“程序启动” [384]。

寻址运算符

寻址运算符 & 可用于线程局部变量。此运算符在运行时计算，并返回当前线程中变量的地址。进程中的任何线程都可自由使用此运算符获取的地址，前提是计算该地址的线程始终存在。当线程终止时，任何指向该线程中的线程局部变量的指针都将变为无效。

使用 dlsym(3C) 获取线程局部变量的地址时，返回的地址是调用 dlsym () 的线程中该变量的实例地址。

线程局部存储节

编译时分配的线程局部数据的独立副本必须与各个执行线程关联。要提供此数据，应使用 TLS 节指定大小和初始内容。编译环境在 SHF_TLS 标志标识的节中分配 TLS。这些节根据存储的声明方式提供已初始化的 TLS 和未初始化的 TLS。

- 已初始化的线程局部变量分配在 .tdata 或 .tdata1 节中。此初始化可能需要重定位。
- 未初始化的线程局部变量定义为 COMMON 符号。最终分配在 .tbss 节中进行。

在分配了任何已初始化的节后会立即分配未初始化的节，并进行填充以便正确对齐。合并的节一起构成 TLS 模板，每次创建新线程时，都会使用此模板分配 TLS。此模板的已初始化部分称为 TLS 初始化映像。所有因已初始化的线程局部变量而生成的重定位将应用于此模板。当新线程需要初始值时，将使用重定位的值。

TLS 符号的符号类型为 STT_TLS。这些符号被指定了相对于 TLS 模板开头的偏移。与这些符号关联的实际虚拟地址与此无关。地址仅指向模板，而不指向每个数据项的每线程副本。在动态可执行文件和共享目标文件中，对于已定义符号，STT_TLS 符号的 st_value 字段包含指定的 TLS 偏移。对于未定义的符号，此字段包含零。

定义了多个重定位以支持访问 TLS。请参见“线程局部存储的重定位类型” [392]、“线程局部存储的重定位类型” [398]和“线程局部存储的重定位类型” [402]。通常，TLS 重

定位引用 STT_TLS 类型的符号。TLS 重定位还可以引用与 GOT 项关联的局部节符号。在这种情况下，指定的 TLS 偏移存储在关联的 GOT 项中。

对于根据 TLS 项进行的重定位，重定位地址在 TLS 模板的末尾编码为负偏移。计算该偏移时，首先将模板大小舍入到 32 位目标文件中最接近的 8 字节边界，然后舍入为 64 位目标文件中最接近的 16 字节边界。此舍入操作确保静态 TLS 模板合理对齐以便可用于任何用途。

在动态可执行文件和共享目标文件中，PT_TLS 程序项用于描述 TLS 模板。此模板包含以下成员：

表 14-1 ELF PT_TLS 程序头项

成员	值
p_offset	TLS 初始化映像的文件偏移
p_vaddr	TLS 初始化映像的虚拟内存地址
p_paddr	0
p_filesz	TLS 初始化映像的大小
p_memsz	TLS 模板的总大小
p_flags	PF_R
p_align	TLS 模板的对齐方式

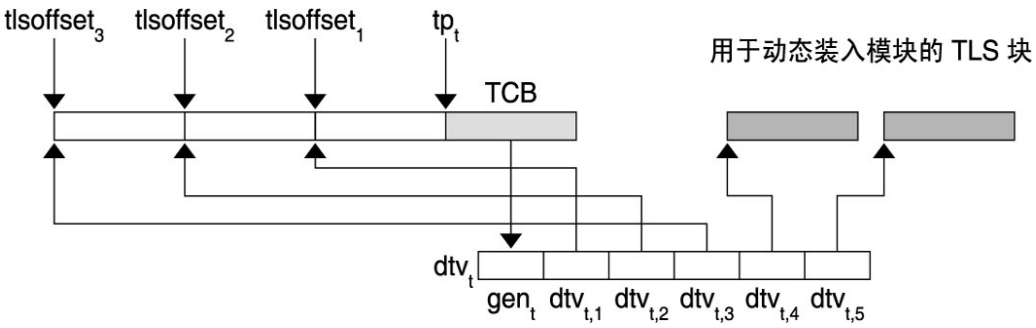
线程局部存储的运行时分配

在程序的生命周期中，会在三个时间创建 TLS。

- 程序启动时。
- 创建新线程时。
- 程序启动后装入共享目标文件之后，线程第一次引用 TLS 块时。

运行时线程局部数据存储的布局如图 14-1 “线程局部存储的运行时存储布局” 中所示。

图 14-1 线程局部存储的运行时存储布局



程序启动

在程序启动时，运行时系统将为主线程创建 TLS。

首先，运行时链接程序以逻辑方式将所有已装入动态目标文件（包括动态可执行文件）的 TLS 模板合并成单个静态模板。在合并的模板中，为每个动态目标文件的 TLS 模板指定一个偏移 $tlsoffset_m$，如下所示。

- $tlsoffset_1 = round(tlssize_1, align_1)$
- $tlsoffset_{m+1} = round(tlsoffset_m + tlssize_{m+1}, align_{m+1})$

$tlssize_{m+1}$ 和 $align_{m+1}$ 分别是动态目标文件 m 的分配模板的大小和对齐方式。其中，$1 <= m <= M$，而 M 是已装入动态目标文件的总数。round(offset, align) 函数返回一个偏移，该偏移向上舍入为最接近 *align* 倍数。

接下来，运行时链接程序将计算启动时为 TLS 分配的总大小 $tlssize_S$。此大小等于 $tlsoffset_M$，加上 512 个字节。这额外的 512 个字节为静态 TLS 引用提供了一个备份预留空间。执行静态 TLS 引用的共享目标文件在进程初始化后将装入并指定到该备份预留空间。但是，该预留空间的大小是固定的、有限的。此外，该预留空间只能为未初始化的 TLS 数据项提供存储空间。为实现最大的灵活性，共享目标文件应使用动态的 TLS 模型引用线程局部变量。

与计算得出的 TLS 大小 $tlssize_S$ 关联的静态 TLS 块将紧排在进程指针 tp_t 之前放置。对 TLS 数据的访问基于 tp_t 减法。

静态 TLS 块与一份链接的初始化记录列表相关联。此列表中的每条记录都描述一个已装入动态目标文件的 TLS 初始化映像。每条记录都包含以下字段：

- 指向 TLS 初始化映像的指针。

- TLS 初始化映像的大小。
- 目标文件的 $tlsoffset_m$。
- 指示目标文件是否使用静态 TLS 模型的标志。

线程库使用此信息为初始线程分配存储空间。此存储空间初始化后，会为初始线程创建动态 TLS 向量。

创建线程

对于初始线程和所创建的每个新线程，线程库将为每个已装入的动态目标文件分配一个新的 TLS 块。各个块可以单独进行分配，也可以作为一个连续块进行分配。

每个线程 t 都有一个关联的线程指针 tp_t，该指针指向线程控制块 TCB。线程指针 tp 始终包含当前正在运行的线程的 tp_t 值。

然后，线程库为当前线程 t 创建一个指针向量 dtv_t。每个向量的第一个元素都包含一个生成号 gen_t，该生成编号用于确定需要扩展向量的时间。请参见“延迟分配线程局部存储块” [386]。

$dtv_{t,m}$ 向量中剩余的每个元素都是一个指针，指向为属于动态目标文件 m 的 TLS 保留的块。

对于启动后动态装入的目标文件，线程库将延迟分配 TLS 块。分配将在第一次引用已装入的目标文件中的 TLS 变量时进行。对于延迟分配的块，指针 $dtv_{t,m}$ 设置为实现定义的特殊值。

注 - 运行时链接程序可以将所有启动目标文件的 TLS 模板进行分组，以便在向量 $dtv_{t,1}$ 中共享单个元素。这种分组不会影响前面介绍的偏移计算，也不会影响初始化记录列表的创建。但是，对于以下各节，总目标文件数 M 的值从 1 开始。

然后，线程库将初始化映像复制到新存储块中的对应位置。

启动后动态装入

仅包含动态 TLS 的共享目标文件可以在进程启动后不受限制地装入。因此，运行时链接程序扩展初始化记录列表，以包含新目标文件的初始化模板。新目标文件将获得一个索引 m = M + 1。计数器 M 按 1 递增。但是，新的 TLS 块将延迟分配，直至实际引用了这些块时。

卸载仅包含动态 TLS 的共享目标文件时，将释放该库使用的 TLS 块。

包含静态 TLS 的共享目标文件可以在进程启动后有限制地装入。静态 TLS 引用只能通过任何剩余的备份 TLS 预留空间来满足。请参见“程序启动” [384]。此预留空间的大小是有限的。此外，此预留空间只能为未初始化的 TLS 数据项提供存储空间。

包含静态 TLS 的共享目标文件永远不会卸载。静态 TLS 的处理将导致共享目标文件被标记为不可删除。

延迟分配线程局部存储块

在动态 TLS 模型中，当线程 `t` 需要访问目标文件 `m` 的 TLS 块时，代码将更新 dtv_t 并执行 TLS 块的初始分配。线程库将提供以下接口以便动态分配 TLS。

```
typedef struct {
        unsigned long ti_moduleid;
        unsigned long ti_tlsoffset;
} TLS_index;

extern void *__tls_get_addr(TLS_index *ti);     (SPARC and x64)
extern void *___tls_get_addr(TLS_index *ti);    (32–bit x86)
```

注 - 此函数的 SPARC 和 64 位 x86 定义具有相同的函数签名。但是，32 位 x86 版本不使用缺省调用约定来传递栈中的参数。相反，32 位 x86 版本通过更有效的 `%eax` 寄存器来传递其参数。为了指示将使用此替代调用方法，32 位 x86 函数的名称中有三个前导下划线。

这两个版本的 `tls_get_addr`（） 都会检查每线程生成计数器 gen_t，以确定是否需要更新向量。如果 dtv_t 向量已过时，例程将更新此向量，可能会重新分配此向量以便为更多项留出空间。然后，例程将检查与 $dtv_{t,m}$ 对应的 TLS 块是否已分配。如果该向量尚未分配，例程将分配并初始化该块。例程使用运行时链接程序提供的初始化记录列表中的信息。$dtv_{t,m}$ 指针被设置为指向已分配的块。例程返回一个指向块中给定偏移的指针。

线程局部存储的访问模型

每个 TLS 引用都遵循下列访问模型之一。这些模型按照最常见、但最少优化到速度最快、但限制最大的顺序列出。

常规动态 *(General Dynamic, GD)* – 动态 TLS

此模型允许从共享目标文件或动态可执行文件中引用所有 TLS 变量。如果是第一次从特定线程引用 TLS 块，此模型还支持延迟分配此块。

局部动态 *(Local Dynamic, LD)* – 局部符号的动态 TLS

此模型是对 *GD* 模型的优化。编译器可能会确定变量在要生成的目标文件中是局部绑定或受到保护的。在这种情况下，编译器将指示链接编辑器静态绑定动态的 `tlsoffset` 并使用此模型。与 *GD* 模型相比，此模型可提供更好的性能。每个函数只需要调用一次 `tls_get_addr`（） 即可确定 $dtv_{0,m}$ 的地址。进行链接编辑时绑定的动态 TLS 偏移会与每个引用的 $dtv_{0,m}$ 地址相加。

初始可执行 *(Initial Executable, IE)* – 具有指定偏移的静态 TLS

此模型只能引用初始静态 TLS 中包含的 TLS 变量。此模板由进程启动时可用的所有 TLS 块和一个小的备份预留空间组成。请参见“程序启动” [384]。在此模型中，给定变量 *x* 相对于线程指针的偏移存储在 *x* 的 `GOT` 项中。

此模型可以从初始进程启动后通过延迟装入、过滤器或 `dlopen(3C)` 装入的共享库中引用有限数量的 TLS 变量。该访问可通过固定的备份预留空间来实现。此预留空间只能为未初始化的 TLS 数据项提供存储空间。为实现最大的灵活性，共享目标文件应使用动态的 TLS 模型引用线程局部变量。

注 - 可以使用过滤器动态选择所使用的静态 TLS。共享目标文件可以生成为使用动态 TLS，并在生成使用静态 TLS 的对应共享目标文件时，充当辅助过滤器。如果资源允许装入静态 TLS 目标文件，将使用该库。否则，将回退到动态 TLS 目标文件，以确保共享目标文件提供的功能总是可用。有关过滤器的更多信息，请参见“作为过滤器的共享目标文件” [127]。

局部可执行 *(Local Executable, LE)* – 静态 TLS

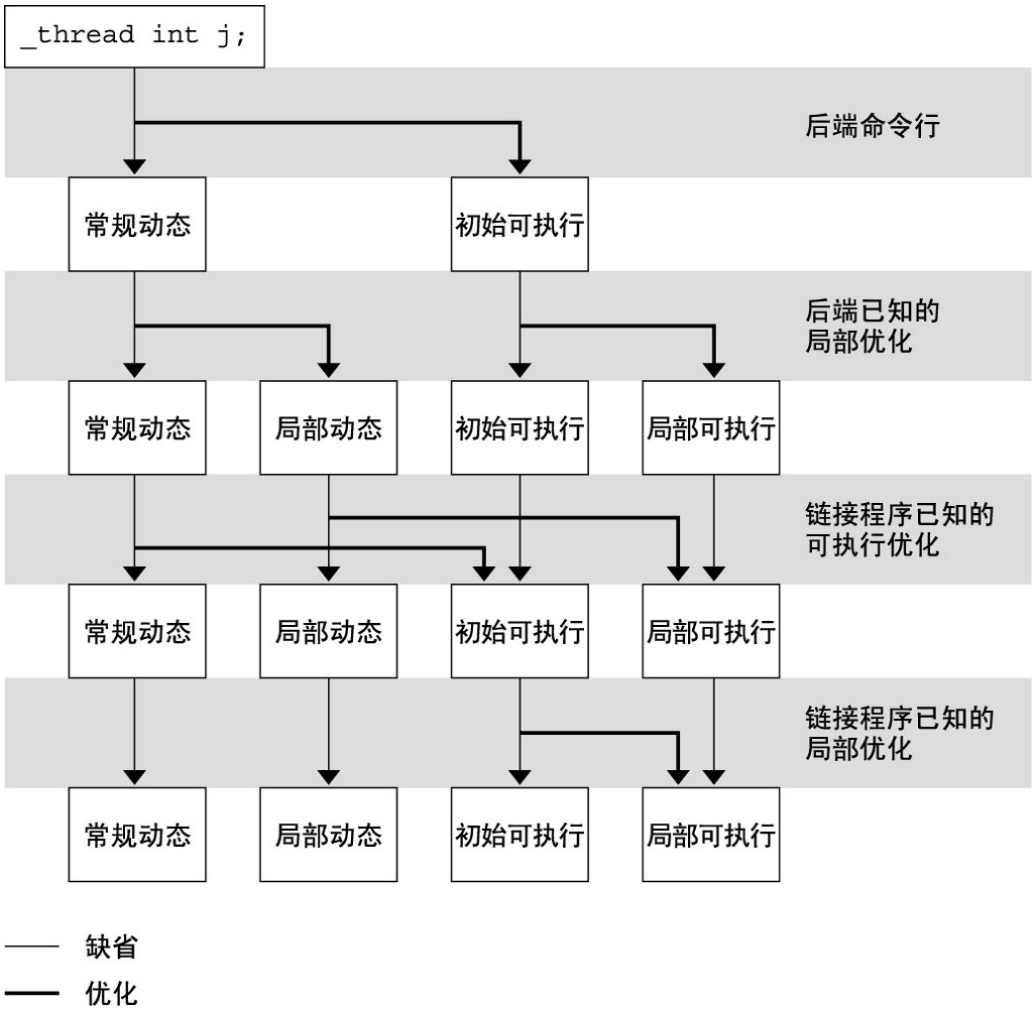
此模型只能引用动态可执行文件的 TLS 块中包含的 TLS 变量。链接编辑器静态地计算相对于线程指针的偏移，而不需要进行动态重定位或额外引用 `GOT`。此模型不能用于引用动态可执行文件外部的变量。

链接编辑器可以将代码从更常规的访问模型转换为更优化的模型（如果确定适合进行转换）。这种转换可以使用独特的 TLS 重定位来实现。这些重定位不仅请求执行更新，还会标识要使用的 TLS 访问模型。

链接编辑器在了解 TLS 访问模型和要创建的目标文件类型后，便可执行转换。例如，如果一个可重定位目标文件使用 *GD* 访问模型，被链接到一个动态可执行文件中。在这种情况下，链接编辑器可以适当地使用 *IE* 或 *LE* 访问模型转换引用。然后执行模型所需的重定位。

下图说明了不同的访问模型，以及从一个模型到另一个模型的转换。

图 14-2 线程局部存储的访问模型和转换



SPARC: 线程局部变量访问

在 SPARC 上，可使用以下代码序列模型访问线程局部变量。

SPARC: 常规动态 (General Dynamic, GD)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 GD 模型。

表 14-2 SPARC: 常规动态的线程局部变量访问代码

代码序列	初始重定位	符号
# %l7 - initialized to GOT pointer		
		x
0x00 sethi %hi(@dtlndx(x)), %o0	R_SPARC_TLS_GD_HI22	x
0x04 add %o0, %lo(@dtlndx(x)), %o0	R_SPARC_TLS_GD_LO10	x
0x08 add %l7, %o0, %o0	R_SPARC_TLS_GD_ADD	x
0x0c call x@TLSPLT	R_SPARC_TLS_GD_CALL	
# %o0 - contains address of TLS variable		
	未完成的重定位：32 位	**符号**
GOT[n]	R_SPARC_TLS_DTPMOD32	x
GOT[n + 1]	R_SPARC_TLS_DTPOFF32	x
	未完成的重定位：64 位	**符号**
GOT[n]	R_SPARC_TLS_DTPMOD64	x
GOT[n + 1]	R_SPARC_TLS_DTPOFF64	x

sethi 和 add 指令分别生成 R_SPARC_TLS_GD_HI22 和 R_SPARC_TLS_GD_LO10 重定位。这些重定位将指示链接编辑器在 GOT 中分配空间，以存储变量 x 的 TLS_index 结构。链接编辑器通过以相对于 GOT 的偏移替代新的 GOT 项来处理此重定位。

x 的装入目标文件索引和 TLS 块索引在运行前无法确定。因此，链接编辑器将根据 GOT 来放置 R_SPARC_TLS_DTPMOD32 和 R_SPARC_TLS_DPTOFF32 重定位，以供运行时链接程序处理。

第二个 add 指令将生成 R_SPARC_TLS_GD_ADD 重定位。仅当链接编辑器将 GD 代码序列更改为另一个序列时，才会使用此重定位。

call 指令使用特殊的语法 x@TLSPLT。此调用引用 TLS 变量，并生成 R_SPARC_TLS_GD_CALL 重定位。此重定位指示链接编辑器将调用绑定到 __tls_get_addr () 函数，并将 call 指令与 GD 代码序列关联。

注 - add 指令必须出现在 call 指令前面。add 指令不能放在调用的延迟槽中。由于后面可能发生的代码变换需要已知顺序，所以必须满足此要求。

用作 add 指令（由 R_SPARC_TLS_GD_ADD 重定位标记）的 GOT 指针的寄存器必须是 add 指令中的第一个寄存器。在代码变换期间，此要求允许链接编辑器标识 GOT 指针寄存器。

SPARC: 局部动态 (Local Dynamic, LD)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 LD 模型。

表 14-3 SPARC: 局部动态的线程局部变量访问代码

代码序列	初始重定位	符号
# %l7 - initialized to GOT pointer		
0x00 sethi %hi(@tmndx(x1)), %o0	R_SPARC_TLS_LDM_HI22	x1
0x04 add %o0, %lo(@tmndx(x1)), %o0	R_SPARC_TLS_LDM_LO10	x1
0x08 add %l7, %o0, %o0	R_SPARC_TLS_LDM_ADD	x1
0x0c call x@TLSPLT	R_SPARC_TLS_LDM_CALL	x1
# %o0 - contains address of TLS block of current object		
	R_SPARC_TLS_LDO_HIX22	x1
0x10 sethi %hi(@dtpoff(x1)), %l1	R_SPARC_TLS_LDO_LOX10	x1
0x14 xor %l1, %lo(@dtpoff(x1)), %l1	R_SPARC_TLS_LDO_ADD	x1
0x18 add %o0, %l1, %l1		
# %l1 - contains address of local TLS variable x1		
	R_SPARC_TLS_LDO_HIX22	x2
0x20 sethi %hi(@dtpoff(x2)), %l2	R_SPARC_TLS_LDO_LOX10	x2
0x24 xor %l2, %lo(@dtpoff(x2)), %l2	R_SPARC_TLS_LDO_ADD	x2
0x28 add %o0, %l2, %l2		
# %l2 - contains address of local TLS variable x2		
	未完成的重定位：32 位	符号
GOT[n]	R_SPARC_TLS_DTPMOD32	x1
GOT[n + 1]	<none>	
	未完成的重定位：64 位	符号
GOT[n]	R_SPARC_TLS_DTPMOD64	x1
GOT[n + 1]	<none>	

第一个 sethi 指令和 add 指令分别生成 R_SPARC_TLS_LDM_HI22 和 R_SPARC_TLS_LDM_LO10 重定位。这些重定位指示链接编辑器在 GOT 中分配空间，以存储当前目标文件的 TLS_index 结构。链接编辑器通过以相对于 GOT 的偏移替代新的 GOT 项来处理此重定位。

装入目标文件索引在运行前无法确定。因此，将创建 R_SPARC_TLS_DTPMOD32 重定位，并在 TLS_index 结构的 ti_tlsoffset 字段中填充零。

第二个 add 和 call 指令分别使用 R_SPARC_TLS_LDM_ADD 和 R_SPARC_TLS_LDM_CALL 重定位进行标记。

后面的 sethi 指令和 xor 指令分别生成 R_SPARC_LDO_HIX22 和 R_SPARC_TLS_LDO_LOX10 重定位。在链接编辑时已得知了每个局部符号的 TLS 偏移，因此将直接填入这些值。add 指令使用 R_SPARC_TLS_LDO_ADD 重定位进行标记。

当一个过程引用多个局部符号时，编译器将生成代码一次获取 TLS 块的基本地址。然后，使用此基本地址计算每个符号的地址，而不需要单独调用库。

注 - 包含 add 指令（由 R_SPARC_TLS_LDO_ADD 标记）中的 TLS 目标文件地址的寄存器必须是指令序列中的第一个寄存器。在代码变换期间，此要求允许链接编辑器标识寄存器。

---

32 位 SPARC: 初始可执行 (Initial Executable, IE)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 IE 模型。

表 14-4　32 位 SPARC: 初始可执行的线程局部变量访问代码

代码序列	初始重定位	符号
# %l7 - initialized to GOT pointer, %g7 - thread pointer		
	R_SPARC_TLS_IE_HI22	x
0x00 sethi %hi(@tpoff(x)), %o0	R_SPARC_TLS_IE_LO10	x
0x04 or %o0, %lo(@tpoff(x)), %o0	R_SPARC_TLS_IE_LD	x
0x08 ld [%l7 + %o0], %o0	R_SPARC_TLS_IE_ADD	x
0x0c add %g7, %o0, %o0		
# %o0 - contains address of TLS variable		
	未完成的重定位	**符号**
GOT[n]	R_SPARC_TLS_TPOFF32	x

sethi 指令和 or 指令分别生成 R_SPARC_TLS_IE_HI22 和 R_SPARC_TLS_IE_LO10 重定位。这些重定位指示链接编辑器在 GOT 中创建空间，以存储符号 x 的静态 TLS 偏移。针对 GOT 的 R_SPARC_TLS_TPOFF32 重定位将会暂停，以便运行时链接程序使用符号 x 的负静态 TLS 偏移填充。ld 和 add 指令分别使用 R_SPARC_TLS_IE_LD 和 R_SPARC_TLS_IE_ADD 重定位进行标记。

注 - 用作 add 指令（由 R_SPARC_TLS_IE_ADD 重定位标记）的 GOT 指针的寄存器必须是此指令中的第一个寄存器。在代码变换期间，此要求允许链接编辑器标识 GOT 指针寄存器。

---

64 位 SPARC: 初始可执行 (Initial Executable, IE)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 IE 模型。

表 14-5 64 位 SPARC: 初始可执行的线程局部变量访问代码

代码序列	初始重定位	符号
`# %l7 - initialized to GOT pointer, %g7 - thread pointer`		
	R_SPARC_TLS_IE_HI22	x
`0x00 sethi %hi(@tpoff(x)), %o0`	R_SPARC_TLS_IE_LO10	x
`0x04 or %o0, %lo(@tpoff(x)), %o0`	R_SPARC_TLS_IE_LD	x
`0x08 ldx [%l7 + %o0], %o0`	R_SPARC_TLS_IE_ADD	x
`0x0c add %g7, %o0, %o0`		
`# %o0 - contains address of TLS variable`		
	未完成的重定位	**符号**
GOT[n]	R_SPARC_TLS_TPOFF64	x

SPARC: 局部可执行 (Local Executable, LE)

此代码序列实现"线程局部存储的访问模型" [386]中介绍的 LE 模型。

表 14-6 SPARC: 局部可执行的线程局部变量访问代码

代码序列	初始重定位	符号
`# %g7 - thread pointer`		
`0x00 sethi %hix(@tpoff(x)), %o0`	R_SPARC_TLS_LE_HIX22	x
`0x04 xor %o0,%lo(@tpoff(x)),%o0`	R_SPARC_TLS_LE_LOX10	x
`0x08 add %g7, %o0, %o0`	<none>	
`# %o0 - contains address of TLS variable`		

sethi 和 xor 指令分别生成 R_SPARC_TLS_LE_HIX22 和 R_SPARC_TLS_LE_LOX10 重定位。链接编辑器将这些重定位直接绑定到在可执行文件中定义的符号的静态 TLS 偏移。在运行时不需要进行重定位处理。

SPARC: 线程局部存储的重定位类型

下表中列出的 TLS 重定位是针对 SPARC 定义的。表中的说明使用以下表示法。

@dtlndx(x)

在 GOT 中分配两个连续项，以存储 TLS_index 结构。此信息将传递给 __tls_get_addr（）。引用此项的指令将绑定到两个 GOT 项中第一项的地址。

@tmndx(x)

在 GOT 中分配两个连续项，以存储 TLS_index 结构。此信息将传递给 __tls_get_addr（）。此结构的 ti_tlsoffset 字段设置为 0，并且在运行时填充 ti_moduleid。对 __tls_get_addr（） 的调用将返回动态 TLS 块的起始偏移。

@dtpoff(x)

计算相对于 TLS 块的 tlsoffset。

@tpoff(x)

计算相对于静态 TLS 块的负 tlsoffset。此值与线程指针相加以计算 TLS 地址。

@dtpmod(x)

计算包含 TLS 符号的目标文件的标识符。

表 14-7 SPARC: 线程局部存储的重定位类型

名称	值	字段	计算
R_SPARC_TLS_GD_HI22	56	T-simm22	@dtlndx(S + A) >> 10
R_SPARC_TLS_GD_LO10	57	T-simm13	@dtlndx(S + A) & 0x3ff
R_SPARC_TLS_GD_ADD	58	无	请参阅此表后面的说明。
R_SPARC_TLS_GD_CALL	59	V-disp30	请参阅此表后面的说明。
R_SPARC_TLS_LDM_HI22	60	T-simm22	@tmndx(S + A) >> 10
R_SPARC_TLS_LDM_LO10	61	T-simm13	@tmndx(S + A) & 0x3ff
R_SPARC_TLS_LDM_ADD	62	无	请参阅此表后面的说明。
R_SPARC_TLS_LDM_CALL	63	V-disp30	请参阅此表后面的说明。
R_SPARC_TLS_LDO_HIX22	64	T-simm22	@dtpoff(S + A) >> 10
R_SPARC_TLS_LDO_LOX10	65	T-simm13	@dtpoff(S + A) & 0x3ff
R_SPARC_TLS_LDO_ADD	66	无	请参阅此表后面的说明。
R_SPARC_TLS_IE_HI22	67	T-simm22	@got(@tpoff(S + A)) >> 10
R_SPARC_TLS_IE_LO10	68	T-simm13	@got(@tpoff(S + A)) & 0x3ff
R_SPARC_TLS_IE_LD	69	无	请参阅此表后面的说明。
R_SPARC_TLS_IE_LDX	70	无	请参阅此表后面的说明。
R_SPARC_TLS_IE_ADD	71	无	请参阅此表后面的说明。
R_SPARC_TLS_LE_HIX22	72	T-imm22	(@tpoff(S + A) ^0xffffffffffffffff) >> 10
R_SPARC_TLS_LE_LOX10	73	T-simm13	(@tpoff(S + A) & 0x3ff) | 0x1c00
R_SPARC_TLS_DTPMOD32	74	V-word32	@dtpmod(S + A)
R_SPARC_TLS_DTPMOD64	75	V-word64	@dtpmod(S + A)
R_SPARC_TLS_DTPOFF32	76	V-word32	@dtpoff(S + A)
R_SPARC_TLS_DTPOFF64	77	V-word64	@dtpoff(S + A)
R_SPARC_TLS_TPOFF32	78	V-word32	@tpoff(S + A)
R_SPARC_TLS_TPOFF64	79	V-word64	@tpoff(S + A)

一些重定位类型的语义不只是简单的计算。

R_SPARC_TLS_GD_ADD

此重定位标记 GD 代码序列的 add 指令。用于 GOT 指针的寄存器是该序列中的第一个寄存器。此重定位所标记的指令出现在 R_SPARC_TLS_GD_CALL 重定位所标记的 call 指令之前。此重定位用于在链接编辑时在 TLS 模型之间进行转换。

R_SPARC_TLS_GD_CALL

此重定位将按引用 __tls_get_addr() 函数的 R_SPARC_WPLT30 重定位的处理方式进行处理。此重定位是 GD 代码序列的一部分。

R_SPARC_LDM_ADD

此重定位标记 LD 代码序列的第一个 add 指令。用于 GOT 指针的寄存器是该序列中的第一个寄存器。此重定位所标记的指令出现在 R_SPARC_TLS_GD_CALL 重定位所标记的 call 指令之前。此重定位用于在链接编辑时在 TLS 模型之间进行转换。

R_SPARC_LDM_CALL

此重定位将按引用 __tls_get_addr() 函数的 R_SPARC_WPLT30 重定位的处理方式进行处理。此重定位是 LD 代码序列的一部分。

R_SPARC_LDO_ADD

此重定位标记 LD 代码序列中的最后一个 add 指令。包含目标文件地址（在代码序列的初始部分中计算得出）的寄存器是此指令中的第一个寄存器。此重定位允许链接编辑器标识此寄存器以进行代码变换。

R_SPARC_TLS_IE_LD

此重定位标记 32 位 IE 代码序列中的 ld 指令。此重定位用于在链接编辑时在 TLS 模型之间进行转换。

R_SPARC_TLS_IE_LDX

此重定位标记 64 位 IE 代码序列中的 ldx 指令。此重定位用于在链接编辑时在 TLS 模型之间进行转换。

R_SPARC_TLS_IE_ADD

此重定位标记 IE 代码序列中的 add 指令。用于 GOT 指针的寄存器是此序列中的第一个寄存器。

32 位 x86: 线程局部变量访问

在 x86 上，可使用以下代码序列模型访问 TLS。

32 位 x86: 常规动态 (General Dynamic, GD)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 GD 模型。

表 14-8 32 位 x86: 常规动态的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 leal x@tlsgd(,%ebx,1), %eax 0x07 call x@tlsgdplt # %eax - contains address of TLS variable	R_386_TLS_GD R_386_TLS_GD_PLT	x x
	未完成的重定位	**符号**
GOT[n] GOT[n + 1]	R_386_TLS_DTPMOD32 R_386_TLS_DTPOFF32	x

leal 指令生成 R_386_TLS_GD 重定位，此重定位指示链接编辑器在 GOT 中分配空间，以存储变量 x 的 TLS_index 结构。链接编辑器通过以相对于 GOT 的偏移替代新的 GOT 项来处理此重定位。

由于在运行前无法确定 x 的装入目标文件索引和 TLS 块索引，因此链接编辑器将根据 GOT 来放置 R_386_TLS_DTPMOD32 和 R_386_TLS_DTPOFF32 重定位，以供运行时链接程序处理。生成的 GOT 项的地址将装入寄存器 %eax 中，以便调用 ___tls_get_addr（）。

call 指令将导致生成 R_386_TLS_GD_PLT 重定位。此重定位指示链接编辑器将调用绑定到 ___tls_get_addr （） 函数，并将 call 指令与 GD 代码序列关联。

call 指令必须紧跟在 leal 指令后面。要允许进行代码变换，必须满足此要求。

x86: 局部动态 (Local Dynamic, LD)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 LD 模型。

表 14-9 32 位 x86: 局部动态的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 leal x1@tlsldm(%ebx), %eax 0x06 call x1@tlsldmplt # %eax - contains address of TLS block of current object 0x10 leal x1@dtpoff(%eax), %edx # %edx - contains address of local TLS variable x1 0x20 leal x2@dtpoff(%eax), %edx	R_386_TLS_LDM R_386_TLS_LDM_PLT R_386_TLS_LDO_32 R_386_TLS_LDO_32	x1 x1 x1 x2

# %edx - contains address of local TLS variable x2		
	未完成的重定位	符号
GOT[n]	R_386_TLS_DTPMOD32	x
GOT[n + 1]	<none>	

第一个 leal 指令生成 R_386_TLS_LDM 重定位。此重定位指示链接编辑器在 GOT 中分配空间，以存储当前目标文件的 TLS_index 结构。链接编辑器通过以相对于 GOT 的偏移替代新的链接表项来处理此重定位。

装入目标文件索引在运行前无法确定。因此，将创建 R_386_TLS_DTPMOD32 重定位，并在该结构的 ti_tlsoffset 字段中填充零。call 指令使用 R_386_TLS_LDM_PLT 重定位进行标记。

在链接编辑时已得知了每个局部符号的 TLS 偏移，因此链接编辑器将直接填入这些值。

当一个过程引用多个局部符号时，编译器将生成代码一次获取 TLS 块的基本地址。然后，使用此基本地址计算每个符号的地址，而不需要单独调用库。

32 位 x86: 初始可执行 (Initial Executable, IE)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 IE 模型。

IE 模型存在两个代码序列。一个序列针对使用 GOT 指针的位置无关代码。另一个序列针对不使用 GOT 指针的位置相关代码。

表 14-10　32 位 x86: 初始可执行的、位置无关的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 movl %gs:0, %eax	<none>	
0x06 addl x@gotntpoff(%ebx), %eax	R_386_TLS_GOTIE	x
# %eax - contains address of TLS variable		
	未完成的重定位	符号
GOT[n]	R_386_TLS_TPOFF	x

addl 指令生成 R_386_TLS_GOTIE 重定位。此重定位指示链接编辑器在 GOT 中创建空间，以存储符号 x 的静态 TLS 偏移。针对 GOT 表的 R_386_TLS_TPOFF 重定位未完成，以便运行时链接程序使用符号 x 的静态 TLS 偏移填充。

表 14-11　32 位 x86: 初始可执行的、位置相关的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 movl %gs:0, %eax	<none>	
0x06 addl x@indntpoff, %eax	R_386_TLS_IE	x

# %eax - contains address of TLS variable		
	未完成的重定位	符号
GOT[n]	R_386_TLS_TPOFF	x

addl 指令生成 R_386_TLS_IE 重定位。此重定位指示链接编辑器在 GOT 中创建空间，以存储符号 x 的静态 TLS 偏移。此序列和位置无关序列之间的主要差别在于，指令直接绑定到所创建的 GOT 项，而不使用 GOT 指针寄存器的偏移。针对 GOT 的 R_386_TLS_TPOFF 重定位未完成，以便运行时链接程序使用符号 x 的静态 TLS 偏移填充。

如下面两个序列中所示，通过将偏移直接嵌入到内存引用中，可以装入变量 x 的内容（而不是地址）。

表 14-12　32 位 x86: 初始可执行的、位置无关的动态线程局部变量访问代码

代码序列	初始重定位	符号
0x00 movl x@gotntpoff(%ebx), %eax 0x06 movl %gs:(%eax), %eax # %eax - contains address of TLS variable	R_386_TLS_GOTIE <none>	x
	未完成的重定位	符号
GOT[n]	R_386_TLS_TPOFF	x

表 14-13　32 位 x86: 初始可执行的、位置无关的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 movl x@indntpoff, %ecx 0x06 movl %gs:(%ecx), %eax # %eax - contains address of TLS variable	R_386_TLS_IE <none>	x
	未完成的重定位	符号
GOT[n]	R_386_TLS_TPOFF	x

在最后一个序列中，如果使用寄存器 %eax 而非寄存器 %ecx，第一个指令的长度将可以为 5 或 6 个字节。

32 位 x86: 局部可执行 (Local Executable, LE)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 LE 模型。

表 14-14　32 位 x86: 局部可执行的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 movl %gs:0, %eax	<none>	

```
0x06 leal  x@ntpoff(%eax), %eax                    R_386_TLS_LE                    x

# %eax - contains address of TLS variable
```

movl 指令生成 R_386_TLS_LE_32 重定位。链接编辑器将此重定位直接绑定到在可执行文件中定义的符号的静态 TLS 偏移。在运行时不需要进行任何处理。

通过使用以下指令序列，可以借助同一重定位访问变量 x 的内容（而非地址）。

表 14-15　32 位 x86: 局部可执行的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 movl %gs:0, %eax	<none>	
0x06 movl x@ntpoff(%eax), %eax	R_386_TLS_LE	x
# %eax - contains address of TLS variable		

使用以下序列可以实现从变量装入或存储到变量，而不必计算变量的地址。请注意，x@ntpoff 表达式不能用作立即值，而应用作绝对地址。

表 14-16　32 位 x86: 局部可执行的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 movl %gs:x@ntpoff, %eax	R_386_TLS_LE	x
# %eax - contains address of TLS variable		

32 位 x86: 线程局部存储的重定位类型

下表中列出的 TLS 重定位是针对 x86 定义的。表中的说明使用以下表示法。

@tlsgd(x)

在 GOT 中分配两个连续项，以存储 TLS_index 结构。此结构将传递给 ___tls_get_addr（）。引用此项的指令将绑定到两个 GOT 项中的第一项。

@tlsgdplt(x)

此重定位将按引用 ___tls_get_addr（） 函数的 R_386_PLT32 重定位的处理方式进行处理。

@tlsldm(x)

在 GOT 中分配两个连续项，以存储 TLS_index 结构。此结构将传递给 ___tls_get_addr（）。TLS_index 的 ti_tlsoffset 字段将设置为 0，并且在运行时填充 ti_moduleid。对 ___tls_get_addr（） 的调用将返回动态 TLS 块的起始偏移。

@gotntpoff(x)

在 GOT 中分配一项，并使用相对于静态 TLS 块的负 tlsoffset 初始化该项。运行时将使用 R_386_TLS_TPOFF 重定位执行此序列。

@indntpoff(x)

此表达式类似于 @gotntpoff，但它用在位置相关代码中。在 movl 或 addl 指令中，@gotntpoff 将解析为相对于 GOT 起始位置的 GOT 插槽地址。@indntpoff 将解析为绝对 GOT 插槽地址。

@ntpoff(x)

计算相对于静态 TLS 块的负 tlsoffset。

@dtpoff(x)

计算相对于 TLS 块的 tlsoffset。此值用作加数的立即值，并且不与特定寄存器关联。

@dtpmod(x)

计算包含 TLS 符号的目标文件的标识符。

表 14-17 32 位 x86: 线程局部存储的重定位类型

名称	值	字段	计算
R_386_TLS_GD_PLT	12	Word32	@tlsgdplt
R_386_TLS_LDM_PLT	13	Word32	@tlsldmplt
R_386_TLS_TPOFF	14	Word32	@ntpoff(S)
R_386_TLS_IE	15	Word32	@indntpoff(S)
R_386_TLS_GOTIE	16	Word32	@gotntpoff(S)
R_386_TLS_LE	17	Word32	@ntpoff(S)
R_386_TLS_GD	18	Word32	@tlsgd(S)
R_386_TLS_LDM	19	Word32	@tlsldm(S)
R_386_TLS_LDO_32	32	Word32	@dtpoff(S)
R_386_TLS_DTPMOD32	35	Word32	@dtpmod(S)
R_386_TLS_DTPOFF32	36	Word32	@dtpoff(S)

x64: 线程局部变量访问

在 x64 上，可使用以下代码序列模型访问 TLS。

x64: 常规动态 (General Dynamic, GD)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 GD 模型。

表 14-18 x64: 常规动态的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 .byte 0x66	<none>	
0x01 leaq x@tlsgd(%rip), %rdi	R_AMD64_TLSGD	x
0x08 .word 0x6666	<none>	
0x0a rex64	<none>	
0x0b call __tls_get_addr@plt	R_AMD64_PLT32	__tls_get_addr
# %rax - contains address of TLS variable		
	未完成的重定位	**符号**
GOT[n]	R_AMD64_DTPMOD64	x
GOT[n + 1]	R_AMD64_DTPOFF64	x

__tls_get_addr（） 函数采用单个参数，即 tls_index 结构的地址。与 x@tlsgd(%rip) 表达式关联的 R_AMD64_TLSGD 重定位指示链接编辑器在 GOT 中分配 tls_index 结构。tls_index 结构所需的两个元素将保留在连续的 GOT 项中（GOT[n] 和 GOT[n+1]）。这些 GOT 项与 R_AMD64_DTPMOD64 和 R_AMD64_DTPOFF64 重定位相关联。

地址 0x00 处的指令计算第一个 GOT 项的地址。通过将 GOT 起始位置的 PC 相对地址（在链接编辑时得知）与当前指令指针相加进行此计算。结果将通过 %rdi 寄存器传递给 __tls_get_addr（） 函数。

注 - leaq 指令计算第一个 GOT 项的地址。通过将 GOT 的 PC 相对地址（在链接编辑时得知）与当前指令指针相加进行此计算。.byte、.word、和 .rex64 前缀可确保整个指令序列占用 16 个字节。由于前缀不会对代码造成负面影响，因此可以使用前缀。

---

x64: 局部动态 (Local Dynamic, LD)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 LD 模型。

表 14-19 x64: 局部动态的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 leaq x1@tlsld(%rip), %rdi	R_AMD64_TLSLD	x1
0x07 call __tls_get_addr@plt	R_AMD64_PLT32	__tls_get_addr
# %rax - contains address of TLS block		
0x10 leaq x1@dtpoff(%rax), %rcx	R_AMD64_DTOFF32	x1
# %rcx - contains address of TLS variable x1		
0x20 leaq x2@dtpoff(%rax), %r9	R_AMD64_DTOFF32	x2
# %r9 - contains address of TLS variable x2		

	未完成的重定位	符号
GOT[n]	R_AMD64_DTMOD64	x1

前两个指令与用于常规动态模型的代码序列等效，虽然不进行任何填充。这两个指令必须是连续的。x1@tlsld(%rip) 序列为符号 x1 生成 tls_index 项。此索引引用包含 x1（偏移为零）的当前模块。链接编辑器为目标文件 R_AMD64_DTMOD64 创建一个重定位。

因为各个偏移会分别单独装入，所以 R_AMD64_DTOFF32 重定位是没必要的。x1@dtpoff 表达式用于访问符号 x1 的偏移。将指令用作 0x10 地址时，可装入完整的偏移并将该偏移与 %rax 中的 __tls_get_addr（） 调用的结果相加，以在 %rcx 中生成结果。x1@dtpoff 表达式创建 R_AMD64_DTPOFF32 重定位。

可以使用以下指令装入变量的值，而不必计算变量的地址。此指令与原始 leaq 指令创建的重定位相同。

```
movq  x1@dtpoff(%rax), %r11
```

如果 TLS 块的基本地址保存在一个寄存器中，则装入、存储或计算受保护的线程局部变量的地址将只需要一个指令。

使用局部动态模型比使用常规动态模型可获得更多好处。每进行一次额外的线程局部变量访问，只需执行三个新指令。此外，不需要生成其他 GOT 项或运行时重定位。

x64: 初始可执行 (Initial Executable, IE)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 IE 模型。

表 14-20　　x64: 初始可执行的线程局部变量访问代码

代码序列	初始重定位	符号
0x00 movq %fs:0, %rax 0x09 addq x@gottpoff(%rip), %rax # %rax - contains address of TLS variable	<none> R_AMD64_GOTTPOFF	 x
	未完成的重定位	符号
GOT[n]	R_AMD64_TPOFF64	x

符号 x 的 R_AMD64_GOTTPOFF 重定位请求链接编辑器生成 GOT 项和关联的 R_AMD64_TPOFF64 重定位。然后，x@gottpoff(%rip) 指令使用 GOT 项相对于此指令结束位置的偏移。R_AMD64_TPOFF64 重定位使用由目前已装入模块确定的符号 x 值。偏移将写入到 GOT 项中，然后由 addq 指令装入。

要装入 x 的内容（而非 x 的地址），可使用以下序列。

表 14-21 x64: 初始可执行的线程局部变量访问代码 II

代码序列	初始重定位	符号

`0x00 movq x@gottpoff(%rip), %rax`
`0x07 movq %fs:(%rax), %rax`
 	
`# %rax - contains contents of TLS variable`
	`R_AMD64_GOTTPOFF`
	`<none>`
		x

	未完成的重定位	符号
`GOT[n]`	`R_AMD64_TPOFF64`	x

x64: 局部可执行 (Local Executable, LE)

此代码序列实现“线程局部存储的访问模型” [386]中介绍的 LE 模型。

表 14-22 x64: 局部可执行的线程局部变量访问代码

代码序列	初始重定位	符号

`0x00 movq %fs:0, %rax`
`0x09 leaq x@tpoff(%rax), %rax`
 	
`# %rax - contains address of TLS variable`
	`<none>`
	`R_AMD64_TPOFF32`
		x

要装入 TLS 变量的内容（而非 TLS 变量的地址），可使用以下序列。

表 14-23 x64: 局部可执行的线程局部变量的访问代码 II

代码序列	初始重定位	符号

`0x00 movq %fs:0, %rax`
`0x09 movq x@tpoff(%rax), %rax`
 	
`# %rax - contains contents of TLS variable`
	`<none>`
	`R_AMD64_TPOFF32`
		x

以下序列更为简短。

表 14-24 x64: 局部可执行的线程局部变量访问代码 III

代码序列	初始重定位	符号

`0x00 movq %fs:x@tpoff, %rax`
 	
`# %rax - contains contents of TLS variable`
	`R_AMD64_TPOFF32`	x

x64: 线程局部存储的重定位类型

下表中列出的 TLS 重定位是针对 x64 定义的。表中的说明使用以下表示法。

@tlsgd(%rip)

在 GOT 中分配两个连续项，以存储 TLS_index 结构。此结构将传递给 __tls_get_addr（）。此指令只能在确切的常规动态代码序列中。

@tlsld(%rip)

在 GOT 中分配两个连续项，以存储 TLS_index 结构。此结构将传递给 __tls_get_addr（）。在运行时，目标文件的 ti_offset 偏移字段将设置为零，并且 ti_module 偏移将初始化。对 __tls_get_addr （） 函数的调用将返回动态 TLS 块的起始偏移。此指令可以用在确切的代码序列中。

@dtpoff

计算变量相对于包含该变量的 TLS 块的起始位置的偏移。计算所得的值将用作加数的立即值，并且不与特定寄存器关联。

@dtpmod(x)

计算包含 TLS 符号的目标文件的标识符。

@gottpoff(%rip)

在 GOT 中分配一项，以将变量偏移保存在初始 TLS 块中。此偏移相对于 TLS 块的结束位置 %fs:0。运算符只能与 movq 或 addq 指令一起使用。

@tpoff(x)

计算变量相对于 TLS 块结束位置 %fs:0 的偏移。不会创建任何 GOT 项。

表 14-25 x64: 线程局部存储的重定位类型

名称	值	字段	计算
R_AMD64_DPTMOD64	16	Word64	@dtpmod(s)
R_AMD64_DTPOFF64	17	Word64	@dtpoff(s)
R_AMD64_TPOFF64	18	Word64	@tpoff(s)
R_AMD64_TLSGD	19	Word32	@tlsgd(s)
R_AMD64_TLSLD	20	Word32	@tlsld(s)
R_AMD64_DTPOFF32	21	Word32	@dtpoff(s)
R_AMD64_GOTTPOFF	22	Word32	@gottpoff(s)
R_AMD64_TPOFF32	23	Word32	@gottpoff(s)

部分 V

附录

♦ ♦ ♦ 附 录 A

链接程序和库的更新及新增功能

本附录概述了已添加到 Oracle Solaris OS 各个发行版中的更新和新增功能。

Oracle Solaris 11.2 发行版

- 链接编辑器可以对调试节进行解压缩和压缩。请参见“压缩调试节” [78]和“节压缩” [295]。
- 可提高运行时审计程序之间的同步性，并通过 `la_callinit（）` 和 `la_callentry（）` 函数实现进程初始化。请参见“审计接口函数” [247]和“审计接口控制流量” [252]。
- `-z relax` 选项可用于放宽链接编辑器的缺省有效性检查项。使用此选项可以创建本会被拒绝的输出目标文件。`-z relax` 选项取代了 `-t` 和 `-z relaxreloc` 选项。请参见 `ld(1)`)。
- 链接编辑器选项 `-z type` 以及额外的 `LD_UNSET`、`LD_{object-type}_OPTIONS` 和 `LD_{object-type}_UNSET` 环境变量提高了选项处理的灵活性。请参见“指定链接编辑器选项” [25]。
- `elfdump(1)` 的新增选项 `-F` 提供了输出格式选项。
- 桩目标文件可以省略在关联的实际目标文件中发现的符号。此技术可以阻止在新代码开发中使用链接到桩目标文件的符号，同时在实际目标文件中维护这些符号以实现向后兼容性。请参见“使用桩目标文件隐藏过时的接口” [73]。

Oracle Solaris 11.1 发行版

- 辅助目标文件允许将运行时不需要的调试节写入单独的目标文件中。请参见“辅助目标文件” [74]。
- 父目标文件通过允许插件直接链接其父目标文件简化了插件目标文件的构造。请参见“父目标文件” [81]。
- `ld(1)` 提供了 `-z aslr` 选项来为每个目标文件提供对地址空间布局和随机化的控制。`elfedit(1)` 已修改，从而简化了对关联的 `DT_SUNW_ASLR` 动态节项的编辑。请参见表 13-12 “ELF ASLR 值、`DT_SUNW_ASLR`”。

Oracle Solaris 11

- 可使用新的 elffile(1) 实用程序更全面地检查归档库及其成员。
- 通过对软件功能属性进行编码，可将 64 位进程限定到较低的 32 位地址空间。请参见“软件功能地址空间限制处理” [58]。

Oracle Solaris 10 1/13 发行版

- 通过链接编辑器的 -z discard-unused 选项，可以更灵活地通过链接编辑丢弃未使用的材料。请参见“删除未使用的材料” [164]。
- 通过链接编辑器的 -z strip-class 选项，可以更灵活地将不重要的节从目标文件中剥离。-z strip-class 选项取代了旧的 -s 选项，可对要剥离的节进行更精确的控制。

Oracle Solaris 10 8/11 发行版

- 链接编辑器可创建桩目标文件。桩目标文件是共享目标文件，完全根据 mapfile 生成，在不包含代码或数据时，可提供与实际目标文件相同的链接接口。桩目标文件可由链接编辑器快速生成，并可用于提高生成过程的并行性和降低生成过程的复杂性。请参见“桩目标文件” [70]。
- 链接编辑器可以使用 -z guidance 选项提供有关创建高质量目标文件的指导。请参见 ld(1)。
- 归档处理现在允许创建大小超过 4 GB 的归档文件。
- 局部审计程序现在可接收 la_preinit（） 和 la_activity（） 事件。请参见“运行时链接程序审计接口” [243]。
- 随延迟依赖项提供了一种更为稳健的用于测试功能是否存在的模型。请参见“测试功能” [114]和“提供 dlopen（） 的替代项” [98]。
- 提供了一种新的 *mapfile* 语法。请参见第 8 章 *Mapfile*。与原始的 System V 发行版 4 语言相比，该语法提供了更适合人阅读且可扩展性更好的语言。链接编辑器仍完全支持处理原始的 *mapfile*。有关原始 *mapfile* 的语法及用法，请参见附录 B, *System V 发行版 4（版本 1）mapfile*。
- 各个符号可与功能要求相关联。请参见“标识功能要求” [51]。利用该功能，可以在动态目标文件中创建一系列优化函数。请参见“创建符号功能函数系列” [59]和“功能节” [306]。
- 由链接编辑器创建的、包含特定于 Oracle Solaris 的 ELF 数据的目标文件在 e_ident[EI_OSABI] ELF 头中标记有 ELFOSABI_SOLARIS。以前，ELFOSABI_NONE 用于所有的目标文件。此项变更主要具有信息性价值，因为运行时链接程序仍继续将 ELFOSABI_NONE 与 ELFOSABI_SOLARIS 视为等效。但是，elfdump(1) 和类似的诊断工具可以使用此 ABI 信息为指定的目标文件生成更准确的信息。

- elfdump(1) 已扩展为使用 e_ident[EI_OSABI] ELF 头的值或新的 -O 选项来标识特定于给定 ABI 的 ELF 数据类型和值，并使用此信息更为准确地显示目标文件内容。Linux 操作系统中显示目标文件中的 ABI 特定信息的能力得到极大扩展。
- 通过使用 dlinfo(3C) 标志 RTLD_DI_MMAPCNT 和 RTLD_DI_MMAPS，可获取装入进程的目标文件的段映射信息。
- 链接编辑器可识别多个 GNU 链接编辑器选项。请参见 ld(1)。
- 链接编辑器为 SPARC 和 x86 目标提供了交叉链接。请参见“跨链接编辑” [25]。
- 链接编辑器现在可以合并 SHF_MERGE | SHF_STRING 字符串节。请参见“节合并” [294]。
- 现在，合并重定位节是创建可执行文件和共享目标文件时的缺省行为。请参见“组合重定位节” [170]。以前，此行为需要链接编辑器的 -z combreloc 选项。-z nocombreloc 用于禁用该缺省行为并与必须应用重定位的节保持一一对应关系。
- 可使用新的 elfedit(1) 实用程序编辑 ELF 目标文件。
- 可使用新的 elfwrap(1) 实用程序在 ELF 可重定位目标文件内封装任意数据文件。
- 提供了其他符号可见性属性。请参见位于“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]和表 12-23 “ELF 符号可见性” 中的已导出单件和删除属性描述。
- 链接编辑器和关联的 ELF 实用程序已从 /usr/ccs/bin 移至 /usr/bin。请参见“调用链接编辑器” [24]。
- 添加了符号排序节，可将内存地址与符号名称进行简单相关。请参见“符号排序节” [334]。
- 动态目标文件中可用的符号表信息已通过添加新的 .SUNW_ldynsym 节得到扩展。请参见“符号表节” [326]和表 12-5 “ELF 节类型 sh_type”。
- 由 crle(1) 管理的配置文件的格式已得到了增强，可以更好地标识文件。经过改进的格式可确保运行时链接程序不使用不兼容平台上生成的配置文件。
- 添加了新的重定位类型，在重定位计算中使用关联符号的大小。请参见“重定位” [316]。
- -z rescan-now、-z recan-start 和 -z rescan-end 选项在为链接编辑指定归档库方面提供了更大的灵活性。请参见“命令行中归档的位置” [30]。

过时的功能

以下项已经过时。这些项提供内部的或很少使用的功能。当前使用的任何关联的 ELF 定义都将被忽略，但是使用诸如 elfdump(1) 之类的工具仍然可以显示这些定义。

DT_FEATURE_1

此动态节标记标识了运行时功能的要求。请参见“动态节” [356]。此标记提供了功能标志 DTF_1_PARINIT 和 DTF_1_CONVEXP。DT_FEATURE_1 标记和所关联的标志不再由链接编辑器创建，也不再由运行时链接程序处理。

Solaris 10 5/08 发行版

- 现在，通过记录应用程序中的审计程序以及链接编辑器的 `-z globalaudit` 选项，可以启用全局审计。请参见“记录全局审计程序” [246]。
- 添加了附加的链接编辑器支持接口 `ld_open（）` 和 `ld_open64（）`。请参见“支持接口函数” [238]。

Solaris 10 8/07 发行版

- 通过 `-z altexec64` 选项和 `LD_ALTEXEC` 环境变量，可以更加灵活地执行备用的链接编辑器。
- 使用 *mapfile* 生成的符号定义现在可以关联到 ELF 节。请参见“SYMBOL_SCOPE / SYMBOL_VERSION 指令” [195]。
- 链接编辑器和运行时链接程序提供在共享目标文件中创建静态 TLS 的功能。此外，还建立了备份 TLS 预留空间，以实现在启动后共享目标文件中对静态 TLS 的有限使用。请参见“程序启动” [384]。

Solaris 10 1/06 发行版

- 提供了对 x64 中间代码模型的支持。请参见表 12-4 “ELF 特殊节索引”、表 12-8 “ELF 节属性标志” 和表 12-12 “ELF 特殊节”。
- 可以使用 `dlinfo(3C)` 标志 `RTLD_DI_ARGSINFO` 获取进程的命令行参数、环境变量和辅助向量数组。
- 通过链接编辑器的 `-B nodirect` 选项，可以更灵活地禁止外部引用的直接绑定。请参见第 6 章 直接绑定。

Solaris 10 发行版

- 现已支持 x64。请参见表 12-5 “ELF 节类型 `sh_type`”、“特殊节” [297]、“重定位类型” [324]、“线程局部变量访问” [399]和“线程局部存储的重定位类型” [402]。
- 通过重新构造文件系统，已将许多组件从 `/usr/lib` 移至 `/lib`。链接编辑器和运行时链接程序的缺省搜索路径已相应进行了更改。请参见“链接编辑器搜索的目录” [31]、“运行时链接程序搜索的目录” [88]和“安全性” [104]。
- 不再提供系统归档库。因此，无法再创建静态链接的可执行文件。请参见“静态可执行文件” [18]。
- 通过 `crle(1)` 的 `-A` 选项，可以更加灵活地定义替代依赖项。

- 链接编辑器和运行时链接程序可处理未指定值的环境变量。请参见“环境变量” [20]。
- 用于 dlopen(3C) 并作为显式依赖性定义的路径名现在可以使用任何保留的标记。请参见第 10 章 使用动态字符串标记建立依赖性。新的实用程序 moe(1) 提供了对使用保留标记的路径名的评估。
- dlsym(3C) 和新的句柄 RTLD_PROBE 提供了用于测试接口是否存在的最佳方法。请参见“提供 dlopen () 的替代项” [98]。

♦ ♦ ♦ 附 录 B

System V 发行版 4（版本 1）mapfile

注 - 本附录介绍了原始的 System V 发行版 4 *mapfile* 语言（版本 1）。虽然此 *mapfile* 语法目前仍受支持，但是建议在新应用程序中使用第 8 章 *Mapfile*中介绍的 *mapfile*语言版本 2。

链接编辑器会自动且智能地将可重定位目标文件中的输入节映射到正在创建的输出文件中的段。通过使用 -M 选项和关联的 mapfile，您可以更改链接编辑器提供的缺省映射。此外，使用 mapfile 还可以创建新段、修改属性以及提供符号版本控制信息。

注 - 使用 mapfile 选项时，您可以轻松创建不会执行的输出文件。链接编辑器可以在不使用 mapfile 选项的情况下生成正确的输出文件。

系统提供的 *mapfile* 样例位于 /usr/lib/ld 目录中。

mapfile 结构和语法

您可以在 mapfile 中输入以下基本类型的指令：

- 段声明。
- 映射指令。
- 节到段的排序。
- 大小符号声明。
- 文件控制指令。

每个指令可以跨越多行，并且可以包含任意数量的空格（包括换行符），但空格后面要跟随一个分号。

通常，段声明后面跟有映射指令。您可以声明段，然后定义节成为该段的一部分所依据的标准。如果在未首先声明要映射到的段（内置段除外）的情况下输入映射指令或大小符号声明，则会为该段赋予缺省属性。此类段为隐式声明的段。

大小符号声明和文件控制指令可以出现在 mapfile 中的任何位置。

以下各节将针对每种指令类型进行介绍。对于所有语法讨论，以下表示法都适用。

- 所有项都为固定宽度，所有冒号、分号、等号和 at (@) 符号都按原样输入。
- 所有以斜体表示的项都是可替换的。
- { .... }* 表示“零个或多个”。
- { .... }+ 表示“一个或多个”。
- [....] 表示“可选”。
- `section_names` 和 `segment_names` 遵守与 C 标识符相同的规则，其中将句点 (.) 视为一个字母。例如，`.bss` 为合法名称。
- `section_names`、`segment_names`、`file_names` 和 `symbol_names` 区分大小写。其他名称不区分大小写。
- 除编号之前、名称或值的中间位置以外，空格或者换行符可以出现在其他任何位置。
- 以 # 开始并以换行符结束的注释可以出现在任何允许出现空格的位置。

段声明

段声明可在输出文件中创建新段，或更改现有段的属性值。现有段可以是您先前定义的段，也可以是下面即将介绍的四个内置段之一。

段声明的语法如下：

```
segment_name = {segment_attribute_value}*;
```

对于每个 `segment_name`，可以按任意顺序指定任何数量的 `segment_attribute_values`，但每个值都要由空格进行分隔。每个段属性只能有一个属性值。下表列出了段属性及其有效值。

表 B-1 *mapfile* 段属性

属性	值
`segment_type`	`LOAD | NOTE | NULL | STACK`
`segment_flags`	`? [E] [N] [O] [R] [W] [X]`
`virtual_address`	`V`*number*
`physical_address`	`P`*number*
`length`	`L`*number*
`rounding`	`R`*number*
`alignment`	`A`*number*

有四个内置段，其缺省属性值如下所示：

- `text` – `LOAD`，`?RX`，未指定 `virtual_address`、`physical_address` 或 `length`，按 CPU 类型将 `alignment` 值设置为缺省值。

- data – LOAD，?RWX，未指定 virtual_address、physical_address 或 length，按 CPU 类型将 alignment 值设置为缺省值。
- bss – 已禁用，LOAD，?RWX，未指定 virtual_address、physical_address 或 length，按 CPU 类型将 alignment 值设置为缺省值。
- note – NOTE。

缺省情况下，禁用 bss 段。任何类型为 SHT_NOBITS（此类型为节的唯一输入）的节都是在 data 段中捕获的。有关 SHT_NOBITS 节的完整说明，请参见表 12-5 “ELF 节类型 sh_type”。最简单的 bss 声明足以创建 bss 段。

```
bss =;
```

任何 SHT_NOBITS 节都是由此段（而不是 data 段）捕获的。此段采用最简单的形式，并且使用与应用于任何其他段相同的缺省值对齐。还可以声明其他既可创建段又可为指定属性赋值的段属性。

链接编辑器的行为方式就好像在读入 mapfile 之前已经声明了这些段。请参见“*mapfile* 选项缺省值” [420]。

输入段声明时，请注意以下事项：

- 数字可以是十六进制、十进制或八进制，所遵守的规则与 C 语言中的规则相同。
- V、P、L、R 或 A 和数字之间不允许有空格。
- segment_type 值可以是 LOAD、NOTE、NULL 或 STACK。如果未指定值，则缺省的段类型为 LOAD。
- segment_flags 的值包括 R、W、X 和 O，分别表示可读、可写、可执行以及顺序。问号 (?) 与构成 segment_flags 值的单个标志之间不允许有空格。
- LOAD 段的 segment_flags 值缺省为 RWX。
- 不能为 NOTE 段指定除 segment_type 以外的任何段属性值。
- 允许有一个值为 STACK 的 segment_type。只能指定从 segment_flags 中选择的段访问要求。
- 隐式声明的段采用以下缺省设置：segment_type 值为 LOAD，segment_flags 值为 RWX，缺省的 virtual_address、physical_address 和 alignment 值，且没有 length 限制。

注 - 链接编辑器基于先前段的属性值计算当前段的地址和长度。

- LOAD 段可以具有显式指定的 virtual_address 值或 physical_address 值，以及段的最大 length 值。
- 如果某个段的 segment_flags 值为 ?，并且后面未跟任何内容，则该值缺省为不可读、不可写并且不可执行。
- alignment 值用于计算段开头的虚拟地址。此对齐值仅影响指定了对齐的段。其他段仍采用缺省的对齐，除非更改了它们的 alignment 值。

- 如果有任何 virtual_address、physical_address 或 length 属性值未设置，链接编辑器将在创建输出文件时计算这些值。
- 如果没有为段指定 alignment 值，则对齐值将设置为内置的缺省值。此缺省值因 CPU 的不同而异，甚至也可能因软件修订版的不同而异。
- 如果同时为段指定了 virtual_address 和 alignment 值，则 virtual_address 值优先。
- 如果为段指定了 virtual_address 值，则程序头中的 alignment 字段将包含缺省的对齐值。
- 如果为段设置了 rounding 值，则此段的虚拟地址将舍入为下一个符合给定值的地址。该值只影响指定了它的段。如果没有给定值，将不执行任何舍入操作。

注 - 如果指定了 virtual_address 值，段将放置在该虚拟地址处。对于系统内核，此方法可生成正确的结果。对于通过 exec(2) 启动的文件，此方法将生成错误的输出文件，因为段与其页边界的相对偏移量是错误的。

使用 ?E 标志可以创建空段。此空段没有关联的节。此段可为 LOAD 段或 NULL 段。只能为可执行文件指定空 LOAD 段。这些段必须有指定的大小和对齐方式。这些段将导致在进程启动时创建内存保留空间。空的 NULL 段可添加程序头项，后处理实用程序可以使用这些程序头项。这些段不应指定任何附加属性。LOAD 段和 NULL 段允许具有多个定义。

使用 ?N 标志可以控制是否将 ELF 头和任何程序头作为第一个可装入段的一部分包括在内。缺省情况下，ELF 头和程序头包括在第一个段内。这些头中的信息通常由运行时链接程序用在映射的映像中。使用 ?N 选项将导致从第一个段的第一个节开始计算映像的虚拟地址。

使用 ?O 标志可以控制输出文件中各节的顺序。此标志用于与编译器的 -xF 选项一起使用。使用 -xF 选项编译文件时，该文件中的每个函数将放置在与 .text 节具有相同属性的单独节中。这些节称为 .text%*function_name*。

例如，使用 -xF 选项编译包含 main（）、foo（） 和 bar（） 这三个函数的文件时，会生成一个可重定位的目标文件，并会将三个函数的文本放置在名为 .text%main、.text%foo 和 .text%bar 的节中。由于 -xF 选项强制实行每节一个函数，因此使用 ?O 标志控制各节的顺序实际上是控制函数的顺序。

请考虑以下用户定义的 mapfile。

```
text = LOAD ?RXO;
text: .text%foo;
text: .text%bar;
text: .text%main;
```

第一个声明将 ?O 标志与缺省文本段进行关联。

如果源文件中函数定义的顺序为 main、foo 和 bar，则最终的可执行文件所包含的函数顺序为 foo、bar 和 main。

对于具有相同名称的静态函数，还必须使用文件名。?O 标志强制按 mapfile 中的要求对节进行排序。例如，如果静态函数 bar（） 存在于 a.o 和 b.o 文件中，并且要将 a.o 文件中的 bar（） 函数放置在 b.o 文件中的 bar（） 函数之前，则 mapfile 项将显示为：

```
text: .text%bar: a.o;
text: .text%bar: b.o;
```

虽然此语法允许具有以下项：

```
text: .text%bar: a.o b.o;
```

但此项不能保证将 a.o 文件中的 bar（） 函数放置在 b.o 中的 bar（） 之前。由于结果不可靠，因此建议不要使用第二种格式。

映射指令

映射指令指示链接编辑器如何将输入节映射到输出段。本质上，就是指定要映射到的段，并指明节为了映射到指定的段而必须具备的属性。节为映射到特定段而必须具备的 section_attribute_values 集合称为此段的入口条件。节必须完全满足段的入口条件，才能置于输出文件的指定段中。

映射指令的语法如下：

```
segment_name : {section_attribute_value}* [: {file_name}+];
```

对于 segment_name，可以按任意顺序指定任何数量的 section_attribute_values，其中每个值由空格进行分隔。每个节属性最多允许具有一个节属性值。您还可以通过 file_name 声明指定节必须来自某个特定的 .o 文件。下表列出了节属性及其有效值。

表 B-2 节属性

节属性	值
section_name	任何有效的节名称
section_type	$PROGBITS
	$SYMTAB
	$STRTAB
	$REL
	$RELA
	$NOTE
	$NOBITS
section_flags	? [[!]A] [[!]W] [[!]X]

输入映射指令时，请注意以下几点：

- 最多只能从上面列出的 section_types 中选择一个 section_type。上面列出的 section_types 是内置类型。有关 section_types 的更多信息，请参见“节” [281]。
- section_flags 值包括 A、W 和 X，分别表示可分配、可写和可执行。如果个别标志之前加有一个叹号 (!)，则链接编辑器将检查是否未设置此标志。构成 section_flags 值的问号、叹号和各个标志之间不允许有空格。
- file_name 可以是任何形式为 *filename 或 archive_name(component_name)，的合法文件名，例如，/lib/libc.a(printf.o)。链接编辑器不会检查文件名的语法。
- 如果 file_name 的形式为 *filename，链接编辑器将决定命令行中文件的 basename(1)。此基本名称用于与指定的 file name 进行匹配。换言之，mapfile 中的 filename 只需要与命令行中文件名的最后一部分匹配即可。请参见“映射示例” [419]。
- 如果在链接编辑过程中使用 -l 选项，并且 -l 选项后的库位于当前目录中，则必须在库的前面添加 ./ 或整个路径名（在 mapfile 中），以便创建匹配。
- 对于特殊的输出段，可能显示多个指令行。例如，以下指令集是合法的：

```
S1 : $PROGBITS;
S1 : $NOBITS;
```

  为段输入多个映射指令行是为节属性指定多个值的唯一方法。
- 一个节可以与多个入口条件匹配。在这种情况下，将使用 mapfile 中第一个遇到的具有该入口条件的段。例如，如果 mapfile 显示为：

```
S1 : $PROGBITS;
S2 : $PROGBITS;
```

  则 $PROGBITS 节将映射到 S1 段。

段内节的排序

使用以下表示法可以指定节在段中放置的顺序：

```
segment_name | section_name1;
segment_name | section_name2;
segment_name | section_name3;
```

以上述形式命名的节将按照它们在 mapfile 中列出的顺序放置在任何未命名的节之前。

大小符号声明

使用大小符号声明，可以定义新的全局绝对符号，以代表指定段的大小（字节）。可以在目标文件中引用此符号。大小符号声明的语法如下：

```
segment_name @ symbol_name;
```

symbol_name 可以是任何合法的 C 标识符。链接编辑器不会检查 symbol_name 的语法。

文件控制指令

使用文件控制指令，可以指定共享目标文件中有哪些版本定义在链接编辑期间可用。文件控制定义的语法如下：

```
shared_object_name - version_name [ version_name .... ];
```

version_name 是指定 shared_object_name 中包含的版本定义名称。

映射示例

以下示例为用户定义的 mapfile。示例中左边的编号用于教学演示。实际上只有编号右边的信息会出现在 mapfile 中。

例 B-1　　用户定义的 *mapfile*

```
1.  elephant : .data : peanuts.o *popcorn.o;
2.  monkey : $PROGBITS ?AX;
3.  monkey : .data;
4.  monkey = LOAD V0x80000000 L0x4000;
5.  donkey : .data;
6.  donkey = ?RX A0x1000;
7.  text = V0x80008000;
```

在此示例中处理四个单独的段。隐式声明的 elephant 段（第 1 行）从 peanuts.o 和 popcorn.o 文件中接收所有 .data 节。请注意，*popcorn.o 能够匹配任何可以提供给链接编辑的 popcorn.o 文件。该文件无需位于当前目录中。另一方面，如果已将 /var/tmp/peanuts.o 提供给链接编辑，它将不会与 peanuts.o 相匹配，因为它没有 * 前缀。

隐式声明的 monkey 段（第 2 行）接收既有 $PROGBITS 属性又有可分配且可执行属性 (?AX) 的节，同时还接收所有尚未存在于 elephant 段中且名为 .data（第 3 行）的节。进入 monkey 段的 .data 节无需是 $PROGBITS 节或可分配且可执行的节，因为输入 section_type 和 section_flags 值的行与 section_name 值所在的行不是同一个行。

如第 2 行的 $PROGBITS“与”?AX 所示，同一行中的属性之间存在“与”的关系。如第 2 行的 $PROGBITS ?AX“或”第 3 行的 .data 所示，相同段不同行的属性之间存在“或”的关系。

在第 2 行中，monkey 段已进行如下隐式声明：segment_type 值为 LOAD，segment_flags 值为 RWX，并且未指定 virtual_address、physical_address、length 和 alignment 值（使用缺省值）。在第 4 行中，monkey 的 segment_type 值设置为 LOAD。由于

segment_type 属性值未发生更改，因此不会发出任何警告。virtual_address 值设置为 0x80000000，最大 length 值设置为 0x4000。

第 5 行隐式声明了 donkey 段。入口条件的目的是将所有 .data 节路由到此段。但是实际上，没有任何节进入此段，因为第 3 行中 monkey 的入口条件会捕获所有这些节。在第 6 行中，segment_flags 值设置为 ?RX，alignment 值设置为 0x1000。由于这两个属性值都发生了更改，因此会发出警告。

第 7 行将文本段的 virtual_address 值设置为 0x80008000。

为了进行说明，用户定义的 mapfile 示例设计为会发出警告。如果要更改指令的顺序以避免发出警告，请使用以下示例：

```
1.  elephant : .data : peanuts.o *popcorn.o;
4.  monkey = LOAD V0x80000000 L0x4000;
2.  monkey : $PROGBITS ?AX;
3.  monkey : .data;
6.  donkey = ?RX A0x1000;
5.  donkey : .data;
7.  text = V0x80008000;
```

以下 mapfile 示例使用段内节的排序：

```
1.  text = LOAD ?RXN V0xf0004000;
2.  text | .text;
3.  text | .rodata;
4.  text : $PROGBITS ?A!W;
5.  data = LOAD ?RWX R0x1000;
```

此示例中处理了 text 和 data 段。第 1 行声明 text 段的 virtual_address 为 0xf0004000，并且没有将 ELF 头或任何程序头作为此段的地址计算的一部分包括在内。第 2 行和第 3 行启用段内节排序，并指定 .text 和 .rodata 节是此段中的前两个节。结果是 .text 节的虚拟地址为 0xf0004000，并且 .rodata 节紧跟该地址之后。

构成 text 段的任何其他 $PROGBITS 节位于 .rodata 节之后。第 5 行声明了 data 段，并指定其虚拟地址必须从 0x1000 字节边界开始。构成 data 段的第一个节也驻留在文件映像内的 0x1000 字节边界上。

mapfile 选项缺省值

链接编辑器使用缺省的 segment_attribute_values 和对应的缺省映射指令定义了四个内置段（text、data、bss 和 note）。虽然链接编辑器不使用实际的 mapfile 提供缺省值，但是缺省 mapfile 的模型可以帮助说明链接编辑器遇到 mapfile 时出现的情况。

以下示例说明 mapfile 在使用链接编辑器的缺省值时的表现。链接编辑器开始执行操作时，行为上好像已经读入了 mapfile。随后链接编辑器会读取 mapfile 并增大或更改缺省值。

```
text = LOAD ?RX;
text : ?A!W;
data = LOAD ?RWX;
data : ?AW;
note = NOTE;
note : $NOTE;
```

读入 mapfile 中的每个段声明时，都会将其与现有的段声明列表进行如下比较：

1. 如果 mapfile 中尚不存在此段，但是存在具有相同段类型值的其他段，则在具有相同 segment_type 的所有现有段之前添加此段。
2. 如果现有 mapfile 中没有任何段与刚读入的段的 segment_type 值相同，则按 segment_type 值添加该段以保持以下顺序：

   INTERP

   LOAD

   DYNAMIC

   NOTE
3. 如果段的 segment_type 值为 LOAD，并且已经为这个可装入 (LOAD) 的段定义了 virtual_address 值，应将此段放置在任何没有定义 virtual_address 值或 virtual_address 值较大的可装入 (LOAD) 的段之前，但应放置在任何 virtual_address 值较小的段之后。

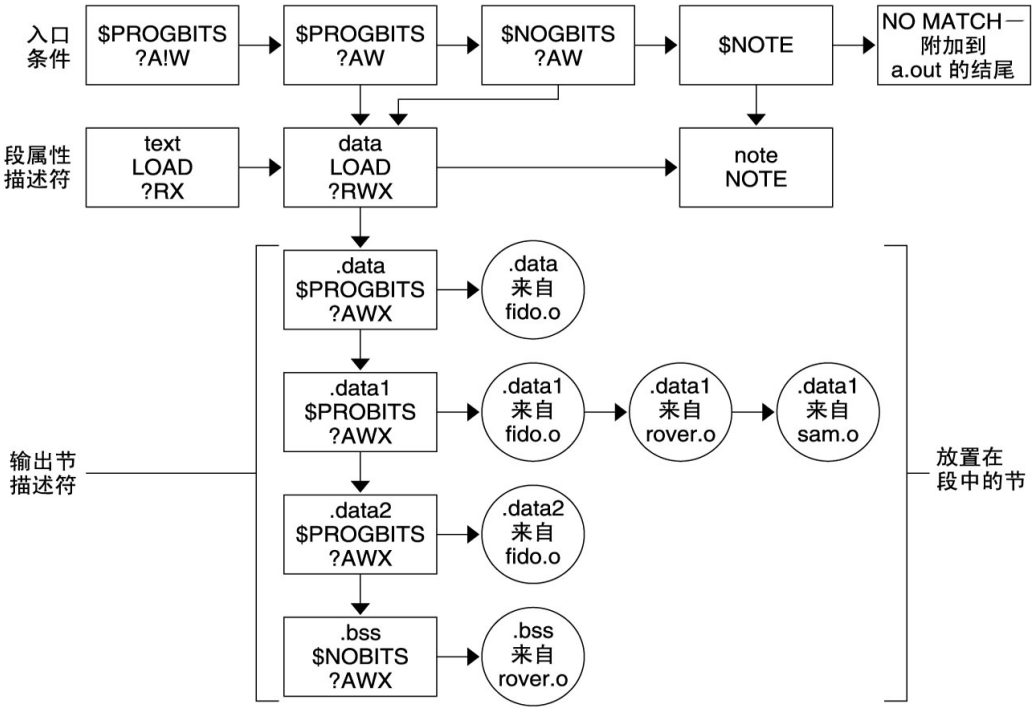
读入 mapfile 中的每个映射指令时，指令将添加在已经为同一个段指定的所有其他映射指令之后，但要在此段的缺省映射指令之前。

内部映射结构

映射结构是基于 ELF 的链接编辑器中最重要的数据结构之一。对应于模型缺省 mapfile 的缺省映射结构由链接编辑器使用。任何用户 mapfile 都可以增大或覆盖缺省映射结构中的特定值。

图 B-1 “简单的映射结构” 说明了一个典型但某种程度上已简化的映射结构。“入口条件”框对应于缺省映射指令中的信息。“段属性描述符”框对应于缺省段声明中的信息。“输出节描述符”框提供了每段下的各个节的详细属性。节本身以循环方式显示。

图 B-1 简单的映射结构



将节映射到段时，链接编辑器执行以下步骤：

1. 读入节时，链接编辑器会检查入口条件列表以查看是否存在匹配。必须匹配所有指定的条件。

   在图 B-1 “简单的映射结构” 中，`text` 段下节的 `section_type` 值必须为 `$PROGBITS`，`section_flags` 值必须为 `?A!W`。此节的名称无需为 `.text`，因为入口条件中未指定任何名称。节的 `section_flag` 值可以是 `X`，也可以是 `!X`，因为入口条件中未指定任何执行位。

   如果不与任何入口条件匹配，则将节放置在输出文件的末尾（位于所有其他段之后）。不会为此信息创建任何程序头项。
2. 当节位于段下时，链接编辑器会检查此段中现有输出节描述符的列表，方式如下：

   如果节属性值与现有输出节描述符的属性值完全匹配，则将此节放置在与该输出节描述符关联的节列表的末尾。

   例如，`section_name` 值为 `.data1`、`section_type` 值为 `$PROGBITS`、`section_flags` 值为 `?AWX` 的节将进入图 B-1 “简单的映射结构” 中第二个“入口条件”框，可以将其放置在 `data` 段中。此节与第二个“输出节描述符”框（`.data1`、`$PROGBITS`、`?AWX`）完全匹

配，将被添加到与该框关联的列表的末尾。fido.o、rover.o 和 sam.o 中的 .data1 节说明了这一点。

如果未找到匹配的输出节描述符，但是存在其他具有相同 section_type 的输出节描述符，则使用与节相同的属性值创建一个新的输出节描述符，并且此节与新的输出节描述符相关联。输出节描述符和节放置在节类型相同的最后一个输出节描述符之后。图 B-1 “简单的映射结构” 中的 .data2 节便是按照此方式放置的。

如果不存在其他具有所示节类型的输出节描述符，则将创建一个新的输出节描述符，并将节放置在此节中。

注 - 如果输入节的用户定义类型值介于 SHT_LOUSER 和 SHT_HIUSER 之间，则将其视为 $PROGBITS 节。mapfile 中没有命名此 section_type 值的方法，但是可以使用入口条件中的其他属性值规（section_flags、section_name）重定向这些节。

---

3. 如果在读取所有命令行目标文件和库之后，段中没有任何节，则不会为该段生成任何程序头项。

注 - 类型为 $SYMTAB、$STRTAB、$REL 和 $RELA 的输入节由链接编辑器在内部使用。引用这些节类型的指令只能将链接编辑器生成的输出节映射到段。

索引

数字和符号

___tls_get_addr，386
__thread，381
__tls_get_addr，386
.got 见 全局偏移表
.plt 见 过程链接表
/lib，31，32，88，106
/lib/64，31，32，88，106
/lib/secure，104
/lib/secure/64，104
/usr/bin/ld 见 链接编辑器
/usr/ccs/bin/ld 见 链接编辑器
/usr/ccs/lib，24
/usr/lib，31，32，88，106
/usr/lib/64，31，32，88，106
/usr/lib/64/ld.so.1，87，256
/usr/lib/ld.so.1，87，256
/usr/lib/secure，104，245
/usr/lib/secure/64，104，245
$CAPABILITY 见 搜索路径
$ISALIST 见 搜索路径
$ORIGIN 见 搜索路径
$OSNAME 见 搜索路径
$OSREL 见 搜索路径
$PLATFORM 见 搜索路径
32 位/64 位，20
  ld-support，238
  rtld-audit，247
  搜索路径
    安全性，104
    运行时链接程序，32，88，106
    配置，90
    链接编辑器，31
  环境变量，20
  运行时链接程序，87

A

安全性，104，234
ABI 见 应用程序二进制接口
ar(1)，27
as(1)，17
atexit(3C)，100

B

版本控制，209
  在映像内生成定义，48，210
  基版本定义，211
  定义，210，210，215
  定义公共接口，48，211
  文件名，210
  标准化，216
  概述，209
  绑定到定义，215，219
  运行时验证，217，218
绑定
  依赖项排序，127
  到共享目标文件依赖项，124，215
  到弱版本定义，222
  到版本定义，215
  延迟，94，107，118
  直接，169
编辑器选项
  -K PIC，163
编译环境，17，19，29，123
  参见 链接编辑与链接编辑器
编译器驱动程序，24
编译器选项
  -K pic，138，162
  -xF，165，304
  -xpg，174
  -xregs=*no%appl*，139

标准过滤器，128，128

C
插入，37，93，96，115
  使用直接绑定，145
  接口稳定性，210
  显式插入，153
  检查，37
程序的解释程序，87，355
  参见 运行时链接程序
初始化和终止，24，33，100
错误消息
  运行时链接程序
    复制重定位大小差异，70，172
    找不到版本定义，217
    无法找到共享目标文件，90，107
    无法找到符号，114
    重定位错误，95，217
  链接编辑器
    *soname*冲突，126
    不可用版本，220
    共享目标文件名称冲突，126
    多重定义符号，39
    未定义符号，39，39
    根据非可写的节进行重定位，163
    符号未指定给版本，48
    符号警告，38，38
    隐式引用中的未定义符号，40
cc(1)，17，24
CC(1)，24
重定位，91，169，173，314
  位移，69
  即时，94
  复制，69，170
  延迟，94
  符号，91，169
  运行时链接程序
    符号查找，91，94，107，118
  非符号，91，169
COMDAT，241，304
COMMON，36，282
crle(1)
  交互，369，369
  安全性，104，104，235
  审计，249
  选项
    -e，174
    -l，90
    -s，104

D
动态可执行文件，18
动态链接，19
  实现，314，352
动态信息标记
  NEEDED，88，124
  RUNPATH，89
  SONAME，125
  SYMBOLIC，173
  TEXTREL，163
段，23，159
  数据，159，161
  文本，159，161
多重定义符号，29，37，304
多重定义数据，304
dlclose(3C)，100，106
dldump(3C)，35
dlerror(3C)，106
dlfcn.h，106
dlinfo(3C)
  模式
    RTLD_DI_DEFERRED，100
    RTLD_DI_DEFERRED_SYM，100
    RTLD_DI_ORIGIN，234
dlopen(3C)，87，105，106，111
  共享目标文件命名约定，124
  动态可执行文件，107，111
  排序效果，110
  模式
    RTLD_FIRST，113，227，229
    RTLD_GLOBAL，111，113
    RTLD_GROUP，112
    RTLD_LAZY，108
    RTLD_NOLOAD，244
    RTLD_NOW，94，103，108
    RTLD_PARENT，112，112，112，112
  版本验证，218
  组，92，107

dlsym(3C)，87，105，113
  版本验证，218
  特殊句柄
    RTLD_DEFAULT，41，113
    RTLD_NEXT，97，113，153，153
    RTLD_PROBE，41，99，113

E

ELF，17，23，271
  参见 目标文件
elf(3E)，237
elfdump(1)，159
exec(2)，23，273

F

分页，348，352
符号
  COMMON，36，282
  LCOMMON，282
  专用接口，209
  作用域，108，111
  全局，209，328
  公共接口，209
  删除，49
  参考，27
  可见性，328，330
    global（全局），92
    local（局部），92
    singleton，92，93，108
    singleton 对直接绑定的影响，153，154
  多重定义，29，37，304
  定义，27
  定义的，36
  寄存器，321，335
  局部，328
  已排序，282
  弱，41，328，328
  归档提取，27
  暂定，36
    COMMON，282
    LCOMMON，282
    输出文件中的排序，41
    重新对齐，45
  未定义，27，36，39，282
  类型，329
  绝对，282，282
  自动删除，49
  自动缩减，211
  运行时查找，108，116
    延迟，94，107，118
符号保留名称，50
  _DYNAMIC，51
  _edata，51
  _end，51
  END，51
  _etext，51
  _fini，33
  _GLOBAL_OFFSET_TABLE_，51，163，372
  _init，33
  _PROCEDURE_LINKAGE_TABLE_，51
  _start，51
  START，51
  main，51
符号处理，35
符号解析，36
  复杂，37
  多重定义，29
  插入，93
  搜索作用域
    world（全局），92
  搜索范围
    group，92
  生成输出文件映像，50
  简单，37
  致命，38
符号可见性，35
辅助过滤器，128，131
filtee，127

G

更新和新增功能，407
功能
  平台，51
  硬件，51
  计算机，51
  软件，51
共享库 见 共享目标文件

共享目标文件，17，18，88，123
  作为过滤器，127
  依赖项排序，127
  依赖项组，92，107
  具有依赖项，126
  命名约定，29，123
  实现，314，352
  已使用依赖项删除，28
  显式定义，40
  补偿依赖项，166
  记录运行时名称，124
  链接编辑器处理，28
  隐式定义，40
归档文件，29
  包含共享目标文件，125
  命名约定，29
  多遍检查，28
  链接编辑器处理，27
过程链接表，300，356
  _PROCEDURE_LINKAGE_TABLE_，51
  与位置无关的代码，162
  动态引用，360，360，361，362
  延迟引用，94
  重定位，316，372
    64 位 SPARC，374
    SPARC，317，372
    x64，324，378
    x86，322，377
过滤器，127
  减少 *filtee* 搜索，228，230
  功能系列，227
  标准，128，128
  特定于指令集，229
  特定于系统，231
  辅助，128，131
GOT 见 全局偏移表

H
环境变量
  32 位/64 位，20
  LD_AUDIT，105，245
  LD_BIND_NOW，94，103，118
  LD_CONFIG，104
  LD_DEBUG，117
  LD_EXEC_OPTIONS，26
  LD_EXEC_UNSET，26
  LD_LIBRARY_PATH，32，89，127
    安全性，104
    审计，249
  LD_LOADFLTR，134
  LD_NOAUDIT，245
  LD_NOAUXFLTR，133
  LD_NODIRECT，146，147
  LD_NOLAZYLOAD，98
  LD_NOVERSION，220
  LD_OPTIONS，25，83
  LD_PIE_OPTIONS，26
  LD_PIE_UNSET，26
  LD_PRELOAD，93，96，105，153
  LD_PROFILE，174
  LD_PROFILE_OUTPUT，174
  LD_RELOC_OPTIONS，26
  LD_RELOC_UNSET，26
  LD_RUN_PATH，33
  LD_SHARED_OPTIONS，26
  LD_SHARED_UNSET，26
  LD_SIGNAL，105
  LD_UNSET，25
  SGS_SUPPORT，237

J
基本地址，347，347
接口
  专用，209
  公共，209
节，23，23，159
  参见 节标志、节名称、节编号和节类型
节编号
  SHN_ABS，282，329，332
  SHN_AFTER，282，292，293
  SHN_AMD64_LCOMMON，282，332
  SHN_BEFORE，282，292，293
  SHN_COMMON，282，328，332，332
  SHN_HIOS，282，282
  SHN_HIPROC，282
  SHN_HIRESERVE，282
  SHN_LOOS，282，282

SHN_LOPROC，282
SHN_LORESERVE，282
SHN_SUNW_IGNORE，282
SHN_UNDEF，282，332
SHN_XINDEX，282
节标志
SHF_ALLOC，291，301
SHF_COMPRESSED，78，292，295
SHF_EXCLUDE，241，293
SHF_EXECINSTR，291
SHF_GROUP，292，305
SHF_INFO_LINK，291
SHF_LINK_ORDER，282，292
SHF_MASKOS，292
SHF_MASKPROC，293
SHF_MERGE，291，294
SHF_ORDERED，293
SHF_OS_NONCONFORMING，292
SHF_STRINGS，291，294
SHF_TLS，292，382
SHF_WRITE，291
节类型
SHT_DYNAMIC，286，356
SHT_DYNSTR，286
SHT_DYNSYM，286
SHT_FINI_ARRAY，287
SHT_GROUP，287，292，305，305
SHT_HASH，286，308，356
SHT_HIOS，287
SHT_HIPROC，289
SHT_HISUNW，287
SHT_HIUSER，289
SHT_INIT_ARRAY，287
SHT_LOOS，287
SHT_LOPROC，289
SHT_LOSUNW，287
SHT_LOUSER，289
SHT_NOBITS，286
.bss，299
.lbss，300
.SUNW_bss，301
.tbss，301
p_memsz 计算，348
sh_offset，284
sh_size，284
SHT_NOTE，286，312
SHT_NULL，285
SHT_PREINIT_ARRAY，287
SHT_PROGBITS，286，356
SHT_REL，286
SHT_RELA，286
SHT_SHLIB，287
SHT_SPARC_GOTDATA，289，289
SHT_STRTAB，286
SHT_SUNW_ANNOTATE，79，79，288
SHT_SUNW_cap，288
SHT_SUNW_COMDAT，241，289，304
SHT_SUNW_DEBUG，288
SHT_SUNW_DEBUGSTR，288
SHT_SUNW_dof，288
SHT_SUNW_LDYNSYM，286，288
SHT_SUNW_move，288，310
SHT_SUNW_SIGNATURE，288
SHT_SUNW_syminfo，289
SHT_SUNW_symsort，288
SHT_SUNW_tlssort，288
SHT_SUNW_verdef，289，337，342
SHT_SUNW_verneed，289，337，339
SHT_SUNW_versym，289，338，339，341
SHT_SYMTAB，286，329
SHT_SYMTAB_SHNDX，287
节名称
.bss，23，170
.data，23，167
.debug，78
.dynamic，51，87，173
.dynstr，50
.dynsym，50
.fini，33，100
.fini_array，33，100
.got，51，91
.init，33，100
.init_array，33，100
.interp，87
.picdata，167
.plt，51，94，173
.preinit_array，33，100

.rela.text，23
.rodata，167
.strtab，23，50
.SUNW_reloc，170
.SUNW_version，338
.symtab，23，49，50
.tbss，382
.tdata，382
.tdata1，382
.text，23
.zdebug，78
解释程序 见 运行时链接程序
局部符号，328

K
可执行链接格式 见 ELF
可重定位目标文件，18
库
共享，314，352
命名约定，29
归档文件，29

L
链接编辑，17，326，352
共享目标文件和归档混用，30
共享目标文件处理，28
动态，314，352
命令行中文件的位置，30
库输入处理，27
库链接选项，27
归档处理，27
搜索路径，31，31
添加其他库，29
绑定到版本定义，215，219
输入文件处理，27
链接编辑器，17，23
使用编译器驱动程序调用，24
外部绑定，50
指定选项，25
更新和新增功能，407
概述，23
段，23
直接调用，24
节，23
调试帮助，83
跨链接编辑，25
错误消息 见 错误消息
链接编辑器输出
与位置无关的可执行文件，18
共享目标文件，18
动态可执行文件，18
可重定位目标文件，18
链接编辑器选项
-64，131
-a，138
-B direct，139，140，146，146
-B dynamic，30
-B eliminate，49
-B group，92，112，368
-B local，48
-B nodirect，155
-B reduce，49，197，224
-B static，30，138
-B symbolic，147，173
-D，83
-d n，137，140
-d y，138
-e，51
-F，127
-f，128
-G，123，138，140
-h，89，124，139，225
-i，32
-l，27，29，123，137
-L，31，137
-M，177
定义接口，139
定义段，23
定义版本，211
定义符号，42，42
-m，29，37
-p，246
-P，246
-r，24，137
-R，33，126，139，140
-S，237
-t，38，38

-u，42，42
-Y，31
-z allextract，28
-z ancillary，75
-z aslr，371
-z compress-sections，78
-z defaultextract，28
-z deferred，99
-z defs，41，139，245
-z direct，146，147
-z discard-unused，164
  依赖项删除，28，140，165
  文件删除，165
  节删除，138，164
-z endfiltee，369
-z finiarray，34
-z globalaudit，246
-z groupperm，370
-z guidance，137，138，140
  未使用的依赖项，165
  未使用的文件，165
-z ignore，165
-z initarray，34
-z initfirst，368
-z interpose，93，153，369
-z lazyload，97，139，140，370
-z ld32，238
-z ld64，238
-z loadfltr，134，368
-z mapfile-add，181
-z muldefs，39
-z nocompstrtab，50，295
-z nodefaultlib，32，369
-z nodefs，40，95
-z nodelete，368
-z nodirect，146
-z nodlopen，368
-z nodump，369
-z nolazyload，97
-z noldynsym，333，335
-z nopartial，311
-z noversion，48，212，217
-z now，94，103，108
-z parent，83
-z record，165
-z redlocsym，333
-z relax，371
-z rescan-end，30
-z rescan-now，30
-z rescan-start，30
-z strip-class，49，50，241，288
-z target，25
-z text，138，163
-z type，17
-z verbose，69
-z weakextract，28，328
链接编辑器支持接口 (*ld-support*)，237
  ld_atexit（），242
  ld_atexit64（），242
  ld_file（），240
  ld_file64（），240
  ld_input_done（），241
  ld_input_section（），240
  ld_input_section64（），240
  ld_open（），239
  ld_open64（），239
  ld_section（），241
  ld_section64（），241
  ld_start（），239
  ld_start64（），239
  ld_version（），238
lari(1)，145
LCOMMON，282
LD_AUDIT，105，245
LD_BIND_NOW，94，103，118
  IA 重定位，378，379
  SPARC 32 位重定位，374
  SPARC 64 位重定位，377
LD_CONFIG，104
LD_DEBUG，117
LD_EXEC_OPTIONS，26
LD_EXEC_UNSET，26
LD_LIBRARY_PATH，89，127
  安全性，104
  审计，249
LD_LOADFLTR，134
LD_NOAUDIT，245
LD_NOAUXFLTR，133

LD_NODIRECT，146，147
LD_NOLAZYLOAD，98
LD_NOVERSION，220
LD_OPTIONS，25，83
LD_PIE_OPTIONS，26
LD_PIE_UNSET，26
LD_PRELOAD，93，96，105，153
LD_PROFILE，174
LD_PROFILE_OUTPUT，174
LD_RELOC_OPTIONS，26
LD_RELOC_UNSET，26
LD_RUN_PATH，33
LD_SHARED_OPTIONS，26
LD_SHARED_UNSET，26
LD_SIGNAL，105
LD_UNSET，25
ld.so.1(1) 见 运行时链接程序
ld(1) 见 链接编辑器
ldd(1)，88
ldd(1) 选项
  -d，70，95，172
  -i，102
  -r，70，96，172
  -u，28
  -v，217
libelf.so.1，238，271
libldstab.so.1，238
lorder(1)，28，84

M

名称空间，244
命名约定
  共享目标文件，29，123
  库，29
  归档文件，29
目标文件，17
  全局偏移表 见 全局偏移表
  基本地址，347，347
  字符串表，325，326
  数据表示形式，273
  段内容，348，348
  段权限，347，348
  段类型，344，347
  注释节，312，313
  程序头，343，346，346，346
  程序的解释程序，355
  程序装入，348
  符号表，326，332
  节名称，302，302
  节头，281，302
  节对齐，284
  节属性，290，302
  节类型，284，302
  节组标志，305
  辅助，74
  过程链接表 见 过程链接表
  运行时预装入，96
  重定位，314
mapfile，177
  局部作用域，150
  指令
    CAPABILITY，184
    DEPEND_VERSIONS，186
    HDR_NOALLOC，187
    LOAD_SEGMENT，187
    NOTE_SEGMENT，187
    NULL_SEGMENT，187
    PHDR_ADD_NULL，187
    SEGMENT_ORDER，193
    STACK，194
    SYMBOL_SCOPE，195
    SYMBOL_VERSION，195
  指令语法，182
  映射指令，417
  条件输入，179
  示例，202
  符号属性
    AUXILIARY，128，128，133
    DIRECT，146，148
    DYNSORT，335，335
    ELIMINATE，49，333
    FILTER，128，133，152
    FUNCTION，129
    INTERPOSE，94，153，370
    NODIRECT，154，156
    NODYNSORT，335，335
  缺省，200
  词法约定，177

语法版本，179
mapfile（版本 1 语法）
大小符号声明，418
映射指令，417
映射结构，421
段声明，414
示例，419
结构，413
缺省，420
语法，413
mmapobj(2)，50，159，265

N
NEEDED，88，124

O
Oracle Solaris 应用程序二进制接口 见 应用程序二进制接口
Oracle Solaris ABI 见 应用程序二进制接口

P
PIC 见 与位置无关的代码
pkg:/developer/linker，255
pkg:/solaris/source/demo/system，254，257，271
profil(2)，174
pvs(1)，211，213，215，216

Q
全局符号，209，328
全局偏移表，356，371
.got，299
_GLOBAL_OFFSET_TABLE_，51
与位置无关的代码，162
动态引用，360
检查，91
重定位，316，316
SPARC，317
x64，324
x86，322
组合过程链接表，377，378

R
软件包
pkg:/developer/linker，255
pkg:/solaris/source/demo/system，254，257，271
弱符号，41，328，328
未定义，28
RPATH 见 运行路径
RTLD_DEFAULT，41，113
参见 依赖项排序
RTLD_FIRST，113，227，229
RTLD_GLOBAL，111，113
RTLD_GROUP，112
RTLD_LAZY，108
RTLD_NEXT，113
RTLD_NOLOAD，244
RTLD_NOW，94，103，108
RTLD_PARENT，112，112，112，112
RTLD_PROBE，41，113
参见 依赖项排序
RUNPATH 见 运行路径

S
生成共享目标文件，40
生成可执行文件，39
生成输出文件映像，50
输入文件处理，27
数据表示形式，273
搜索路径
运行时链接程序，32，88
$CAPABILITY 标记，227
$HWCAP 标记 见 $CAPABILITY
$ISALIST 标记，229
$ORIGIN 标记，231
$OSNAME 标记，231
$OSREL 标记，231
$PLATFORM 标记，231
链接编辑，31
SCD 见 应用程序二进制接口
SGS_SUPPORT，237
SONAME，125
SPARC 符合性定义 见 应用程序二进制接口
strings(1)，168

strip(1)，49，50
SYMBOLIC，173
System V 应用程序二进制接口 见 应用程序二进制接口

T

TEXTREL，163
调试帮助
  运行时链接，116
  链接编辑，83
TLS 见 线程局部存储
tsort(1)，28，84

W

未定义符号，39

X

线程局部存储，381
  节定义，382
  访问模型，386
  运行时存储分配，383
性能
  与位置无关的代码 见 位置相关代码
  使用自动变量，168
  动态分配缓冲区，168
  底层系统，161
  折叠多重定义，168
  改善引用的邻近性，169，173
  最大化可共享性，167
  最小化数据段，167
  重定位，169，173
虚拟寻址，348

Y

压缩，78
延迟绑定，94，107，118，244
演示
  prefcnt，254
  sotruss，254
  symbindrep，254
  whocalls，254
应用程序二进制接口，20，209
与位置无关的代码，162，361
  全局偏移表，371
与位置无关的可执行文件，18
预装入目标文件 见 LD_PRELOAD
运行路径，32，89，106，126
  安全性，104
运行时环境，19，29，123
运行时链接，18
运行时链接程序，18，87，356
  共享目标文件处理，88
  初始化和终止例程，100
  名称空间，244
  安全性，104
  延迟绑定，94，107，118
  搜索路径，32，88
  更新和新增功能，407
  版本定义验证，217
  直接绑定，169
  编程接口，105
    参见
    dladdr(3C)、dlclose(3C)、dldump(3C)、dlerror(3C)、dlinfo
  装入其他目标文件，96
  重定位处理，91
  链接映射，244
运行时链接程序支持接口 (*rtld-audit*)，237，243
  cookie，247
  la_activity()，248
  la_amd64_pltenter()，251
  la_callentry()，250
  la_callinit()，250
  la_i86_pltenter()，251
  la_objclose()，252
  la_objfilter()，249
  la_objopen()，247
  la_objseach()，249
  la_pltexit()，252
  la_preinit()，249
  la_sparcv8_pltenter()，251
  la_sparcv9_pltenter()，251
  la_symbind32()，250
  la_symbind64()，250
  la_version()，247
运行时链接程序支持接口 (*rtld-debugger*)，237，256

ps_global_sym() ，266
ps_pglobal_sym() ，267，267
ps_plog() ，266
ps_pread() ，266
ps_pwrite() ，266
rd_delete() ，259
rd_errstr() ，259
rd_event_addr() ，262
rd_event_enable() ，262
rd_event_getmsg() ，263
rd_init() ，258
rd_loadobj_iter() ，261
rd_log() ，259
rd_new() ，258
rd_objpad_enable() ，265
rd_plt_resolution() ，264
rd_reset() ，259

Z

暂定符号，36
支持接口
运行时链接程序 (*rtld-audit*)，237，243
运行时链接程序 (*rtld-debugger*)，237，256
链接编辑器 (*ld-support*)，237
直接绑定
singleton 符号，153，154
和插入，149
性能，169
转换为，143